Databases â€“ SQL vs NoSQL

In this train, we take a look at the differences between relational and NoSQL databases. We also explore the advantages and disadvantages of each.

Learning objectives

In this train, we will review the following concepts:

- The advantages and disadvantages of traditional relational databases.
- The advantages and disadvantages of NoSQL databases and how they compare to relational databases.

Introduction â€“ Venturing beyond relational databases

Throughout our learning journey, we've spent a great deal of time aroundrelational databases. It cannot be understated how important these data systems are, as their influence is present in large parts of our everyday lives. From a practitioner's perspective, however, while relational databases form the backbone of many applications, their utility can be limited to small or medium-sized data scenarios, often failing to cope when scaling to big data environments.

With this limitation in mind, our goal in this train is to explore alternative technologies that offer the same functionality as relational databases but can do so in a performant manner under big data constraints.

Before we begin, let's detail some of the strengths surrounding relational databases and compare these to their shortcomings in the current big data era.

The advantages of relational databases

Relational databases provide major benefits when querying and writing structured, correctly normalised data. These benefits include:

- **Query flexibility**: Through the use of structured query language, we can always find the data we need in a correctly normalised database using joins, views, filters, and indexes.

- **Compliance**: All transactions performed within popular modern database management systems (DBMSs) are atomicity, consistency, isolation, and durability (ACID) compliant. This provides guarantees against malformed database states.

- **Safety**: The presence of a defined schema and strict constraint checks, including column types, max lengths, and optionally not allowing null values, can ensure that database entries are valid and guarded against insertion errors.

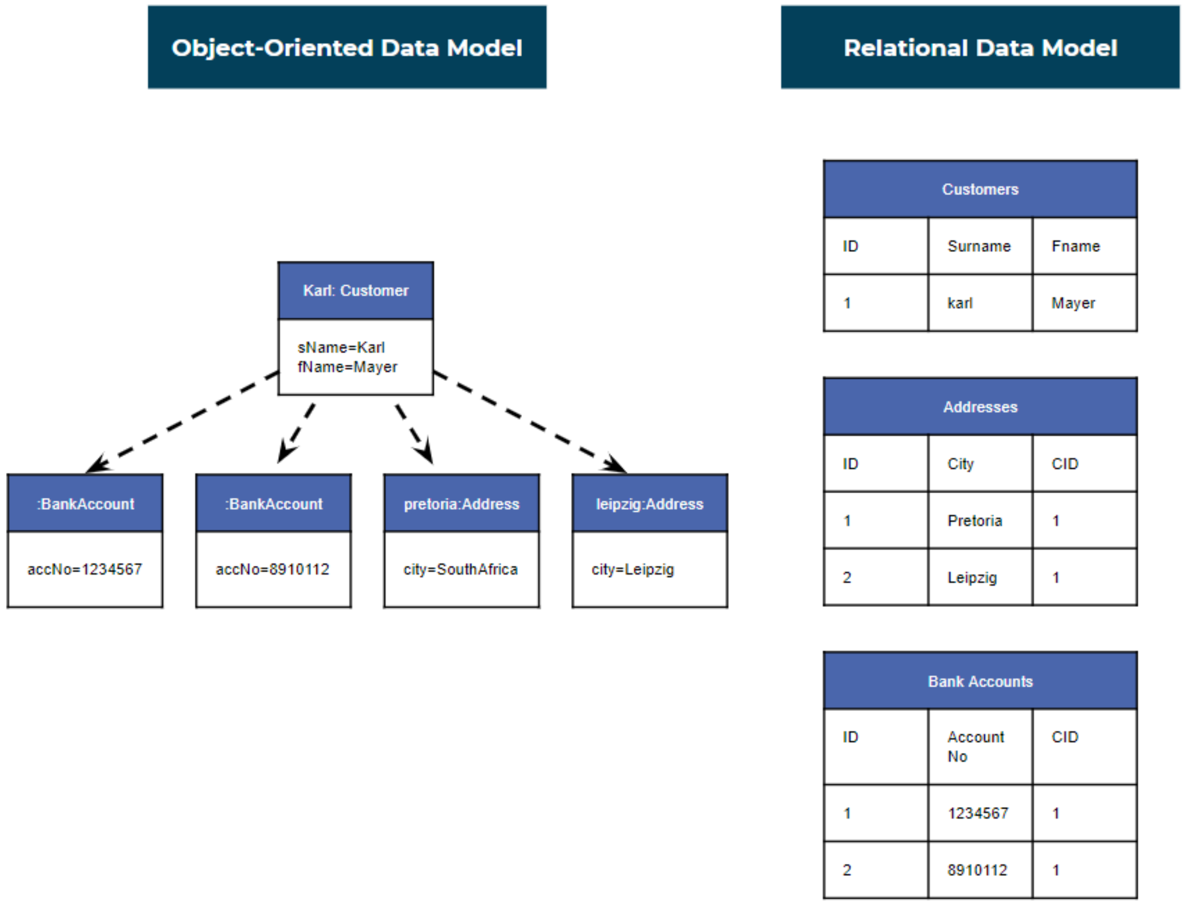
- **Maturity**: SQL and its related database technologies have been in use for over 40 years. This lengthy tenure provides stability to the technology with a large and entrenched community behind it.

