FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

In-depth evaluation of NoSQL and NewSQL database management systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Michael Heinzl, BSc
Matrikelnummer 01325545

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ.-Prof. Mag. Dr. Reinhard Pichler

Wien, 13. September 2018

Michael Heinzl	Reinhard Pichler

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

In-depth evaluation of NoSQL and NewSQL database management systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Michael Heinzl, BSc
Registration Number 01325545

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Mag. Dr. Reinhard Pichler

Vienna, 13th September, 2018

Michael Heinzl	Reinhard Pichler

Erklärung zur Verfassung der Arbeit

Michael Heinzl, BSc
Xaveriweg 4, 7000 Eisenstadt, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. September 2018

Michael Heinzl

Danksagung

Ich möchte mich bei meinem Betreuer Univ.-Prof. Mag. Dr. Reinhard Pichler für die Unterstützung während meines Studiums bedanken, insbesondere für seine Anleitung und Beratung bei der Verfassung dieser Arbeit. Außerdem möchte ich meiner Familie und meinen Kollegen für die großartige Unterstützung und den Rat danken.

Acknowledgements

I would like to express my sincere gratitude to my advisor Univ.-Prof. Mag. Dr. Reinhard Pichler for the support during my studies, especially for his guidance and advice writing this thesis. Also, I would like to thank my family and colleagues for the great support and counsel.

Kurzfassung

Traditionelle Datenbankmanagementsysteme (DBMS) stehen neuen Herausforderungen in den Bereichen Flexibilität und Skalierbarkeit gegenüber. Heutige Cloud-Computing Umgebungen bieten unglaubliche Möglichkeiten und traditionelle DBMS können diese Ressourcen nicht ausreichend nutzen. NoSQL und NewSQL Systeme bieten erfrischende neue Ansätze für aktuelle Herausforderungen in diesem Bereich. Die enorme Anzahl der derzeit verfügbaren Systeme stellt Entwickler vor eine große Herausforderung, wenn sie einen passendes Datenbanksystem wählen. NoSQL und NewSQL Systeme bieten neue Funktionen und Ansätze. Diese Arbeit analysiert die Methoden und Algorithmen von NoSQL und NewSQL Systemen. Ausgewählte Systeme werden in mehreren Dimensionen verglichen, um Unterschiede in ihren Ansätzen und Technologien aufzuzeigen. Der resultierende Vergleich dient als Anleitung für Anwendungsentwickler, die nach einer passenden Datenbanksystem suchen.

Abstract

Traditional database management systems (DBMSs) face new challenges concerning flexibility and scalability. Today's cloud-computing environments offer incredible resources and traditional DBMSs struggle utilizing these resources. NoSQL and NewSQL systems offer refreshing new approaches to current challenges in this domain. The enormous number of currently available systems present a major challenge to developers choosing a fitting data store. NoSQL and NewSQL data stores provide powerful new features and approaches. This thesis analyzes the methods and algorithms used by NoSQL and NewSQL systems. Selected data stores are compared in multiple dimensions to show differences in their approaches and technologies. The resulting comparison provides guidance for application developers searching for a fitting data store.

Contents

CHAPTER 1

Introduction

The first database management system (DBMS) appeared in the 1960s. One of such DBMSs was built by IBM, known as IMS, it was developed to manage inventory and supplies. Certainly, traditional DBMSs were developed with the mindset and requirements of that time. In the next years, the computing performance started to increase and many database systems were developed. All these systems were designed to operate on a single machine because there was no need to distribute the system across multiple nodes. Therefore, database systems were only capable of supporting a relatively small number of concurrent users compared to today's requirements.

Traditional database management systems (DBMSs) in combination with a single machine could not handle that huge number of concurrent requests in a satisfying way. Scaling the system vertically by increasing computing resources of the used machine only improves the performance to some extent. However, this countermeasure is limited to the computing power of a single machine. Additionally, the migration of a DBMS between two machines, in order to upgrade to a machine with more computing power, is a complex and costly process. The migration process also comes along with significant downtime. Considering the requirements of modern web applications downtime cannot be tolerated in such systems.

One common requirement of traditional DBMSs is the need for consistency and valid data at any time. More specifically this requirement is fulfilled by supporting the ACID concept. ACID stands for Atomicity, Consistency, Isolation and Durability. Furthermore, this concept is used to guarantee reliable transactions in DBMSs.

[Asl11, PA16]

1.1 New challenges

With the beginning of the 2000s, web applications became popular. This kind of applications had more demanding resource requirements than applications of previous years, above all the need to serve large numbers of users at the same time.

With the emergence of cloud computing and BigData, the requirements for DBMSs or in general data stores started to change. Traditional DBMS-technology does not provide the needed scalability to handle the growing resource requirements. The inflexible nature along with other limitations poses new challenges to the traditional systems. Today's applications are growing remarkably fast. Consequently, high performance and scalability have become a major requirement, with regard to the amount of data, which has to be stored and accessed. To achieve this, modern applications have to be built on top of heterogeneous commodity servers.

