Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures

Foundations, Cloud Platforms, and Regulatory-Compliant Systems

Sibaram Prasad Panda

DeepScience

Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems

Sibaram Prasad Panda
Decision Ready Solutions

DeepScience

Published, marketed, and distributed by:

Deep Science Publishing
USA | UK | India | Turkey
Reg. No. MH-33-0523625
www.deepscienceresearch.com
editor@deepscienceresearch.com
WhatsApp: +91 7977171947

ISBN: 978-93-7185-196-1

E-ISBN: 978-93-7185-129-9

https://doi.org/10.70593/978-93-7185-129-9

Citation: Panda, S. P. (2025). *Relational, NoSQL, and Artificial Intelligence-Integrated Database Architectures: Foundations, Cloud Platforms, and Regulatory-Compliant Systems*. Deep Science Publishing. https://doi.org/10.70593/978-93-7185-129-9

Preface

A modern entrance to the science of data. This textbook introduces the basic principles of the database system and guides students to advanced subjects such as distributed data processing, NOSQL model and intelligent query. Explanation, with practice on hands and real-world scenarios, prepares learners for both academic and professional activities in data management.

Beyond the tradition, the book examines modern architecture including emerging patterns such as NoSQL database, Amazon RDS and Google Big Query such as cloud-country platforms and distributed and multi-model systems. We also check how artificial intelligence is changing database management through automation, discrepancy detection and future maintenance.

Recognizing the increasing importance of trust and compliance, dedicated chapters focus on industries' rules such as safety, access control, data governance and GDPR and HIPAA. The study of real-world cases from areas such as retail, healthcare and finance provides valuable insight into practical implementation, challenges and migration strategies.

Whether you are a student, data engineer, software developer, or IT leader, this book serves as a complete guide to understand the developed world of database-where basic knowledge fulfils the state-of-the-art innovation.

Sibaram Prasad Panda

Table of Contents

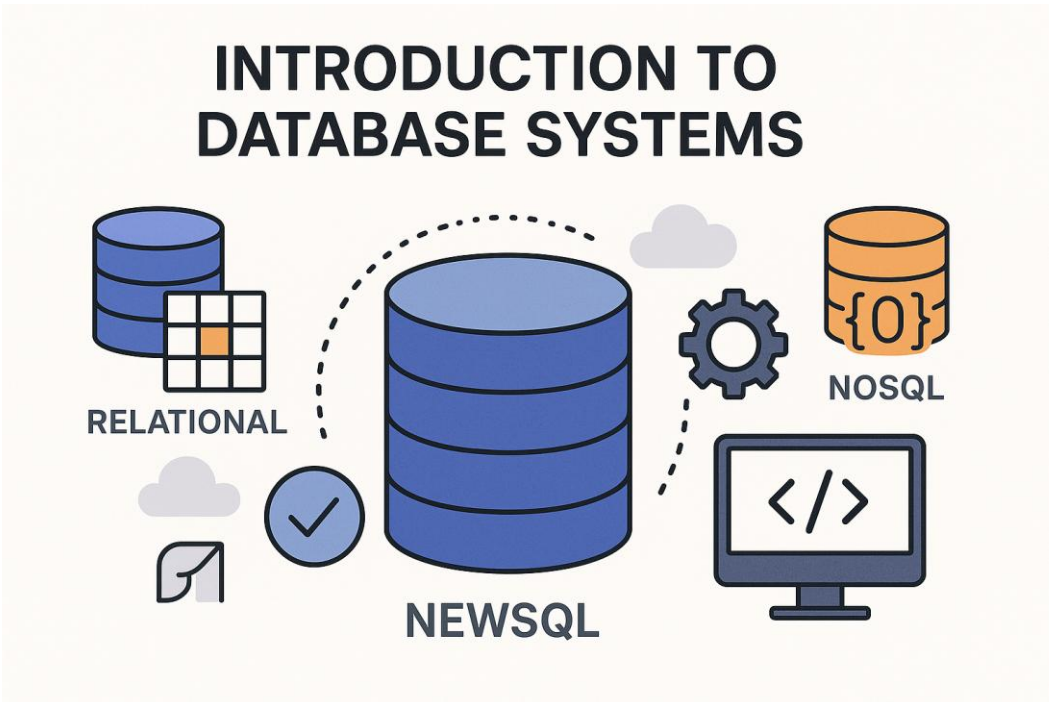
Chapter 1: Introduction to Database Systems

1. History and Evolution

Databases are arguably the most critical piece of software of modern society [1-2]. They are a fundamental component of what has been called the Second Industrial Revolution — the revolution of information. Prior to the existence of database systems, computer programs were written specifically for an application; that is, each application needed a new program to be developed. With the advent of large mainframes and the implementation of time-sharing services a rather inefficient implementation of a centralized database service came to be. That is, a large computer stored the information needed by many organizations. However, even for the simplest of applications, a lot of low-level programming had to be done for each application. Each application implemented its own routines for accessing the database and managing data formats, leaving little time for the programmers to solve the problem at hand.

The first database management application was implemented in the early 60s for a commercial application by a group led by a notable figure. This database system was called IDS and was the first system to allow multiple users to share a common data repository. It did this by implementing a centralized repository stored in the main memory of a computer used in conjunction with a series of disk drives. Commercial interest in databases burgeoned with the success of IDS, and soon other commercial database systems appeared, based on an architecture analogous to the one outlined previously.

In the early to late 1960s there were six or so commercial database systems developed. Most of those systems were based on the model of a centralized repository that was read or modified as required by user applications. The installations were few and far between, constituted proprietary systems, and thus didn't play a major role as examples.



1.1. Early Database Models

The computer systems of the 1950s and 1960s came with tape or card file systems whose main purpose was to store programs, and the data associated with running those programs. While it was obvious that these early machines could do general-purpose computing, the data storage systems were not general-purpose; the physical storage organization had to be pretty much the same as the logical organization if data were to be accessed quickly. This was not a great hindrance, since data volumes were almost always small. But as computers were applied to larger and larger problems, the central problem became data management.

The original general-purpose database systems appeared in the late 1960s. These systems could hold much larger volumes of data than any previous approach, they had elaborate software for managing files, and they provided a variety of techniques for accessing and manipulating those files. Although the term "database" came to be associated with these systems, one could argue that they were more like large file systems than true databases, since they lacked any abstraction to shield users from the record structure and physical organization of data. Rather than a collection of relations, as in the current sense of the term, a database was viewed more like a collection of unintegrated files. For this reason, and because later systems provided significant additional functionality and different structures for information storage, these systems are now referred to as general-purpose database systems rather than simply database systems. It was only much later that the term would separate itself conceptually from the idea of files in computers.

1.2. Development of Relational Databases

Over the period 1970 to 1980, the relational model and its implementation in commercial database systems developed. A system, based on the ideas presented in a 1970 paper, was demonstrated in prototypes but was never released as a commercial product. Later, the conclusions drawn from this system were used in its successor, which appeared in 1983. Concurrently, a project at Berkeley also produced a relational system, and the ideas developed during that project were commercialized in a product line. Other commercial successes later emerged, notably another system. As these successful systems became more widely adopted, the early concerns about the performance overhead of the relational model evaporated: the new systems provided great functionality and relative ease of use when compared to the other available systems, and the performance achieved was acceptable in most cases.

At this stage, dedicated customers started to acknowledge the ease of managing ordinary applications with relational products, and some even ventured to ask why common applications were still maintained with older means, and what was preventing these applications from being converted to the new environment. In fact, this phase was a migration stage, where the vendor community and large users started promoting the conversion of applications to the relational environment. The successful products encouraged the migration vendors to become part of the new business and develop tools that assisted the migration. Several companies entered the market as migration specialists, supporting the conversion of applications and data from older environments. As part of the migration process, application development was started in parallel and was soon to encompass most application areas. However, the new database development was focused on newly invented applications or applications that presented the least technical risk of development compared with other areas still maintained with older database products.

1.3. Emergence of NoSQL Databases

In the early 2000s, the sudden popularity of the Internet, together with advances in social networking and web-based applications, a boom in the generation, collection, and storage of data took place. Companies started to collect and analyse massive amounts of user-generated data. Storing such massive amounts of data in traditional relational databases became impractical. The term Not Only SQL (NoSQL) was coined to describe a new suite of database products and services. Products initially described as NoSQL included large-scale distributed storage systems without an ACID transaction model; a general focus on horizontal scalability, availability, and fault tolerance; a general focus on handling of a variety of data models, such as key-value, document, column-family, and graph; and providing support for high throughput and low latency operations on Petabyte-scale data. NoSQL databases were developed as highly available systems designed for high performance.

However, differences in distribution model, data model, data operations, and data model description dialect attracted different user communities. NoSQL initially included systems like Bigtable, DynamoDB, SimpleDB, and HBase, but later attracted users of document-centric database products like MongoDB, CouchDB, and MarkLogic. Newer data-centric frameworks like Hadoop and MapReduce, Integration-Platform-as-a-Service like MuleSoft, and big data analytics frameworks also began to offer database services originally provided by traditional RDBMS products. In addition, newer SQL extensions added NoSQL features, such as scalability and fault tolerance. As a result, the once clear delineation between SQL and NoSQL databases began to blur, giving rise to NewSQL.

1.4. Introduction of NewSQL Databases

The winners of the "Big Data" database battle were Google, Facebook, Twitter, Amazon, and other web 2.0 giants with petabyte data warehouses. These enterprises quickly adopted NoSQL. Meanwhile, Oracle, IBM, Microsoft, SAP, Sybase, and few other legacy database vendors quietly supported their DBMS products and invented new systems that fundamentally change their core architecture. These inventions became known as NewSQL. NewSQL systems were largely possible due to the significant progress in hardware which lead to an architecture using thousands of server nodes, employing different techniques such as clustered SQL Replicated Storage, Shardable Main Memory Storage, and Hybrid Storage.

Major players in the NewSQL space came from more than a decade of investment in querying-focused, column-optimized hybrid storage systems, a modern take on the enterprise data warehouse gaming the storage layer. Today those products, newly acquired by HP and Teradata, along with other products like IBM's Netezza and Microsoft's massively parallel processing SQL Server, dominate the analytics marketspace. Both those and newer startup efforts released academic papers and delivered engines that went way beyond the capabilities of their millennial ancestors for ad-hoc, large-scale query latency on vast swathes of data.

The rise of cloud computing, where organizations only paid for resources used, made it possible for even the smallest company to utilize these MPPs. But customers soon complained that these systems lacked features of the operational databases they treated as their primary data source. Resorting to data exports scheduled by cron jobs, sometimes managed by ETL software or a small army of data engineers, was simply insufficient, and often unacceptable. Furthermore, it took weeks or even months for data scientists to answer simple “What if?” questions, commanding a hefty salary during that time.

1.5. Current Trends in Database Technology

There is a great deal of current interest in database technology, particularly in large-scale data management problems associated with data warehousing, text and multimedia databases, mobile databases, geographic information systems, and autonomous database resources. Several of these problems have led to the initial and ongoing development of specialized database systems that augment more general-purpose traditional database systems. However, while there is a growing realization that advanced performance features must be embedded in database engines, these features are often used in combination with specialized programming interfaces to achieve an overall result that is more specialized than general.

One class of application is that of data warehouses, which focus on the extraction, cleaning, and consolidation of large amounts of data from multiple heterogeneous operational database sources. Data warehouses provide a basis for decision support systems and are characterized by the following aspects: (1) a large amount of data primarily in a stable form, (2) a relatively slow rate of change, (3) historical data that may date back several years, (4) a variety of different methods for querying and accessing the data, (5) large amounts of aggregated and summarized data, (6) large intermediate and final result sets that may also have to be stored, (7) data that is often shared by many users, and (8) the need to support data mining, or the discovery of interesting and useful patterns in large data sets. The key experimental question is: when do you need to build a specialized data mining engine that is more database than knowledge discovery, and when do you need to build a specialized knowledge discovery engine that is more knowledge discovery than database?

2. Types of Databases

Databases are increasingly becoming the digital foundation of various applications and websites [1,3-4]. A large and growing number of services rely on databases for storing and retrieving data and supporting very high transaction workloads. This increasing reliance has led to several new services that promise a wide variety of features, often with very high availability and very low latency and cost. New services tend to be implemented on top of one or more of the new types of data storage internally.

Over the years, we have seen several different types of database systems being built in response to evolving user requirements, workloads, and technology. These systems can be classified into the following types:

2.1. Relational Databases

Relational databases represent the pioneering architecture and the foundational basis upon which the modern discipline of database systems was built. They formalize the data model upon which most data-centric applications are built, and they were the first systems that provided a high-level, declarative method for specifying what data should be stored and what should be done with it. Although internally they consist of complex implementations built upon intricate algorithms and a myriad of techniques, they provide a simple, high-level, and intuitive interface based on tables, which most end users are exposed to via spreadsheet applications. The standard for expressing the high-level, declarative commands for interacting with relational database systems has become widely recognized.

2.2. NoSQL Databases

NoSQL databases provide alternative data storage options to the traditional relational model. There are several reasons to seek an alternative to the relational model. One reason is volume. New applications—web, mobile, and social—generate very large amounts of data, often in the terabyte and petabyte ranges. These large volumes stress relational databases. A second reason is velocity. Internet applications often demand response times measured in milliseconds; but, at the same time, the requests may be coming in at rates of thousands or millions per second. The third reason is variety. Data is often not well-structured. In fact, JSON has become a popular method for structuring data coming from the Internet. Each JSON object is essentially a mini-document that can be produced by the many services on the Internet, such as tweets.

For these reasons, architects often try to horizontal scale relational databases across clusters of commodity servers. This is a very difficult problem because of the rich diversity and sophistication of the transactional features of relational databases. The alternative is to not use a relational database. Indeed, one direction for both applications and databases is increasingly coming together. Applications are rich in diversity and complexity; they're not primarily transactional and not primarily for information retrieval of structured data. Instead, they're stateful and exploit a mix of data types and data models: unstructured, semi-structured, structured; data for human use, data for machine use; batch processing and interactive data access; analytics and transaction processing, all integrated in an application. Increasingly, NoSQL databases are being used to support this mix of application types.

2.3. NewSQL Databases

Although NoSQL solutions are important for the large volumes of data generated in the age of big data, they trade-off ACID guarantees for speed and reliability. Businesses in a variety of domains still require stringent ACID guarantees on their data, such as processing financial transactions in a relational fashion. Achieving both the freshness and latency, NewSQL databases are a class of systems that are designed to provide the scalability of NoSQL systems with the ACID guarantees that traditional relational databases provide. These systems are not a replacement for traditional solutions that manage a single node but instead are designed to work on top of a distributed architecture.

NewSQL systems use a shared architecture where several nodes work together to distribute the data for performance. They leverage the SQL syntax for transaction processing while also using advanced techniques based on physical and logical data partitioning and replication along with specialized concurrency control mechanisms to enable them to provide fast responses even though they are modelled to handle complex sufficient computations. There are several of these modern databases, which include but are not limited to various systems. In the last decade, there have also been a renewed interest in the development of traditional databases. These solutions have enhanced the distributed and cloud deployment and query capabilities of the classic databases.

2.4. In-Memory Databases

There is a third class of databases known as in-memory databases (IMDBs). Unlike traditional systems, whose data is stored on hard disks, IMDBs are designed to store their entire database in main memory. While the term in-memory database is commonly used to refer to traditional DBMSs that have been extended to use main memory for data storage, this chapter uses this term to refer to two systems: data management systems that follow NoSQL like design principles and newer systems that have been designed to take advantage of the high performance of DRAM. IMDBs are viable options for some Real-Time Enterprise requirements. These requirements are used in the enterprise and must be processed as it comes into the enterprise rather than batched and processed later. Decisions based on such processing must be made within a small time or latency. Correctness of such decisions is of paramount importance. If all of these are true, the application must provide support for updating the processed data.

IMDBs have traditionally been optimized for high throughput, low latency transaction processing, along with excellent performance for analytical queries. They are now being used to support key, niche, high performance transaction processing and analytical applications, usually in the cloud. They process a small fraction of the total data in the enterprise at any one time, but they do so very rapidly. Due to their focus on in-memory data, IMDBs do not do any of the kind of heavy lifting expected from enterprise databases: complex, long-running analytical queries, ETL pipelines, slowly changing data, or long-running data pipelines. In the current enterprise data landscape, they rely on enterprise databases to do the heavy lifting and enable them to be able to present actionable results in real-time.

2.5. Graph Databases

Current database technology has emerged from database research conducted in the 1970s. Back then, relational databases were created. With the increase of the use of internet, political and economic reasons, the landscape changed. New types of databases appeared, implementing new storing and accessing techniques. NoSQL databases were born. Increasingly, industry began using NoSQL databases, not because they were the best solution for the problem, but because that was the only solution left available. Graph databases are specialized for storing, maintaining and querying data of structure that is flat but interconnected. The origins of graph databases can be traced back to the works about network and graph data structures, the file organization and data access methods addressed to those, and specialized databases able to support those data structures and methods. The relation of users with data has been changing. We are now more interested in complex connections between data, and how that can change the state of the data we have. Searching for a data inside an enormous pool of data is more expensive than changing the state of an already existing data. Maintaining social relations data and handling complex searches or updates is easier and much more efficient using a graph database than using a relational database. Access times for queries using a graph database is faster by several orders of magnitude. Access times to either insert or delete a photo of the set are of the same order of magnitude. In the case of an RDBMS, access time to insert a new tuple into the photo album set is much longer than the query access times for the graph database or the values used for the RDBMS. Almost all commercial graph databases are NoSQL. But not all NoSQL databases are graph databases.

2.6. Object-Oriented Databases

An object-oriented database combines functions from both an object-oriented programming language and a database management system. In an object-oriented programming language, data and its associated behaviour are modelled using a single construct called an object. The associated behaviour is implemented using computer code called methods. An object-oriented programming language provides powerful and sophisticated features for manipulating objects and their associated data. These unique programming features include polymorphism, encapsulation, and inheritance. However, existing object-oriented programming languages only support an object model. They rely on file systems to store objects and their associated data in a persistent manner. Compared to a database management system, file systems provide a very basic and primitive means of object retrieval and storage.

Ideally, all objects created in an object-oriented programming language can be stored permanently using a persistent mechanism, such as a database management system. However, this design idea leads to some difficult problems. In general, database management systems help ensure the integrity and security of the objects they store. That is, it is very difficult to ensure the integrity and security of objects and their associated data if they are stored in file systems. Why? Because file systems can provide only the most primitive means—basic read, write, and update capabilities—of accessing objects and their data. There is no way to use transactions to commit updates made by a program in an object-oriented programming language to its objects.

2.7. Distributed Databases

In a centralized database, all data associated with a database is stored on a single computer and is maintained and updated by a single copy of the database system. In a distributed database environment, multiple computers serve as hosts for the database. The task of maintaining the integrity of a distributed database and of organizing and processing the functions required is handled by a distributed database management system. In this system, procedure calls are distributed using one or another of a set of communication standards to communicate across the different machines that are part of the database. The same set of standards can also handle situations in which one of the computers is not operating at a given moment.

The major function of a distributed database management system is to enable a distributed user to present commands in a form like those used with a central database. Centralized database systems use a query language, which generates commands for the computer running the database. In a distributed version of the same database management system, the user's commands generate a discussion among the computers sharing the task. One of the functions of the computation is to break the task down into subtasks that can be carried out in parallel by the different computers. For example, if a user requests a report on sales from each of a group of stores, the aggregated result can be generated by having each of the computers in the group work on the problem simultaneously. For each computer, the required computation is trivial, consisting of a query to the local database to generate a report on the sales from the local store.

2.8. Cloud Databases

Cloud computing is bringing a change in the way we design, develop, manage, and use database systems. Companies are moving from owning and managing large data centres to purchasing database services from third-party cloud providers. A large vendor owns large data centres and rents storage and database services to thousands of users. Cloud database systems tend to have a more relaxed approach to features like transactions to provide high availability and elasticity.

The cloud-based model has well-defined advantages: simple management, elasticity, high availability, and very low cost. These systems are often designed for very large applications that have millions of users. The cloud model works extremely well for web-based applications that must store and query user data and logs. Several such cloud applications are experiencing hundreds of millions of active users. These applications have basic data needs, as they require the types of queries that are usually answered using traditional relational systems. Since these applications experience huge amounts of traffic, they cannot afford downtime. As a result, cloud data systems provide high levels of availability and performance.

We have a situation where companies are moving quickly to the cloud. The cloud was initially seen as a place to host larger scale web business applications serving hundreds of millions of users. These applications are built on languages and tools that can scale for large loads but are not particularly concerned with large-scale transaction processing. They are also not concerned with rich feature sets. As web applications scale, their data management architecture moves data to specialized systems in the cloud. What is the future of cloud databases?

3. Importance in Modern Applications

In the age of information explosion, there is an increasing need on the part of data users as well as solution providers to manage and exploit the vast quantities of data for decision making. This increasing reliance of organizations on the business intelligence derived from data has led to its positioning as an enterprise asset, and to the need for its linking to business processes using state-of-the-art technology. On the technology front, a new class of solutions is emerging, driven by advances in processing, storage, and network technologies, which are making it easier and more economical to capture and analyse data. This transition is paving the way for radically different solutions compared to those in place before. But database systems are likely to continue playing an important part in addressing the data management needs of many organizations, and hence, in the larger picture of linking enterprise processes with the data infrastructure.

The links between enterprise processes and the data infrastructure are not new, and many classes of data-intensive enterprise applications have existed for decades. Applications built around a database core in areas including airlines, banking, insurance, HR, customer relationship, enterprise resource, and supply chain management have provided business efficiencies and process automation benefits, allowing organizations to differentiate themselves from other players. The databases that have powered these applications are known to hold large quantities of critical business data, and special care has been taken to manage these systems. Today, enterprises around the world are trying to re-create those efficiencies and benefits around the latest data- and technology-related trends such as the web, business analytics, business intelligence, globalization, open source, and outsourcing. Database systems provide important capabilities that enable and enhance these enterprise applications.

3.1. Data Management and Storage

Databases are used to manage data in applications and in storing the data used, produced, and shared by the applications. For example, in e-commerce applications, databases are used to manage product catalogues and in storing customer and order information; and in payroll applications, databases are used to manage employee data and in storing pay stub and tax withholding information. Applications interact with databases by sending requests over a connection, using a database API. The requests include commands to create new data or to read, update, and delete the data already stored, known as CRUD commands. The connection is typically to a server-side software package called a database manager or DBMS, which organizes data in a format such that these commands may be efficiently processed. The commands can be either non-transactional commands that are processed in isolation or multi-step transactional commands that must process successively as a group and are therefore subject to strict consistency, integrity, and isolation rules. In either case, the commands are typically issued in a query language that specifies the request, the desired operations to apply to the data, and often the data itself; the request is then processed by the DBMS, which responds as appropriate. Most databases store persistent data in a structured, tabular format on one or more disk drives. While disk drives inherently have very high capacity and provide relatively low-cost, persistent storage, they are also inherently very slow and so providing high-performance data storage requires techniques like caching frequently accessed data in volatile memory and using pre-optimized disk layouts and disk access patterns. Data structures and functions to organize and operate on the data that is stored must therefore be carefully chosen and optimized, by both application developers and DBMS developers, to satisfy the performance requirements of the application.

3.2. Scalability and Performance

When we think about the applications that we rely on today, we likely pay little heed to the systems that support them. At any given time, websites and mobile applications are often hosting, transmitting, and serving some staggering data. Consider for example, that at this moment, there are dozens of millions of users sharing links to images, videos, and other dynamic content. Meanwhile, thousands of millions of users are posting status updates every minute, and a major service is providing high-quality traffic estimation for routes and selling massive amounts of ad space for keywords. Given this scale, any hiccup on these systems would likely affect millions of people – which is hardly acceptable today.

While hiccups are unacceptable for many applications today, the problem is still far worse. A well-known incident is the launch of a major online service, which led to the temporary unavailability of the online stock brokerage. One major area of concern for large Internet companies is availability and performance. Performance refers to the expected wait time for an operation on the database to be executed, or the rate at which a specific operation can succeed over time. Availability refers to the expected length of time for which a system is available for service. A system can be available but performing poorly. For example, online transaction processing systems for small banks tend to provide good availability, but a transaction may take a long time to finish due to low performance. Performance Saturation refers to the point at which a system can no longer maintain its expected performance. Consider the protocol used for serving web pages. After a certain number of concurrent connections, the server responds to each request in a variable time interval. Beyond this point, incoming connections are queued, creating a backlog, which may be a large amount of time.

3.3. Data Security and Integrity

Databases are used in all environments to keep track of all transactions that take place there. This relates to Banking Systems ensuring transactions are logged properly as well as Tracking Systems used by companies dealing with Logistics companies. The security and integrity model within a Database System are essential to ensure that the data is safe from manipulation and the accounts of all people are made to take the correct amount of money and that the tracking environment of packages is correct. Also, unless you're handling an online application, the typical business model of software developers would be to have an application database system running at the workplace with the software of the employer. These databases are usually used to track daily expenses and working schedules of people and ensure that meetings are at decided times. If a Bank Database is hacked, the funds of a lot of people can go to any random place and would go unnoticed till someone starts facing a malfunction when checking their utility expenses like checking the amount on their electricity bill to see if they are paid. Similarly, if the tracking database of a logistics company is hacked, people can easily obtain other people’s packages and see when they will be at a certain area of the country. If these logs can be modified, let's say company X would want to wait till Z comes out, but his packaging is scheduled to arrive at a different area.

3.4. Support for Big Data and Analytics

The recent increase in volume, velocity, and variety of data that is being collected has given rise to what is more commonly referred to as Big Data [2,4-5]. Specifically, more and more devices are being deployed which are easily able to collect and transmit data at an unprecedented scale and speed. The range and type of these devices are extremely diverse ranging from temperature sensors, cameras, and phones to medical devices, smart watches, social media, and mobile apps. This explosion of sensors and devices has triggered large scale data collection and dissemination by both organizations and individuals. In addition to traditional tabular data that is stored in relational databases, huge amounts of data are being generated in different formats. The volume of textual data, social network activity, video, and other unstructured or semi-structured data being generated by users, businesses, and sensors is unmatched. The sheer scale of this data is overwhelming. The use of data and the value derived from this data is increasing rapidly in solving business problems, scientific exploration, and understanding user preferences and behaviour.

This chapter discusses the database technologies that are being used to manage and analyse this new class of large-scale unstructured/sensor data. Traditional database systems are not well-suited to address the challenges posed by the fast-paced and scale of this new type of data. Instead, a new class of architecture and systems have emerged in recent years. Applications and cloud computing technologies have enabled many of the big data companies and services. These services allow companies to seamlessly collect, store, and analyse this massive volume of current and historical data without worrying about day-to-day scalability issues. These technologies play a key role in marketing, advertising, fraud detection, image and video analysis, scientific research, and many other areas.

3.5. Role in Web and Mobile Applications

Relational database systems with their tabular structures, SQL languages, and high-level access mechanisms are very popular for web development. The reasons are simple. To begin with, most web applications involve the storage and processing of various forms of data, from those that implement logic such as user profiles and authentication tokens, to others that implement business and sales processes like user-generated reviews about a product, or your transaction history with an online provider. Furthermore, as with various types of enterprise applications developed in the past, being able to store, manage, and manipulate these large collections of user data and transaction logs in some centralized repository is key to the service. So is quickly making it available for access through some application programming interface that allows users, third-party application developers, and other services to query this data, as well as update it with new products, transactions, and reviews. RDBMSs also supports many of the basic building blocks of web and mobile apps out of the box in a type-safe way. They make it easy to ensure that all reviews have some associated ratings, that all transactions have an associated payment, and that product descriptions comply with internal standards.

Web and mobile applications run on distributed cloud platforms. This removes several of the constraints of traditional enterprise applications. It is now common to deploy many copies of the same application, easily accommodating tens or hundreds of thousands of users. Software is provided by hosted cloud platforms, often as part of a service. These services often provide the basic components of data-driven applications: storage, authentication, sharing, monetization, and replication. However, mobile and web applications have completely different property requirements than the RDBMS systems originally used. These requirements include speed, flexibility, ease of use, resource constraints, and the ability to handle structured, semi-structured, and unstructured data.

3.6. Integration with Other Technologies

Rather than being isolated systems, nearly all modern database systems are integrated with a variety of other important technologies, some of which are important partners with database systems – such as data warehouses, analytic processing, data lakes, cloud, and big data systems. Others provide a function the users of database systems require for building applications and using the data held in the database. These include ETL tools and data integration/federation tools that package the databases for the various functions that are carried out in user applications. The addition of the new database type adds important functions and capabilities for the data management function in the enterprise. These systems typically support one or more types of storage and processing that have become important for the new requirements for data management systems caused by big data. We may see dependencies between the various vendors because of these capabilities. A specialized database vendor may license features from a more general vendor to be able to handle the new types of user requirements across the much broader range of big data-type applications. But the vendors are not the only suppliers of new technology. Large-scale data warehouses now run on clusters. Column-store databases also do some of the same functions well and some of the same functions poorly. They too are another choice that enterprise data management technology users can evaluate and adopt depending on their requirements.

3.7. Outlook and Innovations

Innovations in both hardware and software for database systems are central to the innovations in enterprise and web-scale applications and devices. These innovations and their synergistic effects fuel future innovations in technologies and application domains. We summarize the future of database systems along the following threads in the context of cloud-scale data and AI-driven applications.

Massive Deployment of Specialized Accelerators: Artificial Intelligence, Machine Learning, and Deep Learning applications and workloads dominate the demand for active compute. Specialized accelerators for AI such as TPUs and

GPUs are deployed at scale, along with domain-specific accelerators. Computational capabilities such as performance, energy efficiency, and cost-profile are not unique to the AI domain alone, but extend to other mainstream workloads, including databases. It is no longer economically feasible for hyper scalers to submit baselines of these workloads to cloud service providers. Databases cannot enjoy an endless honeymoon over the absence of special-purpose hardware. Hyper scalers are also actively utilizing more General-Purpose Graphics Processing Units acceleration for unstructured data such as video and image transcriptions at both compute- and data-intensive levels – bottlenecks for latency requirements.

Disaggregation and Elasticity: Computation and storage systems are provisioned according to requirements. Provisioning is flexible and can scale to requirement with little to no time delay. However, there is memory latency, bandwidth, and size requirement, which must be set aside and met within limits. Naturally, database vendors have done efficient workarounds for resolving these constraints, and their improved profiles for establishing bottlenecks will require innovation in the intersection of data and AI. Using close systems cohesive around tight APIs, Rapid Innovation via Combination, and the low friction of building infrastructure for accessing resources, continue to coalesce of infrastructures with complementary strengths and weaknesses.

4. Conclusion

This essay briefly introduced Database Systems. A data model gives a way of compiling records into a database; a database programming language gives a way to create and administer databases, as well as define and manipulate database records; and DBMS provides a set of services that executes the definitions and operations specified in the programming language. We reviewed several of the most popular data models and the most important DBMS services. A data model gives a concept of what the basic building blocks of a database are, what structure is imposed on the data contained in the database, and what form the relationships between the different records take. A data model comprises: a set of potentially infinite records, each describing an arbitrary number of fields; each field belonging to one and only one of the record's fields; a relationship specification that defines relationships between records. Examples of data models are the key-value, the document, the table, and the object data models. Examples of record definition languages are XML, HTML, and document DBMS query languages.

We then analysed several of the most important DBMS services, namely DDL, DML, database access management, data formatting services, concurrency control, auditing, backup & recovery, and two-level security services. Those services were reviewed in the context of several popular DBMS operational models. A database programming language defines and executes data operations in a database. At the highest level, there are two types of data operations: manipulation operations and schema definition operations. Manipulation operations execute data processing; schema definition operations execute administration. There are two types of programming languages for databases: data definition languages and data manipulation languages.

References:

[1] Paton, Norman W., and Oscar Diaz. "Active database systems." *ACM Computing Surveys (CSUR)* 31.1 (1999): 63-103.

[2] Liu, Ling, and M. Tamer Özsu, eds. *Encyclopedia of database systems*. Vol. 6. New York, NY, USA: Springer, 2009.

[3] DeWitt, David, and Jim Gray. "Parallel database systems: The future of high-performance database systems." *Communications of the ACM* 35.6 (1992): 85-98.

[4] Ullman, Jeffrey D. *Principles of database systems*. Galgotia publications, 1983.

[5] Elmasri, Ramez. *Fundamentals of database systems*. Pearson Education India, 2008.

Chapter 2: Relational Database Management Systems (RDBMS)

1. Introduction to RDBMS

With the rapid technological growth in IT, many new platforms are coming into the market. These platforms are primarily made for the storage and manipulation of databases. A large volume of data is growing every second due to the use of these platforms. This large amount of data can be manipulated easily but the main task is to maintain its integrity and security [1-3]. To maintain the integrity and security of this large volume of data, we must require an efficient one and that efficient is known as RDBMS. RDBMS is the backbone of data storage management.

A relational database management system (RDBMS) focuses on the relational model. An RDBMS manages data as a collection of tables, in which each row has a unique identifier, and each column has a fixed data format. The relation model includes several advantages over the hierarchical and network model.

First, relations are conceptually simple and intuitive. Relations have a general structure, consisting of tuples and attributes but the data in the tuples do not need to fit a structure or format. As a result, there are relations in which some attributes do not have values or have values of different data types or structures.

Second, the independence of the logical and physical data structure means that changes can be made to the way the data is stored without needing to change the way the data is related to other data. For example, an application programmer can modify how a certain relation is stored without needing to make any modifications to the other relations or to the applications that use those relations.

2. Database Schema

2.1. Definition and Importance

In the context of Relational Database Management Systems (RDBMS), a database schema defines the logical structure of an entire database. This representation of a database is achieved logically using a representation defined in the original design of the database. A schema defines how data is stored in a database and its relations. It defines its dimensions, data types, tables, and their relationships and validation rules. The schema is implemented using a collection of definitions contained within the database's metadata. The schema dictates the logical structure of the data and how it is stored, processed, and accessed.

The schema represents how the database will be perceived by the users. The logical structure of the database may change on different occasions, the schema may be altered or changed, or a new schema may be defined to be created or be applied to new data. The current schema may control the data definitions. Maintaining schema is crucial to ensure that implementations of various parts of a database system can share data, and at the same time implement different and evolving access and processing algorithms for that data. Schemas serve to protect the data from accidental or unauthorized changes.

2.2. Types of Schemas

In RDBMS, two types of schemas are used: Logical Schema and Physical Schema. A logical schema defines all the objects in the database in a way that users understand; it draws connections between objects and makes use of things like authentication keys or table indexing. The logical schema acts as the logical structure of an entire database and defines how the data is organized and how the relations among them are associated. It includes the entire database's entities, the relationship between those entities, and constraints on the data. It is a complete representation of the data, implementation-independent and doesn't include any physical data.

A physical schema provides a low-level description of the database, which describes the way data is stored in the database. It deals with data characteristics such as how data is encoded to preserve its accuracy, how indexes and partitions may be implemented, what the inferred data types will be, and where data will be physically located in the storage space. The physical schema provides information about how the schema is physically arranged in the hardware. The physical schema involves the description of a database that describes the various mappings of relationships and the actual storage of data so that it can be accessed faster.

3. Tables in RDBMS

Relational databases organize data into logical structures called tables. A table consists of attributes and data entries. A tuple is a collection of attributes that together define a logical record in the database. For example, a tuple in a table called STAFF contains the attributes StaffID, StaffName, and StaffRole. The attributes together define a staff record with the identifier, name, and role of the staff setting.

Tuples in a table must be unique; otherwise, the data could become corrupted. Uniqueness is ensured by using a PRIMARY KEY attribute. A primary key has the following requirements: Each value of a primary key must be unique, and the primary key cannot consist of a column with an empty value. Consider, for example, the STAFF table. StaffID can be made the primary key attribute because each staff can be assigned a distinct ID, thus ensuring that the StaffID attribute value is unique. Alternatively, assuming a small staff translated the same names and roles, StaffName can also be considered a primary key. However, given the constraint of StaffName being unique, it is preferable to use StaffID.

Besides a primary key, other feature constraints exist at both the table and attribute level. These constraints, called ENTITY INTEGRITY and REFERENTIAL INTEGRITY, respectively, ensure the fidelity of the structure and relationships among tables as well as uniqueness of rows in a table. Because data is stored in tables within a database subsystem, it is helpful to define how tables are structured and what relationship they have among themselves. Subsequently, we investigate these two key elements of a database: the structure of tables and relationships among the tables.

3.1. Structure of Tables

A table in an RDBMS may be as simple as a record of a list of students, their addresses, and their registration dates OR it can be a complex structure recording bank transaction, including loan numbers, account numbers, interest rates, and transaction types. In either scenario, a table design or structure defines the data to be stored. This design consists of elements that include the table's name, its columns, their data types, and any constraints on the data that is to be entered. The rows of a table display the actual data that is stored in the table. Each row is a record containing the data associated with each item, person, or event. In the example table that follows, each row corresponds to a specific registration transaction for a specific student. The columns provide what is known as the data dictionary of the table, specifying the name and data type of each field stored in the record. Based on the title of the column or the description that is usually placed at the top of the column, it is easy to see what data is contained in the column. In the example table that follows, the first column, labelled "student ID," contains a unique numeric value assigned to each student. By scanning vertically, we can see that those numbers belong to students whose IDs range from 1,001 to 1,011. This simple data representation and presentation is the structure and design of a table in an RDBMS. The set of design rules and constructs that is used to create a table is described as its schema. The words design and schema are used interchangeably in RDBMS documentation and data design discussions. Since a table is composed of design components such as its name, columns, data types, constraints, and the relationship of the data in the table to the data in other tables, any change to a part of those components can affect the schema. In general, the following attributes describe the key components of any database table schema.

3.2. Data Types and Attributes

Any value that can be stored in a column of a table must belong to some data type. The data type of a column determines how the values in that column can be stored and interpreted. Different RDBMSs provide support for different data types. The following are some of the more common categories of data types found in current commercial RDBMSs. Most current RDBMSs support structured data types that associate with a column a set of attributes which are visible and can be used for query evaluations. Effectively, the operators used for querying attributes of the structured attributes are functions that are bundled with the type definition for the structured type. The type of definition can include a list of input functions and a list of output functions. These input and output functions can take various forms since user-defined types are created within a programming language environment. Manipulations using structured data types are likely to be straightforward since all structures are strongly typed. User-defined data types are typically based on built-in data types. The following forms of user-defined data types are provided in many RDBMSs. User-defined scalar data types are built on user-defined functions that provide aims to convert a value of the user-defined type to a built-in type, and from the built-in type to the user-defined type. A user-defined scalar type can be used to store integer values but will use a different representation. Column constraints can be used to restrict the values in a column to a predefined set. Constraints can also be defined at the table level.

4. Relationships in RDBMS

Less structured, more contained, abstract representations of real-life things are called entities, and the different characteristics or attributes that make an entity unique are called fields, or attributes [2,4]. For example, an employee entity could have fields like employee number, employee name, job title, department, and so on. Whereas a department entity could have fields like department number, department name, and department location. The linkages or relationships between the entities like employee and department need to be established to make the database model complete. Relational databases are based on the entity-relationship model, which allows defining a database in terms of simple entity sets and specific relationships among these entities.

4.1. Types of Relationships

A relationship is an association between multiple entities. It describes a relationship set within ER modelling. RDBMS employs three types of relationships to describe data associations: one-to-one, one-to-many, and many-to-many relationships. Relationships explain how two or more entities are related to each other in an RDBMS.

One-to-One Relationship (1:1): The simplest type of relationship is a One-to-One relationship. In such a relationship, an entity is related to, or only associated with, one occurrence of the other entity. For example, a person has a Social Security Number. Each citizen has a unique Social Security Number.

In a One-to-One relationship, both the "one" entity and the "one" of the "other" are described in two separate tables. To maintain referential integrity between both tables, a Primary Key is created on either entity where the "one" entity is further described and placed in the other relationship table, known as the Foreign Key. This establishment of relationship assigns a singular relationship between the two tables querying for the One-to-One relationship. A common example of the 1:1 relationship is found in educational institutes, where each teacher teaches one class and one student will study in a specific class.

One-to-Many Relationship (1:N): The One-to-Many relationship is a more complex relationship than the One-to-One. In this relationship, an entity is associated with multiple instances of the other entity. A One-to-Many relationship is the most common type of relationship found on a relational database. For example, each student has multiple examination scores. Each score belongs to only one student.

4.2. Foreign Keys and Referential Integrity

In a relation, an attribute or a set of attributes may have meaning outside that relation. This is the case with references to the entities corresponding to other relations, such as the customer references in a set of sales records. The principal use of relations is to provide a source of values usable in making assertions about the entities corresponding to the tuples of other relations. We need some way of defining that an attribute of one relation provides such a source for the values used in making assertions about the entities identified by the tuples of another relation. This is done by declaring that the attribute is a foreign key for the other relation. An attribute is a foreign key for another relation if references are made to that other relation from this one. Any attribute of a relation that is a component of a primary key must be the foreign key for references made to this relation from the other relation. The foreign key of an attribute relationship is a foreign key for the relation.

Being a foreign key is an extra property that we sometimes want to associate with a relation attribute. Such properties are typically not of interest for relations. When we deal with the representational model, we make a distinction at a higher level. We assert that there is an attribute relationship between the two relations, but we do not indicate that attribute of one of the relations is a foreign key for the other one. In a relational database, an attribute relationship between any two relations is represented by the attribute of one of the relations being a foreign key for the other.

5. SQL Basics

5.1. Introduction to SQL

No matter how RDBMS is implemented, there must be a standard language that enables the user to perform operations and control on such data stored in these systems. Hence, databases have a standard interface at the front-end, although back-end issues may be quite different. Structured Query Language, or SQL, has been the traditional database standard language, although implementations differ from manufacturer to manufacturer. SQL is a standardized language that is used for query processing in relational databases. Although it is prone to vendor lock-in issues, the significance of SQL as a universal language for database access cannot be understated. Because SQL is intended to be a standard language, it is quite flexible, with many extensions depending on the specific vendor or the underlying data model. SQL was originally developed at IBM in 1974, although different companies, databases, and vendors have put their own flavour into how it is implemented or extended. As a SQL user, either querying or manipulating the data and/or structure, you will need to keep a lookout on how a particular vendor has implemented SQL because there are numerous differences, not only in the features offered by different databases, but also variations in the syntax and their parameters options. In general, users should be able to issue commands that are near similar for its basic SQL functionality when using a different vendor or database. Hence, you may have heard phrases like vendor lock-in problems for traditional database vendors with their proprietary implementations. However, it is the additional features unique to databases that differentiate between the vendors and databases.

5.2. SQL Syntax and Structure

Structured Query Language (SQL) is the standard programming language for managing data stored in a Relational Database Management System (RDBMS). SQL is used for storing, manipulating, and retrieving data as well as creating and modifying schemas, tables, and objects. SQL is a declarative language that consists of a number of statements. Each SQL statement consists of keywords, identifiers, and clauses. Keywords are reserved words. Identifiers are the names specified by the user (like table names, column names, usernames, etc). The meaning of the same keyword can change based on its position in the statement. For example, the word SELECT is interpreted as a keyword in a SELECT statement but can also be used as an identifier if the table has a column named SELECT. Clauses give additional information to the statement. For example, in the SQL statement "SELECT * FROM employee WHERE age > 40", the keyword SELECT tells the SQL engine that data is to be retrieved, the identifier "employee" denotes the name of the table, and the clause WHERE specifies what data is to be retrieved.

Any valid SQL statement is case-insensitive. However, using capital letters for keywords and lowercase letters for identifiers is a widely followed syntax convention because it enhances the readability of the SQL statement. SQL statements are required to have a specific order. Keywords and clauses that precede a group of keywords/clauses must be specified before specifying these keywords/clauses. Statements are usually executed in order from top to bottom. Clauses that require sorting and filtering of data set must be executed before keywords/clauses that work on the filtered data set. To improve performance, SQL statements may not always be executed in the order of the defined syntax.

6. Data Definition Language (DDL)

The Data Definition Language commands allow you to create, alter, and drop tables within a database. Normally the Data Definition Language for supported database systems is implemented using a specific set of proprietary commands. These commands are generally similar but can differ depending upon the database system in question. For example, in one system your commands may look slightly different than those in another system, in that they have different keywords, data types, etc., but the concepts behind DDL are still the same. It is very similar to a programming language in the sense that it consists of words or keywords, operands, and punctuation in a predefined order.

Most of the time, tables are created upon the initial database design and rarely are ever changed. On occasion, an existing table must be altered to adapt to changes in business rules. Times of heavy application and usage of the system utilize the tables more frequently, so adding new columns, replacing existing column attributes, as well as other modifications, should be attempted during maintenance windows when the database is less utilized.

DDL commands typically require certain permissions before being allowed to complete. This is because changes made through DDL are generally permanent and irreversible. Dropping a table and then re-creating that same table with the CREATE statement is not the same as inserting a new row of data through a Data Manipulation Language command. The INSERT command can be repeated numerous times which affects only that one row, but the DDL commands change the table structure during the whole process, not just affecting the one row. These database objects cannot be repaired with a simple DML command so the required permissions are in place to ensure that these statements are given the needed levels of scrutiny.

6.1. Creating Tables

Data Definition Language (DDL) is a subset of SQL commands that are used specifically for defining, modifying, and controlling access to the database schema. DDL commands define the database structure or schema; they contain commands such as CREATE, ALTER, DROP, etc. The keyword DDL stands for Data Definition Language.

Relational Database Management Systems (RDBMS) make use of tables to represent the data as well as its relationships. Data is stored in rows and columns in the respective tables. The columns of the table have a specific datatype defined based on the type of data they will hold. A column can have constraints defined on it which specify the conditions or restrictions for the data that can be stored in the respective column. When creating a table, it is necessary to define not only the columns, datatypes, and constraints but also the table comments, and then grant privileges on the table. A table with column definitions looks like the following.

Creating a Table Comments A Table Comment is an optional statement that can be defined while creating a table, and is also a way of providing a description about the table. The syntax to comment a table is as follows. After the first row of the table is inserted, queries can be run that would retrieve the description of the table using the table comments. Database developers can use table comments to document the purpose of the table. Developers can make use of the comment to see what type of information is conveyed by the table in a separate table. It could be a simple statement such as "This table stores user details". Writing a comment is not mandatory; however, it is highly recommended for logical clarity and easy understanding.

6.2. Altering Tables

Due to how data is structured inside a relational database, it is common that you will need to make changes to your data definition after creating a new table. This might entail, for example, adding or dropping new columns or even altering the definition of a particular column like, for example, changing its datatypes or whether or not it is nullable. You will find that most of the popular RDBMS allow you to make these changes easily, yet some restrictions on the types of modifications you can make may vary widely from one product to another.

Modifying a table usually employs the DDL command ALTER TABLE or a variant of it. The syntax and capabilities of ALTER TABLE offered by different RDBMS may vary, though. In many cases, you can add a new column or drop an existing column from an existing table definition with the following two commands, respectively:

ALTER TABLE table_name ADD new_column_def; ALTER TABLE table_name DROP column_name;

where the new_column_def represents a legal column definition for the new column, including the column datatype and optional constraints. Keep in mind that you usually will not be able to add a new column to a table that already contains data unless you set a default value for this column, in which case a value will be assigned to existing rows for this new column.

If removing a column, you may not need to worry about existing data, as this operation will usually drop the data for that column altogether. Some products may not allow you to drop a column if that column is part of the primary key or if any indexes or constraints rely on that column. Others might also restrict dropping columns from tables with dependencies like referential constraints. If this is the case, you may need to first drop all dependencies that reference the column you want to drop or even the entire table.

6.3. Dropping Tables

A RDBMS typically creates tables for holding the entities represented in its skeletal schema. But over time, as an organization’s needs and requirements evolve, there may be old entities that no longer need to be represented and thus old tables that are no longer needed. The DROP TABLE statement can be used to delete these tables from the database system. The syntax of this statement is:

DROP TABLE table-name Where the basic requirement is simply the name of the table to drop. This operation deletes the named table and all its data. It also deletes any entries in the system for the dropped table, so the system will no longer recognize the table. Attempting to issue queries against the dropped table will result in an error response.

Note that dropping a table deletes the table and all its data. When the DROP TABLE statement is executed, the RDBMS may automatically delete entries for the dropped table from any other tables, as well as from any secondary data storage devices.

A feature of primary key and foreign key constraining is that any table that is related to another table by a foreign key must be dropped before the primary key table can be dropped. Otherwise, reference integrity rules would be violated, and the database would be left in an inconsistent state. Hence, in this case, the DROP TABLE statement will fail to be executed. Further, dropping a non-empty table will delete all its data.

7. Data Manipulation Language (DML)

Data Manipulation Language (DML) is a segment of SQL that allows for the data stored in the database to be manipulated. The DML operations include inserting new data into a database, updating existing data, deleting existing data, and querying existing data. While the first three operations change the state of the database, the last operation is used to retrieve data from one or more tables. DML operations can be categorized into two types based on their effect on the database.

The four basis operations of DML are Inserting, Updating, Deleting, and Retrieving Data.

7.1. Inserting Data

Data Manipulation Language (DML) is a computer programming language that enables users to perform operations such as inserting, updating, deleting, and making queries on data stored in RDBMS (Relational Database Management System) using RDBMS structures. These specialized system languages are usually proprietary, and the syntax is not the same across systems. On the other hand, many DML Databases tend to be like the SQL capabilities. SQL is the de facto language for data manipulation with RDBMS and is widely implemented in RDBMS systems.

When creating a table in a database, the table is empty until data is inserted into it. The SQL command that is used for inserting data into a table is unified and is in fact called INSERT. INSERT means to put in and is the logical opposite of DELETE. Data can also be inserted into a table using a single INSERT command; however, this command can be long and tedious if data is inserted in bulk. Inserting bulk data can also be done quickly and efficiently using different techniques in interactive mode using a few commands. One of the most useful aspects of a database is that they allow users to create and process multiple records which are collectively called a relation. Each record in a relation represents a unique entity from its domain.

INSERT command is usually the first command executed for a new table, after the CREATE command. First, individual or small numbers of records and/or tuples are inserted into a table. After the database reaches some preliminary state, either more records are inserted using the INSERT command, or all records are inserted in bulk. For example, an airline or travel agency database would be empty at first. It would gradually be filled with some airline and flight records, starting with a handful of airlines and all their flights that are initially in operation. Then the database could be filled with all flight records for the coming years.

7.2. Updating Data

Data might need to be modified during its existence owing to certain upgradations. For example, salary of an employee may need to be increased, or a particular employee may need to be assigned to a different project, or a project may need to be assigned to a different client or certain changes may need to be made to the specification of a particular project. At any point of time, some values of the attributes of a tuple in a relation might need to be changed while others might remain unchanged, this action is termed as updating or modifying the relation.

The SQL command used for this operation is called UPDATE command. The UPDATE statement in SQL is a data manipulation statement that is used for modifying the data of a database. Using an UPDATE statement, we can modify field values of one or more records in a database table. This modifier can be a single value, and we can modify the value of a single column, or the modifier can be an expression that can modify values of one or more columns. If we want to modify values of a single column in multiple records, we can provide a condition that must be satisfied by the records to be modified. If we want to modify multiple columns in a record, we can specify the condition, and the condition must be satisfied by a single record. We can also update a record without specifying any conditions. However, if we do this, the value of that field of all records of the table will be changed. For example, if we update the statement will change the salary field to replace whatever value are present for that field in all records of the employee table by the value.

7.3. Deleting Data

The SQL language provides a command to delete rows from a table. The data in a table may become stale over time. Data deletions are commonly performed to keep set data collections current. In some cases, data deletions occur to keep each table representation reflecting current reality.

The SQL syntax for removing rows from a table is:

```
DELETE FROM tableName WHERE expression;
```

For example, to remove a row from the student table for the student John Smith, we could use the command:

```
DELETE FROM student WHERE sName = 'John Smith';
```

Notice that the WHERE clause does not include any means of determining the value of sID. Therefore, the restriction on the WHERE clause is not sufficient to ensure that a single row is removed from the student table. Should a single row have been the only row matching where clause, the command successfully removes that row. If two or more rows have been matched by the where clause, that erroneous command would still remove all the rows matching sName = 'John Smith'.

The consequence of taking this error path while issuing a DELETE statement can be disastrous, especially if we are not model controlled, and either some of the rows in the student table do not get deleted, or all rows in the student table get deleted. Having the appropriate cascade deletions occur correctly as well as not having the app delete other unintended rows becomes a very complicated task. As such, the DELETE command should be used judiciously.

7.4. Retrieving Data

Any application that uses a relational database will eventually need data, and the Retrieval of Data is how that occurs. For a general-purpose RDBMS, statement construction will rarely be limited to any one area of the abstraction hierarchy. Most systems are accessed at different points by different levels of the hierarchy for special purposes. Very few systems can extract data without some assistance, nor can external processes perform a full range of queries. Local processes generally use external specification to communicate with the higher level of application abstraction. For example, an external Data Extraction process might allow the user to build a request that could be sent to a more or less permanent local Extract process, which would retrieve the data into a file.

RDBMSs use a formalism known as Relational Calculus to specify those queries and other data retrieval and manipulation exercises. Calculus is a powerful tool that allows a level of specification that is more intense than the traditional argument formalisms, and therefore more concise. However, most application programmers use the more traditional argument formalisms specified by the DBMS. This is particularly true for efficient and common queries that are usually built by application programs. C provides the model of interaction with the RDBMS provided by argument lists, while Image provides the more general interface based on control commands. Many SQL-based RDBMSs provide a compiler that generates an executable module from SQL statement bundles. The statement bundles may have to be in a particular form, and the compiled modules tend to be small. But the module is infinitely logical, so that SQL is an important formalism. SQL is usually invoked through some argument method.

8. Data Control Language (DCL)

Data Control Language (DCL) commands grant and take away special permissions whereby certain users can perform various operations on a relational database and its various objects, such as tables, indexes, views, and stored procedures. The two most used DCL commands are GRANT and REVOKE. These commands give and take away user and role privileges to select, insert, update, delete, execute, alter, or create database objects. These commands are typically issued by a database administrator or an intermediary.

8.1. Granting Permissions

DCL is responsible for data access permissions and security levels. Each company will have different requirements for the security hierarchy connected with an RDBMS. DCL is implemented via the keywords GRANT and REVOKE in the SQL language. GRANT gives users access privileges to a database. A user is an entity that accesses the database. Each user will have its own individual set of access rights. The administrator must keep close track of the access rights for each user, ensuring that rights are not granted to people who should not have them and that rights are not removed from users when their access is still needed. Revoking certain permissions can conflict with other privileges. The actions specified in a REVOKE command will fail if the user trying to carry out the operations does not have sufficient access privileges.

SQL allows the manipulation of data in the database, as well as control access to the database data through commands in the Data Control Language (DCL). Some of the commands that we can find in DCL are the GRANT and REVOKE commands, which are related to the access and permission management of a database. All the permissions defined for the respective users or user types are set using the GRANT command. This command is used to define permissions for an operation to be executed by a user. In addition, it has some options for defining the type of specific permission on a certain object. In the REVOKE command, permissions can be revoked for a certain operation that was previously permitted. The need for the authorizations in the databases is to ensure confidentiality, integrity, and data protection. When we define and manage authorizations in a database, we are protecting accesses from unauthorized users, allowing only those users that we wanted to manipulate the data or the database objects. However, authorizations are also prepared to balance the access that users have over the data. For example, some users may need to see only data from a certain department, while other privileged users must be allowed to view all the data in the database. So, in this case, we must be guided by the principle of the least privilege. This means that authorizations should be created and defined in order to provide a user access that is sufficient to perform their tasks only within the limits of necessary. With DBA authorizations, that is the only user that has complete access to modify the entire database access limitation in this access that only some users can work with privileged data.

8.2. Revoking Permissions

REVOKE takes back permissions granted with the GRANT command. A REVOKE statement removes the access rights that were given to a user (or group of users) by the GRANT command. The suffix of the command specifies which permission is going to be taken away. Revoking permission for certain actions may conflict with other privileges that have been granted to the user. The actions specified in a REVOKE command will fail if the user trying to carry them out does not have sufficient access rights.

The command for revoking or taking back the permission given on some database object is called REVOKE command. Syntax of REVOKE command is as follows:

REVOKE privilege_type ON object_name FROM user_name;

Here, • Privilege_type – specifies the privileges granted on the object to the user. User can be a single or comma separated, or all users may be specified here. • Object_name – specifies the name of an object supported by that database. • User_name – specifies a specific user, a comma separated user-list, or all users.

Note that the ALL option specifies all the users to revoke the specified privilege from. The specified user does not need to have been granted that privilege in order to execute this statement. If the user executes an UPDATE statement without specifying any condition, or if the condition specified will never be true, this statement could deny any privilege for any user. The ALL and ALL EXCEPT options cannot be used in the same statement.

Now that we have gone through granting privileges on the database object, we will create an understanding of why and when we need to revoke privileges. In some cases, usually for security reasons, we may need to revoke previously granted privileges. For example, we may decide that the accountant for our company should no longer be allowed to select the salary information from the Employee table. If this information is included in the Employee table, we will need to make SELECT and UPDATE and possibly DELETE privileges on it to revoke. There is nothing that prevents us or even disallows us to revoke a privilege that we previously granted for a user.

9. Transaction Control Language (TCL)

Transaction Control Language are the commands of SQL that manage the changes made by DML commands. TCL commands are used to manage the changes made by DML commands. But the problem lies with the fact that once you have executed the DML commands, the changes appear immediately in the DB. This can create data inconsistency problems if changes resulting from a DML command are not committed and made sure to be permanent. For example, a bank wants to transfer money from User1's account to User2's account. To transfer funds, the money is first deducted from User1's account and then added to User2's account. Now suppose that after deducting the money from User1's account, suddenly the system crashes and the record is not updated. So User1's money has been deducted but User2 has not yet received any amount.

So the system should always ensure that either the fund transfer is complete, and the money is deducted from User1's account and added to User2's account or that none of the tasks has been performed. This property of a transaction is called Atomicity. A transaction must be completed in its entirety. If the transaction is interrupted for any reason, the database must be restored to its previous state and all tasks done during the transaction be undone to be performed again. TCL commands do the task of committing and undoing the transactions. A transaction control language consists of commands like COMMIT, ROLLBACK, and SAVEPOINT that control the changes performed by DML. The commands are provided in pairs. The changes are saved with a COMMIT and are undone with ROLLBACK.

9.1. Understanding Transactions

From a Database Management System (DBMS) point of view, a transaction is one logical unit of work that accesses and possibly updates various data items. A transaction may be as small as a single SQL command that updates a database or a much larger unit consisting of numerous commands that perform a more complex task. Typically, a transaction comprises a sequence of operations, all of which must be carried out if the transaction is to be considered complete. Transactions will be consistent if each of the transactions operates on a snapshot of the database taken at a specific instance of time. One program unit may consist of numerous statements, such as the following example, which one to put in a transaction: The above statements include inserting some records into all the tables. The transaction will be approved if a record has been inserted into all the units. If a record is not inserted in a unit, several databases must be rolled back to the last version transferred before the failure of actions. There are numerous points to consider with respect to transactions. A transaction is valid and is said to have legitimately executed if it obeys the ACID properties. The ACID properties state that a transaction is atomic, consistent, isolated and durable. The transaction must be either done or not done. Only one transaction can execute at a specific instant. When the transaction is completed, the change is still in the database whatsoever. The transaction is consistent when it performs the last actions.

9.2. Commit and Rollback

Introduction Database Management Systems are essentially concerned with the storage and retrieval of information. With this, Database Management Systems offer features with which we can control other aspects of that information and how it is changed by the programs which use it. Transaction control is one such set of features, and the commands associated with this mechanism are called Transaction Control Language (or TCL commands). The transaction control language commands are mainly concerned with the commit and rollback of the transactions. Commit A commit statement is executed when the formulators or users have finished making all the changes to the data and want to make sure that all changes successfully made by them are permanently recorded in the database. A commit statement contains no parameters and can refer to any transaction that is currently in a committed state. After you commit a transaction, you can't undo it. If you change your mind and decide that you want to reverse those changes, you must take a step to reverse the action; therefore, commit is the last step in a transaction. Once the action was said to be committed, there's no going back. A commit not only allows the user to be sure that the effects of a transaction will not be lost, but it also releases locks which might be held on affected entities. Rollback A rollback statement is executed when a formulators or user of the database has made a mistake and wants to restore the database to the state that was reached just before the transaction began. In other words, we are saying that this transaction is not being successful. A rollback statement takes no parameters and can refer to only one transaction at any specific time. Rollbacks are based upon a logging mechanism, which keeps track of changes made so that those changes can be undone in the event of a failure. Roll Back command restores the database into the preceding state. When a Roll Back command is issued, the actions of a transaction are reversed in the reverse order. A rollback restores the affected entities to the state they were in prior to the transaction being executed.

10. Constraints in RDBMS

In computer science, the term constraint refers to a restriction on the values that an attribute can take. The relational data model provides a formal foundation for a class of constraints that ensure the consistency and validity of a relation instance. Such rules or constraints can be applied to individual or to multiple records and can be checked at any point in time during the life cycle of a relationship. Usually though, they are checked every time that a new or modified record is included in the relationship. Relationship constraints correspond to the postulates of the underlying entity semantics. They include existence rules governing possible values for both simple attributes and relationship types.

Most of these validity rules are called integrity constraints. In relational database management systems, these limits are bound to the relations defined in the data dictionary as special attributes. These integrity constraints are a set of conditions and restrictions that ensure the quality and accuracy of data during runtime. This subsystem rejects the modification and insertion of any data that do not fulfil the integrity rules. Constraints offer a restricted form of data validation at the database level and enable databases to enforce some of the basic concepts of the relational data model, in particular, entity integrity and referential integrity. When a table is internally created, by default, there are no constraints on its Input/Output operations, allowing disparate data to be inserted, which can lead, during use, to various run-time errors.

10.1. Types of Constraints

One of the key requirements for database design is creating a database that accurately reflects the entities and relationships being modelled by the information. But we should also try to ensure that the information in our database is accurate, complete, and useful. This requires implementing certain rules and restrictions on the values that are stored in the database.

Database constraints may be applied to tables, attributes, or relations to enforce certain restrictions and rules with respect to stored values. These restrictions are required to ensure accuracy and consistency of stored values and to eliminate or reduce invalid values. These unwanted values usually arise because of incorrect entry or update operations. If the DBMS were to allow any values to be entered, then we could not be sure that the information retrieved from the database would be meaningful or correct. Constraints preserve the integrity of the information in the database.

For the purposes of discussion, we categorize database constraints into two main groups. The first category consists of integrity constraints, which limit the allowed set of values in an attribute or relation. Integrity constraints disallow certain values from being stored both at the attribute and the relation level. The second category consists of security constraints, which restrict who may perform operations at the attribute or relation levels. We look at each of these categories in more detail.

Within the integrity constraint category, there are three major subcategories. Domain constraints are the most basic type of integrity constraint. They are specified in terms of an attribute's domain, which is built into the schema description. Domain constraints restrict an attribute's value choices to a smaller subset of the possible domain values. Domain constraints are in fact specified by placing constraints on the data types of attributes.

Redundancy constraints specify that a value in one attribute must be equal to a value in another attribute (but not necessarily the other way around). Redundancy is a term we will use in discussing data redundancy. Null value constraints indicate that a value in a specified attribute cannot be null.

10.2. Implementing Constraints

When we specify constraints on the schema of a relation, the one rational consequence of this specification is that whenever a tuple is presented to the RDBMS, some check is made to ascertain whether the constraints are satisfied. If they are not, the relation is not modified in accordance with the insert, delete, or update instructions; otherwise, the indicated change is made. This approach imposes an additional burden on the system, although supporters argue that the constraints are enforced by the RDBMS as a service to the user, and that the user is the one who determines whether the load is beneficial. In fact, defining schema-level automatic integrity checks is primarily a user requirement and an RDBMS constraint is just a statement of that requirement. Second, the primary advantage of such checks is that they can be performed each time a modification is attempted. The temporary tuple set that may violate a particular constraint is created whenever changes are made, where it may not be possible to perform a check without changing some part or parts of them.

There are generally three different levels of support for integrity constraints for an RDBMS. At the highest-level support of the predefined set of constraints which is available in all RDBMS; they are handled completely automatically by the system kernel and users cannot affect any part of their implementation. The kernel checks that each integrity constraint is preserved after every modification and refuse to carry out the operation if it is violated either during the operation or at completion.

11. Normalization in RDBMS

Normalization is a data design technique used by designers to reduce redundancy and eliminate undesirable factors like insertion, update, and deletion anomalies. The general strategy is to divide larger tables into smaller tables and define relationships among them. However, the designer needs to exercise caution when using normalization to guard against excessive performance cost. After normalization, the implementation of a normalized database may include some denormalized tables for reasons of performance. The exact performance characteristics will depend on design efficiency, database size, application design, and the anticipated workload.

The normalization process involves a series of transformations applied to the database to produce a predictable set of designs that are efficient and stable. However, the implementation may not be fully normalized. The nonlinear process by which a designer makes the design trade-offs necessary to produce the final database structure is known as denormalization. During the process of normalization, redundant data structures are identified, and the database is divided into relatively small, simple structures called relations that conform to several conditions known as normal forms. Each of these transformations is guided by a set of normal forms. A relation that does not satisfy a normal form condition is not in that normal form and is said to have the associated redundancy.

Despite the advantages offered by normalization, too much normalization can also adversely affect database performance. Consequently, many real-world database implementations contain denormalized structures that trade off some redundancy for improved performance. Data that has not been normalized is said to be denormalized. With respect to databases, denormalization is the opposite process of normalization, where the data is deliberately intentionally duplicated and combined into a single structure. Such desirably reduces the number of foreign key restrictions, enhances the efficiency of relation joins, and improves read speeds for operations, while adversely affecting update speeds.

11.1. Purpose of Normalization

Normalization is a systematized way of ensuring that database tables are properly constructed. The purpose of normalization is to make data in the database as simple and unobtrusive as possible. It does this by reducing redundancy and dependence by organizing fields and table relationships. Irrelevant duplicate data can create data anomalies that may degrade system performance, cause unnecessary updates, affect data integrity, and slow file systems. Prior to normalization, the data structure is often tested and analyzed to uncover any possible dependencies present within the table layout. These dependencies can have an impact on the outcome of the normalization process. A normal form, or data structure, is a structure designed to eliminate all structure dependencies.

There are two types of dependencies: functional dependency and a multi-valued dependency. Function dependency refers to a relationship between two tables in a one-to-many relationship with the primary keys dependent on one another. Multi-valued dependency is used to store many-to-many relationships. A database structure will be considered normalized if it meets a minimum set of requirements, including all tables being in at least boyce-codd normal form, the third normal form, the second normal form, and the first normal form.

11.2. Normal Forms

Database Normalization is the process of organizing a database in such a way that it reduces redundancy and dependency. Logical Data Structures in Database

Normalization is classified into various Normal Forms based on the order. A database will only be considered normalized if it is in the First Normal Form (1NF), Second Normal Form (2NF) and Third Normal Form (3NF) and Boyce-Codd Normal Form (BCNF) or 4NF or 5NF or higher. There are 5 Normal Forms but in practice, we only use 1NF, 2NF, 3NF, and BCNF.

1NF: First Normal Form 1. Basic Definitions 2. Table has unique rows 3. No column can have multiple values 4. No duplicate columns in a table 5. All entries in a column must be of the same kind

2NF: Second Normal Form 1. Basic Definitions 2. Must be in 1NF 3. Every non-prime attribute of the table is fully functionally dependent on the whole of every candidate key of the table

3NF: Third Normal Form 1. Basic Definitions 2. Must be in 2NF 3. No transitive dependencies exist

BCNF: Boyce-Codd Normal Form 1. Basic Definitions 2. Must be in 3NF 3. For every FD X->Y, X must be a super key of the table

4NF: Fourth Normal Form 1. Basic Definitions 2. Must be in BCNF 3. Multi-Valued Dependencies (MVDs)

5NF: Fifth Normal Form 1. Basic Definitions 2. Must be in 4NF 3. Lossless Join is associated with every join.

11.3. Denormalization

Denormalization is a database design technique used on a previously normalized database, which is the process of attempting to optimize the read performance of a database by adding redundant data or grouping data. Denormalization is often necessary for systems with high read performance and/or high data access complexity while serving queries with many and complex joins. Denormalization is also often performed in data warehouses for speed.

Denormalization, however, is not without downsides; it increases the complexity of the database and requires that any database changes be made in more than one place if redundancy is introduced.

Denormalization is a part of the design of:

• A star schema, which by nature has denormalized dimensions (though a dimension could in theory be normalized). • Data marts. • A table in a data warehouse oriented towards a speed performance, typically dimensional.

A star schema is a type of data warehouse schema that is a subset of dimensional modelling. Star schemas can be a good option when designing a cloud-based data warehouse, allowing you to quickly and easily deliver reports to your organization. A star schema consists of a centralized fact table surrounded by one or more-dimension tables, like a star. Denormalization occurs when data from the dimension tables is redundantly stored in the fact table; however, databases can handle joins between dimension tables and the centralized fact table, allowing them to store significantly less redundant data than fully denormalized star schemas.