- **Deep functionality**: Along with their long-held support, relational databases support a wide array of functionality, including triggers, stored procedures, advanced indexes, views, etc. that allows them to solve diverse and complex problems in elegant ways.

As noted, however, relational databases are limited and do have their fair share of drawbacks, including scalability, schema evolution, and performance. Let's discuss some of these aspects in greater detail in the next section.

The drawbacks of relational databases

- **Object-relational impedance mismatch problems**



Customers

ID	Surname	Fname
1	karl	Mayer

Addresses

ID	City	CID
1	Pretoria	1
2	Leipzig	1

Bank Accounts

ID	Account No	CID
1	1234567	1
2	8910112	1

Figure 1: Multiple normalised tables to achieve equivalence.

Due to the inherent mismatch between the object-orientated and relational data model paradigms, an object-oriented entity often needs to be represented by several normalised tables to achieve equivalence, as depicted above.

Most apps or tools that we use on our phones and computers are written in an object-oriented programming (OOP) language and implement related concepts such as abstraction, encapsulation, inheritance, and polymorphism (big words, but they are what separate these languages from other functional and procedural types). However, OOP languages and relational databases don't always get along. In the OOP world, instances (known as objects) have established relationships through references (polymorphism). These languages build a graph or hierarchy to establish these reference connections.

On the other hand, relational databases use a row for each entry (instance) and columns for properties of the specific entry. If we want to store our object graph in a relational database, we will need to extend our database to multiple normalised tables. This act in itself can break many principles upon which OOP is based (such as encapsulation), causing a direct mismatch between these data modelling paradigms.

- **Difficult schema evolution due to an inflexible data model**

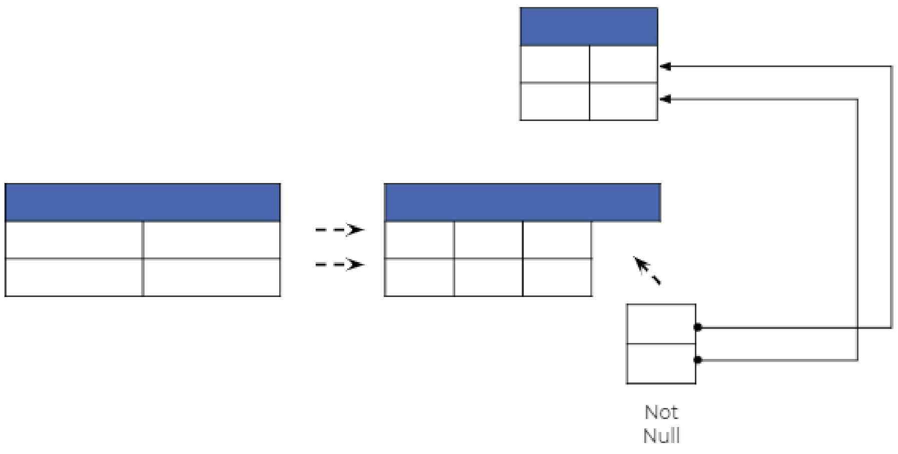


Figure 2: Updating a relational database's schema can be a difficult and painful process.

Relational databases consist of tables containing key relationships of our data model. This process of data modelling is performed upfront and is designed to be fixed in time. For instance, it could become very difficult (not to mention stressful) to add or remove a column in a production database with many existing entries. Here, to undergo a needed transformation, one or more migration scripts would be necessary.