A commodity server, in this context, is a server that is not built for a specific purpose. The hardware is inexpensive and in addition easily exchangeable. A single commodity server is not capable of carrying out a high-performance task. The benefit of using this type of server is being independent of certain manufacturers and it is easily exchangeable if a server breaks down. This kind of machines is heavily used by cloud-computing providers. To utilize these resources and reach a high degree of scalability heterogeneous commodity servers along with their weaknesses have to be supported. [GHTC13]

Web applications tend to show fluctuation in their access patterns. To cope with the fluctuation elasticity has to be provided by the application. Another requirement in this context is fault tolerance. Fault tolerance becomes more important because distributed systems as the name suggests are composed of a huge number of individual machines and each machine poses a source of error. Therefore, the probability of failure is increased by the number of used machines, which results in a significant failure rate.

Also, long periods of downtime cannot be accepted for crucial systems. Hence availability is also a key requirement for modern web applications.

All these new requirements and challenges of present-day applications influenced the evolution of DBMSs. Traditional DBMSs are not able to fulfil these new requirements in a satisfying way. Out of the need to deal with these challenges new types of data stores emerged, so called NoSQL data stores. [PA16, RFG+18]

1.2 New Technologies

1.2.1 NoSQL

The term NoSQL [Eva09], short for 'Not Only SQL', refers to a family of non-relational DBMSs. The name indicates that the key objective is not focused on SQL-based systems. An enormous number of new data stores fits in this classification. A large portion of these systems is tailored for a specific field of application. NoSQL data stores, other

than relational DBMSs, are not primarily based on tables. Furthermore, they share the following characteristics:

- Generally, NoSQL data stores do not support the ACID properties. They use a more relaxed concept, which is known as BASE (Basically Available, Soft state, Eventual consistency).
- NoSQL data stores make use of modern cloud environments, in terms of utilizing heterogeneous commodity hardware. Hence, NoSQL data stores are able to scale horizontally and allow huge numbers of concurrent requests.
- Due to the use of multiple servers and redundancy NoSQL data stores provide high availability and can handle failures of server instances.
- In contrast to relational DBMSs most NoSQL data stores support flexible data structures and schemas. A wide range of structured, semi-structured and schema-less systems is covered.

NoSQL data stores are built on top of different data structures and data models. They can be divided into four categories based on the underlying data model:

- **Key-value stores:** Items in this data store are simple key-value pairs. These pairs are stored in a hash map or dictionary. The key is a unique identifier within the data store, which is associated with the value. The associated value can be of an arbitrary form (e.g. string, list, object, ...). Therefore, queries can usually only be performed against keys and not against the stored values.

- **Column stores:** Datasets in column-oriented data stores are organized by column rather than by row. Values of a single column are stored adjacently to optimize queries referring to particular attributes. Therefore, these systems are often used for statistical and analytical purposes.

- **Document stores:** Similar to Key-value stores document stores also organize data as key-value pairs. However, the value is a semi-structured document. Most document stores support encodings like JSON and XML. The semi-structured data allows for limited querying capabilities.

- **Graph databases:** The underlining data structure, as the name suggests, is a graph. A graph consists of nodes and edges. In this context, nodes represent data objects and edges are the relations between data objects. Nodes and edges can store information in key-value pairs. This specific data model is well suited to manage deeply interconnected data sets.

These diverse data models provide a wide range of systems. A lot of systems are tailored to fit a few use cases, other than traditional relational DBMSs, which provide support for a broad range of use cases. [Pok13, MH13]

1.2.2 NewSQL

NewSQL is a class of modern relational DBMSs. They are closing the gap between traditional DBMSs and NoSQL systems and were designed to combine the benefits of these two worlds. One of the core features of NewSQL is the support of ACID transactions, in contrast to NoSQL data stores. Furthermore, they use a relational data model, although some systems may use a different data model for internal representation. The support of a relational data model enables the systems to offer SQL as the main interface language. Additionally, NewSQL data stores seek to provide the same horizontal scalability and read-write performance as NoSQL data stores. Hence, NewSQL data stores are able to process huge amounts of concurrent requests, while still maintaining consistency and integrity of the database.

In contrast to NoSQL, there are not that many systems which fit in the class of NewSQL. Also, the categorization of NewSQL systems is not that transparent as the categorization of NoSQL systems, which are categorized based on the underlying data model. Andrew Pavlo and Matthew Aslett [PA16, MBT16] distinguish these three categories of NewSQL systems:

- **New Architectures** - These systems are built from scratch and aim to fulfil all modern requirements of distributed relational DBMSs. Other than traditional DBMSs these systems are designed for distributed applications and use cases.

- **Transparent Sharding Middleware** - NewSQL systems in this category make use of already established DBMSs. A transparent sharding middleware manages the access to underlying systems. The middleware is responsible for data distribution, queries and transactions.

- **Database-as-a-Service** - The whole database stack is managed by the provider of the service. The customers of these services do not have to manage their own server hardware or infrastructure, they only have to select a fitting product depending on their needs.

1.3 Problem definition

The current landscape of modern database systems provides an enormous set of distinct solutions. Each solution is tailored for a specific set of requirements and provides different

features. All these systems have in common that they are confronted with a challenging set of requirements. The requirement to handle huge numbers of concurrent requests and the large amount of data over a distributed environment represents a key challenge to database systems. To cope with these requirements a database system has to address several key aspects such as concurrency control, sharding and distributed querying.