12. Performance Considerations

The performance of an RDBMS is a core issue, affecting the degree to which the users can be served. This section discusses some of the important points in this regard. Speed of access to records in the database affects an RDBMS's performance. The most used speedup technique is indexing. The response time for executing a set of operations on a set of relations also affects RDBMS performance significantly. An RDBMS typically has a single query optimizer, which generates a single query-execution plan to evaluate any query posed to the RDBMS. The quality of the generated plan affects the performance.

An index is a data structure that provides a speeding mechanism for retrieving rows using a specific column value or a group of columns. Consider a relation that has no index created on it. If you need to retrieve rows based on where clause of the following form: where A = some A value, the query-execution engine needs to read every disk block that contains the tuple, possibly examining every tuple on the block. Given the popularity of B+ trees in commercial RDBMSs, this data structure will now be described. A binary search tree is used to represent an ordered set of values and pointers, where the key of each node is larger than the keys of its left child and smaller than the keys of its right child. A B+ tree can be viewed as a variant of a binary search tree, where each node has multiple keys, generally of order high double digits or low triple digits. The increase in the order of the tree allows each node to be stored in a single disk block, so that, unlike a binary search tree, almost all nodes of the tree can be kept in main memory.

12.1. Indexing

Indexing is one of the most important performance-related features of an RDBMS. Almost every RDBMS provides support for indexing, because without it, the task of efficient processing of queries is very tedious. The creation of an index for a column of a database table enables faster search and retrieval of rows from the table based on the values of that column.

Let us explain by example, how indexing can improve the performance of some DML operations. Say you have a database table that records the credit history of people, the credit history being specified by the SSN of the individual, the time for which the credit history is being specified and the various items for which credit is provided. Performing queries on the SSN column could be extremely slow. Because for a given SSN there can be hundreds of thousands of records in a country as vast as the USA. The entries in the SSN column may not be unique for your table. And performing queries with other criteria for columns, whenever the SSN column is not specified will be too slow for a database of this kind, however fast with specified SSN. So it is better to place an index on the SSN.

It is also possible to have inverse indexes in RDBMS. Here, each entry in the index file points to the values of the column that is being indexed, rather than pointing to the addresses of the rows in the main database. In an inverse index, for every unique value in the indexed column, an entry is placed in the index, with addresses of rows from the main database which have the corresponding value in the indexed column, as the values associated with the corresponding index entry. These addresses could be of the form of a sequence of pointers or just a list that keeps the addresses of the corresponding rows.

12.2. Query Optimization

Once data has been organized and indexed, and is stored on disk, executed queries can retrieve results quickly. The final step in minimizing query processing time is to transform user-created queries into single SQL statements, preferably those that use the least number of resources needed to satisfy the query. Database designers and developers can play a large role in effective query design by encouraging the same style of queries. For example, in a general research database concerning authors and their publications, it would make sense for authors to use the same index field to store the signature fields of their metadata records. All signatures begin with the author’s last name and first initial, since that uniquely identifies a publication for most authors. If the same index field is used for all publications and all authors have associated metadata records in the database, there will be an easy way to search for all documents by the same author, cascading document-level replication.

However, the flexibility of relational databases is such that any metadata schema allows for diverse searching strategies. Neither as restrictive nor as deterministic as their predecessors, RDBMSs do not inherently provide the ability for precompiled multi-table queries or recursive queries. It is up to the backend query processor to analyse the user query and provide a signature field routing or fully specified compound statement at runtime, then return record ids. This is the most syntactically complex part of this processing that a relational database engine does. It is also the time costing part of using a relational model integrated with a combiner. After plan generation, the root query could just as well be elementary query then collected and intersected, any structure. It is also the time costing part of using a relational model integrated with a commutative operator such as a reverse index combiner.

13. Security in RDBMS

This chapter deals with security aspects of RDBMS. Security covers two aspects in any database management system — user security and data security. User security deals with authentication of users in the system, so that each user has restricted access to only the part of database he is allowed to use. The second aspect is data security, which involves how secure is the data from unauthorized users. Security provides many user-related features such as user creation, deletion, modification, assigning storage space, assigning security, etc. The SQL commands related to user security are stored procedures, which help execute the command and create the user.

Most RDBMSs allow the user to keep important and private data as encrypted data, which cannot be accessed by non-allowed users. The process of encryption and decryption is slow, so this feature is used only for selected data. Methods such as Data Encryption Standard allow a few seconds for encryption and decryption. However, explosion of available computer power has made using encryption on large amounts of data slow without the use of special hardware. Such databases contain a substantial portion of the world's sensitive data: personal bank accounts, medical histories, credit cards, etc.

For this reason, RDBMSs must provide the option of encryption, so that private user data can be secured. Such databases have almost no flexibility: adding or removing copies, changing key-settings combine to make this process quite time-consuming. Management requires ongoing monitoring by the administrator of which users should receive copies on a part-time basis, and under what conditions logs of accesses and changes also are crucial.

13.1. User Authentication

Most RDBMS require user authentication before granting the user access to the database system. The authentication may be as simple as entering a username and password, or even as complex as injection of an intelligent card that generates dynamic passwords. The first scheme is one of the simplest forms of security and user data may be stored without encryption. Often, the only demand is that the password be of a certain minimum length using a combination of upper- and lower-case letters, digits, and special characters. In this case, it can be cracked easily using static dictionary attacks.

To make such user authentication more secure, passwords can be stored in an encrypted format such that the actual password cannot be reconstructed even if the file containing the encrypted passwords is accessed. Usually, user passwords are hashed with a sufficiently strong hash function combined with a salt to resist dictionary attacks that use pre-computed rainbow tables. Further, successive login attempts after a certain fixed number of unsuccessful attempts should result in a certain time delay before further attempts. Some companies might also have a security policy of forcing users to change their passwords after a certain period. This is especially important when each user's access is not limited, i.e., a user has access to an entire database or multiple databases whose contents are not restricted to a specific area. Modern RDBMS also support two-factor authentication using OTP generator apps.

13.2. Data Encryption

While username-password pairs can validate if a client is who it claims to be, they do little to prevent another user on the same internal network from capturing and manipulating that user's request and response. Encrypted connections, which encrypt and decrypt the exchanges by using techniques that are relatively easy for authorized parties but virtually impossible for a third party to unlock. Encryption is the best way to make sure that the communication exchange between the client and RDBMS server are not being touched.

There are several data encryption techniques. The most common one is the asymmetric encryption, using a pair of public and private keys. Public keys are stored in a third-party certificate server, called Certificate Authorities. Certificates contain identity information of each party and their digital signatures of the CAs. To encrypt information, an entity uses the public key from the other party. Only the other entity, the one who has the private key, can decrypt the information through a function that "inverses" the encryption function. Other encryption techniques are symmetric techniques. In these techniques both parties share the same session key, and both encrypt and decrypt messages using that session key. These are often protocol wrappers. However, because sharing the session keys can signal a vulnerability, other entities should not send any sensitive information until some other data has been exchanged using asymmetric encryption.

14. Backup and Recovery

Backup and recovery are critical components of any database management system. Database systems are a repository of large volumes of changing data and thus require stringent measures to ensure that the information is never lost and remains consistent. Database servers are thus required to provide specialized functions for backing up and restoring a database. The methods provided by the RDBMS can vary from the primitives provided for copies and logs to complete different database copies and disaster recovery scripts. Local and remote copies of databases, logs, and snapshots are the methods used under various circumstances built into the RDBMS. The automated activity of copying the data and/or structure is called backup. Most database servers provide backup options that would create backup copies of an entire database or of just a portion of the database. Many database backups are incremental, where only portions of the changes made since the last complete backup are written out. These are often faster and permit you to conserve space. There are two types of database restoration methods: Restore and Point-in-time recovery. Restore is simply bringing the backup copy back in use again. Point-in-time Recovery is bringing the database to the state that it was in at any moment before the crash with no lost transactions.

14.1. Backup Strategies

In computer systems, "backup" means to make or keep a copy of something, and "recovery" means to restore from it. A computer backup is a copy of important files. A database backup is, therefore, a copy of a database data file. It protects the data or structure from user actions, which include program bugs, database bugs, and even hardware bugs, and from catastrophic events like power failure, fire, or criminal intent. A database backup is a vital component of a complete disaster recovery strategy.

Occasional backup of important data files is done because it is the least expensive way to safeguard against critical data loss. By periodic, we mean hourly, daily, weekly, and monthly backups, using tape drives for storage. Archiving, the transfer of data from active tablespaces to backup storage, so that the active tablespace remains within bounds, is frequently done by schedulers via scripts.

Consequently, full and incremental (and differential) backups are the most common full database backup methods. Methods related to incremental backups include stamped, online, archived log, and incremental backup method. The differential backup is the second least common full database backup method, and the file-group method the third least common. Log shipping is a method used in disaster recovery database. The hot backup method is the least common full database backup method. Finally, networked tape backup is as the name implies.

Full and incremental backups are the most stout-hearted full database backup methods. Full backups are backed up from all data files at once. Incremental backups create backups from active data and the put log files to incremental storage. Thus, file copies are divided and backed up according to which partial subcomponents have changed. Incremental backups are made after every event that changes the database.

14.2. Restoration Methods

Restoration of a database to repair the effects of logical corruption is a more complicated exercise. It may involve making irreversible changes to the database, which would lead to the loss of some actual data modifications and not allow rollback of some transactions in progress at the time of the database corruption. In the simple case of a logic error, a simple restore from backup may suffice. This is the case when the error is detected after existing transactions have been committed but before new transactions have started. Other scenarios are less joyful.

One method is to perform a point-in-time restore to somewhere just before the corrupting error occurred and then apply redo log records to fix it up. The danger with this technique pertains to the choice of point-in-time. If the operation that caused the error was a simple modification, rather than a new transaction being committed or an old transaction aborted, we have no way of knowing the instant at which we should stop applying redo logs. Errors caused by transactions that are aborted can often be fixed up by just applying the undo logs from a short distance earlier, but that may entail losing a lot of committed updates that were done to the database.

Another alternative technique would be to do a restore to some earlier instant at which we have a backup and then use the redo logs to “catch up” to the present time, just as the database management system does after a restart from after a crash scenario. This method has the advantage of being easy to administer. If the error detected isn’t detected for a long time, however, the redo logs may be large and unwieldy to apply. The second technique fails in some circumstances; In-doubt transactions that were committed after the point of database corruption are not easily handled unless those transactions only did inserts or the appropriate triggering actions have been placed on the inserts.

15. Future Trends in RDBMS

Clearly, databases have come a long way since their inception back in the early '70s. They have grown to adapt to all requirements and applications, coming in all shapes and sizes. I would like to outline a few upcoming trends in how databases will be used, and which services will change in the next few years.

Cloud Databases I think the most important future change is the upcoming shift of databases to the cloud. Furthermore, this shift will probably take out a lot of storage hardware from applications. The last years have shown increased acceptance of hosted services for data storage. E-mail is probably the most prominent service to make that shift to the cloud. Services have not only become popular for private users but are also starting to penetrate the enterprise sector. Hosted services for company e-mail are becoming more and more interesting.

The storage of e-mail messages is one thing. Nearly everyone has a few thousand e-mail messages, while other data may be around a few terabytes in size. Nevertheless, what is now seen as a shift from an internal to an external storage copy will soon move on to application data storage. Hosted solutions for document creation are now being used by companies around the world. The acceptance of such services is increasing. Hence, the residential architecture in which all data of all users around the world is copied to hosted solutions is close at hand.

NoSQL vs RDBMS: Another noticeable trend in the past few years has been the rise of NoSQL databases. NoSQL databases fill a gap that has existed in the database landscape for some time. Applications have emerged that require features not offered by the RDBMS approach. In other words, the relational model has gotten into trouble with applications that cannot be handled correctly in this model.

Cloud Databases: The cloud is a revolution in the way we implement computing services. The various services provided on the cloud simplify database hosting, maintenance, high availability, scalability, and security. You get all the RDBMS traits established in the previous sections as a service. A DBaaS allows fast deployment of application, allowing the developer to concentrate on building the application and letting an expert cloud provider to take care of maintenance. The work of hosting the database is shifted to an outside cloud-hosting provider.

In a cloud-hosted environment, security must be considered. The stored data is sensitive and/or critical for the operation of the customer using the DBaaS. Moreover, the data stored usually belong to many different customers, making their security particularly difficult. We will summarize the RDBMS traits below. In a traditional approach, a customer would either share the hardware resources with customers with similar pricing or pay a lot more and have dedicated resources. In either situation, the provider has the option of encrypting sensitive data to protect it from wholesale access.

An RDBMS must provide multi-tenancy to lower the costs. Therefore, companies providing DBaaS need to integrate a way to prevent access from employee of native host provider. A good implementation strategy is to create resources who inherently prevent data access typically implemented by allowing each tenant to have its data/metadata in dedicated partitions or folders.

NoSQL vs RDBMS: An increasingly popular and perhaps more appropriate option for big data applications is NoSQL databases. NoSQL is a term that describes a broad category of database management systems that are different from traditional RDBMS engines in some way. Some use key-value pairs instead of a tabular schema, others do not require a fixed schema at all, others use a data structure called a document, and many implement a distributed database architecture by default. Nearly all the NoSQL products are open-source projects built by passionate communities. The space is still evolving, and many questions related to schema design, user communities, and system features remain to be answered.

RDBMS systems have been with us since the early 1970s, and they have matured into well-understood and useful tools for many common applications. As an industry segment, the RDBMS has a wealth of knowledge, many standards, and guidelines for best practices, which most of the NoSQL systems lack. These guidelines include methods for modelling data, indexing strategies for improving data read times, query capabilities, and rules for database normalization that analyse database queries and identify redundant fields with the goal of improving updates and deletes.

It should also be noted that while NoSQL databases aim for speed and scalability by optimizing specifically for write performance by design, adding an ACID guarantee for consistency can be very difficult, or may degrade performance. As a result, many NoSQL databases offer eventual consistency, leading to temporary periods of inconsistency between servers in a distributed system. These periods of inconsistency happen when the database is written to more frequently than it can synchronize consistency across distributed servers. In contrast, RDBMS don't allow periods of inconsistency, which guarantees the user is never given invalid data. RDBMS do this with serialized write locks that guarantee mutual exclusion while writing to the database, which require low latency disk accesses.

16. Case Studies

To better understand and appreciate the various capabilities and characteristics of RDBMS, it is useful to look at real-world application scenarios or case studies for RDBMS. Through these case studies, organizations seeking to adopt databases can better formulate their decision on the right technology that meets their application needs at the time. This chapter first lists some example applications of RDBMS and then provides a comparative analysis of selected RDBMS products. This section lists several example applications where RDBMS are utilized by organizations. Some of these organizations either provide the RDBMS or use RDBMS in their own IT architecture as a back-end database server for their applications. For example, one organization provides its own RDBMS and uses it in-house for running its own business applications. Another organization has its own RDBMS, which it utilizes for powering its web-based ecommerce transactions. On the other hand, an online payment network for customers to perform transactions and bank operations runs with another RDBMS. An organization that runs a well-known online encyclopaedia uses a different RDBMS as its solution. A government agency runs a specific RDBMS to support tax filing and reporting tasks. An international organization utilizes another RDBMS to power its internal services. Finally, most enterprise applications utilize one of the major RDBMS products as the back-end server to support either front-end applications or cloud-based services.

16.1. Real-world Applications of RDBMS

Companies around the world store their operational, financial, marketing and customer information in various database systems. These data, if properly maintained, lead to improvement in business. Some of the applications of databases are as notification systems, record maintainers, processing systems and decision support systems. The decision to implement a database is based upon several aspects like costs of database implementation, increase in productivity, customer relations, need for better information etc. Some of the major applications of databases in use by various data providers are listed below.

The use of databases in life science applications has been increasing for several years, in various sectors such as drug discovery, drug development, clinical trial processing, patient care etc. Recent years have seen a major growth in the amount of data in Life Sciences, in both structured and unstructured forms. For example, there is a large amount of unstructured data in the form of medical literature, patents and clinical trials information as well as structured data in the form of biological and chemical databases. RDBMS systems not only maintain the highly sensitive and important data for Banks, such as customer account information, deposit, withdrawals, loans etc, but also process large volumes of transactions that occur every day and ensure full data integrity. RDBMS systems are used by Banks to ensure that internal policies are met during transactions and ensure the safety and security of customer data. Potential RDBMS applications for E-Commerce services include marketing, sales, revenue, conversion rate, customer analytics, marketing campaigns, order history, spends, and interest’s analytics.

16.2. Comparative Analysis of RDBMS Solutions

This survey presents a comparative analysis of commercially available RDBMS. A great deal of research has been invested in the development of highly scalable distributed systems and large-scale data management. Moreover, there are many commercially available products providing most of the required features. Therefore, we decided to list commercially available products and state their specifications according to six categories: Architecture, joining framework, schemas, optimization, size and structures, and main usage.

The clustering category specifies either shared-nothing or shared-disk architecture. If the product is a parallel system, it is denoted by a parenthesis containing "shared-nothing" or "shared-disk". If the product is a centralized RDBMS, it is denoted by "Centralized". A single system cannot be both Parallel and Centralized, but we put them both in the same column because of their common RDBMS features.

Structure schemes is the only attribute that can be different for different schemas in the same database (if support is provided). It specifies the schemas that support the organization of dimension tables differently than the organization of fact tables, or only support fact tables. Snowflake schema can also be considered a star schema, which manages a complicated hierarchy in a dimension table. The second attribute used is the "materialized views". It specifies whether there is support for maintaining materialized views. Optimization is currently a major issue in the database community. Many different optimization strategies have been suggested. We classify the optimizers into two groups: cost and heuristic based. In the "Cost-based" column, we list the optimization techniques that include exploration of all possible query plans.

17. Conclusion

In conclusion, Relational Database Management Systems are an essential element of information technology and are widely used in almost all organizations that work with data. They allow an intuitive representation of real-world concepts and provide an efficient mechanism for its storage and operation. There are many vendors that produce RDBMS products. The system most used in organizations is an RDBMS, which pioneered many of the models considered standard today. Other RDBMSs are widely utilized by organizations of all sizes and produces RDBMS, which is mainly used in the telecommunications industry. An open-source RDBMS is perhaps the most well-known, used primarily for web applications and is noted for its extremely fast response time. Another open-source RDBMS is also noted for its robustness and good support for OOP features.

RDBMS technology is thriving. With the introduction of object-relational systems, RDBMS can overcome their original limitations and are expected to remain the model of choice for many years to come. Substantial investment by vendors and constant improvements mean that their operation is becoming more interactive. They provide tools for monitoring their operation and tuning configurations for increased performance and throughput. Their design is flexible, allowing them to cater to a variety of user needs while ensuring that the data remains secure. Data is always time-ordered, helping users get a better insight into normal operation. Functions for triggers and rules allow for a variety of predefined data operations to be executed automatically. Additionally, relational database Systems support parallel operation across many nodes ensuring high availability and redundancy. Thus, the growth in RDBMS technology, while subjected to continuing competition from the object and document databases, is expected to continue throughout this century.

References:

[1] Date, Christopher John. *An introduction to database systems*. Pearson Education India, 2006.

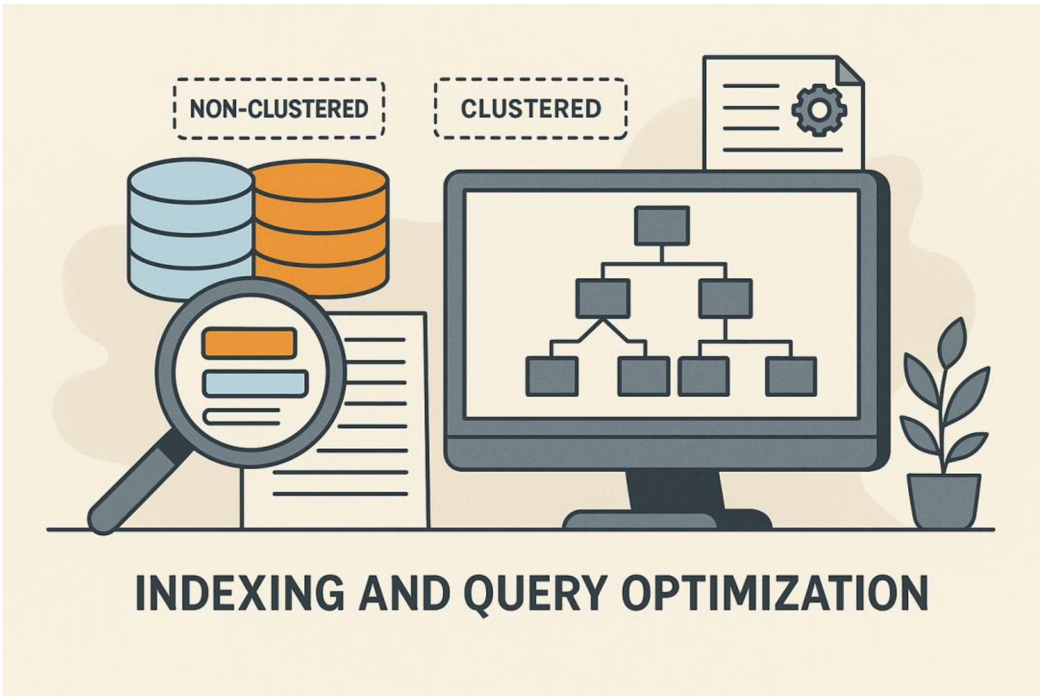
[2] Özsu, M. Tamer, and Patrick Valduriez. *Principles of distributed database systems*. Vol. 2. Englewood Cliffs: Prentice Hall, 1999.
[3] Bonnet, Philippe, Johannes Gehrke, and Praveen Seshadri. "Towards sensor database systems." *International Conference on mobile Data management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001.
[4] Silberschatz, Abraham, Henry F. Korth, and Shashank Sudarshan. "Database system concepts." (2011).

Chapter 3: Indexing and Query Optimization

1. Introduction to Indexing

Historically, the primary representation of a database was in the form of a collection of flat files, consisting of records within files. Each record in a file stored the values for the same collection of attributes, while different records stored the values for different objects. Such a representation was inherently inefficient because significant amounts of time had to be spent on reading the files in their entirety or in reading blocks from the files from which records were to be selected. Primary and secondary indices were developed as structures to speed up the selection and access to individual records. The use of primary indices for forcing a linear order on records within a file to reduce access times was clearly limited in its implementation. However, the use of secondary indices to access records without disturbing their normal representation was quite effective. Thus, assortment of secondary indices was developed to reduce access time to flat files. When the development of the hierarchical and network database models took place in the 1960s, the need for indexing became less urgent, as the use of links allowed direct access to any record desired.

As the volume of data and the number of users increased rapidly, the hierarchical and network database models could not keep on meeting the requirements of large database applications [1-3]. Therefore, the relational database model was proposed as a step ahead in terms of data modelling. The data in a relational database was organized into relations such that a relation stored the values for a single entity set. A relation was assumed to be a logical construct whose content was not dependent on the number of users or the volume of data. However, being a logical construct, the content associated with any relation in a deployed relational database made it a performance bottleneck when compared to the other components of the database system. Database designers were faced with three competing objectives: high data independence, low data redundancy per relation, and low access time per operation.



2. Clustered Indexes

It is common to picture a database with a single larger relation and have access paths described for this larger relation. To support query processing efficiently however, one typically decomposes real world entities into smaller entities, called relations. A relation includes the same properties or attributes as the real-world entity, but a tuple in the relation represents a grouping of the same properties for a specific instance or entity. All tuples in the same relation are associated to the same real-world entity. For example, consider the ancestor of all databases, the genealogical database. A person relation contains all the properties concerning persons, but a tuple in the person relation contains properties concerning a specific person, such as the person's name, father, mother, and date of birth. Consider next the relation for country capitals. Each tuple of the country capitals is associated to a capital city, while the properties corresponding to that capital city are the capital's name, and the country it belongs to. Clustered indexes support finding information in a relation associated with a particular value or a small set of values of the index keys.

The first use of a clustered index dates to the 1960s, a technology that was implemented both in the System R relational database and in the multivalued data model embedded in the PICK database. The clustered index pages are the physical, on-disk organization of the items in the relation, ordered by the key attribute values. In general, the relation pages are organized in the shape of a B+-tree whose leaves are the relation data pages, that point to the actual data in the relation. Because the keys are physically ordered in the relation pages, all tuples that are close to one another for the ordering of the key attributes are physically stored that way, on the same or physically nearby pages. If a query asks for the tuples with a specific key value, or for the tuples whose keys are in a small range, then a single disk access to the index suffices to return all tuples in the response, and probably even less time than it would take for other indexes. Because the relation pages are organized as a B+-tree, insertions and deletions of tuples in the relation can be performed efficiently.

3. Non-Clustered Indexes

In a non-clustered index, the entry of an index is an attribute of a relation, but the records themselves are stored as a separate entity. The index typically has an index structure that supports efficient searching so that, for any value of the indexed attribute, the identity of one or more records containing the key value can be accessed efficiently. Non-clustered indices are widely used for secondary access on a relation. The main attribute in the relation can be organized in the clustered index; secondary accesses performed using the other attributes of the relation are organized using a non-clustered index on the attribute.

Compared to a primary index, a non-clustered index is less efficient, needing to perform an extra I/O call to retrieve the actual records containing the data. But the functionality of non-clustered indices allows users to define additional access methods that can be maintained to make retrievals on the non-clustered index less costly. A relation can have multiple views. For views defined on a single relation, additional indices on that relation defined on the queried attributes make it easier to handle other effects of personalized views on various attributes. Thus, the need for a personalized view that requires access via a different attribute than the primary attribute can be efficiently supported with the assistance of a non-clustered index.

4. Comparison of Clustered and Non-Clustered Indexes

Clustered and non-clustered indexes are two types of database indexes that differ in structure and purpose. A clustered index determines the physical order of data in a table and stores the data rows at the leaf nodes of the index. Each table can only have one clustered index because the data rows can only be sorted in one order. The clustered index key is the primary key of the table, and it exists in the leaf nodes of the index. The leaf nodes also contain the actual data: Data rows. The non-clustered index is a structure separated from the data rows and sorts of data by a key column that is a non-clustered index key. The physical order of the data is not the same as the order of the non-clustered index key values. A table can have many non-clustered indexes. You can create each index on a separate key column, and the non-clustered index key can differ from the physical order of rows in the table.

Clustered indexes are faster than non-clustered when the same column is used to search, whereas non-clustered indexes are faster for different possible search columns. Clustered indexes are good for large datasets that are used in a specific order, whereas non-clustered indexes are for smaller datasets or when complete datasets are not being requested. If a table has a clustered index, retrieving data by searching using the clustered index is faster; therefore, the table should be small. If a table does not have a clustered index, retrieving data is quicker by searching with a non-clustered index. When a search is performed on a table that has a non-clustered index, the index is used to look up the data on the main table and its data is retrieved.

5. Query Execution Plans

5.1. Understanding Query Execution Plans

A Database Management System (DBMS) translates high-level SQL statements into low-level operations on database tables before executing them. Because such low-level operations are difficult to program, and because performing them in the wrong order would be inefficient, the DBMS uses a procedure called query optimization to construct an efficient plan. A plan is a tree of low-level operations, called query execution operations, that represent the translation of a SQL statement into actions on the database. The DBMS executes the plan whenever it visits a node in the tree. In turn, each node calls the procedures provided by the DBMS for performing the action of that query execution operation. Execution of the plan tree proceeds in a bottom-up manner, from lower nodes to upper nodes, until the entire SQL statement is executed.

Query plans can be very complex. They consist of sequences of query operators and access methods. Query operators represent the various relational algebra operations, such as joining, filter, and project. Access methods represent the various ways that a DBMS can read data from storage, such as scan, index lookup, and random lookup. Parameters associated with the operators indicate the predicates and join keys for the operations. Parameters associated with the access methods indicate the files, indices, and blocks being accessed by the methods. For example, for a join operator, there are parameters that define the selectivity estimate for the join operation, the inner relation's name, and the outer relation's name. For an access method, the parameters can indicate the index or data file, together with the record ID or index entry. Each query execution operator also has a unique identifier; when the query execution plan is executed at run time, this identifier is used to indicate which plan node is currently being processed.

5.2. Components of Query Execution Plans

There are five basic components to every query execution plan: (1) the input relations, which are the base tables referenced in the SQL statement; (2) the operators, one for each physical operation required; (3) the methods, one for each physical operator, detailing how the operation will be performed; (4) the access paths, one for each physical operator, describing how data will be retrieved, and (5) the estimated resource usage, one for each operator, representing the total cost to run the operation being described. These components are both identifiable from an actual execution plan, and important for understanding query processing. The input relations are the tables specified in the query's FROM clause. Unless the query contains a joint, the query execution plan will usually contain only one input relation. If the query contains a joint, however, the query execution plan will typically contain one operator per base table being referenced in the join's ON clause or referenced in a WHERE clause. The operators or physical operators are the physical implementations of logical operators originally created by the SQL statement. Some of the more commonly used operators are a logical joint, a logical update, a logical delete, and a logical insert. Both logical and physical operators have a similar role, namely mapping specific input sets to specific output sets. The primary difference is that logical operators represent a mapping from a set to a different set, while physical operators represent a mapping from a set to an empty set and may furthermore have side effects.

5.3. Interpreting Query Execution Plans

Query execution plans can be visualized in many ways and using many notations. Different systems contain different features that might be explicit. Features not discussed here will have to be interpreted in an implementation-dependent fashion; we will focus only on such concepts that are relevant in most DBMS products.

The root of a tree is usually the operator that is going to perform the result output of the query. Most non-select operations set some attribute that will be used to perform the result selection, these are usually denoted as having single output tuples. Instead, some other operations that are used to generate the output will have multi-purpose operand pipeline; these will be usually data flow operators, such as joins. Intermediate nodes have attributes too, that will be used by their parents. Nodes are connected through edges that define operator dependencies, i.e. the parent operator will not start its processing until one of the child operators flush all its output. Leaves of the plans are the operators that will consume data; this can be either data input or a temporary table input containing intermediate results.

In tree executions plans, evaluation starts when data flows down to the non-leaf nodes, and from there control data flow propagates to the leaves. Control flow for the operators, data flow for the leaves. The data flow edges are annotated with important information, such as the number of tuples or their cost. For the operators and tables in the leaves, important characteristics of the computed tuples or intermediate result tuples are annotated.

6. Factors Affecting Query Performance

The database's infrastructure has a substantial influence on the time cost of query execution [2,4]. Choosing a suitable query execution plan enhances the system’s performance. Careful consideration of this influence during the database design phase can significantly reduce or eliminate a query execution plan's redesign expense later, in addition to the alternative cost due to longer execution times. Cost-based optimization aims to minimize execution time by inspecting all possible plans, gathering information related to their cost, and comparing. Some of this information is available from the metadata maintained in the system catalogue. The remaining operations comprise scanning the base table(s) for the query, collecting statistics about frequently queried columns or relationships, and estimating the gathered statistics. Index selection is at the core of these optimizations. Different orientations of query constraints will require the use of different indexes.

The database designer's job is to choose an array of useful indexes that will speed up query performance while keeping overhead minimal. We discuss the optimization configuration trade-offs related to index selection in this section. We go on to discuss how join operations are affected by database design. Joins, the various techniques used for their execution, and some of the design-related issues that influence how these joins are executed are covered. Finally, we discuss data distribution. Understanding how data is distributed impacts the selection of correct query plans. Certain considerations about data distribution also come into play during the index selection process. As such, we look at data distribution from the discussions above and then finally look at the role of data distribution during data loading.

6.1. Index Selection

An efficient selection of indexes strongly influences the performance of the access methods of a query optimizer. Database systems support a variety of index types, including B+-tree indexes, hashed indexes, full-text indexes, R-tree indexes, and Bitmap indexes. These indexes can be built and maintained on demand for single relations, or clustered by join columns to provide for efficient join operations. Certain non-indexed access methods may also be supported to provide for sparse and more efficient query processing at query compile time.

An index is typically specified for a column or a set of columns of a relation and can be accompanied by a specification of order of sorting and cardinality properties of its attributes. Certain databases allow functional indexes to be defined on non-attribute functions, and composite indexes that support several indexes together at a go. Indexes can also be defined as unique and enabled for non-null attributes only.

Although several techniques and algorithms have been suggested for the selection of an optimal or near-optimal set of indexes, many systems choose to provide for the ad-hoc or semi-automated selection of indexes due to the complexity of the problem. Indexing remains a subject of active research, with emphasis on personalization and dynamic index construction in order to reduce overhead.

6.2. Join Operations

In this section, we describe other factors that affect query performance. First, we note what makes joins interesting, and present a model of join operations. Next, we investigate the relevance of join selectivity and motivate the use of special-purpose join algorithms. Finally, we present a few words on join strategy optimization.

Joins are perhaps the most important operations in multi-relation queries. Joins relate many tuples from one relation with tuples in other relations. When executed, the number of output tuples can be larger than the number of tuples in the relations that are being joined. Furthermore, other operations, such as selections or projections, cannot eliminate tuples from the final result of a join. Joins puzzle the query optimizer because their cost depends both on the sizes of the relations and on their content. Cost estimation of joins has led to the definition of cardinality estimators based on very little information, such as the number of tuples in the original relations, and the number of distinct values of the attributes being joined. Relations that are joined may be very large, but if the selection conditions of the query can be used to greatly reduce their size, the cost of the join may turn out to be very small. Even selectivity estimation for individual joins has proven problematic.

For performance tuning of very specific operations in our system, such as join selectivity estimation, we rely on the use of small sample-based statistics. We need to care about second-order effects. These second-order effects include output result size effects, either due to selection conditions or to very selective joins appearing somewhere along a query operator tree. In particular, handling join selectivity modelling during cost estimation would allow us to explore the effects of outlier effects before we start execution of the query and use a smart dynamic programming strategy to compute join result sizes on-the-fly.

6.3. Data Distribution

The shape of the data distribution plays a very important part in the properties of any query optimization algorithm. In general, we can assume that we must optimize a predicate over the relations involved in a query and that this predicate is defined over the attributes of these relations. In the ideal situation, we can assume that the data distribution is uniform and that we shall visit the pages of the database as randomly as possible, when executing the predicate. With this assumption, the distribution of the computation time over the pages will be uniform and the optimization problem is simply to minimize the total cost of all the accesses.

Unfortunately, due to the shape of the data distribution, the cost for pages not conforming to the distribution will be much larger than for pages that do conform to the shape of the predicate distribution. The costs will be much larger in the case where the data distribution conforms to these shapes, leading to a sharing of computations over the different pages by the involved relations. The local distribution, defined by unique combinations of the attribute values of the relations, characterizes the page access costs. However, the nature of the page access costs is very important for optimization. Simple thresholding cannot be simply optimized. In this case, the factors influencing the page access behaviour must be treated separately. Therefore, the query optimizer must exploit this distribution information, which may be local or global in spread of this information over the set of stored relations.

7. Index Maintenance

Databases would not be very useful if they were static; almost all database applications involve dynamic data. Thus, both the size and contents of a database are constantly being modified by creation, deletion, and modification of data. We know that selecting the best index scheme at each point in time is important for query performance; however, just selecting the proper index at a constant interval is not enough, as performance can degrade before the next chosen index is adopted, due to data modifications. Thus, index maintenance becomes important for database systems.

Also, once the data characteristics, as they relate to query performance, are known, we may not only want to optimize our choice of index, but more importantly, we may want to choose which operations should be performed next on the indices. If for instance an index is getting too large, which could make index updates costly, or if an index is highly unbalanced, which would increase the time needed for searching using the index, we may want to delete the index. As for choosing which operation to perform on the index, we can optimize our choice based on the choice made on the base relations, that is, we must select an operation that minimizes the expected query delay. Having several indexes can greatly speed up access for a single operation; however, in an active system it is possible that too many updates are necessary to keep the indexes accurate, i.e. the index maintenance cost gets too high. Thus, techniques of index maintenance are important areas of research to make the benefits of having an index outweigh the cost of maintaining it.

7.1. Importance of Index Maintenance

Database systems are characterized by an implicit agreement between the users and the database manager: while the users do not interfere with the internal processing mechanisms, the database manager guarantees the automatic reorganization of data for efficient process execution. However, with the increasing use of database managers to support large and complex applications, it became evident that, in some cases, only the users have an effective understanding of the database usage patterns that directly determine the performance of the system, and therefore they are in the best position to assist the database manager in effectively managing the data.

While, at design time, the user can suggest tuned access paths by providing hint commands to guide the database manager in the query optimization process, during the regular functioning of the system, it is up to the database manager to recognize when granting this structural suggestion can enhance its effectiveness to respond to temporal and acritical bursts of requests of similar nature, on data whose intrinsic characteristics and application usage models warrant special treatment. The problem of fulfilling this implicit agreement throughout the entire lifespan of the database system is one of index maintenance. Accordingly, different techniques, such as duplicate data, partial indexes, and index lookup tables have been suggested. However, the most traditional and most implemented type of index maintenance is based on the idea of reorganizing an index – that is, rebuilding it from scratch.

7.2. Techniques for Index Maintenance

Fielding and Eick provide a few proven techniques for index maintenance, which we discuss next.

• Use static indexes sparingly. If the indexes are used on a heavily updated base, then static indexes are often not the best choice. For mostly static data or read-mostly databases, static indexes can offer orders-of-magnitude savings.

• Use static indexes with user-defined maintenance schemes and threshold functions for selected applications. Using user-defined threshold functions with a static index according to user needs can be a good compromise between cost and performance. In their experiments, it was shown that for mostly read and heavily updated indexes, using user-defined threshold functions can yield a performance improvement of two orders-of-magnitude.

• Replicate dynamic indexes on multiple data sites. Replicating dynamic indexes speeds up access, and the cost of maintaining the replicated indexes is more tolerable than the cost of accessing nonreplicated dynamic indexes. However, the best approach here is to combine periodically updated replicated static index with periodically updated dynamic replicated replicas near the access sites. The filter page technique can offer further improvement.

• Cache dynamic indexes, but make sure that the cache is big enough to keep the active parts in memory. Often, dynamic indexes benefit from caching. When the index is cached, the cache management needs to address caching performance. Addressing this problem may include implementing LRU caching or use of more complex techniques such as pseudo-LRU or frequency-based methods. With good caching, index access times are comparable to those of static indexes.

• Choose hybrid techniques to best meet your needs. Hybrid techniques combine dynamic index techniques with static concepts. However, objects in these structures need maintenance as well. Various examples of hybrid techniques are presented. With either hybrid keys or objects internally divided into variable-length records, these techniques can avoid frequent maintenance and speed up access as well.

7.3. Impact of Index Fragmentation

Indexing is an integral part of a database management system. Inadvertent disorganization could make indexes larger, thus the index lookup more expensive. An index is considered fragmented if the index pages are not stored in contiguous clusters. In addition, pages may become poorly utilized over time, leading to pages having too few keys. These pages require additional I/O for index traversals, leading to a performance penalty for index lookups and other index operations. Underlying systems maintain index fragmentation heuristically, by means of space overhead parameters. The higher levels of overhead parameters for a given workload indicate a lower expected working set. In addition, since packing as many keys as possible minimizes page I/O, this exhibit relates the kernel object size to the overhead parameters. Furthermore, fragmentation has a negative impact on database operations such as insert, update, and delete.

External fragmentation is defined by Page Density and Page Usage ratios shown in the graphs. Internal fragmentation is defined to be packing ratio which identifies the splitting of entry keys in the overflow pages. Higher density minimizes the overflow and fill the pages substantially to improve search performance. A completely empty page also introduces overhead in terms of wasted space. The diversity of key and data sizes, together with data volatility, directly impacts the page utilization characteristics. Eventually metadata caches are searched for cached objects propagation. Additionally, it is not only important that the B+-trees achieve the best performance at steady state; it is also critical that they properly adapt to key insertions and bursts, even if the trees are in a fragmented state. Building caches larger on key bursts and smaller on key insertions will result in network bandwidth and disk access savings. It has been therefore proposed that slowdown or diffusing lookups during slowdowns would recover during lookup upticks.

8. Best Practices for Indexing

Indexes are powerful tools that can greatly enhance the performance power of database management systems (DBMS), but they can also decrease overall performance and even reduce performance below the level without any indexes whatsoever. Indexes take up disk space and can slow down data modification operations. Hence, they should be used judiciously and according to best practices, akin to the many measures that can be taken to avoid redundancies, data anomalies, or data integrity and consistency issues in the database schema design process. In this section, we discuss several best practices for creating and maintaining good indexes, such as selecting the best type of index, avoiding adding too many indexes and redundancies, and monitoring index usage. These guidelines should help relievers of DBMS to optimize the performance of their data retrieval and modification operations.

8.1. Choosing the Right Index Type

While it may seem overwhelming at first glance, the world of index types is not as complicated as it seems. The reason we have so many index types is because the data stored in a database system is in many different forms, and the queries that retrieve that data are also in many different forms. If your database supports only a few types of indexes is a dangerous approach to index selection. Existing index types have evolved through a long history of research and development into very sophisticated and efficient solutions tailored to classes of databases and classes of queries. Leveraging the benefits that well-chosen index types can provide can significantly improve the performance of your queries or, sometimes, minimize the performance hit that they incur.

Choosing the most appropriate index type for a query is both the simplest and most complex part of the indexing process. It is simple because it can often be done by following a set of heuristics that map query characteristics onto available index types. It is complex because there is usually no one-size-fits-all answer, and the best index for a query is not always the best index for a related query. Specialized index types can only index specific types of data and take advantage of specific types of query predicates—i.e., equality conditions, inequality conditions, or match queries. A data set containing one type of data might benefit from a specialized index while a data set containing another type of data would be best indexed with a completely different specialized index.

8.2. Monitoring Index Usage

Evaluation of index usage can provide a measure of the effectiveness of an index. For query workloads of moderate size, it may be possible to evaluate index usage simply by examining the queries in the workload description. However, for database systems that support large or heavily modified databases, such as general-purpose systems that incorporate large amounts of input from multiple users, or transaction-processing systems that incorporate large amounts of update activity, the task of index usage evaluation may require an approach that is more sophisticated.

Although there are some challenges involved in performance tuning in large commercial database systems, we are fortunate that many such systems have now been operational for many years. Over this time, they have provided input to several projects that have attempted to build systems that can automatically search for and eliminate redundancy in database schemas so as to optimize the performance of data loading, querying, and updating processes.

Index usage, if correctly evaluated, can also give valuable clues about how the existing set of indexes should be modified, or whether additional indexes should be included in the schema or other indexes removed. The monitoring and logging of index access are closely co-related to monitoring direct access paths to data, as a B+-tree index represents a logical ordering of keys. A full scan of a B+-tree index should consist of successive visits to the leaf pages, as the number of scans are usually counted based on the number of times all leaf pages of an index are read.

9. Common Pitfalls in Indexing

There are two common pitfalls in indexing: over-indexing and under-indexing. The consequence of over-indexing is that we will incur high overhead for executing insert, update, and delete operations due to index maintenance and high storage overhead in storing the indexes, thus making the database system overall inefficient. For the case of under-indexing, we lose the chance to utilize the indexes to speed up query execution and thus render the database system as inefficient as a system without indexes. Balancing between over-indexing and under-indexing is thus crucial to efficient database systems, especially for those working in high demanding online scenarios where both retrieval and update require low overhead.

9.1. Over-Indexing

We say the system is over-indexing when there are too many or redundant indexes for the database. It is common that indexes incur high overhead for executing insert, update, and delete operations because every time when a data page is modified, its corresponding index pages must also be updated. More importantly, each index page must be read, modified, and written back to the disk and this introduces high access pressure on the disk storage. Usually, creating an index takes a certain period of time. Assuming the database is used for read access only, the database can benefit from the index once the index is created. After that, there may be some delayed increase in update time. However, if the database is used for both inserts and read access at a low balance, the excess time taken to maintain an index for frequent inserts may at times outweigh the benefit of using the index for answering queries. Moreover, with large volume of data, the storage overhead for storing the indexes can be considerable.

The use of common and specialized indexes to speed up transaction execution is certainly appealing in a database environment with many concurrent users executing a variety of transaction types. However, if every possible index is built for each relation in a common database on the assumption that one of them may help accelerate processing and response time, then the question "are we optimizing?" does not have a straightforward "yes" answer. Performing a join or a query on the result of a two-way join operation usually results in an operation that is more expensive than the corresponding operation that is performed on the original data relations. Note that for a join operation where several indexes joins at several levels of the join-tree are performed, if some of them do not run faster than a sequential scan by classification or filtering, then we are, in effect, improving the efficiency of some of the join operations and decreasing the efficiency of some (possibly many) other join operations. This meritless overuse of indexes would imply that we are going to delay many of the update transactions much longer than we could have been using common data structures, compromising the latencies on several other transactions since we will be incurring the overhead of the index maintenance. The situation becomes worse when we realize that some index accesses would incur not just an expensive I/O cost since the memory page could not be found in main memory but also a page fault by the classification or filtering. To top it all, certain remote procedure calls that involve messages transferred from a remote server using low bandwidth networks may exceed, by far, the estimated execution time of transaction execution.

In summary, index maintenance, especially post update, becomes expensive whenever we make many updates and/or insertions. Furthermore, the performance degradation will be more pronounced for indexes that index many values and/or tuples and/or are at a lower or intermediate level of a join.

9.2. Under-Indexing

Having no index, or an incomplete index, is another common problem. Each index typically models only a subset of the queries that may be issued. For example, a database used to issue reports on insurance rates may only contain a limited index, providing quick access on the most common set of keys. How does the system speed up the other types of queries that do not correspond to the keys in the limited index? The answer is that without additional help, those other kinds of queries will be processed much slower because no useful index is available. The common term for the absence of an index is called under-indexing. Data joins are often much slower due to the also common characteristic of the absence of dense join indexes. Think carefully about all future queries when designing the indexes. Relatively small indexes can be constructed to speed up relatively large numbers of queries, just as large indexes can be constructed to speed up relatively few queries. Not only a particular query, but type of queries should be done carefully, and the type of selective attributes should be examined. It is often the case that some queries may be issued on some attribute combinations, but not all combinations, and on those combinations, the cardinality is also likely to vary in a fairly large range. This leads to a well-known indexing pitfall, called under-indexing. Without an index on a particular query, the query must scan the entire relation even if only a few tuples match, or if the query involves a range, the query will scan a much longer segment than what it should scan.

10. Tools for Query Optimization

In terms of real database systems that utilize the various optimization techniques that we have described, there are several well-known commercial products, as well as sophisticated tools that help with the task of optimizing queries on databases. The term Database Management Systems (DBMSs) encompasses a very wide range of systems, varied in complexity and use. For the purposes of this section, we consider only those systems that can handle large datasets, made available in an efficient way. By this definition, we will include systems such as

Postgres, Oracle, as well as several “big databases”, including the ones found in multi-computer installations.

10.1. Database Management Tools

Database management tools (DBMT) are software packages and libraries providing modules that manage the input and output data of the databases. A database management tool is an answer on the DBMS problem, one of the first in the impulse of the rise of the interest in databases, is the location of the data, i.e. the disk storage. The basic resource used for retrieval of data is the index structure, and a lot of tools involve in the usage of the proper index. The surface resources are cache memory, disk buffer, disk, which are usually resized in the hardware design. The design of these components creates micro-architecture of a computer system, providing all the improvements for the tools of databases management. According to database administration, some external or other included in the system DBMTs are diagnostics and feedback tools, Analyser’s, Monitoring tools, Tuning manager, Indexing management tools (IMT), SQL tuning tools, Workload-Manager, data warehouse managers.

10.2. Third-Party Optimization Tools

Tools designed to help administrators optimize their systems are broadly classified into DBMS management tools and third-party optimization tools. Management tools are available either with the DBMS installation itself or from the suppliers of DBMS software. The distinction is that third-party tools are designed for heterogeneous environments, in which they interact with different DBMS systems. Heterogeneity is, therefore, a key aspect behind the development of third-party optimization tools. The portability of access patterns and the independence of any specific database management system are at the heart of what is implemented in these tools. Third-party tools present several advantages, at least for relational databases, when compared with management tools. They most commonly produce detailed reports of the overall status of the databases under consideration, pointing out problems in terms of poor performance that may be related to the absence of a recommended index, due to inappropriate data distribution, security problems, and so on. Many commercial and laboratory tools are available for various database systems. They certainly are attracting the interest of the scientific community as well. The availability of this set of tools indeed shows that this part of the knowledge which is relevant to the analysis and optimization steps is most easily implemented. These implementations are the data collection and access pattern analysis that support database design, and the design of suggested rules of thumb that help data administrators tune and manage small-scale databases.

11. Case Studies on Indexing Strategies

This short chapter will present some case studies on different organizations in different domains, that explored different aspects of indexing technology. The objective of this chapter is not to provide every such case study, rather present a few pointers in each major area of database systems, which used unique ways of assessment, in either perspective or implementation phases. This appendix should be taken as pointers to the reader for future exploration.

The case studies we discuss are: A case study from the business sector, where they discuss the requirements of using a database management system in an “analytic model” work in systems. This case study discusses some needed extensions to the native database indexing mechanisms, to support a multi-level database architecture. It is interesting as it puts forward the needs of extending the native capabilities of a commercial system, regarding indexing mechanisms. The last study points in the other direction, adding indexing mechanisms to some state-of-the-art text-processing systems, to extend their capabilities. The subject that we will cover is the implementation of a universal data blade. This blade takes as composite modules, a novel event-based indexing device and the R-tree. The last example that we will present is a product from a local commercially available database product, that started as small business and moved sideways into the voice processing market. We have expressed previously the need for commercial solutions for small to medium businesses. We have also compelled that most database systems are custom made. The product can address both.

12. Future Trends in Indexing and Query Optimization

Indexing and query optimization have progressed significantly over the years, and we are witnessing a radical change with the advent of large data collections and powerful computer systems that are widely and cheaply available. New hardware capabilities have a tremendous impact on query processing techniques. To this end, we will summarize some important architectural advances, and their impact on optimization, that will affect future databases and IR systems. During the last few years, we have observed a continual drop in RAM prices with an increase in processor capabilities, while secondary storage systems remain much slower than main memory systems. Hence, it has become possible to put large collections of data in main memory and expect that the systems will perform faster than previous ones that relied on disk storage. Key parts of a database or inverted index can be cached and queried directly from memory. Although the basic search algorithms remain the same, coaching alters their performance characteristics, and hence their implementation choices. For example, for query evaluation, coaching allows the system to forgo costly disk I/O at the expense of access latency for the most popular pages. The simplicity afforded by main memory architecture suggests a design philosophy in which main memory dominates other considerations. For example, although it may be cost prohibitive, at least in the short term, to develop a system that can process entire large-scale data collections in main memory, such an environment becomes one of data retrieval, as opposed to data management, where page caches minimize secondary storage I/O, disk latency, and data accessibility.

13. Conclusion

Generating Effective Queries. Generating effective queries that use only accessible data is key to the performance of a query plan. Instead of simply rewriting the original queries, which all systems need to do, we should align them with the rewrite methods used in the system startup to create the index. Then they will return results that are consistent with the index, should perform faster, and can be more complex, allowing for different access methods that would not be used otherwise. If there is no useful plan for rewriting, we must create a substrate to allow the discovery of new ones. This can be like a pre-index, with light-weight features and approximate scores. We must support the query rewriting from text-based queries to the generating or retrieval model used in the final execution of the query. In order to enable generation in the query rewriting, we can use optimizing generative methods that map the query as a token sequence. We map this token sequence so that it describes how to obtain the actual expected output with the least tokens possible.

Research Directions. There are still many open issues in the field of indexing and optimization of database queries. New models to represent the index space and better understand the design trade-offs are needed. Furthermore, heuristics for choosing indexing parameters for different use cases are still unclear, making the application of the methods difficult for practitioners. The use of machine learning for choice of embedding, or selection of the indexing parameters is an area of active research, but still in its infancy. Therefore, all the challenges of managing the huge amounts of available knowledge on a scale can only be properly addressed with a combination of already-known methods. Finally, there is still the problem of selecting which functions and models to build the index as for some of the common types of indexes used in retrieval, there are no embeddings to get mappings to and from vectors which would guarantee that the vector space does work as a vector space in the mathematical sense of the meaning.

References:

[1] Bertino, Elisa, et al. *Indexing techniques for advanced database systems*. Vol. 8. Springer Science & Business Media, 2012.

[2] Yaqoob, Ibrar, et al. "Big data: From beginning to future." *International Journal of Information Management* 36.6 (2016): 1231-1247.

[3] Das, Sudipto, et al. "Automatically indexing millions of databases in microsoft azure sql database." *Proceedings of the 2019 International Conference on Management of Data*. 2019.

[4] Chaudhuri, Surajit, and Vivek R. Narasayya. "An efficient, cost-driven index selection tool for Microsoft SQL server." *VLDB*. Vol. 97. 1997.

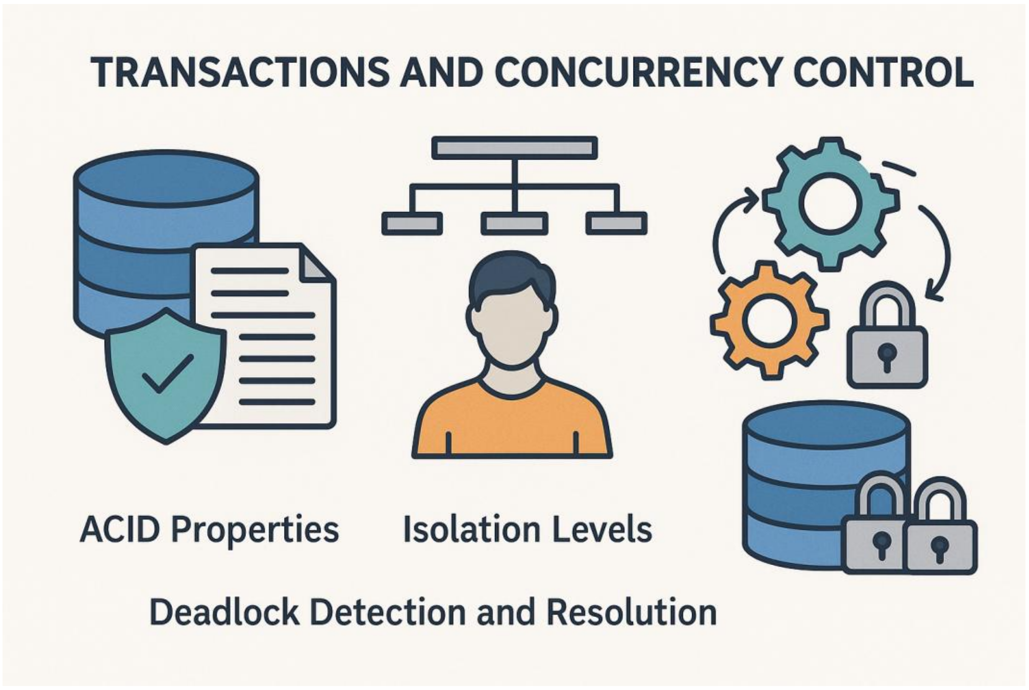
Chapter 4: Transactions and Concurrency Control

1. Introduction to Transactions

A transaction is a logical unit of work that contains one or more operations, such as read or write, on data. These operations must follow a specific order according to the rules and semantics of the applications that use the data [1-3]. A transaction may be short, involving only a read operation on a small item of data, or long, involving thousands of operations on millions of items of data. The typical examples of database transactions are the operations related to an automatic bank teller machine and a reservation system for airlines and hotels. In a bank teller machine, a user may do one of the following operations: deposit money into an account, withdraw money from an account, check the balance of an account, or transfer money from an account to another account. Each of these operations is a transaction that modifies the state of a bank account. A transaction for a bank account is modelled as a series of operations over the account. A reservation system keeps track of the status of airline and hotel reservations, which may be full, empty, or partially reserved. A transaction for an airline reservation may be to make a reservation, cancel a reservation, or change a reservation. Similarly, a transaction for a hotel reservation may be to make a reservation, cancel a reservation, or change a reservation. These operations read and write values stored in some tables.

Transactions are important for several reasons. First, correct and accurate answers to queries are essential for the integrity of any information-based system. These queries involve reading and writing values in one or more of the tables.

For any query operation, the answers must match the meaning of that query. Second, a poll is used to cache the state of remote sites involved in the answer to the query. This cached state must be coherent and must reflect the latest changes made at those remote sites in response to other queries. Third, the cost of processing query operations can be reduced if results can be cached. The process must be able to retry the transaction and to use the outcome of the retry in deciding whether to cache the results.


TRANSACTIONS AND CONCURRENCY CONTROL
ACID Properties
Isolation Levels
Deadlock Detection and Resolution

2. ACID Properties

A transaction is a series of operations that are evaluated as a single logical unit of work. A transaction must exhibit the following properties; collectively known as the ACID properties. These properties guarantee that database transactions are processed reliably.

Consider a bank where an account will never have negative balance. Consider the two operations Deposit and Withdraw that are to be executed as a transaction. Suppose we are withdrawing some amount from an account and simultaneously, a deposit transaction operation is being executed. It is possible at some point that account balance may be negative. Generally, it is assumed that information is passed through to a new state which may not allow undoing the operation. Even then, these operations must not be done simultaneously.

Consider a transfer of some money A from account x to account y. The account must withdraw A from x and deposit A to y in such a way that no other transaction does withdraw/deposit operation in between them. If so, at some instant, the account balance may be negative. Transactions with such properties are said to be concurrent, while others are not. In a bank scenario, it is very important that the transactions be consistent, reliable and predictable. However, the concurrency control must be done without compromise for performance. Transaction management protocols implement transactions such that they conform to ACID properties. Such a framework of protocols is called a Transaction Model.

The ACID properties are as follows: Atomicity states that either all the operations in a transaction execute or none execute. Consistency states that a transaction cannot violate the integrity constraints. Isolation states that concurrently executing transactions cannot interfere with each other. Durability states that once a transaction commits, the updates should be permanent, that is, they survive subsequent failures.

2.1. Atomicity

Transactions should be atomic. This means that what is done by the transaction is all done or none of it is done. The motivation for this is easy to explain. In the simple bank example, if we were to transfer an amount from one bank account to another, it is possible that we could inadvertently take the money out of one account and not add it to the other account. For example, suppose that the transfer transaction has two operations, one for subtracting the money and the other for adding the money. Assume that when the transaction is in the process of executing these two operations, a rogue process tries to watch the transfer operation and executes, in parallel to this transaction, the operations to withdraw money again, even though the transaction has already partly subtracted the money from the first account. If there is no atomicity, it is possible to have one account with less money than it originally had, and the other account has more money than it should have. The system as a whole is left in a bad state.

The transaction is considered to have committed when it has completed its operations successfully and the transaction is rolled back when its operations cannot be completed. Committing and rolling back is to record these processes with the help of a log. If transfer of money from one bank account to another cannot be completed, the log records that event and the log can be used to return database from bank transaction to its original state. When transactions are not atomic, the concurrent execution of the transactions will make the system yield some bad results. Atomicity can be ensured using locks and locking protocols.

2.2. Consistency

Consistency is a criterion for the correctness of a transaction[2,3-4]. The execution of a transaction on a database takes the database from one valid state into another valid state. A valid state of the database is a state that satisfies all the declared integrity constraints on the database. Some integrity constraints are declared using a Data Definition Language. For instance, the unique constraint that maintains the uniqueness of the primary key is declared in the DDL. When a transaction modifies a database state, the intermediate states may violate some of the integrity constraints declared on the database. However, the transaction must ensure that execution of the transaction does not violate any integrity constraints of the database, before execution of the transaction and after completion of the transaction.

The constraint called database consistency can thus be stated as follows. If a transaction modifies a database state, the database state need not be consistent after the transaction reaches a commit point; the transition from the consistent state to an inconsistent state may occur. However, the database must be made consistent again before the transaction reaches the commit point. The database can become inconsistent after the commit point where the commitment of the transaction is guaranteed. However, if another transaction executed after the commit point reads the modified data items and executes other operations depending on their values, then database inconsistency may occur. Therefore, the operations on the database by a transaction must preserve the consistency of the intermediate states of the database.

2.3. Isolation

On the most basic level, isolation ensures that if transactions are executed simultaneously, the results of the execution are the same as they would have been if the transactions were executed in some sequential order. But this statement contains several colourful words requiring further explanation. “Executed” means that the execution will never be rolled back, and this statement applies only to committed transactions. Furthermore, several sequential orders are possible when the executing system allows some kind of interleaving of operations from different transactions. These interleaving are called schedules. While thought of this way, isolation seems to alleviate some of the problems of concurrency control; in fact, it is the other way around. Isolation is a consequence of concurrency control, meaning the isolation properties of a system are determined by the underlying concurrency control mechanism.

Quotations in this case are used to emphasize that this is the behaviour that should be observed, not the way it is implemented. A naive implementation would be on a file level, meaning that only transactions that are touching the same file would impact each other's performance. This would work to achieve the proper effect, but it would also impose a considerable bottleneck on performance. More elaborate implementations do read and write locks at the level of individual records in the file. Higher sophistication implementations may employ additional means to selectively find those transactions that are indeed interfering with the others' results and allow them to interleave their statements without impacting the validity of the results. This serves to improve throughput without sacrificing isolation. This notion is summarized by saying that isolation corresponds to serializability.

2.4. Durability

Durability refers to the ability to recover from hardware crashes or logical failures in software [1,5]. A database may be dropped but if some pages in disk storage are not properly erased then the information from the old database may be recovered. In a distributed environment, a client can follow a faulty path on the network, move around different sites and obtain copies of the same data or related data that have been changed by update transactions using propagation messages. A physical design of a database normally relies on some system available on the storage devices to recover from media failures.

The durability requirement is, in a sense, the most difficult to satisfy and relies on careful design of the implementation. For instance, modifications made to disk during the processing of transactions are not done immediately but rather collected and then written to disk in batches. During a system crash, or even during a system failure due to software bugs, a batch may be partially output to disk, creating an inconsistent database. Moreover, after modifications are made to disk, the data may reside in volatile storage and be lost due to a system crash. Thus, regardless of careful design of whatever storage management has been implemented, there is a risk of partial changes being made to disk at any transaction commit point. A decision that is made by the implementation to cancel all changes or complete all changes must be made on the updated database and on the messages used for inter process communication in a distributed environment, and such a decision may not always be simple. Having addressed the Durability Requirements, we will explore how transactions are implemented inside a DBMS in the next two chapters.

3. Isolation Levels

In database management systems, transactions provide an important mechanism for controlling concurrent access and updating of the database. Transactions encompass several operations or statements, which must be executed in an atomic way because a proper database is designed to meet the ACID (Atomicity, Consistency, Isolation, and Durability) properties. There are two components of a transaction: data modification statements and data query statements. Data query statements are the set of search statements that defines the transaction. The main goal of a transaction is to provide isolation, which means that each transaction should operate like it is the only transaction in the system. However, this is not true. For the execution of transactions that perform read and write on the same data item, the effect may be that one transaction is executing just before or just after the other.

When it comes to the isolation property, there is a trade-off between consistency and performance. By allowing some inconsistency to occur for some duration, we can achieve a greater degree of concurrency that yields a better performance. A database system must provide different degrees of isolation based on the requirements of application programs. A few application programs can tolerate a high degree of inconsistency during some periods. For these application programs, we would normally choose a low level of isolation that results in better performance, while for other applications, we would choose a higher level of isolation that guarantees validity and consistency. Various levels of transaction isolation are possible, including the following: Read Uncommitted allows transactions to see formally uncommitted data changes made by other transactions. Read Committed guarantees that any data read by a transaction is committed now it is read, and not modified by other transactions before the reading completes. Repeatable Read guarantees that all reads within the same transaction will see a consistent snapshot, while the transaction itself is modifying data. Serializable prevents other transactions from modifying any data accessed by the transaction until it is complete.

3.1. Read Uncommitted

The lowest level of isolation in SQL databases is Read Uncommitted, specified by the command SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED. At this level, a transaction may read data modified by other transactions, even if those modifying transactions have not committed. Thus, if Transaction T1 modifies data value a but does not commit, and T2 reads a, T2 can see the new value of a. This behaviour can lead to invalid data being read by

T2. Reading may produce data that was created then erased by another transaction, such as the value of a bank transfer that has been cancelled. This is known as the dirty read problem. Dirty reads can be a problem when the data being read is used to produce some result, like a financial report. If the dirty read is performed early in the reporting transaction, the account could be zeroed out by the second transaction that modifies the first, forcing the reporting transaction to show incorrect data. Dirty reads may also cause cascading deletes, because if Transaction T1 reads data being modified by Transaction T2 and T2 decides to roll back the modifying operation, T1 could end up executing operations based on data that no longer exists.

Databases generally implement Read Uncommitted isolation with locks on modified data. The locks prevent reading, but they do not prevent other modifying operations, which is how the lower isolation levels are implemented. Using this work with caution. Cascading deletes can be avoided with this transaction with the use of appropriate two-phase commit procedures. The advantage of Read Uncommitted levels is speed. Transactions can be completed much more quickly because they operate at the lowest level of locking. This state is therefore appropriate for state information, such as the current transactional information of customers in a database. Reading this data can be done often as a function of time without compromising the accuracy of the result.

3.2. Read Committed

A read committed isolation level prohibits dirty reads; a transaction will only read committed rows. Reads and writes to the same row are committed in the order processed, so if a transaction modifies a row and another transaction reads it afterward, the reader will read the changes made by the writer transaction. These modifications act as both changes and locks on the rows. In read committed, once a transaction has acquired a lock on a modified row, the row is locked until the transaction releases the lock. This level accounts for the case that transactions require a higher guarantee than a read uncommitted level, and need to read a row that another transaction has defined and updated. The transactions at the read committed level will issue a copy of the modified row.

Many database products provide read committed as the default isolation level. Some database engines justify not allowing dirty reads, stating the dirty read might read either at least one runtime error for the transaction that modified the row, or the newly inserted line. Dirty reads, as they provide a view of the modified row based on a new Transaction ID, contradict the core of Transaction Processing since these operations have to be atomic and isolated in every aspect.

Although all locks are released at the time of commit, if you configured the READ COMMITTED SCRAP isolation, the scratch settings will never be removed for the project. Subsequently, the scratch will appear on any new run, and you can see it in the SCRAP box. The transaction needs to set its isolation level to scrapping READ COMMITTED SCRAP before it begins inserting, updating, moving, or deleting rows from the dragged table. Subsequently, it uses a scope parameter to specify the duration of the new setting. The READ COMMITTED SCRAP setting is not allowed if you are running in an isolated transaction.

3.3. Repeatable Read

The repeatable read isolation level allows transactions to read rows that were previously read during other transactions, but without being blocked by any locks taken by the other transactions. Thus, in repeatable read, transaction A may read in repeat mode any consistency level from transaction B that is using either dirty read or committed read in transaction-per-operation mode, but transaction A cannot read the consistency level READ ONLY from transaction B. Also, there is a lock on a previously read row (by either transaction) until transaction B ends.

There is no defined moment when transaction A blocks; that is done by the transaction that is changing the row values. Thus, the performance issue is that transaction A is not able to process rows at full speed, due to potential restarts. And for that not to be a big issue, the transaction should scan only a small portion of the rows (say 5% of the total or less). Otherwise, there would be blocking like in the read committed isolation level.

This isolation level has its practical issues. Notably, in the case of a read-only transaction that is taking on many rows in a scan and blocking all the modifications by other transactions, the modifying transactions will wait at the commit point for transaction B to finish. Meanwhile, they are also producing a lot of locks, which may end up eating memory and keeping many unnecessary non-modified rows on disk. These problems may get worse if the modifying transactions are bigger than transaction B.

3.4. Serializable

Serializable is the strictest isolation level called level 3 defined by The SQL Standard. Serializable treats concurrent updates as if they were executed one after the other, in some serial order. Serializable prevents all occurrence of the following phenomena: dirty reads, non-repeatable reads, and phantom reads. This is done by locking the rows during the execution of the transaction, and the rows are kept locked until the transaction finishes. The main disadvantage of

Serializable is that it reduces the level of concurrency offered by the concurrency control algorithm.

Isolation level 3 guarantees not only that dirty reads and non-repeatable reads will not occur but also that phantom reads cannot occur; that is, a transaction with isolation level 3 views a consistent state of the database with respect to execution of other transactions. An execution of several transactions is said to be a view serializable if a transaction sees the same state of the database during its execution as it would have seen had the other transactions done all their work on complete copies of the database. The idea behind the view-serializable is that a non-serial schedule of these transactions would not have produced intermediate states of the database that differ from the intermediate state that the transaction sees when the other transactions are working on copies of the database. A schedule is view-serializable for more than one transaction if that transaction produces the same output or alters the same final state of the database for all initial states of the database.

Preventing phantom reads is one of the goals of SQL and most other systems. Database information is typically stored in a set of records that can be indexed for efficient retrieval. If a transaction follows the requirement of level 3, then it is guaranteed that after a transaction verifies that a given record exists in the database for some parameters, this record will not disappear until it is altered or deleted by a transaction.