The added difficulty and stress of updating a relational database's schema can also become a development obstacle, limiting system improvement and innovation. For instance, working on a database that is in production may introduce enough potential business risk that developers decide to only create features around the already existing database and its definition. This hinders the constant improvement (continuous delivery) of a product and leads to misuse of the schema.

- **Poor horizontal scalability**

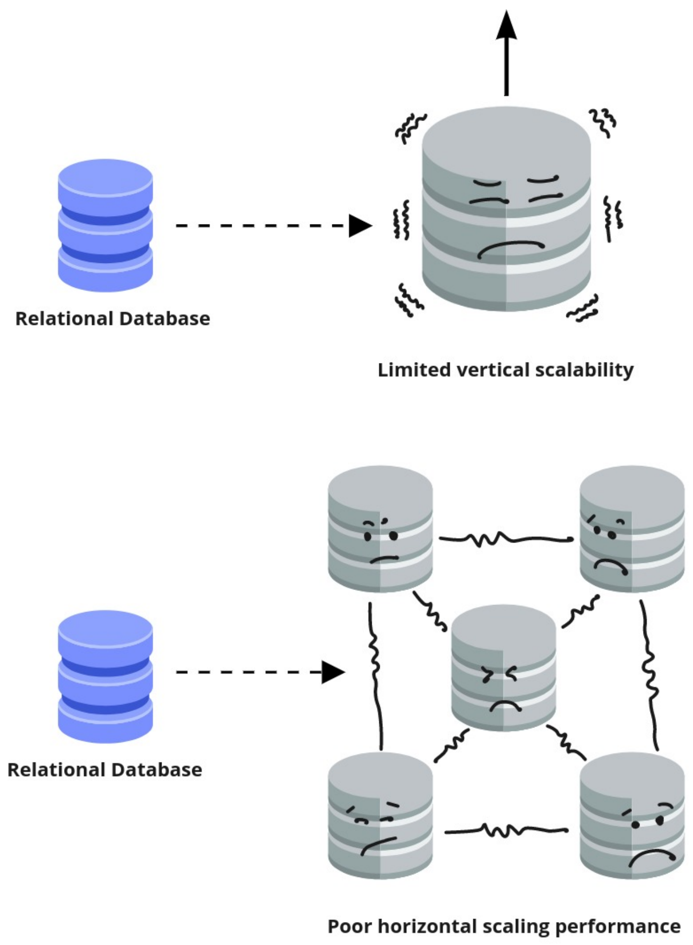

Relational Database
Limited vertical scalability
Relational Database
Poor horizontal scaling performance

Figure 3: Traditional RDBMSs struggle to scale appropriately to meet the demands of big data.

Relational databases can scale vertically. This means that by making more CPU power and RAM available to a relational database system, we can improve its performance, effectively increasing the number of requests that the system can handle. However, this solution is limited by the maturity of the underlying technology and the budget we can spend on scaling resources (which can become immensely costly when using high-performance infrastructure).

In contrast to the relative ease with which relational databases scale vertically, these systems, unfortunately, do not scale well horizontally.

Horizontal scaling is the process of distributing a single database system over a cluster of multiple machines, enabling the system to process numerous requests concurrently as the processing load is shared amongst the machines of the cluster. The problem of horizontal scaling for relational databases is introduced due to their enforcement of ACID (consistency) requirements. This is because the mechanisms required for consistency (locks and blocking approaches performed) require multiple back-and-forth calls between the database (now split up across multiple machines) and users. These calls have to be made over the network for every single database transaction, leading to bottlenecks in database performance.

- **Low performance due to relational data constraints**

While providing great flexibility in many scenarios, the powerful functionality of relational databases and the relational representation of the data itself can impose heavy computational overheads on system functioning. For example, when a relational database is normalised (say, to correctly represent an object-oriented model), the resulting normalised tables have to be rejoined to obtain a result. These join operations are computationally expensive, leading to slow performance and generally more expensive hardware requirements. The same can be said for the enforcement of ACID-based transactions (which, as noted earlier, are further impediments in a distributed environment).