Depending on the underlying architecture and design of NoSQL or NewSQL systems, various approaches and implementations are currently available. The majority of the available systems use different algorithms and technologies for the same problems. The immense number and the diversity of the current systems poses a difficult problem to select a suitable solution for a given task.

1.4 Research questions

In order to reach the expected results following research questions are answered in the course of the thesis:

- What are the key criteria to compare NoSQL and NewSQL systems?
- In what manner do particular NoSQL and NewSQL systems compare according to these criteria?
- Which different methods and algorithms are used by NoSQL and NewSQL systems to cope with these key criteria?

1.5 Expected results

The first part of the expected results is a detailed list of key criteria of NoSQL and NewSQL data stores. Based on these criteria the main objective is to provide a comprehensive comparison of selected NoSQL and NewSQL data stores. This includes a detailed description of the criteria and of the used methods and algorithms. In addition, the resulting information is presented in a way to assist the reader in the selection of a suitable system.

1.6 Methodology

The following steps are taken in order to answer the research questions and arrive at the expected results:

1. Comprehensive literature research on NoSQL and NewSQL and the used technologies
2. Determination of the key criteria to compare NoSQL and NewSQL data stores

3. Systematic selection of NoSQL and NewSQL data stores based on available rankings
4. Further research on the particular NoSQL and NewSQL data stores
5. A detailed comparison of selected data stores for each criterion

The data stores are selected based on prior research and data store rankings from DB-Engines [dbe18]. The selected data stores will be used in Chapter 3 for a detailed comparison. DB-Engines ranking is based on the current popularity of the system including the following parameters:

- Number of mentions of the system on websites
- General interest in the system. (Google Trends)
- Frequency of technical discussions (Stack Overflow and DBA Stack Exchange)
- Number of job offers, in which the system is mentioned
- Number of profiles in professional networks, in which the system is mentioned (LinkedIn and Upwork)
- Relevance in social networks

1.7 Relation to Business Informatics

It is rarely the case that a state of the art web application is built without an underlying data store. In the age of Big Data and Internet of Things (IoT) scalable and flexible data, stores are essential parts of modern applications. The immense volume of data which these applications are confronted with is in many cases handled by a NoSQL or NewSQL data store. Design, development and usage of these applications are core competences of business informatics.

A wide range of fields in business informatics is highly related to database technologies to mention only a few of them: Software engineering, distributed systems, data analysis and business intelligence.

CHAPTER 2

Background

2.1 NoSQL

The term NoSQL [Eva09], short for 'Not Only SQL', refers to a family of non-relational DBMSs. The name indicates that the key objective is not focused on SQL-based systems. A lot of new data stores fit in this classification and most of them are tailored for a specific field of application.

A major part of today's applications is confronted with a huge amount of data. Traditional DBMSs are no longer a viable choice to handle this situation because they are only capable of scaling to some degree. This is achieved by vertical scaling (also called scale-up), which means increasing the resources of the machine the DBMS operates on. Unfortunately, at some point, an upgrade is no longer possible or the costs do not justify the additional benefit.

NoSQL data stores overcome this lack of horizontal scalability. They are designed to operate in a distributed environment with many nodes. Therefore, they are capable of handling extremely large sets of data by design. NoSQL data stores make use of modern cloud environments, in terms of utilizing heterogeneous commodity hardware. Hence, NoSQL data stores are able to scale horizontally and allow huge numbers of concurrent requests.

It is not only the volume of data relational DBMS struggle with, also the variety of data sets is an issue. NoSQL, in general, provides a rich set on different data stores, each data store tailored for specific needs. In contrast to relational DBMSs, most NoSQL data stores support flexible data structures and schemas. A wide range of structured, semi-structured and schema-less systems are covered. Hence, NoSQL data stores provide more flexibility than traditional DBMS.

NoSQL data stores, other than relational DBMSs, are not primarily based on tables. Furthermore, the majority of NoSQL systems do not support SQL as a query language

and often only provide limited querying capabilities. Nevertheless, NoSQL data stores are very efficient when querying data records by key.

BASE Generally, NoSQL data stores do not support the ACID properties. They use a more relaxed concept, which is known as BASE. This acronym stands for **basically available**, **soft state** and **eventual consistency**. (Compared and discussed in Section 2.3.) Due to the use of multiple servers and redundancy NoSQL data stores provide high availability and can handle failures of server instances. Basically available means that the system is available at any time whenever it is accessed, even if parts of the system are unavailable. Soft state indicates that the state of the system may change over time, although no further inputs are made. This concept highly correlates with the last BASE property eventual consistency. Traditional DBMSs use a strong consistency model like in ACID. NoSQL data stores follow a looser approach of consistency: eventual consistency. Changes in a NoSQL data store propagate through the distributed system visible to all users. During this time span, the corresponding records are not locked for other users and may cause an inconsistent state of the data store.