4. Concurrency Control Mechanisms

The concurrency control problem arises in a database because of high activity levels in different transactions which may read/write common data objects. Transaction size is large and the probability of conflicting operations in different transactions is high. The real-world transactions manipulate the data in intrepid fashion that is during the lifetime of transactions a large number of data elements will be modified by some transactions and a number of data elements will be accessed by some other transactions. As a result, transactions are in confrontation with each other. Such confrontations of transactions can cause problems such as uncommitted data, inconsistent retrievals, deadlocks, and extremely long transactions.

There are two approaches for concurrency control mechanisms. The pessimistic concurrency control mechanism precludes conflicts among the transactions by not allowing conflicting operations to execute. The optimistic concurrency control mechanism permits conflicts but ensures that these conflicts do not lead to any type of inconsistency. In pessimistic methods, any type of operation of a transaction may be delayed or be denied. In optimistic methods, only operations that lead to an inconsistency can be detected and undone. Pessimistic methods are starving methods. In optimistic methods, we delay conflict and use a method that, with most transactions, will execute without any inconsistency detection and recovery. The optimistic concurrency control has two phases: the read phase and the validation phase. The length of the read phase is allowed to grow indefinitely while transactions are selectively validated.

4.1. Pessimistic Concurrency Control

Concurrency control is required in database systems to keep the data in a consistent state, as it is being shared and manipulated by many users and applications simultaneously. Concurrency control is achieved in data management systems by several mechanisms. These mechanisms may be divided into two broad categories of pessimistic concurrency control and optimistic concurrency control. The Pessimistic Concurrency Control also known as blocking control prevents the conflict between concurrent transactions by using locking methods. This control methods force a transaction to wait until locks have been released by other transactions, while the optimistic concurrency atomicity is ensured by using transaction timestamps, and by committing a transaction only when it is certain that no other concurrent transactions have accessed the object.

Pessimistic concurrency control methods use locks to prevent inconsistency, and since the locking overhead can contribute to major contention and delays due to transaction delays, they are expensive items in the processing of concurrency control. The overheads of using locks include the lock allocation; release of the locks and the time that a transaction waits to by locked objects. Two locks are provided to a transaction on a given resource, which are Shared Lock and Exclusive Lock. With Shared Lock a transaction can read a data object but cannot write on that and Shared Lock is required when a transaction wants to read an object. With an Exclusive Lock a transaction can read and write a data object, and either the first time a transaction accesses an object or tries to write the object. So, while a transaction has the locks, no other transaction can read or write the locked data item. Transaction locking is implicit in most DDBMSs, and DBMSs to maintain the data integrity. Active transactions would be waiting on locks that are currently held by the blocked or deadlocked transactions, thus causing high execution delays due to the long wait times. Generations of the longest transactions will take the longest locks on the resources and so release the locks faster than those of throttled transactions. So, to reduce the cost of lock delays optimistically the transaction time duration is throttled as the transaction is delayed on locks without any activity.

4.2. Optimistic Concurrency Control

Optimistic concurrency control mechanisms assume that data accesses will not interfere and use timestamps to avoid conflict. At its most basic, all conflicting reads and writes are checked for conflict at the commit point, and failures occur in the case of a conflict. This is known as optimistic two-phase locking. Transaction reads and writes are executed in unprotected memory, and the actual transaction data are simply compared to the original values in the transactions table at commit time. It is possible to extend this scheme to permit conflicting reads but not writes.

The basic optimistic control approach is simple to implement in distributed systems. It can easily tolerate long networks delays, even when the delays are unbounded; it has low overhead for handling normal transaction interaction; it can effectively manage bursts of access to the same data elements; and because it has no blocking, it is very effective on low-contention data. In addition, users are freed from the burden of forced locking, and in some cases the transaction pages are not locked, which allows the sharing of data across transaction boundaries.

Unfortunately, this method can also run into problems. Namely, it is possible for many transactions to be rolled back because they have tried to do conflicting writes, meaning that the commit checking overhead becomes a problem. Furthermore, the protocol does not allow certain actions, such as modifying files in a non-transactional way, since two transactions may attempt to write identical records at the same time. A possible solution to this is to assume there are known low contentions on a certain record, removing the commit checking for that record.

5. Deadlock Detection

This paper studies transaction processing models and their concurrency control mechanisms. Our goal is to assess whether the models and mechanisms are both adequate in executing transactions that contain both read and update operations at any frequency and for any length of time. Inadequacies may be either in the models or in the mechanisms. For example, if the models allow read-only transactions to block an update transaction, thus delaying the completion time of the update transaction indefinitely, the mechanism that detects a deadlock against the update transaction is considered inadequate. On the other hand, if the models guarantee that read-only transactions will never block an update transaction, then such blocking will not occur under a time-based priority ordering protocol, and no deadlock detection mechanism will be needed. There is no a priori way of knowing if a time-based priority ordering scheme will or will not induce a deadlock. Thus, any such induction may, and usually does, create additional execution requirements.

5.1. Deadlock Definition

The preceding chapter described a technique to guarantee that a schedule is conflict-serializable, thus free from anomalies. There are restrictions on resource allocation to ensure that the system to be designed is free from deadlock. While these restrictions will help us to utilize resources so that the system does not become deadlocked, this does not mean that it will always be not before detection algorithms or deadlock prevention methods can be applied, the deadlocked system's resources must be managed by some protocol, so as to allow detection and resolution by the operating system.

A system is said to be in a deadlock state when there is a set of processes such that P1 is waiting for resource R1, which is held by P2. P1 wants an additional resource but is waiting on P2, and so on through Pn, and when Pn wants resource Rn, which is held by P1. In a database transaction-management context, a process awaiting a lock is considered to have no resources. Therefore, the deadlocks we will consider consist of processes waiting for permanent locks, as it were, on resources. Note that we allow a process to be waiting on one of its own resources, as long as it is not being delayed in its working on that resource. In this extended sense, all transactions are potential deadlocks, since the waiting transactions will wait indefinitely if the resource being waited upon is not ever released.

To summarize, the classic deadlocked state is one in which a set of processes are each waiting for some resource that is held by another waiting process in the set. In current transaction environments, with the transaction being viewed as a process, all possible deadlocked states are transactions waiting for a resource lock. In fact, transaction manager functions have the responsibility of enforcing a lock protocol in which such states cannot occur.

5.2. Detection Algorithms

The simplest approach for the deadlock detection is to periodically check for the existence of cycles. This approach is only suitable for small systems. For large systems this will be rather large overhead.

An alternative is to take advantage of the constant flow of request messages. The idea is to save information about the network condition in special data structures. Such data structures are made up of FIFO lists capable of recording the request and waiting messages of the connections disallowing any single connection to be deadlocked. A structure called request list (list of pending requests) is maintained for each connection. A node (temporarily, the requesting node) can add to the list a pointer record denoting the message pending for this connection and the event time of adding the record, if the requesting connection is already contained in the list. If the request comes from a node which is not currently reserved by this connection, all pointers of the records are shifted up the list, and at last the pointer to the new record is appended to the end of the list. During the backtracking phase the waiting nodes are flagged while sending the backtracking message because the receiving nodes must not store any pointer to the flagged nodes on their lists, thus preventing deadlock.

A disadvantage of this deadlock avoidance algorithm is the gradual increase of computation overhead in the request message flow producing a decrease of the system throughput. In large systems with high cost for a message increasing the number of positive messages is of great concern. In smaller systems decreasing the processing time minus the communication time should help to minimize the computation overhead while the request message flow remains low.

6. Deadlock Resolution

We have seen how the presence of a cycle in a Wait-for graph indicates a deadlock. However, we have not yet thought about how to resolve it once it is detected. Deadlocks in a database system may occur with a substantial frequency, even when deadlock prevention methods are used. This may happen since deadlock prevention methods frequently deny requests of transactions that can cause deadlocks. The denial of requests can produce delayed executions of such transactions, thus increasing the probability of occurrence of deadlocks on the execution of transactions. Additionally, deadlock occurrences are aggravated by the ever-increasing demands imposed on database systems since the overall volume of transactions trying to access shared resources is huge.

The simplest way to deal with a deadlock is to kill one of the transactions participating in the deadlock. The selection of the transaction to be killed can be arbitrary, or it may be done based on an arbitrary cost function. Killing a transaction allows the other transactions to proceed, thus breaking the cycle. If the terminated transaction has modified certain objects, these modifications must be undone, and the corresponding objects must be made available to other transactions. The cost associated with rollback is neglected for the moment. Some of the operations can be rolled back. However, the cost associated with killing and restarting a transaction may be huge and may increase with the time the transaction has been waiting for the locks to be released. Hence it may be advantageous to kill the transaction that has been waiting for the lock for the smallest amount of time.

6.1. Wait-Die Scheme

A wait-die scheme is a scheme in which an older transaction may wait for a younger transaction to release a lock or may be killed. Locking or unlocking resources must happen in a certain order to comply with the scheme. If older transactions must wait for younger transactions, the younger transactions may die while they wait for the older transactions. If older transactions are killed for doing something they cannot avoid doing, it brings about a consequence more dire than if they had waited for longer. If properly supervised, young transactions do not die. If sufficient resources are not given to the wait transactions, the chances of starving those transactions increase.

Assuming T1 is the older transaction, T2 is the younger transaction and R(T) is a resource request, W(T) is a wait, D(T) is a die (or roll back) and Alpha is a timestamped ordering of the transactions that determines the wait-die rules, the wait-die rules can be defined as follows: When R(T1) is requested by T1 and T2 requests R(T) (where T1 ≠ T2). If R(T) is held by T1, then:

The condition is W(T2) if Alpha(T1) < Alpha(T2).

The condition is D(T2) if Alpha(T2) < Alpha(T1).

If an older transaction must wait for a longer transaction, the wait-die concurrency scheme will die. The die may be pre-empted if a transaction older than it is not engaged in a read-only action and the current transaction has not worked for a significant period. The request for a wait is simply passed to a transaction manager. The wait-die rules do not actually prevent deadlocks; they minimize and manage the deadlocks. Deadlocks will be resolved by resource pre-emption rather than squashing all the wait-die actions.

6.2. Wound-Wait Scheme

The wound-wait scheme is a method for deadlock resolution of transactions in a database system. The basic idea of the algorithm is like that of the wait-die scheme. The only difference is that the priority of a younger transaction is greater than that of an older transaction in the wait-die scheme. A younger transaction is one that is active later than that of another older transaction. However, the priority of a younger transaction is lower than that of an older transaction in the wound-wait scheme. An older transaction is one that is active at an earlier time than that of another younger transaction. This means that the old-old, directed arrows are exclusive. The concurrently running old-old threads cannot be detected. Because both transactions are waiting for the same lock, there is a path directed from the older transaction to a younger transaction and another path directed from the younger transaction of the same active old-old threads. The transactions wait to lock a resource that is locked by an old transaction. If the transaction waits and is older than the older transaction of the same old-old threads, it is aborted, otherwise, it is allowed to wait. Therefore, when the wait-wait scheme is executed concurrently, the resource will be locked afterward, and the thread will finish its operation. Such threads are completed in the old transactions.

In the wound-wait scheme, the waiting is blocked by the older transaction. Because locked resources exist for older transactions, the younger transaction cannot be executed. In the wait-die scheme, in the direction toward the older transaction, the younger transaction is aborted. The condition is enforced that only younger transactions for a particular lock are aborted. Hence, in such a situation, even its active resource must be aborted to carry out the implementation of deadlock resolution.

6.3. Resource Pre-emption

Resource pre-emption cannot be categorized as either deadlock prevention or deadlock avoidance but is better categorized as deadlock resolution. To be more precise, we can state that resource pre-emption is a practical means of resolving deadlocks after they have been detected. It is also a practical means of avoiding deadlocks in time-sensitive systems, as we will elaborate later. In this section, we first examine resource pre-emption for databases.

A database transaction is a time-consuming series of operations that manipulates the contents of the database, and the database management system allows concurrent execution of thousands of transactions. Transactions are allowed to execute concurrently with the ability to access shared data structures. For performance reasons, the database management system keeps copies of frequently accessed items in main memory. These memories are read and updated by transactions at will. However, if a transaction is suspended during execution and its memory addresses are delinked from main memory, then access to this non-visible memory is prohibited. This poses a significant problem. When a transaction is suspended while accessing memory, it may not be able to be resumed until that transaction obtains all the shared items it had previously manipulated. This raises the possibility of transaction restarts and suspension and enhances the likelihood of a deadlock. If a transaction needs to be pre-empted, then the database management system must destroy or restart it, as the probability of eventually freeing the memory is small. The usual practice is for transactions to request resources but without blocking.

The no-blocking, resource-pre-emption policy is easily adapted to database transactions. Locks may be placed on a database page without blocking, and then transactions may be pre-empted and can restart later. The most common strategy uses most recently written values for page components and forces a transaction restart when it again needs to access a page that has been locked by a predecessor transaction.

7. Best Practices for Transaction Management

Transactions are an essential feature of database systems. In fact, in many cases the only feature that distinguishes a database from a simple file system is transaction management. Thus, it is imperative to use transactions correctly and efficiently. The checkpointing algorithm allows inserting checkpoints into a transaction. However, there is no similar concept for inserting transactions into a database application or system design. In this section, we discuss best practices that a user should strive to follow to achieve these goals.

The ACID properties describe how transactions should behave, rather than how users should implement them. Transaction management is not mandatory in a software system, but when it is used, it should be invested with all the responsibility that it deserves. For instance, the user should enclose only those operations that need atomicity in a transaction. This is mainly to prevent serializing concurrent operations without any real need to do so, thus lowering throughput and response time. Some steps can help identify what needs to be part of an atomic operation. For anything to be logically atomic, the whole operation must not only be affected by the failure or success of the transaction but also by the time duration. A good example is an operation with a high penalty for failure, such as setting up a complex and costly video conference. In this case it would be wise to include setting up the conference in the transaction, as well as everything that would like to be part of the same transaction. Other examples include withdrawal after insufficient funds, and adding to a cache or index after an unordered film.

8. Performance Implications of Concurrency Control

Transactions generally involve some performance overhead. This overhead comes from both the concurrency control mechanisms and the logging structures used to maintain durability. Such overhead is exacerbated in nested transactions. Consider two transactions T1 and T2, where T1 is afraid of failing and T2 has called T1 to log on. Each time T1 does logging within T2 it must globally commit. Such global commits are expensive for large applications.

Yet, using transactions presents some overhead, namely the anticipation of failure. If a transaction depends on non-transactional updates, then that transaction must check those updates or rearrange the execution as to ensure that the transfers it requires have been executed by the third party or at least be somewhat compatible. Moreover, if the other party is non-transactional or if the requested transfer is particularly large, then a conventional non-transactional mechanism must also be used.

This non-transactional overhead must, however, be looked upon as a cost of synchronization; and synchronization may certainly be performed using such classic techniques. Moreover, using such a mechanism is probably preferred if these transactions contain many updates. Transactions are clearly less preferred if they maintain low-throughput, but high-throughput is needed for either efficiency or cost.

9. Case Studies and Real-world Applications

Transactions have deep roots in computer science dating back to the early work on distributed systems. However, it was not until the invention of the relational database that transactions found a widespread application. Transactions quickly became one of the major reasons to use a relational database. Many of the early users of distributed systems pointed out that the distributed system must support an implementation of transactions. More recent work on security, fault tolerance, replication, and distributed data management all focus on transactional models to unify the handling of these different concerns. More recently, some intelligent email systems incorporate messages as transactions, offering the user the guarantee that either all or none of the related messages are available in the inbox. Email transactions are also used for a user’s basket of goods in electronic commerce systems. These baskets are collections of items pertaining to an ongoing transaction used in conjunction with operational protocols, such as the share one, take one, and purchase protocols. The associated protocols are intended to govern the success or failure of a transaction, allowing the transaction to succeed only if all messages are acceptable according to the associated protocol. The worst case occurs when the user fires a total catastrophe, with no user available to respond to an acknowledged catastrophe. In this instance, the baskets may contain pending items, which may be recorded and translated into a cumulative basket transaction; that is, a basket that has accumulated transactions since the last acknowledged basket. After the user responds to the catastrophe by recovering activities, any unacknowledged transaction can be fired to terminate the basket. Whenever a user invokes a transaction, sharing activity relies on the appropriate low-level protocol that dynamically services protocol activity demands and handles all of the basket's changes until the user invokes an acknowledgment of success or otherwise.

10. Future Trends in Transaction Management

Transaction management has been around for several decades, but data management needs continue to evolve. Although many enterprises operate with simple database transactions, many other enterprises manage complex systems containing a variety of resources and services that do not behave like traditional database transactions and for which traditional transaction management solutions are inadequate. Examples of such systems include aerospace, telecommunications, and power generation systems. The challenge in these high performance, frequently real-time environments is to ensure reliability while managing diverse resources with very different behaviour. The basic difficulty is to accommodate different forms of concurrency control that are appropriate for the various services and data objects.

The current vision of traditional database management systems as service providers for use “by the applications” is unlikely to suffer significant change in the near term. After all, it has taken a significant number of years to achieve the current degree of acceptance of these utility-like systems. However, there does seem to be agreement on one interesting point. These general-purpose database management systems are unlikely to serve as the only or even the dominant providers of database services soon. Rather, they are expected to occupy niches within an ever-increasingly diverse marketplace. Given the foreseeable trend for applications to be constructed out of reusable components, it is also understood that some of these components will eventually carry their own transaction control policies. The main question is, how to assess and compare how well these components – and how well these diverse transaction strategies – operate when they interact in the same universe?

11. Conclusion

Considered one of the most important functions of a Database Management System (DBMS) is concurrency control and recovery management. Concurrency control ensures the database remains in a consistent state despite concurrent updates from different inputs. Recovery management protects the integrity of the database against crashes and other types of failure. Using logging and checkpointing during execution of transactions, periodic checkpoints save a snapshot of the current transaction table as well as the current database. For a rollback after a failure at some point during transaction execution, the log must contain information necessary to undo or redo each update made. This section gives a brief introduction to transactions and some of the practical issues related to their execution.

The transaction is a mechanism for describing a sequence of database operations. A transaction can be thought of as a small program that is executed atomically. This means all the operations on the database are either executed or none of them are executed. The transaction is the unit of work in a DBMS and serves two main roles in a DBMS. Transactions define all the actions required for successful completion of a task and transactions preserve the consistency of the database. A single transaction can perform a number of tasks – for example, transfer currency from one account to another account in a Banking DBMS. The various tasks within the transactions might be transferring money from one account by subtracting the amount from the first account and depositing that amount to the second account in the database. While the customer is transferring the money, the database is also maintaining consistency by not allowing other customers to withdraw, deposit or transfer amount in those accounts.

References:

[1] Casanova, Marco Antonio, ed. *The concurrency control problem for database systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981.

[2] Franaszek, Peter, and John T. Robinson. "Limitations of concurrency in transaction processing." *ACM Transactions on Database Systems (TODS)* 10.1 (1985): 1-28.

[3] Bernstein, Philip A., and Nathan Goodman. "Concurrency control in distributed database systems." *ACM Computing Surveys (CSUR)* 13.2 (1981): 185-221.
[4] Barghouti, Naser S., and Gail E. Kaiser. "Concurrency control in advanced database applications." *ACM Computing Surveys (CSUR)* 23.3 (1991): 269-317.
[5] Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Vol. 370. Reading: Addison-wesley, 1987.

Chapter 5: NoSQL Databases: Types and Use Cases

1. Introduction to NoSQL Databases

A NoSQL database is any non-relational database or data source. NoSQL databases differ from traditional relational databases. Relational databases store data in tables with rows and columns. They often enforce rules by requiring certain columns to contain information types [1-3]. For example, a column labelled "Birth date" might restrict entries to the date format. Application developers query data within a relational database using a language called Structured Query Language. SQL is a powerful language for querying and manipulating data, but it has its limits. Most relational databases cannot handle unstructured data types, such as photos, videos, or sound bites. As a result, organizations that need to store large amounts of unstructured data cannot rely exclusively on a traditional relational database.

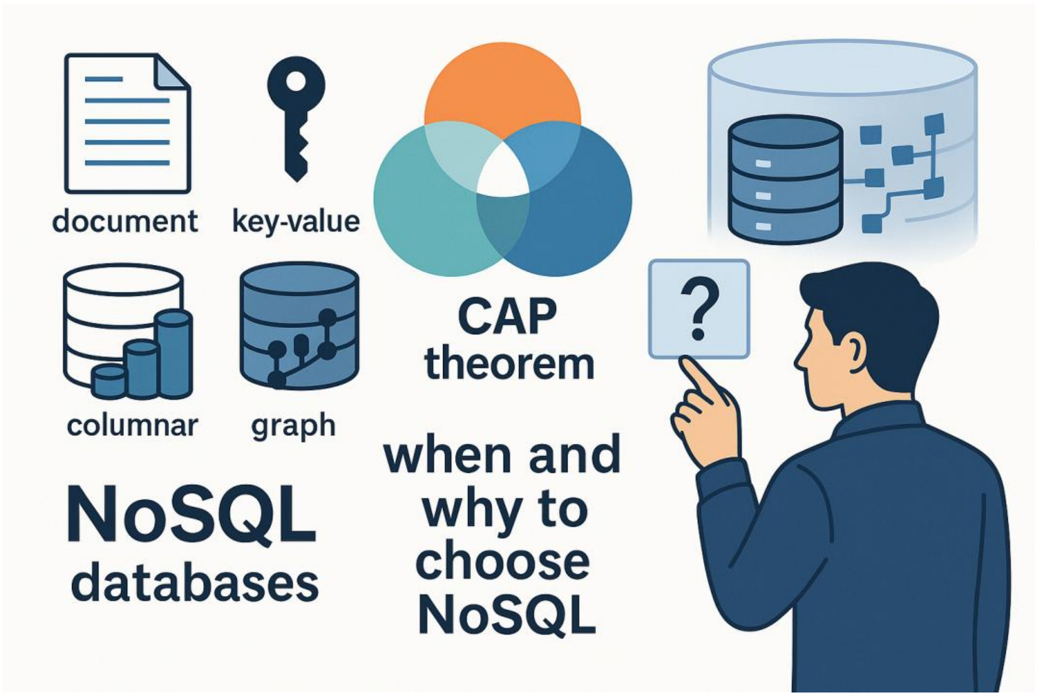
Several new database solutions are designed to overcome the limitations of traditional relational databases. These solutions, known collectively as NoSQL databases, can be divided into four kinds:

1. Key-Value Stores: Key-Value stores provide a mechanism to store large numbers of items, each identified by a unique key. This kind of data store treats any data value as opaque or unstructured and does not interpret the data in any way. Distributed key-value stores have remained popular, and several modern architectures rely on them.

2. Document Stores: Document stores are a form of key-value store in which the stored "value" can be a sophisticated data structure, described as a document. In general, a document is just a collection of fields, each of which is identified by a name.

3. Column Stores: Like document stores, column stores are a kind of key-value store in which the value is a complex data structure. However, data in a column store is arranged into columns rather than documents.

4. Graph Stores: Graph stores hold data as a network of nodes and edges that can represent relationships between arbitrary kinds of objects. Graph data stores are gaining popularity for analyzing complex relationships among various kinds of entities.



2. Types of NoSQL Databases

NoSQL databases or non-relational databases are classified into four major types, Document-oriented, Key-value pair stores, Column-oriented databases, and Graph databases. Each type has a unique architecture designed to solve specific problems. You can choose any type of NoSQL database based on your requirements. Let’s understand how different NoSQL types design their architecture.

2.1. Document Databases

In Document databases, a “document” is the most fundamental unit of data. Documents can be viewed as an extension to the rows in the table-oriented storage. A document database stores data in the document format usually JSON, BSON, or XML. These documents have key-value pairs like data stored in a key-value pair store, but each document can have its own unique structure. This database is best for organizing the data into documents and offer schema flexibility. A classic example where document databases work well is a blog. Each blog post is a document and can consist of a different number of content fields.

The document store is the most widely used NoSQL database type. In the document model, what is usually called a "record" is a so-called document. The core value of the document format is that it allows for an arbitrary and variable schema that is flexible and document-centric, and hence well adapted for most applications. Documents are a flexible and unnormalized data model where instead of having structured tables with a fixed type and schema, which require a new table to be created whenever you want to add new columns of data, it allows records that belong to the same collection to have completely different fields and data structures, with each record having a title that can be any kind of data, typically a string of some predetermined length. Documents are usually stored in collections, and they can be accessed using a key or a query. Although it may seem to be a less structured model, documents can still hold highly structured data. Furthermore, just like with tables, queries can return documents matching a specified query, filtering the current set of documents and reducing or modifying the data relationships returned. In fact, most NoSQL databases implement special query languages to allow for a more natural query. Other NoSQL databases built on top of distributed storage systems expose APIs based on the MapReduce pattern, which allow developers to implement queries in any programming language.

2.2. Key-Value Stores

In Key-Value Data stores, a large amount of data is stored as a collection of attributes. Each attribute is stored as a key along with its value. A Key-value database is one of the fastest NoSQL types. They provide a very simple interface to store and retrieve data in a very efficient way. Both the key and the value can be string-based identifiers. But the value can also take other forms such as a set of values, a list of values, objects, JSON, and many others. They do brilliantly well in applications that require a fast response time. A classic example of these applications is for a recommendation engine for e-commerce stores.

There are efforts to classify NoSQL databases under three or more categories, which could also be another interesting topic or the first part of a chapter. The focus and design concepts showcased by NoSQL systems are diverse; however, for a simplification, we shall group them into the four following categories. Originally, NoSQL systems draw inspiration upon other similar technologies, or they try to cover some of the limitations exhibited by traditional relational systems or even adapt the existing key-value systems. Key-value stores can be seen as an evolution of the hash table idea, providing persistence and distribution properties present in databases. By wanting to maximize the write performance of a storage system at the cost of consistency and flexibility, key-value stores should be subjected to multiple limitations. They usually cannot represent the rich data types of structural schemas, such as those present in a RDBMS. Key-value stores represent simple structs consisting of opaque values for a specific data type, usually blobs containing serialized objects from languages such as JSON, BSON, or XML, while the keys are simple types, usually strings. However, redistribution, the usage of keys, and scale-out capabilities provide key-value stores with the properties sought by most current applications, typically the same that motivated the first wave of NoSQL technology. Key-value stores are often easy to modify, apply easy and physical partitioning strategies, or are built on physical storage techniques.

They usually provide some API with low-level commands to execute at least simple operations, such as fetching or storing an opaque value from or into a specific location, or Writethrough cache-type-looking features, where the set command can also affect a remote database. Simplicity predicates the access design, and key-value stores do not implement more complicated queries, as range queries or joins or even indices on keys, requiring modification and careful key design done by the users or the application developers. Wildcard operations or sequences of keys can typically be found as key access missing functionalities provided by the systems. The first wave of NoSQL technology stimulated key-value stores popularity and the NoSQL proposal, originally embracing technologies.

2.3. Columnar Databases

Data is always stored within some structure. Relational databases store data in datasets known as tables. Each row in the table is a dataset called a tuple, while the columns of the table represent other datasets known as attributes. Columnar databases differ from relational databases in structure as they store data tables by their columns instead of storing them by their rows. Depending on the NoSQL database implementation, data may be primarily stored in disk blocks organized by columns, or it can be stored by rows or tuples, but internally, each column of a table is stored separately. In the latter case, such databases are said to provide columnar indexes or columnar views on the data.

Most columnar databases provide advanced features to quickly access only the requested columns of a dataset, thus improving query execution speed. Such databases make it easier to perform aggregation and analytic functions. Columnar databases are also capable of handling very large datasets.

The data accessed in the columnar databases are big in terms of width. Such types of databases are mainly used in analytical applications and solution space includes applications like business intelligence, data warehousing, reporting, etc. These are just a few examples that help us identify applications in the columnar solution place. Columnar databases have storage optimized for queries that touch just a few columns but almost all rows.

2.4. Graph Databases

The graph database is a relatively new approach that models' data with a graph structure[2,3-4]. In a graph structure, data entities are modelled as nodes, and connections between nodes are modelled as directed/undirected edges. Nodes can have properties, which are applied as key-value pairs and can represent additional data attached to the node. Edges can also have properties but are not usually needed. Data entities that relate are represented in the graph as connections by edges.

Graph databases are like traditional graphing packages used for social network graphs and other use cases. The difference is that they apply the NoSQL paradigm to be queried for search, graph traversal, and other data functions that use the properties and configuration of the graph to optimize the operation. Graph databases use some specialized query languages for traversing the graph as well as APIs for other use cases. But similarities to common programming constructs, for loops that walk edges and nodes, graph databases are especially flexible and efficient for directed-connected data entities and quickly analyzing their relationships without explicitly querying for connections.

Common uses of graph databases include social relationships, understanding fuzzy connections and interpreting sentiment, and other use cases are for recommendation engines. Other types of related use cases that utilize the unique characteristics of graph data collect and present specific classifications of data relationships and their attributes, include hierarchical directories and taxonomies, linking to similarity, concept nodes, topics, and presenting similarity graph patterns.

3. CAP Theorem

Three fundamental and possibly conflicting design properties required in distributed data management systems are consistency, availability and partition-tolerance, known as the CAP theorem. Though CAP has important implications for all distributed computing, it is much more relevant to NoSQL systems because traditional distributed system built on a centralized database architecture focus primarily on consistency. About NoSQL systems their more advanced architectures and distribution requirements make them more tolerant of lower dataset consistency.

The terms consistency and availability are borrowed from the field of distributed computing. For distributed databases two definitions apply. With respect to the database they are two properties of the transactional model, which for distributed systems is different than for centralized systems because a transaction, which is defined as a set of operations that must be executed in an all-or-nothing manner, runs by necessity against replicated state machines, may execute concurrently on several machines, and thus require a protocol for mutual exclusion. Consistency means that when the transaction commits it brings the replicated machines to a new consistent state. For availability, if any transaction execution does not eventually commit it is considered unavailable, which can be, for example, when there is network partitioning. However, with either consistency or availability other operations may not be performed to guarantee up-to-date replicas in agreement with the transaction, for example using locks.

3.1. Understanding Consistency

The definition of what a database is required to do is given by the term consistency. Approximately, a consistency requirement is a set of constraints on the values of the database at a given time. For example, if the database consists of representations of bank accounts and the operations on those accounts are transfers between accounts, then one consistency constraint is that the sum of the values of all the accounts must remain constant. More generally, the set of consistency requirements specifies legal database states and also the changes to the database that are permitted by the operations of the database when the database is in a given state. There are many variations on this theme, depending on the semantics of the operations, the kind of database values, and the applications.

Basic database consistency definitions are captured by the idea of a transaction. A transaction is a sequence of changes to the database. As a matter of definition, the system state just before the first change of the transaction is the state of the database before the transaction. The principle of one-copy serializability states that a sequence of transactions produces a consistent database state only if it is equivalent to a serial sequence in which each transaction is executed completely before the next one begins. In this case, the distribution system acts as a single copy of the database. A distributed database system is one in which the database can be spread across a number of servers. The transaction operations require the system to behave like a single copy of the database. In a distributed database system, failure of a node or a number of nodes requires that the one-copy principle be violated in both time and space.

3.2. Understanding Availability

Barbara Liskov showed that it is possible to develop a practical, secure available, non-blocking distributed system without the need for transactional capabilities. This was the case with the fabulous Vax clusters run by DEC into the early 90s. But the Vax clusters had naive on-demand resource management capabilities, and no notion of scale beyond a handful of nodes. Today, we look to NoSQL DBs to provide some near-optimal mix of all properties in the CAP theorem that is both economically sensible and functionally useful.

As seen in the consistency discussion above, there is a difference between “non-blocking” systems and available systems. NoSQL DB systems tend toward being available but non-blocking systems. Today’s almost-mature NoSQL databases tend toward being highly available but prone to offering partial/non-serializable consistency semantics.

How to interpret availability is still somewhat fuzzy even among the hammer-and-nails, as we will shortly see. Also, how to have your cake and eat it, too, is also a big question. Many NoSQL DB systems have tried to offer availability when you need it by running in an eventually consistent mode by default but providing consistency on demand. This is exactly the tack the community has taken: when you have to have consistency during some times of day, you simply turn on internal DB locking services, and your DB will provide the level of consistency you seek, with little to no business-based regrets.

Regardless of how "availability" is interpreted, NoSQL databases almost always favour replication so that high availability can be achieved. Many NoSQL DBs use an adaptive hybrid bead model for availability. This mixes rigorous single-site availability during normal service periods with multiple-site, replicated, after-the-fact-agreed availability during other, less service-demanding times.

3.3. Understanding Partition Tolerance

In distributed computing, a network may become partially faulty such that some messages being exchanged between nodes can get lost and the senders and receivers are partitioned. More precisely, a partition is said to have occurred for a particular pair of nodes if any messages sent from one node to the other are permanently deleted before they are received. Communication failures such as link failures are examples of network partitions. A network of persistent nodes can work in this partitioned mode only for a limited amount of time because of a lack of reliable broadcast and process clocks. Time becomes your enemy in a distributed system. Clocks may drift apart, and unexpected, independent events may occur in different parts of the system out of order. The nodes can operate in isolation, and failures in one part of the system may affect the correct functioning of other parts due to incompatibility in state stored at the nodes. To ensure correctness, an effectively working distributed system running in partial mode must enforce certain restrictions. Distributed algorithms can be designed for important special cases of independent processing in which:

1. For groups of nodes, a partial mode is made to look like a centralized system by employing a sequence of control tokens in turn mode or timestamping of events to impose a global order.

2. The independent processing of nodes is managed correctly by a proxy at each node utilizing a recently received time-stamped message to ensure the correct order of messages from that node.

3. While in partial mode, the operations at different nodes do not conflict with one another.

3.4. Implications of the CAP Theorem

CAP's original formulation highlights an important trade-off: While C and A must both be provided when P is not an issue, one of the properties can be sacrificed to ensure the other (and P) when P is an issue. CAP's original formulation also mentions the different approaches that are common to distributed systems and different levels of the traffic being handled. Expanding on this trade-off, a system can be either CA (and non-partitioned) or CP or AP.

A CA system can be thought of as being a centralized system that is offering speed and high service as would be expected from a centralized system but does not functionality of a distributed system. It thus postpones Distribution. A CP system will always provide Ca (possibly, with high latency), so it can be visualized as being like a distributed system where the different nodes periodically engage in extensive reconciliation with each other. A CP system will also be utilizing only one of its many nodes when handling accesses at any point in time. It would be reduced to CA if it followed this behaviour all the time. An AP system will simultaneously be catering to either of the two properties, especially when the latter is instanced using unique identifier generators and other such systems for Create actions, UIDs being unique to all entities such that they are eventually consistent.

4. When to Choose NoSQL Databases

When choosing a NoSQL database, it is essential to understand the underlying requirements of your application [1,5]. Choosing a NoSQL database without a clear understanding of your application requirements can lead to unexpected results. In this section, we will review scenarios where NoSQL databases fall short and then delve into some widely accepted guidelines to help make an informed choice. In general, NoSQL databases should be chosen when the database must satisfy one or more of the following conditions.

The first and foremost condition is the scalability requirement of an application. Applications today demand seamless scalability which can keep up with the application and business needs. SQL databases excel in vertical scaling and certain database applications can afford to scale vertically. However, for a majority of the applications and business use cases, the desired scalability level is beyond the capabilities of SQL databases. Applications that have large and frequently fluctuating workloads need to select NoSQL databases due to their distribution capabilities. Businesses that want to prioritize uptime and low latency response times may also need NoSQL databases.

Flexibility and high throughput are other reasons to move away from SQL databases to NoSQL databases. The database must allow rapid changes to capture new application and business domain changes. Other than flexibility, NoSQL databases are known to scale up and out according to application needs. They can be partitioned and replicated across multiple nodes to deliver a higher throughput. If you have an application that needs to achieve a high write throughput, it is most probably a good use case for a NoSQL database. Examples of such applications include content management, social media, Big Data analytics, etc. NoSQL databases have outperformed their SQL counterparts due to the ability of their databases to deal with high write throughput.

4.1. Scalability Requirements

Relational databases run vertically and scale up on a single logical node, meaning that you cannot simply add hardware to the others. Expanding the capacity of the “node” hosting the database is often difficult, sometimes unbearable, and in the end impractical. In the last few years, Storage Area Networks have grown in popularity and have become an in-elastic expense, and disk I/O is the bottleneck in many database applications, particularly in OLTP systems. Elasticity is the name of the game if you wish to operate at Web scale, and it is preferable to add new computing nodes every time you need to gain capacity.

NoSQL databases can afford to scale out on many inexpensive nodes simultaneously, and this is a very appealing feature for their adoption on large applications. Due to the specific treatment of queries and storage models, they can distribute data across many commodity nodes, transparently to the user. The user is only faced with the cost of new nodes when scaling out. Scalability is achieved at hardware expense, and the architecture is within the limits of the commodity hardware available to flatten this expense. Data are usually located in or replicated in many nodes, and data needs to be kept consistent during writes. Database operations are distributed across the many nodes to achieve the needed levels of concurrency. Backends are usually asynchronous and allow for temporary data inconsistency during burst loads but are always synchronous for important transactions. There is often no consistency in data versions following a concurrent write, and this is the price to pay for unprecedented scalability. Although materialized views are often used for real-time operations, NoSQL databases are more suited for serving read requests on data that are not subject to real-time updates.

4.2. Data Structure Flexibility

With the rapid dynamism of modern businesses, the above requirement stated above is more applicable than ever. Employee records, for example, are becoming more complex than before with many additional attributes due to the diversification of employee positions and roles in organizations. Some of the included attributes are employee dependent information such as medical benefits and tax holding status, commission policy attributes for employees working on commission, travel booking system attributes for employees issued credit cards, and so on. These attributes are changing often, and introducing them or changing the meta data for all existing records in traditional RDBMS may require significant overhead. The schema-less, semi-structured, or unstructured databases with their flexible data structures can help businesses solve such issues. More specifically, the support of semi-structured data modelling using schemas that can change or evolve over time per record is an attractive feature of a whole family of databases that support schema less or schema on read capabilities.

Databases offer various approaches to flexibility and expressiveness in terms of their data model and actual representation of the data. Key-value stores and document databases are at the flexible and expressive end of the scale. Data is stored in a format that makes sense for an application, and since applications often use custom-designed data formats, such databases are able to accommodate often very disparate data structures and attribute values. No two data items in a key-value store or document could have the same collection of attributes and types, and the collection of attributes can change from one data item to the next. Another way to put this is that such databases are semi-structured in their attribute model, as opposed to column-family databases, which are hierarchical in their attribute model, to referred to as structured repositories, since items of data are made up of collections of attributes that can have varying hierarchical structures.

4.3. High Throughput Needs

The ability to handle a great number of concurrent requests is an important requirement in many applications. However, the traditional database system architectures have important limitations when it comes to scaling in order to satisfy this need. On the one hand, in the social networks, sharing of user-generated content demands that the database serves a huge amount of read requests for different contents that have a very high temporal locality. On the other hand, services like ad serving require an extremely fast execution of write requests that are often small.

High throughput needs can be addressed by employing a shared-nothing architecture that partitions the database into a number of small sub-databases, each residing on a separate server. Both read and write needs can be executed in parallel, targeting different servers. Analysts have openly spoken about the fact that many NoSQL systems had been designed specifically for certain needs. Among those say that a NoSQL system may scale to hundreds of thousands of updates per second, with data models based on high-speed queries to satisfy multiple user-generated or ad-targeting requests based on location or interests. A popular distributed NoSQL database is a system that provides for horizontal scaling for both storage and processing by employing a shared-nothing master-slave architecture and is used in supporting the low-latency needs of many companies.

4.4. Handling Large Volumes of Data

When you are struggling to fit lots of data into one database server, it may be time to consider a NoSQL database. We have already talked a bit about horizontal scalability in the Scalability Requirements section, but one point that we haven't stressed too much is that NoSQL databases love big, distributed data. A lot of times the reason you struggle to fit everything into one server is just because the data's too big. For example, you are querying up terabyte large blobs of data because that is the size of your logs. Or perhaps you are storing high-resolution user images and every user from every site you own is uploading pictures without any sort of prudent size limits in place. From a more social perspective, perhaps one of your social sites is imploding under the content it generates, and the one million active users are wreaking havoc on the architecture as they all upload all their pictures, all the time. And of course, this situation seems even worse when there are very few government or societal restrictions on the data itself.

A lot of database products support general use, and it would probably take an army of engineers (or a few really talented ones) to build these systems. At the same time, there is still a great need to be able to reliably and quickly consume large amounts of data; archiving that data within the same system; defining high-level, ad-hoc queries; and still enabling cost-effective structures to aggregate large data sets for fast processing. One common demand is the need to deliver and transform results quickly or to provide services for visualization or search, or event triggering services, at a Data Warehouse level but under very tight production constraints. There are unique questions one must ask to find the right product for a solution.

5. Use Cases for NoSQL Databases

In recent years, a wide variety of web-based applications have emerged that require relatively complex database structures yet do not rely on the strict adherence to these structures to operate, as traditional data workflows employed by enterprise transactions demand. Instead, these applications are likely to function well even with a significant amount of missing or irregular data in their queries. Recognizing these nuances in the relationship between applications and the data on which they depend, NoSQL databases were developed to optimize performance and availability for such applications by permitting arbitrary flexibility in the database structure. NoSQL databases bring distinctly different advantages by trading a measure of strictness, reliability, consistency, integrity, and structure, while providing greatly increased scalability and ease of modification. This section describes typical use cases in the context of five applications that cover a high percentage of real-world NoSQL databases in use, including content management systems, real-time analytics, Internet of Things applications, social media platforms, and e-commerce applications. The wide variety of database structure flexibility offered by their schema systems indicates that NoSQL databases are not restricted to any particular application domain. Each of these application areas has different performance characteristics—some are dominated by large numbers of reads; others have high amounts of writes or data ingestion, while others exhibit high variances in traffic volume. Some provide data access in short time periods; others never delete data at all.

5.1. Content Management Systems

NoSQL databases are ideal for content management systems (CMS) because they can easily handle large amounts of unstructured data and allow for fast access to all that data. When NoSQL was introduced in response to the limitations posed by the relational database model, it became easier for developers to build a CMS that would properly organize all types of digital data assets. From images to videos to articles to podcasts, how businesses capture, curate, control, manage, publish, distribute, and share all of their digital data becomes critical to their content marketing strategy and their overall success. Many businesses are said to operate as media companies. But building a CMS using a relational model was extremely tedious. Relational databases were never intended to support the speed and variation of data demands that marketing on the digital landscape mandates. The content that these businesses create, distribute, and share comes in many different formats, including text, photos, 3D, and video. Each plays an integral part in the larger content marketing strategy and throughout the customer journey. Managing all of the content assets that a business needs to carry out its marketing initiatives can be, and often is, challenging. Many companies enlist the help of a third-party vendor to manage that process for them. Other marketers use a CMS from a third-party vendor to store images or videos or use multiple CMS. Using a CMS created by a vendor means that all of a business's content is secure and protected while also being accessible. The benefit of a CMS for a business is that they gain the ability to collect all of their content into one central location, make it searchable, and throughout their entire organization. All employees can use the CMS for their marketing needs and knowing how to use it to aid those processes becomes critical.

5.2. Real-Time Analytics

Databases that support real-time analytics must ingest streamed or batched data on a daily or hourly basis and provide management and access through a standardized interface. In addition to the task of data ingestion, these databases also have to build summary tables that support efficient queries and ensure that the results are current even while new data is entering the system. The key design point for these databases is that they must achieve high read rates on published data while still keeping the data current. This task is greatly simplified if the volume of changes to the underlying tables is low enough that the current results can be kept in memory, thereby avoiding disk accesses.

That explains why some companies find their real-time business dashboards updated at least every few seconds with fresh data, while others choose to provide similar analytical views of the business every one or two hours because the overhead of continually updating their databases is too great. Since ad hoc analysis is typically run infrequently and can be delayed until a specific time of day, latency is also less of a concern for these data warehouses than for traditional real-time dashboards. NoSQL databases also support on-the-fly schema changes that allow business analysts to quickly modify existing queries if they discover new insights while analysing their schemas.

Furthermore, real-time analytics is not limited to traditional relational data warehouses. NoSQL databases can support interactive user experiences on web properties with very high traffic levels. Personalization calculates user preferences to the extent that an individual visitor to the e-commerce site sees ads that are likely to be of interest.

5.3. Internet of Things (IoT) Applications

A rapidly growing number of devices, such as sensors, RFID chips, mobile phones, smart meters, smart home appliances, and numerous other embedded systems, are capable of collecting, storing, processing, and transmitting data to other smart devices in their environment and across the Internet. This technological trend, commonly referred to as the Internet of Things (IoT), involves joining the physical world and the virtual world through the ubiquitous Internet and addresses applications from smart cities, smart grids, and smart buildings to autonomous vehicles, precision agriculture, environmental monitoring, and health monitoring. The diverse IoT apps generate massive data streams, from hundreds of gigabytes to petabytes of data per day.

Traditional SQL databases were primarily developed to manage structured data, where both the data model and data-processing operations are defined in advance. Such data management systems are incapable of managing the diverse unstructured and semi structured sensor data streams, characterized by high-rate continuous data generation, data volatility, data complexity, and diversity of the corresponding data models of all the devices connected to the IoT. This type of data management belongs to the experimental field of NoSQL databases, which have a flexible data model and horizontal scalability, fault tolerance, and high availability necessary for IoT applications.

Distributed NoSQL databases have been developed to appropriately manage the data-centric architecture of IoT applications. These types of NoSQL databases are managed across large geographical regions, at multiple logical levels, providing high availability and scalability demanded by data streams of different granularities. These databases monitor the condition of smart blocks, installed in each of the networked IoT devices, and then store, retrieve, and query their health status.

5.4. Social Media Platforms

Social media platforms are one of the largest data producers. Each user produces unstructured data on a day-to-day basis, in the form of photos, videos, status updates, comments, and interviews. Users can also produce structured data such as profiles. Due to the sheer volume of data produced, and the level of user interactions, social media platforms have become experts in NoSQL technologies. NoSQL technologies wrap achieving high availability and achieving low latency for high-velocity web applications, into a computer science model for the engineering process. Other industry leaders such as content delivery services, or online marketplaces, also have similar requirements to social media platforms. The scale is generally less than what social media platforms experience, but there are many online destinations that have multiple billion dollar fiscal sales. For such destinations, optimizing for speed is crucial as there is a loss of revenue for each millisecond of lag in the user experience. Online video platforms also share similar requirements for low latency, and high availability.

Due to the dynamic nature of the data, users will also delete the object they created in real-time. Failure of a primary node will not halt read and write access to the active data through a secondary node, but can cause delays until all data has been transferred back to the primary node or until a new primary node is established. As a result, NoSQL stores with distributed architecture, and partition tolerance, are attractive to the social media business model. Data is constantly being reeled to update the view for all users. Data is also added in huge bursts. The short history of the data makes it actually beneficial to remove the lag effects, by pushing the data back to the active data view, right after each update.

5.5. E-Commerce Applications

E-commerce data workloads consist of user events and user generated content such as product reviews. The number of user events is very large compared to the user generated content store size; they generate high velocity time series and usually require event store technologies capable to scale horizontally such as Wide Column and Document type NoSQL databases.

Another important aspect in e-commerce applications, such as web sites or mobile applications, is the high traffic during festive seasons, such as Christmas or Black Friday dates. The main user generated data in relation to e-commerce are product catalogs and user generated content such as product reviews and comments. The volume in relation to the number of product items is comparatively small, but consumes a lot of read operations. These workloads are typically handled by data stores specific to analytical workloads – key-value or document type NoSQL databases that must be scalable and have low latency to answer a high number of users read and write requests.

6. Challenges and Considerations

Although NoSQL remains a great alternative to relational databases when it comes to storing large scale unstructured or semi-structured data, this does not mean that NoSQL solutions do not come with their own challenges and considerations. It is also important to note that not all applications are suitable. Certain traditional applications such as online transaction processing and legacy applications highly rely on relationships among data that NoSQL databases are not able to support at scale.

6.1. Data Consistency Challenges

NoSQL technologies set out to challenge the traditional ACID-based designs to meet the expectations of performance and scalability, while primarily supporting business-critical applications that require a limited set of the operational properties. Distributed transactions can be expensive to perform, leading to a somewhat limited command set and a possible absence of strong data guarantee properties, typically implemented as part of multi-document maintenance transactions: isolation, consistency, and related principles such as concurrency, fixpoint, and least commitment. In practice, such automated data coherence services do not scale, leading to the partition-tolerance theorem suggesting that cost-aware designers should trade one of the three pillars (consistency, availability, or partition tolerance) from the required cost functions supported by the specific use case. Therefore, it is paramount to understand NoSQL properties and limitations to trade the legal level of consistency with the availability, complemented by design approaches and strategies such as sharding, partition tolerance, record replication, denormalization, and finally temporal tracking. ACID properties lead to earlier use of strict data models that minimize attempts at breaking them to enhance throughput, while CAP principles directly impact the design of applications that rely on NoSQL paradigms. Notwithstanding, techniques have been defined to mitigate the challenges of eventual consistency and minimize the overhead in data management.

6.2. Query Complexity

NoSQL databases are usually combined in a distributed manner, where the queries span across multiple NoSQL systems. The complexity of such queries may be higher than that of the SQL equivalent; particularly if the systems are heterogeneous, whose schemas differ from each other. However, the analysis of the query models of each NoSQL system proves that various models support queries of complexities comparable or even equivalent to simple SQL queries. The tuple-oriented model of these systems allows for the evaluation of join operations on key equivalences, associative operations, and other predictable join conditions. Similarly, the property-oriented model has the same tuple-oriented supports as the others, especially for NoSQL systems based on a relational tabular structure.

However, these analyses do not justify the actual inefficiency of NoSQL systems in answering complex or disk-based queries. The NoSQL systems may perform such queries more slowly than other systems specially designed to achieve such operations efficiently. It is also important to note that for a typical NoSQL system, batch operations are much more effective than interactive ones, which usually return only a small amount of the processed data. In addition, the existence of a multitude of NoSQL systems leads to the need to deal with the issues of interoperability and compatibility every time the application requires a non-trivial query, for example, making relational joins between tabular NoSQL systems and key-based associative queries. Indeed, this issue could grow in the future when because new kinds of NoSQL systems appear with increasing frequency, often having complex query models.

6.3. Data Migration Issues

As previously discussed, one of the major challenges that data architecture designers face with multi-database systems is how to keep data across databases synchronized. There is much more effort and cost in keeping data synchronized than simply moving it for a particular query and then discarding it. Many commercial database systems have data synchronization mechanisms that allow data to move between disparate database types or database vendors. As with the multi-database approach, there are inherent complications with using these available data synchronization methods, including complexity of configuration and operation; monitoring for errors; performance overhead; and transaction integrity issues.

Many companies having databases from multiple vendors or system types use data synchronization only for high-demand business functions. They regularly create bulk data synchronization runs followed by a continual synch for data fields; then, at some quieter times, do a complete data dump that is imported into the Reporting Database, Warehouse Database or Archive Database. These methods present issues of content integrity and currency for business functions with near-real-time needs. Companies that have bought multiple-sized copies of database servers from a vendor use a third-party bulk data migration utility to perform backups on the primary database that are imported into the secondary databases. However, issues of content integrity and currency do arise. Until recently, some companies still used cobbled-together scripts to perform a manual bulk database synchronization.

7. Conclusion

Apart from column, document, wide column, key/value pairs, graph databases, and object store databases present themselves as a heterogeneous mass of structures that define and support the NoSQL world. Whether new databases will need to be created to fill a niche or existing databases will need to be generalized remains to be seen. Most likely, the adoption of NewSQL and its relatives by the world of academia for the teaching of databases will encourage a new generation of pioneer researchers in need of new ideas.

The rise of databases in the cloud, which by force of necessity cannot be SQL databases, has increased interest in NoSQL databases. NoSQL databases in the cloud may become important research topics. What type of cloud-specific data sharding allows the databases to execute performant and efficient queries?

Methods for building mission-critical applications using SQL plus NoSQL databases in the cloud may also become an interesting area of research. Much of the action of NoSQL databases for the foreseeable future will occur outside of the traditional areas of the database. Rather, NoSQL databases will reside as components of larger architectures, including those for applications.

NoSQL: a misnomer? Not at all. It is a convenient name that denies the homogeneous universal structure, available for any operation of the whole data management system described by the DBAs for the first generation of DBMS. It gathers under one definition a family of components of a heterogeneous system for a joint purpose: manage many heterogeneous kinds of data that share only a high-level semantics, that eventually have relationships. The first part of this purpose is fulfilled by diverse heterogeneous systems for managing various data models that have appeared during these years. The second part of the purpose is currently developed by the SQL model. For this purpose, we will describe in the following chapters components of this broader system with greater care.

References:

[1] Han, Jing, et al. "Survey on NoSQL database." *2011 6th international conference on pervasive computing and applications*. IEEE, 2011.

[2] Stonebraker, Michael. "SQL databases v. NoSQL databases." *Communications of the ACM* 53.4 (2010): 10-11.

[3] Stonebraker, Michael. "SQL databases v. NoSQL databases." *Communications of the ACM* 53.4 (2010): 10-11.

[4] Okman, Lior, et al. "Security issues in nosql databases." *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2011.

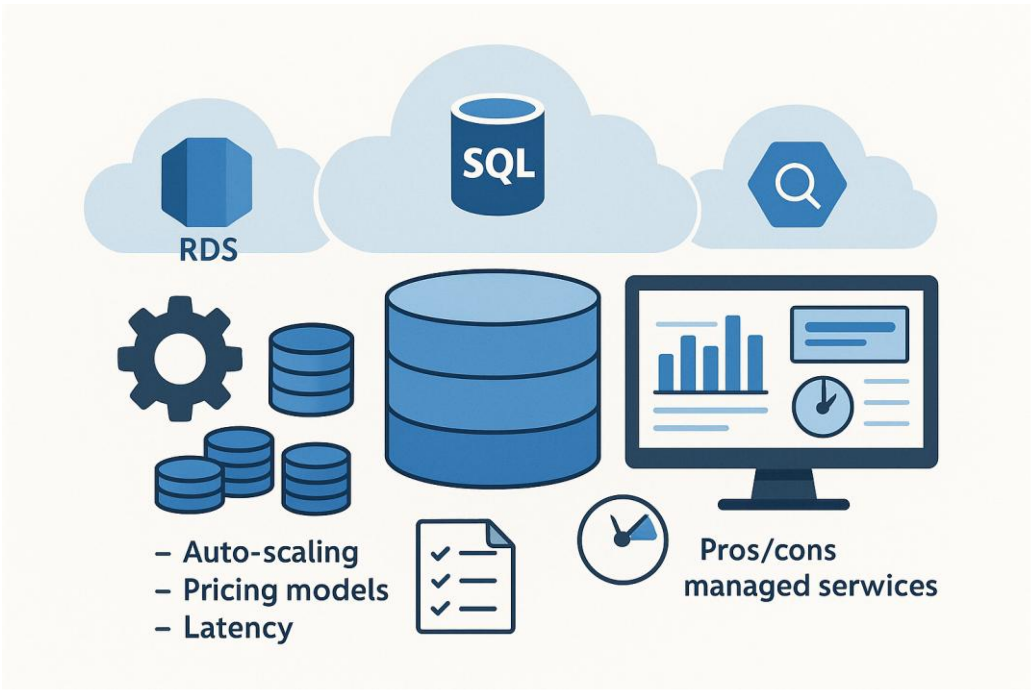
[5] Li, Yishan, and Sathiamoorthy Manoharan. "A performance comparison of SQL and NoSQL databases." *2013 IEEE Pacific Rim conference on communications, computers and signal processing (PACRIM)*. IEEE, 2013.

Chapter 6: Cloud Databases and Serverless Data Platforms

1. Introduction to Cloud Databases

Cloud computing has transformed data management and storage by providing on-demand services over the Internet. Over the past years, several types of services have emerged in the cloud computing world, mainly cloud storage, virtual machines, and cloud databases. The first two services have been widely adopted by users, but cloud databases are just beginning to be fully utilized. These more trusted and secured data management infrastructures are key to achieving the full potential of the cloud. Nonetheless, while important, data management is still just one (although critical) service provided by the cloud. The first two cloud services mentioned above have a user base that is several orders of magnitude larger than cloud databases. The letting of terabytes of data remain offline, often unchanging for long periods, represents a business opportunity yet to explore. Cloud Storage is also in a position to dislocate some of the smaller Storage Area Networks, now usually used for shared access to storage, but which command high maintenance costs.

This chapter intends to provide an overview of cloud databases, from definitions and expected features to comparisons between traditional data management technologies on-premises and in the cloud. Topics explored include service architecture, usage and performance considerations, billing issues for developers and businesses, and a brief comparative analysis to existing on-premise systems. The highlights presented indicate that, despite some severe limitations, the cloud service undergo continuous improvement, may provide an alternative worth considering for many developers and companies.



2. Overview of Serverless Data Platforms

The value of Cloud Data Platforms lies primarily in the value of Data and the Data Processing pipelines constituting Cloud Data Engineering. Cloud Data Engineering is important but primarily linear work that is highly dependent on domain knowledge, and thus somewhat tedious, but which if performed well, enables Cloud Data Platforms to deliver actionable insights for the purpose of Digital Transformation. Cloud Data Platforms offer this value at scale because of their foundational element: at scale Data Pipelines – Data Ingestion, Data Quality, Data Preparation, Data Transformation and Data Availability, Function as a Service Products.

Serverless Data Platforms, on the other hand, address the bane of capability for all but the large organizations with specialized Data Engineering Teams, by making Easy to Use, No Code, Low Code Data Pipelines that allow organizations of all types and sizes to create value from Data Processing. The Serverless Data Pipeline enables on-demand and just-in-time utilization of underlying infrastructure resources, without concerning the user with resource management.

Serverless Data Platforms cater to the organization where the Data Pipelines that on-demand process the Data are not mission critical to the organization's core business, but important enough and have enough of a frequency of processing for Data Exhaust to justify having User Managed Data Pipelines or Bare Metal Data Pipelines.

The Serverless Data Platform with user managed Data Pipelines is among the simplest and most cost-effective means to organize and deliver Data as a Service. Delivering Data as a Service from a Cloud Data Warehouse is a complex effort requiring Enterprise Data Warehouse skill and expertise, which are not easily available, or because of rarity or because of cost. Serverless Data Platforms allow you to create User Managed Data Pipelines from External Service part of the Data as a Service System. Data Exhaust from Digital Interactions winds up Five or More times in External Systems than the Data the organization owns, Controlled Flows Model.

3. Amazon RDS

In 2009 Amazon unveiled the Relational Database Service (RDS) to run relational databases on Amazon Elastic Compute Cloud (EC2) servers. RDS is a service enabling easy and cost-effective relational database provision, operation, and scaling. Amazon RDS provides the following functions: it is easier to deploy and configure replicated MySQL, Microsoft SQL Server, PostgreSQL, or Oracle databases; it takes care of health-checking, failover, and the replacement of dead primary or replica nodes; it takes care of backups and point-in-time recovery; it automates operating system patching and the patching of the database engine, including security patching; it monitors performance and provides recommendations for improvements; it supports read-only replicas in local and live-remote geographic regions; it manages resource access control; it provides database parameters management for tuning; it enables storage scaling on the fly, and it enables the computing capabilities of the running instances to increase.

3.1. Features of Amazon RDS

Amazon RDS is an easy-to-use service for managing, scaling and automating database instances in the cloud. Relationships and data types are pre-defined in a standard schema and new records are identified and accessed by a primary unique index. Each record in a relational database is part of a table, which, together with constraints on data types and values, define the relational schema, and hierarchical relations between tables support structured queries or joins. Amazon

RDS is designed to host a relational database. Hence, it is essential to give a brief overview of the principles of the relational model and the language with which a relational database is controlled, SQL. Amazon RDS can be used to provide support for an operational workload by hosting an operational database or to host an analytical database that is optimized to support analytical workloads. Amazon RDS also supports database engines that can be used to host both types of workloads. In this section, we focus on operational workloads, software requirements, and features of operational database systems that make Amazon RDS an attractive solution.

Every operational database has a software component, called a database engine, that manages the transactions and queries executed against the database. The database engine accepts transactions and queries in the form of SQL commands, guarantees that they finished executing in ACID-compliant fashion, retrieves the requested data residing in the file system and returns it to the application server, and updates the file system after receiving an update command. The database engine and features provide attributes to the operational databases hosted on behalf of clients. Operational database engines come with different requirements and features, and Amazon RDS supports the following engines: Amazon Aurora, MariaDB, Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, and Amazon RDS Custom for Microsoft SQL Server and Oracle database. In the following of the section, we describe some of the features of Amazon RDS.

3.2. Pricing Models for Amazon RDS

Amazon RDS offers hourly price rates for the database and storage requirements, and these vary according to the database engine, database class, and region. Users can scale up a database to a more powerful instance, but they need to specify the database instance type for a minimum period of time. When a user creates an Amazon RDS database instance, the user selects both the database class and the amount of provisioned storage to associate with the database instance. Amazon RDS provides pay-as-you-go pricing that enables users to terminate database instances when they are not in use.

Under these pricing models, users are charged for storage, database instances, and backups, as well as provisioned IOPS for database workloads that require additional IOPS. Prices are based on how much data the user transfers in and out. An Amazon RDS database instance requires compute capacity to function. The amount of capacity depends on the instance class. AWS provides a variety of instance classes that enable users to select their ideal compute resources, balancing the level of service needed against the cost.

Users must select an instance class for the intended workload and manage the underlying infrastructure. Amazon RDS provides multiple instance classes with multiple sizes to choose from. The RDS Reserved DB Instance option allows users to reserve a DB Instance in a specific DB Engine for a one- or three-year term and provides a significant discount compared to the hourly cost of On-Demand DB Instance usage. Reserved DB Instances are recommended for production workloads that require a predictable level of database capacity. Unused Reserved DB Instances count toward if the user has made a significant commitment to long-term committed use.

3.3. Pros and Cons of Amazon RDS

Amazon RDS can simplify your life and save you time if you do not need to focus that much on data and database. It is a well-designed, reliable service, built onto a very enviable base. While working on this service since many years, Amazon has found proven solutions for security, scalability, and availability problems that usually occur in databases. You get the whole nine yards: CRUD APIs, replication, snapshots, backup/restore, patch management, instance monitoring, multi-DC failover HA, performance management, parameter tuning, storage expansion, connection pooling, data migration, and object storage, all bundled, and in one flavor or another. These established best practices are difficult and time-consuming to reproduce in an in-house environment. Amazon allows you to leverage them in an AWS-managed service.

You give up some level of customization and some flexibility of the underlying software. You will be forced to use the Amazon feature set. Additionally, you lose the ability to use your tools of choice especially in terms of operational management. With greater ease of use comes greater operational expenditure. This is especially true for small or medium-sized databases: once they hit a certain size threshold, self-management obviously starts to become more economical than the monthly fees for such utilities. Amazon will upcharge you for the power of these utilities.

4. Azure SQL Database

Azure SQL Database is Microsoft's flagship managed database service. It supports a subset of the feature set available in on-premises SQL Server products, in a bundle that continues to grow as Microsoft manages the database for customers. Customers do not have to install, patch, or manage the database or the underlying operating system. Azure SQL Database version 16.0.1072.1 is available in all 54 Azure data centres, and features hub/spoke geo-sharding in four regions. Azure SQL Database access uses publicly routed and encrypted traffic. Automatic recovery, instant failover, and port protection for DDoS prevention create resiliency. With Premium tiers, geo-region production can be delivered with very short 5-minute RPOs and RTOs in under 1 target objective. Latency for Microsoft Azure is comparable to that of Oracle and AWS, and below that of GCP.

4.1. Features of Azure SQL

Azure SQL Database is one of the widely used cloud implementations of SQL Server. It is designed for cloud and to take advantage of the cloud features such as scale and high-availability, in-built. The entire Azure SQL is a family of SQL cloud offerings which include Database, SQL Managed Instance, and SQL Server on Azure VM. Users can deploy any of these depending on their compatibility concern, focus on development and operations, with other cloud features such as price/benefit etc. Azure SQL is based on the latest version SQL Server and is released as a new version in sync with the SQL Server new release. Azure SQL provides additional cloud-related features such as Geo-Replicas, Database Copy and Long-Running Operations which are not available in the on-premises offerings. Users can scale their solution in response to the demand using a simple command.

Azure SQL provides features such as Query Store which can help analyse the query performance over time and give Autotuning recommendations. They are like the on-premises features such as Performance Schema and SQL Query Store. It has integration with Azure Machine Learning platform to allow the users to deploy and operationalize their experiments. Using the easy interface, a user can deploy his/her model created in Azure Machine Learning onto SQL Azure and call it using T-SQL in the stored procedure. It has embedded features to support the databases with temporal data. It encrypts the data at rest when you enable TDE. A new service called SQL Database Auditing can help retain all the auditing events in an audit log file or send to Azure Storage Account. The new support for Azure Active Directory Seamless Authentication allows single sign-on experience for users with their domain credentials within their corporate networks or from the internet.

4.2. Pricing Models for Azure SQL

Pricing is one of the most important and debated topics in the cloud world. Without a clear understanding of pricing models, it is impossible to answer the question of whether the cloud delivers cost savings. Because Azure SQL represents quite a wide range of services and capabilities, its pricing intricacies may differ depending on whether a customer is using Managed Instances, databases on DB As or Hyperscale tiers, shortened SQL costs on Hyper-V or Virtual Machine systems, or using a serverless (paused) edition of an Hyperscale-triggered Database. Nevertheless, in all these cases, Azure SQL costs are driven by underlying resource provisioning. The sizing of these resources and operations’ intrinsic characteristics, including input/output characteristics, prediction, and predictability, are vital for the relative and real determination of Azure SQL costs.

Overall, Azure SQL Database abstracts away the underlying hardware and virtualization mechanisms from users. Pricing is based on logical databases sized by DTU or vCore limits intervals. DTUs are a bundled unit of measure that includes CPU, memory, and read and write rates. Actual DTU values are equal to 5 DTUs but are usually between 5 and 2,000. There are two options of the DTU-based model available. The first is called Basic and is limited to 2 GB of memory for individual databases, as you need a small number of concurrent sessions and transactions, primarily to manage small lookup tables for back-office applications. The second option, Standard, supports a maximum database size of 1,000 GB with 3,000 concurrent compute sessions and 30 transactions per second or even 40 TPS on the fastest 2,000-DTU servers or super-speed government servers.

4.3. Pros and Cons of Azure SQL

Azure SQL's biggest, unexpected feature is just how far into the branch of full-featured relational database you need to go to run into missing features that depend on the branch of databases for which Azure SQL is optimized: serving applications that have little or no load temporarily and can afford to be restarted frequently. By moving the products more along the axis of integration with the hosting environment and away from being full-featured centralized relational databases, Azure SQL becomes far cheaper for hosting applications than traditional databases. However, if you're running a mission-critical application that directly depends on the speed and reliability of relational database, you'd be better off moving towards the fully-featured branch.

To be sure, in many cases, "cheaper" is an angel's whisper in your ear about which devotee to use. Certainly, the minor extra costs of running a full-featured centralizing resource make traditional ready-to-run database packages in the exact location for which minimal latency is required a little hard to justify. So if you probably don't need any of the more sophisticated features of a mainstream commercial database but want some of those features because they're what your tech group is used to, Azure SQL might be worth checking out.

5. Google BigQuery

Google BigQuery is a serverless, highly scalable, and cost-effective multi-cloud data warehouse. Using BigQuery, you can execute highly interactive, ad hoc queries of huge amounts of data in seconds and perform SQL-like queries on large, petabyte-scale datasets in near real time. If you have large datasets that are updated on a regular basis and you need to be able to query this data for reports, business intelligence, or analytics, BigQuery may be an excellent solution. BigQuery's storage and compute infrastructure is massively parallel, and separating the query service from the data storage allows for fast query performance while being economical. In addition, because BigQuery is serverless, managers of all backgrounds can leverage the power of data analytics without paying big dollars for complex infrastructure.

Data can be loaded into BigQuery from CSV or JSON files in Cloud Storage, or by streaming from logs using the streaming API. BigQuery supports a simplified SQL syntax based on the standard dialect but lacks support for transactions, subqueries, and stored procedures in the current version. Query results can be returned in a variety of formats, including CSV and JSON; they can even be written in Avro format for the purpose of enabling heavy processing of BigQuery output into another service. In the Data Transfer Service, BigQuery can also automatically load data from other Google services, and there are hooks to automatically transfer data from external sources.

Besides the variety of external source support, BigQuery also features an Audit Log that keeps track of all queries running through your project. You can then set a quota for the logs into which queries are written to avoid excessive costs or just track your costs manually. This is extremely attractive if you are running a lot of ad hoc queries. The pricing model is generally dependent upon the amount of data queried, but there is also a flat fee pricing model available if desired.

5.1. Features of Google BigQuery

BigQuery is Google’s fully managed data warehouse solution. BigQuery is designed for large-scale analytics and large-query data sets, up to petabytes and beyond. Instead of using a database paradigm of representing relationships with tables, BigQuery uses a different paradigm that takes advantage of things common to analytical workloads: compute-heavy processing of denormalized tables. Normalization is not used to improve the storage efficiency or bandwidth efficiency of accessing the tuples in the joint result since analytical queries typically run with such low frequency. Denormalization of tables has a negative effect on transactional DBMS workloads that update or insert information with high frequency. However, that is not a consideration with data warehouses; their content is read only for long periods of time, and then they are refreshed by uploading a complete new version in one bulk operation.