An alternative â€“ Introducing NoSQL

These days we capture a very wide array of data types and structures from an even greater pool of sources. NoSQL, a schema-less model, allows for these data types to be captured and processed effectively.

The table below provides a high-level overview of some key differences which exist between relational and NoSQL databases:

Property	Relational	NoSQL
Database type	Relational databases	Non-relational/distributed databases
Structure	Table-based	Key-value pairs, document-based, graph databases, and wide-column stores
Scalability	Designed for scaling up vertically by upgrading expensive custom-built hardware	Designed for scaling out horizontally by using shards to distribute load across multiple commodity (inexpensive) hardware instances
Strengths	Suited for highly structured data and fixed data schemas; working with complex queries and reports	Able to handle abundant unstructured data sources; suited for high transactional loads; pairs well with fast-paced, agile development teams

Types of NoSQL databases

So far we've only provided a very loose definition of what a NoSQL database is. To add clarity to this topic, let's review some categories of NoSQL database types that enjoy popular use in several application domains.

Document databases

These types of databases store data in easily parsable document formats, including JSON, BSON (Binary JSON), and XML. These documents can easily reference certain elements, allowing for easy and fast querying. The benefit of document-type databases is that data can be stored and retrieved in the exact form it is required within an app (think titles, links, and text from a wiki page). This means that less translation (expensive joins) is required compared to data stored relationally. Additionally, document databases allow developers to easily change any underlying data structure/schema, as they aren't left with a strict schema.

For example: A document database could be used to model data held by a pet insurance company. The database might contain a series of JSON documents, with each document representing a unique dog that is insured by the company. As shown below, this database entry could contain fields for the unique document `id`, the `name`, `age`, `breed`, and `owner_name`. In a relational database, these fields would be represented by columns present within one or more tables.

```
{
    "_id": {
        "$oid": "5968dd23fc13ae04d9000002"
    },
    "name": "Spot",
    "age": 2,
    "breed": "Chow Chow",
    "owner_name": "Naledi"
}
```

Given the above document model, if a developer were to decide to add a "last_vet_visit" field to one or more documents, they could do so easily (due to the flexibility of the underlying JSON syntax), and the update won't hinder the functionality of the other entries. Thus, a partial or full adaption of the data schema could be achieved seamlessly. Alternatively, in a relational database scenario, intervention by DB administrators would be required to perform a similar update (in the form of adding a feature column).

Use cases include: social network feeds, e-commerce stores, and list-oriented applications.

Graph databases

A graph database focuses on the relationship between data elements known as "links". Each element is stored as a node, such as a person in a social media graph.

In a graph database, connections are first-class elements of the database, stored directly. This means that the links within a graph database are memory pointers storing the address of a memory location containing a specific element of data. In relational databases, links are implied using data and expensive joins to express the relationships.

Very few real-world business systems can survive solely on graph queries. As a result, graph databases are usually run alongside other more traditional databases.

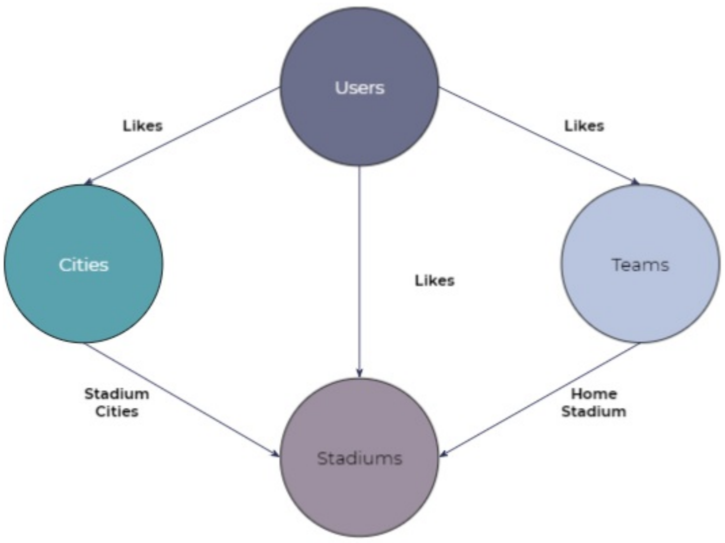


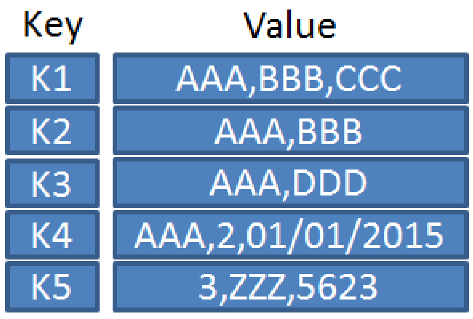
Figure 4: A graph database example.