[PA16, MH13, Pok13]

2.1.1 Data model

A traditional database stores information using a relational data model, which consists of tables and relations between them. These tables are composed of rows (tuples). In the NoSQL domain, a row is a limited data structure, because it cannot contain other rows or complex structures. Nesting rows or lists of rows in other rows are in a relational database not possible or would violate normalization constraints. However, object-relational DBMSs systems such as Oracle [ora18] or PostgreSQL [pos18] also offer the possibility of mapping more complex data structures.

NoSQL data stores take a different approach, due to the need for flexibility in the data model. In practice, it is often more useful to operate on a more complex data structure than a row. This complex data structure should be able to contain multiple properties, lists and other nested data structures and is called **aggregation**. This notation of aggregation is not directly correlated to notation in used traditional data modelling. Aggregations are often used as a unit for data manipulation in the majority of NoSQL data stores. It also emphasizes a more structural approach for developers when manipulating these complex data structures in scope.

The following example (Figure 2.1) shows a simple entity-relationship model with the following entities: `Company`, `Employee` and `Address`. This model will be used to create a relational data model and a NoSQL data model to compare the different approaches.

The tables illustrated in Figure 2.2 represent a data model for a relational data store.

The NoSQL data model is represented in JSON (JavaScript Object Notation). JSON is a lightweight data format, which is human-readable and often used in NoSQL data

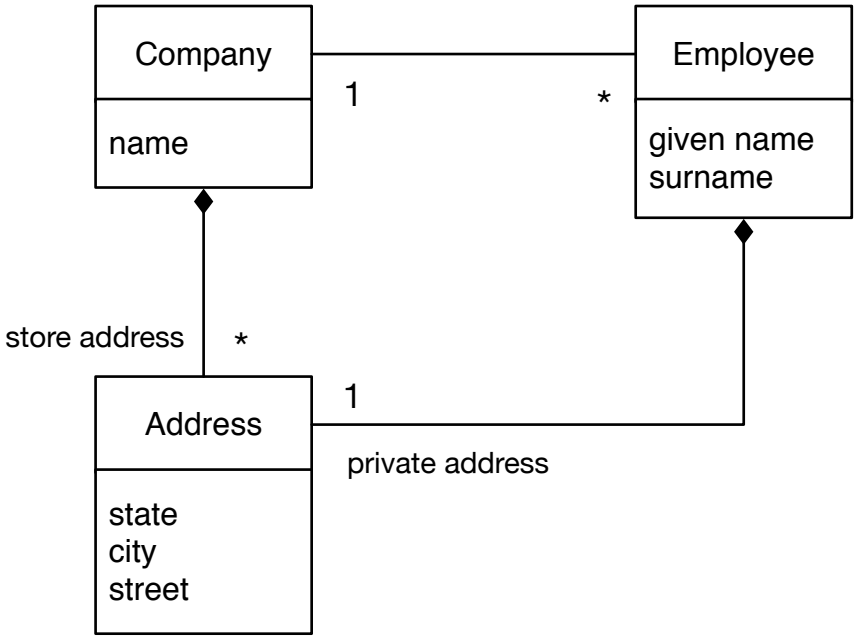

Figure 2.1: Example: ER-diagram

stores. The composition syntax (black diamond) used in the entity-relationship model represents the aggregation structure used in the NoSQL model.

In this example, a `Company` has multiple store addresses and an `Employee` has a private address. The aggregation structure of these relations leads to nested data records as shown in Listing 2.1. The relation between the entities `Company` and `Employee` is just a simple relation and does not lead to an aggregated data structure.

Like in a relational data model records can be referenced by keys (identifiers). In the relational model (Figure 2.2) an `Employee` contains an attribute `CompanyId` to reference the `Company` the employee works in. In the NoSQL data model it is vice versa, the `Company` contains a list of `Employees` which work in this `Company`. But in general, there is no strict rule which entity has to contain the references. The NoSQL data model would still be valid, if it were the other way round or if both records contained the references to each other.

Another difference is that in the NoSQL data model the same logical address appears multiple times. The `Addresses` are treated as simple value structures and are not referenced by an identifier. In a relational data store, the addresses would be referenced by an identifier to avoid multiple instances. In an aggregated data structure records can be copied in place if it fits the problem domain, but not all relations have to be modelled as an aggregated data structure. The relation between `Company` and `Employee` is a relation between aggregates, these records are not nested within each other.

Company

Id	Name
1	Big company

StoreAddress

CompanyId	AddressId
1	50
1	51

Employee

Id	First name	Last name	CompanyId	PrivateAddressId
101	Jane	Doe	1	55
102	John	Doe	1	55

Address

Id	State	City	Street
50	Austria	Vienna	Favoritenstraße
51	United States	New York	Broadway

Figure 2.2: Example: Relational data model

The boundaries of aggregated data structures can be chosen freely to fit the application's needs. For example, the relation between `Company` and `Employee` could also be modelled as an aggregated data structure. There is no single way to design the data model, the aggregate boundaries can be set according to the application and their use cases.

In general, there is no universal answer how to design such a data model, many factors are included in the design process. A major factor is the way the application is accessing and manipulating the data. If `Companies` are accessed with all it's `Employees`, then a single aggregated data structure per `Company` would be preferred. On the other hand, if the application is focused on accessing and manipulating a single `Employee` a more separated data model should be considered.