BigQuery uses advanced techniques from organized data processing and exascale systems for reading data while executing a query. MPP technology with extensive use of disaggregated storage and cloud storage is the basis for this execution capability. The use of disaggregated storage allows BigQuery to process a large number of queries concurrently while paying only for data storage costs for each of the small number of multi-terabyte tables. The object storage system used to hold the tables has the benefit of low price, while MPP execution of queries using large numbers of nodes allows the platform to achieve a very low query latency for SQL queries. To provide a familiar interface, BigQuery adopts a subset of SQL, which is the dominant single-node DBMS query language. Also, to improve performance for certain classes of queries, BigQuery can use a columnar compressed representation created by querying an external data source using a SQL statement.

5.2. Pricing Models for Google BigQuery

Google BigQuery enables users to query data in BigQuery itself and creates, manages, and automates BigQuery resources (data sets, tables, jobs, etc.) for users. Generally, there are two relevant services, the BigQuery data query service and the BigQuery data management service, and they can be charged on different pricing models.

The BigQuery data query service provides an interactive and a batch query mode alternatively. In the interactive query mode, users can submit their SQL queries to the service for immediate results. It is priced on a pay-only-when-you-use basis. Users are charged based on the number of bytes processed by each query. Queries that reference external tables do not incur charges based on the bytes processed by the query.

For the batch query mode, users can submit SQL queries in a batch mode, in a manner similar to the Hadoop MapReduce approach, as job requests to the service, resulting in jobs, named query jobs. Running BigQuery query jobs in batch mode generates lower latency, in many cases, than using Hadoop

MapReduce to perform the same task and BigQuery accomplishes it with no provisioning required from the client. On this pricing model, users are charged a flat fee based on the number of bytes processed for each job. The processing fee is established on a monthly basis. Given the potential efficiency and speedup achieved by using BigQuery for batch query jobs, it could be much cheaper than using fully-managed Hadoop clusters.

5.3. Pros and Cons of Google BigQuery

While Google BigQuery has a lot of advantages, it is not the best alternative for everyone. Some of its advantages are: • Almost limitless storage, speed, and accessibility. Google BigQuery uses an unusual architecture that consists of two different engines at its base that rely on a shared data repository. The data repository is based on a technology designed to work on a global scale. It breaks down records into smaller "chunks" and can add redundancy to prevent data loss. Once inserted into that fast repository, structured and unstructured data can be retrieved and analysed by the engine in seconds, or minutes at worst. The engine can process massive parallel queries through many servers at the same time. That speed is hard to match. Implementing it is also very easy. All you must do is create a dataset, load your data into storage, launch a "load job," and you will be able to quickly access and manage it through SQL commands. From then on, the system will deal with all the throttling or maintenance concerns you must deal with for classical database management systems. • Advanced features and solid performance. Google BigQuery allows you to use Data Definition Language (DDL) to create or modify databases and tables. In addition, experiences with the previous version of BigQuery also saw that it could deliver solid performance. Some of the cool features that make Google BigQuery attractive are:

a) The possibility of using regular expressions for queries.

b) Very flexible input format;

c) The possibility of querying data through OAuth and Integrated Query.

d) Enhanced load and export techniques.

e) Byte serving for replies and exports; and

f) The possibility of parallelizing queries. While Google BigQuery has many advantages, it is also important to consider its disadvantages.

Among the most noteworthy are:

a) The lack of support for specific database features or engines.

b) The lack of direct support for certain data sources;

c) Lacking some visual monitoring tools; d) The lack of certain SQL features; e) The cost of some integration features; and f) Evolving gui tools.

6. Auto-scaling in Cloud Databases

In cloud database systems, auto-scaling refers to the capability of the system to automatically (de)provision computational and storage resources in response to workload variations [1-3]. It is one of the major features that distinguishes a cloud database from traditional database solutions and can significantly reduce the operational complexity of the database. For instance, maintenance tasks such as capacity estimation for peak-workload periods are handled by the auto-scaling mechanism of the cloud database in a fully automated manner, while with traditional database systems the user needs to plan such resources and actions for themselves. Due to the reduced operational overhead and increased flexibility to cost-effectiveness, auto-scaling is one of the most highlighted features of serverless database solutions. Although some traditional databases may offer an auto-scaling capability, in this chapter we mainly focus on cloud databases with a dedicated infrastructure, outlining how auto-scaling mechanisms work internally for these systems.

Developing an effective, fully automated, performance-controlled auto-scaling mechanism for a cloud database is challenging. For instance, in a cloud database system with an auto-scaling capability, it is desirable to quickly respond to workload variations so that the system performance is controlled, while the reaction times of the auto-scaling mechanism should not introduce overhead on its own. In addition, scaling actions that can be triggered in a cloud database system may concern different system resources. For instance, depending on the cloud database architecture, a system may trigger scaling actions to adjust the number of computing nodes or storage nodes provisioned.

6.1. Mechanisms of Auto-scaling

Auto-scaling platform services to accommodate fluctuations of data operations is one of the most popular features in Cloud Databases. The first Cloud Database services did not include auto-scaling. Unlike managed NoSQL Database, which delivered the business novelty of managed NoSQL for Big Data, the business novelty of Cloud Databases, and more precisely of the first cloud SQL services, was the migration of SQL Databases to the Cloud as a standard service. With increasing demand for services Cloud Data Providers launched more Cloud Database services that took care of any limitations in terms of performance or convenience.

How does auto-scaling work? In principle users need not take care of it. The service supports some SLAs describing performance expectations that will be guaranteed. The service provider monitors the current requests submitted to the service and its estimated maximum available capacity, keeps track of the expected request intensities and increases the workspace to allow an optimal response to an increased workload. The activity may be initiated by the user, by temporarily increasing capacity usage over an expected workload. Then some rules are activated, and the service monitors the conditions for applying them, and decides what action to take, when to take it and what resources to provision. These may include adding instances executing the service or taking some of them off too. The entire decision and reaction process may take minutes or hours. In some service providers SLAs guarantee upper and lower thresholds, estimated delays for the increase and decrease of the system usage level, and the estimated number of resources provisioned in response to the service adaptation.

6.2. Benefits of Auto-scaling

Modern serverless cloud data platforms provide auto-scaling for their processing components which manage SQL and/or NoSQL workloads. They can also provide auto-scaling for background operations such as streaming ingestion, extraction, and data movement. Digital businesses increasingly depend on high-volume workloads, driven by data-in-motion for application users or subsystems, which push data platforms to capacity, causing a degradation in service proportional to the number of data transactions being processed. Platform near-constant data activity often occurs, as do periods of sudden activity or absence of activity. Degradation can be extreme, and result in lost transactional consistency, messages, or latency SLAs. Organizations may then require a re-architecture of the data subsystem, or a redesign of how data databases are accessed.

Auto-scaling is a form of automatic resource allocation for public-cloud resources, meaning workload demands trigger the allocation/deduplication of compute resources being shared among customer workloads. In other public-cloud usage areas, such as virtual compute, load balancers determine capacity needs vertically by tier — “what cloud resources are needed to support the current demand?”. Auto-scaling for data platforms also operates horizontally — “how much extra compute do I need to support a surge in demand”, adding servers to balance a number of concurrent transactions — but also auto-scaling operates vertically, scaling single servers both up and down, based on current activity demands. Auto-scaling for databases and other data services is not as easy as auto-scaling for sites hosted on servers. Data platform operations, such as transactions, are not stateless. Therefore, care must be taken with direct connections between data services and user applications while scaling is in flux to avoid a service disruption.

7. Latency Considerations

Latency is frequently thought of as the biggest difference when comparing NoSQL databases and cloud databases, mainly the serverless ones. The challenge resides in the ability of utilizing NoSQL databases as volatiles or features that can be read very frequently at an extremely low cost. Latency stored-in-place semantics approaches that not always treat the data as truly transient data and the serverless design trying to avoid charging for the infrastructure usually make both very different in terms of usability.

Latency has different meanings depending on the layer of abstraction by which data access is being considered. For applications accessing a storage system, latency refers to the response time of the request used for putting and getting the object. For applications accessing low-level storage, latency refers to response times. For applications accessing relational data through drivers, latency usually refers to the response time of the core transactional requests or calls. In addition, there are higher-level application frameworks in between layer abstractions that more loosely define latency and do internal buffering optimizing for bursts instead of individual requests.

Different types of latency have different underlying causes and must be accounted for separately when evaluating a system's overall performance and cost-effectiveness for a particular solution. The underlying causes of high latency can be at any layer in distributed systems, from the application implementation calling the data layer, to the service architecture and implementation with intermediate network routing layers, to the compute and storage choices and their networking, to the only networking costs, to any intermediate layer from the provider's implementation. Therefore, the application must first organize throughput requirements and any alliance in charge of guaranteeing latency, together with desired level of service, before guessing which levels can be improved a user-defined task.

7.1. Factors Affecting Latency

Latency is a term that describes the total time needed to process a request or transaction, comprising the total time from input to output, including both input and output durations, as well as data processing times. Several factors affect latency. First of all, the distance between the request input, and the data processing nodes and data output locations affects latency. Networking latency depends on the number of network hops the packets must traverse, the bandwidth specified for the connection, and the round-trip latency across each hop. If a query reads data in one geographical location and the output is sent to a location far away, it must traverse a lot of network hops and Network Latency will be high.

Computing latency is defined as the amount of time taken to complete the entire process, from data arrival to result generation, by the core services to treat the data. It is clear that the greater the number of operations available, the longer the time necessary to produce the result. Notice that by increasing the number of operations and/or the number of services available, a single operation can be calculated in a shorter time. Latency therefore grows with the number of checks that must be validated by the pricing public cloud provider or by the public utility to design a product or solution that is more effective for latency. Private cloud or hybrid systems allow a portion of data processing to be local without passing requests through the pricing provider, thus reducing latency times but increasing the use of on-premise resources, usually for high availability. Other than this constraint, the public utility must be shared and the price must be billed for each transaction.

7.2. Mitigating Latency Issues

Latency issues may indeed undermine the ability to serve requests in a timely manner. However, in this section, we will discuss ways to enhance throughput and limit the impact of latency on request servicing. The following suggestions apply both to APIs exposed by the cloud database service provider, and also to such APIs that may be specifically implemented by your organization's Data as a Service offering. Ideally, your organization should monitor the average latency. Specifically, you should closely monitor the average latency during bursts of activity. If the platform has not been implemented yet, then it is usually advisable to limit the distribution of load. For example, if the platform provides an online search capability, then it may be advantageous to limit the number of documents that are indexed at the same time. Once the system achieves a steady state, you should tune the number of worker nodes so that the average latency is acceptable. In general, however, it is advisable to limit request load on the system and allow the database system to have extended idle periods. A DBMS may choose to take advantage of idle periods for pipeline input and output processing which could improve throughput. A well-designed cloud database system would be able to dynamically increase the required number of active worker nodes, based on monitoring activity, possibly driven by the service-level agreement which the organization has signed with its customers. An organization implementing a Data as a Service offering should also strive to enhance throughput so that idle time is the exception. If demand for service is distributed around some average activity at the request level, this enhancement could entail a small overhead accompanying high demand periods.

8. Comparative Analysis of Managed Services

Providing many services as managed (or serverless) requires a high effort in engineering and precise trade-offs. We will provide in this chapter concrete elements in this discussion, presenting actual performance results for the most well-known managed solutions. We use the informal concept of functions: “Questions, we should answer with functions”, as opposed to services as cloud native or serverless. We discussed in the previous chapters key characteristics of a data management in the cloud via some essay elements describing the service characteristics. We think some metrics format tables could help. Why providing Functions As A Services Then? We see a clear and special motivation in the serverless concept. Actually, acting in FaaS will be an excellent debugging process. Once the service seems to work, moving to an event type like in SAAS – with no overflow response for the question seems clear. The costs involved in providing the managed service are complex, consisting of many elements. Quality of Service, namely latency truly experienced, and not only in the last mile, keeping in mind possible SLA negotiated; also maximum times to execute – high percentiles; and for data specific aspects containing size, volume, type of anomalies must be taken into account; moreover availability, and especially type of error response, and provider in the usual cloud native duality have to be evaluated. Latency and availability will be provided by the cloud SLA; error types will depend on the technology involved, with some special reactions for dedicated algorithms managing/learning the data and needing specific type of test set ground truth related. We will provide many details comparing a few cloud environments usually are considered for “recruiting” serverless components; the involved time allows for item process optimization and potential cost constraints; budget defined during the contractual process; how to simplify usability and development, possibly trying model-architecture only – as indicated by some specific scalability procedures before directly programming in a specific architecture language.

8.1. Performance Metrics

Comparing performance across different platforms is challenging since even traditional benchmarks are not directly translatable to the capabilities of any database. Each managed service has its own code paths for execution of SQL queries, connected to different data engines with different structures and optimizations and capable of supporting different execution plans for the same query. Some service providers emphasize performance, while others focus on availability and service reliability.

To understand performance, we walked through diverse workloads on some of the most popular managed services with a simple set of workload types. We ran the same set of queries on purpose through a Postgres-compatible interface, mainly for storage cost estimation, but the service should route requests through the database-type container chosen in the cluster and send requests to the respective database container. Part of the request routing and load balancing does occur at the cluster level, so routing overhead should be less for managed services. The results below discuss performing differences across different vendors based on these use cases.

Our cluster for the database and service comparisons consists of three instances that support the necessary memory and storage requirements. Each node has 8 CPUs, 32 vCPUs, 128 GiB memory and 1200 GB storage space. The region for the cluster is in the US East. The parameters used for the individual runtime launches are summarized, along with the approximate storage cost per month for a 24-hour usage of the instance. The values vary by vendor and depend on a bunch of factors including usage time and reservation commitment. The results for queries from the various workloads are summarized and displayed.

8.2. Use Cases for Different Platforms

This section provides a brief description of features and use cases for the query fleet model, the founding characteristics of serverless data platforms that use it, and other types of data platforms that follow different paths and are optimized for different types of executions. The approach we follow in this section is an attempt to interpret and map a divergent set of managed services to specific use cases. Different services aim at helping different customers with different kinds of data analysis workloads. From how we see it, the query fleet model supports interactive exploratory analytics on larger data sets using fewer data pipeline operations with lower development and operational costs. Serverless data platforms built using the query fleet model are primarily optimized for exploratory analytics use cases with natural language query interfaces or SQL-centric query engines. The other paths taken by data services and platforms in the market are more suited for workloads that need more specialized data expertise in the querying and analytics processes, and intuitive, easy-to-use, higher-level abstractions mapping the operational intricacies of the systems for batch-type processing. Such other paths are nevertheless still valuable for their capability of serving customers working with heavily regulated data types and use cases, or larger organizations needing enterprise data governance capabilities with disparate teams performing a variety of data analytics and processing functions.

Shared data is the foundational and common characteristic of the query fleet architecture. It is a key feature in enabling lower operational costs by sharing the portion of the infrastructure that has the single largest cost component—the compute resources. The compute engine is always shared across multiple queries. This allows the centralization of costs of storage and storing the dataset in the best representation for answering different types of queries by caching copies of data in the optimal physical layout for different incoming queries, rendering fast interactive responses for ad hoc queries based on the query type.

9. Future Trends in Cloud Databases

This chapter prognosticates developments of cloud database technology in the next 3–5 years. Some predictions are easy to make since they are already underway, such as the continued growth of holistic cloud database and serverless data platform offerings from cloud providers, the continued growth of relational database cloud migrations, and the growth in demand for the most proven database services, such as analytics on cloud data lakes and transactional support provided by high-end operational cloud services. As evidenced by the never-ending numbers of database product names at all three levels of the database hierarchy, vendor differentiation is alive, albeit in a much smaller market than all the private-label products introduced in previous decades. This chapter does not directly cover database development database life cycle and database performance, availability, and reliability, but trends around those topics will have cascading effects on cloud databases and services.

The demand for capacities to govern, protect, manage, enable, and control data are not likely to go away soon. Therefore, growth in cloud database services will increase as cloud providers innovate, broaden, and expand densities of functionality and agility. Accorded the ephemeral, tributary nature of data in the cloud that is occasioned by the many new cloud services now available, data may become the silos of the past. That being said, myriad considerations and patterns governing cloud data use may bring acceptance of a new cloud data mainstream, where cloud databases become predominant models for storing and querying data in the cloud and where cloud data management becomes the rice scribe of an organization's digitally transformed future. Such a future would ideally consist of cloud service architectures that seamlessly and automatically provision and configure appropriate cloud database services in the appropriate functional contexts.

10. Conclusion

Part I of the book provides a broader understanding of cloud databases. The authors collectively discuss the history of databases with a perspective that dates to specialist file systems and early key-value stores and suggest the guiding principles necessary for a cloud-aware database implementation. An overview of what it means for a data system to be designed for a public cloud environment is provided. The discussion looks across layers of the stack at data storage systems, data processing systems, and transactional capabilities. While databases exist in an ecosystem, the cloud radically changes the way that provisioned services interact, and designers must consider that when building cloud-native data systems. In the second part of the chapter, an opinionated overview of some new near-databases or serverless data platforms that anticipate what may be considered the next generation of databases: event stream platforms, data warehouses, analytics engines, and Kappa and Lambda architecture products for stream processing is offered.

Serverless design enables developers to focus on their immediate problem at lower and lower levels of specialization in the tech stack. It breaks vertical specialization into a set of easier-to-solve horizontal problems, each with its own observability and billing. This squaring of the cloud computing circle is driven by the scale of existing, specialized services — whether it be the simple usage patterns or the complex pricing and implementation of specialized services. At the same time, building a several-ordered-level-higher computational and storage system remains extremely hard, reflected in the prices and current outages of the specialized services. It redirects attention to the types of workloads suitable for higher-order, stateful services. An explicit example of that latter concern is a recent map of a state-based solution on top of a cloud data warehouse.

References:

[1] Gupta, Anurag, et al. "Amazon redshift and the case for simpler data warehouses." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015.

[2] Arora, Indu, and Anu Gupta. "Cloud databases: a paradigm shift in databases." *International Journal of Computer Science Issues (IJCSI)* 9.4 (2012): 77.

[3] Deka, Ganesh Chandra. "A survey of cloud database systems." *It Professional* 16.2 (2013): 50-57.

Chapter 7: Data Warehousing and Analytical Processing

1. Introduction to Data Warehousing

Data warehousing is concerned with the storage of huge amounts of data in a manner that allows efficient analytical processing of that data [1-2]. A data warehouse stores a large, consolidated, historical, and well-organized collection of data to support relatively easy access and quick response time. The warehouse stores data from a variety of operational and external data sources. Consolidating and preparing the data for analysis necessitates data cleansing, transformation, and integration. To support analytical processing, the data is loaded into the warehouse database using an architecture that supports efficient loading. The warehouse schema is carefully designed to support a variety of analyses in an efficient manner. What data is stored in the warehouse and how is determined by a close collaboration with the users through a series of ad hoc sessions exploring their analytical querying needs. The users also require facilities for query submission and display of query results, usually through user-friendly graphical interfaces.

The business operations, which the warehouse supports, are carried out on a day-to-day basis using a variety of operational databases. Information about these operations is also gathered from a variety of external data sources. This operational data is often used for maintaining and updating the corporate memory, which is stored in the warehouse. This data allows the corporation to derive business performance metrics. These performance metrics act as corporate guidelines and goals for the business operations. The warehouse acts as an analytical engine that derives information and transforms this data into competitive advantages for the corporation through accurate forecasting, detecting business trends, and identifying new business opportunities. Data warehousing is an integral and essential part of the overall corporate data and information infrastructure.

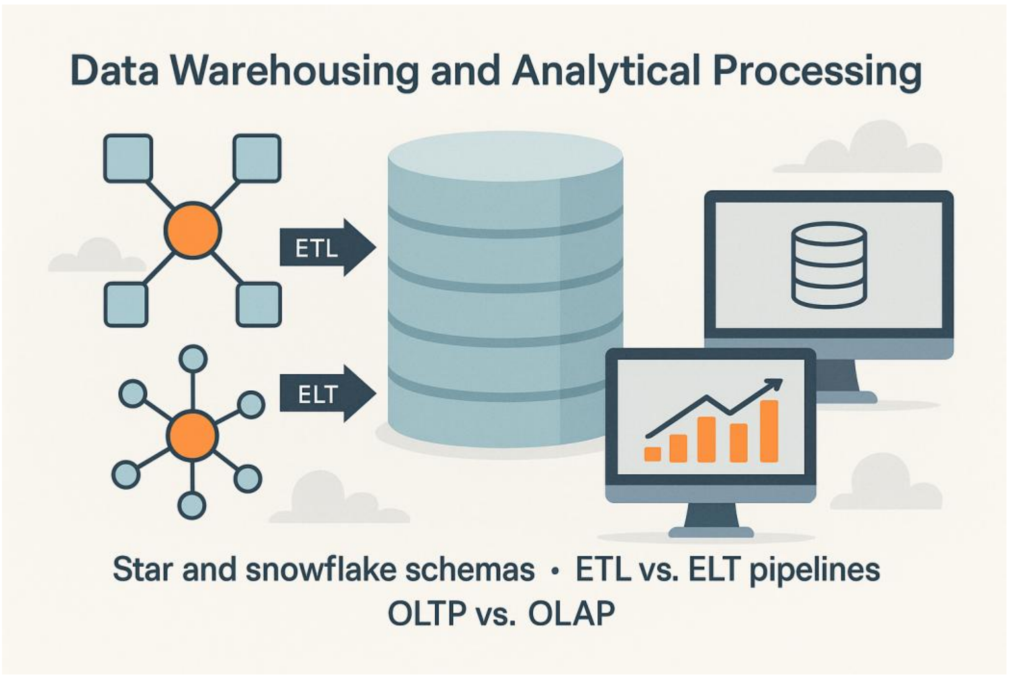
Data warehousing has evolved over the last 30 years into a field that encompasses many styles of systems, many application areas, and many system usage characteristics. It is no longer merely about building a large, centralized repository of integrated operational business data or enabling traditional decision support query workloads. Data warehousing has branched out into a hierarchical, tree-like system with both large and small systems of various timescales and implementation styles. Parallel and distributed architectures support rapid query answering on important business data. Data marts provide specialized presentation of data important to specific business functions within different data complexities and integration styles. Stream processing addresses time-critical delivery of specific business data. Analysis in motion demands ever-accelerating primary operational business processes. Decentralized data-capture systems facilitate and accelerate strategic global organizational services.

The diversity of architectures, designs, and data marts now extend far beyond what practitioners might have conceived merely a decade ago. For many business needs, a data warehouse might serve as a complete repository. In addition, new data sources, both external and operationally driven, are constantly evolving. Business metadata is now used to support user data understanding, ease of use, and business operational efficiency. Quality metrics are now in place to ensure timely and accurate delivery of business-critical information. Accompanying all of this is a growing market of user tools for easy, accessible business information exploration, analysis, visualization, and consumption, vendor solutions to support business intelligence, metadata, and quality needs.

2. Data Warehouse Architectures

2.1. Overview of Data Warehouse Architectures

The notion of architecture encompasses the features and fundamental properties of a software application. It refers to stable and substantial decisions made by designers that comprise the gross structure and design of the application, as well as its key characteristics, including modifiability, performance, robustness, security, and usability. Software architecture is a high-level description of the software system. Because software architecture represents the major building blocks of a software product and the relationship of these abstractions to one another, it is an important element of any software product. This section discusses common architectures that commercial data warehouse products use. Data warehouse architecture is but one factor to consider in choosing a product. Other factors include tools and development effort, data movement, and cost.


Data Warehousing and Analytical Processing
ETL
ELT
Star and snowflake schemas • ETL vs. ELT pipelines
OLTP vs. OLAP

Unlike OLTP systems that serve only day-to-day operations, data warehouses serve various purposes for different types of users. Data warehouses are mainly used for decision-making, forecasting, long-term data storage, and scientific and mining purposes. BI and analytics systems have unique features as compared to OLTP systems. With these diverse functionalities, data warehouses need to address various issues including design, performance tuning, backup and recovery, load balancing, and security. A data warehouse is a set of decision support data integrated for a particular purpose. The physical implementation of a data warehouse in a computer system depends on the decision support system requirements. The decision support requirements determine the data, frequency of updates, processing overhead, degree of integration, and the performance requirements. These parameters guide the data warehouse modelling, physical implementation, and architecture.

In the past decade, organizations have transformed their operations though data warehousing and analytical processing. Data warehouses offer support for the functions of decision-makers and users typically positioned at the top, or at least close to the top, of an organization's decision-making hierarchy. Examples of such functions are forecasting, time-sensitive analyses of data along multiple dimensions from many perspectives, and analyses of very large volumes of current and historical data stored in data warehouse. A high-level summary of the key phases of data warehouse development is shown in the next chapter, along with the diverse personnel typically involved in such efforts.

Thus, data warehousing is the process of providing low-cost, low-latency, non-disruptive access to integrated, near real-time, historic, data from all sources, scaled to support the entire organizational user population. Explicitly excluded from this definition are islands of operational data marts containing integrated data from a few sources used by a few local analysts running ad hoc queries that meet local needs. Unlike data warehouses, data mart queries tend to cause disruption because the data are often used in operational transactions; further, the data mart data are targeted for local, current decision making, not for corporate level, historic decision making. The remainder of this text discusses product architectures for data warehousing. These architectures differ in their strategic direction, objectives, principles of operation, and implementation specifics.

Data warehousing has emerged as one of the key tools for leveraging information assets that were previously unutilized and, in many cases, underestimated as valuable resources. The main driver for moving to data warehousing is the need to better serve the information needs of an organization's decision makers, executive staff, and other users of the organization's data. The need spans all organizations: corporate, government, or academic, and all functions: business, engineering, or science.

Data warehousing and analytical processing are tightly coupled activities. Not only is the data warehouse the repository which serves the analytical processing activity, but it is also the result of multi-time variance extensions over today’s operational data, incorporated to the data warehouse from multiple applications. Furthermore, building data warehouses and engaging in analytical processes must happen in conjunction to one another if business value is to be had from them, and be agile enough to respond to continuously varying business objectives and requirements. Accordingly, the most successful data warehouse deployments have happened in environments where business leadership has taken the responsibility for defining the mission and objectives of these efforts, in line with the rest of the organizations. Data warehouses allow users to make ad hoc queries, produce reports, verify hypothesis, and extract the data they need, to whom it may be useful, and whenever it may be convenient.

A variety of architectures have been proposed as solutions to the data warehousing and analytical processing activities. Use of the data warehouse as a staging area for analytical processing is one common approach to making multiple time-frames available, engaging in deep evaluation over large data volumes, timeliness and dependability of data freshness, or keeping storage and associated operational overhead costs low. This is the primary function for which on-line analytical processing servers were built. Fact-specific hypercubes allowing efficient analysis and display of small, aggregated surface areas are also a popular solution for making high analysis throughput demands scalable. At the same time, multidiscipline data- and operational-extraction activities are growing in usage and capabilities and are often integrated into the warehouse and analytical processing solution in hybrid implementations.

Data warehouses hold consolidated data from one or more source systems. Data is cleaned, transformed, and stored in the warehouse where it becomes the basis for operational monitoring, historical tracking, and decision making. This chapter introduces data warehouse architecture, modelling, and validation. It explores the whys, how's, and what of data warehousing. A basic premise of this chapter is that data warehouses are analytical processing structures. They promote decision making based on data queried via SQL from a relational model. If you want any other architectures, like the OLAP cube or any special considerations for other structures, skip this chapter. This is largely a book on data warehousing. Data warehousing is the functional separation of analytical from operational processing. Yes, the two are often implemented in the same database. No, this does not mean that the concepts are not useful. Indeed, the best OLTP database design is in many ways a worst-case OLAP design. Introducing data warehouses is akin to saying that we need both logic and algebra. Furthermore, discussing data warehousing is our way of introducing the concept of time-and group-based analysis.

We cannot rely on data collections used for operational processing – if only because these collections are posed in the context of current transactions. The hardness of these data verification problems cannot be overstated. Discussing warehouses is easy; getting OLTP and operational data verification right is the hard part. The what-questions related to security, schema, and reliability are often reduced to can-I-sweat-the-data-to-prevent-monsters reclaiming it. Instead, thank you for providing me with the means to write this book.

2.2. Benefits of Data Warehousing

Data warehousing is like all disciplines in that it is a substantial up-front investment of time and resources. It also has long since-term value [2-4]. The benefits vary from organization to organization but usually fall into the following areas.

Data Quality and Integrity. Different operational systems typically represent the same real-world data in different and inconsistent ways. Because source data in data warehousing is often cleaned and transformed so that the data warehouse supports a consistent data model, analysis done using the data warehouse is generally much higher quality, leading to better insights and decisions.

Time-Saving Analytical Processing. For many organizations, the cost of running queries and storing results is significantly less with a data warehouse than with operational systems. Before the advent of data warehousing, data analysis within organizations was typically run directly against operational databases. Data warehousing allows organizations to offload all these analytical activities to a separate, optimized database, allowing for faster response times and fewer constraints to normal transaction processing.

Data Permanence for Analysis. Data warehouses allow very large amounts of historical data to be retained, and to be retained just like the source data. This is very useful to analyse trends over extended periods of times. Organizations can then more easily make decisions about where to allocate resources to minimize lost sales due to stockouts or lost customers.

Enhanced Analytical Processing Capability. The combination of optimized data warehouse storage structures and data warehouses query processing systems provide analytical processing capability that is not possible today with conventional operational systems. For example, the use of multidimensional data warehouse schema architectures allows a vast range of sub-second response time analytical query processing to occur.

There are numerous advantages of data warehousing and the functions it provides, including:

• Integrated view: Different business functions are focused on different subjects like sales, customers, marketing, and other values. A warehouse provides a unified view needed for cross-functional analysis. For example, customer purchases and order details from a data mart on the sales function can be correlated with advertising spending across media and sales regions from the respective data mart on marketing functions to compute the response effect from an ad campaign.

• Filtered history: Data warehouses summarize and filter transaction data. Beyond this filtering function, they also provide time-varying values and concepts excess capacity by the time-variance modifier associated with most dimensional hierarchies. These time-varying values are essential for management and business planning. Typical examples of these values are projected sales revenue and salesperson commission. These values do not natively appear in the respective sales volumes and sales revenue fact tables.

• Shared complexity and consistency: Data warehouses contain complex derived data structures so that each data mart does not have to contain all these complex derived data structures.

• Higher performance: Data warehouses contain data structures specifically designed for analysis rather than transaction processing. They also contain the computer resources required to analyse the data. Thus, using a warehouse, pre-processed data to satisfy management reporting and decision support requirements will perform faster. Higher frequency of such specialized processing will also be affordable.

3. Star Schema

3.1. Definition and Components

The schema diagram depicts the data warehouse schema planting stored data collections in a star-like structure, commonly known as a star schema. A star schema contains a central large market transaction fact table stored as a table and a collection of smaller classification or dimension tables. The hierarchical organization of the classification tables represents categories relevant to analysis, e.g. local store location, product description, and time. Individual fact records identified in the fact table are stored on a per-market basis. Each record in the fact table stores values for the various number of units sold measures. Each record in the fact table is connected through either a single key or a pair of keys, to classification tables in the star schema, designating the store, the product, and the time applicable to the set of sales.

The star schema is a type of database schema that is a widely accepted elementary model for the classification, design, and understanding of analytical processing databases. The primary structure of a star schema consists of one centralized fact table that describes primary key attributes used for the classification of fact records relative to the fact space, and being related to multiple multidimensional dimension tables that provide additional descriptive attributes for classification, filtering, and grouping. In a star schema, the option for dimension tables is to contain for each dimension a denormalized set of attributes that fully describe the dimension space as well as the temporal and contextual semantics of the facts, except for hierarchies, which are sometimes represented in specific auxiliary tables.

A star schema has several important characteristics and design rules. First, the fact table contains data on large numbers of transactional events or measurements and is usually historical, meaning that rows are created as new records are created for the given events, and there is rarely if ever, modification of existing rows. Second, the basic measurements stored in a fact table, active or passive, are numeric, and the cardinality of dimension keys from those dimensions must be less than the cardinality of the fact table for those keys. Third, the fact table is usually partitioned into small parts based on clumping on an attribute or set of attributes and preferably on a single date partitioning attribute where record retention policy uses the date as input.

Third, the dimension tables are usually small in that the size of the largest dimension should, preferably, not exceed 10,000 records. Fourth, dimension tables provide context and semantics for the facts stored in the central fact table. In addition to the shared attributes for keys, dimension tables usually model a set of additional attributes that describe the various types of clumping, grouping, filtering, and classification of the facts, and are frequently organized in an attribute hierarchy.

3.2. Advantages of Star Schema

Star schema poses a very simple and highly efficient data structure for access in analytic processing. The market actions of the customers are viewed and analysed in the highly structured business areas of time, location, and product. The type of measurements relevant to the basic business actions are completely specified. This level of sophistication can be easily used and understood by business analysts, statisticians, and market specialists. Given the uniformity of its information content, the star schema also permits considerable storage and performance optimization. The specification of a star schema also provides a data warehouse designer with a compact representation of data warehouse requirements. The star schema, therefore, serves as the equivalent of a logical data model for a data warehouse.

Multidimensional data models, including the star schema, enjoy favor with data warehouse designers for their high performance for ad hoc retrieval. Performance is important in supporting high numbers of users performing complex queries to summarize and derive value from data. The quality of service typically expected for a data warehouse is for queries summarizing a year to execute, or be optimized to execute, in seconds. When a user is waiting for the results of a query for five minutes, that user is not satisfied. Additional factors which contribute toward satisfactory performance include:

- A star schema's data is stored at the right level of detail. A dimension in a data warehouse usually contains few to many rows with descriptive attributes that qualify measures at other grain levels. When a star schema is implemented properly, the level is that relevant to the business question posed by the user. The detail attributes in a star schema do not, generally, include customer name and address; they include data at the correct level for analysis addressing the business question. - A star schema's fact table is constrained to contain few to many attributes. A fact table in a data warehouse schema usually contains many rows with values for measures. Typically, the table row contains measures at different grain levels; it represents a point in time, e.g., daily sales or a specific customer order. When a star schema is implemented properly, this level is that relevant to the business question posed by the user.

The star schema model has many advantages. First, compared with the normal form or snowflake schema tables, it greatly reduces the number of joins needed to satisfy data retrieval requests, and such joins are thus inexpensive. This speeds up the retrieval of data from the warehouse. The result is that the star schema can be used to meet the needs of a wide variety of users and applications. Second, the star schema is easy to understand. Dimensions are usually small enough to be presented in crystal clear detail to users. This means that most users will be able to easily understand the structure and contents of the data warehouse. In turn, this means that users will have little trouble expressing their needs in terms of data warehouse queries.

Third, the star schema can easily be used for multitier architectures where a data mart associates local sources together with the data warehouse. More local data on interest to users can easily and efficiently be associated with the global enterprise warehouse. Fourth, the star schema can easily accommodate anything that is not planned, such as additional dimensions or additional attributes or hierarchies in dimensions that already exist. It is easier to extend a star schema than a schema based on the snowflake structure. Fifth, the star schema effectively serves the purposes it is designed for: data retrieval. It is purpose-oriented towards processing the types of requests for which data warehouses are primarily used. For example, if the database is subject to frequent updates, a star schema corresponding to a multilevel or snowflake schema is not an efficient structure. In contrast, a reduced star schema can speed up retrieval time considerably. Even so, the star schema allows a wider variety of query types than can typically be allowed by an ODS.

3.3. Use Cases for Star Schema

Star schemas, as the building blocks of data warehouses, allow business analysts to create large amounts of analytic data from dozens or hundreds of source tables. Business users then employ that analytic data using standards-based SQL tools or proprietary dashboards. This analytic data might consist of fact tables of sales, shipments, and inventory data each joined to a star of related product, customer, and geography tables. Alternatively, it might also consist of fact tables of a wide variety of different company processes each joined to a smaller set of shared product, customer, and geography tables. Joins with "small" dimension tables are fast to execute in any relational database, and they tend to be even faster when the dimension tables are small; we recommend that they be smaller than a few megabytes.

Most studies of user query workloads find that more than half of queries are cluster scans on dimension tables, while about 30% of queries are joins between fact and dimension tables. In addition, tables of distinct values of frequently queried attribute sets are often much smaller than other dimension tables. Moreover, dimension tables are often periodically updated. All of these factors make the star schema a strong candidate for optimization in data warehouses and in databases that process analytic workloads over operational data. However, star schemas are not the only schema choices available. In addition to possible schema alternatives, the star join optimization has its own complications.

Star schemas are often utilized for reporting/analytical processing requirements that draw data from a single business unit for a single point in time and that have a small number of possible aggregations. For these applications, star schemas provide the physically simplest, most accessible database design. The underlying data is denormalized; all the data from the fact table are accessible with a single I/O operation that fetches a disk block. The normal way to group and organize the physical data for an application is to make the expected user queries as efficient as possible. Since often only a handful of data aggregations are actually wanted from analytical-processing systems, it makes sense to deformalize the actual physical data for just those few. Star schemas provide the simplest access method to obtain those few.

The dimensions of the star schema are relatively few but are often very fat, meaning that a variable in the dimension table can have many different values. Again, this fact has no effect on the efficiency of a point query for an individual row of either the fact or dimension tables, since the dimension lookup is performed in memory from the dimension cache. Point queries are also implemented in the simplest possible way – in the SQL sense, those queries simply involve a join operation. Some star-schema applications involve truly gargantuan dimensions to produce reports that report by detailed combination. In those extreme cases, some thought may need to be given to the actual on-disk data structure that implements the fat dimension, to provide efficient access to that dimension on disk.

4. Snowflake Schema

A snowflake schema is a structure of data that incorporates tables into sub-dimensions. Multiple related tables come together to form the schema shape, which resembles a snowflake. It is a collection of star schemas, which are normalized. A normalized data structure saves disk space and improves data input processes and decreases data maintenance. However, the trade-off is that it increases disk usage and degrades data retrieval speeds. Snowflake schemas are usually applied to operational data stores or data marts or are presented to the client in specific cases. Data warehouses typically present user views. In this way, a snowflake schema is often less pleasant to end users because it is further removed from a single table expressed in business terms.

4.1. Definition and Components

The Snowflake Schema is a logical arrangement of tables in a relational database in a way that the ER model of that database resembles a snowflake shape. A snowflake schema is a type of data warehousing schema that is a logical arrangement of tables in a relational database. The snowflake schema consists of a central fact table and one or more levels of normalized tables representing dimension data. When these fact and dimension tables are joined together, their structure resembles a snowflake pattern and enable users to perform complex queries of transaction data together with its associated business context covering the many different possible dimensions of a business.

A snowflake schema is like a star schema, but with the difference that the dimension tables are further normalized into additional tables. Snowflake schemas are usually found in data warehouses as well as in other data marts. However, snowflake schemas introduce challenges with respect to performance and query complexity since joins are required to gather the necessary dimension information. Snowflake schemas are typically designed with a much higher level of normalization to reduce storage space and redundancy. Colocation of related columns can often result in the derived attributes calculated from the functional dependencies on a dimensional table being in fact tables instead. Some BI tools do not efficiently handle such schemas, in which case star schemas may be preferred.

Nevertheless, the snowflake schema can potentially save storage space. It is used in a data warehouse considered to be a hybrid of the two other schema designs. In a normalized design, the focus is on data integrity; the stars are preferring fast query performance, while the snowflake schema partially sacrifices query speed for a storage saving. It also uses a concept of recursive join to create additional levels of hierarchy for dimension tables.

4.2. Advantages of Snowflake Schema

Star schemas provide advantage of simplicity. Snowflake schemas induce some additional complexity, but they have three key advantages. First, some fact tables will contain attribute values that differ based on the granularity of the fact table, for example, currency, exchange rate type, and so forth for financial transactions. These attributes are relatively small in terms of capacity, and but they can cause a significant amount of replication in a star schema. For small snowflake dimensions—i.e., snowflakes that are at a higher level than the fact table granularity—these attributes can be engulfed in the snowflaking dimension table and can thus be removed from star schema where they would otherwise be replicated.

Second, fact table volumes can be immense. Bigger fact tables can benefit from denormalization in different ways than smaller ones. For example, assume that we have dimension called "Salesperson" that includes the name of the salesperson, commission rate, and so forth. For a data warehouse that is predicated on years of sales, the Salesperson dimension will inevitably include very large number of unique member keys. Unlike smaller fact tables, dimension tables that reference large fact tables become cumbersome to maintain because of their size. A snowflake schema can offer relief, to drive down the size of your dimension tables. In this case, small size is your goal rather than big size.

Third, at least one vendor has claimed speedier response times for a snowflake schema than a star schema—especially when the dimensions are small, and there is not a lot of dimensions to "join"—callbacks, in the case of snowflake schemas depending on the dimension sizes. The vendor claims speed gains come in part because the snowflake schema can result in lower impedance mismatches blended, accessing data rows, and thus a more optimal process, due to closer data location in disk blocks of the unit maritime data.

4.3. Use Cases for Snowflake Schema

As a logical model, the snowflake schema does not prescribe clustering or data organization. This lack of specificity affords flexibility in implementing a snowflake schema in many varied environments, necessitating certain compromises in how the organization maps its data to such a model; as such, our discussion may also introduce by example some simple conventions that simplify things. Often, however, snowflakes are not implemented exactly as depicted, particularly the outer layers or descendent relations, but the principles they embody can still be of significant assistance.

There are situations where using snowflakes is preferred. Where data are highly hierarchical, snowflakes are as effective as providing specialized support for the associated hierarchy as any other structure. The better usage of disk space resulting from this factor is also an advantage in such cases. With descendants being stored separately, snowflakes are also more efficient when ancestors are joined with facts, in the direction of the ancestor-to-descendant join. The specialized structure also makes it easier to implement view control mechanisms that hide elements of hierarchies, such as a view that prevents access to any leaf elements. Snowflakes may also improve interaction with external tools and applications that interact with the data warehouse since they closely resemble external reference data dimensions.

Eventual snowflaking of dimensions into more detailed structures can also occur as a warehouse matures, and a dimension that was initially kept in simple table form through entry-level attributes is refactored as use of the dimension becomes more intricate and based on a deeper understanding of its profile. Finally, snowflakes can become the permanent structure if the environment or charts.

5. ETL vs. ELT Pipelines

In this chapter, we cover the process of implementing ETL and ELT pipelines, and we go into detail about various ETL and ELT tools for data movement and transformation. Before that, we establish the need for ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) processes, and we outline the different types of ETL and ELT processes. In the next section, we explore the two branches of the data movement and transformation process: ETL and ELT.

ETL and ELT processes are designed to take data from disparate systems, prepare that data, and load it into a target system, such as a database or a data warehouse. The difference between the two approaches is how the preparation process is accomplished. ETL processes transform data before loading it into the target system, in-between the extract and load phases of the pipeline, while ELT processes load the raw data into the target system and then transform it once it is there. The target systems for ETL processing are usually non-SQL-based systems such as data warehouses.

5.1. Definition of ETL

In the ETL pipeline, data is extracted from a source system such as a transactional database. After extraction, the data is then transformed, with the transformations usually including operations such as filtering, joining, cleaning, and summing. Finally, the transformed data is loaded to a target system, which is usually a database management system or data warehouse. The transformations are accomplished by a middleware processing engine, which can be coded in a general-purpose programming language, or an ETL-specific programming environment, which may offer a simpler programming model using pre-coded data transformation functions.

Fittingly, the architectures of systems used for data analytics processing pipelines are typically referred to as ETL: extract-transform-load, or ELT: extract-load-transform. In essence, both ETL and ELT pipelines copy source data, transform it, and then load the transformed data to a destination system. ETL pipelines store transformed data in the destination analytic system. In contrast, ELT pipelines first load raw data into the destination analytic system and then transform it there.

ETL pipelines extract data from the source systems, apply transformations, and then load the transformed data into the destination analytic system. ETL pipelines require that target analytic models must be defined and the source data transformed according to those target analytic models using prescribed business rules before loading the data into the destination system. ETL systems do the data collection, integration, and preparation before the actual data analysis, which would take place only after the data has been extracted, transformed, and loaded.

An ELT pipeline uses database functionalities offered by modern data warehouses to store and transform collected raw data as needed when responding to analytic queries. ELT pipelines typically also support ad hoc analytics and self-service reporting. ELT pipelines gather raw data from different sources, and with little or no transformation perform a bulk load of the raw data into the destination system. Periodically, the bulk loaded raw data is then transformed as needed and removed or curated into business-analytic-friendly formats for operational reporting or in support of performance metrics in responsive analytic dashboards.

5.2. Definition of ELT

ELT processes first extract and load data into a staging area and then perform transformation steps. ELT architectures rely on data processing engines that are physically located in the destinations holding the analytical datasets. ELT–using systems process data in the cloud since the destinations are typically large cloud-based data warehouses or lakes. In cloud environments, the cost of loading data is typically much lower than the cost of transporting data to additional processing engines, required by ETL systems to perform data transformations. Furthermore, staging data – data in their original, untransformed formats – in databases or data lakes that preserve all detail are also emerging patterns of analytical processing.

The use of cloud architectural principles has changed the meaning of the ETL acronym. ETL – Extract, Transform, Load – was used for systems where the data transformations were performed in an engine that was separate from the analytical data repositories. ELT – Extract, Load, transform – was used internally in specific products and research prototypes. However, cloud systems enable other deployment patterns as well, and there has been significant interest in leveraging other deployment patterns with potentially lower TCO. Internally, we refer to these systems as Data Movement Engines – Data Extraction Systems that use these design principles. We do present a few design decisions below that we think are essential for any state-of-the-art Data Movement Engine.

Typically, when we copy source data to a warehouse and then convert it to a target schema, we refer to the operation as extract, transform, and load. The operation is referred to as extract, load, and transform if we copy data to the warehouse without converting it. It is important to clarify that this does not imply that the data necessarily resides in the warehouse in its native format. On the contrary, most sources have heterogeneous formats, and few warehouses can resist the temptation of moving data into a homogenous structure to facilitate access and query optimization.

This is still a widely used method for populating data warehouses. A comment about the pragmatic use of data formats is that one tool is a database search engine written by an engineer and another tool is a database storage engine written by a scientist. In the past, full extract-and-transform costs made it impossible to transform all data, but the recent dramatic increases in available storage space make this prohibitive approach ever more attractive. We must store the data anyway, and it makes little sense to be selective about the data that is transformed and the data that is not. When we transfer data from a system to a data warehouse, we usually do it in chunks, because it involves huge amounts of data. The extracted chunks must be transformed individually either before or after being sent to the warehouse.

5.3. Comparison of ETL and ELT

Using the above definitions, we can make the following observations. First, both ETL and ELT can be used for exposing data to analytics users. As traditional tools for warehousing and BI have historically depended on ETL for sundry operational and semantic transformations, ETL first emerged as a "data preparation" designation. However, a new generation of cloud data warehouses allow ELT to be used for such preparation, analytically pushing down, via SQL, the operations typically associated with ETL tools. With this "warehouses as preparation" model, ELT disaggregates the data modelling phase, enabling different organizations within an enterprise to independently model their own "data marts" with custom logic exposed in easily discoverable ways. Further, like ETL, ELT can also be used for application-oriented data integration and other tasks beyond analytics, though this is relatively rare. The advantage of ELT in the preparation process is its capability for near-real-time updates.

Second, ETL and ELT differ in how they manage schema evolution in the source systems. With ETL, data stays tightly coupled with the source application schemas. ETL relies on manual management of schemas and semantic mappings, as well as per-source issues of data freshness and level of updates, meaning it often requires human intervention to fix the inevitable problems. ELT acts as a middleware layer for the data, decoupling source data from the business logic. As a result, many ELT offerings automatically manage per-source issues using algorithms to deduce mapping from activity, key, type, and value-frequency change monitoring, along with refresh issues based on source database technology. ELT is also the more pragmatic choice in cloud environments, which provide very different modes of data modelling.

Data movement pipelines designed to acquire, transform, and load data into its final deliverable location have a long history that began with the birth of data warehousing in the 1980s. But these data pipelines are also beginning to evolve, as new technologies emerge to help users build pipelines faster, with less technical effort.

At a high level, there is a distinction to be made between two general pipeline design strategies: ETL, designed to load transformed data into the delivery database, and ELT, designed to facilitate loading untransformed, raw data into the delivery database, and only then performing data transformation using the database engine. ETL development pipelines are oriented around testing data transformations outside the data warehouse, using a dedicated transformation engine to build the transformation logic, which is often executed using a software development kit that is validated using sample data and then deployed invisibly as part of a larger job framework that moves data or executes jobs periodically. ETL has also evolved to involve semi-automated tools that help users discover and define transformations that can be executed during the loading process. ELT development pipelines favour executing and validating transformation logic directly in the data warehouse system, with no intermediary steps in a different transformation engine. In this design, data is typically moved into the warehouse using the COPY command, which utilizes the fast data loading capabilities of the database, and transformation is performed by executing a series of SQL commands that implement the transformations.

Both approaches have their strengths and weaknesses. While ETL-based data pipelines are less efficient than ELT-based data pipelines – that is, they take longer to execute and require separate engines to perform the loading and data movement processes – they also tend to be better suited for typical data integration tasks, such as pulling together logs and integrating data out of multiple data sources.

5.4. Choosing Between ETL and ELT

When one wants to build an analytical application that gathers data from multiple heterogeneous data sources and makes it available to facilitate analytical processing and data mining at shards, the most important architectural decision to make is how use of an ETL or ELT pipeline for staging has consequences for the efficient execution of this task and the costs associated with it. Considerations that may affect this choice may include performance and throughput, e.g., speed for initial loads, latency for incremental updates, and data freshness, as well as costs associated with the resources used to execute the task and operational management, including monitoring, error handling, and data recovery.

Making the right choice is important because switching makes operational costs expensive and complex. We know of systems that originally used ETL switches that increasingly depended on database extraction and operation performance and purity of the external database, switching to ELT. At the other extreme, we know of many powerful ELT systems that have moved complex external transformations to expensive appliances dedicated to such tasks. The ETL vs. ELT choice is made even more difficult because hybridization is common. Moreover, both ETL pipelines and ELT pipelines are in widespread use. It is possible to be flexible and sometimes move logic from the analytical data pipelines to application logic to sometimes become an ELT data pipeline for a particular data transformation task timeshare on a high-availability ELT on a commodity cloud-based virtual machine.

When choosing between ETL and ELT, several factors must be considered. The first is simplicity vs. flexibility. ETL is simpler for many people to understand and can require a simpler pipeline to build, for example when the cloud data warehouse is used as a staging area for the transformed data. ELT-capable pipelines are more complex because they can require the data in the warehouse to be transformed for different analyses, and there often needs to be orchestration and some caching mechanism controlling what transformations are applied and when. The flexibility of ELT comes in when there are multiple users, all requiring different transformations for different analyses. In this case, only the first transformation of the raw, source data has to be done in the ETL/ELT process. All subsequent transformations can make use of these earlier ELT-transformed, staging tables, containing the raw data, in whatever form required by the analysis.

Cost is another consideration. With ETL costing money every time a database is iterated over, it can be cheaper to have many users applying their own ELT transformations than to ETL every data source every time someone wants to create some complicated report and visualizations from it. ETL can be cheaper if the number of users is small and they have a small number of set reports that need to be created and run often. Performance can also be an influencing factor. Data warehouses are designed to run complex queries efficiently. When those queries involve huge source and target tables, are extremely complicated, and are used often enough, they can rival the performance of dedicated and optimized ETL engines. When the queries are run less often however, involving the high overhead of loading, etc. ETL is faster and cheaper.

6. OLTP vs. OLAP

To obtain useful information from the vast amount of available data, it is necessary to perform some operations on them. An increasing number of organizations are choosing to collect their own data and to perform operations that can bring these organizational benefits, such as increased income and profits. To this end, it is useful to use a database that provides different types of support, but it is crucial to use the right structure to perform operations on the data efficiently. In this way, if we want to perform Online Transaction Processing, that is, we want to create applications to insert and extract small amounts of data from a large amount of data; or if we want to perform Online Analytical Processing, that is, we want to perform data discussions, that is, to perform complex extraction queries to obtain summarized information from a large amount of data and to present the information obtained through a Data Visualization process.

6.1. Definition of OLTP

There are many different types of database applications, which differ in the way they store data, how the data is modelled, or which operations are typically executed on the data. Some of the main differences between OLTP and OLAP applications have already been mentioned, but there is more to say about the two main types of databases, Online Transaction Processing, OLTP, and Online Analytical Processing, OLAP.

First, let us closely look at OLTP. Modern enterprises usually maintain a system to manage their day-to-day operations, which include every single transaction. For example, enterprises in the retail sector must manage each purchase, which consists of the bought items, their price, and identity of the buyer. A financial institution needs to record every account transaction. An airline company must handle every ticket reservation. The systems that store and manage these day-to-day operations access a large and constantly changing data set at a fine level of granularity. Management of day-to-day operations typically requires frequent modification of the database, including the addition of new tuples as well as committal and roll-back opportunities. These operations are typically executed by many concurrent users in a relatively short period of time, and they must be highly reliable to guarantee that no error occurs during the transaction process. That is why data integrity is particularly important in OLTP. It is common practice in OLTP applications to access just a few tuples of the database, and OLTP is typically designed for quick responses to short queries, which cause relatively little overhead to the processing system.

The term Online Transaction Processing (OLTP) is applied to several systems providing support for the execution of an organization’s day-to-day transactions. Characteristically, these systems are based upon a relationship or complex data model, supporting primarily the operations of reading and writing of short records, and are implemented using many concurrent application programs. Access to the data is structured around a small number of short records and their immediate back and forward pointers; thus, an access pattern is created which is strongly divergent from simple or complex record scans. Nevertheless, the data must be organized and managed in such a way that both the record-intensive accesses and the few-and-large-access pattern performance propagation problems are well controlled. Typically, an OLTP system may be used to support any number of different organizations. Even within the same company, numerous databases holding similar types of data are created and maintained. Finally, the duration of the organizational transactions is short, with transaction times commonly on the order of seconds. OLTP systems adopt relational or complex data models primarily because its denormalization of records provides access time characteristics that very closely emulate those associated with the record types used by the OLTP application programs. Data transaction entry and posting details are usually maintained within the OLTP systems, with activity summaries and summarized balances sent periodically to the analytical processing system for permanent storage. Thus, OLTP systems are primarily used for a temporary status transaction area, with the data warehouses being used for longer-term, more detailed transactions of greater operational significance.

6.2. Definition of OLAP

Online analytical processing (OLAP) is a specialized technology that allows users to view, analyse, and explore data through a variety of means. OLAP users are usually interested in getting summarized views of large amounts of data, often through interactive queries requesting quick responses. Because OLAP users are usually business analysts, data exploration is usually performed by generating reports that give the analyst some summarized, but somewhat static, view of the data. During exploration, the user may pivot the data with different commands, or drill into the data with operations that provide more detail. Reports are generated very quickly, even when the query is accessing several hundred million or billion records.

OLAP operations have special characteristics that differentiate them from relational database management systems (RDBMS) relational operations. OLAP operations are predominantly read-only requests for data that have already been computed and stored in the database. Finally, the reports generated by business analysts typically do not provide insight into what occurred, nor why it occurred. For example, an analyst might look at a report that provides the sales figures for every branch in the New England region for February 2000 and the corresponding figure for the previous year. If the data for February 2000 shows an exceptional increase in sales compared to February 1999, the analyst might wonder what factors contributed to the increase. In this case, the analyst typically reviews the report to find branches with sales increases that deviated significantly from other branches or significantly contributed to the overall increase.

An OLAP application is one where the users need to analyse business data from different points of view or dimensions. For example, users might want to see the company sales ... It might also be useful to examine these sales. By this, we mean that useful insights can be gleaned from viewing the measures for some time periods and comparing these with other time periods, by viewing the measures for some stores and comparing these with other stores and so on.

Furthermore, OLAP applications require the exploration of the data using aggregation functions. Using fewer attributes from the data and comparing the aggregates for two or more attribute values may also help in data exploration. For example, we might want to see the Company Sales by Product Category, or by States, or by Year; we may also want to see the average sale amount, or the average sale amount by State, or by Year. The sales data for any state may also be analysed to check for a change in the average sale amount across years. Hence, it might be useful to see State Wise Sales Data for Different Years; the measures are summed for each pair of attribute values, and the results could then be used to see the comparative performance. By multi-dimensional OLAP, we mean an OLAP implementation supporting more than two-dimensional OLAP analysis.

6.3. Comparison of OLTP and OLAP

To understand these systems better, we will present a point-by-point comparison of Online Analytical Processing (OLAP) and Online Transaction Processing (OLTP) systems. First, we compare the characteristics of a typical OLTP and OLAP system. The data in an OLAP system is static, while the data in an OLTP system is highly dynamic. This leads to another difference — an OLAP system is updated less frequently, while an OLTP system is updated more frequently. Typically, in an OLTP system, transactions insert, delete or modify current data. In contrast, an OLAP system receives update requests to refresh only a few summary or reference data. Consequently, it is common in an OLTP system to have concurrent users performing many transactions simultaneously. In an OLAP system, on the other hand, transactions typically run for a long time and may take several minutes to hours for completion.

Other typical differences are: An OLTP system requires many short and simple queries while an OLAP system requires few long and complex queries. Queries for an OLTP system typically access recent data from short time intervals while queries for an OLAP system access a large volume of data that may span long time intervals. A large volume of data of an OLTP system is stored in a highly normalized form while an OLAP system generally stores data in a denormalized table structure. Finally, data in an OLTP system are frequently archived, purged and compressed, while this is less commonly done in an OLAP system. Thanks to these and other differences, both OLTP and OLAP systems can achieve the goals for which they were designed, even though they use similar data.

In contrast to OLAP, Online Transaction Processing (OLTP) emphasizes fast query processing and maintaining data integrity in multi-user environments and generally contains large amounts of data that requires low latency. OLTP transactions are usually short, involve predetermined operations over a limited number of database tables, and can place a heavy load on the server and I/O system. Consequently, OLTP systems tend to be write-dominated and require an extremely fast write throughput. Data analysis in OLTP systems can only support operations such as summarized data through database views. An OLTP database is usually of a smaller size, with a small number of columns in a shallow schema, and some columns in a database study form frequently change. Data is often split into many tables using foreign key relations within the database, which is normalized to save storage space and ensure data integrity. As a result, much of the data must be retrieved by joining together many tables during processing. In summary, OLAP and OLTP have different purposes and as a result have systems that differ in design, optimization, and implementation. An OLAP database is quite large and contains summary data that allows large query executions with minimal execution time. OLAP systems are designed to perform both read and write operations efficiently, but the read operation is prioritized for minimizing query execution time. During a read operation execution, an OLAP system must return several kilobytes of data in a few seconds to satisfy end user needs. OLTP, on the other hand, is a transaction monitoring tool with a consistent commit rate that functions by quickly processing short transactions at peak request and reservation times. An OLTP system must provide a low average execution time with high throughput to be an effective transaction server.

6.4. Use Cases for OLTP and OLAP

Moving into the territory of practical uses of OLTP and OLAP systems, we will present several use cases that illustrate how online transaction processing can exist side by side and complement online analytical processing in just about every kind of business that requires extensive processing of transactional data. In fact, it is important to recognize both OLTP and OLAP applications when designing and implementing information systems that allow a corporation or enterprise to meet its needs for transactional processing, management control, and business forecasting. The digital world is a system of interrelated economic, technological, and social or behavioural facets that serve to produce and exchange goods and services.

OLTP is the heart of a corporation's transaction processing system. Without it, no customers could purchase goods and services; no approved credit transactions could pass between buyers and sellers; no purchases could be made; no bills could be paid, and no sales inventory could be kept current. Given this fact alone, OLTP must be executed and monitored flawlessly. A glitch here could result in customer dissatisfaction, loss of sales or revenues, and irreparable harm to the corporation. Business users depend on OLAP for the information delivered from that processing in the form of reports, slides, and dashboards, so much so that the age-old phrase, "information is power," takes firm root here. In fact, one of the motivations of establishing executive information or business intelligence systems, instead of merely depending on operational reports, is to allow users to interact with the data and conduct "what-if" reasoning.

While we focused mainly on just a couple of areas of use, these comments should broaden our consideration. An information system with immediate and automated feedback is one of the cornerstones of OLTP—a requirement so firm that a company that does not satisfy it is unable to compete.

The purest definition of OLTP is that it records the regular day-to-day transactions of a company. Examples of OLTP workloads are purchase order processing, inventory queries and updates, bank transactions, hotel bookings, flight reservations, or signups for a workout class. Consider how a banking application lets you transfer to somebody else, show you your current balance, and tell you about scheduled bill payments. These operations need to be processed very quickly; that is why they are run in memory. But they are also critical and need to be run with high transactional guarantees.

OLAP, when seen with a more open definition, includes any reporting and analytics workload, be it from data marts or from the data warehouse. From data marts, the key workloads are dashboards that show key performance indicators, whether on revenues or on operational metrics such as conversions, active users, churn, and so on. These dashboards are usually refreshed periodically, typically every hour at worst, automatically or on demand. They can also be Historical Reconciling reports, which help auditors reconcile the data processed by the line of business applications, and monthly or quarterly bookkeeping and closing operations that happen at the end of a period. Reports that are especially impacted by the slow speed at which data can be ingested are the 'drill btn' reports. These reports give a user the ability to navigate easily through the data to analyse revenue or incident counts by different dimension attributes, usually the ones that are used by the business for tracking purposes.

7. Data Warehousing Best Practices

Building and maintaining a data warehouse and an analytical processing environment can be a complex and challenging task. Not only does such an environment house the data you heavily rely on for decision making, but due to its nature, it can be difficult to create an organizational culture for its development. The engagement of many stakeholders, use of many technologies, a long implementation cycle requiring investments, all make it difficult. Therefore, the goal of this chapter is to introduce you to the best practices in data warehousing and analytical processing. While we cannot promise that following these practices will guarantee success, we can promise they will improve your chances of success. The chapter focuses on techniques and strategies that span multiple phases of the data warehousing process.

We start with data modelling techniques that define the way your data is represented. These techniques build the foundation of your data warehouse. Next, we present techniques that optimize the performance of your data warehouse or analytical processing environment. The more optimized your environment is, the better the experience of your users will be. Finally, we present the importance of data governance and data quality within a successful data warehousing process. A data warehouse houses the data assets of your organization and thus being able to be trusted in their integrity is what separates a data warehouse from other digital storage spaces.

Every major data warehouse system has its own management and operations tools, which you should use alongside the best data warehousing practices. These tools help you perform many of the functions involved in managing and maintaining a data warehouse. This chapter introduces a few widely adopted data warehousing practices, such as developing a business glossary, building a data model, monitoring pipeline status and standards, and placing published reports on a BI portal. Here, we discuss several other general best practices. To that end, this chapter answers several questions:

■ What data modelling techniques work best?

■ What performance optimization strategies are commonly used?

■ What data governance and quality techniques can be useful?

Data Modelling Techniques

Typically, a data model presents a graphic representation of the data warehouse entities and their relationships. A data warehousing-friendly data model should fulfil various criteria. One such criterion would be to describe the intended structure of data and its content, so that you could examine if the model captured what was intended and whether it met the requirements. Other criteria include ease of use and understanding, and ability to be matched against data or database structure, among others. A data model should also help recognize gaps or missing elements in planned data systems, helping to define and consolidate process-related, data flow, and transition phases and timing. The modelling technique and data presentation employed should also contribute positively to the data and database maintenance tasks.

7.1. Data Modelling Techniques

Data modelling techniques provide a formalized representation of the data warehouse structure, along with taxonomy, data types, data relationships, and the semantic meaning of data. Data modelling techniques can be broadly classified into four categories: canonical models, conceptual models, logical models, and physical models. Canonical models define a generic data representation for multiple sources, whereas conceptual models provide high-level abstractions depicting the primary data and the relationship between the different data items. Each canonical or conceptual model can be elaborated into a logical model. The logical model captures the data representation associated with a particular project or system and is not concerned with how the data is structured in a physical implementation. A physical data model is a direct representation of a logical model constructed for a particular implementation environment. The physical model is DBMS-specific and represents the actual logical data structures created in the database.

Data modelling for data warehouses has its own particularities. The first issue is whether the data modelling language provides for both data structure and semantics. The second issue is whether data engineering and data delivery is conducted in a top-down or bottom-up manner. Conventional operational data stores and applications follow a top-down technique. The original entity-relational models for data modelling were themselves designed with business operations in mind; thus, they capture well the operational model of an enterprise. The top-down approach commences with the development of data models for all the operational processes of an enterprise. Emphasis is placed on the attributes of the different entities; a process is but a source and sink of events associated with a time attribute. Conventional data models and modelling tools work well in this scenario.

Data warehouses are typically built using the dimensional model, but other modelling techniques also exist. We start with the dimensional model and then present some alternatives, including the inverted model, the Data Vault model, and the anchor modelling technique. Dimensional modelling techniques such as star schemas, snowflakes, and galaxy schemas are also discussed. We end this section with a discussion on the "no-model" modelling style advocated by many data warehouse practitioners.

A dimensional model is a database structure optimized for Data Warehouse use. It usually consists of both facts and dimensions. Facts are the quantities of our business that we want to keep track of. They are usually numbers that can be broken down into smaller parts, such as sales revenue, the number of items sold, quantities shipped, and so on. But a fact table also contains several foreign keys to dimension tables. The dimension tables contain attributes that add context to the facts. Examples might include the product being sold, the location of the sale, and the time of the sale. The combination of these facts and attributes gives us the numeric quantity along with the explanatory context we need. The dimensional model is a true data structure designed for the purpose of providing users with easy access to their business data for queries and analysis.

A dimensional model frequently has a star shape to it, but dimensional models can also be snowflake-shaped or galaxy-shaped. A Data Warehouse may contain so many different dimensions that two or more different fact tables are needed to keep the system organized. A dimensional model can also include slowly changing attributes, those attributes in dimension tables that may change periodically. Different techniques for addressing this include using separate dimension tables for the changes; storing historical records in the same table; creating additional columns to store different historical records; and flagging record with a "current" column. Star and snowflake models support all of these techniques.

7.2. Performance Optimization Strategies

Data warehouse applications are performance sensitive. The data presents an alternate view of an application area and is used for executing long-running batch jobs to extract and generate knowledge. The system requires a read-only access model and frequently supports many concurrent users. Disk storage is expected to be very fast, since most of the design approaches are based on security from I/O bottlenecks. This has forced the use of very large disk caches to accommodate most of the active disk working set in memory. The queries to the warehouse are usually expensive. Thus, data warehouse system design is focused on either speeding up long-running batch jobs or speeding up the response to requests possibly by several users in parallel.

Performance optimization techniques usually fall into one of the following four broad categories: appropriate physical designs, cached and indexed data patterns, query rewriting and optimization and storage design. Database performance improvement attempts to improve the performance of the physical language methods in procedural language interfaces. The method removes unused parts of the database to speedup response time by producing a simplified or smaller database, thereby permitting the lower-level code and index techniques to operate with higher efficiency. Query optimization of natural queries is driven by ease of query specification. Common natural language and graphical query interfaces operate at a higher level of abstraction than the physical wrapper language offered by the system. User maps showing shows and perhaps even query session history can greatly help speed up the generation of fast solutions.

Data warehouses are designed to be read-intensive environments, in which queries can be run at varying levels of complexity. Because data warehouses are highly structured, with a multitude of relationships between structures, they can be optimized to provide good query response times. In turn, optimizing performance for heavy analytical and ad hoc workloads presents its own unique set of challenges. Given their specialized workloads, data warehouses generally benefit from different performance strategies than online transaction processing systems used for transaction heavy workloads. The performance strategies that we describe in the sections below are just a few of our favourite techniques for optimizing a data warehouse.

Considering the high volume and variety of tasks associated with running a data warehouse, establishing and maintaining a balanced and optimal design for environments that support workload performance requirements is a significant challenge. Considering that there may be many diverse schedules and types of workloads that affect key objectives, including query performance, query optimization, resource utilization, schema design, structure incidence, index selection, and data partitioning and distribution, we discuss some of the most important performance tuning strategies for operating a data warehouse.

Most common consolidation techniques use hardware system characteristics that leverage the capabilities of a machine that can support multiple partitions. Examples of such techniques include optimizing computational resource performance for query initiation and execution, monitoring the data warehouse for query performance as well as key utilization statistics, isolating and scheduling key metrics with potential performance impact, partitioning the data warehouse with computing capacity partition and query workload utilization levels, tuning data warehouse architecture as well as resource monitoring and utilization scheme characteristics, and storing historic values of resource utilization, database load, and data warehouse query performance to produce data warehouse performance curves that illustrate the potential impact of physical changes.

7.3. Data Governance and Quality

Equally to the importance of Data Warehouse availability, there is also an alignment that can be made with the quality of the information being delivered. In effect, quality issues in source systems will always be reflected on the information delivered by the data warehouse. In this sense, a databank can be a useful tool to map what this information is, what is the state of the information in source systems, the operating schedule and the error recovery processes defined for each data mining process. The databank can support processes of information quality classification, as well as the maintenance of information quality metrics that allow for the monitoring of DW quality. The quality of the information also depends on privacy and security controls over information for which data protection and information access processes are established. Therefore, it is possible to define a database with control metadata.