This example of a graph database shows that one user can like a football team and one or more football stadiums and cities. Also depicted in the example graph database model, one football stadium can be liked by multiple users.

Use cases include: fraud detection, social networks, and knowledge graphs.

Key-value stores

Key-value stores are some of the simplest database types, similar to dictionaries within a programming language. Database elements are stored as a key-value pair consisting of an attribute name (or "key") and a value. In key-value databases, the key serves as a unique identifier and various values are associated with a key. In these types of databases, both keys and values can be anything, from simple singular objects to complex compound objects. Key-value databases allow for massive horizontal scaling and are, by nature, partitionable. Other than scalability, key-value stores have the added benefit of being flexible and data reading/writing efficient.



Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

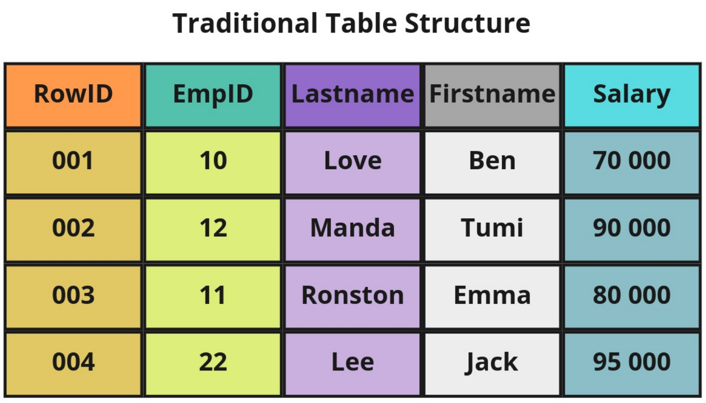
Figure 5: A key-value store depiction where multiple values, with different data types for each value, can be associated with a single key.

Use cases include: shopping carts â€“ users' online e-commerce data can be stored in a key-value database where the key responds to the unique user ID, and the products in the shopper's basket are the associated values. We can also associate favourite products and most frequently bought product categories to a unique user ID.

Column-oriented databases

Column-oriented databases store data in columns, while relational databases store data in rows. Consider the following example:

Figure 6 below shows a typical table structure containing a list of employees' names, IDs, and salaries. This two-dimensional view is an abstraction for us to view it easily:



Traditional Table Structure

RowID	EmpID	Lastname	Firstname	Salary
001	10	Love	Ben	70 000
002	12	Manda	Tumi	90 000
003	11	Ronston	Emma	80 000
004	22	Lee	Jack	95 000

Figure 6: A representation of simple data stored in a relational form.

Within our memory, the table actually follows the row-based storage model depicted in the top portion of Figure 7:

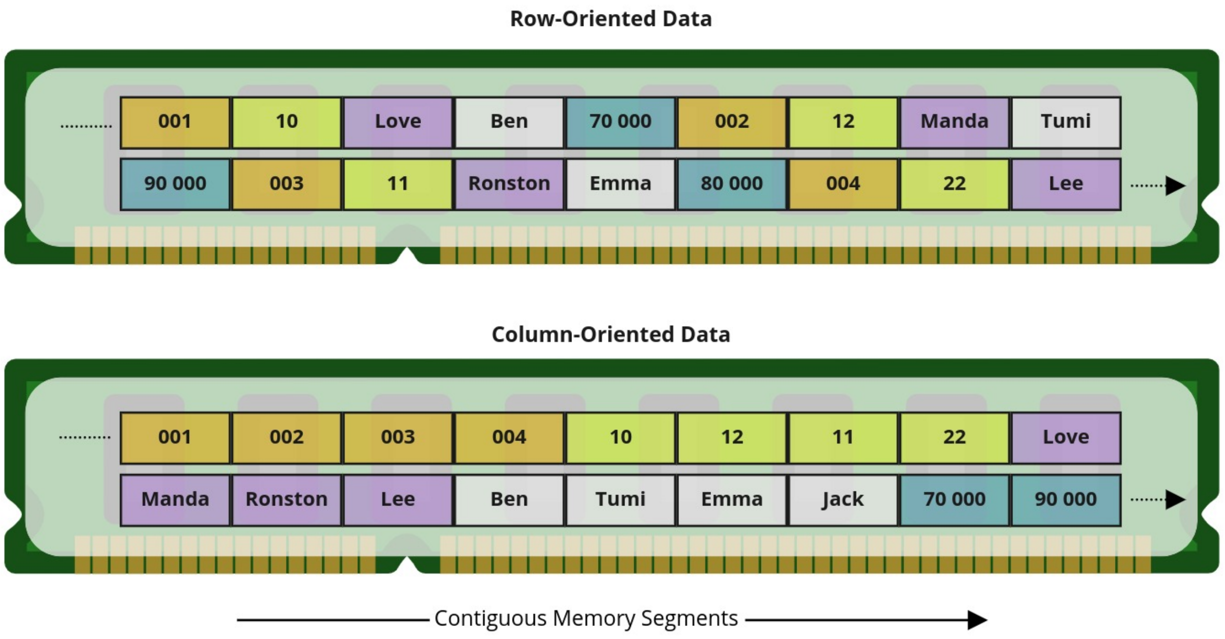

Row-Oriented Data
001 | 10 | Love | Ben | 70 000 | 002 | 12 | Manda | Tumi
90 000 | 003 | 11 | Ronston | Emma | 80 000 | 004 | 22 | Lee
Column-Oriented Data
001 | 002 | 003 | 004 | 10 | 12 | 11 | 22 | Love
Manda | Ronston | Lee | Ben | Tumi | Emma | Jack | 70 000 | 90 000
Contiguous Memory Segments

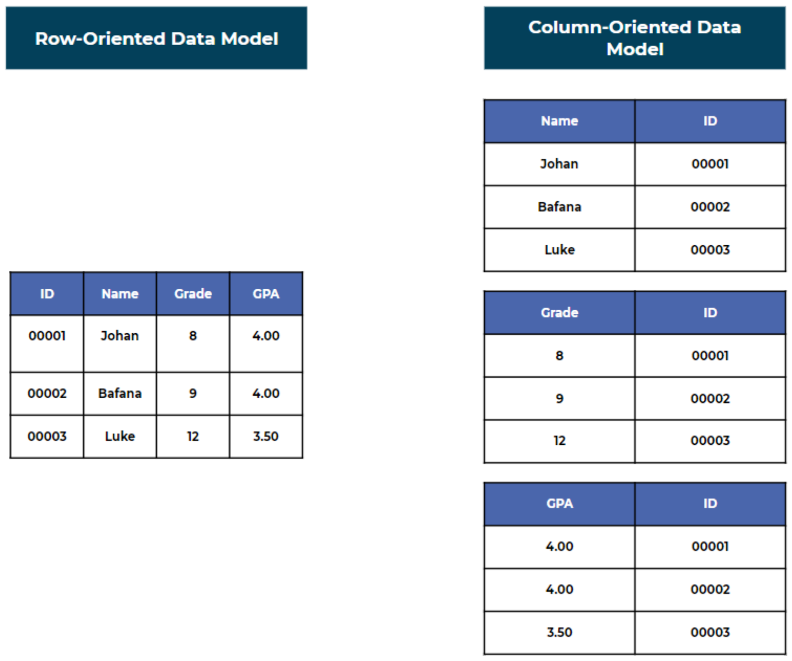
Figure 7: A depiction of how data are stored physically in a row vs columnar data format.

In a column-oriented database, on the other hand, the data appear as represented in the bottom portion of Figure 7. Instead of reading from left to right, the table is now read from top to bottom.