In a traditional relational data model, there is no semantics to express whether a relation is an aggregation or not. This information, which relation can be interpreted as an aggregation, is from the perspective of the distributed system of high importance because it can be used by the system to distribute or combine certain junks of data over multiple nodes. For example, if `Companies` and `Employees` are distributed over two nodes in a distributed environment, a query over a company containing its employees would involve both nodes. If the system has the information, that the relation between `Company` and

Listing 2.1: Example: NoSQL data model using JSON

```
{
  "companies": [
    {
      "id": 1,
      "name": "Big company",
      "storeAddresses": [
        {
          "state": "Austria",
          "city": "Vienna",
          "street": "Karlsplatz"
        },
        {
          "state": "United States",
          "city": "New York",
          "street": "Broadway"
        }
      ],
      "employees": [101, 102]
    }
  ],
  "employees": [
    {
      "id": 101,
      "first name": "Jane",
      "last name": "Doe",
      "privateAddress": {
        "state": "Austria",
        "city": "Vienna",
        "street": "Karlsplatz"
      }
    },
    {
      "id": 102,
      "first name": "John",
      "last name": "Doe",
      "privateAddress": {
        "state": "Austria",
        "city": "Vienna",
        "street": "Karlsplatz"
      }
    }
  ]
}
```

`Employee` is an aggregation, it would not split the data over multiple nodes. In this scenario the entities would reside on a single node and the system would only have to access this node for the query.

The information about aggregation between entities is not a property which can be inferred from the data model. It has to be provided by the developer and in most cases, this information depends on how applications will process the data. Furthermore, if the data is being used by many applications and for different scenarios, it is very problematic and difficult to set up aggregation structures which satisfy all parties. Already if a data store is used for operational and analytical purposes the aggregation structures for these use cases can be conflicting.

Data stores can be split into aggregation ignorant data stores and data stores which support aggregation. Most NoSQL data stores support the concept of aggregation except for graph data stores. Aggregation ignorant data stores have the ability to provide data in a generic and universal way because they are not constrained to aggregations. This way the data stores can serve a broad array of applications. This concept fits the traditional database because it is often used as a general-purpose data store.

Data stores which support the concept of aggregation, like most NoSQL data stores, benefit in multiple ways by using this data model. One key aspect of NoSQL data stores is a flexible data model, this suits the concept of aggregation very well because the aggregated data structure can be adapted to fit the needs of the data store. As mentioned above aggregation facilitates the usage of distributed systems, because aggregation enables the system to decide which records belong together and which can be easily distributed across different nodes, this also fits the characteristics of NoSQL systems. Furthermore, NoSQL data stores are in most cases used and built for a very specific use case, so they are not used as a general-purpose data store. This fact offsets the disadvantage of aggregation for NoSQL data stores.

[SF12, SH18]

2.1.2 Classification of NoSQL data stores

NoSQL data stores are built on top of different data structures and data models. They can be divided into the following four categories based on the underlying data model: Key-value stores, column stores, document stores, graph databases.

Key-value stores

Key-value stores have the simplest data model compared to other NoSQL systems. As the name suggests, a key-value store holds a collection of key-value pairs. Generally, a key-value store is able to store any binary data, but the key is in most cases limited to a certain size. The key is unique in the collection and in most cases the only way to retrieve data from the store. Some key-value stores offer additional options to query data by value. The querying possibilities are very restricted because of the lake of information

of the stored binary data. In addition to the binary format, most key-value stores also provide support for other data types.

The simple nature of a key-value data store allows the system to scale rather easily and distribute data across multiple servers. The underlying flexibility of the system also comes with drawbacks. Key-value stores mostly provide basic operations to access and modify the stored data. Because of the high abstraction, the system has less information about the stored data than for example a RDBMS, which uses a schema and defines data types for each field. This flexibility shifts more tasks and responsibility to the application developers using the key-value store. If a value is associated with a data type the system can provide higher level support for the developer e.g. conflict resolution and secondary indices.

In most cases, key-value stores are not able to handle complex applications, because they do not support relations. Nevertheless, they are used in many scenarios. Frequently they are used for simple tasks, which require fast response times such as caching or session management. Key-value stores are used to cache whole websites or JSON-documents. This is where key-value stores shine. They are designed for high availability and performance. Also, many modern database systems use a key-value data store as a foundation and build their system on top of it. For example, CockroachDB [coc18a] a NewSQL database uses RocksDB [roc18] as an embedded key-value store at its core to persist data to disk.

Many key-value stores are based on Amazon's Dynamo [DHJ+07]. After the release of the paper, many systems implemented this architectural approach and more or less started the era of NoSQL. Many systems like Riak [ria18] and LinkedIn's Voldemort [vol18a] are nearly exact implementations of Dynamo's architecture. DynamoDB [dyn18] is still very popular today. Various other systems adopted only parts of Dynamo's design or followed a completely different approach like Redis [red18a].

Column stores

Before NoSQL data stores became popular some database systems already used column-orientated data models. In comparison to today's NoSQL column stores, these database systems are more related to traditional RDBMSs. They use a relational data model and support the standard SQL interface. Most important, these column databases store the values not row-wise but column-wise. Both groups of systems are often called column stores or column-oriented data stores. The naming convention is often the reason for confusion and misplaced expectations.