One of the information services crucial for data warehousing is the Information Catalogue, which is a metadata database for all information stored in the DW and its sub-schemas. The catalogue allows users to know what information is available, in addition to helping them explore the structure of the data warehouse system. The catalogue typically also provides data source and access information. By itself, it would not answer questions like 'What is the semantics of this attribute?' nor would it provide the semantic conversion related to the semantic translations. It would simply give a list of all the attributes. However, it can be used to help users figure out 'What is the topic of this attribute?' The mapping metadata for attributes in the DW would refer users to the Metadata Management System for more detailed coverage.

Data Governance and Quality Data governance spans both policy and process to deliver and maintain actionable data for an organization. Data is viewed as a strategic asset, and proper data governance ensures its proper use, quality, documentation, and lifecycles. In a typical organization with several data sources and domains, the responsibilities of data governance are usually distributed as follows: First, business governance, conducted mainly by the business users, sets the business policies for the proper use of data in identifying and addressing industry mandates and business processes. Second, data ownership, performed mainly by business management, steers the implementation of policies provided by business governance and defines the SLAs of any data delivery in terms of data accuracy, consistency, data-source signal-to-noise quality or reliability, and data-outage value impact. Data owners inspect mission-critical data and provide for its maintenance, endorsement, and change control and approval process. Third, data quality oversight, performed mainly by the data governance or data management teams, coordinates and establishes controls for the enforcement of SLAs over all data flow processes until data is consumed. Fourth, data stewardship, which usually augments the oversight function of data governance, is usually executed by domain and technical specialists from different departments. Data stewards collaborate with data users to help them understand data definitions, data acquisition, and refresh frequency, along with providing estimates for data flow insights of data accuracy, consistency, completeness, reactivity, and timeliness.

8. Future Trends in Data Warehousing

The field of data warehousing and analytical processing has been maturing for a long time. Today, enterprise data warehouses store XML, spatial, text, and different types of document data along with all the relational data. New techniques and technologies are capable of processing data coming from sources like clickstreams, sensor networks, automobiles, and RFID. Some of the current trends include self-service data preparation and business intelligence, on-premise versus cloud-based DW and business intelligence services, in-database analytics, intelligent procedures in DBMSs, easy-to-use data science with Auto ML, real-time or near-real-time stream processing, integrated big data and enterprise data warehouse systems, DW and business intelligence support for large and unstructured data stores for accessibility and scalability. In this chapter, we will focus on innovations that will extend the overall capabilities of data warehouses and their use for analytical business processing.

8.1. Cloud-Based Data Warehousing

The most significant development in DWs in the last few years has been their migration to the cloud. Organizations no longer wish to build large data warehouses in-house. They prefer software as a service DW solutions. Cloud SaaS services that provide hosted DW solutions offer traditional data warehousing and recently developed big data and AI/ML capabilities.

The concept of cloud-based data warehousing refers to the hosting and maximization of data warehousing technologies, tools, and services through the cloud. With it, organizations can utilize several data warehousing services on top of cloud infrastructure resources offered by service providers. Only a few seek to operate their own cloud-based data warehouse; the vast majority employ data warehouse service providers. Their data warehouses effectively run on the service providers' cloud platform, using their data warehousing infrastructure. One of the first cloud data warehouse services was offered by a major provider in 2008, based on a hosted version of an open-source RDBMS. Several others now compete with it, including other major providers. These and other cloud service providers now maintain and operate large-scale cloud computing platforms that provide, on-demand, secure access to a shared pool of configurable computing resources.

The increasing volume of data generated and stored by different organizations opened new dimensions for data warehousing technology. On-premises deployed data warehouses started reaching their limits in terms of flexibility and analytical workload performance, mainly due to their rigid scalability. In addition to that, their high initial costs are presenting a major challenge for small and medium-sized enterprises or startups which are trying to take advantage of analytical processing technologies in order to get faster and better-informed strategic decisions. Cloud-based data warehousing allows for a faster implementation, at a much lower initial cost, while providing a pay-as-you-go model for handling the variable end-user demands regarding capacity.

These limitations in terms of the adoption of data warehousing processes contributed to the rising popularity of cloud-based storage systems. Other NoSQL-based, cloud-stored storage solutions, although having other advantages, are not capable of providing a solution for classical data warehousing tasks, like analytical query processing support for business intelligence. In a cloud data warehousing system, although data is stored in cloud systems, the analytical processing of data still requires the operational support of traditional processes for data extraction from heterogeneous systems, data cleansing, and data transformation into a star schema model and resorting. While the processing capability is strongly scalable, governed by the workload applied to it, this does not have the same level of scalability for the analytical query processing workload. Cloud-enabled data warehousing systems have started to provide massively parallel processing architectures that allow for much better elastic scalability.

8.2. Real-Time Data Processing

Exploding interest in fast data i.e. data that is transient and needs to be quickly ingested and kept, has led to an explosion of technologies, advancing analysis of this data in a faster time. Technologies have come up to support ML-based approaches for simple ETL data processing.

In the years to come, data warehousing is expected to witness unprecedented growth. Several emerging technological innovations are already in process of redefining the data ecosystem. While unconventional models such as data lakes and data as a service are being touted as substitutes for traditional data warehousing, they require certain specialized conditions for consideration as surrogate. In fact, cloud-based data storage architecture, with its various advantages, is fast becoming the de-facto standard for data warehouse technologies. Complemented by data processing as a service offering powered by machine learning, the coming years are bound to witness a phenomenal expansion in the amount of data that organizations will manage as well as the ways in which they will leverage it to their competitive advantage. Marching towards the decade mark of data miniaturization, companies are increasingly leveraging data from an ever-broadening array of transaction data channels. Driven by the advent of the Internet of Things, organizations are expected to not just own massive data pools but are also expected to act responsibly in terms of the governance of this data. As part of their corporate strategy organizations are likely to implement plans for ethically responsible practices around the monetization of their own data as well as data collected from consumers. With data as a currency of the future, it is only natural for enterprises to escrow the right to dispense such currency units on the data services technology partners they choose to work with. Cloud vendors are racing to introduce nascent self-service business intelligence, data virtualization and machine learning and artificial intelligence driven predictive-as-a-service technology offerings as they meanwhile scramble to strengthen data governance capabilities to be services in synch with enterprises' evolving citizen developer models.

A primary motivation for the creation of data warehouses was that analytical queries were not well supported in operational systems and that these queries often exerted a large performance impact on those operational systems. However, both operational systems and data warehouses have many applications where data is needed quickly for reporting, analysis, or operational execution. The time frames for this need have historically had a large overhead due to batching routines that extract data from operational systems and place it in data warehouses. In recent years, there have been many new technical developments that have reduced this time frame from hours to minutes, or seconds, or even less. These developments include specialized tools for near real-time extraction of operational data, data replication technologies, the increased use of data marts, which pit processing load on the warehouse against processing loads on the operational systems, and advanced technologies for speeding up the loading of data into the warehouses or making the loading processes more incremental.

The result of these developments is that many organizations are able to supply near real-time data to their data warehouses for both reporting and analysis. Furthermore, many organizations increasingly require real-time data that does not go through a data warehouse at all but instead goes into operational analytic processing systems that are supported by the same types of query workload optimizations that are used in data warehouses. To meet these needs, companies have developed many general techniques for supporting real-time analytic systems that process business transactions and generate real-time reports and analyses.

There is a growing need for real-time data analysis, which affects both the architecture of data warehouses and the ETL process. Real-time access to data and real-time analytical data processing will change the research and development trends in how we develop data warehouses. Information systems are evolving from hierarchical and relational systems, based on transaction-level normalization and integrity, to multidimensional systems that offer a view into the collective knowledge of an enterprise. Traditional transaction-oriented databases focus on day-to-day activities. From the organization's operational perspective, this is critical but limited to present-focused, biased data. Much of the information in such a database is not useful for future planning decisions after a limited time. In contrast, multidimensional databases facilitate data analysis over years for decision support. Data is usually read-only, based on a high degree of denormalization required by user needs, and subject to optimization for space and speed.

ETL Updates and Load Schedules. As organizations strive to perform business operations as close to real-time as possible, so does the accompanying desire to have the ability to query the most up-to-date version of the data warehouse. Thus, the schedule of data extraction, cleansing, transformation, and loading into the data warehouse becomes a more pressing issue. Cleansing of data will still require a good deal of time before actual loading occurs, but the period of unleashing the data warehouse to loading for historical data may decrease. The loading required by real-time data warehouses may impact the OLTP systems that feed them because of the contention for resources used for both data entry and ETL. Thus, data warehousing is not without its list of issues that will need to be dealt with now and in the future. The rapid development of data warehousing environments brings with it a unique combination of challenges and opportunities desktop-based solutions cannot address.

8.3. Artificial Intelligence in Data Warehousing

Artificial Intelligence (AI) refers to the ability of a digital computer to perform tasks commonly associated with intelligent beings, such as learning, solving problems, and perceiving. Machine Learning (ML) refers to a sub-field of AI that studies and designs computer algorithms that can improve their performance given a set of data. Data Warehousing (DW) is the technology that enables the extraction and transformation of operational data from deep within the organization to populate one or more repositories with structure and content suitable for analytical processing. Data warehouses (DW) and other Data Management (DM) technologies are researched and designed to enable dimensional and multi-dimensional models that provide currency, structure, content, and orientation to functions such as Online Analytical Processing (OLAP) and enables timely and accurate decision making for business executives in large business enterprises.

Due to the enormous amounts of data generated by organizations and individuals today, the development of Data Warehousing (DW) and related Decision Support Systems (DSS) is of great interest. However, the increase in data volumes, the improved global connectivity afforded by the onset of mobile networks, the need for increased page loads, and the greater use of richer content will mean that Web scale Data Warehouses, Optical-Based Digital Hierarchical Storage Management Systems (DHM), massively parallel processing, and associated Data Management (DM) solutions will need to adjust and adapt to the needs of the New Knowledge Economy. This paper will examine the component processes involved in DW, Online Analytical Processing (OLAP), and DSS, and present some future trends related to the current state of the art for this enabling technology.

The Data Warehouse (DW) is designed to provide a unique Multidimensional view of decision support data. However, the Multi-Model DBMS catalogues the schema on write at transaction time; whereas the DW schema is modelled on read, and DW loads take three key steps: Extract, Transform, and Load (ETL), followed by a batch schema management operation, if required. Thus, DW may take longer to ingest data. Further, the OLTP workloads of large scale OLTP DBMS support are now also quite diverse and include support for high read and/or write concurrency. At the same time, DWs still rank as the most important enabler of e-business, Digital Business, and Big Business Intelligence.

The role of artificial intelligence in data warehousing is twofold. On one side, AI enables a new class of smart data warehousing solutions that embrace automated databases, augmented analytics, autonomous data engineering capabilities, and preservation of knowledge through the lifecycle of data. On the other side, data warehousing remains an indispensable ingredient to success for all AI initiatives based on the combination of AI efficiency, affordability, and computational performance, asset liability for a time, data-centric investment.

Many traditional data warehousing solutions were not designed to handle advanced analytics and machine learning workloads. Increased workloads have increased costs. Smart data solutions help data engineering teams be more productive by automating tasks that consume a significant amount of infrastructure and manpower resources. Smart data warehouse solutions are still behind the anticipated adoption rates. Adoption has been slower than expected. This is partly due to the checkered history of machine learning and AI. Most data engineers are excited at the promise of these solutions.

Moreover, while the headline features are exciting, the actual implementation of these features is often without substance. They do not fundamentally change the engineering burden, nor the ability to deliver trusted data at scale. We have heard numerous data warehouse requests. We have also seen systems that have safeguarded that space, are hedging bets, with cloud and on-prem. More than the automation, it is about understanding the problem set that AI is helping with, solving the implementation challenge of one-click enablement and ongoing lifecycle management.

For completely differentiating offerings, we expect some leading-edge cloud data warehouse players to take the routes to simplify implementations, but also delivery speed and turnaround to enablement of explained ML features. The second area of differentiation is the equation of Data Warehouse and data lake in the hybrid landscape of cloud-native data pipelines to enable key workloads like experimentation with explainable ML.

9. Conclusion

We focused our survey in this chapter on what we consider to be the most distinguishing characteristics of data warehousing and analytical processing.

What distinguishes data warehousing and analytical processing from other databases and data management applications is, we believe, a combination of three key features: The notion of the data warehouse as a central repository for integrated, and non-volatile data, the interactive, exploratory, user-driven nature of data analysis and the high-performance requirements of decision support systems. In the past decade, there has been an explosion of interest in data warehousing and analytical processing, both from the academic research community and commercial vendors. There has been some research on the architecture of DSS tools and data warehouses, culminating in a new technology called the data warehouse. The major vendors in the relational database space have invested heavily in new products to enhance the performance of analytical processing. In addition to enhancements in current commercial and research systems, there are new tools dealing specifically with the extraction and transformation of data, systems using directories for efficient management and retrieval of models for on-line exploration, and integrated environments for query optimization, workload management and resource allocation.

In conclusion, the projects we summarized in the various subsections of this chapter are not solutions for a specific problem that fall into neat packages; on the contrary, they provide building blocks for specific solutions to specific problems. They also provide design techniques for at least some of the various components of a data management solution for DSS applications. Data warehousing and analytical processing is a relatively young area of database research and development, but it is already a rich area, and no doubt will become richer as the field matures. We believe that the next decade of research will create exciting experimental systems, solve more of the open problems we outlined in this chapter, and better bridge the gap between research and commercial products.

References:

[1] Inmon, William H., Derek Strauss, and Genia Neushloss. *DW 2.0: The architecture for the next generation of data warehousing*. Elsevier, 2010.

[2] Ponniah, Paulraj. *Data warehousing fundamentals for IT professionals*. John Wiley & Sons, 2011.

[3] Devlin, Barry. *Data warehouse: from architecture to implementation*. Addison-Wesley Longman Publishing Co., Inc., 1996.

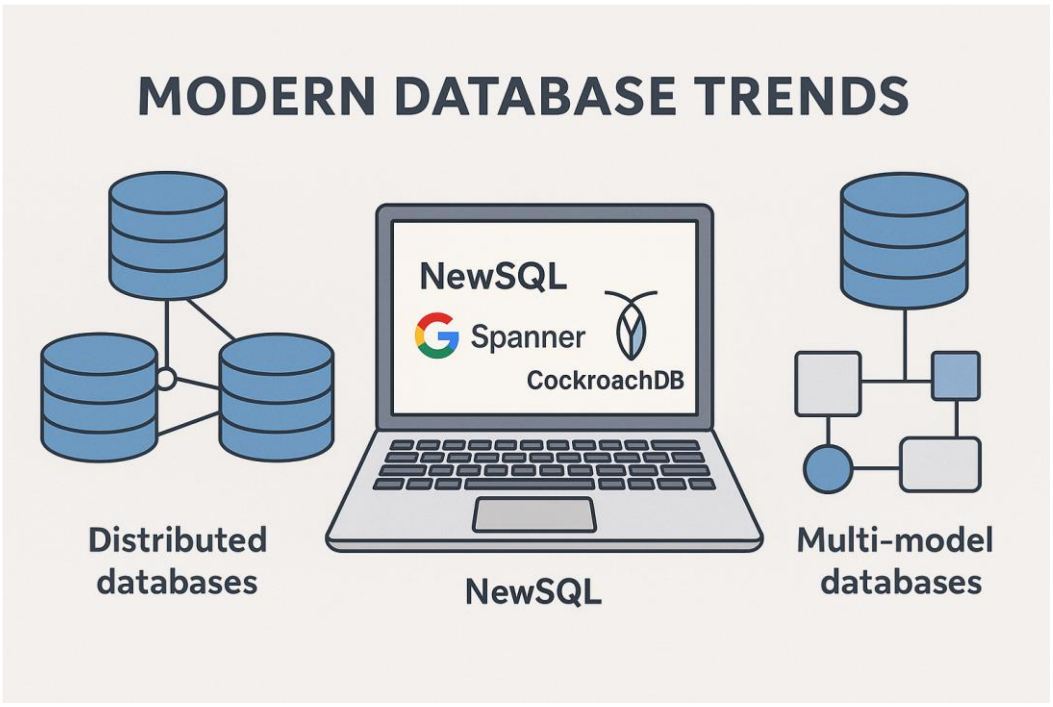
[4] Widom, Jennifer. "Research problems in data warehousing." *Proceedings of the fourth international conference on Information and knowledge management*. 1995.

Chapter 8: Modern Database Trends

1. Introduction to Modern Database Trends

The term database is a bit more complicated than we think. Many of us store computing data in a table or associate specific variables with specific values in multi-dimensional associative arrays. Data in some shape and form also exists in our e-mails, HTML pages, web search indexes, thick client applications, and even in distributed storage systems. Yet, by and large, we recognize only a section of these storage systems as databases, mainly because they support something we call database management systems. A DBMS provides the user with a uniform interface to the underlying physical storage, regardless of the way a particular data item is stored within that physical storage layer.

The history of databases started in the 1960s with the original work on hierarchies and networks, and the advent of commercial and academic database systems, based on the relational model. Principles and landmark papers established a playback for future operations in the field of query optimization, functional dependencies and normalization, transactions and concurrency, and indexing. Today, the mainstream of modern DBMS implementations revolves around four major concepts for enterprise data management. The first concept is that of a relational query language that allows users to specify answers to specified questions without having to specify methods for answering them. The second is a mathematical model of physical data organization based on logical data independence, which is unique to DBMSs. The third concept deals with the control over concurrent, distributed access to data in presence of network partitions and system crashes. The last major concept is a hardware model based on magnetic disks and buffer pools, which are unique to DBMSs.



2. Understanding Distributed Databases

Distributed databases allow data to be stored across multiple sites to achieve higher performance, greater availability, and improved reliability than their functions of a single-site database. Most modern distributed databases use a single-site database as a model, meaning that the application software running against the database does not need to be modified to take advantage of distributed functionality. Such distributed databases use a combination of hardware and software-based technology to provide this distribution capability. Historically, distributed databases first emerged as mainly replicated databases with applications that involved read sharing and a small proportion of updates to the data. Using techniques from the distributed systems field, such as various consistency protocols and data distribution strategies, database systems emerged in the 1980s to allow both read sharing and write sharing on distributed databases.

These products evolved into active replicated databases that kept the replicas always consistent, at the cost of decreased update performance and increased complexity. These products did allow some partial queries to be executed using only the local replica at a site, but applications still had to be written so that certain constraints were obeyed to maximize the likelihood of using this optimization. Most of these systems then transitioned into distributed directory-assisted databases in the late 1980s and early 1990s, which were popular with early Web applications that had very high read-to-update ratios. However, as the read-update ratios for many of these applications shifted to lower proportions, the performance of these systems decreased as well. Maintaining the mapping was particularly troublesome as the number of partitions increased. Partitions were also stored in file systems instead of directory servers.

2.1. Definition and Characteristics

While all databases distribute data among other machines, distributed databases should replicate or partition the data in such a way that users are completely unaware of the fact that the data is not physically located in a single place [1-2]. Distributed databases are normally utilized to usher data within various sites to facilitate access and to enable reliability. A distributed database system can be a centralized database with multiple users or a distributed processing system with multiple databases, but to provide transparency, it must be two--a distributed database with one user and a single distributed database with multiple applications. In a distributed database system, users should have a single image of the database across the devices. There are different configurations of distributed databases, but the features that upload one distributed database definition apply to all implementations. The most important characteristic of DDBMS is that it provides a single global schema to access various local databases that may or may not have a single schema unifying them locally. These may be uncentered databases with no single governing concept. The information may be available in diverse formats at various locations. The pattern transaction may require information from various databases combined by the globally available schema. A DDBMS, like an SDBMS, is user transparent when a user request reference.

2.2. Advantages and Challenges

A distributed database provides multiple advantages to users when compared to centralized databases. First, a distributed database has a higher level of availability and reliability as data is replicated across multiple nodes; if a node goes down, the database is still operable, and if one or more of the available nodes also replicate the data proactively, backup copies exist. Second, large volumes of data can be processed in parallel at different nodes. Since distributed databases are horizontal-scalable, they can also easily grow within the cloud and reduce the cost of adding new disk space. Distributed databases provide fault-tolerant properties, especially when data is replicated across multiple nodes in different data centres in the cloud. Fault tolerance is a sign of fail safety, which enhances the quality and availability of database services. When an existing node or data centre goes down, clients can reach the data from a replicated copy in another node. Often, fault tolerance goes hand in hand with geo-replication of data and transactions among different nodes. Other properties that distributed databases are expected to provide include high performance and support for massive data volumes, high transaction rates, and transactions of long durations.

However, distributed databases come with also some difficult challenges. First, distributed databases suffer from high costs associated with insensitive loading and replication of network traffic during high-volume peak periods of transactions. Second, the consistency problem presents a major obstacle to providing ACID transactional properties across distributed nodes: ensuring that the same value is returned for read operations on the same database object by different transactions when a concurrent write operation occurs. Beyond the performance aspect, data design is also a difficult problem: avoiding data replication and establishing a proper replication scheme is complex. Third, automation and consistency design are major obstacles for full use of the cloud.

2.3. Use Cases and Applications

Many applications can be found in the Internet domain, money transaction services, cloud and online services, network services, and continuous data services [3-4]. Some of their data are stored in MySQL and, some of their data are stored in various NoSQL systems. Also, a heterogeneous DBMS system is used for metadata stored in MySQL, and in the NoSQL world. The translation of the data from one DBMS to another is made by the services offered by a certain system. In the environment of the small and medium enterprise, some DBMS vendors have offered their own heterogeneous DBMS solutions. A certain vendor offers SQL Server and provides remote Data Access services for small and medium size enterprises and many others.

These environments give more flexibility to companies by building better and more appropriate DBMS systems to their needs. The demand for bigger and stronger databases is fundamental. And the use of NoSQL beside traditional RDBMS is a solution that is becoming widespread all over the world, especially due to the need for low-cost, high-performance solutions. If previously, many companies were very restrictive in allowing the use of RDBMS beside their own main system was used, today things are changing dramatically. The necessity for horizontal scaling and NoSQL environment no-structure or low-structure databases no longer die. Faced with this challenge, various new vendors, old DBA tools vendors, and big RDBMS vendors are working in this perspective of acceptance and vertical integration between the two worlds.

3. NewSQL Databases

Overview of NewSQL Traditional SQL databases cannot keep pace with the high-volume and high transaction velocity of current scale-out web applications. The only kind of database that achieves good performance with such applications is NoSQL, a class of databases that have simplified SQL over the centuries – they favour availability and partition tolerance over consistency and on-line analytics performance over transaction throughput. However, as NoSQL adoption increases, it is apparent that "more SQL" in areas of consistency, transaction guarantee, and analytics performance is highly desirable for many user communities, including banks, online brokers, retailers, etc. For these user communities, NoSQL's limitations, including inconsistency during updates, eventual consistency, lack of on-line analytical processing, and lack of tools for programmatically expressing and executing analytic queries present serious problems.

3.1. Overview of NewSQL

NewSQL refers to an emerging class of databases that attempt to provide the same scalable performance for OLTP transactions that NoSQL systems provide, while still under the ACID guarantees of a traditional SQL database. NewSQL systems augment an existing SQL database or are a completely new implementation. Most of the NewSQL systems provide a complete features of SQL, while some may not. Most of them also take virtualization or cloud deployment into account. As is typical with any new technology, the set of features varies widely between the NewSQL offerings. Some of them may not provide a full transactional model but perhaps only some subset or weakened version of that, i.e., isolation levels.

NewSQL systems also embrace a new architectural model, one that is designed to be distributed and that takes distribution into account in any pricing. Traditional databases, SQL or otherwise, require organizations to think carefully about layout and proximity, typically needing a well-designed master / slave relationship within a replicated or sharded configuration. It is very easy to create a NewSQL system by applying a distributed architecture to what would otherwise be a traditional database system. Indeed, there are NewSQL systems that are implemented in distributed systems, turning them into distributed SQL databases or other different Flavors of database systems. Substantial differences exist among NewSQL offerings, both in architecture and feature set. At one end of the spectrum, systems with Data Vault and/or Near-Sync messaging provide real-time updates and reports, enabling system users to operate across live OLTP transaction data. Structured data organizations then have a SQL database in which they can report at any speed.

3.2. Key Features of NewSQL

There are two common characteristics shared by the vast majority of NewSQL databases. First, they all support distributed database architectures and can provide global transactions with scalability. However, some of these databases offer limited support for partitioning data across several nodes by providing it only for horizontal scaling, hence lack the ability to do it for load balancing, global transactions, and data locality requirements. The second main feature of these architectures is that they are relatively new projects; few solutions in this space have been around long enough to be perceived as mature.

A rule that most of the NewSQL solutions obey is that SQL support is something important in their design decisions. Only a few of them do not care about SQL support in their design. It seems that even the NewSQL projects that don't natively speak SQL have at some point recognized that proper SQL support could have given them a significant advantage and had a SQL front-end solution or an implementation of the SQL-like language used by some databases. Those that do support SQL have chosen to entirely comply with it, and some of them prefer to comply with ANSI SQL standards rather than the SQL dialect defined by others.

Concurrency and fault tolerance are also essential concerns. After all, one of the reasons why NoSQL databases became popular was the guarantee of very high availability, and bottom consistency for distributed transactions. NewSQL solutions aim to provide guarantees of a different nature. Most of them comply with linear programming principle solutions to the two-phase-commit protocol reservations. However, some of NewSQL solutions do provide higher availability and lower latency responses than traditional databases.

3.3. Comparison with Traditional SQL Databases

When comparing NewSQL with traditional SQL databases, one major difference is the distribution concept. One feature of traditional SQL databases is their monolithic architecture, which tightly couples server functions to single process-space instances. As a result, traditional SQL databases can only be made fast and reliable with single-instance shared memory, on which atomic commits can only rely for the guarantee of transaction isolation. However, for operational efficiency, such as OS and caching, it is necessary to distribute storage on arrays of servers, with fault-tolerant replication with linked processes. For strict consistency in $$ \leq 2$$, the replication needs to be synchronous, causing a bottleneck when commits go through the master process. Also, traditional SQL databases enable the consumption of only one virtual CPU for single transactions as processes block on I/O. Therefore, the performance bottleneck requires scalable problem solving in a certain range of transaction sizes.

With a monolithic architecture, while traditional SQL databases can scale linearly, they cannot be made scalable and reliable for large transactions, and so they are not suited for huge data-intense applications. The need to alternate the application of distributed transactions needs to alternate between committing to high speed on the network while writing and reading from disks and consider to batch the write and read processes of such distributed transactions. It is often in the transactional systems of large social networks that this need is particularly obvious, where $$ P (a, b, k) : a $$ promotes $$ b $$ for some $$ k $$ of its followers, and $$ P (b, a, k) $$ is the back transaction, as there is a high probability of mutual dependence when both transactions are from the same data.

4. Google Spanner

Google Spanner is a distributed data management system that has received significant attention because of its purported novel claims and because of its scale and visibility. It has been in active use since 2010, and has sustained substantial application load, delivering services in search advertising, YouTube, and other offerings. At the same time, it powerfully implements the traditional SQL transactional access model, while also achieving wide-area horizontal scaling, and roll forward commit and distributed transactions. Spanner also offers support for several extensions to the traditional relational model, including semi-structured data, user-defined types, and schema less design. The reason that Spanner has been able to achieve some of the above claims are that it is a well-designed system development effort and carefully executed effort.

Because of growing demands from its internal application developers for globally extending the availability, scalability, and performance optimization of its services, a decision was made to build support for the desired capabilities as the successor to previous systems. The design effort was first conducted to understand the desired requirements for a data management system, needed features, design priorities among trade-offs, as well as the functional objectives for users and non-function objectives, such as high availability support, ability to operate at high scale, low operational costs, and other properties. The model preferred by application developers was not just simple row store, but also supported columnar, relational, semi-structured, and schema less storage and access features. Funding was required to sustain the execution of the project, and considerable simulation data needed to be generated and presented in order to justify the desired feature set and associated design parameters.

4.1. Architecture and Design

Spanner's architecture and design are critical to providing its load balancing, performance, and strong consistency. In this section, we first place Spanner in the hierarchy of existing database architectures, and then describe the key new concepts introduced in Spanner. We then describe how Spanner uses its concepts to address the challenges mentioned above.

Spanner fits into the general hierarchy of database architecture as follows. At the bottom is storage management, which deals with storing, retrieving, and updating bytes in large numbers efficiently and reliably. Above storage management is data management, which organizes the stored bytes as data structures such as tables and indices and provides higher level services such as replication and recovery. Above data management is query processing and optimization, which translates logical queries into efficient execution plans. The top layer is transaction processing and concurrency control, which provide the isolation and reliability guarantees that are required for a variety of database applications.

Existing databases have caused all four layers to be tightly coupled, making it difficult to introduce advances in other layers. For example, lack of strong physical time sources has caused existing distributed databases to opt for either low-cost, but weak isolation guarantees not supported by Spanner or very expensive two-phase locking. Similarly, lack of efficient timestamp-based transaction processing has caused existing NoSQL systems to abandon the powerful transactional interface. The three design points of Spanner are based on the goal of having a clean architecture that could separate the best design in each layer from those of existing systems. It uses a combination of several interesting architectural ideas. It includes hierarchical storage management over SSDs with a dynamic data placement policy, use of a new query language that extends SQL-like queries with support for query execution over sparse remote index tables, a new form of two-phase commit protocol that is possible because of the use of timestamps for concurrency control, and a fault tolerant and efficient external clock synchronization service.

4.2. Scalability and Performance

Spanner is designed to scale without altering the semantics of the data model or the consistency guarantees offered by the system [3-5]. Scalability is achieved through a data distribution scheme, where data is organized in a structure called an index tree, a specialized version of an external memory B-tree. In Spanner, indexes are not just associated with tables; rather, they are used to index the entire database. Because the set of indexes can be both extensive and application-specific, Spanner's data distribution and partitioning scheme can be implemented as an external memory kd-tree. Partitioning is hence achieved by a method that reduces the sum of the surface area of all the tree nodes. Partitioning is also guided by the data model. Spanner embeds structured, arbitrarily large, hierarchical data items consisting of strings and byte streams recognized by a set of user-supplied sequence definitions. Thanks to these underlying hierarchical structures, querying on such documents can be as efficient as direct database access. Along with hierarchical data structures, Spanner also applies the notion of a secondary index to facilitate searching the database.

As in any design that provides scalability, the Spanner design allows for a potentially very large number of partitions within the database. Besides scalability, Spanner also optimizes for performance by partitioning the indexes according to the common query patterns. When querying data from a given partition, Spanner uses local project/transform/append phases followed by a global collapse phase. An important aspect of index performance is the controlling of the index size during the entire lifecycle of the system. In addition to the partitioning of index items during the taint cycle, Spanner also performs compactions based on the notion of a sequence definition. Each time a user-specified maximum percentage of the index has been deleted, Spanner invokes an external application to iterate overall index items and delete any that do not conform to the sequence definition.

4.3. Use Cases and Industry Applications

The technological landscape of the 21st century has necessitated a deeper understanding of the correlation between databases, use cases, and application requirements. In this section, we will present a use case study that attempts to decompose popular industry application settings and their database system requirements. By correlating important use cases for system requirements, we attempt to derive low-level requirements that are useful to both users building systems on specific infrastructure, and to the developers of the infrastructure stack. Consider a variety of industry use cases: In online retail, companies use database systems to maintain product catalogues and inventories, log user account session and activity, and process user orders and payments. These companies deal with millions of users simultaneously browsing or purchasing products in different geographical locations across the world. They rely on the massively automated backend processing of stored data for success. In addition to their primary services, these companies also use these systems for a variety of internal operations such as data warehousing for reporting and analytical operations, market research, campaign management, order fulfilment, supply chain management, and recommendation engines.

In social networks, popular companies store information about their users, including friends and connections. They interface with billions of users who generate hundreds of terabytes of data every day in trillions of messages, comments, and exchanged status updates about their friendships, relationships, events, and lives. These messages need to be stored, indexed, patterned, and queried in real time. In the information technology and cloud computing world, there are numerous Service Provider companies who aggregate information about their clients' employees, accounts, infrastructure applications, and content. These companies use databases to perform storage, messaging, monitoring, maintenance, and migration of their clients' resources and data.

5. CockroachDB

5.1. Overview and Key Features

CockroachDB is a distributed SQL database that is designed to make data easy. It provides the resilience, scalability, and simple development experience of cloud-native applications. It is said to be built on the foundation of a hardened key-value data store, but with support for a familiar SQL interface and transparent autoscaling, and it handles replication and partitioning automatically. Key features of CockroachDB include distributed ACID transactions to provide snapshot isolation for distributed transactions without introducing roundtrips, efficient execution of OLTP and OLAP workloads thanks to distributed execution engines and matrix multiplication, and full SQL support, including JOINs, Transactions and EXPLAIN, backup and restore.

While CockroachDB provides only the basic features of a true database, it implements these features efficiently in a cloud-native way. For example, scales by adding machines, not shards, and elastic horizontal scaling without an external loader, uses a unique architecture for multi-region clusters, so that local reads from distant data invoke fewer remote calls than a single Region lookup; and uses dynamic, cross-replica data balancing and placement in multi-region clusters, moving data when necessary to maintain a desired level of locality.

CockroachDB projects, organizes, and manages distributed data differently from most data platforms. Its proprietary, distributed key-value pair data model allows for flexibility, data locality, and customizability. In fact, it goes a step further to enable the creation of multi-model data platforms, which can also natively support the storage of documents, graph, and time-series data along with the usual structured data. Depending on its configuration, CockroachDB can also serve as such a multi-model platform. Thus, instead of a document collection or table, CockroachDB creates a database catalogue with multiple key space catalogues for each tenant to store both structured and unstructured data.

5.2. High Availability and Resilience

The resilience of CockroachDB comes from the use of replication and consensus. Data is replicated using a configurable RF configuration and is distributed sparsely using a range router. Each replica is hosted in a different availability zone, which can be implemented on shallow clouds by a user-specified zone map. Pre-defined health-check endpoints allow the orchestration platform to monitor the health of each node, and the built-in protocol allows nodes to be aware of the overall cluster state. CockroachDB uses the Raft consensus algorithm for commits and automatically attempts to recover from failures.

5.3. Comparison with Other NewSQL Databases

What differentiates CockroachDB from other NewSQL systems? It is easy to set up; it runs in a tiny container, and storing persistent data is just a matter of configuring a filesystem mounted by all nodes. You don’t need a well-architected clustering setup to start with CockroachDB. It may also be the only NewSQL DB that supports high availability and seamless scaling when nodes fail; it takes care of all the details. In summary, CockroachDB may provide a better first experience on commodity hardware, in a setting where its performance is sufficient.

5.4. High Availability and Resilience

CockroachDB possesses a unique combination of characteristics that mandate it to be continuously available, but none more than the fact that it was specifically designed to be a cloud service as a key use case. Cloud services invariably suffer from cloud operator and maintenance outages, and they are expected to tolerate those outages. CockroachDB was designed by convolution from a database kernel that possesses an AVZ property and a cloud system-wide availability architecture with the expectation that failures would occur almost continuously. Each individual component of both the database kernel and the cloud-wide architecture has been observed in real database workloads. The AVZ property can be observed in real workloads that once a transaction commences, it should either complete within a short period, fail if it cannot complete, or simply be invisible to any client that is attempting to read data from the database server. The cloud availability architecture has also long been observed in real cloud systems — the overall system can be current at best by reflecting the latest non-faulty component states at best.

CockroachDB adopts the same basic architecture as most of the cloud databases that we have identified: each region to be serviced by the database is assigned a storage cluster made of a set of storage nodes, with each node becoming a cloud virtual machine as a shard of the data storage. Such a cloud-wide architecture is simple enough, but there is a critical question as to whether there is a missing property required by a cloud database service. Cloud application services are expected to be continuously available for access, while cloud storage services are expected to be continuously available for delivery, but cloud database services are also expected to be continuously available for manipulation. Indeed, it has been observed in cloud storage services that data must be continuously available for delivery — i.e., at least one copy must always be current and intact for delivery.

5.5. Comparison with Other NewSQL Databases

In terms of performance, given that both Federated Database and Galera Cluster use synchronous replication, they will show latency for both reading and writing. In addition, the network I/O for both operations will be higher, as all the reads and writes need to be sent to all the nodes in the cluster. MongoDB however does use asynchronous master-sensitive replication, which adds latency to write when the slaves are not in sync but will allow for very low latency for read and write when the slaves are in sync. Compared to MongoDB, CockroachDB can be used in scenarios requiring transaction guarantees, especially when isolation is important. Finally, while support for partitioning is available in most database systems, only Orator supports automatic, semantic partitioning and considers such partitioning as its first-class citizen functionality.

Comparing CockroachDB with other NewSQL systems, it has the benefit of using the PostgreSQL wire protocol and the JSONB type that brings CockroachDB to part with the NoSQL world. Most of the other NewSQL solutions are custom solutions and functions available depend on the implementation. To provide a custom, more-familiar-than-no-SQL experience, NuoDB introduced the concept of a distributed ACID transactional database, providing an SQL based solution to NoSQL. Unlike other NoSQL systems that provide limited schema definitions for their tables, NuoDB allows for creating tables which are fully defined using the SQL DDL commands and guaranteeing an ACID-compliant behaviour. While one could argue that the transaction support by NuoDB for non-acid operations is an additional source of overhead, it allows easy migration of old systems to new database systems without the added complexity of NoSQL systems.

6. Multi-Model Databases

6.1. Definition and Importance

Multi-model databases are gaining more importance in both research and industry. A multi-model database is a system which combines different data models in an integrated architecture but does not necessarily provide a support to widely varied functionality for the various models. The database collectively permits building a dataset consisting of different data types and model formats in different structural arrangements or layouts. The models may be traditional models such as hierarchical databases and standard relational or key-value models or more modern models such as document and graph models. The varying data models may reflect varied structure within the datasets, or otherwise varied structural requirements based on user or application considerations. Or, differing data models may be dictated by varied application needs, such as different models at disparate points in a user's path or journey.

Multi-Model Databases are one of the more recent classes of databases. In fact, during the past few years, they have received significant interest from both academia and industry, and currently, technology and product offerings are available from several vendors. The goal of Multi-Model Databases is to provide a unified environment to manage multiple data models. Unlike a hybrid or polyglot approach that uses separate systems for each data model and integrates them at the application level, a Multi-Model Database integrates different models at the data management level, thereby automatically managing integrations, consistency, and performance tuning. Multi-Model Databases also aim to create a more flexible development and management environment. The ability to use and mix different data models at the application and the data management level provides application developers the ability to choose the most natural and effective data representation for each kind of data, as well as the most efficient programming model for the implementation of the application. Furthermore, Data Model Designer, Data Model Business Owners, and Data Administrator can optimize performance and optimize technology if they can integrate different technologies elegantly and correctly. In other words, Multi-Model Databases allow not only application developers, but everybody involved in the design, development, management, and maintenance of an application to be more productive in their respective roles.

6.2. Benefits of Multi-Model Approach

The primary benefit of multi-model databases is that they allow for heterogeneous data which may require entirely different representations to co-exist without data redundancy. Redundant copies can lead to data integrity issues from concurrent updates on disparate copies. A unified multi-model approach facilitates outcome benefits from economies of scale but also prohibits catastrophe scenarios from the common “single-point-of-failure” issue. Consider the scenario of a large data warehouse that combines diverse facets of a single enterprise. Or consider data that defines product roadmaps correlated to marketing data sets for demand forecasting and correlated to supply chain data sets for component availability tracking.

6.3. Benefits of Multi-Model Approach

There exists a multitude of data models, each optimizing its capability for a certain data type or a use case. For instance, while a relational model is excellent for relational data, it is inefficient for graph data processing. A JSON document can better represent a web page due to the unstructured tags. However, with the growing popularity of NoSQL databases, which thrive on multiple data models, the traditional single model databases are losing their appeal. The recent emergence of multi-model databases is inspired by the capability of NoSQL databases to handle multiple data models while also supporting ACID transactions like relational databases.

Multi-model databases dynamically change their data model at runtime. This contrasts with a typical relational database that defines its data schema when created, and a NoSQL document store that represents data as hierarchically structured documents. If an application requires data across various models, it needs the data from different database systems, using the appropriate query. This adds to the complexity of the application logic since it needs to manage the interactions with different database systems. Multi-model databases simplify this effort by consolidating multiple models in a single database. Applications taking advantage of a multi-model database operate upon a closely knit schema across multiple models without having to deal with multiple systems.

6.4. Examples of Multi-Model Databases

Several databases support more than one data model, among which are some notable multi-model databases. Perhaps the most widely known multi-model database is Microsoft Azure Document DB, which supports both document and key/value data models, but which also can support a column-family-oriented storage model. Document DB is among the first databases to go beyond just supporting document storage, which is also the most common storage mode for new cloud-based databases being developed.

Another well-known distributed NoSQL multi-model database is OrientDB, which is considered a key/value, document, graph, and object database. Oracle recently introduced the Oracle NoSQL database, which describes itself as a key/value, document, and table database. Recent releases of the Pivotal GemFire database have enhanced its multi-model capabilities considerably. In addition to its earlier support for its native data set format, it now supports a key/value storage model, a document storage model, and a column-family-oriented data format.

Another example of a NoSQL multi-model database is ArangoDB, a multi-model distributed database that supports document, graph, and key/value data models. A more recently developed NoSQL multi-model database is Couchbase SERVER, which supports a key/value storage model, a document storage model, and a native data format that are well-suited for working together; integrating the capabilities of both modes into applications that otherwise would need to use two different databases. Both databases, among many other more specialized NoSQL databases, allow users to define data in a way that take advantage of the special capabilities offered by the specific NoSQL multi-model database. Various advantages of schema-less data definition to application developers are noted. Other databases support both document and column-family-oriented storage models.

7. Comparative Analysis of NewSQL Databases

NewSQL databases have been recently discussed and evaluated from different perspectives. Performance and scalability of NewSQL solutions have been compared with some standard databases. Other works analyses the cost for the development of such systems and, eventually, the user experience. We set goals, performance classification, and results description for each comparative analysis. In the first, we analyse some performance metrics of five of the most known NewSQL systems, published in expressiveness from the SQL standpoint. We will also discuss scaling and availability aspects regarding concurrent and geo-distributed execution, adding some considerations about cloud computing. In the second part we analyse cost, with a complete cost analysis that considers additional aspects, but the cost effectiveness analysis based on normal load, supported by classic subsystems. Finally, we discuss user experience and usability, which recommend research and development choices that may be interesting for industrial adoptions, including testing under real workloads.

Performance Metrics The performance presented are based on the standard benchmark, which has a micro-benchmark that implements SQL statements such as SELECT, INSERT, UPDATE, DELETE, and a small application, all targeting a dataset. In the case of one system, we have also included the analysis of a single micro-benchmark, the SELECT test. According to the results, all systems scale mainly with read operations, while only a few scale well on updates, too. The latency values stabilize after several seconds that is usually application dependent. Virtually all works observe that performance under normal load is not representative of total execution time, but they should at least match the throughput supported few seconds after startup, in the number of typical concurrent transactions.

7.1. Performance Metrics

Databases are used to store a wide variety of data models, including social networks, user sessions, payment gateways, sensor data, and so on. Due to modern applications like smart devices, network clouds, and client-server architecture, the amount of data to be stored is huge, leading to database scalability and scalability problems. One strategy is also using cloud databases, which bring extra cost, privacy, data locality, availability, and regulatory issues. With the growing demand for low-latency transactions and the use of both relational and non-relational techniques, NewSQL databases have gained a lot of attention from the academic community, testers, and practitioners.

Transaction processing systems use a set of solutions, techniques, and approaches to evaluate the performance of completing a transaction in a database. These approaches start with performance metrics. The state of current NoSQL solutions, as well as recently proposed NewSQL solutions, introduces the demand for clarifying the performance evaluation of transaction processing systems in modern database solutions. Workloads should accurately reflect real-world scenarios, throughput and latency should address the dual aspects of scalability and elastic scalability, while TPASS should address both performance peaks and constant throughput. NewSQL Databases promise to bridge a performance gap relation of ACID guarantees from SQL solutions and unified caching for consistency in a distributed environment.

Despite the existence of several database benchmark proposals, there is still no consolidated repository grouping proposed benchmarks, especially for NoSQL and NewSQL database solutions. TPCC for OLTP transactions and TPCDS and Star Schema for OLAP problems have been widely applied for traditional DBMS technologies. In fact, this is the first organized review on benchmarking database technologies. Presently, has heavily based their proposals on TPCDS.

7.2. Cost Analysis

Many database applications are designed to run as a service. This will typically mean that servers are being rented or leased for database use. How many servers are needed? This will vary with the amount of load being placed on the service. The cost of running the various services will also depend upon the features that a particular application is using. Some services are billed according to storage capacity used, while others are billed according to the read and write operations. Some services may charge for connections opened to the database or a combination. Such differing charging strategies make it hard to compare the cost of NewSQL databases, especially when any one database may charge very differently for different applications.

If a considered NewSQL database is a cloud service, then due consideration must be made of the costs associated with choosing such a cloud service. An organization may prefer to run a managed rather than a self-managed service. In such cases, while the cost of the managed service may appear to be higher, there is the cost of maintenance of the database, hiring and firing the database administrators, monitoring performance, etc., which are assumed by the cloud service vendor.

Another factor to consider is whether the workload requires features that only an on-premises database can provide. Certain databases cannot be hosted in a public cloud service. Often, for data justice or security reasons, a database cannot be hosted in the cloud. Finally, does the organization have people who are experienced with cloud services and able to make an appropriate choice? Knowledge of databases and costs alone are not generally sufficient to make a good NewSQL database choice.

7.3. User Experience and Usability

The cost, performance, and scalability of a database system are not the only measures that are important when evaluating a system. User experience and usability are also critical factors, especially for systems used in enterprise applications. The usability of database systems can usually be classified into three categories: application programming interface, application development environment with tools and support languages that simplify the process of constructing an application that uses the database, and language extensions and tools added to the database.

The interfaces and APIs of existing database systems have traditionally been low-level and require a good understanding of the system to be able to write an effective application. Users are required to write code in C/C++ or Java using the API provided by the database vendors, which include low-level functions for making connections, defining objects, and executing queries and functions such as storing or retrieving data. Code quality has suffered by this lack of API level abstraction because a lot of error-prone code is required, and programmers are not as productive as they are when building database-enabled applications using languages and libraries built on top of the API provided by database vendors. The lack of well-defined, high-level database APIs and tools is a major drawback for many of the databases, especially NoSQL databases.

NewSQL and cloud databases have taken steps to address these issues with the development of frameworks and tools such as middle-tier frameworks, APIs, code and templates, functions, and interfaces to support popular languages. Several currently popular middle-tier frameworks hide the complexity of database operations and the original database API, making it easier to build a database application.

8. Future Trends in Distributed Databases

All trends, projects, and developments in the IT area show that requests from users and application developers directed toward databases will increase dramatically soon. The areas of interest are those of improved performance, scalability, availability, and self-managing systems. These requests are not limited to the traditional areas of transactional processing and novel applications that exploit massive data; instead, they encompass a much broader spectrum that includes archiving and processing of massive amounts of data coming from the dynamic Web and sensor networks, as well as support to real-time streaming applications. For this reason, we foresee a reinforced combined interest by users and developers toward data and coding algorithms and middleware's that make relational and non-relational datastores more reliable and with better characteristics in terms of scalability, availability, and self-management.

We also see the expected further evolution of the database area. On the NoSQL side, the triad "big data, more unstructured data, extreme scalability requirements, and massive elasticity" will translate in the obvious development of more efficient and smarter coding for Map-reduce, column-store, and document stores. The relations with external unstructured data will further improve through more sophisticated techniques for the definition and extraction of useful information. Moreover, we expect further development in the areas of automated web services and quality of services for distributed applications.

On the NewSQL side, there will be a trend back to supporting normalized data models with traditional relational schemata. Application developers will request a return to the full set of relational characteristics, including foreign keys, standard query languages with unrestricted power in retrieving and filtering data, and integrated transactions that control the updates of single or multiple data fragments for data-centric applications with stringent data integrity needs. The response of systems developers will be to answer the requirements for vertical and horizontal scalability on cloud resources with servers for transactional processing and near-line operations as targets.

8.1. Emerging Technologies

Several exciting technologies are emerging. First, the ability to store and process large data sets with low or no cost has the potential to vastly change the big data and database landscapes. Storage services on massive amounts of data on a rental basis, using cheap commodity hardware, are becoming more common. A parallel processing framework for processing massive amounts of data across many computers in a fault-tolerant manner has been developed. Add to this large commodity computer clusters, very fast collocated databases, and software to allow users to express simple operations on massive data sets using a programming model. Certain types of scientific workloads, such as analyzing gene sequences, may be "dropped in" to this environment to take advantage of its capabilities. As this infrastructure becomes available, new companies may emerge who could use the capabilities of these back-end services to offer innovative database and big data services.

Second, we are witnessing an explosion of interest in NoSQL database technologies. Both by accident and by design, application and database developers are innovating in areas in which traditional relational approaches cannot compete. Massive Web and other application data sets need to be collected and manipulated in ways that are traditionally used in data warehousing, but usually outside of the need for transactional integrity, high levels of concurrency, and standard SQL. The column-store approach works well when it is hard or impossible to determine the exact loading and querying accesses. Many other

NoSQL systems are also springing up. NoSQL database technologies are changing the rules for how we deal with big data and what is practical and useful.

8.2. Predicted Developments

Founded in the 1970s, distributed database technology has matured tremendously to become the fertile ground from on which the ideas of cloud computing and big data have sprung. This is a reasonable predictor of some of the further development directions of the enabling technologies. In some cases, more abstraction layers will be added. In other cases, the original ideas will be reformed and revisited on the cutting-edge challenges posed by cloud computing and big data. Indeed, the increased availability and commoditization of cutting-edge distributed hardware technologies, including distributed query processors, justify a reconsideration of long-standing principles in data management.

The biggest impact will likely come from the seamless envelopment of the disparate layers of data management in a unified service layer. Fundamental pieces of middleware and building blocks will be made available as cloud services. These include entity resolution, holistic indexing, integrated models of data processes, storage systems and algorithms for diverse data models, near real-time materialization and updating of models and schemas over evolving data, sophisticated language facilities for language agents, and so on. This combination of fresh developments in long-established fields, size of data volume and velocity, broad diversity of data structures and formats, and use of distributed and cloud technologies for both data storage and storage processes creates a magnificent opportunity for near-term and long-term advances in the science of data management.

8.3. Impact on Data Management

The new database technologies and products that are examined in this chapter will change the DBMS technologies that data management people use. They will also change, to a lesser degree, the applications that data people interact with. The greatest change will be for distributed databases and for multi-model databases. Distributed databases have not been a commercial success due to ease of management issues. Built-in management and automatic optimizations are beginning to make these products easier to manage. Multi-model databases have had an initial impact in supporting data types and data models not supported by existing products. The new multi-model technologies will extend these early efforts to better support integrations of different models and types of data. As a result, the new systems will hold advantages over existing products.

NewSQL databases will primarily add speed to the existing SQL technologies. This impact is somewhat mitigated by the clamouring for faster NoSQL solutions by OLTP customers. Data scientists will be exploring the balance of capabilities between these technologies and what constitutes an optimal design for various use cases. As with most DBA tools, tooling aimed at no-code and low-code development and analytic efforts will remain agnostic to the underlying technologies and will continue as an untouchable industry.

Finally, as with every shift in technology and business, it becomes the proper task of the IT, business, and data stakeholders to evaluate the current pressures and issues that promote the vendor and product shifts. As new solutions emerge, the task returns to the Data Management team and the interview stakeholders to evaluate where and how the shifts can translate to improved processes and business value.

9. Conclusion

A new approach to applications is taking over the way we develop and architect solutions to business needs. Distributed architectures are gaining momentum with a new breed of distributed databases. Partitioning data, namely sharding it, on the application level was the only possible approach for many applications in the early years of the Internet. Next came specialized servers with load balancers in front. Work is being sent to many database servers, but they each hold a portion of the data. Then came high-availability databases with replication, to solve the replication problem. Even though the database technology at the service of the applications has been in this state for many years now, the surge of new applications and the low cost of hardware is pushing databases back into the spotlight.

History tends to repeat itself and this is what is happening again in the database universe. Data has centres of interest. Each application runs on its own application and database tier. Databases become key single points of failure in the applications. With the new environment of large companies, the new breed of database servers is efficient enough to spread the load of a large number of applications and to depart the data without having DBA sculpting partitions as starts to be done to relational databases in the real enterprise. Because we have forgotten NoSQL these past years, we are prepared to see it surpass the traditional databases. The challenge of those new databases will be to provide a SQL-like interface, with all the available features of optimizing execution and robustness provided by the relational engines. The success of those new databases will not only depend on performance, but also on the capability of addressing the new application models and the ability of easy internalization by developers and architects.

References:

[1] Bernstein, Philip A., et al. "Query processing in a system for distributed databases (SDD-1)." *ACM Transactions on Database Systems (TODS)* 6.4 (1981): 602-625.
[2] Bernstein, Philip A., and Nathan Goodman. "Concurrency control in distributed database systems." *ACM Computing Surveys (CSUR)* 13.2 (1981): 185-221.
[3] Corbett, James C., et al. "Spanner: Google's globally distributed database." *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013): 1-22.
[4] Han, Jing, et al. "Survey on NoSQL database." *2011 6th international conference on pervasive computing and applications*. IEEE, 2011.
[5] Guo, Zhihan. "How to Design Disaggregation in Large-Scale Transaction Systems."

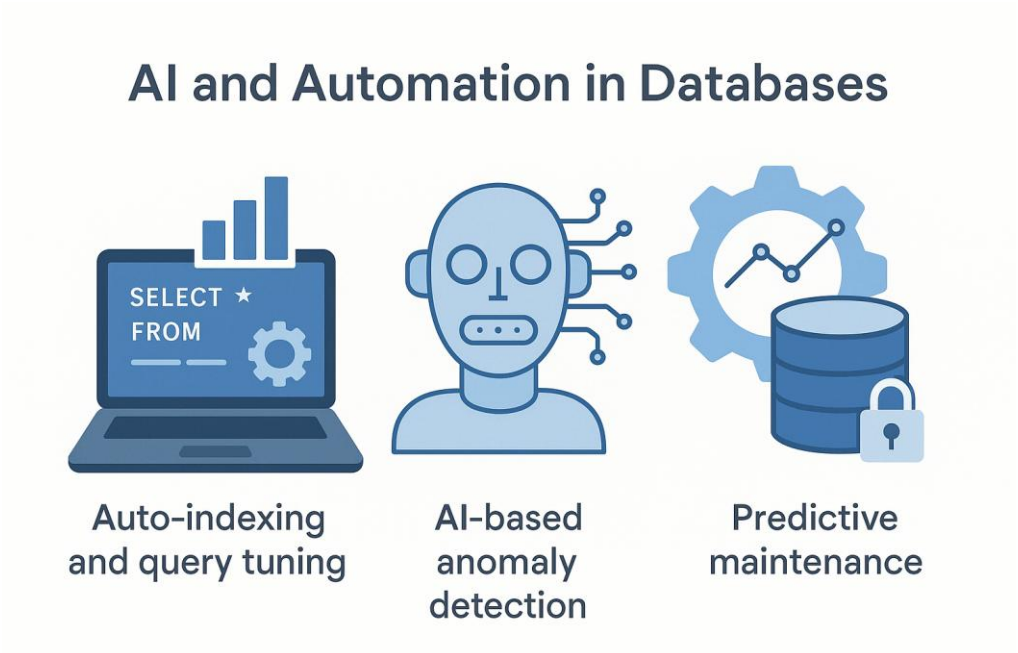
Chapter 9: Artificial Intelligence and Automation in Databases

1. Introduction to AI in Databases

AI has emerged as a leading technology in recent years, and it continues to deliver advances in multiple areas across science, engineering, and societies [1-3]. However, such advances in AI would not have been possible without the use of databases and, in general, of Data Science. Most of the AI models that provide those advances require a significant amount of data to be trained and applied to be useful. They also rely on multiple areas of Data Science, including efficiently acquiring data from diverse sources, data cleaning with the use of Data Warehousing, curating or annotating the data, and performing analysis on data, making knowledge discovery possible using different forms of Data Analytics. And databases are not just sources for data used in AI, but they are also essential tools for storing and managing data generated by AI processes. However, even if there has been a major interest in the use of AI for Data Science, there have not been many studies on the use of Data Science for the development of AI models to further automate its different processes. Among the ones that have provided a comprehensive view of this latter area, this essay intends to focus mainly on those related to databases.

In summary, the two areas of AI and Databases, apart from relying on each other, are also becoming increasingly interrelated and more dependent on each other. While AI is enabling more automation and assistance within Data Science, AI is also benefiting from using Data Science for its enhancement and composition.

This motivates a better understanding of these two areas. More specifically, here we provide a closer look into their interrelation from the databases perspective, in other words, into the convergence of Data Science and AI. To that goal, we will present how Data Science contributes to AI development, how AI is being used to automate Data Science processes, what databases have been built to store and manage AI models and results, and what areas of databases are being enhanced or become new applications with the use of AI on Data Science methods.


AI and Automation in Databases
SELECT *
FROM
Auto-indexing and query tuning
AI-based anomaly detection
Predictive maintenance

2. Auto-indexing and Query Tuning

The design process for databases requires decisions on whether to use indices, their type, and how to maintain them across updates, which requires significant domain knowledge. Automating this is a critical area of research for databases. Query tuning deals with tuning the query to improve performance, including determining what indices to create. While most of the industry solutions focus on tuning the query, their backoff solution is using heuristics, simple cost functions and generally do not support hardware accelerators or cloud environments. Auto-indexing deals with adding and dropping indices, automatically discovering workloads, and refresh models. For example, background jobs are used to drop or add indices based on usage. Additionally, usage frequencies per workload are maintained, so analysts can manually tune index usage.

These companies can automate the discovery of the workload and decide on a refresh strategy to avoid overhead and have minimal disruption during usage. Query tuning as a service is provided to clients. At the query tuning level, a blackboard and model browsing method is used to tune SQL traverse pre-processing. Query tuning affects external decisions such as caching and updating technologies. Indices serve a dual purpose in speeding up point access to data or for joining, and caching tends to postpone index accesses. When a join is applied, data from the two tables is retrieved and cached for further use. Caches serve the purpose of requiring lower latency than disk storage, and logical analysis can determine the preference for slices and caches based on how frequently they are referenced.

In this case, all other queries should also have data in cache. Indices should only be used as a last resort when it is observed that some resources have low usage frequency or cache misses. Refreshing indices might tend to force a choice to miss or delay some queries from cache to be done before, but which should be compensated with the time its results are valid. Additionally, both indices and caches delay update operations. Caches are best positioned when there is evidence that data won't change during the cache validity time window, and both index refresh and use must take care that the data storage is consistent.

2.1. Overview of Auto-indexing

The key sub-services of a database management service are stored data, indexing, query processing, and integrity, security and lineage modifications. The stored data is partitioned to maximize locality and optimize replication for availability. An automatic sub-service offering deals with dynamically controlled automated indexing with no user idle time and with no user involvement in the extensive and complex matter of database partitioning and the decision of when to stop the heavy computation cost of various phases of tuning. User-defined data integrity, security, and lineage policies are themselves stored as part of the managed data, and not separately maintained.

Automatic indexing, or auto-indexing, is a sub-service to support reliability, performance and cost-effectiveness objectives. Reliability deals with seldom-executed queries that become increasingly more difficult to put into a fixed set of common index strategies. Auto-indexing creates different index strategies at different times to try to enforce semantic and intentional locality for query response time. In cloud computing, processing any single query is relatively very expensive because of external I/O, and the auto-indexing can be optimized for memory and disk usage. The key performance and cost-effectiveness issues are for which queries and what times, and what storage and response-time overhead are acceptable for other users demanding availability which include responsiveness to processing support of business operations.

2.2. Benefits of Auto-indexing

Indexing is one of the most efficient techniques for database query performance. Developers typically build additional indices after they run an application and observe the performance by monitoring query plans, query runtimes, and missing index warnings. Adding an index has a computational cost; it makes DML operations such as INSERT and UPDATE more complex, requiring extra time for indexing. Each index also increases the amount of storage space needed for the table or materialized view. Therefore, it is hard to balance the cost and benefit of adding an index. Justifying the cost of creating an index is a long process. Furthermore, a database workload may change over time, and the indices added may lose their effectiveness after the change.

Auto-indexing can automatically suggest the needed indices, tune the involved index parameters, and create or remove indices based on the changes in workloads. Auto-indexing can significantly accelerate database usage for developers. It can carry out expert systems research and automate the repetitive task of index management based on the observability data acquired from databases. By automating these activities, databases can operate around the clock, instantly creating needed indices with parameters needed for any workload. While this process does not eliminate the administration workload needed for tuning a database, it utilizes it more efficiently.

2.3. Techniques for Query Tuning

There are several techniques that can be deployed for automated performance optimization, referred to as query tuning. We categorize query tuning techniques into three types: index-based query control, operator control, and statistical estimation.

Index-based Control Queries retrieve the required data from a set of indexes. However, when available indexes are not enough, or some are not utilized while others are misused, query performance suffers. Hence, index-based query control creates, drops, or modifies indexes to improve the query performance. For instance, user-defined indexes can be created or recommended. Or existing indexes can be dropped, modified by the storage structure, or modified by adding or dropping index columns.

Operator Control Once the query optimizer selected a low-performance plan, tuning techniques can estimate the cost of each step of the operator to either dynamically tune it during runtime or regather error statistics to improve a future selection. Since it is difficult to efficiently implement a hierarchy of optimizers entirely based on estimations, the easier approach is controlling selected non-efficient operators. The tasks of operator control include (i) dynamically tuning PE or resource allocation, (ii) reconfiguring a plan step during execution, (iii) recomputing the tuples passing the step of the filter operator, (iv) dynamically selecting an alternative operator implementation, or (v) controlling join orders and to determine if subsets of query residue need the evaluation.

Estimate Prediction The use or estimation of performance and resulting statistics are fundamental to the statement of many of the issues, not only for query tuning but for many issues within and related to databases. We mention here some of the typical estimators. Statistics for shared samples depend only on the number of distinct values in a column, but we also desire to be able to estimate the number of tuples that match predicate P or apply join J.

2.4. Challenges in Query Tuning

Query optimization is a fundamental aspect of DBMS, as query response time significantly affects the cost of a database system. Optimizing a query, both from a logical and physical design perspective, is a tedious process that requires a high level of domain knowledge, is rarely done empirically, and is critical for the user predictive system. Query optimization has a multidimensional cost and uses multiple resources including CPU, memory, disk bandwidth, and network bandwidth. Query optimization is a delicate balance between exploring and exploiting aspects of a given database workload, an extremely complex balancing act that may have the opposite goals for different users. For example, one user may want to speed up one query in each workload, while others may be trying to lower the overall cost of the system. Each user effectively has their own private model, which is reflected in the parameters that they have given. For example, one user may care primarily about their frequency of internal errors, while another may care about their estimated query execution time and invalidation due to the database structure. The parameter settings that produce the best values for user i's private model will change for different query types, with different input data sizes, with different patterns of query invocation, and with different workload diapason.

2.5. Case Studies on Auto-indexing and Query Tuning

In this section we will describe several research works that studied the problem of self-tuning indexes in different manners. They use various approaches to tackle different types of problems and study different problems posed by the real-life database systems. After having introduced the more classical approaches in index selection we summarize the more recent works that have incorporated workload learning in the decision processes. Finally, we finish describing works that expand the area of search of index selection. After having reviewed the most classical papers, we review the works related to learned cost estimation. These works require using the indices' observability property to work and can augment the search space index selection with novel data configurations. The latter, when made open transforms the database self-tuning into a closed-loop self-tuning system.

We then switch to system-agnostic self-tuning tasks wherein the learning algorithms need to be incorporated into existing systems to take full advantage of the observability heuristic. For this task they take care of the database's observability property. These works need further implementation of the concepts present in the index observability and the learned task. The learning algorithm must be built to take the complete advantage of this feature if we want such implementations to yield positive results. The last two works described are close to this last category. They explore self-indexing with self-learned physical models; hence tasking the overhead of such systems with a self-learning task.

3. AI-based Anomaly Detection

In this chapter, we explore AI-based data anomaly detection. The study of anomaly detection deals with the problem of identifying patterns that deviate from expected behaviour. We consider anomaly detection on a dataset that learns a prediction model from examples that are labelled as normal or anomalous. The model prediction is then used to detect outcomes in time that deviate from expected behaviour. Formally, let D be a dataset of examples and X the features contained in D. Each example D contains a label Y that indicates whether the example is normal or an anomaly. Given D, the goal of anomaly detection is to learn a function f(X) that predicts the label Y of any new example in the future.

The study of anomaly detection has received considerable attention in the research community and industry due to its importance in many applications, including fraud detection, manufacturing monitoring, network intrusion detection, and physical security, among others. Anomaly detection deals with many challenges, both technological and theoretical, including the need for a reliable learning paradigm, as well as the ability to handle skewed classes, deal with missing values, or detect anomalies in different types of data. There are also many practical challenges, for example, learning effective models and making the systems usable. These challenges make the study of effective, scalable, and widely usable approaches for anomaly detection attractive from a technological point of view as well as exciting from a machine learning research point of view as it provides many opportunities for advancing the state of the art in machine learning.

3.1. Understanding Anomaly Detection

Anomaly detection is a technique for identifying abnormal data patterns that are rarely observed in normal, routine, or expected behaviour. Anomaly Detection is popularly studied in domains like Cyber-security, Fraud Detection, Disease Surveillance, Fault Detection and Monitoring, Video Surveillance, and primarily in Sensor Networks. In a particular domain of interest, 99 (or higher) percent of the data following the same pattern is known as the normal pattern. The patterns or data points that are not in this cluster of usual connective behaviour are anomalies. Anomalies vary in variety but they cause huge amounts of damage to the respective operating areas. Some examples of detected anomalies vary from the detection of spammers in social media to escaped criminals or terrorists out in patrol and missing children boards. The world is certainly getting automated, and such algorithms have a huge role to play in order to continuously monitor such activities in networks to alert human beings before time to avoid any possible bad occurrences.

Unlike regular process operations, anomaly detection refers to identifying data points generated from an “abnormal” process – e.g., from malicious activity that can cause economic or reputational damage – while the majority are generated from a “normal” process. Anomaly detection has a different research formulation compared to other classification problems. In classification problems, the major concern is to reach the best possible classification training errors with little concern for model simplicity, while in anomaly detection the major concern is with low false-negative error rates – since failing to detect a dangerous anomaly can have severe consequences.

3.2. AI Techniques for Anomaly Detection

This section gives an overview of AI techniques that can be used for anomaly detection. Here, we refer to AI techniques, meaning specifically AI techniques from machine learning, data mining, and statistics. Classical techniques like sequential and non-sequential hypothesis testing, control charts, and density estimation are all very powerful and important but can be classified as traditional techniques rather than AI techniques. By their very nature, traditional techniques are based on fixed ideas on how data behave, and our contention here is that AI techniques can be more subtly tuned towards the problem than classical techniques.

The AI techniques we will cover in some detail include supervised learning techniques, unsupervised learning techniques, and statistical techniques. We hope to convey a sense of excitement about these techniques and a sense of their attractiveness for real work in anomaly detection. Even more than many areas of AI, anomaly detection is characterized by the diversity of application areas and the methods used in them, and there have been few attempts to do overviews across areas, which is our objective here.

Indeed, much of the early work in anomaly detection involved one-off-system-specific solutions, often inspired by statistical modelling, that were possible due to rich domain knowledge. For example, value prediction for multidimensional time-series is an important early anomaly detection method. More recently, the increasing amounts of data in many application areas, the need for automated online solutions for anomaly detection, and the advances in computing power, available software, and diverse AI methods have combined to make AI well suited for the task. Moreover, the availability of data sharing standards is making it possible to share data, systems, and results across projects and application areas, even though work in anomaly detection is still mostly isolated in project-based efforts.

3.3. Real-time Anomaly Detection Systems

Anomaly detection systems observe computer systems for environmental changes that may indicate a machine's harmful actions or an intruder's attempts to penetrate a system. Anomaly detection has been studied in multiple communities, including network security, medical imaging analysis, system log monitoring, sensor fusion analysis, social media monitoring, and process monitoring. Despite the diversity in applications, the elements in all systems are largely the same: a data source that produces a multivariate time-series data stream, a feature extractor, a classification model that can determine whether the input is normal or not after supervised learning, and an alarm generator.

We offer our own description of the most prevalent anomaly detection systems, in which alerts are generated from trained AI models. Machine learning models have been widely studied. For a labelled training set, anomaly detection is a supervised learning problem. The training data usually contains two classes, normal (majority) and abnormal (minority) labelled data points. The decision models should learn to separate the two classes. During inference on unknown future data, the prediction on a data point would be anomalous if it belongs to the not normal class. Some other time-series anomaly detection methods, which label an entire time-series as normal or not, solve the multi-instance classification problem.