We may consider when one of these storage orientations would be beneficial over the other (if at any time at all). It turns out that two major factors here surround data compression and querying speed:

- When we speak of **data compression**, we refer to how much space is required to store our data in memory. Generally, if we can store the same *types* of data together, we can more easily compress it, therefore allowing us to save on precious storage space (a 1% storage saving on a terabyte of data is 10 Gigabytes, so this economy of storage becomes important for big data). As can be seen in the figure above, using a column-based storage format allows all data within a column that share a common underlying data type to be stored together, therefore increasing its ability to be compressed.
- In terms of **querying speed**, if our database analytics frequently require *wide projections* to be made (where many fields need to be returned as the result of a query), then row-based data storage is suitable. This is because the data from many fields can be obtained by reading sequentially along a memory segment. However, many analytical queries nowadays require *narrow projections* to occur, where only a subset of the fields from several tables are required to be filtered, joined, or compared to return a query result. In these scenarios, having our data stored sequentially in columns allows us to skip reading through multiple data entries to only retrieve a small portion of their total data. Thus, column-based data storage can drastically increase query performance in certain big data scenarios too.

As a final example of a column-oriented database, see Figure 8 below. Notice how the data are *split* to move from a row-oriented data model to a column-oriented data model. In the example, the userID is used as a unique value to partition the data into columns.



Row-Oriented Data Model

ID	Name	Grade	GPA
00001	Johan	8	4.00
00002	Bafana	9	4.00
00003	Luke	12	3.50

Column-Oriented Data Model

Name	ID
Johan	00001
Bafana	00002
Luke	00003

Grade	ID
8	00001
9	00002
12	00003

GPA	ID
4.00	00001
4.00	00002
3.50	00003

Figure 8: A depiction of the differences between data storage in relational and column-oriented database types.

Use cases include: analytics â€“ column-oriented databases are well suited for analytics as these database types allow for efficient data retrieval, i.e. the fact that data storage is by columns rather than by rows reduces the query time.

Benefits of NoSQL

Having considered various forms of NoSQL databases, we can now directly address their merits:

1. **Big-data-ready**: NoSQL databases were created to handle big data as part of their fundamental architecture. By removing the consistency enforcement constraints of relational databases, these systems are capable of greater levels of scalability and availability. NoSQL databases are often based on a scale-out strategy, allowing them to use low-cost commodity hardware instead of the expensive machinery involved when using the scale-up approach that relational databases take. Scaling out also allows for zero system downtime, as users can be diverted to different machines within a cluster in the event of a machine failure or network fault.

2. **Agile development**: Compared to relational databases, NoSQL allows for much faster development time. It is much easier for developers to change the schema of a NoSQL table compared to a relational one. This is an improvement and falls in line with the agile development methodology many software development teams use today.

3. **Flexible data modelling capabilities**: The object-relational impedance mismatch problem is solved by a NoSQL database. NoSQL databases are capable of modelling unstructured, structured, and semi-structured data in one database. This enables them to store data in the same form as the objects used in applications graph databases and allows for computationally efficient databases.

4. **Able to service multiple workloads**: The scalability of NoSQL databases enables them to serve both analytical and transactional workloads simultaneously. We touch on examples of these workloads in the form of OLAP and OLTP in a later section.

Drawbacks of NoSQL

Although we've stressed many of the strengths of NoSQL databases, there are drawbacks to using these systems, including:

1. **Lack of consistency**: Since these systems prefer availability over consistency, they fail miserably in cases where consistency is the top priority. One such example is in financial transactions, where there is a risk of a system failure due to the non-synchronisation of data nodes.
2. **Lack of standardisation**: There is no specific programming interface for NoSQL databases. That means there are different languages used for the design and querying of NoSQL databases, and as such, standardising tools, processes, and systems around the use of these database types become tedious.
3. **Security**: NoSQL databases are generally much more configurable than SQL databases as a result of the wide range of unstructured data they may contain. This has led to many open-source NoSQL databases (MongoDB, Redis, Neo4j) being configured less securely out of the box compared to SQL databases. This was done to limit resource usage, simplify the initial setup, and take into account that the vast majority of NoSQL data are non-critical. However, it is still possible to secure a NoSQL database through encryption and strong user authentication policies.
4. **Transactional nature**: NoSQL fails in cases where the database needs to compete against a high volume of transactions per day, as they require a highly scalable, consistent database. For example, it is important in financial

areas to facilitate fraud detection before completing the transaction and check for balance whilst on a call. The balance of achieving consistency vs availability between database types is known as the ACID vs BASE comparison.

Conclusion

In this train, we have focused on the major database types. This included a discussion on the drawbacks of relational databases and how NoSQL databases address these drawbacks. We also covered the types of NoSQL databases and their advantages and disadvantages.