The relational column-oriented databases originate in the data warehouse and analytics domain. Data warehouses are rarely interested in processing values of individual rows. They are built to aggregate values of a single column across all rows because this way statistical conclusions about the data set can be drawn. Conventional RDBMS store data row by row, so the access of a single column across all rows requires an iteration over all corresponding rows, which results in a high workload for the system. Column-oriented

databases solve this issue by storing column values physically next to each other. This increases the read performance for this specific use case.

From now on, we will use the term column store in the context of NoSQL. This kind of NoSQL data stores can also be referred to as column family databases, wide column stores or columnar database. Once again the naming in this domain is often the reason for misunderstandings and confusion.

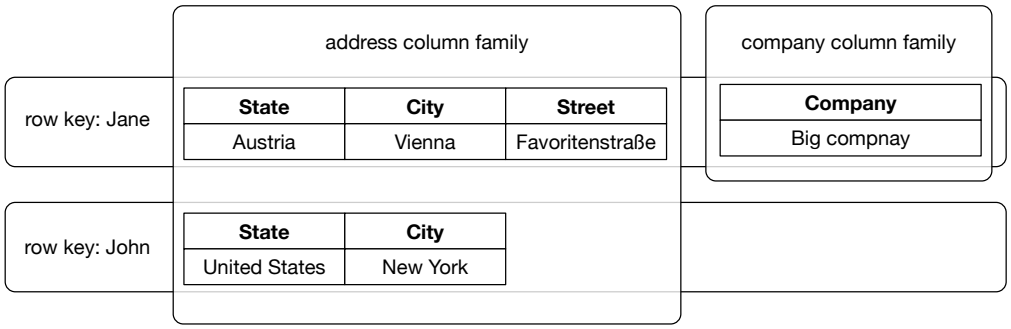

Figure 2.3: Column family example

Column stores use the concept of a keyspace. A keyspace is the coarsest element in the data model. It contains a set of column families. Column families contain multiple rows and can be compared to tables in a relational data model. Unlike relational rows, each row in a column family can consist of a different number of columns and has a unique identifier called row key. Also, the columns do not have to be identical to other rows in terms of column names or data types. So each column is only associated with its row and does not span over all rows in its column family. Furthermore, a row can be linked to multiple column families as shown in Figure 2.3. This Figure also shows the grouping of columns to column families.

A column is commonly a triple consisting of a name, a value, and a timestamp. (The timestamp was neglected in these examples.) Column stores use the timestamp to resolve write conflicts and store multiple versions of a value in the system. Columns are dynamic elements in a column store, they are not defined within a schema, so new columns can be created with every insertion of a new value. Also, a big advantage compared to relational columns is, that empty columns do not take up space. The data model of NoSQL column stores can be seen as a multi-dimensional map. Essentially, data is in most cases not stored by column, like in traditional column stores mentioned before, but by row. A column family is stored in a row-based fashion.

Besides, column-families can be used as so-called wide column families. This approach uses the name of a column as a place to store additional information. This means that the columns can be seen as a collection of key-value pairs within a row. Figure 2.4 shows an example of a wide column family. The columns in the co-worker's column family are key-value pairs of the name and the department the person works in. This example also

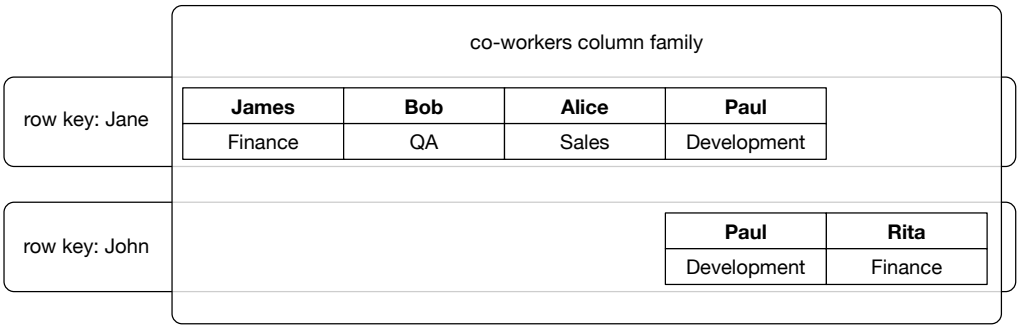

Figure 2.4: Wide column family example

shows that empty columns do not take up disk space. Essentially, a row can be seen as a collection of key-value pairs.

Most column stores are inspired by Google's BigTable [CDG+08]. BigTable is a high-performance storage system and pioneer in its domain. Today's most popular column stores are Cassandra [cas18b] and HBase [hba18].

Document stores

As the name suggests, a document store stores its data in structured documents. The format of the document varies between data stores. The most commonly used document formats are JSON and XML. The definition of document stores does only include the storage model. It makes no statement about other characteristics like scalability or ACID compliance. Document databases are one of the most prominent representatives of NoSQL databases. A reason for their success is the impedance mismatch. The impedance mismatch refers to the differences in structure between a relational database schema and an object-orientated model. Documents map very closely, and in some cases directly, to objects. This makes it easier for the developer to integrate a document store in an object-oriented application. Another reason why document stores are so popular is the good interaction with current web technologies. Especially JSON document stores like MongoDB work well with JavaScript-based web applications.