Symbolic embedding feature extraction is a well-known approach to unsupervised time-series anomaly detection and has many roots in computer security. At a high level, the approach represents each time-series input as an item in a dictionary, transforms the time domain into a low-dimensional vector space, and finally uses a class imbalance classifier on top of the vectors of lower-dimensional representations. These classifiers, which have shown remarkable success, indeed have few well-known predecessors. Empirical evaluations show that the methods using symbolic representation of each time-series as a dictionary item perform much better than the primitive original time-series models without this embedded representation.

3.4. Evaluating Anomaly Detection Methods

The validation of anomaly detection algorithms is challenging, and different approaches have been proposed. Anomalies are usually rare, there is no unique best way to define a normal behaviour, and the properties of normality and of the various types of abnormal behaviour can change in time. Generally, they belong to a very wide range of domains and can be detected at different time granularities.

The diversity of types of possible anomalies and the number of domains in which anomaly detection algorithms can be applied call for an agreeability on standardized datasets where labels for normal and anomaly behaviours are provided. There has been effort in that direction, but it is unlikely that direct comparisons can be made unless the algorithms compared evolve very slowly. Therefore, to reduce bias on the choice of the approach to be followed, it is emphasized that when researchers test their algorithm, they should adopt diverse datasets, similarity evaluations, threshold selection, and evaluation logic. This idea is present in many smart and computational models. Being highly multidisciplinary, the adoption of anomaly detection algorithms in autonomous intelligent systems for real-world problems requires flexible approaches that cannot be easily standardized, because the domains may be dissimilar, the time characteristics variable, the evaluation logic specific, the synthesis of final validation results subjective, and the existing example datasets incomplete. Thus, the validation and the evaluation of the different application domains require careful definitions.

3.5. Case Studies on Anomaly Detection

Anomaly detection systems, automatic or semi-automatic, provide an early warning when something is wrong in the system. Alternatively, they can also provide other forms of exploration of the dataset, like finding outliers or conducting hypothesis testing. They have been widely used in several applications, such as intrusion detection systems, where they recognize unsolicited attempts or attacks on a system. Fraud detection is another well-known exemplar of anomaly detection usages, where anomalies are used to identify the people trying to hide illicit or illegal results. Fraud detection methodology can be used in various applications like banks, insurance companies, phone carriers, and e-commerce platforms. Other applications reside in communication sector, health sector (used to identify associated diseases and symptoms), and video axon systems (for conducting objects tracking). One of the hardest tasks would be anomaly detection on spatio-temporal data, which is crucial for security and safety applications.

Few of the available literature showcase the deployment of automatic systems. One such system is for live traffic data. The system first analyses moments of a city regarding its periodic nature to determine when might anomaly happen. The users can configure the “working hours” at which anomalies are expected for the traffic data. The data is then analysed in periods of 15 minutes. The system presents prominence levels for extreme anomalies and the colours provide the significance of the anomaly.

4. Predictive Maintenance

Predictive maintenance is improving the repair and maintenance of components or systems in case of their pending failure in an automated way, such that it is performed at the appropriate time, such that their operations are not interrupted. By knowing when a component or system is going to fail, it is possible to ensure timely disassembly, cleaning, and overhaul of the component or system, without incurring the costs and impact of unnecessary or emergent maintenance. In the last few decades, as intelligent devices become more easily available and the processing and storage costs undergo a remarkable decline, artificial intelligence (AI), driven mainly by machine learning (ML) and deep learning (DL), is finding its way into every aspect of our life. It starts to address challenges in maintenance that were considered impossible a few years ago, such as the impossibility of accurately predicting the future performance of individual components/systems based on their past operation. AI is finding increasing applications in predictive maintenance. In this section, we focus on the application of AI to predictive maintenance. We describe the data requirements for predictive maintenance, and its benefits, as well as current industry applications. The term predictive maintenance concept was popularized in 1980 in the sale literature of predictive maintenance technologies. The popular predictive maintenance tools are acoustic emission, thermography, vibration analysis, oil analysis, motor circuit analysis, non-destructive testing, ultrasonic testing, and shaft alignment. Predictive maintenance works well in highly regulated industries such as energy, mining, oil, and gas. Other examples of predictive maintenance applications are electricity generation.

4.1. Concept of Predictive Maintenance

Predictive Maintenance (PdM) is a direct consequence of Industry 4.0, which a new era of industrial activity. Average operational costs have been historically rising, and the implementation of Industrial Internet of Things (IIoT) technologies and standards captures, stores and processes large amounts of data both from machines and their surroundings. Predictive Maintenance aims to detect failures of industrial equipment on time in a non-intrusive and reliable way. It is also known as Predictive Analysis of Failure or Predictive Operations Management, and it is one of the most important applications of machine learning technologies for the Industrial Internet. The Internet has been combining machine data with data from the plant the machines are by a huge collection of real-time and historical databases, deploying machine learning algorithms for data training and generating predictive failure algorithms, enabling new predictive maintenance business models.

Traditional equipment maintenance of machine tools, automatic, manual and semi-automatic machines was based on Failure Replacement policies or Time-Based Replacement, in which periodic preventive maintenance visits were scheduled and performed. PdM is a transition to Data-Driven Decision Systems for business operations. In the past years, industry's focus has been on lowest operating costs with conservative and reductionist policies on optimization, but Higher Energy Operating Costs climate and political challenges are pressing for many industries to switch to sustainable predictive decision systems to optimize energy consumption of machine production and operation beyond predictive machine failure minimization.

4.2. AI Approaches to Predictive Maintenance

The necessity of performing planned maintenance on all systems is usually determined a priori, generally through domain experts. Experts evaluate the potential risks related to the failure of each separate asset. Maintenance is thus performed, generally adopting a Reliability-cantered Maintenance approach, when the system is still functioning. Alternatively, if analysis is performed after a failure has occurred, the overall maintenance planning may be subject to several inconveniences, including loss of production and correction costs. Such an overall strategy may not always yield the optimization of maintenance efforts and costs. Furthermore, since all these actions rely on expert knowledge, they may be affected by subjective bias.

As AI research develops, amazing results are showcased in an increasing number of trials and implementations. We can say the same about Predictive Maintenance solutions based on AI algorithms. These solutions can be internally developed for very specific needs but are also offered by several renowned software houses within broader enterprise solutions. Graphics are often stunning and crowds' endorsement enthusiastic. Whenever data is available, AI models can be trained to discover hidden correlations among failure occurrences and either one or various conditions, or features, of the monitored subsystem or machinery. Thanks to the fast learnings of AI models and the inherent technological progress, these types of solutions can achieve good results, even in the early stages of deployment. With the vast amount of industrial processes' data being stored and available, we could expect them to achieve outstanding results in close future, greatly supporting further investments towards Digital Twinning and integration of Digital Twin and AI models.

4.3. Data Requirements for Predictive Maintenance

The first requirement for any predictive maintenance is a high-frequency system telemetry. It allows to reliably catch temporal faults. Utilizing high-frequency data, we can reach the best prediction quality due to the high number of faults occurring during the telemetry period. Temporal data can also be augmented with additional information. For example, using drone technology, high-frequency geo-localized image data can be created. These images can then be used to assess the impact of the faults on the surrounding ecosystem and safety. Another example is an augmented industry system-temporal data containing information about the current and predicted weather. Such a data stream can accompany the system data and allow a more precise fault prediction because many industry systems are impacted by weather.

Another data type that is sometimes used for predictive maintenance is system events. Events summarize the most relevant operating states of the industrial system and some additional domain-relevant information. In addition to being less informative than telemetry, the temporal granularity is often lowered because only a subset of relevant states is recorded. For example, many manufacturers typically store for their engine's abnormal events such as IDLE, POWER UP, TURN OFF, START and abnormal engine behaviours at checks on engine level, check execution, and after other relevant events. Such events can be then used on top of the telemetry, resulting in the scored and possibly also labelled events for further prediction.

4.4. Benefits of Predictive Maintenance

While preventive maintenance is often necessary, when lack of knowledge or unreliable model will affect the performance of a predictive model, it will result in unnecessary equipment in many cases causing loss of production or income. In this aspect, predictive maintenance with AI can provide accurate prediction based on the condition of components and/or systems. There are several merits of applying predictive maintenance model: 1. Reduce unexpected failure - Predictive maintenance can lead to fewer crashes and shutdowns and machines run "broken" and "not able to run" at the same time. Therefore, predictive maintenance reduces unexpected bad consequences. 2. Decrease repair costs - Predictive maintenance can restore a machine or component to normal functionality (versus defect-free) while avoiding damage that requires rebuilding or replacing a part or component. 3. Improve performance - Predictive maintenance is increasingly able to calculate when precisely it is best to run and/or repair equipment at peak performance levels. 4. Increase physical asset service life - Predictive maintenance can lead to machines and components running longer and better due to fewer overhauls, and parts and machinery running consistently at ideal levels of usability. 5. Reduce labour burden - Predictive maintenance can lead to lower work related to production maintenance. For example, fewer service men look over machinery each week when companies implement predictive maintenance. Moreover, fewer men need to repair on-site accidents. 6. Increase decision-making factors and skills - Predictive maintenance can enable the predictive maintenance decision-making process to rely less on a particular configuration set. Predictive maintenance frees decision making from overly reliant factors such as maximum likely overhauls, downtime days, repairs each and average work hours.

4.5. Industry Applications of Predictive Maintenance

Predictive maintenance has its roots in the manufacturing, processing, and logistics sector, where it is tightly connected to the concept of Industry 4.0, with disruptions in the manufacturing process causing significant losses with each shut down. It is, however, by no means limited to this area. In fact, the current recommended practice for healthcare sector predictive maintenance applications is to treat them as a version of hospital readmission prediction, so different modelling and evaluation processes are required. In healthcare, predictive maintenance is often called intelligent healthcare. Several pilot projects have been initiated over the past years with extensive idea, data, and results sharing. Predictive maintenance for the healthcare sector aims to be a central checkpoint that assesses patient health and readiness for invasive treatment. Many also refer to predictive maintenance in this sector as the concept of a hospital at home.

5. Integration of AI Technologies in Databases

Databases are understanding complex machines, they manipulate complex data structures that represent reality or knowledge required for translating actions. Therefore, a possible line of investigation is to employ AI techniques in database management to try to automate certain aspects of database management by employing the knowledge present in these structures. This integration is done in two complementary ways: AI integrated in databases, where certain AI techniques are implemented inside the DBMS, and database tools for AI and intelligent agent support, where the integration is such that certain DBMS functions make it easier or assist the tasks of AI processes.

Our contributions to the topic are: a general architectural framework for integration of AI operations in a DBMS, where AI-related designs for several different DBMS tasks are intended be incorporated as plugins; and an implementation of such a framework with actual AI designs in real DBMS modules, including XML retrieval problems, and tools for various AI research themes, like ontologies, enterprise modelling taxonomies, supporting AI where databases play a basic role, and providing user support to intelligent agents. We also propose specific configurations for enhancing information retrieval from XML document repositories and manuscript collections and offering support to origami design. The research is carried out in the context of an intelligent agent-based digital library.

It is also possible to call "AI approaches" such methods and algorithms that incorporate AI ideas and were pioneered by AI researchers but have become de-facto standards in many future applications, like Deep Learning, graphical models, or reinforcement learning. Of course, a proper definition of what AI techniques are needs a subjective characterization, and there does not exist a consensus about it today.

6. Future Trends in AI and Automation

As AI technology continues to advance, it will most likely create additional demand for database capabilities. In turn, databases will need to continue to accelerate to meet that demand. We do not think of AI as delivering a single killer app for databases or unduly increasing demand for data. Rather, we see AI as its own separate but related trend that will create additional demand for fast, scalable, resilient and secure databases. Should each or any of these requirements get too heavy, it may cause a slowdown in the overall growth of AI and/or database capabilities. The push for a near instant response time is very much in line with humans' expectations for fast results. Just like waiting a few seconds for search was once considered perfectly acceptable, today's technology offers searchers the opportunity to be presented with results that come back instantaneously. For most of us, the next unacceptable level of wait time is a few hundred milliseconds. Once that threshold is passed, people rapidly leave the system. AI does not operate in a vacuum; it runs as part of an overall larger system where I/O bandwidth is a key performance driver. While there's still debate on who is responsible for driving improvements in information retrieval, the I/O vendors because they can make their devices more efficient, or the AI vendors because they can either cut down on the amount of information they need to sift through or the approach that they take on the sifting or both, at some point AI won't be able to scale without storage improvement since one cannot indefinitely increase FAST or innovative storage efficiency while driving down marginal unit cost. On the margin, this type of investment tends to shift between I/O vendors and AI vendors. And a range of new I/O devices are becoming available for consumption that offer lower latency at a lower marginal cost. As we move into the next few years, being able to rapidly respond will drive demand for devices that offer low latency for demand while also giving sensible performance scaling for the I/O bandwidth consumed. Still, with low latency not being a constant across the I/O usage pattern, making comparisons and generalizations normal for storage performance relatively challenging.

7. Ethical Considerations in AI and Automation

In the rapidly evolving landscape of AI and automation in databases, ethical considerations take on a heightened significance. The automated tasks provided by databases services are based on deterministic outcomes, which means they do not learn from their previous actions nor adapt based on received rewards. Advanced AI or reinforcement learning are outside of the scope for embedded AI services in databases products, as neither of them is yet scalable with absolute certainty, nor guaranteed to have a global optimum solution, nor failure protection mechanisms in place against actions that maximize expected reward but would cause catastrophic failure scenarios. The overlap between database automation tasks and AI-related features offered by cloud providers for their database products grows daily, yet whilst the cloud providers allow their users to determine the parameters of their AI/ML models or techniques, the automation tasks and user expectations from the products for automation capabilities lack guidelines and checks and balances.

The concerns here are more around user expectations, historical context, and the dangers of being lulled into complacency by the illusions provided by self-service systems, than about databases products harming the user directly. As such, vendors, practitioners and customers need to work together to develop guidelines and user/personalized experiences in terms of the automation offerings. AI and Security needs to be introduced to avoid bad use cases. When DataOps needs to be vetted using compliance guidelines, automating DataOps using databases would need to explicitly highlight the possibility of an undetected compliance lapse. Development teams require checks and balances from their respective Data/ModelOps practices. Clear documents need to articulate the proposal offerings for Deliverables, Deadlines, Degrees of Service available.

8. Conclusion

Database management is a complex task that demands significant investment in both time and expertise. Many mundane tasks that DBAs undertake are not necessarily value-adding for the organization and suffer from human errors. Automation is not infant consigned to only entry-level jobs anymore; it has morphed into a sophisticated solution that is being utilized across various verticals to raise productivity and eliminate data-bias. We present an overview of the work that is being done on automation and AI-enabled features in database systems. The work and research we include covers a range of tasks from some ancillary services exposed in the DBMS management plane like backups, upgrades, monitoring, and tuning, to the core services provided in the data path like data modelling, access layer generation, REST services, and optimization. While the initial tooling is nascent, recent advances in the field of deep learning have suggested that significant inroads can be made in automating several tasks currently performed by human experts, not only in terms of correct suggestions for automation, but much higher percentage solutions through personalization and reinforcement learning techniques. Given that databases are increasingly becoming part and parcel of an organization's digital twin, we believe that significant research and engineering can be applied in the component both from an application and infrastructure perspective toward creating reliable solutions for enterprise customers.

In conclusion, as the role of DBMS continues to evolve from a niche support plane providing bespoke access and storage services to the data needs of an organization to backend services on which the organization bases all of its critical services – monetizing on providing DBMS service reliability while promulgating utility in enterprise management – investment in intelligence-driven automation of mundane tasks is critical for sustainability and growth.

References:

[1] Manolopoulos, Yannis, Yannis Theodoridis, and Vassilis Tsotras. Advanced database indexing. Vol. 17. Springer Science & Business Media, 2012.

[2] Giles, C. Lee, Kurt D. Bollacker, and Steve Lawrence. "CiteSeer: An automatic citation indexing system." *Proceedings of the third ACM conference on Digital libraries*. 1998.

[3] Lancaster, Frederick Wilfrid. *Indexing and abstracting in theory and practice*. Library Association, 1998.

Chapter 10: Database Security and Access Control

1. Introduction to Database Security

A database is a collection of logically related data. The term “database system” refers to the system software that manages data stored in a database. Since the proliferation of all things digital, data has become a valuable, if not the most asset, for all organizations and enterprises, business or government. Data are everywhere, in various forms and sizes; in the CRM systems, corporate domain servers, in smartphones, distributed on the Internet. Many organizations, including criminals from organized crimes to terrorists, leverage data to gain insight about the attempts to achieve their motives.

Individuals in crime syndicates or terrorist organizations collect sensitive or classified data to attempt to hack into corporations or government databases. Hackers, both locally and remotely, adversaries with ulterior motives, and viruses continually threaten the sanctity of data. Many hackers attempt to breach customer databases, stealing credit card information or classified information containing social security numbers, birth dates, and other sensitive identity information about innocent individuals. Credit card companies cannot afford the failure of their business operations if their transaction databases are breached. Sensitive data must be adequately protected to uphold the value of the organization, whether it is a business or a government. Therefore, database security must prevail to gain the customers’ or citizens’ trust.

Ensuring security not only protects and defends the database from unwanted breach attempts but also protects the image and goodwill of the organization that is responsible for protecting the sensitive and various data from digital beats and hacking attempts. Truthfully, no system, whether it be a computer, a mobile device, a server, or a database, is ever totally secure. A system can be made more secure, but absolute security is not feasible. But because all systems can be penetrated, therefore are vulnerable, information assurance is the practice of ensuring that the information is reliable and can be trusted.

Database security is the overall protection of a database from accidental or intentional misuse, falsification, or destruction, while at the same time ensuring adequate availability and legitimate use of the data. Access control is the first line of defence in database security. Authentication is the process of identifying and validating a user’s identity, using any of the credentials such as passwords, passphrases, PINs, or biometrics. Authorization determines the user’s access control rights and privileges, which define the user’s roles, grants or denies permission based on the authentication performed, and decides if the action requested should be allowed or disallowed on the database objects such as tables, rows, and columns.

Database security systems combine access control, input validation, cryptography, data masking, encryption, user activity monitoring, and auditing, to create an integrated security architecture that protects the database from compromise, abuse, and misuse.

2. Authentication Mechanisms

This section introduces the various mechanisms for authenticating users in a database environment. It discusses the problems inherent in relying solely on password authentication and describes several factors modern database systems employ to provide stronger authentication. The security that authentication and password management mechanisms provide is the first step towards ensuring the confidentiality and integrity of data in the database. When a database is authenticating users, it is very important to avoid leaving any weaknesses in the process. Compromising the authentication mechanism can allow intruders to bypass or compromise access control mechanisms and functions of the database.

Organizations and users rely on various types of evidence, usually referred to as factors, to identify whether an entity is who it claims to be. Proof of identity is the basis of nearly all secure transactions. This factor can be a physical ID card or device, such as a Passport, driver's license, or Smartcard, which is presented to an inspector for examination. It can also be personally-recognizable biometric data such as fingerprints, retinal patterns, or facial characteristics. These identity verifications are the sole responsibility of the user. Two-factor identification is a second form of verification, which is independent of and supplemental to the first form.

Several security issues call for authentication mechanisms to not only allow users access to the database but to also verify and validate that users are authorized to perform the actions they are requesting. These issues can include a flight recommendation itinerary website that would return no results for requests outside the scheduled flight timetable. Therefore, authentication data cannot be limited to just userID and passwords. The importance of authentication has gained increased awareness given the numerous online breaches. With the user ID and password being the most common authentication combination in existence, generally weak password choices, an emphasis on continual password changes, and the static nature of most passwords makes the traditional credential set particularly susceptible to compromise.

2.1. Types of Authentications

The authentication mechanism is accountable for the security of a given system. It is the first line of defence to prevent attacks against data. It is essential to select an appropriate mechanism to minimize compromise. There are several ways to authenticate users to a computer system. The authentication mechanism can be categorized into three basic categories:

Maintaining privacy is critical in any authentication system. Authenticating a user requires some knowledge. Providing this knowledge over an insecure channel can reveal secrets to an onlooker. For example, if a password is used as authentication for a transaction, the service may be vulnerable to interception of the user’s password. A similar risk exists if the user’s password is transmitted without encryption for an interactive computer session. Therefore, passwords should be known only to their owners and provided only over secure channels.

However, their greatest risk lies in storage. If a hacker gains access to the database storing password files, the stolen passwords can be used for authentication without the knowledge of the legitimate user. It is advisable to store only one-way “fingerprints” of the passwords. This technique uses special algorithms called hash functions that transform a password into an irreversible representation. On the login page, the system compares the user’s entry to the stored fingerprint. If the match is found, the user is authenticated. All passwords can be discovered, but the attacker has sufficient work ahead that they will not attempt to guess each password in a database. More likely, the attacker will reverse the hash function and obtain the password for only those users whose account names are stored in the database.

2.2. Multi-Factor Authentication

Authentication is the management of security information that designates or associate’s entities with their corresponding subjects, resources, or privileges. Passwords are too often the sole means of binding access to accounts, systems, and information. That is no longer adequate. Password authentication can be broken or bypassed. Compromised credentials provide hackers with the same access as legitimate users. These credentials are often cheap. Attackers will utilize several ways to commit credential theft, including phishing, man-in-the-middle attacks, and even social engineering, all of which can exploit the user action of inputting their password. Vulnerabilities allow attackers to use malware to obtain the password as it is typed in. Up until now, passwords have also been easy to poorly manage. Users reuse the same credentials across multiple sites that hold different levels of importance, with personal and financial information located on social media and banking accounts. If one of them is attacked, it becomes trivial for an attacker to get into your others. They also fail to require training of users. Black-hat hackers create a myriad of social engineering-type attacks, including fake websites and warning messages. Users don't consider these at all. Recently, though, passwords have been getting more challenging to deal with. Websites are requiring long passwords, along with complexity requirements that encourage unique passwords for each site. Password managers have sprung up to aid in their management. Sites also sometimes have a time window that prevents logins after several failed attempts. But even with this, passwords are still a solid attack vector.

Taking the extra steps to utilize multiple forms of verification when getting access to an account makes the login process more secure. Multi-factor authentication can utilize more than two factors or forms of authentication, but most recognize it as a two-factor system. Multi-factor authentication, as its name denotes, brings in at least one more means of authentication beyond a password. It provides a much higher level of assurance. Although it is not bulletproof, using multiple steps means that an attacker is less likely to be able to impersonate a user. Multi-factor authentication is already built into many websites, most notably financial sites and e-commerce.

2.3. Best Practices for Authentication

Securing your databases requires specialty protocols that make manual commit/rollback actions on the outside difficult []. Passwords are at the centre of most authentication discussions. Security is compromised as employees deploy weak passwords or make them available via memo notes. Choosing a strong password in a system that supports password expiration must be considered a high priority. Passwords should be long enough (minimum 8 characters; 40 for maximum strength) to slow down guessing schemes. Users should select different passwords and not use them on multiple systems, which rely on access from the database. Also factor in how long a password could be useful without being changed. How long would it take me to break into a specific system, i.e., email? Depending on the available resources, it would take about 4 seconds with a 7-character long password. Naturally, this weak link, passwords, should be phased out. Plans should be in effect to fully utilize Unix-type authentication schemes, be deprecated when possible, or other options. Unfortunately, routers and switches do not support this on their own.

Conditional access can help organizations to even further mitigate security issues. For example, with workforce use of open data or hosted applications, you can require that access is only granted when users or devices in certain geolocations check it (such as during work hours where you have a network connection to your firewall appliance). You can also check for device compliance before granting access. This is a tighter control as this screens out attacks coming from rogue web cities.

3. Roles and Privileges

Understanding and providing privileges to access controlled database objects is one of the most important tasks that a database administrator must perform. Granting excessive privileges to database users or roles increases the attack surface and thus, can become a major security concern if principle of least privilege is not followed. At the same time, misconfigured false denials of legitimate access requests can bring the business operation to a halt. Hence, careful planning and ongoing monitoring of privileges is critical. In this chapter, we shall start the topic with a discussion regarding roles which summarize all the privileges of a specific user group and how they can be used in database access control. Then we continue with a discussion of privilege management and its complications in the context of database security. We would finally conclude this chapter with a discussion on Role-Based Access Control implemented in some database management systems.

Understanding Database Roles A role is a collection of privileges that can be assigned to database users and user groups to simplify access management. A role aggregates all the privileges that are needed to accomplish a task or function. It is not uncommon for a set of users to have the same privileges; they may be responsible for the same type of action on a set of data within some similar data contexts. For example, user advisors have the same or similar actions to perform on student data on a regular basis – those actions include viewing the students' grades or advising their study tracks. Thus, those users usually have identical privileges. The database administrative role of a database is responsible for managing security by managing user account, authentication, and profile privileges before granting other privileges to users. In practice, several administrative roles are created. The payroll administration role may have additional privileges on a payroll database.

3.1. Understanding Database Roles

Database users execute SQL statements in a DBMS, which performs tasks on behalf of the user. However, the tasks themselves are not without authorization; some users can perform only specific tasks, while others can perform all tasks. Users who can carry out various types of tasks include database developers and DBAs, who maintain the database security, performance, and availability. However, allowing users to perform all tasks is dangerous. For example, an application user who has both database access and delete privileges can, if provoked, delete the whole database. To prevent malicious or erroneous actions from database users, administrators carefully assign individual security privileges to users.

Over the past several database system versions, such fine-grained privilege management has become more dangerous and more tedious in database systems with a high number and constantly evolving number of functionalities. Additionally, the number of application database users is usually large and constantly evolving. The ever-expanding privileges, privileges underlying constantly evolving database system functionalities, user base size, and database system requirements usually require drastic reconfiguration and simulation of the matching security policies of multiple related databases. User or group level-based revocation of specific privileges may not match any trusted policy for a limited-activity duration. These drawbacks are better addressed by database roles, which are easier, faster, and safer. Database roles allow a global user base and enable the addition of specific users and applications in the role database operations. The role concept helps automate privilege granting. Decision support systems and data warehouse systems are better suited for role support than transactional systems. They usually have a smaller user base who only do read operations and do not often need the most up-to-date information. Therefore, dealing with a small number of insignificant read-lock contention and performance penalties at a time with read consistency is acceptable.

3.2. Privilege Management

As we have seen, all access control techniques check in some way if the subject (the user) that is trying to access a given object (the resource) is authorized to do so. This check is usually done by using credentials associated with the user in question, such as passwords in the case of a simple authentication. In a database, this check is usually combined with some other techniques, since the user alone normally only sees his or her own data. The user identity is checked against a privilege table that holds all user privileges or a hash tree that joins user identity and data object. Privilege tables are usually used to store a small number of users, while hash trees make a better job of accessing user privileges when many users are being used by a single application.

Privilege management, as one may call the administration of privileges, is a very important issue in secure database administration. The integrity, confidentiality, and availability of sensitive database information largely depend on how well the extensive set of database functions are made available to the user community. This, in turn, depends largely on the error-free configuration of privileges at the user, role, and context levels. Achieving satisfactory security requires considerable knowledge of the database system as well as of the business performed by the company that relies on very sensitive data stored in its database. The configuration of privileges is therefore not as easy as the execution of a set of administrative statements; it requires a lot of testing. Guessing a set of user role combinations that allow users to perform their assigned tasks, while preventing misuse, is a difficult task.

3.3. Role-Based Access Control (RBAC)

Role-based access control (RBAC) is a very popular access control model in which permissions are associated with roles, and users are assigned to roles. RBAC is attractive because it reduces the complexity of privilege management and can help administrators allocate permissions in compliance with the principle of least privilege. For example, if a user requires certain permissions to perform a job function, the user can be assigned to a specific role associated with the permissions instead of granting or revoking the permissions individually. In addition, permissions do not have to be assigned for each user, only for roles. If a new user is in a position within the organization that already has access to sensitive objects, the new user can be added to the role instead of having to duplicate all the permissions. Likewise, when a user transfers to a different department, the user can be removed from the role in the former department.

RBAC is easier to use than DAC and MAC in several ways. The administrator does not have to grant permissions each time to a new user or group of users—permissions need only be associated with the role, and a user may be assigned to several roles. Further, the set of roles can be restricted for each user so that the user can use the system only in certain ways. Each user is assigned multiple roles to use the desired permissions at a certain time. While a role usually maps to what a user is doing in the system at a particular moment, the purpose of restrictions is to forbid the user from taking advantage of holding multiple roles that belong to different organizations or departments to perform prying tasks. Thus, although RBAC abstracts that users and permissions are tied together by roles, it also entails some aspects of DAC or MAC. RBAC is a generalized model that can encapsulate both DAC and MAC rules. For example, if all permissions are assigned to separate roles and users are allowed to define an arbitrary number of roles, RBAC is an explicit form of DAC. The definition of the role, role-to-permission relationship, and restrictions can be used as a wrapper for implementing DAC and MAC mechanisms.

4. SQL Injection

4.1. Understanding SQL Injection Attacks

Structured Query Language (SQL) Injection is proprietary to all SQL-based databases. SQL Injection attacks remove security protections, allowing users to destroy or otherwise violate the integrity of the data, if they can formulate a query that is properly sanitized and verified. An attacker can leverage an SQL injection vulnerability to bypass application security measures. Some of the documented cases of SQL Injection attacks have been motivated by cyber espionage, political objectives, and even, more oddly, hacks for the good. These altruistic hackers disclosed the attack as a way of drawing attention to security deficiencies in a government. Given the significant persistence of SQL injection attacks, it should be no surprise that research to minimize the threat from, and damage inflicted by, these attacks is equally persistent. There are both intrusion detection systems and intrusion prevention systems which attempt to mitigate the risk by inspecting application traffic.

Because of the powerful features of structured query language, it is widely used for developing relational database systems for web applications. Dynamic web applications are designed in such a way that they transfer requests to back-end servers containing databases. The requests fired by users from the web servers are checked by the back-end servers against SQL to see whether they are valid or not. Databases store valuable and confidential information; therefore, SQL servers need extra protection and security. Attackers may access database content and may tamper information in databases. They may even crash the service. SQL injection is one of the most powerful paradigms of attacks on various web applications. Although web application firewalls may protect from some attacks, they may not be helpful against SQL injection attacks.

In general, an application accepts user input and builds a SQL statement using that input. If the application does not filter or escape the input, it also allows an adversary to insert additional SQL syntax into the query. The adversary is able to access data unrelated to him, or maybe change the data, or even execute other commands or operating system commands that are not allowed, or may bypass authentication mechanisms. This may allow an adversary to do any operation that is permitted by the database management system related to the logged-on user. Because SQL language is standardized, SQL injection attacks may be performed on multiple databases.

4.2. Common Vulnerabilities

Input filtering is the best-known prevention technique against SQL Injection attacks; however, developers inadvertently create inputs that fail to sanitize properly. Side effects from this lack of sanitization lead to the significant volume of known attacks still being successful by successfully crafting a malicious input. Furthermore, there are plenty of exploits left as specific applications are known to have poorly defined interfaces and inadequate input validation. The reasons for this failure to sanitize database queries include not recognizing user input as sensitive, lack of awareness of sanitization, relying on user input formats that are too strict, reliance on third-party interfaces, regressive security choices, limitations imposed by inclusion of legacy code, and including input parameters in concatenated SQL statements.

The most common vulnerability related to SQL injection is the unauthorized viewing of data. This can occur when accessing sensitive information such as credit card numbers and user lists. Additionally, SQL injection attacks enable attackers to compromise the confidentiality and integrity of any sensitive database. By attaching malware to the database or by releasing consumer data to competitor websites, the attackers can damage the business owner's reputation and credibility. Unauthorized viewing of data may also incur heavy fines.

Users are often unaware of the data on a system. When a valid and authorized user connects through an application interface, he or she should be presented with the right credentials. SQL injection can lead to the unauthorized viewing of database data. Inputting a malicious SQL query through the input interface can give an attacker access to any data that the actual intended user could view. For instance, if a user connects and views some of the columns of data in a user table on a SQL database, an attacker could use SQL injection to list other column names in that database. This would result in unauthorized viewing of columns that the actual user could not see, thus leading to a dangerous vulnerability in the system.

The attacker can input a malicious SQL query that could return credit card information. If a user reports a lost or stolen credit card, the bank will freeze the account until investigations are complete or would issue a different credit card to the customer. If the attacker sells these stolen credit card numbers and other customer information, he or she could easily pocket millions of dollars with little effort required. Stolen credit card information can affect e-commerce businesses by destroying their reputation and trust with their consumers.

4.3. Detection of SQL Injection

Database security and access control systems should be able to detect and respond to SQL injection attacks. This task is usually simplified by the fact that it is ad-hoc code that is exploited by SQL injections. This means that a generic database monitor can't usually be used to detect such injections as it will trigger too many false positives. Indeed, a large database performance product allows instrumentation of ad-hoc code, but it shouldn't be used this way all the time, as the associated costs are prohibitive. This document is usually referred to as a monitor or an audit. The latter term usually refers to maintaining a possibly large and separate database with the execution of each query, while the former stores only the metadata. We conduct specific ad-hoc probes that will detect all sorts of SQL injection attempts with an acceptable level of false positives.

The typical use of a SQL injection exploit is to access tables and columns containing sensitive information, although arbitrary code execution exploits fit into the same category. Logging access to metadata information, specifically access to system tables and system columns, will produce a relatively small number of audit records and is more efficient, with a balance between efficiency and detected events. However, log data can be forged to avoid detection, which does not hold for system tables metadata access review: The normal user does not have access to system table resources. A SQL Insertion Exploit Query Tracking would also produce a large monitor or audit, which is not practical without having some methods to filter the output. In the contrary logic and track specific exploit queries. Most logs and auditing systems built-in already track only the type of qualifiers we are interested in.

5. Mitigation Strategies for SQL Injection

In essence, an SQL injection is a kind of attack whereby the attacker tries to access objects available in a database system to either read sensitive information or even erase. Applications typically use SQL statements to access database systems. They use inputs received from users to create SQL statements without verifying that such input is safe. Attackers can use such input fields to send SQL statement modifications that will allow them access to information that they are not supposed to see. An SQL injection detection system uses different techniques to detect, alert and possibly prevent an incoming SQL injection attack. Below we describe most common SQL injection mitigation strategies used by web application vulnerability scanners or security policies adopted by organizations being prone to SQL injection attacks.

Prepared Statements and Parameterized Queries

The most used technique to eliminate the risk of injecting SQL queries in program-source code is the usage of Prepared Statements and Parameterized Queries. Importantly, the use of such SQL statements means that SQL queries are defined using placeholders that are only later given values to prepare for execution. Once the SQL statement is compiled, any data used is checked to ensure it's safe for executing the operation originally specified in the statement. Since user input is never directly put in the SQL statement, there is no risk of harmful SQL code being executed at any time.

5.1. Prepared Statements and Parameterized Queries

Introduction to Mitigations Strategies for SQL Injection

SQL injection is considered one of the most dangerous threats for Web applications, besides being persistent in the time. Mitigation techniques vary and try to reduce as much possible the risk by validating input args, designing the queries with built-in control mechanisms, and “paraphrasing” the SQL queries not allowing them to be executed as written, but to be interpreted in some way that also checks for correctness. In this chapter, we will discuss some mitigation strategies that are the most used and more friendly to the development implementation.

A prepared statement is a feature used to execute the same (or similar) SQL statements repeatedly with high efficiency. A prepared statement is compiled and stored in a prepared statement template. In this template, placeholder parameters are used to replace actual parameter values supplied at execution time. When a prepared statement is executed, the DBMS creates a new SQL statement by combining the prepared statement template and supplied parameter values and then executes the new statement. Security is guaranteed at the database engine level that only allows you to bind values to specific logical data types, allowing runtime parameter type validation.

5.2. Input Validation Techniques

One of the main concepts behind input validation is to treat users as malicious by default. Since user input cannot be trusted, all input should be checked to detect potentially dangerous and malicious input. Input validation is useful for protecting applications from a wide variety of malicious input that could trigger validation vulnerabilities. Relying on input validation alone is rarely sufficient. Security procedures, such as sanitizing or filtering, signatures, sanitizing, prepared statements, and use of stored procedures, should be implemented to work with input validation to provide more reliable security. Before validating input, developers must first understand both the requirements for valid input and the allowed input to match those requirements. Each application must validate input based on its context and the system implemented by developers. Application or business logic must dictate the extent and form of the validation checks performed. Input is valid if it matches expected, highly restrictive criteria that the programmer has designed for input. Even with extensive validation, it is not possible to prevent every possible validation error, for users can sometimes present unexpected or unpredictable input. For instance, one of the real-world SQL injection warnings is shown in the table. The warning indicates that the Kamiya character cannot be encoded in Shift JIS. Since output encoding cannot correct input validation errors, developers must examine their validation rules to determine if they need to add special handling for the erroneous input.

5.3. Web Application Firewalls

To reduce the risk of a successful SQL injection attack, it is possible to use systems that are installed on the user's network before the web servers. Those programs analyses the requests and responses to prevent successful SQL injections, by removing the probable SQL injection patterns. By eliminating these commands, the system tries to preserve the interaction of the user with the web application by hiding what is happening. Even though some systems can be incorrectly set and still allow the attack to occur, or block some transactions that users want to perform, and reduce the usability of some web sites, companies install those systems as a step against prevention or detection from the attacks. Those devices are web application firewalls. Those devices differ from the firewall that is already a consolidated part of the network security. Regular firewalls usually block SQL injection attacks, protecting only a few parts of the application layer, following a predefined set of security policies. This information is usually related to the transport protocols, ports, or address multilayers. Even though these protections help web application security, they do not perform deep inspection of the web applications since they look only for signatures. Therefore, regular firewalls are not very effective against SQL injection or other attacks. Web application firewalls were created to enforce the application layer vulnerability protection. They operate in front of the web server and act as intermediaries, allowing or removing packets based on rules. By doing this deep packet inspection, web application firewalls verify the content of the packet and perform checks based on the state of the session. The main advantage of those devices is that they execute a more granular analysis of the traffic on the application layer and therefore allow the discovery of vulnerabilities that regular firewalls would not find. Their performance is better than regular firewalls since they have specific rules for the application, but they tend to be more expensive.

6. Data Masking

6.1. Concept of Data Masking

Data masking is a security mechanism for providing controlled access to databases, particularly sensitive information about customers or employees that cannot be made publicly available [1-3]. For example, a company's employee database may contain national identity numbers and bank account information, which are very sensitive from the employee's perspective. A realistic database of employees is useful in testing applications that are supposed to be read from and/or write into this database. However, granting access to this database without removing this sensitive information poses a security risk, in case the testing application has a bug that causes the sensitive data to leak out.

Data masking techniques create a database like the original but without sensitive information. One simple way to do this is to randomly scramble the values of all character string columns in the original database. However, the values of a certain column of the original database and the corresponding column of the masked database may be related in some manner. For example, a column containing postal codes may have the value of 10001 for many rows in the original database, due to many employees working in the headquarters of the company located at 10001. If the values of this column in the masked database are randomly scrambled, it is possible that the same postal code is assigned to different rows in the masked database. This would not happen in the original database, making it possible to detect that this column has been masked. This suggests that the values of certain columns need to be masked in some coordinated way.

6.2. Techniques for Data Masking

How are the values of certain columns to be masked in a coordinated way? First, we can assign all the values of a particular column to a set of values, rather than masking each value separately. For example, imagine a column of postal code values, such that the only possible values are 10001, 10002, ... , 10010. To mask this column and keep its relation to other columns, it would be fruitful to randomly assign the values in the set {10001, 10002, ... , 10010} to the rows of the database using a random permutation. In a similar manner, all the values in a column can be assigned to random values within a set of possible values. This is how data masking works, essentially by scrambling values within a column or assigning a random value from a known set. This approach has two obvious downsides. First, the database needs to have some kind of consistency, meaning that it is unlikely to have a very large number of unique postal codes or bank account numbers. Second, to build the relationship between the original database and the masked value databases would take considerable time and effort.

6.3. Use Cases for Data Masking

Let us consider when data masking makes sense. Data masking is a good choice in the following use cases: Developers and testers are working with realistic data and therefore are required to have access to the original database. The original database has security-critical but non-business data, such as personally identifiable information or PCI-compliant credit card numbers. Dev/test databases can be easily created from the original database but using the original data as gives rise to security concerns. Data masked to the maximum extent possible would still be realistic and help in test cases where the actual data is a critical part of the test. Realistic data is required in test cases that cannot be tested with synthetic data. It is also advisable to pill the data within a business context. For example, a software testing company may require a developer's database but may not be in the loop on which company the developer's documentation is for.

7. Encryption in Database Security

Database and data security has come in recent media attention due to theft of personal, confidential information [2,4]. Choosing the right encryption for databases is an important decision that can affect performance, return on investment, and top-level security. Encryption solves three primary protection requirements: Data protection, Data integrity, and non-repudiation. Incorrectly used encryption may cause denial of service. Thus, organizations must plan carefully what data to protect with encryption and what type of encryption to apply. Encryption may be applied to certain pieces of data in a column like Social Security, credit card number or encryption may be applied database wide. Furthermore, column level encryption can be implemented in a variety of algorithms. Furthermore, Encryption applies to data at all three stages of the information lifecycle: Data in Use, Data in Transit, and Data at Rest. Proper planning decisions must cover the slew of potential scenarios. Encryption offers protection against unauthorized access, as well as adds integrity checks to protect against unauthorized data modification. There are two major types of encryption algorithms. Traditional, symmetric algorithms use the same key for both encryption and decryption. Data encryption standard and Advanced Encryption standard are the most used symmetric algorithms. Asymmetric algorithms make use of two keys – one for encryption and a different one for decryption. During a secure session, one machine would be encrypting data using the destination machine’s public key, with the destination machine decrypting the data with its private key. While RSA and Diffie-Hellman are the most used asymmetric algorithms, asymmetric algorithms are considerably slower than symmetric algorithms, lending only to intermittent use.

7.1. Types of Encryptions

In such case that attackers get hold of encrypted files, they will only see a stream of data with no meaning. The various types of encryptions can be classified into two major mechanisms: symmetric encryption and asymmetric encryption. Symmetric encryption is a single key encryption, using one single key to do both the encryption and decryption processes. The security of symmetric key encryption relies on the secrecy of the chosen key, which should always be kept secret. If an unauthorized individual gets hold of the key, it renders the whole encryption process useless. Thus, in practical applications, often, the symmetric key is exchanged by other persons using other secure means. Symmetric encryption is very fast, and it can be employed for data encryption irrespective of the amount of data to be encrypted.

Some examples of symmetric encryption algorithms include AES, RC4, DES, and its key lengths vary. Asymmetric encryption is the public key encryption. It is called public because it uses a public key and a private key. Based on the RSA algorithm, asymmetric encryption deals with a unique public key and a private key, where the public key is used to encrypt data and the private key is used to decrypt data encrypted by the corresponding public key. As stated earlier symmetric encryption is faster, but its performance could be very slow for bulk data encryption compared to asymmetric encryption. As a result, asymmetric encryption is more suited for small amounts of data, and it employs symmetric key encryption for bulk data, where the symmetric key is encrypted by asymmetric encryption and transmitted.

7.2. Encryption at Rest vs. Encryption in Transit

It is crucial to understand the difference between encryption at rest and encryption in transit for developing an effective data protection strategy. Data at rest means inactive data as it is stored physically in storage media and not actively being moved around, whereas data in transit is data actively moving from one location to another, such as across the internet or through a private network. Encryption at rest is a data protection method that secures stored data; it encrypts data that is "resting" or in a database, file, or storage device. With encryption at rest, the data is encrypted before being written to the driver and remains in an encoded state until the authorized user accesses it, usually through encryption management software. Further, encryption algorithms, hashing and key management are generally associated with data-at-rest encryption. Typically, file types that are targeted by encryption that is performed on data at rest are database table files, application files, and other types of business files, compressed files, backup images, and VM images virtual appliance files. Encryption of data at rest can be done at different levels. File level encryption is the oldest form of data at rest encryption, where the system encrypts a single file at a time. The second level is Volume or Disk level Encryption, which encrypts volumes or disks in their entirety for their entire existence.

Conversely, encryption in transit refers to protection protocols that are used during transmission of data; it secures data that is actively moving through networks and the internet, such as between a user and a website or between data centres and remote servers. With encryption in transit, the data is encrypted before it leaves the sender's end and stays encrypted until it reaches the intended recipient, wherever in the world that might be – usually via a secure communications line. Data in transit is comprised of data packets. Encryption-in-transit methods include network layer encryption, especially for network-to-network traffic, and end-to-end or application layer encryption, for scalable security for applications such as web browsing, webmail, and file transfers.

7.3. Key Management Practices

Key management is not only a separate subject of study in cryptography but also a part of the government documentation of almost every respected cryptographic algorithm. There are two reasons why it is so important. The first is that if your key is compromised you cannot trust the results of the cryptographic algorithm, and the second is that if you lose your key, you cannot retrieve your data. Unfortunately, the need to protect information is often greater than the practical problems involved in key management and as a result the techniques are used hastily and without a clear strategy. To their regret, the users too often find out the hard way that access and authentication become difficult when the database key management is designed poorly. Fortunately, tools are available that can alleviate many of these problems. However, the development of a solid, long-living key management policy relies heavily on individual needs. The following is a list of considerations that a database administrator should consider while designing a key management strategy. Time Requirements: How long do you encrypt the database? Informally, what is the lifetime of your key? Some factors to consider: the keys may need to be available periodically, but not all the time. When will the data you need to decrypt become uninteresting? Using the same key too long is another security threat. Available Equipment: Are there processors with sufficient computational power available for encryption operations? There is a well-dated recommendation that the latency from the Request stage to the Data stage should not exceed 250 ms otherwise, AES based encryption cannot be performed in software in a commercially viable manner.

8. Compliance and Regulatory Considerations

The advent of databases has opened a multitude of creative business models, while presenting renowned challenges in how organizations share and protect data assets. Data protection regulations are a fundamental aspect of a data security infrastructure as they are guidelines that local and international organizations use to define how sensitive data must be secured, to protect against unauthorized disclosure, loss, theft, or misuse. As we have seen in the past, databases are not secure just by virtue of being databases. As organizations continue to acquire more and more data, they need to ensure that their data security infrastructure meets both compliance and regulatory requirements. A regulation is a binding legislative act, while compliance is about conforming to a rule or a standard. When it comes to compliance and regulatory frameworks, meeting requirements is more than a business exercise to avoid fines. It is about taking appropriate steps to build and maintain data security infrastructure and policies that can not only protect sensitive data but also allow your organization to respond effectively and quickly to any data breach.

It is also essential that organizations pay attention to violations of regulations as history shows that violations often result in crippling fines and penalties. Date back to 2003, the Payment Card Industry Council began requiring merchants and financial institutions to process credit card transactions to protect customers' sensitive credit card data, imposing substantial consequences for breaches resulting in fraud. Similarly, in 2008, the Federal Trade Commission instituted the Safeguards Rule, mandating financial institutions protect consumer data such as Social Security numbers and bank account information. Over the years, violations of these mandated regulations have resulted in billions of dollars in penalties.

8.1. Data Protection Regulations

Access control is not just about fitting the right people, both internal and external, to the right roles; it also involves meeting relevant data protection regulations. Data protection regulations set out legal obligations on how organizations manage and protect data. These laws often cover all types of data and data subjects including individuals, employees, and customers, as well as other third parties such as suppliers which means access control in the broadest sense plays a significant role in compliance.

From the perspective of everyday business access control, these obligations mean that organizations must define and document the processing activities on personal data and other regulations that apply to them and applicable for which countries or areas of activities. This information forms the basis of classifying data and consequent assignment of access control identifiers. As part of doing this, the organization must document the rationale for the assignments, the technical controls used to enforce. Many of these data protection regulations require accountability to be demonstrated, so organizations will need to perform audits of access control and other activities periodically; for many, there is a specific requirement that audits must be carried out at least every two years.

8.2. Impact of Non-Compliance

Failure to comply with industry regulations can carry hefty fines and lead to litigation. In the United States alone, more than $40 billion in fines were issued in 2022 due to infractions, predominantly for mishandling sensitive data. Fines are high enough to put businesses out of operation. For example, in early 2021, a major airline was fined $22 million for regulatory infractions due to a data breach compromising sensitive customer personal information.

Sensitive data breaches often lead to identity theft and monetary financial loss for affected customers and/or employees. The impact of a data breach often involves more than just the financial component. Getting affected employees and customers up to speed on mitigating steps they may have to take can drain human resources, accounting, management, and customer service functions. Companies must also account for customer trust issues as word spreads about any sensitive data breach. Filling the information vacuum left when customers lose faith in internal communications is critical.

Companies must also ensure they have resources to manage customer inquiries related to the sensitive data breach. There could be a fallout if there are not enough employees to answer queries and questions so that customers do not feel abandoned. In today's global economy, there is also risk related to companies losing business because of being associated with "doing the wrong thing" in a committed corporate environment. Data breaches reaffirm that a service organization may not be capable of looking out for external customers. Any fallout ties into a company's brand, corporate image, and reputation, and may need to be rebuilt if the company promotes a culture of openness and honesty.

9. Future Trends in Database Security

Database security continues to evolve rapidly, and with it, so does the categorization of the importance of topics related to that evolution. However, two themes recur, several current topical areas reflect the interests and research of researchers worldwide. The two areas of interest are: Emerging threats related to the behaviour of databases, experienced professionals, researchers, and notable vendors agree that threats due to careless behaviour, malice, and mistakes by humans continue to be the number one cause of database-related breaches. Malicious code is the new frontier for the internal threat. Over the past decade companies have sharply improved internal protection systems and established prudent security policies that address many human behaviour issues related to the web: e-mail filtering systems that spot and quarantine outside malicious e-mail are common. Active content on web pages that are not properly authenticated has become a rarity, or at least one of the untrustworthy major defences that are easily recognized. Policies that teach employees about the dangers of web-based malicious content are commonplace. All it takes is one employee ignoring Company Policy to open the floodgates of disaster, setting the company up for an internal breach. Traditional attack and defending response procedures are not enough. Passive defences such as network-based firewalls, links dedicated to transaction activities, threat models for protection, or examining code and revoking raw SQL privileges are starting points, not solutions. Continuous automated monitoring for detected abnormal behaviour is a requirement for safeguarding the company's data — alerting with the option to correct or allow the activity to continue through a manual process. External threats are constantly changing in behaviour, or even mode of entry. As they are not under the control of the firm's protective measures apart from the security policy, it is left up to employees and customers to avoid malicious breaches of sensitive data. They can only do so if they are aware, trained, and diligent in their business behaviour.

9.1. Emerging Threats

Advancements in technology and in supporting cyberinfrastructure continue to produce difficulties in database security, and these are tending to increase rather than diminish. New classes of devices, such as smart sensors and wearable devices, produce unprecedented and constantly growing volumes of sensitive data. At the same time, challenges are posed by the users of this data wanting to gain and leverage insights from the secured resources built up over many years involving individual users whose whole lives are stitched together by the information provided to cloud-based systems. The threats are broadly classified as Data-Resource Cross-User and User-Resource Cross-Domain using the above multi-layer architecture of the cloud database environment.

Connected devices will continue to surface huge amounts of dynamic data and possibly sensitive personally identifiable information (PII)-related data as well as business-critical data collected in various industries and sectors. Handling such a massive volume of sensitive data, including verification of legitimacy and authenticity, the continuous flow of data remains a daunting challenge. As businesses and individuals across the globe continue to adopt technology, any vulnerabilities or loopholes within the connected systems can be exploited by threat actors to continuously compromise sensitive data. Furthermore, as vastly more connected devices with large footprints and egregious corpuses of PII enter the Internet landscape, the myriads of threat vectors will also proliferate. External threat actors continue to develop methods for compromising proprietary systems and product lines, whether domestically or internationally. Advanced persistent threats (APTs) from malicious foreign actors have exploited critical roots of trust in the hardware supply chain and information technology industries such that firmware is insecure to the externally deployed and relatively unprotected hardware.

9.2. Advancements in Security Technologies

Security has long been a consideration in database design for many reasons, not the least of which is the fact that databases store large amounts of sensitive data. The need to secure sensitive data must be balanced with the need to provide that data when it is needed and with the need to conduct transactions at the lowest cost possible. Neither of those needs can be sacrificed. Over the centuries, we have made great strides in improving security technologies, and as we go into the future, we will be doing even more of the same. The enhancements fall into three major categories.

Improvements in software helper technologies. Artificial intelligence is only one of the technologies that has made it easier to create robust security solutions. The hope is that as they continue to develop advanced machine learning and fuzzy-based security mechanisms, security technology developers will be able to move beyond statistical models of "normal" and "anomalous" behaviour and instead build solutions that can "learn" a particular enterprise's operations and adjust over time without constant returning. The increasing use of trusted operating system environments, coupled with virtualization and application container technologies, should make it easier and easier for corporations to enforce access control policies and policies that delimit the environment and resources for each database.

Improvements in strength of the module capabilities. If you are reading this in 2023, you probably know both what a biometric device is and how it works. While they have been around since the earliest days of computing, their newfound ease of use and reliability have allowed biometric systems to begin to replace traditional authentication solutions as the answer to the question of "who is accessing this resource?"

10. Conclusion

There has been a substantial amount of research on database security, both in terms of proposals for security controls to manage a wide variety of threats as well as actual deployments of solutions to secure databases. These security solutions basically span three categories: access control/authorization models, encryption-based approaches, and detecting unauthorized database usage. We discussed various specific solutions in these categories and the classes of database security threats they manage, as well as the limitations in their ability to counter threats. In addition, we discussed the potential impact of such attacks on the organizations affected by data breaches, as well as the legal frameworks that mandate certain forms of database security for some companies as well propose monetary incentives for adopting best practice approaches for protecting customer data from various forms of data breaches.

However, applying security patches to database management systems, using risk management techniques to decide which databases should be encrypted, adopting an exit control time limit strategy for preventing SQL injections, etc., are not specific to anyone-layer. Database security policies span all three layers and will evolve as attackers change strategies and motivations. For example, clients perform most database accesses. Therefore, the security protection associated with this layer will determine the overall effectiveness of a three-layer database security strategy, especially since client-side security management has slackened. If threats originate from the client side, then a database security strategy focusing on multiple layers of protection is redundant. A similar point can be made regarding attacks that involve client platforms. A database security strategy can be effective at multiple layers only if database access by enterprise applications and customer clients can be monitored to detect all unauthorized or abusive activity.

References:

[1] Bertino, Elisa, and Ravi Sandhu. "Database security-concepts, approaches, and challenges." IEEE Transactions on Dependable and secure computing 2.1 (2005): 2-19.

[2] Benantar, Messaoud. *Access control systems: security, identity management and trust models*. Springer Science & Business Media, 2005.

[3] Blackley, John A., Thomas R. Peltier, and Justin Peltier. *Information security fundamentals*. Auerbach Publications, 2004.

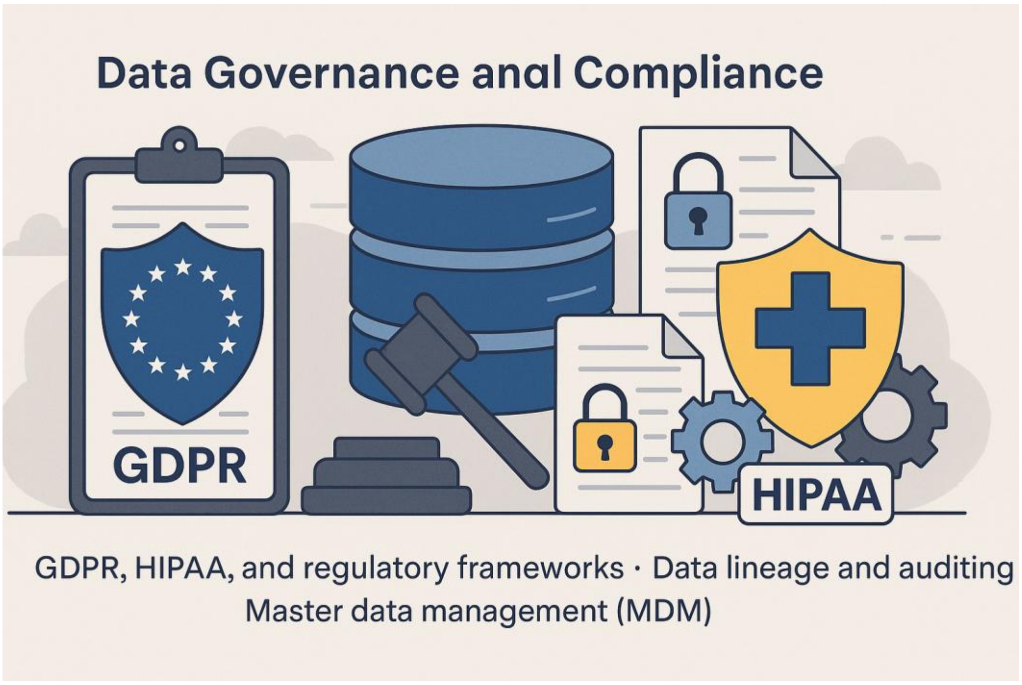
[4] Ungar, Michael, ed. *Multisystemic resilience: Adaptation and transformation in contexts of change*. Oxford University Press, 2021.

Chapter 11: Data Governance and Compliance

1. Introduction to Data Governance

Data governance is a business function that is increasingly becoming a dominant topic related to data management policies and implementation of activities. Data resources are often regarded as the new gold mines of information technology and data systems. In the era of big data systems, companies are losing control over their data resources. The general perspective on data governance is that to achieve maximum value and to make sure that data are an asset and not a liability, large scale IT and Business Management efforts need to be invested. To get a better understanding of data governance practices, the main topic of this book, a framework of various elements of data governance will be presented and discussed. The basic components of data governance are: 1. Strategic Alignment; 2. Data Stewardship; 3. Data Value; 4. Data Fiducial Responsibility; 5. Data Policy; 6. Data Compliance and 7. Data Principles. Other components can be added to this list, but they are considered as the cornerstone elements of any data governance initiative.

The term data governance is often confused with concepts and terms such as data management, data strategy, data quality, data stewardship, data architecture, and data modelling. These are all relevant topics on how to better manage data resources, but data governance is different in the sense that all of these concepts are part of the whole picture and are components related to the data governance initiative. Having said that, it is also important to distinguish these concepts from each other. Data governance is defined as the organizational function that is responsible for establishing and implementing policies and procedures related to the representation of data objects and their meaning, use and structure.


Data Governance anal Compliance
GDPR
HIPAA
GDPR, HIPAA, and regulatory frameworks · Data lineage and auditing
Master data management (MDM)

2. Overview of GDPR

The General Data Protection Regulation (GDPR) is a piece of regulation that was introduced in the EU in 2016. It sets the standard for data protection and privacy legislation in Europe and applies to organizations who process the personal data of residents in the EU. The GDPR is applicable to the processing of data that identifies or is related to persons or their personal data including name, identification number, location, or an online identifier. Further, GDPR covers information about physical, physiological, genetic, mental, economic, cultural, or social identities of such natural persons. GDPR came into effect on 25 May 2018 and replaces data protection regulations within the EU member states, establishing a unified legal regime that provides the same coverage and enforcement within EU states. Organizations from outside the EU are also required to comply with GDPR if they process data of residents in the EU. Non-compliance with GDPR can attract heavy penalties of up to €20 million or four percent of the organization's annual global turnover in the preceding financial year.

2.1. History and Purpose of GDPR

The history of the General Data Protection Regulation (GDPR) can be traced back to the need for a unified and comprehensive legal framework for data protection and data privacy in the European Union (EU). The EU Data Protection Directive was one of the first laws to regulate international transmission of personal data. It served its purpose but was dated and could not tackle the vast changes driven by technology, leading to severe criticism of the data protection framework in place. In January 2012, the European Commission announced a proposal to strengthen online privacy rights and reform the EU's existing data protection rules. The data protection reform was to provide a uniform, simple, user-friendly tool for citizens to control their information, better address the challenges posed by globalization, and give Europe a competitive edge in the emerging data economy. As citizens were demanding greater respect for their privacy, especially by businesses that were cashing in on the information themselves, pushing forward with data regulations would help restore faith in the online economy. In January 2012, the European Commission proposed a comprehensive reform of the EU's data protection rules. It includes a General Data Protection Regulation and a Data Protection Directive for Law Enforcement Agencies. Both proposals are designed to help reinforce citizens' fundamental rights in the digital age and allow companies to fully benefit from the Internal Market's potential.

The purpose of the GDPR is to give citizens back control of their data and to simplify the regulatory environment for international business by unifying regulation within the EU. The GDPR is a regulation in EU law on data protection and privacy in the European Union and the European Economic Area. It also addresses the transfer of personal data outside the EU and EEA areas. The GDPR aims primarily to give control to citizens and residents over their personal data and to simplify the regulatory environment for international business in the European Union. The GDPR is the most important piece of legislation in data privacy for the European Union. The GDPR governs how organizations use personal data, which includes anything that relates to people, such as names, pictures, email addresses, bank details, social media posts, medical information, or computer IP addresses. Organizations need to be vigilant about data privacy and security to safeguard this sensitive information.

2.2. Key Principles of GDPR

GDPR is cantered around seven key principles, which create a framework through which organizations are to operate to achieve the overarching goal of the regulation. The first principle is lawfulness, fairness, and transparency. This foundation of data protection means that organizations must have a legitimate reason to process data, need to avoid unfair processing, and must maintain transparency regarding the data processing operations. The burden of proof to satisfy the first principle lies with the organization. The second principle is purpose limitation, which means that organizations can only use personal information for the stated lawful purpose for which it was originally collected. The third principle is data minimization; organizations can only collect as much data as is necessary to serve the purpose of processing. The data must also be kept up to date, so it is accurate and not misleading. The fourth principle is storage limitation, which refers to the requirement that organizations do not store data longer than is necessary for processing. In the same vein, the fifth principle is integrity and confidentiality. Organizations need to have sufficient physical and technical security measures in place to avoid losing personal data or having it unintentionally disclosed. The sixth principle is accountability. Organizations are required to be able to demonstrate compliance to the relevant supervisory authority. Finally, the seventh principle, international transfers, refers to the additional requirements regarding transferring personal data to outside of the EEA.

The first principle together with the principle of non-discrimination establishes that any processing of personal data is only legitimate if it is performed based on one of the legal grounds explicitly provided in GDPR. Such legal grounds are consent, performance of a contract, compliance with a legal obligation, protection of vital interests, performance of a task carried out in the public interest, and safeguarding legitimate interests. Consent has drawn particular attention in the digital age where we often click “I Agree” with our eyes closed, and GDPR has introduced specific and elevated requirements regarding how organizations need to obtain consent and the rules for plus and minus options. However, it should be noted that consent is only a valid basis for more limited types of processing and each of the legal grounds listed above have specific requirements that must be met.

2.3. Rights of Data Subjects under GDPR

Article 12 of the GDPR lays out rights for individuals (i.e., the data subjects). The rights are listed in an open-ended sense in Article 12 but fleshed out throughout the rest of the Act. Because these rights are particularly important to individuals, an overview of the right is included in the following section. Accordingly, the extent of these rights should be understood as a policy goal and stretched to their possible practical limits, when justifiable. The right to be informed about the collection and use of personal information is an important aspect of transparency in a democracy. The right to access enables individuals to be aware of and verify the lawfulness of the processing. The right to rectification allows individuals to have inaccurate personal data amended or completed if it is incomplete. This is necessary to ensure that personal information is accurate and up to date, especially when decisions are made based on the information. The right to erasure allows individuals to request the removal of their personal data. It is a measure of their level of control over their personal data before something harmful or prejudicial occurs. The right to restrict processing enables individuals to stop or pause the processing of personal data when this is questioned. The right to data portability enables individuals to transfer data that appertains to them and have it easily reused. The right to object enables individuals to challenge a request to process their data or ask by what lawful basis and whether it is justified. Finally, the right to not be subject to automated decision-making includes decisions that have legal or significant effect for individuals, which are based solely on the automated processing of personal information. This is to safeguard against the automated decision that does not allow human interaction.

3. Overview of HIPAA

The Health Insurance Portability and Accountability Act, or HIPAA, was enacted in response to increasing health care costs, as well as medical record privacy concerns raised by state-led initiatives and changing to the electronic transmission and security of personal medical information. HIPAA was intended to provide a uniform national standard for certain electronic health care transactions and to protect the confidentiality and security of health information released from medical entities. As patients are more frequently exchanging personal health care data with their providers, private health data becomes increasingly open to security breaches along the electronic transmission chain, whether through wrongful data access or inadvertent third-party disclosures. Thus, with an infinite number of ways in which patient confidentiality may be breached, the purpose of HIPAA is to give patients greater access to and control over their medical records. As a result, confidentiality and security of medical records is of the utmost importance to HIPAA.

3.1. History and Purpose of HIPAA

The Health Insurance Portability and Accountability Act (HIPAA), signed into law in 1996, is a US legislation that provides data privacy and security provisions to safeguard medical information. Even though it was originally designed to enable workers to transfer health insurance plans when they change jobs, the legislation has evolved into one of the most important tools the US federal government has to protect an individual’s health information. It accomplishes this by determining how covered entities, including healthcare clearinghouses, healthcare providers, and health plans who are involved with the processing of health information, can use personally identifiable information. The legislation covers specific individual identifiable information, including name, address, birth date, and social security number. HIPAA also introduced mandatory standards for electronic health care transactions regarding the privacy and security of health data. It aimed to improve the portability and accountability of health care and protect the integrity and confidentiality of patients’ sensitive data, in part, during electronic transmission. In August 2002, the Office for Civil Rights issued the Privacy Rule, which established national standards to protect individuals’ medical records and personal health information. Following this, in April 2003, the Department of Health and Human Services issued two other final rules concerning the security of electronic health information and the establishment of national unique health identifiers for providers, health plans, and employers. In 2005, the Department of Health and Human Services introduced an interim final regulation adopting standards for the Nationwide Health Information Network. The Department of Health and Human Services incorporated additional modifications in the Privacy Rule in numerous subsequent notices, some also involving input from organizations with expertise in health data standards or patients’ rights.

3.2. Key Provisions of HIPAA

HIPAA comprises five distinct titles, with provisions covering a wide array of issues relevant to the healthcare services and insurance sectors. Some of these provisions stipulate broad policies about how the country’s health information will be managed, while others are specific rules governing the use of information, what will happen to organizations in violation of privacy principles, and how states’ rules will relate to HIPAA privacy policies. The five titles are as follows: Title I prohibits the use of preexisting condition exclusions by group health plans, imposes stringent guarantees of health insurance portability, and in general protects the employee’s right to change jobs without facing financial consequences. Title II establishes a set of nationally mandated rules governing the use of electronic data interchange in the healthcare process. It defines a structure for the transfer of administrative data electronically between payers, providers, and other entities. Title III covers issues of fraud and abuse in healthcare, including penalties. Title IV consists of the provisions that regulate health insurance coverage. Title V covers miscellaneous provisions relating to the healthcare industry. Although it has five titles, much of HIPAA is devoted to establishing the proper procedures for identifying waste, fraud, and abuse in healthcare services at a national level. In fact, the most visible aspect of HIPAA thus far has been the establishment of national guidelines, which are designed to ensure the confidentiality of personal healthcare information. These guidelines were established partly as a way of minimizing prohibitive costs and competitive disadvantages that result from non-standardized EDI transactions.

3.3. Patient Rights under HIPAA

Understanding the rights provided to persons under HIPAA provisions is important for owners of personal health information. The specific provisions that govern patient rights are, increasing access to the information covered by HIPAA, limiting the purpose of use and disclosure of the information by covered entities, the right to restrict the release of certain information, the right to request changes to protected health information, and the right to file a complaint with the Department of Health and Human Services.

The Patient's Right to Access HIPAA provides individuals or their representatives with the right to access protected health information in a designated record set and obtain a copy of that information. A designated record set is defined by HIPAA regulations, which says it is a group of records that are shared by a healthcare function for making decisions about individuals. Individuals are therefore allowed to review and obtain copies of their Designated Record Set. The request can be made for the records in any format, and the covered entity should comply, if it is not technologically infeasible to do so. If a healthcare provider denies an individual access to records, the individual has the right to modify that decision.

The Patient's Right to Limit Use and Disclosure Affected individuals have the right to request restrictions on HIPAA covered entities' use and disclosure of their health information. Accordingly, health plans are obligated to comply with any individual's request to restrict disclosures of protected health information when the disclosure is related to payment of healthcare expenses by a party other than the individual or is to carry out treatment for the individual. However, covered providers are not bound to comply with the request.

4. Comparative Analysis of GDPR and HIPAA

Similarities between GDPR and HIPAA While understanding GDPR compliance, HIPAA may recall the fact that there are existing laws on Data

Privacy in the United States. HIPAA is more focused on the protection of patient health information, while GDPR is a broader data privacy regulation. The core goal of both laws is to protect data subjects' data privacy and security by imposing obligations on businesses that handle the data subject's data. GDPR penalties are steep; however, HIPAA also has hefty penalties, especially for repetitive and serious offenders. In the case of HIPAA, data breaches may lead to prisoners-imposed sentencing as well. Both GDPR and HIPAA laws cover a range of data privacy provisions; therefore, it is advisable to comply with both laws if applicable. Also, both HIPAA and GDPR provide for third parties whose requirements must be fulfilled to have the bundle being compliant with the laws.