Before JSON document stores became popular, document stores used the XML format as their basis. XML document stores or databases are today not as significant as JSON document stores, but many companies still use it, because they acquired a lot of XML documents over the years. XML is also used for many web protocols, most importantly Simple Object Access Protocol (SOAP). XML also offers a huge environment of tools and standards for querying (XQuery, XPath), transformation (XSLT) and schema definition (XML schema). XML document stores implement these standards to provide the developers with known technologies. As of today, many popular DBMSs implement the XML SQL extension and displace XML document stores.

In the age of Big Data, XML documents became unpopular. The amount of data made XML documents impractical. They use a lot of disk space compared to the contained information and are resource expensive to parse. JSON is a better fit for the current challenges regarding web technologies. It is lightweight, flexible and well integrated into modern development frameworks. All these factors made JSON document stores so popular today.

There is no definition or specification stating what requirements a JSON document store has to fulfil. They only have to store JSON documents, but in most cases, they have a similar architecture. The basic unit is typically a document, which can be compared to a row in a relational database. The second primary data structure is a collection. It can contain multiple documents and corresponds roughly to table in a relational database. Documents in the same collection can be of different shapes. The concept of aggregation, as described before in Section 2.1.1, is deeply rooted in document stores. This is also why collections are not used in the same way as relational tables. They are used less frequently because document stores typically use embedded documents instead of separate collections.

Document stores represent an important part of the current NoSQL systems. Compared to plain key-value stores, document stores offer a richer data model and more complex features. The light-weight and flexible JSON format and its integration into current web technologies are the main reasons for their success.

The most popular representatives of document stores are MongoDB [mon18a], Couchbase [cou18a] and CouchDB [cou18b].

Graph data stores

RDBMSs and most NoSQL systems are mainly built to store elements or values. They are also capable of handling relations, but the focus lies on the stored elements. The focus of graph data stores is not on the elements but on the relationships between the elements. Many domains face the challenge of this kind of data structures. The most prominent example is a social network with a multitude of relations between each person. Other NoSQL systems and RDBMSs are also able to model this kind of data structures, but these systems usually have performance issues when working with this degree of relations. Especially other NoSQL systems cannot deal with this kind of data, because they do not have a suitable query language. Graph data stores are optimized for this kind of problems. Especially, they provide a rich query language which supports the developer in graph related queries.

Graph data stores are based on a strong theoretical foundation, the graph theory. The Graph theory is not a novel problem, many sciences outside of computer science like sociology, economics, and mathematics, also use graph theory. These are also the domains in which many problems scenarios are well fitted for graph data stores.

A graph contains the following elements:

- Nodes (or vertices): The node is the fundamental unit in a graph. In a social network, a node would represent a person. As a person can have multiple properties, like name and address, a node can also have multiple properties. These properties allow the system to associate additional information with a node.
- Relationships (or edges): A relationship is a connection between two nodes. Relationships are as well able to store properties to associate further information.

As mentioned before, a graph could also be modelled in a RDBMS [UO18]. The problems RDBMSs face when working with graph data are performance issues. Particularly, the graph traversal, which is needed for many graph related problems, degrades in performance as the depth of the graph increases. Graph data stores use an approach called index-free adjacency. This means that each node knows the physical address of all adjacent nodes. This approach allows graph data stores to efficiently traverse massive graphs. But this approach comes with a drawback regarding the distribution of the graph data set across multiple servers. If an adjacent node is located on a different server, the time-consuming traversal across the network eliminates the benefit of the graph model.

Besides pure graph data stores, graph compute engines offer many features of a graph data store without the need to use a native graph model. These systems can be used on top of RDBMSs and NoSQL data stores to provide graph processing algorithms. Graph compute engines do not perform as efficient as pure graph data stores, but they offer a rich set of graph algorithms without the limitations considering the distribution of graph data stores. Examples for graph compute engines are Apache Giraph [gir18] and GraphX [gra18].

Graph data stores use a very special data model for specific use cases and are probably the most unique class of NoSQL systems. The goal of graph data stores is not to replace all database systems on the market, they are not general purpose database systems, but they are optimized for their needs. The most popular graph data store is Neo4j [neo18a].

2.2 NewSQL

With the publication of *"The end of an architectural era:(it's time for a complete rewrite)."* [SMA+07] Stonebraker and his research team started questioning the architecture of traditional DBMS. In particular, the paper gets to the bottom of the design decisions and assumptions, which were made during the development of these systems. They performed various tests with traditional DBMSs and modern database workloads. It has been shown, that the underlying design decisions and assumptions, based on former hardware, are outdated considering the current workload and needs. The proposals made by Stonebraker influenced the design of many new database systems.