4.1. Similarities between GDPR and HIPAA

The GDPR and HIPAA are two regulatory frameworks that set standards for the protection of sensitive data and establish respective liable penalties in case of data leakage or misuse. The two regulations have several commonalities and use a similar vocabulary to establish the rules for collection and processing data. Therefore, service industries or companies that in any way assist other organizations with the management of sensitive data might confuse following one regulation with following the other. As both GDPR and HIPAA share the same essence, we will dive deeper into the particulars of the two frameworks to provide clarity on the spheres of influence of the two data regulations.

While the HIPAA regulates the use of sensitive healthcare information, the GDPR serves the purpose of regulating any data related to European citizens. The GDPR focuses on the protection of personal identifiers, while the HIPAA emphasizes the importance of protecting personal healthcare-related identifiers from a healthcare business associate. In other words, any organization is subject to the GDPR regulation, while only healthcare providers and businesses closely related to healthcare and health insurance are mandated to follow HIPAA guidelines. Because of that, many companies must handle both regulations and ensure compliance with both personal data protection frameworks. The GDPR is a far-reaching regulation that includes all organizations that operate in the EU or offer goods or services to EU individuals, whereas only certain kinds of organizations such as healthcare companies and service providers are subject to HIPAA restrictions.

4.2. Differences between GDPR and HIPAA

GDPR and HIPAA are policy frameworks from differing jurisdictional scopes, covering different types of privacy and data operations, and providing different possibilities for violations, consequences, and redress. GDPR is an umbrella policy framework on general data protection, while HIPAA is a mandate on specific health data privacy and protection. However, both policies show similarities in spirit, if not in letter.

GDPR is a law on data protection and privacy in the EU and the European Economic Area. GDPR regulates the processing of personal data and the free movement of such data. It also covers the export of personal data outside the EU and EEA areas. GDPR is cantered on the individual or data subject, both within and outside the EU/EEA jurisdictional orbit, allowing data subjects significant privacy rights. HIPAA is a U.S. enactment that specifically regulates the protection and privacy of PHI data within the healthcare sector, with massive enforcement provisions. It specifically addresses the protection of confidential patient information in the healthcare sector. Although HIPAA has limited extraterritorial effect, it nevertheless affects foreign companies that conduct business in the United States and create, receive, maintain, or transmit health information in the course of providing healthcare services.

5. Regulatory Frameworks and Compliance

Enforced by respective supervisory authorities, regulatory frameworks establish legally binding requirements on individuals or organizations that must be met. While regulatory scanners are available that keep organizations updated around current and future requirements, there are more than 600 widely-adopted standards that help organizations avoid non-compliance with legally binding requirements or demonstrate due diligence in the case of a data breach. These standards also define best practices and recommendations around the governance of an organization's data holdings. Regulatory frameworks can either be horizontal, addressing a large set of organizations across many industries while focusing on much general principles, or vertical or industry-specific, defining roles and responsibilities for participants in a specific industry. Horizontal regulatory frameworks often define how an organization should set up its internal controls and what kind of audit trails should be established, leaving it up to organizations to define sufficiently stringent internal controls and necessary audit trail data architectures to conform. Industry-specific regulations, revitalized by extensive regulations introduced during and after the financial crisis, are usually much stricter, listing and detailing required controls and audit trail data elements and giving organizations little flexibility in defining their data governance policies.

5.1. Understanding Regulatory Frameworks

Regulatory frameworks are acts and regulations, typically holding the force of law, that are published by a regulatory or standard-setting body and that establish a minimum compliance requirement for covered organizations. In most cases, these frameworks apply to organizations in specific industry verticals or geographic regions, whose lack of adherence would cause unacceptable risk to the governmental authorities or to the larger society and economy, should there be a breach of the organization's data.

Examples of regulatory bodies that publish data management frameworks and associated recommendations or practices for specific verticals are government organizations and various standard-setting bodies. Organizational examples include international standardization entities and security standards councils. Examples of data privacy and protection frameworks include regulations enforced by government agencies with financial and remediation penalties, among others, for failure to comply with the regulatory requirements.

5.2. Compliance Challenges in Data Governance

In implementing a formal data governance initiative, organizations will address a series of related data governance compliance challenges. Each of these discussions examines a necessary and sufficient condition for an organization to achieve optimal data governance alignment with the data governance regulation framework. In the completion of this chapter, we will answer the following high-level questions:

- What is the relationship between organizational compliance and enterprise data value? - Is data governance a cost or an investment? We apply a holistic perspective to the questions: What are the roles and roles of technology in data governance compliance? - Do we need compliance silos or aligned, integrated capability levels? - What functions and business units should operate, lead or guide the data governance compliance program?

The questions are grouped in pairs. Both questions are concerned with goals. The first question of each pair seeks to confirm that data governance compliance is aligned to what is ultimately socially, economically, and technologically valuable: optimum business and technology outcomes. The second question of each pair seeks to propose potential provisioning modes of the data governance compliance operation. Data value focuses on what really creates value to an organization and how to best enhance this value for the business. This alone justifies the interest of executives in addressing this issue. The best answer to this question, however, also declares the type of answer we expect to submit to the second question of our sequence.

6. Data Lineage and Auditing

Data lineage broadly refers to tracking the origin and the transformations of data elements within an organization. Data items are often transformed based on specific business rules, using ETL processes and through data processing pipelines, before residing in analytical platforms or data warehouses, from where they are queried for decision making. Data lineage aids in documenting these transformations, so that the various data stakeholders can follow data item movement along its lifecycle. This process is indispensable for a range of activities, from regulatory compliance, to testing data modelling changes, to estimating the impact of data inaccuracy or corruption on decision making. The growing body of data protection regulations underscore the need for proper documentation of data lifecycles, from origination to usage within organizations. With higher reliance on data for decision making, organizations have much more stringent data accuracy and security requirements than before, including for sensitive data, which underscore the need for documenting data governance efforts.

Data lineage can be defined in metadata, which describes how the data within an organization change over time. Traditionally, data lineage has been documented by Data Stewards or Data Analysts through either manual data tracing, or by using tools for data lineage tracking, both of which are tedious methods and need to be done continuously. Automated tracking techniques have recently emerged that can track data processes without manual intervention. Data Lineage solutions use tracking to identify the tools and transformations that have created, transformed, or deleted data. While good data quality can be ensured through other techniques, combining this with data lineage techniques help create a more robust system. Accurate tracking of operations on data enables organizations to inspect or audit the data at any point in time. This is important for organizations with Operations Databases, from which Enterprise Data Warehouses are refreshed routinely.

6.1. Importance of Data Lineage

Data lineage refers to the process of knowing where the data is coming from, where it is moving to, the transformations taking place on that data during the process, and how it is related to other data. Data lineage helps organizations in having a holistic view of understanding their data. Many organizations are utilizing multiple sources, tools, and systems for data processing and analytics. These sources can be anything like ERP systems, databases, cloud storage, etc. These systems may run on-premises or on the cloud. The different types of tools include ETL tools and data warehouses. Outsourcing new data services and processes may seem a good idea, but not knowing the downstream effects makes it practically impossible to guarantee the quality and safety of the data involved. Although the positive effects of good Data Governance have been well documented, many organizations are still struggling with the practical application of Data Governance Principles and Frameworks.

In this modern world of Data, privacy and compliance have become essential for organizations. With the introduction of laws and regulations, Data Compliance and Data Governance have become critical services with data lineage and auditing services being vital components. In the time of Cloud and Outsourcing, businesses are moving toward the Cloud for analytics, business intelligence, and AI/ML tools to gain insights from structured, semi-structured, and unstructured data. These analyses are highly dependent on the quality and safety of that data. With Data Governance Principles being applied at the organization level, organizations are looking for Data Governance solutions and Core Data Infrastructure solutions to implement and put these principles into practice.

6.2. Techniques for Data Lineage Tracking

Organizations need to be proactive as they undergo data-driven transformation and to compile a set of data lineage best practices and data lineage techniques that they can implement at various skill and resource levels. There are generally three ways in which data lineage is tracked: manually, through reporting tools and metadata repositories, or programmatically. Manually tracking data lineage can often be as simple as creating visuals on a whiteboard that show how the data for a given report is sourced and then processed within the report-building tool. More complex is involving several people for each report writing down their understanding of how data flows throughout their organization and compiling that into a searchable database of reports and dashboard dependencies, or a programmatically created, visually rich hierarchy of dashboards or reports linked to the database tables that serve as their sources.

While the basic graphical interfaces developed for business intelligence tools may help when first learning reporting tools, connecting to the actual database for these tools can yield far richer results. However, these tools can often disturb normal operations of shared, critical production systems, so tools that pull data lineage information from above the tool level to maintain database size and availability are often preferred, and data lineage tools are among the many tools implementing the second solution. Organizations have also implemented variations of the second solution as publicly available, open-source tools.

6.3. Auditing Data Access and Usage

While many organizations have automated policies that restrict access to sensitive data such as highly confidential personnel records or protected health information for medical patients, most have limited visibility to understand whether employees are accessing those records for legitimate business reasons. Human resources, finance, and clinical system personnel often have oversight and review of employees' sensitive data, but deep auditing capabilities are often lacking to ensure compliance with regulations. In health systems, logical access right audits are often performed by industry-prescribed tools, but that process could be greatly elevated by extending the capability to provide end-to-end lineage for those sensitive data sets across disparate systems. And once a user accesses sensitive data, those activities should be "marked" in the system so that if they access that data outside of regular access protocols at unusual hours, a warning could be issued to provide visibility to potentially nefarious behaviour.

External protections such as firewalls and VPN are one path to restricting unwanted access. But compromising a staff member's login credentials is a common pathway for breach attempts. So having cross-systems tracking services and a data lineage tool should be checked off in your data strategy. Data governance should not only seek to protect the health systems' sensitive data but should also promote appropriate use of that data. Advanced data lineage tools should be involved at a more profound level than providing a view of data attributes for discovery and brainstorming. Foundational data sets such as claims, encounters, medications, diagnoses, and other health system sensitive data elements should be subject to strong data access and usage auditing controls. At minimum, a self-auditing model should be in place to report how many times sensitive data has been accessed based on the ownership of the data set or driven query logic on demand.

7. Master Data Management (MDM)

Master Data Management (MDM) Data governance comprises a complex of roles, policies, processes, and technologies within an organization and assigns responsibility for decision-making and for ensuring the consistent and appropriate use of data across the organization [1-3]. MDM refers to the processes, governance, and tools required to create a trusted view of an organization’s critical data. An organization must leverage these tools to consolidate, update, and manage the integrity and custodianship of central reference data subjects such as customers, accounts, products, employees, and vendors for which it maintains core shared business operational processes and business applications. The challenge in any organization is that reference data about key business subjects is touched by many data producers. Without proper insight, guidance, and ongoing custodianship, this data can become disorganized and disparate, leading to inaccurate and insufficient operational applications. To be effective, master data management requires an organization to explicitly define the roles and responsibilities for the stewardship or custodianship of key reference data domains as well as the overall MDM processes that connect and integrate those data domains. Data governance policies must regulate not only the organizational owners but also the structure, management, and quality of the data itself. Data governance frameworks guide organizations in getting the most from their data in a seriously regulatory world. Well-designed approaches can offer organizations a roadmap to use data in precise, consistent, and valuable ways that address business strategies, initiatives, priorities, resources, and risk tolerance.

7.1. Introduction to MDM

The term master data management (MDM) represents a set of processes and technologies that aim to provide a consistent view and usage of enterprise-wide master data. An organization's master data may include details about products, customers, suppliers, accounts, locations, and other entities that are critical to its operations or reporting and that support cross-functional processes and may thus span multiple operational systems. MDM typically consists of at least the following components: an inventory of important master data, a detailed description of the content and format of each attribute, a workflow for approving changes, a repository of approved master data and rules for its distribution, policies and best practices for creating and maintaining master data, and a list of systems that use or provide master data. The root cause of many operational and reporting problems is the poorly designed, poorly implemented, and poorly governed master data that are the example attributes of interoperability.

Throughout industries and companies, almost every business function must make decisions influenced by external data. Company size does not appear to mitigate the issues either; despite their size and the significant systems investments they may have made, many global organizations still struggle with dirty data issues. The need for diverse but inconsistently managed data is at the heart of many problems—poor customer service, distrust of management reports, and inability to integrate acquisitions. As operations people wrestle with a lack of trust in the data, they may introduce process workarounds and increased control, both of which add to the company costs. And yet decisions must be based on the available information, and when it is inaccurate, the decision may be incredibly detrimental to the organization.

7.2. MDM Strategies and Best Practices

MDM Strategies and Best Practices. There is no one unique strategy that organizations need to pursue for MDM. There are, however, some typical approaches that projects have taken that can be applied to guide projects in addressing certain common needs. For example:

Data MDM can be built on operational needs – Examples of operational needs and the nature of these could be in an enterprise transaction retail application that needs an up-to-date view of product information to enable product order and return transactions to be done efficiently. It could also be in an insurance application that needs the latest information on the insured to be able to issue new policies and make premium quotes. For this organizations have built operational data stores that contain master data. These are special transaction databases that keep the master data in sync with the parent systems.

Data MDM can also be built on analytical needs. In this case, master data are being utilized for data integration and reporting to support key enterprise processes such as marketing, promotions, etc. In the analytical context, the customers, product definitions, promotional activities, marketing segments, etc. are typically of high interest. They are usually involved in marketing, channel management, customer service, or corporate performance monitoring.

Now that organizations have experience with MDM, they should build a strategic approach to MDM. The strategic approach of MDM should start addressing the needs for operational systems and analytical systems. The two must converge responsibly towards the concepts of a single view of master data that are accessible to both operational and analytical systems within the enterprise systems architecture.

7.3. Role of MDM in Compliance

Master data management (MDM) is a comprehensive and precise approach of IT and the business that ensures the uniformity, accuracy, stewardship, and accountability of the enterprise's official shared master data assets. Besides being an IT-enforcement initiative, ensuring correct and properly informed data environments, MDM reflects the business's executive functions and its best practices and standardization initiatives. Today, an increasing number of organizations perceive MDM as a means of complying with both enterprise-wide regulations and industry-specific regulations. MDM is perceived as a significant enabler for a variety of important regulations.

These organizational regulations require the creation and maintenance of a single, accurate, complete, and current view of an organization's key business entities, such as customers, employees, vendors, and other third parties involved in business transactions and processes. MDM is also seen as a key technology for industry-specific regulations that impose specific data-related requirements on financial services. These requirements include the maintenance of accurate records of customer identification, verification that the payer involved in any financial transaction is not a known suspect involved in terrorism or other illegal activities, tracking of specific customers who transfer more than $10,000 in one transaction, and properly storing business records to enable effective auditing tasks.

It is widely recognized that business success in the financial services industry over the past decade has been driven by a strong corporate culture cantered on accountability, compliance, and the effective governance of enterprise-wide initiatives, such as risk management, data management, and business process management. Regulatory compliance efforts are often focused on increasing corporate compliance and governance capabilities. Industry organization and compliance and risk management are increasingly viewed as the hallmark of successful business within the finance industry.

8. Integration of Data Governance Frameworks

The clarity of regulations such as the GDPR and HIPAA is compromised by their non-prescriptive nature, leaving organizations vulnerable to regulatory scrutiny. This lack of clarity can impede organizations' ability to uphold their promise of protecting customers' data, thus missing the marketing potential of promoting compliance with existing regulations. Organizations need regulations to assist them in avoiding data breaches and, once a data breach occurs, laying the groundwork for a stringent response plan. What is needed in tandem with such regulations is a cohesive Data Governance Framework that blends regulation with best practices for data management and protection.

Because the regulated environments of the GDPR and HIPAA are different, there are distinctions between how the regulations may be achieved. However, there are also many similarities and points of intersection that allow adoption of frameworks to be leveraged for building a data governance strategy across not just a single revenue generating area of the business, but for the entire organization. By leveraging a well-respected framework, an organization can avoid data governance pitfalls and instead, pursue a well-defined multi-stakeholder, business-friendly agenda, across the regulated vs non-regulated boundaries of the business.

These benefits also apply towards other frameworks that must be implemented by organizations to stay afloat and stay compliant. While proper risk management principles should be applied towards all facets of data governance, including cybersecurity, Data Quality, Data Privacy processes, and Compliance, the resulting Data Governance Framework should tie together all the standalone siloed efforts into a cohesive structure for responsible sharing vs use of enterprise data. Weaving such a framework shall not only maintain not just regulatory compliance but also good business sense throughout, the Return of Investment on the multi-stakeholder effort needed to set such a strategy is paramount.

8.1. Aligning GDPR and HIPAA with MDM

The emerging capability of Master Data Management (MDM) technologies to support an organization’s ability to protect and govern key enterprise data, along with its compliance with various local and international regulatory frameworks, should be given proper attention and considered when selecting and enhancing the MDM from both technology and business points of view. The ability to have a trusted, reliable, accurate, timely, and properly governed Master Data is an important step in the right direction for any industry. In some zones, such as regulated industries, not being compliant poses important risks and repercussions for the companies because of imposed fines or damages exposed by the enterprise.

Usually, in companies operating in a regulated environment, such as the Health and Life Sciences vertical or even enterprises dealing with Business Licenses and Regulatory Taxes, authorities impose penalties, especially for excessive delays or incorrect information. Knowing that regulatory bodies often use third parties to perform investigations, it makes sense to invest in a specialized capability that ensures data can be used by various operational and analytical use cases while adhering to desired regulatory and quality standards. In addition, such compliance improves efficiency and ultimately reduces operational costs and the risk of errors. While there are various MDM building blocks, both on accommodations and solution levels, these address specific industries; companies applying MDM across other industries may also gain advantages on the operational side if they would adapt implemented processes and guidelines to the regulatory organization’s recommendations.

8.2. Implementing a Unified Data Governance Strategy

The variation and integration of enterprise data increases both its value and management complexity. Privacy and security regulations provide direction for required data protection measures such as data classification, usage and purpose, data access, and data sharing. A unified enterprise data strategy provides a coherent approach—across enterprise data—and direction for enterprise-wide data management initiatives. With mechanisms and organizations in place for the governance of enterprise data across its life cycle, enterprise data can help develop and drive business strategy while ensuring regulatory obligations are met. The creation of a shared understanding of enterprise data maximizes its value. Data models link business concepts to the data organized within information management systems. These models describe business operations, information flows, and data usage for business intelligence and decision-making. By providing a common vocabulary within a business area, they help ensure enterprise-wide consistency and minimize data ambiguity for subject areas. This, in turn, enhances communication, maximizes the value of data, and lowers costs associated with managing data. Data classifications, established by the data governance committee, enable compliance with data privacy and protection laws. In addition, a record of processing created for GDPR, when combined with data models, allows an organization to leverage data's value within operations.

9. Case Studies

Compliance with regulations is easier to articulate than to implement. Below are a few briefs, non-exhaustive notes on some projects related to key areas of focus. They should be of assistance in clarifying some of the more technical and implementation-related aspects of the work performed in the fields of data governance, compliance theory, and its practice.

9.1. Case Study on GDPR Compliance

Data catalogues underlie the key technical characteristics of many data regulation compliance capabilities within enterprises. These catalogues connect themselves to internal and external data sources and perform classification and related activities allowing relevant data assets to be classified and tagged with the regulation rules that they may infringe.

Implementation of data catalogues is possible via a web-based application, which can either rely on an internal data source context or a broader semantic knowledgebase of external and third-party context. This functionality calls for the combination of machine learning and web scraping to allow end-users, the enterprises' business analysts and data engineers, to visualize aspects including usage, possible classification, views of privacy rules of any data item within the database, and sources of intrusion into private and sensitive data.

The General Data Protection Regulation, or GDPR, is a powerful piece of legislation and non-compliance is generally recognized as being a bad idea both commercially and financially. Many law firms promote their services around companies needing to comply with GDPR and being able to prove this for or even provide mechanisms for clients to audit these providers using tools and techniques to prove compliance. Most organizations recognize the need to comply with GDPR but may not know how to do it or what resources are needed to accomplish GDPR compliance. GDPR compliance requires not only technical controls, but also legal aspects, and provide the human resources around the process, legal, and technical mechanisms for ensuring and providing proof of on-going compliance. In researching the components, humans, processes, and technology to deliver GDPR compliance, we found several organizations writing about mapping technology to assist in GDPR compliance. We also found a collection of technical tools addressing some aspects of this compliance, along with several organizations and consultants selling services enabling organizations to comply with GDPR. Further research showed that most of the organizations or consultants assisting with GDPR compliance were also selling technology solution. The technology components necessary for GDPR compliance were found to fall into four categories as core components involving technology solutions: get the data, protect the data, discovering and mitigating violations, and responding to violations.

9.2. Case Study on HIPAA Compliance

As a back-end layer, deploying rules on privacy either on input/output to the organization, or in relation to analysis operations (especially sensitive operations such as profile discovery, that is, might infer future courses of development for individuals whose data are stored in the databases) require discovery capabilities.

The implementation of compliance solutions or documents for privacy, regulation, or security are designed to address a complex set of requirements within a specific framework. Due to this complexity, organizations will likely find it difficult to construct supporting documents from the bottom up. Conversely, it is unlikely that an organization can or should utilize existing compliance templates. A better approach is using templates as guides for developing or validating an organization’s documents and processes. Below, I provide an example process used to create its current compliance documents and security processes. The example is inspired, but not directly quoted from guidance.

The organization incorporates cloud-based platforms supporting both Document and Unified Communication Voice Infrastructure. The compliant solution includes the Document and UCVI application services deployed with a cloud service provider. The organization chose this provider to comply with relevant acts and leverage its infrastructure, methods, and processes already existing to support compliance with partners.

The business model requires the transmission and storage of PHI as defined in the Privacy Rule. The organization is a Business Associate and therefore is covered under the Business Associate Law. As a satisfaction guarantee, the organization is committed to providing responsive, reliable, and hassle-free services to our customers. Staff members and associates who do not comply with the policies and procedures as specified in the Security Rule violate the rules of patient privacy and digital confidence. This can be regarded as "one strike and you're out," as there are no acceptable excuses for breaching the privacy barrier.

10. Future Trends in Data Governance

The future of data governance is multifaceted, engaging with both disruptive tech and the regulatory framework that seeks to rein in the challenges posed by technology on data rights and responsibilities. The pace of change in governance models across the corporate and public sectors is such that existing models may no longer suffice. Penalties are being levied on regulators as they are slow to respond to the credible threats posed by AI technologies. We must explore a new dynamic for the identification of charter values, the redistribution of rights and responsibilities, and the formation of governance ecosystems.

Disruptive technologies such as blockchain, artificial intelligence, virtual reality, and others will continue to change how information is stored, accessed, and used across public and private sector organizations for the benefit of society. Where these technologies offer significant opportunities for positive change, at the same time they also present challenges around security, identity, privacy, data rights, and what it means to be human. In this changing world, rules and regulations alone will not suffice to govern data and address all concerns. Other mechanisms, such as codes of ethics and charity will play a role as well. While governments are mandated with the responsibility for lawmaking, the diffuse and evolving nature of many of these technologies requires networked, engaged governance actors for accountability and effective checking of data rights and responsibilities. Accordingly, we can expect the governance landscape to evolve over time in a decentralized manner, with some aspects similar to the way social media has evolved.

10.1. Emerging Technologies and Data Governance

As governments and other regulatory authorities around the world grapple with how to work with the private sector regarding prevention of social harms, privacy atrocities, and failures to investigate and prosecute criminal behaviour, some technologists are exploring the use of emerging technologies such as blockchain and machine learning to encode compliance and governance into systems—they are building trust layer protocols that closely intertwine compliance and governance into the way any application or protocol handles code and data structure, and serve to ensure that compliance requirements are fulfilled as part of application behaviour and that governance structures enforced by the application are carried out. In addition, some of the private sector companies offering trust layer technologies are decentralized organizations pursuing a public policy mission to provide a layer of technology to support compliance and governance for other companies at lower cost, allowing them to redeploy money previously allocated to those functions into their core business. However, these technologies do not operate for free. Government and regulators will have a central and essential role to play to enable the adoption of these technologies by business and the use of the outputs of those technologies for monitoring social harms and illegal behaviour. The regulation will also have to balance the need to not put business using trust layer technologies at a competitive disadvantage with international competitors who are not required to use them as part of their operations. By adopting these technologies, business can offer services in a more efficient manner, allowing for lower prices and better quality.

10.2. Predictions for Regulatory Changes

In the short-term, the focus of organizations will be on developing standards for the application of existing regulations. Primarily, this means revisiting implemented marketing systems to demonstrate compliance with GDPR. This includes revisiting the consent/selection and marketing updating process, ensuring that it achieves an acceptable standard. It is also likely that private sector organizations will publish standards against which third parties can be certified as compliant with GDPR, although it remains to be seen whether adherence to such standards an acceptable approach to regulation is. In addition to updating marketing approaches, organizations will also be busy developing data breach response plans, given the vastly increased financial penalties that accompany non-compliant data breaches.

The development of an EU Data Protection Agency is likely to support member states in their efforts to introduce regulatory approaches and associated punishment structures. It may be that a special EU court is established to handle the many cross-border data breaches that are likely between EU member states. If so, the imposition of fines will take the form of an international multi-country legal action, as is already the case for certain levels of negligence and harm.

The prospect of multi-data-breach fines for organizations makes it more likely that organizations, with multinational footprints, will be fined by the EU or US authorities than by either. Organizations with multinational footprints are already aware of the security policies that those authorities are laying down. They are already answering requests for data or have publicly disclosed in compliance with new privacy regulations.

11. Conclusion

In many respects, this book is a primer on several information protection laws that impact the regulation of health information. These laws present overlapping — yet distinct — requirements that organizations utilizing health information must consider as they turn their attention to compliance, as well as the governance processes and technology that support such compliance. Each of the laws discussed in this book has its own focus and nuances that an organization must navigate. Nonetheless, there are several shared attributes that information governance professionals can take away from this book that provide at least a modicum of direction. The regulatory frameworks discussed in this book would benefit from harmonization, but such harmonization is unlikely given the present political, economic, and social climates. Furthermore, note that much of the focus of this book has been directed at large organizations with large budgets that can afford the full gambit of compliance-related costs and associated activities. But there are also numerous small organizations in the space who are just starting on their compliance journey — or who desire to completely overhaul their compliance programs — and lack the funds to undertake complex and sophisticated strategies and activities. Whatever the size of the organization, compliance programs should not be overly complicated or provide more information than is necessary to accomplish the goals of the enabling regulation. Unless required by the enabling regulation, the proposed compliance activities and programs need not be any more complicated than is necessary to meet the goals of the regulation, considering the size of the organization and the risk of the activities to consumers and others.

References:

[1] Khatri, Vijay, and Carol V. Brown. "Designing data governance." Communications of the ACM 53.1 (2010): 148-152.

[2] Simuni, Govindaiah, and Amaranatha Atla. "Hadoop in Enterprise Data Governance: Ensuring Compliance and Data Integrity." Available at SSRN 4982500 (2024).

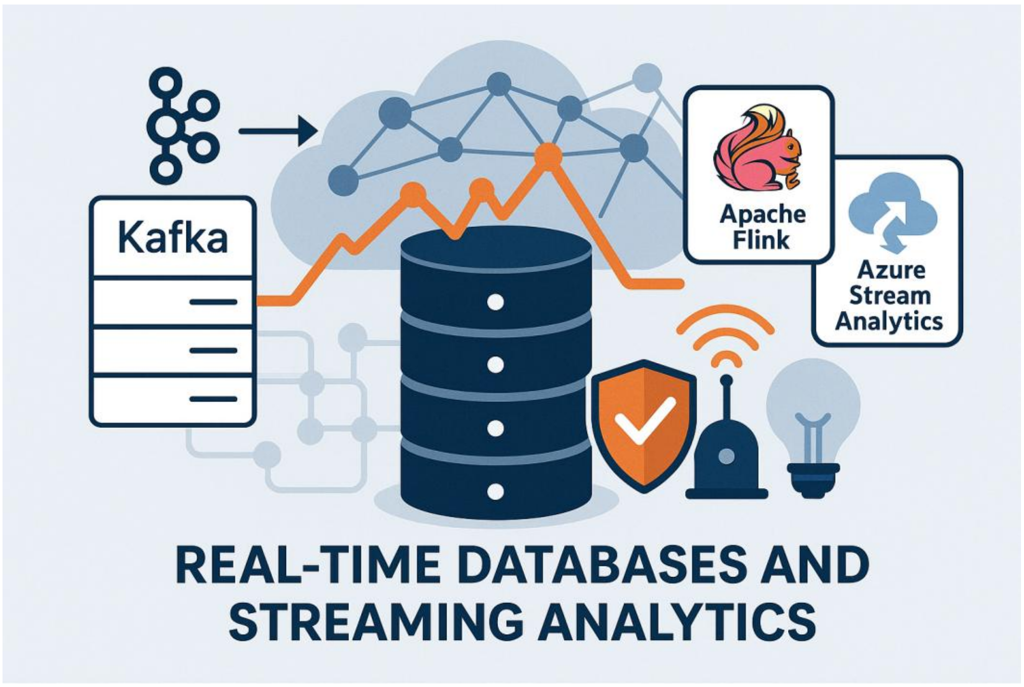
[3] Janssen, Marijn, et al. "Data governance: Organizing data for trustworthy Artificial Intelligence." Government information quarterly 37.3 (2020): 101493.

Chapter 12: Real-Time Databases and Streaming Analytics

1. Introduction to Real-Time Databases

The increasing adoption of low-cost sensors, RFID-tagged assets, and social media is enabling industries and organizations to easily and continuously collect, and thus generate, large amounts of real-time streaming data. Significant amounts of this data come from a wide range of cyber-physical systems, including vehicular traffic systems, smart energy/utility grids, smart buildings, health-care systems, environmental monitoring systems, disaster response systems, and manufacturing systems. The process of continuously receiving, generating, collecting, and eventually analysing this data represents the next big opportunity for industries and organizations. However, continual improvements in hardware along with innovative frameworks that are optimizing collection, storage, and processing of this data are creating some fundamental questions about design and use of infrastructures. For example, the convergence of the Internet of things and Big Data is creating fundamental questions about the future of event/message-driven systems. An emerging need is for specialized data management platforms that can efficiently manage specific forms of streaming data and help applications query and/or process this data without undue delays.

Traditional databases have focused on providing ACID transaction facilities, while advanced query processing techniques have been provided for non-real-time data. As organizations process increasing amounts of data in near-real time, the need is for transaction management and query processing techniques that can efficiently handle real-time data along with classical data [1-3]. Researchers have explored diverse sets of algorithms and architectures that can provide specialized storage and processing capabilities for real-time and streaming data. The objective of this text is to describe research and development that have enabled the evolution of real-time databases and provide state-of-the-art techniques and algorithms that have pushed the boundary of real-time and streaming data management techniques within both academic and industry environments.



2. Overview of Streaming Analytics

The broad paradigm of Data Analytics encompasses many areas of research. Statistical analysis, general data mining, predictive analysis, and high-performance data warehousing, such as online analytical processing systems, are just a few specialized fields. Streaming Analytics, or Stream Processing, is one of these areas. It focuses on the challenge of processing possibly large volumes of data in real-time. This requires that the latency from event generation to event processing result output be small, and the processing of events is done as closely as possible to their generation. In this essay, we focus on Streaming Analytics over data that come in a Stream or as Event Messages that are managed by a Real-Time Database System. Data Streams include the following characteristics. First and foremost, Stream Data arrive dynamically over time. This Stream would eventually cease in some defined future window, i.e., the Event for that Stream would eventually be Ended/Closed. However, the Stream essentially behaves like a single-row table, with a new row being inserted at every time instant until its Closure. These time-ordered rows in the table represent the changing state of a single entity in the real world.

Streaming Analytics is the process of querying, analysing, and extracting insights from Data Streams. Data Streams are large or unbounded collections of events that are generated continuously in a temporal sequence. Querying Data Streams differs dramatically from conventional Query Processing techniques, which query static, finite datasets. Because Data Streams are continuously changing, a query may deliver very little detail, such as the current value of an attribute. Moreover, a Data Stream may need to be queried in a batch fashion or in a continuous fashion, meaning that results are maintained based on the current values and constantly updated as new event messages are aggregated. New incoming Data Stream events are processed incrementally, as they arrive.

3. Apache Kafka

Request for Correction: This text mentions that should be cited which come from other scientific research papers. Please revert it to previous state, corrected: better keywords with scientific contents, having this text be quoted proposing the introduction itself, segue before the article body or another research topic in current paper, complying the IEEE format. Thanks! The more I read the introduction, the more I realize that in the end it should be an introduction to something bigger, because it has lots of keywords, however, it is all mixed with keywords referring to itself: Keywords: Kafka, Message queues, Real-time, Stream processing. Abstract - Apache Kafka is a distributed platform derived from one of the projects which consist in collecting streaming log data. Initially developed for processing large amounts of data in real-time, today it is also being used for durable storage of logs due to its high performance and the disk space savings that it can provide. Thus, it is not only one solution for stream processing that has relatively low latencies, but also a new durable and scalable log service. Its popularity has increased Fastly within organizations of different sectors, due mostly to its design capabilities of dealing with huge quantities of data, fast, scalable and fault-tolerant and to its Integration with Stream Processing Engines and Batch Processing Frameworks. More than a message queue, it is much more than just streaming processing capabilities. This paper presents a complete overview about Apache Kafka. We focus on its architecture, the concepts and components of the system, the ecosystem of tools that has arisen around Kafka and the adoption of Kafka by large and small companies. Finally, we present Kafka's limitation and challenges and possible future improvements.

3.1. Architecture of Kafka

Apache Kafka is a data streaming solution. It serves as a data transport tool as well as a persisting unit at the same time. Kafka needs to be able to persist the data in a fault-tolerant and durable manner. For Kafka to meet the goal of real-time processing of the data, it needs to support horizontal scalability. Moreover, Kafka operates as a pull-based streaming platform in which the user needs to tell Kafka about the offset of the next record to process rather than Kafka pushing the data whenever it is written into the system. This architecture helps Kafka achieve performance because multiple consumers can read from a topic at the same time, making it happen to achieve data locality.

Kafka achieves the design goals through the architecture that consists of brokers, topics, partitions, producers, and consumers. Kafka runs as a server that holds and transfers data. It allows multiple servers to be added into a cluster to achieve horizontal scalability. The replication model built on top of the partitioning model provides the fault tolerance feature of the platform. To achieve higher throughput, Kafka allows the producers to write messages in bulk. The consumers also pull data in batches and maintain the offsets to support distributed consumption in which multiple consumer instances could participate to divide the work. In addition, the use of the LZ4 and Snappy libraries help to achieve a balance between the throughput and the network footprint.

The Kafka broker is the core unit of a Kafka cluster. It hosts the Kafka topics and accepts the read and write requests from the producers and consumers. Deployed as a cluster of Kafka servers, brokers are responsible for persisting and decrypting the stored messages and routing the user requests. A Kafka cluster is made up of multiple brokers that provide the capability of fault tolerance, message retention, and burst-processing load. The user can transparently connect to a broker to interact with the cluster using the producer and consumer APIs.

3.2. Use Cases of Kafka

In addition to its usage as a messaging queue, Kafka has proved to be quite useful for an existing variety of different use cases. "Log Aggregation" is one of the first use cases identified, with many companies and organizations doing it at huge scale. Applications generally print to the standard output or standard error, which alias a terminal-based console. Console I/O is both slow and transient, so that we lose log data when a machine crashes. Because application Output Stream and

Error Stream are not referenced in the application code, applications cannot also communicate with other distributed applications.

But there are many cases of applications that do "log aggregation", using the "logging frameworks" offered by their languages. These logging frameworks write to files that the applications can process, which allows addressing specific challenges. In these cases, handling logging is a separate service. Applications offer logging traffic but add little to the cost of processing: logs are written as files as needed, and applications communicate with routing commands when they want to.

The general pattern here is "write logs to the file system, and hands-off log management to others". For most applications, long-term retention of log files is not necessary, something transiently written once is enough. Other services too such as those that do network file replication or that do file transfer among workers of a task, also have the problem of being transient but forming part of a distributed application processing. Log aggregation has then too formed part of several architectures, like those for unstructured data processing, monitoring, etc. Other services that are strictly "synchronous" include all services that pay-by-signal, that are "real-time notifications".

Many services also ask that returning rapidly is not that important: they just want guarantee that the service system will collect the request. Notifications from services thereof for "less-than-critical" applications such as batch processing, are also "weak". Note that: "make a backup for less-than-critical services".

3.3. Integration with Other Technologies

Kafka is a popular messaging system that integrates easily with other utilities and technologies. It integrates with distributed processing technologies that are valuable in the Big Data context, such as Hadoop, Apache Spark or Apache Flink, and it also integrates with big data stores such as HDFS, Amazon S3 or Google Cloud Storage. It has specific native connectors to integrate with many different databases, so that one can easily store messages in Kafka or retrieve messages from it. One such tool is Kafka Connect, which brings data in and out of Kafka and is used to stream data into and out of other systems such as relational databases or NoSQL databases, data lakes, or search indexes. There are many connectors already available for the most popular databases and services, ranging from Elasticsearch, HDFS, S3 or Sphinx, to popular databases such as Oracle DB, Amazon Redshift or Snowflake.

Kafka has a flexible mechanism to integrate with stream processors and stream processing engines. It is very easy to develop our own stream processing applications by using consumer and producer APIs within the programming language of our choice. We can use streams of data in Kafka in Apache Spark Streaming programs, which combine micro batching stream processing with batch processing. It facilitates the Spark Streaming infrastructure by integrating the topic offset metadata that is stored in Zookeeper. Spark Streaming heavily relies on the features of Kafka to achieve efficient and fault tolerant processing. If we decide to use a standalone Spark Streaming cluster, we need to make sure that the enterprise policies for the Kafka topic configuration that we are using in Spark Streaming are fulfilled for Spark Streaming to work well with our topics.

4. Apache Flink

Flink is a stream-processing platform, like Spark and Storm. Flink was initially released in 2011 and is now a top-level project under the Apache Software Foundation. Flink started as a batch engine for big data, like MapReduce or Hadoop but then gradually grew into a full-fledged streaming framework. Flink is written in Java and Scala, is open source and free to use, and runs on massive clusters with thousands of nodes. People mostly use Flink for analytics workloads in data pipelines, machine learning, or streaming ETL.

Flink's operational model is primarily micro-batch-based like Spark Streaming or Storm but differs in that it allows low-latency batch processing. Tasks can use much larger memory buffers than the micro-batches to perform LUTs on the incoming streams, making Flink also suitable for low-latency applications like web or social network analytics.

Flink's offerings go beyond those of the other streaming platforms. It is not just a low-latency MapReduce, it is general distributed processing, optimized for the speed of low-latency analytics workloads that need distributed parallel processing. Its main goals are fast economically feasible execution of streaming tasks, ease of use, fault tolerance, and general applicability to a large class of analytics workloads. It is a framework for creating optimized parallel distributed applications with data flows of operators, a combination of data cleansing, transformation, enrichment, and analysis, that are run on the cloud. Once the application flows are defined, Flink configures automatic optimization, scheduling, and logging of the application, fast memory read and writes, and memory and cluster management.

Overall, Flink combines the best of the micro-batch-based frameworks with the best of the high-performance, low-latency stream processors: general applicability, ease of use, and fault tolerance of Map-Reduce, low-latency execution, transparency, and speed of low-latency DBMSs.

4.1. Core Features of Flink

Flink pushes the envelope in terms of stateful, fault-tolerant micro-batching of data streams, with massive parallelization of independent or relatively-independent processing tasks. As such, it has many important features that other systems typically do not consider providing. In this section, we outline the core features that are pivotal for supporting real-time data processing.

Event Time Processing: One of the most important differences of Flink's abstractions from batch and MapReduce systems is explicit support for event time and out-of-order data. Streaming systems deal with continuously flowing streams of data from sources such as user interactions, sensors, or other software systems. These messages are usually time-stamped by the user or the upstream source that generates them. However, when streamed over the network and ingested by the stream-processing system, many messages can come late, and messages can often come in an arbitrary order, as there is no coordination done beforehand. Often these messages are bundled in low-latency batches to reduce the overhead of network calls. As a result, queries that depend on time, such as session queries, windowing operations, or time-based state storage cannot rely on the ingestion time of messages and instead must rely on the event time of messages. Flink stores the event time for each message and provides a robust runtime and query language for working with event time, deadlines, and session queries.

Support for Per-Message Watermarks and Event-Time Session Management: Although event time processing solves many of the challenges of real-time data processing, massive parallelization means that datasets are spread out over many worker nodes. These nodes are often independently processing messages and that can still lead to late messages for time-based queries. Flink adds per-message watermarks to the user-defined session management and timeout rules to allow these rules to be triggered even when messages arrive out of order.

4.2. Flink vs. Other Streaming Frameworks

Apache Flink is often compared to Spark Streaming, the most widely adopted framework for distributed batch and near-real-time data processing. Unlike other streaming frameworks, Spark is not a pure stream processor. Instead, Spark Streaming provides discretized streams (D-Streams), which organize streaming data as a series of temporal mini-batches for micro-batching processing. This approach loses the true real-time guarantee, and any operation requires at least a couple of seconds or more. Moreover, D-Streams do not support the complex event-time features provided by Flink, like windowing, either.

Unlike Flink, Spark is based on micro-batching and is designed to work in mini-batch or near real-time mode. Hence, Spark is not a pure stream processor, and any operation requires at least a couple of seconds or more. Moreover, micro-batching is designed to process data in fixed duration. So, if the usage pattern is logically discrete data with a batch size of 1 (which is a very common usage pattern for stream processing), it will not be efficient with micro-batching. Optimization of Spark Streaming is possible, and the response time can be reduced to sub-second level, but the bottleneck analysis, which is complex in nature will not be trivial. Flink is designed from the ground up as a true stream processor system and written in a way that it supports various usage patterns and scalability. Flink primitives also allow the user to implement complex pattern matching and time-based query processing efficiently. Flink has built-in support for federated stream processing where link types are heterogeneous, and the components are in different scales with different effectiveness and efficiency.

4.3. Real-Time Data Processing with Flink

Apache Flink is a powerful distributed runtime and library for stateful processing of data streams. It was originally developed by an academic group for the realization of the Vienna Map, following the datacentre computing paradigm but providing several novel features for stream-based processing. It is now maintained by a large open-source community, operated by companies and organizations, and has a large user base including initial partners. It supports either batch or stream data – by using the so-called streaming API for a streaming application or the Dataset API for a batch application, builds dataflow graphs, and automatically chooses the optimized version.

Flink is event-time centric, implements the concepts of event-time, watermarking, and event time windows and time-aware join operations with a combination of latency bounds. These features allow the user to specify bounds on the maximum processing and, optionally, the maximum latency of stored results to a specific output sink. Within specifications bounds, Flink will optimize the flow of events through the processing pipelines and limit the processing where the bounds are expected to be violated. An important part of Flink is that it implements mechanisms for efficient data processing that uses shared state with spectacular efficiency with support for indexing and for stateful computations. It provides out-of-the-box libraries for building more complex pipelines and easier application development, such as libraries for streaming machine learning, temporal joins and regular expression matching.

5. Azure Stream Analytics

Microsoft Azure Stream Analytics (ASA) is a real-time analytics service designed to process and analyses streaming data from the Azure cloud and on-premises devices. With an easy and streamlined interface, ASA allows data analysts and developers quickly to run queries on multiple data streams. Built on a highly scalable infrastructure, ASA offers enterprise-ready jobs that can be deployed through the portal interface or programmatically.

Unlike other solutions, which are usually specialized on one aspect of streaming analytics, ASA allows running data transformations, aggregations, and arbitrations. With its native support for dimension tables, built-in geo-spatial functions and ability to learn the dynamics of the data, ASA offers all the building blocks of streaming analytics. Unlike complex, hard to manage big data solutions, ASA automatically scales your analytics based on the amount of incoming data and the complexity of defined transformation queries. If built on Azure SQL Database or Azure SQL Data Warehouse, ASA allows to easily create reporting dashboards in Azure Power BI.

5.1. Key Features of Azure Stream Analytics

Azure Stream Analytics has many capabilities, which make it one of the essential building blocks of the Azure Cloud. Deploying an Azure Stream Analytics job is as simple as defining the input streams and the output streams, identifying the data processing queries, and defining the processing units. There are several aspects to an Azure Stream Analytics job that make it easy to configure and manage. Azure Stream Analytics can process multiple input data types, including IoT hub telemetry data in JSON format, event hub data streams, log data from blob storage, and custom data streams via both Azure Function and REST APIs. These input streams can be easily correlated with other events issued by Azure Cloud Services, Line-of-Business Applications hosted on-premises or SaaS providers.

Key Features of Azure Stream Analytics Azure Stream Analytics is a service built on top of Azure cloud to analyses streaming data from sensors and devices, or other sources, and produce real-time alerts and notifications, control actions as well as provide query results for further processing, listening, or visualization in dashboards, business applications, etc. In this section, we briefly summarize the key features of Azure Stream Analytics.

First, Azure Stream Analytics gives you the capability to create jobs to process streaming data and query results materialized and presented in real time. Such query results from streaming data can be ingested into other data sinks from Azure stream or batch ecosystem, business applications, NoSQL or SQL-type databases including Azure Cosmos DB, Azure Blob storage, Azure Data Lake Storage, Microsoft Dynamics 365, or customer locations including Azure Arc-managed data services in the European Union due to GDPR. Time parameters can be defined in jobs, allowing jobs to run infinitely long, while keeping results from different sources or systems in sync with user-defined code. Jobs can ingest streaming data from one or more sources including Azure Event Hubs, Azure IoT Hub, or Azure Service Bus easily, by providing the source resource name, query access policy to connect, and other parameters in the job.

Azure Stream Analytics scales out query processing along multiple dimensions. Provisioning of Internal and External jobs for processing high-velocity, big data volumes like sensor data is as easy as checking an Enable Azure Stream Analytics Job Scaling checkbox, and as easy as uploading a scaling configuration file. Azure Stream Analytics data ingest mechanism is also robust, ensuring that messages are not lost during ingest due to temporary outages or spikes in data traffic. Azure Stream Analytics is serverless, abstracts away all complexities of managing infrastructure layers underneath, and has simpler pricing based on number of Streaming Units – unit of compute and memory resources required to run jobs – provisioned to run jobs, making it ideal for small businesses and individual developers.

5.2. Deployment Scenarios

Azure Stream Analytics (ASA) is a fully managed, real-time analytics service with enterprise authentication and security, and industry-leading integrations, that offers automatic scaling to handle large volumes of data and with security, monitoring, and operational features built-in. Because of ease of use, any developer or domain expert with knowledge of SQL can quickly build a Stream Analytics job and monitor its progress and view its output. Some of the built-in functions like geo-spatial functions, temporal functions for Temporal Joins and PARTITION BY Session ID, or Machine learning for anomalous detection, helps the users quickly get the job done. ASA integrates well with Azure ecosystem. It has built-in connectors for all common data format and connectors like JSON, Avro, CSV, Parquet, sending and Receiving events to and from Azure Event Hubs, Azure IoT Hub, Azure Blob, ADLS Gen2, Service Bus. All services are managed services, so users don't have to watch out for availability or scalability or scaling optimization.

ASA Jobs can be run locally on developer's machine or on Azure or on Edge devices plus on onsite data droplet for offline scenarios. Based on deployment location and job complexity, users can select job from a different size, with different hosting options or pricing models. ASA provides Batch Processing feature to the developers who have offline/on-demand analytics related needs. A Batch Processing job can be scheduled to run in the requested time window to get the output for the time range. Stream Processing jobs can also be scheduled to simulate batch jobs to combine the result for a particular time or to aggregate data with coarse granularity.

5.3. Integration with Azure Ecosystem

Azure Stream Analytics is a critical part of Microsoft Azure's data analytics ecosystem. Its primary goal is to offer a serverless cloud-based analytics service to ease the queries and analysis of real-time data streams coming from cloud clients such as Azure IoT Hub, Azure Event Hub, or Azure Blob. Traditionally, real-time query support for such streams has been one of the big challenges in the cloud. Because clients send huge amounts of event data expected to arrive in the cloud at petabyte scale, it is difficult for cloud analytics services to support low-latency, ultra-fast ingestion of events, and at the same time, efficiently execute streaming window queries with millisecond response times. This challenge becomes even harder if the analytics engine is part of a serverless infrastructure and is not (fully) under user control.

In Azure Stream Analytics, especially, the focus is on supporting Azure users working with IoT scenarios who face the challenge of analysing large data workloads coming from sensors. Azure Stream Analytics emphasizes easy and fast deployment of streaming queries for easy-to-use, pre-built connectors with managed services, which work as users' data sources or storage results. Azure Stream Analytics serverless service architecture hides the complexity of managing and scaling the job cluster. The service integrates tightly with other Azure services in the cloud, and it seamlessly scales micro-query clusters up and down according to workloads. As a result, users can deploy streaming queries in minutes by using logic connectors, reducing the need to code and deploy traditional complicated data pipelines based on other Azure data analytics services to achieve similar tasks.

6. Event-Driven Architectures

Introduction: Event-driven architecture (EDA) refers to a major software architectural style based on event messaging for communication. Event-driven architecture is more than just the use of events or event handlers in a system; it extends to how those events are used for communication between software components, and how those events trigger changes in the state of the system. EDA embraces and defines relationship patterns and principles for how software components exchange events, and how they derive useful results and side effects from those events over time. It also includes event message formalization.

Principles of Event-Driven Design: An event is a descriptive message about something that has happened, which is published to notify interested parties. Event-driven architecture (EDA) is based on event communication: generating descriptive messages for interested parties, without requiring them to ask for status or current state; and it is the preferred architecture style when conditions are present. Event-driven design begins with the identification of interesting events and their format. Only if the amount of event traffic would be excessively large does it make sense to purposely lower the quality of the event message. Shared events provide a high-level of shared visibility of the behaviour of the system over time, at the expense of bandwidth. Therefore, resource consumption by events must be managed via priority and volume.

Benefits of Event-Driven Architectures: The goal of event communication is to reduce tightly-coupled polling-style interactions between components. The advantages of event-decomposition and event-driven interaction can be achieved by a systems developer via the careful implementation of timer and status check mechanisms in a request-brokered architecture. However, the developer receives no help from the architectural model in using the request-based style to build a decoupled, asynchronous, event-driven system, because synchronous, request-based communication is the idiom that is used for both the architectural interface and the API and protocol used by the components of the system. Furthermore, the event-driven idiom is a much more efficient implementation strategy in a distributed environment than a polling mechanism, because polling requires that the requestor consume network resources in obtaining and interpreting an answer.

6.1. Principles of Event-Driven Design

Event-driven programs differ from procedures in important ways. An event loop or dispatcher waits for events to happen, extracts them from the event queue, and calls the appropriate handler routines. The instance of an event handler that operates for a particular event occurrence is invoked when the event is handled and suspended when the event is dispatched. Events are low-level state change announcements: e.g. “a button was pushed,” or “the window has been covered.” High-level event descriptions can be built on low-level events. For example, a high-level “mouse clicked” event can be built on button-pushed low-level events, one for the left button, one for the right button, and a low-level-time-passed event, which is a kind of low-level timer that is not otherwise captured by the purpose-built timers. The dispatcher for a derived event calls the appropriate handler for that event when it occurs, but not the various handlers for its low-level components.

Events in a conventional event-driven architecture are generally of two kinds: input source events, and events from handlers, streams, or other devices. Some programs using an event-driven architectural style may handle events from a variety of different sources, not necessarily directly related to each other or synchronized in the sense of being conceptually part of the same occurrence sequence. Input events may be received from any of a variety of independent peripheral devices, at any number of times during the lifetime of the program. In addition, device handlers may generate compound events corresponding to a variety of states within the device being handled or to intermediate stages in a data processing operation, communicating information to the program via the event-handling mechanism.

6.2. Benefits of Event-Driven Architectures

An event-driven architecture provides a distributed communication model based on the global concept of "events." The communication protocol is typically straightforward and asynchronous. In the most basic form, a message is sent that notifies other applications that something has happened. This low barrier to integration creates a sense of ease about building event-driven systems and explains the popularity of publish-subscribe infrastructures based on an event-driven architecture. In addition, because of the characteristic that applications use fire-and-forget methods to send messages, the tight coupling of states is broken, which leads to a reduction in the failure of the system.

There is a tendency to build a significant number of infrastructure services or agents to abstract various application concerns. For example, reliability, delivery guarantees, protocols for local area networks or the Internet, peer-to-peer discovery mechanisms, security, and management and monitoring are some issues that agents on top of messaging middleware may abstract. This low barrier to building reusable decomposed components reflects well the reuse factor that is both encouraged and facilitated using messaging infrastructure. In addition, provided an adequate middleware infrastructure, these events can be consumed and discovered anywhere and by any party in real time, stored for future looking, or aggregated and processed in real time by a complex event processing engine. Adding the event definition use of XML with schemas increases the degree of decoupling since semantically meaningful events are traded using defined interfaces, making development easier.

Real-time events truly decouple many components of enterprise software. In addition, the distributed change logging and data synchronization capabilities resulting from using an event-driven architecture facilitate introduction of intelligent decision-making systems. Multiple inference engines or sets of inference engines can automatically take actions based on a problem space when constraints are violated by monitoring these events. EDA also provides a natural way to implement business process workflows or other state machine definitions.

7. Use Cases in Fraud Detection

Fraud detection is one of the most critical areas where real-time databases and stream analytics are used [2-4]. By continuously monitoring user behaviour in real time, financial institutions, social networks, and remote patient monitoring systems can detect fraudulent activity within seconds, blocking transactions and accounts before they cause any further damage. In the financial world, fraud monitoring in credit cards and stock exchanges has become a big business; it uses fraud detection systems that do not stop at merely gathering and analysing data, but also operates with the knowledge and competence of highly skilled human experts. Real-time fraud detection is thus an important area, consisting of both the real-time store of data and the intelligence embedded within it that allows for accurate detection. What is important for a real-time fraud detection system is that it be automated: often, human expertise is limited to tuning the rules used to identify the abnormalities. With the advent of artificial intelligence applied to credit card transaction monitoring, the evolution of fraud detection systems has moved to algorithmic automation, in which the analytics engine adapts to what is normal for a user and what is abnormal, significantly speeding up the speed of implementation.

In fact, cloud-based fraud detection systems for credit card transactions have recently become extremely popular. Leveraging the big data cloud infrastructure, payment card transaction monitoring has become a lucrative business because it guarantees the protection of potentially huge losses. When credit card fraud is detected, the operations that are likely fraudulent are blocked and the customer is contacted. Thanks to real-time, streaming analytics capability, the applications are truly dynamic and react on the fly to changes in the state of the operating environment, adapting to the instantaneous characteristics of the traffic streams.

7.1. Real-Time Monitoring for Fraud

When it comes to fraud detection, one of the most useful models is the one that consists in comparing two time-series data streams. In the banking domain, this model is particularly useful, as banks carry out time-series activity monitoring, looking for sudden changes that differ from the normal profile of a user. As an example, let us consider a client that makes a purchase of $100 at a supermarket in the afternoon and, at night, he buys a plane ticket at the airport in Rio de Janeiro for $1000. In this situation, a bank would notice that the client is not logically making the purchases as expected—that is, that there is no logical explanation for such purchases.

In general terms, the algorithm that banks have profiles user behaviour, estimating activities such as the average value of transactions, their temporal dependence, the normal network for that client, and so on. The model of the temporal series that is implemented collects information from the user and analyses it. For the temporal profile model, the logic for this activity analysis is as follows: the bank has a time series for this client, which talks about his transactional history from the past until today for specific transactions (where, when, how much). A model can be estimated, under the assumption of limited rationality, which explains what the client is expected to do. At this point, we have to verify how well this transactional history currently matches the predicted profile for the specific transactions. Based on variables such as distance, time, and nonconforming transaction value, fraud alerts will be generated when there are differences. The algorithms implemented in such systems are called alerts engines and are responsible for catching fraudulent transactions. The activities that abnormal alerts generate are called investigations. This monitoring reaches real-time levels of milliseconds, which are not observed in other solutions available in the industry.

7.2. Machine Learning in Fraud Detection

Machine Learning has become increasingly popular due to its accuracy, adaptability, scalability, and availability in fraud detection. However, it needs to be improved because machine learning is not a solution for all fraud detection cases. There have been several advances in using Deep Learning and Unsupervised Learning models. Today’s fraud patterns evolve rapidly, and large amounts of transaction data help fraud patterns to evolve. Therefore, the solution goes into two stages: the first one is to use classic Machine Learning for feature engineering, heuristics systems to define features to be used in the second stage. The second stage is to use complex models like Deep Learning or Unsupervised

Learning models. Heuristics models will run in parallel using different parameters and thresholds to compare results.

In order to be successful, fraud prevention measures should be effective and efficient at real time without delaying affected transactions so that they continue to be profitable for the company. Therefore, a multi-segment approach sometimes is needed with multiple models and different times of analysis. In most cases, the best models for dimension model are not the most complex ones. The heuristic first stage is the base to define the outcomes more concentrated or disperse from business point of view. Considering that it is important that the first step model has a good performance because the second step will have to analyse the remaining population after the first step. After the second stage, Deep Learning with neural heaps or unsupervised models, or add stack supervised models, a shadowing or a testing period must be considered for those who will make validation.

8. Use Cases in Internet of Things (IoT)

The Internet of Things (IoT) allows the connection of different kinds of sensors, actuators, and devices to collect, send, and analyses the real-time sensor data. IoT data processing is typically associated with real-time and streaming data processing systems, because the generated data does not only come from different sources but also comes with different velocities, varieties, types, and formats. In addition, the system should start from a machine-to-machine (M2M) communication, which has a minimal human intervention. However, it should be also able to support the human, with higher interactivity, via different interfaces, such as mobile applications and web dashboards. The system should also be scalable to combine, and the components should be used at edge level, for example, Fog Devices for preprocessing, data reduction, and preprocessing; and Cloud systems for storage and advanced analytics, which might involve complex and heavy machine learning functionalities.

This section covers, in general, the IoT data management from two perspectives, the use cases on real-time data processing services, such as Data Stream Management Systems (DSMS) and Complex Event Processing (CEP) systems; and the required functionalities that should be supported. We start by presenting some of the already existing IoT use cases and platforms for IoT Real-Time DBs and Streaming Analytics. After that, we focus on the features that should be supported, namely a processing model that combines hierarchical data models, efficient query processing and indexing techniques, activity recognition, and complex event detection, as well as security.

8.1. Real-Time Data Processing in IoT

The Internet of Things (IoT) is an emerging technology paradigm that represents a sustained investment from industry and government. A wide variety of devices and sensors can be used to measure a great number of variables. These devices can communicate the data generated to centralized services, which can further process the data into knowledge. IoT technology can be applied to a wide number of application domains, such as cities, manufacturing, health, and energy, among others.

The IoT research community has been focusing the efforts mainly on the device side of the architecture, developing more efficient protocols for data generation and transmission, which are still very relevant and important topics. Data processing on centralized services also deserves attention and has the potential to add great value to IoT applications. Like other technology waves, Big Data technologies have appeared to allow managing large scale services. These technologies continue to evolve and can potentially bring many benefits to the IoT centralized services, even though IoT services are typically not only larger in scale in terms of the volume velocity and variety of data generated, but also quite distinct in terms of the specific requirements of real-time processing of the data being collected.

As the IoT systems have specific processing context, novel solutions and techniques must be designed to take advantage of this context. Furthermore, not all IoT applications have real-time data processing as a requirement. Considering that the cost for real-time processing may be very high, it is important to understand which IoT applications can really benefit from real-time data processing. For those applications, what are the best ways to reduce the cost of generating the collected data from the devices/sensors and the cost of processing this data into knowledge on centralized services?

8.2. Challenges in IoT Data Management

What are the main challenges and problems in real-time data management in the special context of IoT? Answering this question is not trivial. Although from one side the IoT environment is very heterogeneous, thus demanding its own data management system in this sense, some approaches to centralized and distributed data centres are also being used. Therefore, there seems to be a confusing mixture of diverse data management approaches in the IoT environment. This subsection addresses this question in a complementary way. First, we give an overview of some reports and surveys that identify the main challenges related to IoT data management in general and real-time processing, in particular. Then, we present our own review of the challenges of IoT data management, focusing on the real-time data management challenge.

In an extensive recent survey, the challenges of IoT data management are organized and presented into five main groups: data acquisition, data storage, data security, data analytics, and data visualization. A recent overview focused specifically on data filtering and analytics; the security challenges relate to data privacy and integrity. The related challenge on enabling end-to-end IoT data security is also apparently about society for information management enabling sustainability. Among the main technical challenges related to IoT are its real-time nature, its large scale, dealing with streaming big data, the need for semantic modelling and interoperability, information curation and quality, mobility issues, and the requirements for an effective cross-disciplinary approach.

Real-time data processing is among the key technology components of IoT systems that should be provided. An efficient real-time data processing infrastructure should be developed, considering both the software applications and the system hardware used to deploy the infrastructure, as well as the integration among them. Key research challenges related specifically to the real-time aspect of data management in the IoT environment include error tolerance, dependability, and uncertainty. These three-related challenges on dependability and uncertainty belong to the sense and control layer of the IoT architecture, while error tolerance relates to the analytic framework layer. More broadly, the consensus is that real-time data stream management systems must be concerned mainly with processing an increasing flow of large data streams of varied types.

9. Comparative Analysis of Technologies

This section compares the technologies presented in this work with a focus on the streaming capabilities of the database technologies. The streaming capabilities of streaming databases and real-time databases are compared, and the chapters point out the issues that must be solved for the developments of those database technologies and tools if they aim to be real-time. The functions that enable a system to process a stream of published messages in real-time and in an efficient way should be compared. Example functions are stream filtering, data caching, ordering filtering, time-based aggregation, event-detection, card-triggering, transcript and query handling, access over-time, push technology. All provided help over the API of a solution should point that the solution is for real-time. Moreover, it is important to expose problematic issues such as continuous query scheduling to implement more sophisticated, capable, and scalable real-time database systems.

One of the often-seen approaches are the stream processing engines which have hence been pioneers and have very rich APIs to perform stream processing operations. However, what rules out these engines is some tuples delivered late because the purpose of these engines is that of best-effort. Other limitations are that Stream Processing Engines today do not focus on the expressive aspect of publications and subscriptions and do not aim to hide the complexity of the declaration of the detection of an event or trigger an action in the pub/sub context through an easy-to-use API dedicated to real-time. Today those engines are only query engines, but they offer different modification abilities compared to the databases. If a database can also allow data modification over time through certain functions over its APIs, the streaming operations offered by the other are better and are more numerous.

9.1. Kafka vs. Flink vs. Azure Stream Analytics

Throughout its journey in the last decade, the stream processing area received many significant contributions, resulting in the emergence of several novel and specialized tools, striking new balances and trade-offs in the design space just mentioned related to streaming analytics. Several products and frameworks have been proposed, which specialized in aspects such as availability, loading-events-on-the-fly, user-friendly deployment, DL support, scalability, fault tolerance, etc. In this section, we'll analyses and compare some popular options and candidates for the various components of a streaming analytics solution, namely messaging brokers, data pipelines.

We restrict our analysis to some well-known products: Kafka, Flink, Azure Stream Analytics, and DynamoDB Stream. The first two can be used together while the correspondence to the three components is not rigid since some products assume the role of more than one. Further below, we'll give an overview of these products as well as their proposed combination and we'll then discuss their relative pros/cons. We can think of products such as Azure Stream Analytics and Kafka Streams as being wrappers on top of the other technologies to provide a more user-friendly high-level API to configure stream processing jobs. Indeed, using such wrappers or libraries/protocols over the core products may induce lower coupling in the solution therefore facilitate deploying other/alternative products.

10. Future Trends in Real-Time Analytics

The past years have seen an accelerated interest towards the development of systems and tools for real-time analytics mainly for two reasons, on the one hand, major technological strides and continued innovations have significantly reduced –in most cases– the costs of the HW/SW stack that is at the heart of Data Management (clouds, fast and non-volatile storage, fast interconnect networks, racks of servers, etc.). On the other hand, the quick and radical changes on the methodologies for carrying out business have seen more and more operations taking place in the digital space; businesses have ranked real-time decision making, recommendation and analytics based on recurrent patterns and micro-batch processing as the analytic capabilities that are of most interest to them, to the point that they would require assistance from third-party vendors to increase their capabilities in these areas.

In this respect, the current direction of commercial tools appears to be moving in the direction of supporting advanced analytics capabilities directly on the transactional databases, enriching operations that were traditionally relegated to the Data Warehousing solution at the end of the process pipeline or were pushed outside the database to post-process the output of streaming transformations into the databases. These advanced capabilities presumably will be able to run either in near real-time or in no more than a few minutes in the data that has just arrived at the transactional storage. Indeed, this is what the business is requesting, an end-to-end real-time pipeline that would allow carrying out advanced analysis in the transactional space without having to necessarily consider the complexity of managing two (or more) different systems.

11. Conclusion

The explosive growth of the amount of data and methods that generate it has led to a need for databases to store it and in parallel to interact with streams of data whose real-time processing has become fundamental to obtain useful knowledge quickly. These two areas, real-time databases and stream analytics, have been independently studied for a long time and the incorporation of the features of one into the other has been traditionally limited. Recent real-time database proposals have included streaming support, while some stream processing engines are increasingly providing database features. Despite the independent origin of these two areas, which focused mostly on storing the data and providing means to query it or in writing programs to process data as they arrive, there are several very interesting ideas that when integrated provide a much more powerful toolkit for applications.

With the incessant evolution of the amount of data and the methods for its production, data generating processes must be better integrated into the systems that manage data, real-time systems must go beyond their traditional boundaries and provide streaming support, while stream processing engines cannot ignore the support for persistent storage of the incoming data if they are to be trusted with providing the quality of information that users need. In this context, we described the seminal ideas of each of the two areas. The path is ready, what it calls to us is to act and integrate databases and stream processing engines and take advantage of their best features so that they can be better at their jobs, helping users of data to obtain more and better knowledge from their data.

References:

[1] Kreps, Jay, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." *Proceedings of the NetDB*. Vol. 11. No. 2011. 2011.

[2] Gupta, Rajeev, Himanshu Gupta, and Mukesh Mohania. "Cloud computing and big data analytics: what is new from databases perspective?." *International conference on big data analytics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

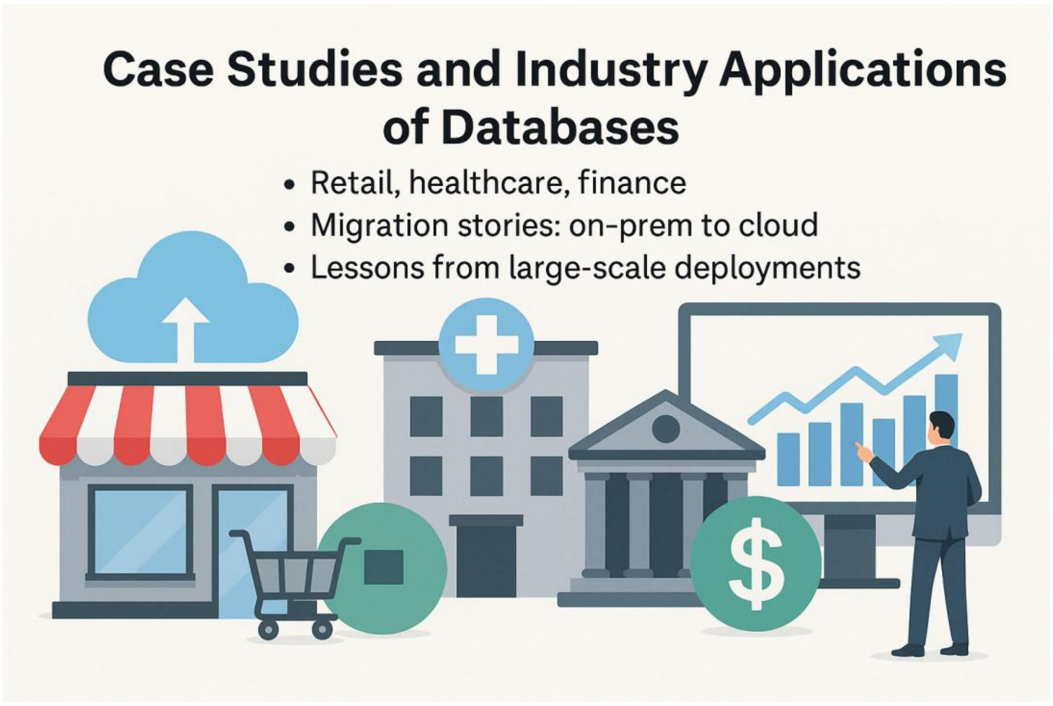
[3] Kolajo, Taiwo, Olawande Daramola, and Ayodele Adebiyi. "Big data stream analysis: a systematic literature review." *Journal of Big Data* 6.1 (2019): 47.

[4] Ranjan, Rajiv. "Streaming big data processing in datacenter clouds." *IEEE cloud computing* 1.01 (2014): 78-83.

Chapter 13: Case Studies and Industry Applications of Databases

1. Introduction to Databases in Various Industries

To introduce the various applications of databases in diverse industries, we will explore various real projects in real-world scenarios that have shaped database technology. The advent of database systems has largely impacted the Information Communication Technology sector and the various industries around the globe have adopted various features of database technology to have carved it in a way to suit their respective business needs. Databases are now being used for almost every function at an organization be it Recruitment, Marketing, Sales, Operations, Finance, etc. Organizations have matured themselves into data-driven organizations, facilitating various pros at their end with the help of Database technology. Having seen the impact at the user end of various software products utilizing database technology, we will explore the case studies of products from the various domains: networks, telecommunication, web, gaming, e-commerce, finance, GIS and CAD, supply chain. We will explore what exactly a database is, the types of databases available, different structures of databases, types of database servers, the major players in the market today having different database technologies, and how their adoption has transformed organizational processes. Databases are now being used for almost every function at an organization be it Recruitment, Marketing, Sales, Operations, Finance, etc. Organizations have matured themselves into data-driven organizations, facilitating various pros at their end with the help of Database technology. Having seen the impact at the user end of various software products utilizing database technology, we will explore the case studies of products from the various domains: networks, telecommunication, web, gaming, e-commerce, finance, GIS and CAD, supply chain. We will explore what exactly a database is, the types of databases available, different structures of databases, types of database servers, the major players in the market today having different database technologies, and how their adoption has transformed organizational processes.


Case Studies and Industry Applications of Databases
• Retail, healthcare, finance
• Migration stories: on-prem to cloud
• Lessons from large-scale deployments

2. Retail Sector Applications

Retailers have a long history of using database applications [1-2]. Several concepts, such as market analysis and forecasting, were created and pioneered by retailers. New concepts, such as 1-to-1 marketing, are being implemented and refined by retailers. The initial database usage focused on inventory management. This is still an important aspect of retail applications. However, the main area of database applications in the retail sector is sales analysis and customer relationship management.

2.1. Inventory Management Systems

Long before computers, retailers had implemented systems for managing inventories. These have normally been break-bulk operations, buying large volumes of goods at low prices. This large volume is due to economies of scale by purchasing large quantities from the manufacturer. More goods are then broken into smaller packages for resale. Each of these packages is sold at a higher price than for wholesale. The difference is the retailer's gross margin, which is the difference between the purchase cost and resale price.

Efficiently managing the inventory is critical to the retailer. If the inventory of a product is too low, the retailer will lose sales, and the customer might switch to a competitor who has the product. But if the inventory is too high, the retailer will incur unnecessary costs and might have to discount the product. Therefore, the retailer has the incentive to always have available the optimum quantity of each product.

Apart from day-to-day transactional activities, inventory management systems focus on the design of models for system development alongside reporting tools that substantiate business decisions based on the performance of inventory. Cycle service level is a demand determinant on a product-by-product basis since some products incur much at stake and hence should be available for most of the time while some products can afford to be out-of-stock, not constituting a significant loss to the business. Stacking products to the hilt always can escalate the carrying cost of inventory to an all-time high and entails a compromise on low prices, thus driving customers out and eroding overall profits. Another crucial metric that helps identify high-selling products is the product life cycle, which estimates the time sequence of sales growth followed by decay for groups of products. Detection of inventory shortages or overstocks is also done using inventory management systems. Although it is impossible to sell all colours and sizes of every product all the time, available-to-promise tracking keeps an eye on upcoming orders and helps in short-term planning of inventories according to peak demand. When real-time monitoring of actual and predicted stock positions is in order, collaborative and efficient inventory systems signal users of restock or depletion needs of major selling retail items.

2.2. Customer Relationship Management

As with almost every area in life, the information technology revolution initiated a movement toward companies trying to use their resources more efficiently. In the last couple of years, we have seen the emergence of the so-called "information economy." To reach this goal, companies have sought a better understanding of their customers. The idea is to identify loyal customers, create a long-term relationship with them, and segment customers according to their importance. This helps the retailer to direct marketing efforts to those groups of customers that are most likely to buy. This results both in increased sales and more efficient allocation of marketing resources.

Customer relationship management (CRM) applications allow enterprises to analyse and manage customer interactions, intending to improve customer relationships. At its core, CRM captures actionable knowledge about the direction, duration and content of company-customer interactions, which are then turned into customer-centric policies. Today CRM systems distil data from multiple company databases to build individual customer profiles and then aggregate results cross-sectionally or longitudinally to synthesize patterns for traditional market segmentation or for predictive modelling to optimize one-to-one customer contact. Rather than build a centralized warehouse and power it with expensive knowledge workers, many enterprises are using SQL and NDMS to capture and manage customer-specific knowledge primarily for their own use.

While all marketing managers support the claim that optimizing customer lifetime value is far more important than optimizing a single transaction, it is not sufficiently appreciated how data and databases can assist in conveying that philosophy down to the sales force on commission. The management infrastructure to convert each individual sale into a small but pertinent piece of corporate knowledge to be fed back to headquarters on a regular basis has yet to be a major investment of many firms. Most companies still prefer to recruit and reward salespeople for their ability to close sales rather than their ability to also record customer feedback. This is unfortunate, as closing sales should not be a different objective from managing customer relationships effectively, and requiring salespeople to do the latter should not be an unreasonable expectation.

2.3. Sales Analytics

Sales Analytics plays a critical role in analysing transactional data of product/service sales during different time durations to derive useful insights that can help drive increasing sales in the future. For example, learning about trends in sales numbers through different days, weeks, months, or years. Such reports tell businesses about the stability and predictability of sales over long periods or seasons. They also help predict future demand for products/services based on past trends. Learning about contributions and trends in sales numbers made by different products is vital to ensure the overall sales numbers are favourable. If one of the products has been consistently declining in sales, then it is time to act for that product. On the other hand, if one of the products has been recently booming in sales, then the business needs to see if it can be further promoted to take advantage of the situation. Preparing reports for comparative analysis of different products can also help in discovering new trends in the combination of products sold. For example, identifying a combination of two products bought from the same customer could encourage cross-selling that could be valuable for future sales. It would also show how different products contribute towards income generation, profitability, and cash generation, lending to decisions about new product development/product modification. This area of analytics also aims to detect customers who make irregular purchases of one or more products over time so that appropriate actions can be taken to encourage such customers to make consistent purchases in the future.

3. Healthcare Sector Applications

The healthcare sector has been impacted by a mix of standardization, regulations, and investment in both developing tools to improve services delivered and the tools to integrate and communicate across the entire healthcare infrastructure. The effect on the healthcare database has been a large and increasing reliance on the storage of semi-structured and unstructured datasets and the integrated use of disparate sources of information to medical care improvements. The adoption of the electronic health record has been described as a substantial milestone, but it is only one of many different applications that utilize databases to help in efforts to improve healthcare outcomes.

3.1. Electronic Health Records

In this section, we will take a detailed look at one of the pioneering works in the database applications for the healthcare industry. The Electronic Health Record (EHR) is perhaps the oldest and one of the most widely used healthcare applications. The EHR has also generated several large databases that are publicly available such as the 30-Day Hospital Readmission Rates, the HCAHPS Survey Data, etc. Therefore, studying the design and construction of EHRs and the EHR databases will help us to understand the various heterogeneous and complex aspects related to the development of health data management databases and applications.

An Electronic Health Record (EHR) contains the medical history and medical data of patient encounters across his/her entire lifetime, as created by multiple providers involved in the patient's care. This contrasts with Electronic Medical Records (EMR) which refer to the digital medical history and medical data of patients which are created and maintained by just a single provider. An EHR usually includes the following types of data: demographics, progress notes, medications, vital signs, past medical history, allergies, radiology reports, and laboratory data. In addition, the EHR integrates data from multiple sources, with the unique symptomatology and clinical expertise from the varied disciplines involved, especially if the patient is being treated for any chronic or infectious disease. The data may include diverse types such as text, structured data, images, and genetics data.

By definition, a typical EHR contains the medical history and medical data of patient encounters across his/her entire lifetime, as created by multiple providers involved in the patient's care. However, traditionally, most of medicine is handled in silos, where various physicians do not communicate, care for the patient in isolation, and the laboratory tests or imaging tests are ordered at different separate places. What happens is that after every symptom-based appointment, the physician does a quick information treatment, ordering not just a cure but also expensive tests. The data are siloed and isolated in different systems created by different physicians without communication.

3.2. Patient Management Systems

The significance, usefulness, versatility, and increasing acceptance of databases and data management systems for various industries, including communications, banking and finance, and insurance, have received ample coverage in the preceding paragraphs. In this section of Chapter 3, however, we start discussing three particularly popular areas in the healthcare sector for applying these tools and technologies: electronic health records, patient management systems, and health outcome support and analytics.

In this section, we introduce and briefly discuss the idea of patient management systems, following which we discuss the more widely adopted idea of electronic health records, and the not-to-be under-considered issue of how to use records and databases to improve health outcome support and analytics. A patient management system is a database system containing records of all interactions a patient has with healthcare providers, such as admissions, surgery, examinations, laboratory test requests and results, medications, and discharge. These systems are particularly appealing to a hospital or a group of hospitals because they enable the linking of a patient's relatives across generations. They also frequently build a relationship between patients and their physicians.

There are many reasonably good commercial solutions available that take care of the functions listed above. Such payroll directory solutions frequently run into two problems: firstly, their linking capabilities are frequently poor – they end up creating very poor family trees; secondly, because they are point solutions, they create islands of information that are not easily shared or used by other health care providers. We had proposed the idea of a centralized patient management system for a region, using state hospitals as data source, for the specific purpose of linking the region’s population and for use by state authorized health care providers only, mainly to improve the quality of its services.

3.3. Data Analytics for Health Outcomes

The quest to leverage data for improving patient outcomes has been the holy grail for healthcare applications. Data scientists rarely claim originality with their algorithms. They use the works of others to build models that predict and train regardless of the domain. But to few have gone deeper into healthcare domain-specific challenges to deliver breakthrough successes. The use of data is maturing from descriptive to prescriptive because of the demand from payers and providers for risk-taking models. Accordingly, the healthcare sector is embracing analytics to address the shift from volume to value.

Healthcare processes are often complex; the relationships can be intricate, and the data may be convoluted. From vaccine and drug development to personalized and predictive medicine, to telemedicine and remote monitoring, the healthcare landscape is constantly evolving. Advanced analytics and big data can play a significant role in several aspects of health. While in many professions employees work best when left to their own devices, in the healthcare profession, something more than “guidelines” is needed. Yet today, the high variability in clinician performance appears to point to a lack of sufficient data for the understanding and promotion of provider best practices. No two patients are alike, and the likelihood of having cognitive errors when assessments are based on subjective judgment are considerable. Integrating large volumes of high-quality objective patient data with decision tools can help bridge this gap, improving outcomes for patients across large populations by developing risk profiles that trigger guided responses for clinicians.

4. Finance Sector Applications

The finance sector was one of the main early database customers and finance companies are still heavy database customers. Corporate and commercial banks use databases for customer management, transaction processing, regulatory compliance, commercial lending systems, retail banking systems, real estate processing, trade finance operations, asset liability management, and credit processing. Investment banks use databases for supporting fixed income and other trading systems, finance planning, equity syndicate processing, valuation, rating, and back-office operations. Insurance companies use databases for customer relationship management, underwriting, policy administration and claims processing, anti-money laundering, life insurance work, risk management, fraud detection, re-insurance work, and research analysis. Other segments of the finance sector also use databases, including hedge funds, mutual funds, pension funds, and private equity firms. The applications are diverse and large-scale, from the database size perspective.

Corporate and investment banking are two segments of the broader finance sector, and their respective applications vary greatly. Commercial banks have a broad range of high transaction volume, low value processes, such as retail banking operations which support branch banking, corporate deposits, consumer lending, correspondent banking, and treasury management. Commercial banks also have low volume, high value process, such as corporate lending which supports appraisal, documentation, approval, and booking. The retail banking operations support data customers, both individual and commercial, and keep track of the various transactions. These transactions are deposited in data centres using operational databases. These data are then used for financial analysis to prepare periodic reports.

4.1. Risk Management Systems

Risk management is a complex practice that is at the core of banking activities, encompassing the entire decision model, which is the primary function of data management in banking. How to measure risk, how to assess each client's ability to cover a certain degree of risk, how to quantify the capital required to cover such a risk, how to hedge risk, and in the end, how to set the price for risk? With the current technological capabilities, the answers to these questions are firmly based on the ability to access relevant data in a timely manner. Risk management is one of the primary functions of databases in banks, and financial institutions usually maintain several risk management applications. Risk management activity aims to identify the business risk profile of the bank and is usually divided into two main sub-applications, which relate to the Measurement and Quantum of Risk and the Hedging of Risk.

The first sub-application covers procedures that relate to the estimation of the level of risk exposure of the bank. These procedures provide a sound basis for daily trading decisions and for capital allocation decisions. The goal of the risk quantification forms a systematic framework for determining the risk factor sensitivities of each business unit in the bank based on a return model. In addition, provisioning models continuously monitor the estimation of the probability distribution of market risks as well as liquidity risks. The capital allocation model is based on optimizing the risk-return profile of all business units and is used to determine the capital allocation for specific lines of business based on their sensitivities to the different risk factors of the bank.

4.2. Fraud Detection Algorithms

The digitalization of services is accompanied by greater risks of frauds and abuses, especially in the finance sector [1-3]. People around the world started using technology-driven services, trusting that web services that guarantee instant help will act as heavenly angels, but which are in fact promoted by brutal money-making machines. In fact, compared to only ten years ago, the number of services available on the internet that allow immediate satisfaction of needs has increased significantly for retirees who are more afraid of being part of a scam, and who are no longer behind the steering wheel of life. From my point of view, this makes them less capable of understanding the evil of technology.

Criminals and Scammers now hire the best computer engineers to create fraud-propagating algorithms for them. The internet allows criminals to operate internationally, while legislation offers little protection. For these reasons, banks, then insurers, and then the entire finance sector began to invest in fraud detection systems, whose purpose was to support police intervention in solving the largest possible number of crimes. These algorithms start from the analysis of the behaviour in historical data of users who have been reported in the past for fraud and create a discriminant function based on user behaviour in order to classify suspected people, estimate how probable their fraud is, and therefore how sure the company should be to notify law enforcement, or the customer involved, who is unaware of the potential fraud. However, not all financial and insurance companies are interested in these data and have them in their databases or even offer companies incentives to implement this type of system. In addition, banks for example have recently focused on credit card transaction flow analysis, as calculations have shown that the economic benefits resulting from customer protection are greater than the costs of fines associated with fraud claims.

4.3. Customer Data Management

In the world of business, it is widely held that "the customer is king," and as such, organizations go to great lengths to build customer loyalty, support product branding, and invest significantly in customer advertising. Cumulatively, in an organization, these advertising activities add up to enormous expenses. Towards this end, organizations also spend considerable effort on collecting and updating data on customers and their preferences. The way in which organizations utilize and leverage this customer data can play a critical role in the success or failure of their strategy. An essential component of managing customer data is the creation and use of a customer database. Customer databases are designed and optimized to store large volumes of relatively small-sized tuples in a format that facilitates rapid and frequent updates targeted to a subset of the tuples. Furthermore, the sizes of these databases could range into hundreds of gigabytes. The databases would also be characterized by large volumes of transaction processing applications that could demand significant throughput. However, the underlying data access patterns would be very different from those typically present in the database. Data Warehousing customer database provides a single global view of the customer and so becomes the basis for generating reliable insight.

Efficient creation, maintenance, and management of customer databases are generally critical success factors for corporations since the customer database becomes the single reliable source of information on customers. Over the years, several corporations have implemented solutions that use commercial or homegrown customer databases and validated the steps suffer low performance. They used large consumer databases in a generally acknowledged loss of quality among, especially about address accuracy. Most customer databases remain external databases that contain less than 8% of the physical addresses and of the people in the United States.

5. Migration Stories: On-Premises to Cloud

5.1. Challenges Faced During Migration

As the demand for cloud databases continues to grow and migration away from on-premises databases accelerates, organizations about to undertake migration frequently seek the experience of those that have gone before them. They look for the best practices, lessons learned, and other shared experience and insights that may aid their own migrations. In this chapter, we present case studies of migration—those who have done it and what they learned. The case studies offer a variety of cloud use cases, including distributed operational databases, Lakehouse analytics, edge and hybrid databases, and NoSQL and XML use cases. The enterprises span various sectors, including financial services, retail, entertainment, social media, publishing, and cloud-native companies. This chapter also covers the migration tools and strategies used, factors like costs driving the migration, the impact of local edge computing and hybrid computing, and industry and enterprise considerations that factor in.

Some aspects of migrating on-premises databases are straightforward. The cloud guarantees immense amounts of elastic compute and storage, so scaling issues become less critical since you can add resources back to near their on-premises peak when needed and drain them away afterwards to minimize costs. The round-the-clock cloud database services, including most of the data engineering and operational tasks you had to do yourself for your on-premises database, enable you to offload those tasks and free yourself by deploying cloud databases in a self-service manner that allows a self-service culture to spring up, shortening time-to-value, increasing agility, increasing innovation, and simplifying the provisioning process.

Migrating data from a corporate legacy environment to a cloud-hosted Infrastructure as a Service platform can result in the overall improvement of the Information Technology grid since doing so eliminates the need to maintain an in-house data centre. However, several challenges can be presented relating to the migration of legacy database structure and data belonging to enterprise applications. This chapter explores such challenges with particular emphasis on the impact on enterprise applications at a major conglomerate on its quest to migrate enterprise applications from an in-house data centre to a cloud-based IaaS.

With IaaS, tenants do not have control of the underlying physical structure or basic virtualization infrastructure. Relying heavily on an enterprise application vendor to implement all aspects of the migration can put the integrity of what is expected to be a proven reliable solution at risk. Lack of control over the physical layer can affect interface availability with other Internet-based external applications that require either interface access to the IaaS physical layer or interface programs that have been written using obsolete and deprecated program languages. Any of a variety of issues can occur during one of the many migrations phases, resulting in a somewhat paralytic situation relative to system progress. Even though nothing will be done with the actual migration until the issues have been addressed; no other aspects relative to the migrated applications can be either started or completed.

In addition to the issues having the potential to lie within the control of the enterprise application vendor migrating the application to IaaS, there are possible issues to be confronted and overcome that lie within the control of the organization migrating. Either interface integration needed between the enterprise application being migrated and external applications utilizing those interfaces or a lack thereof can affect the overall success of the actual data migration.

To be summarized, IaaS can provide substantial cost savings to an organization if the organization can successfully migrate enterprise applications from their current on-premises Data Centre and rely on a third-party provider to maintain the IaaS Data Centre and its associated infrastructure at an affordable cost. However, migrating existing enterprise applications from an on-premises Data Centre to an IaaS relies on the existing enterprise application vendors providing IaaS Data Centre migration services and application support. During the migration and associated testing phases, the organization is usually crippled as far as being able to address any issues that arrive from the enterprise application vendor for the duration of the migration and testing.

5.2. Success Stories and Best Practices

For organizations that need to deploy new systems quickly, cloud databases provide comprehensive, secure storage in a matter of minutes. Many organizations are running production systems on public clouds or are about to migrate to them, and they are reporting successes in enabling use cases ranging from cloud-native applications to information sharing. Organizations that migrate to a cloud database typically report a successful project, trust the process, report migration ease, and would recommend a cloud database service to a friend. Take the case of a global hotel chain whose franchisees deploy a cloud customer service, increasing revenue while staying within budget. A large financial services company rehosted its investment data warehouse on a cloud database for compliance and security, avoiding a regional bank breach incident while also enabling better reporting.

One organization built its IT strategy on public clouds to achieve speed and scale to accommodate its rapidly growing customer base. With the presence of new regulations in the digital banking space, this company relied on modern cloud-native architecture, adopting a microservices-based strategy, using agile methodologies to ensure rapid deployment of applications for business use. With cloud services, the business had increasing access to databases for scalability, while keeping the cost to an optimum.

Cloud databases help organizations scale easily and offer a myriad of functionality because of public cloud vendors building increasingly powerful and rich ecosystems around their databases. Organizations are scaling applications massively or transforming businesses by gaining access to databases at breadth at pay-as-you-go costs. Organizations are being tempted into heterogeneous database supporting applications because of the ability to easily set up, build, and operate many databases. Organizations are experiencing this elasticity at newfound price points. Optimizing for simplicity is inducing complexity in enterprises which organize their business logic into silos driving faster time to market and adoption of database services for new applications.

5.3. Cost-Benefit Analysis of Migration

Cost estimation of technical migration is perhaps one of the most challenging tasks to perform. It is important to gather numbers from both large on-premises deployments and from cloud deployment to build a reasonable and relevant cost picture because the exposed costs of the cloud model are very different from that of on-premises solutions—if a simple software cost comparison could be enough for a local deployment, in the case of the cloud, it is also necessary to consider performance and deployment characteristics. One of the reported complex cost estimations is that of the data processing system, as the estimation varies according to consumer model being used, number of scales outs and corresponding number of cores used during those scale outs, amount of data ingress and egress, and amount of data being processed, especially when taking into account transient costs associated with over-provisioned hours and machine types not being fully utilized.

A major gain reported in the migration cases is linked to the reduction in time-to-market thanks to the near-zero cost of creating test and staging environments, as well as production environments that could be scaled up quickly just for the processing of the user demand such as the holidays buy cycles. In the case of large companies, these gains could far outweigh the advantages associated with the unique purchasing capability and higher probability of receiving special treatment that big corporations. Other gains that have an important impact on total expense are associated with avoiding upfront capital costs for deploying new infrastructures—companies using the cloud business model no longer must support months or even yearly delays in scaling the testing and staging environments to reach and adapt to business demands for new software releases.

6. Lessons from Large-Scale Deployments

Consider the scheduled future shipments of over 7 billion terra- and petabytes of files annually within and increasingly between companies in areas as far ranging as finance, film, healthcare, e-commerce and telecommunications. These shipments, particularly of sensitive or regulated materials, are increasing. Furthermore, compliance with the relevant regulations governing these materials is driving enterprise security requirements for and expenditures on their databases used to sustain these missions. Finally, the volume and variety of these types of data are likewise growing to the point of overwhelming conventional enterprise database systems. These trends are converging to make complex yet highly reliable distributed databases cheap enough to implement to make every enterprise in the economy address these same issues.

The enterprise data storage network, together with its implications for enterprise data governance, provides an umbrella for the lessons learned by designers and implementers of large-scale distributed databases. Deployments such as that of the industry-specific distributed database operating at millions of transactions per second, as well as others mentioned, have addressed a subset of the more mundane storage management issues facing enterprise data governance and compliance. By analogy, enterprise data governance is to MRM as transaction processing is to OLTP. All enterprise transactions are subject to MRM principles and guidelines that accord international, regional, and national laws and government regulations as well as corporate policies defining what data can be stored where. The enablers of MRM are policy-driven, centrally controlled enterprise data storage networks that implement one or more cost-sensitive variants of push-pull download-upload and store-and-forward.

6.1. Scalability Considerations

Real-world deployments of database systems at scale are uncommon; focused case studies, therefore, provide the most detailed accounts of industry needs and design decisions made to satisfy them. These case studies describe operational requirements ranging from data commons to real-time serving to long-term archival, and varying underlying technologies including SQL, NoSQL, and hybrid systems. The limitations and enhancements of existing systems are important to share, both in the hopes of raising the level of future designs and for allowing designers to draw on the lessons of others when building for new classes of workloads. Word storage volume is an important scaling consideration, but not the only one. Data changes and turnover are critical, as are data diversity and distribution, query diversity and frequency, and data accessibility and privacy. A system proposed to meet these demands does not need to meet them all simultaneously; underpinnings such as hierarchical pause, archival tape storage, and intelligent data placement allow hybrid filesystem-database systems to handling active and inactive data together. Efficiently merging diverse workloads—real-time query and updates, bulk query, archival with periodic or no access since being written—placed upon these systems is possible, but additional research is needed. Databases for virtual environments, federated databases for disparate enterprise divisions, or middleware for composite service layer access gatherings of dispersed user work should also be addressed.

6.2. Data Governance and Compliance

In our discussions with enterprise customers from various sectors, we have perceived the importance of data governance, data ownership and compliance. Large-scale organizations frequently house diverse teams and data departments that hastily deploy data systems, with little consideration on how the new additions can alter the existing landscape. Without centralized control, data policies that determine access controls, data retention timelines, classification, and regulation compliance checklists can quickly obsolete. In the backdrop of an increasingly privacy sensitive era, the inability to extend policies becomes a risk. Several customers have overgrown their data policies and found themselves in threat of compliance violation, at times even leading to lawsuits.

Usually, data policy checks are done on tables via manual intakes, neglecting raw data and raw blobs. As only a small fraction of PII is stored in otherwise valuable metadata, which only offers tags this process must cover, every ingestion of PII data into a system must adhere to data governance principles, lest the dependencies between systems and raw objects be established and kept in sync. Instead of only governing the metadata, we believe customers need assistance on extending the data management, access control, and PII-related data policy checks down to the data residing in data lakes and raw storage. Processing objects in the storage must be planned in a manner like that of streaming sources, given the unpredictable costs that a database call can incur when these files are operated over and over again.

6.3. Performance Optimization Techniques

The apparent ease of use for casual users, as desirable as it is, can generate a challenge for system deployment. Performance optimization for large clusters is difficult to achieve and even more difficult to automate. Document parsing and natural language preprocessing must be done for the data to be usable. Both take a significant amount of time when done for all said data and worse, they are not parallelizable since they target small chunks of text. Moreover, using analysed data for user queries also leads to problems at high data volumes. Dirty data will have to be filtered out and it should also be noted that underlying data might change while users are working. Finally, the models used to analyse the data might not be optimal for the specific data addition and while better models exist, they may not be available yet. One solution to these data issues is the application of more DB inspired tools, such as monitoring, quality control rules, and logging at data import time. Improving user experience and performance on large clusters either by implementing as much preprocessing and formatting as possible or designing easy user interfaces for tools with masses of configurable parameters.

We want to widen user adoption while also preserving rich user interaction. The goal is to make interaction at ingest time as natural as possible while allowing for complex queries that can make optimal use of the index features at query time. For the query part, the use of smarter tools for both automatic analysis and support are also required for mass adoption for all these applications. It might be useful to learn from architecture, since they are at least focused on the same problem: making it possible to query large and unstructured data at high speeds while relying on user feedback to guide the process.

7. Future Trends in Database Applications

The studies presented in this volume span topics related to data management applications and software tools deployed in a variety of application domains. Applications in health care and life sciences use databases to track patient records, clinical trials, and large-scale genomic data. Criminal justice applications leverage databases to detect fraud and model criminal networks. Smart cities integrate heterogeneous data streams from social media, sensors, and traffic systems for urban planning. Enterprise applications in service and retail use databases to analyses customer transactions, interactions, and sentiment. Data warehousing, OLAP, and data mining still serve core enterprise applications.

The recent booming interest in big data and the rapid growth in the scale and complexity of related technology and applications is a major trend in data management. Other emerging trends in database applications are also noteworthy. One of them is the rapidly evolving landscape of mobile applications. Widely popularized by the success of smart phones and tablets, mobile apps have opened an entirely new API and user interface model that present unique data management challenges and opportunities as more detailed and structured information get captured, exchanged, and shared by enthusiastic users. Cloud computing is another major trend that has accelerated the adoption of database as a service, pushing the frontiers of traditional database management technology. Service availability, reliability, and performance are critical elements that shape future cloud DBaaS usage models. Owing to their special operating and service delivery models, compared to local deployment of database systems in cloud-based setups, cloud services require a new approach to data management that focuses on the essential aspects of the cloud paradigm.

8. Conclusion

We have provided a general overview of the challenges of database management and how such challenges are met in industry. We have then explored relevant case studies and industry applications that explain how various companies are using databases to meet their industry-specific challenges in both their core businesses and their technological areas. We hope our work serves as a useful research resource in the domain of case studies dealing with database industry applications.

While there is a vast collection of academic research papers and books focusing on database technology, there are significantly fewer efforts that study the industrial side of databases. We believe the reason for this lack of information stems from the sensitive nature of many industrial applications and corporate strategies. This comes as no surprise, as study results and usages may help competitors in each specific industry or technological area. However, companies continuously collaborate with universities in a range of different projects, and work-in-progress presentations and documentation from such collaborative research initiatives would be one way of lessening this gap in academic knowledge. Additionally, education could also benefit from more of such publications – while students may learn the theoretical side of databases in classrooms, they generally do not get to see how such theories translate into practice. Students thus miss out on understanding the capabilities of database technology, the impact of technological decisions on design and performance, as well as the processes of installing, maintaining, and troubleshooting databases. Case studies on industry applications of databases offer insight into those lessons.

References:

[1] Corrao, Giovanni, and Giuseppe Mancia. "Generating evidence from computerized healthcare utilization databases." *Hypertension* 65.3 (2015): 490-498.

[2] van den Braak, Susan, Sunil Choenni, and Sicco Verwer. "Combining and analyzing judicial databases." *Discrimination and Privacy in the Information Society: Data Mining and Profiling in Large Databases*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. 191-206.

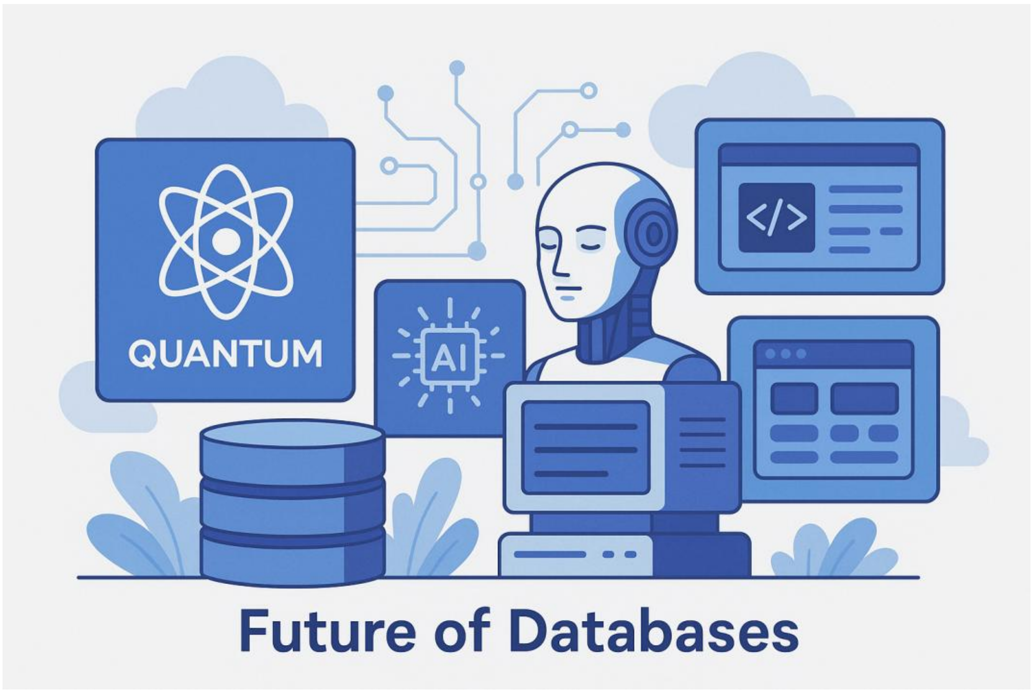
[3] Raman, Ananth. "Retail-data quality: Evidence, causes, costs, and fixes." *Technology in Society* 22.1 (2000): 97-109.

Chapter 14: Future of Databases

1. Introduction to Future Database Technologies

Work platforms (WPs)—software and services that enable organizations to design, create, and deploy digital work—have rapidly evolved from standalone applications to integrated yet modular ecosystems. They serve the full and diverse range of digital work needs for organizations and their external stakeholders, and they integrate data at a scale and richness previously unattainable. WPs harness a unique combination of powerful factors—cloud computing, AI, mature development communities, open development environments, and standards-compliant integration. Together, these factors have radically transformed digital work and accelerated the pace of marketplace innovation. Meanwhile, relational databases that form the backbone of WPs are stagnating. Recent improvements in RDBMS—pinning performance hopes on Moore's law, offloading work to massive in-memory caches, sharding tables, and multi-threading query execution—are beginning to crest. Industry experts recognize the need for new database designs to support the work of the future, and they're reaching back beyond the established boundaries of current databases to seek innovation opportunities.

Disrupting a mature and scale-hardened industry like database systems is no small endeavour. The disruptive innovation requires carefully matching the fault lines of current database technology with the increasingly diverse needs of application developers and business subjects. It requires minimizing the costs and risks of migrating to a new infrastructure, alleviating the DBA burden that inhibits experimentation with parallel and distributed schemes, and proactively managing the stake of current database vendors. Yet there is no shortage of compelling need. We increasingly inhabit a future where applications composite disparate operations, executed within separate environments and utilizing diverse data representations, semantic models, query workloads, processing pipelines, storage layers, optimization criteria, execution patterns—an ecosystem teeming with innovation yet vulnerable to technology stagnation. In this essay we explore in detail some such emerging technologies and what the prospects are for their future adoption and impact.



2. Quantum Databases

2.1. Overview of Quantum Computing

Quantum computers are special devices that can solve certain problems much faster than traditional computers. This happens because of a property called superposition. One qubit in a state of superposition can represent 0 and 1 at the same time. A register composed of n qubits in superposition can represent up to 2^n numbers simultaneously. Quantum computing leverages both superposition and another property, called entanglement, to perform useful computation. In recent years, there have been several breakthroughs in the near-term application of quantum computers. Companies already provide cloud quantum computing service that allows everyone to use their devices.

Quantum computers can be programmed using quantum algorithms, which are roughly categorized in two classes: algorithms that can be classified as an example of a new computation model, and algorithms made from the composition of simple quantum primitives. Several primitives have been proposed, but we can roughly categorize them in three classes: Oracle, Grover Search, and Quantum Fourier Transform.

Quantum computers, exploiting quantum mechanics principles such as superposition, entanglement, and interference, efficiently solve problems whose classical counterparts are considered intractable. Critical breakthroughs in cryptography, optimization, artificial intelligence, and machine learning have spurred deep interest in quantum computing. Building a large-scale fault-tolerant quantum computer is, however, an open challenge. At the current frontier of quantum computing, there exist noisy intermediate-scale quantum computers capable of executing quantum circuits with a few dozen qubits. Noise in these computers severely limits their application to quantum algorithms prone to noise like the variational quantum eigen solver and QAOA. Other promising applications for these computers assume quantum advantage over classical computers. Networking and using multiple computers are a way to build larger systems and utilities like error mitigation that increase the depth of quantum computations can be used to further expand the capability of these computers.

Quantum datasets, databases, and other fundamental data structures are first-class citizens of quantum algorithms. These quantum data structures represent a quantum analogy of their classical counterparts with phases associated with quantum likelihood amplitudes governing the outcome of quantum measurement operations. Many of the common operations on quantum datasets arise in quantum algorithms that either run in a trained quantum computer or map onto a learned quantum circuit with fixed parameterized angles. Implementing training and learnability procedures for quantum datasets is an exciting area of current research within quantum machine learning. Other algorithms-related areas of active research for quantum datasets include what it means for quantum datasets to be efficiently classically learnable and how quantum datasets arise in quantum algorithms for learning classical datasets. These quantum datasets form a crucial building block of quantum algorithms and require precise mathematical definitions. Concepts for quantum datasets in quantum algorithms such as capability of quantum measurements, translation invariance, and joint encoding need a similar treatment for quantum datasets.

2.2. Architecture of Quantum Databases

Quantum Database Management System (QDBMS) demonstrate several apparatuses in the system's design to extract the benefits of quantum parallelism, where processing many inputs at once accelerates operations over conventional databases by an apparent factor of the input size. The Quantum Database Design (QDD) framework constructs QDBMS as a middleware layer, orchestrating between a conventional Object Management System (OMS) and Quantum Processing Unit (QPU) for optimized applications of quantum computing in the development and daily operations of conventional and quantum databases. QDBMS help applications to optimize invocation of a quantum processor integrated into their business logic by selecting correct parameters, such as moment, frequency, and purpose of QPU invocation. Furthermore, QDBMS optimize the conventional stage of database processing to minimize the data exchange with the QPU, boosting the overall performance and avoiding bottlenecks in inter-device retrieval.

Quantum CRUD provide general use QDBMS functionalities to facilitate manipulation of the quantum databases. More advanced QDBMS level functionalities explore the use of quantum techniques for objective acceleration of specific quantum CRUDAPI functions, such as QS for massive speeding-up of querying colossal quantum databases; QU for maximum push from quantum caching by updating small parts of quantum databases by quantum processors; AQC for renting small sides of relatively large quantum workloads by quantum computers – quantum parallelism does not come for free for a single input-side quantum workload. Specific QDBMS-level functionalities could further explore novel quantum techniques, like accelerated entanglement on dynamically built Bell State Trees for speedy-up of retrieval from highly entangled physical quantum databases.

2.3. Advantages of Quantum Databases

A quantum database has several advantages over a classical database. One advantage is the rapid search capability, which provides an exponential speedup for unstructured database searches and a quadratic speedup for some structured database search queries. A quantum database supports new types of search queries not supported by classical databases. For example, relational databases return records that match search criteria specified by queries. A quantum database allows the computation of a weight function during the retrieval process to return a selected weight function value or join matching records, usually not performed by database systems. Another advantage is massive parallelism in data manipulation. This represents an NP-level speedup over classical database technologies that fit many NP problems into a database framework.

As another example, massive parallelism lets a quantum database choose the values for an aggregate function such as sum and delegate the basic operations to the quantum bit in the quantum states. Furthermore, although the current query optimization strategies focus on relational quantum databases, there can be innovative quantum functionalities that can lead to novel query optimization statistics leading to various optimization strategies for quantum databases. In the case of real-life databases that usually represent a non-factual representation of reality, there can be a lack of a pattern for answering queries. But unlike classical databases, a quantum database can be modelled to carry out even off-balanced weight or random-type functions using quantum states. Such quantum states can facilitate the answers to quantum database queries, enabling novel constructions of specific structured quantum databases with optimizations based on specific quantum functionalities.

2.4. Challenges and Limitations

Despite all the numerous benefits that quantum databases can theoretically bring, the reality in developing these new structures presents many difficulties. The practical implementation of quantum computing systems, quantum algorithms, and quantum systems software is in its infancy stage, sparked recently by major investments from the global top players in technology infrastructure. So far, these investments have not resulted in any practical quantum advantages over classical systems, nor the ever-delayed promise of quantum supremacy. The quantum computing hardware stacks are limited in several aspects, suffering from noise, low fidelity, and decoherence. The cost of running quantum jobs is also extremely high since maintaining quantum systems is orders of magnitude more expensive than traditional semiconductor-based systems and is accessible only to a privileged group of users.

This high cost prevents most of the practical development of original and robust applications. Most interested parties are simply renting quantum devices to test and run their algorithms via cloud services provided by the hardware vendors. This might change when the quantum stack matures, with lower noise components, more qubits, more powerful quantum CPUs, when the quantum system becomes available for all and accessible in an equal way, and the software stacks are optimized with quantum libraries, compilers, programming languages, and finally application building blocks. But until that happens, all work being done in academia and industry is mostly at the experimental level. Therefore, it seems acceptable to assume that practical use of quantum databases powered by quantum storage and implemented on real quantum devices with practical quantum speedups is still very far away, at least decades of work in the quantum trenches is still needed.

2.5. Use Cases and Applications

There are multiple applications of quantum databases, often inspired by classical database or computational queries. One such query can be the Unstructured Search query that uses Grover's algorithm as an oracle. It has been proved that Grover's algorithm can be realized using quantum databases. This application can be useful for searching a DNA string or a password. Another popular application is the Coin Flipping query, which allows a user to run large-scale coin-flipping tests that traditionally cannot be run without trusted parties. The other applications of quantum databases include the Unstructured Database Search, the Histogram Query, k-Similarity Join, Spatial Queries, etc.

One important feature of quantum databases is that they are expected to support multiple users. There also exist some applications that intentionally achieve such "multi-tenancy" scenarios. The applications of quantum databases that realize such multi-tenancy are the Quantum Distributed Database and the Quantum Customer Relationship Management. The Quantum Distributed Database achieves scalability by segmenting the database into separate distributed regions and storing them in quantum databases of different nodes, which can provide fast parallel query processing. The Quantum Customer Relationship Management provides practical services and capabilities for the customer in a quasi-quantum world and pushes quantum computing users from regular users to special users. Other quantum database prototype models are summarized. These quantum CRUD database models also provide clear encoding techniques for data users and developers to implement quantum database CRUD functions. This recall function allows other quantum database models to quickly recall functions for easy collaboration.

3. AI-Native Databases

3.1. Defining AI-Native Databases

AI-native is a term proposed to describe new systems, applications, and capabilities built on the growing needs of machine learning and AI. AI-native systems are born in a world that has fundamentally changed by the new demands and capabilities provided by machine learning, whether improving existing systems or enabling entirely new levels of capabilities and performance. For systems supporting the demand of a growing number of AI and machine learning workloads, the challenges are often more than just scale. Feature engineering is a complex domain-specific task that requires domain knowledge. Furthermore, it could be challenging to find the right dataset and manage the machine learning lifecycle. Many organizations could create huge amounts of unstructured data but struggle to manage and analyses them. We characterize AI-native databases as databases designed to support AI and machine learning, with first-class capabilities including machine learning integration, data management and processing, and model management and optimization.

AI-native features and capabilities are introduced at some levels in a variety of modern systems deployed today. However, the integration is often not deep enough or holistic enough what we describe as a true AI-native database designed specifically for the needs of AI and machine learning. Often, these systems need to be complemented or intelligently chained with other modern systems. Furthermore, the idea of ephemeral and special-purpose databases is not new. Many machine learning projects go through a series of iterations experimenting with many different model configurations and data transformations. Machine learning engineers would create and delete datasets of feature transformations of the underlying data rapidly. AI-native databases evolve ephemeral datasets and ecosystems beyond ad-hoc. By providing accessibility, these capabilities also give citizens or business users the ability to execute complex machine learning processes collaboratively and at scale.

3.2. Machine Learning Integration

Machine learning (ML) has permeated many sectors and has been incorporated into many systems, including Cloud, Literature, or the Web. Some of these systems involve databases, where ML has been used to assist with various internal mechanisms such as sample automated suggestions or reinforce the outcomes of some forms of query processing or enhancement through additional ML models. Databases have also assisted ML algorithms, especially deep learning, by helping to efficiently retrieve data for these algorithms to perform inference or training, as well as other management functions such as version control for datasets.

However, this is only a small sliver of what is, or could be, accomplished by the symbiotic relationship between databases and ML. This dimension often does not take advantage of the optimization properties of the tasks involved, or the complexity involved when either optimizing for performance or ease of use. AI-Native Databases exploit these capabilities to provide deeper integration with additional optimizations in both directions and increased ease of use, which can significantly alter the nature of the relationship between ML and databases. Furthermore, the expertise that can be shared between the two areas can help drive deeper innovation.

There are three distinct dimensions where knowledge sharing, optimization, or ease of use can take place. The first two of these dimensions focus on ML algorithms decision-making tasks: the incorporation of ML primitives within traditional databases to add automation and ease of use, and the acceleration of the traditional algorithms with the use of databases for these algorithms. The third dimension concerns the streamlining of the data management and processing for the domains in which ML is applied, which also opens additional functions never considered.

3.3. Data Management and Processing

Even though data integration tools are commonly available, they present a variety of challenges, particularly around the volume and variety of data being used. First, the separate integration of data reduces the ability to reason over disparate data. Second, traditional data integration processes are largely developed by hand by engineers and, as such, are brittle, require a significant amount of engineering effort, and are difficult to maintain, both because they are not aware of the semantics of the data and because data integration is not a one-time task. In addition, traditional data integration pipelines are designed according to a specific purpose and, therefore, only provide a limited number of capabilities to integrate data from alternative modalities and different points in time. Finally, with data science becoming more standardized and ubiquitous, the level of expertise required to create complex data integration pipelines is dropping. Therefore, it is vital to have tools that allow data practitioners to compose complex pipelines easily. These observations highlight that traditional central data management systems and systems customized for a specific task can complement each other for building pipelines. Indeed, pipelines are increasingly designed to execute their entire workflow — ingesting and processing, training and evaluating and then deploying models — without human intervention in a closed loop manner where they reside within the centralized database management systems and leverage the general data management capabilities of such systems and/or their own versions of data management capabilities specialized for pipelines.

Given the importance of database capabilities for performing these workflows, it is essential to tightly integrate these capabilities within the database systems since they run as subroutines of the entire task. Indeed, we can view data management as the means to perform the system-level optimizations required to maximize throughput, minimize latency, and reduce resource consumption for the frequent invocation of database-task subroutines during the execution of the entire task. The database capabilities are embedded in the system in two complementary ways. First, existing capabilities such as fast data retrieval, data-level caching, and parallelism for data retrieval are automatically used by the pipeline without requiring the developer to use such optimizations.

3.4. Benefits of AI-Native Approaches

Over the last decade, advanced AI techniques have achieved unprecedented success in a wide variety of fields, due to their ability to deliver superhuman performance. In many industry sectors, organizations are eagerly adopting AI solutions to accelerate business innovation and maintain a competitive advantage. But the deployment of AI techniques is not limited to open-ended applications. Increasingly, AI algorithms are being put to work in an assisted fashion, as cognitive services that automate processing and decision making for specific domain functions. These developments would suggest a radical transformation of data management systems. Long gone would be the days of exclusive management of tabular data and the direct treatment of each query with explicit functions that crawl data for processing. At the other end of the spectrum, traditional data management systems have long been eclipsed in speed and performance by specialized solutions achieving optimal performance by departing from data-genome scale. By contrast, the AI-native approach looks to a truly symbiotic relationship between the broadly applicable natural analytics of learned function and the systematically optimal function of old. Queries originate from traditional operations such as ingestion, enrichment, annotation, feature construction, and the storing of explicable artifacts that are invoked by the AI algorithms employed in cognitive services. Therefore, AI-natives use the analytics of learned prediction to flexibly compose and/or trigger sequencers that orchestrate the set of data operations specified by the query. AI-natives employ algorithmic execution engines, whether serving up computational functions or triggers for reconfiguring analytic workflows. AI-native systems utilize statistical Almaden that automatically stores chunks of data over which sector-wise PCA is performed. When triggered, the rank-reduced PCA transform then compresses and pre-processes the relevant data. Control logic generates ad hoc predictions, either from retained models or by submitting for learning the small amount of residual data that maps external inputs to the desired target variables.

3.5. Real-World Implementations

The idea of AI-native data systems has already inspired work in both commercial and open-source projects, as the concept is still nascent. Corporate landscape examples include the launch of Oracle's AI-Native Database, as well as the tenets of the Data, AI, & Insights pillar at Databricks. ChromaDB, Vectara, and Weaviate are examples of open-source databases that have embraced it in their architecture. As such, it is possible to highlight certain traits that those systems expose.

While traditional databases were designed to be able to answer a large array of queries, internalize arbitrary logic to impose business rules, and support multiple use cases, these emerging systems typically focus on one specific type of retrieval task, without any other forms of logic internalization or query optimization. For example, automatic question-answering of complex documents or embeddings-based semantic search over long text have become incredibly popular, thanks to massive models. Those data systems typically apply "develop-once" and "operate-over-time" machine learning and AI models, where the model is usually non-intrusive and agnostic to the structure of the data being queried, which differentially enables their relative accessibility as compared to traditional databases. Generally, this option is seamlessly orchestrated within the data system at query time, abstracting it from the user.

Use of non-intrusive pipelines with a core, shared model across queries is in many cases sufficient to guarantee real-time performance. However, it is important to notice that it does not always apply. Scenarios such as revenue forecast, demand metrics, and pricing optimization along promotional periods over sales data, or travel time estimation over traffic data are good examples where occasional re-optimization of the algorithm has been required for business operations. Such scenarios require specialized incorporation pipelines and do not rely on algorithms trained over other databases and are difficult to automate in an AI-native mode.

4. Low-Code/No-Code Platforms

As technology continues to grow and change, so does the development of complex methods for doing everything from accounting to data management. Therefore, what does that mean for the future? It means simpler methods will take over. Businesses won't care how you get the data formatted, they just want it done and they want it done fast. They will turn to low-code/no-code platforms to simplify their tasks, lowering their costs and their learning curves. By using fewer resources on creating these complex, all-inclusive search engines, they can turn their focus to enhancing them.

Powering these specialized databases will be low-code/no-code data pipelines and analytics. Specialized tools that hide the complexities of knowledge engineering and machine learning will enable anyone in the organization to analyses and summarize the regular data. Simply building a data pipeline to filter the data down to common characteristics and programmed, or low-code/no-code machine learning tools will then build the analytics for producing summaries of the large claims database, identifying changes in patterns that indicate fraud.

4.1. Introduction to Low-Code/No-Code Development

The success of software development heavily relies on skilled engineers. Unfortunately, there is a national and global shortage of software talent, and businesses offering lucrative salaries are poaching talent from adjacent industries. At the same time, the rising costs of hiring software teams has created an explosion of work backlog within businesses. This backlog is especially burdensome for business units that did not prioritize building strong cross-functional engineering teams. Low-Code/No-Code platforms help alleviate the pressure on scarce engineering resources by allowing business users to develop, modify, deploy, and manage software applications. These platforms often have simplified graphical user interfaces and an abstraction of the underlying codebase, facilitating collaborative software creation among software engineers and business users.

Despite the catchy name and simplified interface, Low-Code/No-Code platforms are not a panacea for all software development problems. These platforms are most effective for simple applications with well-defined outcomes and minimal architecture risks. Activities such as systems integrations, managing complex customer interactions, ensuring data integrity for sensitive clients, and processing large volumes of real-time transactional data still require specialized software engineering skills. Predictions suggest that up to 80% of business applications will be built using Low-Code/No-Code platforms in the next few years. The debate is no longer whether these platforms will succeed but how they will affect the evolution of the software industry.

4.2. Key Features and Tools

Low-code/no-code development platforms provide a wide variety of key features, targeted at a varied audience: designers, developer teams, or non-developers, with varying development capacities. These features include GUIs for an easier design of pages and internal tools screens, interfaces to wire the internal tools to APIs from other software or internal company software, or tools to help in data management design, along with improving audit capabilities. Here we classify the tools based on the main platform user, internal tools end-users, and IT/Development team members that oversee the use of these platforms and their implementations and use cases. Internal Tools User Features. Internal users of internal tools built using a low-code/no-code platform are primarily business users. They are the ones that will be interfacing with the software, and thus, have three special needs. A GUI to facilitate interacting with the presented data is a necessity. Specific data to display is often linked to SQL queries or REST APIs that bring back the displayed data. However, some features, like being able to sort or filter data tables, are frequently requested by users. Visibility and data manipulation rights map roles inside a company, an area where some platforms are stronger than others. Ease of use is another requested feature. Such tools are part of the daily lives of business users in the marketing, sales, HR, and support teams, who need to receive and act on tasks assigned to them, as quickly as possible. Low-code/No-code Platform Developer Features. Low-code/no-code development platform developers are specialized profiles, normally part of a centralized team in charge of responding to requests from multiple departments with internal tools across the organization. These developers need tools that help them. Such developers need to connect the platform to backend APIs from multiple internal and external systems: HR software for onboarding processes, email and other messaging platforms for task assignment, and CRM marketing software for data enrichment, to name a few. Some industry-specific internal tools, like ERPs for supply chain management, also use low-code/no-code platforms.

4.3. Impact on Database Management

Database management systems (DBMS) have a well-established set of concepts and abstractions to manage database infrastructure, but their traditional role is shifting. Nowadays, most databases are constructed from mega-vendors that offer Database as a Service. Cloud vendors and other managed database services are understanding the operating environment of databases and investing in self-managing capabilities that push towards zero administration effort. Designers of data applications no longer install or maintain the infrastructure on which databases run; they simply make use of cloud services to do this on their behalf.

One major consequence is that DBMSs are now primarily focused on building complex, large-scale, rich, transactional, performance-demanding but narrow-scope applications. These applications are known as OLTP and may be powered by a variety of different types of databases. Other types of repositories such as data lakes may be used for general-purpose analytics purposes. In an architecture where a set of specialized repositories execute specialized tasks, we see data orchestration and ETL efforts rising in a magnitude that is draining company resources. The move towards a low-code and no-code approach forces a new approach to traditional and emerging problems.

Most LCNCDP favour the adoption of a few very popular sets of database technologies and operational environments. This specialization allows for rapid prototyping, guild specialization, enforced integration patterns, ready-made connectors, etc. but there is a latent risk of overloading a few vendors. These constructs are known as "fractals" and are seen in many-effort domain companies. However, at some level these applications cannot be created from services and connectors anymore and risk losing performance.

4.4. Case Studies of Successful Implementations

In this section we present three concrete case studies. Each of them explains how low-code/no-code platforms helped organizations solve implementation problems.

Case Study: Miami-Dade County Works to Improve Operations with Locally Built Apps Miami-Dade County's Internal Services Department supports various operations including human resources, facilities maintenance and repairs, information technology and telecommunication services, and budget and management. For the County's ISD, the budget process took about six months every year. It involved each department submitting budgets in various formats. Managers had to call multiple times to get adjustments made. They used spreadsheets to put together the budgets for review. The reviews involved multiple versions as some budgets needed to be revised several times. Then, the Basement consolidated the budgets into a large spreadsheet for the County management to review. After a few meetings, the County Mayor submitted the budget to the Board of County Commissioners for review and modifications which took as long as five or six months. The budget process needed an improvement.

The process radically changed when the IBD used a low-code platform. It was a major shift for the department from doing things manually and using spreadsheets to a more efficient process. The time to prepare the budget decreased from over six months to less than five months which represented an almost full month of the time of the employees involved reviewing and revising the budgets. A major advantage is the reduction of errors. The whole process became much more efficient for both the management and the IBD employees because of the constant interaction between the two groups. The budget platform also facilitated communication between both groups.

4.5. Future Trends in Low-Code/No-Code

As stated, the future of low-code/no-code is not limited to a simple expansion into new sectors but to the rise of new tools that allow anyone to create new and complex applications, taking away the need for the help of professional programmers. Right now, for the case of low-code the tools are simply some tools that usually would be paid for by professional programmers, either a code editor, an IDE or simpler tools for APIs, etc. But how about if a business powered by 10 low-code projects eventually releases a tool that can serve as an IDE for corporate business projects, an IDE for 10 existing LC/NC projects. Low-code development may have the potential that with the rise of low-code projects, the most known chosen code will be incrementally improved, so that any new similar project can be simply modified from the available code.

What about no-code? No-code is simpler and in theory easier to reach and have the advantage of being entirely people-focused. It is common that new breakthroughs in technology will reduce the cost for large sectors of the population of serious risks, such as risking their money by providing money savings that reduce the risks of their investing. From that perspective, the reduced overhead brought by people-focused no-code-like systems that companies could eventually build would mean reduced services and product prices for the greater population. In this sense, as we have seen before, companies have taken advantage of access that technology offers in allowing a greater number of people to meet or rent spare bedrooms. If there is a good demand and no solution, and no-code technology continues to mature and decrease in cost, people could do much easier simple E2E projects that provide new services that solve the problem.

5. Comparative Analysis of Database Technologies

Despite emerging radical architectures with a visionary impact, databases in widespread utilization today have undergone decades of evolution and cataloguing. Thus, it is no surprise that pragmatic deployments of production systems across multiple disciplines often involve mixed deployments of database technologies, creating what is referred to as “hybrid” data management systems. In the same way that other areas of the computer infrastructure have settled on standard taxonomies and mature best-known practices, so too has the domain of database technologies. The different shapes that database deployments take, and the orthogonal space of distributed-system infrastructures imply that there is no single-cut answer for any specific use case; the solution space is explored largely in terms of principles of trade-offs related to the requirements imposed by the application workloads, as well considerations imposed by the modelled problems and the solution patterns.

The remainder of this section reviews the comparative landscape for databases in current production use, by outlining the most dominant existing technologies, along with their primary attributes and motivations for consideration or dismissal. We begin first with a discussion of performance and scalability and summarize the main requirements that the different classes of technologies enable with regard to compliance and security. We then turn to a detailed discussion of the merits and drawbacks of the various database implementations, and how they relate to the underlying use-case workloads. By performance metrics, we refer to the client centric properties such as latency, throughput, resource footprint and their variations over a spectrum of load as seen by the client.

5.1. Performance Metrics

All database systems have been designed for certain type of workloads. Traditionally, database systems have supported the Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) workloads. However, nowadays most solutions put more focus on solving either of these workloads, departing from old fashions of performing both types of workloads using a single database. A clear path that most vendors have adopted is to deploy a hybrid architecture to resolve these different needs, delegating OLAP queries to external data warehouses while maintaining thousands of transactions being processed in core databases.

The OLTP is typified as a high concurrent performance workload but for short running transactions, that typically generate high throughput response times while OLAP operations are characterized by their extended duration, executing less often across the full data set. Making a comparison of two database technologies designed for OLTP and OLAP workloads may not fit the premises of a reasonable analysis. However, both workloads require different approaches on how to define performance, finding three families of benchmark solutions designed for transactional processing and three for decision support.

Considerations for OLTP systems are number of users, number of transactions per user (mix), average transaction response time and throughput. OLAP systems have in common, the intensity of the load of the ad-hoc queries, the extent of the result, strength of the update, and the nature of the queries submitted to the system. Hybrid workloads that normally do not act well with shared environments and are normally recommended to run separately, are growing demand for the current enterprise environments asking for faster and mostly reliable SLOs. The question beside these scenarios is then how to perform a realistic benchmark test that models this type of workloads.

5.2. Scalability Considerations

Achieving scalability is one of the primary goals of many of the new data technologies. "Scalability" comprises different elements including the ability to horizontally add additional nodes to handle increases in data sizes or query loads. Scale-out architectures offer obvious advantages for many use cases, especially for read scalability. However, not every type of workload scales equally well with a large increase in nodes. Most transactions need to touch a small number of nodes. For certain relational models requiring ACID transactions with ex ante specified triggers, the cost of ensuring durability, especially when coordinating multiple threads or processes touching the same row or tuple, is too high for typical web and large-scale applications. ACID has been overhyped by vendors in the last decade or so; but many applications just need BASE and don’t require “all” of ACID. Shared nothing and physical partitioning are an obvious way to scale-out, and all the cloud vendors have offerings along those lines. However, there is no magic bullet—different architectures scale to different degrees and for certain workloads. Moreover, shared-nothing and physical partitioning by field in a key-value store does not work for a generalized graph, since all nodes are interconnected. This is not to say such graph stores are not useful for some class of very large graphs—but then both scaling difficulties and theoretical inconsistencies need to be carefully managed, by adopting purpose-built data flow architectures like graph traversals, or elastically partitioned by nodes and accessed in specific manners to minimize inter-nodal communication.

There are significant differences in the scalability architectures between data stores that are adopted. Certain types of geo-distributed stores scale relatively easily in certain ways; there are relatively high costs in replication around multiple datacentres and low query speeds for certain classes of queries, especially when queries need to aggregate information from different available physical partitions. Overall, careful analysis of architecture details, and tuning for specific workloads, are critical to managing scalability for specific applications.

5.3. Security and Compliance

Increasingly, organizations are looking to the cloud for their database needs, particularly considering concerns about availability and reliability. They are offloading responsibility for much of the worry about these issues on third-party vendors. While this is understandable, it does raise serious security and compliance issues. Part of the value proposition for ubiquitous cloud access is that sensitive corporate data can be made available to those in the organization and the ecosystem who need it when they need it. However, this does raise the possibility, and it is reflected in compliance standards, that confidential information might not be accessed only by those with the legitimate need – not only employees but during business requirement coordination, also contractors that work with the enterprise on a periodic or ad hoc basis. Access control systems must be put in place to continuously evaluate access permissions and to revoke them immediately when a need is no longer present. There are also strict rules about the levels of security required depending on the type of data being processed. Traditionally, these focus on data-at-rest and on data-in-motion, and comprehensive risk analysis must be conducted to determine the additional security measures necessary to ensure that these data protection goals are achieved. With the advent of new privacy preservation regulations, databases must be able to meet new accountability and discoverability requirements covering the entire data lifecycle.

It is critical for organizations to partner with cloud vendors that are committed to keeping data secure and compliant. A good cloud vendor will take several precautions, including encrypting the data even at rest using different keys that only users with proper access credentials can unlock, putting granular access control policies in place, and creating employee training programs to highlight the importance of keeping confidential data secure and the risks associated with compromise. Additionally, the vendor should offer regular compliance certification reviews. Database and cloud vendors are continually requested for information about their compliance plans and processes.

6. Integration of Emerging Technologies

Emerging technologies may also impact databases and data management, or at least the models and instruments we must deal with new waves of data produced or required by these technologies. Many of the traditional data management technologies were created to store and process business data over the last decades. Business applications have very HR and consensus protocols, with data being available at a low latency written mostly by a few users/sources, but with very unlikely mistakes or manipulation. Emerging technologies may provide us at the same time with data structures and data management requirements in a different way. These different data structures and models for data management are also related to the different world “connected” by other ways by the New Technologies. Databases engines defined as NewSQL are an option that has been proposed to also support New Data Management Perspectives under the New Technology.

6.1. Blockchain and Databases

Blockchain and Databases Blockchain is a kind of data structure and processing found of databases, using some management protocols, like consentient, replication and consensus snapshot. As the database was invented for banks and their transaction processing, able to handle a huge number of consistent transactions with a bank, security, and recovery from faults, faults that are potentially guilty of huge amounts of Gipps on banks, block and chain lead us to data management for digital currencies, creating with bitcoin Smart and Transaction. This new way that proposes a transactional machine is interesting not only for cryptocurrencies and bitcoins, but also smart contracts and the creation of a new infrastructure for handling transactions in a particular scheme without the intermediary of banks. Proposed a new structure with special properties that leads to the “block and chain” exploitation, its novelty cannot be the exclusive consequences of cryptocurrencies.

As a technology that enables the storage of data in a distributed fashion, blockchain technology is often compared to traditional databases. Clearly, a comparison to replicated databases is very relevant, but a comparison with non-replicated databases is equally relevant. However, for the recent advent of blockchain technology, databases could only be deployed in architectures where data was centralized or replicated using synchronous or asynchronous protocols.

Because its data organization directly aligns with basic principles of traditional databases, blockchain has been described as a decentralized database. Still, this appearance can be deceptive because blockchain is decentralized and is not owned by a single organization in the way a traditional database is. This is important to many applications that have suffered from single-point failures and challenges with centralized governance, such as digital currency or assets. In fact, owning an asset without a unique bank number is often essential for its existence. This is also true in the reverse case of well-known replicated databases. Despite these advantages, other traditional database features are more limited in a blockchain, where data can be read but not generally updated, or data consistency is typically reduced to a model of eventual consistency. Furthermore, maintaining a complete index while supporting a high volume of indexed data and transactions is a challenge.

Thus, it is conceptually clearer to analyses the uses of blockchain that fall outside the scope of traditional database technologies. Transactions involving assets are currently the most common use of blockchains. All transactions have identified outputs that imply ownership of digital coins, even though the transaction inputs reference the unique identifiers instead of the identifiers associated with a specific coin.

6.2. Internet of Things (IoT) and Data Management

Internet of Things (IoT) and Data Management Another important topic is the role that databases could present for the Internet of Things (IoT). This topic is important because it is a supply of a huge amount of data transmitted with special requirements of sampling frequency and latency that are either one of three options: frequent updates of non-consistent new data or frequent update of consistent data. The two other input options are the update of consistent small groups or single and consistent data.

The second technology is the Internet of Things (IoT), a new generation of the Internet that connects huge masses of physical objects through the Internet, enabling the collection, exchange or storage of data. The IoT offers enormous opportunities and poses many challenges in different fields. The most dynamic field is smart cities, which cover mobility and transportation - autonomous vehicles, connected vehicles, or smart roads - energy - lighting and electricity - eHealth - smart wearable devices - or services - smart parking, mobility, and governance - among others. However, other sectors are also undergoing digitalization, such as logistics, whose new business models are based on the use of IoT and the collection of its data, or agriculture, in precision farming.

Privileged are cyber - physical systems, the technological leap that offers the convergence of the cyber domain - composed of physical, virtual, semantic cyberspaces - and the physical domain, which are economic, social, climatic, and political infrastructures. The digital revolution and the datafication of the economy offer endless possibilities, but they also hide great challenges, especially in terms of economic - the creation of monopolies associated with the collection and control of data - or security - the life of connected objects. Data management offers many challenges in the IoT, both envisaging innovative data models or programming interfaces, integrating heterogeneous data and in real time - the rapid business decisions are based on this data - with data coming from devices that have uncertainty, inaccuracy, incompleteness or that require timeliness - knowing the humidity index in a field to decide when to irrigate it.

7. Future Trends and Predictions

In this closing chapter, we attempt to look into the future of databases. However, it is quite impossible to be specific what exactly the future of databases is. Technology vision from both industrial and research perspectives certainly differs: Industry often invests in the near-future with expectations for increments of services and technologies available. In contrast, research is motivated by the long-term impact with an emphasis on fundamental advances. To an extent, industry and research have different speeds of development as we can see the rapid introduction of certain data management capabilities by cloud vendors in series and parallel with new database system research prototypes being developed in universities. As usual, the truth is probably somewhere in the middle. In this chapter, we touch upon some emerging trends that can interactively shape the future of databases technology and its research.

7.1. Evolving Data Architectures

Data systems continuously evolve with new data formats, new types of data coming from new sensors, and the incorporation of domain-specific knowledge into how data is gathered, served, and stored. Increasingly also data architectures are not focused entirely on storing and reliably serving data. Data architectures are also focused on data processing, offering complex transformations on the data in close to real time. One example of an infrastructure data architecture that services external applications are search engines. Examples of applications that would use a real-time data processing architecture would be a service that generates 3D animated clips according to user-defined attributes and a mapping service that incorporates dynamic travel time data.

A key concept in the evolution of data architectures is to embrace optimized subsystems with a soft layer for data persistence. The soft-persistence layer is where data is stored long term and acts also as a backup or as a lower-cost option for ad hoc, non-critical querying. In general, data architectures will become solution-oriented, where the data architect fits the various pieces of data management functionality to specific application requirements around data freshness, query complexity, and service scalability. For key applications, the work on defining the solution-oriented architecture will be carried out in conjunction with the application architects.

7.2. The Role of Data Governance

While the current demand for data products and services may be scratching the surface of a vast ocean of possibilities, the future of databases and data-centric systems is not purely a problem of technology. Existing corporate, regulatory, and government policies and processes that have been used in the selection, governance, and management of information resources will not, for the most part, enable the realization of this potential. In their current forms, such processes are antiquated and are generally governed by a lack of data literacy. It is only through the combination of better governance of data management operations, aligned with regulations and laws where needed, plus an accelerated effort to promote data fluency at all levels in all organizations, will the true promise of the digital economy, powered by data, be realized. But this is a tall order. From a data governance technology perspective, the reality is that few existing solutions meet the pressing need for better data management technology that is both agile and meaningfully aligned with the data asset strategy of a business and its data-related business objectives. By "meaningfully aligned," I mean that the role of any data governance solution is to support the governance model employed by a company as part of its data strategy, and it is a moot point if these solutions do not allow flexible customization. Responsible data management that enables and supports the delivery of business-critical data products and services at scale, while monitored and governed by such data governance solutions as colliders, cannot be an afterthought or something handled by an additional layer of controls and checks placed at the end of the data management chain.

8. Conclusion

The Future of Databases is the matter of belief as well as foresight. It is as difficult as telling the future of fashion. The pace of changes doubles on approximately every 5 years, so, some of our present-day database systems may become obsolete and may not even be available in 20 years from now. Other systems will form the backbone of all modern IT development. It is as difficult as doing market research for technology around the founding year. The total expenditures for services, tools, applications, merchandise and infrastructure were estimated at a significant amount, in that case, the market would be larger than the computer services market, which included hardware maintenance contracts, business applications consulting, and systems integration activities.

There are many databases centric services which will enable enterprises and institutions to provide secure self-service access to operational systems across enterprise boundaries, with local and distributed support for reporting, query and analysis. Paradigms of the future will facilitate access to logically centralized data within heterogeneous databases created, maintained and controlled by nontrusted providers. These increasing and changing demands require new database capabilities in future database systems. Other areas may change the face of computer systems in the long-term. Although its origins go back to the 1970s, the advent of Wireless communication is nothing short of a revolution that has appeared in the last few years. Communicating devices will be joined by many other types of small, embedded devices. The Internet network almost doesn't exist in the early years. The present days Internet is growing at about 100% every year and accounts for a significant percentage of the workstations in companies. This is why the research and development of the future databases are still at the beginning. In this paper, we have painted a current panorama of the main lines of innovations in many domains that touch databases.

References:

[1] Grover, Lov K. "A fast quantum mechanical algorithm for database search." *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996.

[2] Jakobi, Markus, et al. "Practical private database queries based on a quantum-key-distribution protocol." *Physical Review A—Atomic, Molecular, and Optical Physics* 83.2 (2011): 022301.