NewSQL describes a class of modern DBMSs. In contrast to NoSQL, there are not as many systems, which fit in the class of NewSQL. The main focus of this systems is to cope with the current challenges of scalable and distributed systems. The term

NewSQL was first used by Matthew Aslett in *"How will the database incumbents respond to NoSQL and NewSQL?"* [Asl11]. NewSQL systems seek to provide the same horizontal scalability as NoSQL systems without sacrificing the ACID properties of a traditional DBMS. Furthermore, a key characteristic of NewSQL systems is the support of the standard SQL interface. Compared to the NoSQL domain, which comprises various data models, all NewSQL systems use a relational data model like traditional DBMS. This allows developers to query, access and modify the database using the language and tools they are used to.

There is no universal definition for the class of NewSQL systems, but they share the following characteristics:

- **Horizontal scalability** - NewSQL data stores seek to provide the same horizontal scalability and read-write performance as NoSQL data stores.
- **Standard SQL interface** – NewSQL systems support the standard SQL instead of a proprietary interface, already existing tools can be used.
- **Relational model** – As traditional DBMS, NewSQL systems also use the long-established relational model.
- **Support of ACID properties** – NewSQL systems still maintain ACID guarantees, while providing horizontal scalability.
- **Shared-nothing architecture** – A shared-nothing architecture refers to a distributed-computing architecture, which consists of independent nodes with no single point of failure. The nodes do not share particular resources like disk space or memory.
- **Lock-free concurrency control mechanism** – A lock-free concurrency control mechanism enables the system to perform concurrent read operations without write operations blocking them.

NewSQL systems cannot be classified by the data model like NoSQL, because all of them use the relational data model. Nevertheless, the landscape of NewSQL systems can be grouped into three categories:

- New Architectures
- Transparent Sharding Middleware
- Database-as-a-Service

The category of **New Architectures** contains the most promising and interesting NewSQL systems. These systems are built from scratch and aim to fulfil all modern requirements of distributed relational DBMSs. Rather than building on top of an existing

DBMS or extending an already established DBMS, these systems are built from ground up without a legacy codebase. Other than traditional DBMSs these systems are designed for distributed applications and use cases. They operate in a shared-nothing environment. This includes replication, sharding, distributed querying, concurrency control and fault tolerance. These database systems were designed and implemented with these aspects in mind, therefore all components of the systems are optimized for the distributed use case.

Most of the systems implement their own communication protocol to efficiently share information between nodes. The communication between nodes is especially important when considering distributed queries and query optimization. Queries have to be re-written to send subqueries to the data rather than the data to the query. Due to new and different design approaches, the individual systems can differ greatly from each other in their architecture. Examples for these NewSQL systems are: CockroachDB [coc18a], VoltDB [vol18b], NuoDB [nuo18a], Clustrix [clu18] and Google Spanner [goo18].

NewSQL systems in the category of **Transparent Sharding Middleware** make use of already established DBMSs. Each node of the system runs one instance of a traditional DBMS and manages a part of the whole database. These standalone instances are not meant to be accessed directly by a top-level application. Instead, a transparent sharding middleware manages the access to these instances. The middleware is responsible for data distribution, queries, transactions and all other relevant features.

For example, if an application executes a query, then the middleware transparently rewrites the query to multiple subqueries and forwards them to the particular DBMSs underneath the middleware. Each DBMS executes its subquery and reports the results back to the middleware. After all results have been returned the middleware can combine the individual results to one, which corresponds to the original query and reports this result back to the application. The middleware is also responsible for transaction coordination, data placement, sharding and replication across all nodes. The advantage of this approach is that the middleware can be installed on an existing DBMS with little effort and in most cases, there is no need to adapt the applications using the database. Examples of a transparent sharding middleware are: AgilData Scalable Cluster [agi18], MariaDB MaxScale [mar18] and ScaleArc [sca18].

The third category of NewSQL systems are **Database-as-a-Service** (DBaaS) systems. The whole database stack is managed by the provider of the service. The customers of these services do not have to manage their own server hardware or infrastructure, they only have to select a fitting web service depending on their needs. Therefore, the provider of the service is responsible for physical configuration, replication and fault tolerance. The customer is usually provided with a dashboard or API to configure and manage the system. Cloud computing companies like Amazon, Google and Microsoft offer this kind of services. Examples for DBaaS systems are: Amazon Aurora [aur18] and ClearDB [cle18].

Transaction model Traditional DBMS implement the well-established transaction model to satisfy the high requirements of data consistency and integrity. NewSQL

systems also follow this approach in contrast to NoSQL systems. This is also a key advantage over NoSQL systems because in many cases developers using a NoSQL data store re-implemented those features on the application side.

A transaction represents a sequence of reading and writing operations. The execution of a transaction preserves the consistency and integrity of the database if it runs in isolation from other transactions. A transaction in this context has the following properties:

- **Atomicity** – This property requires that the entire transaction or nothing is executed on the DBMS. If one fragment of the transaction fails, the whole transaction fails and the state of the database is unchanged. This guarantee has to hold under all circumstances. For instance, network errors, power failures or crashes of the whole system have to be handled.

- **Consistency** – DBMSs contain a set of rules and constraints regarding the stored data. If all rules are fulfilled the system is in a valid state. Consistency ensures that the system is at any point in time in a valid state. If a transaction modifies the data in a way so that the resulting data violates a rule or constraint, the system would end up in an invalid state and therefore has to undo all changes of the faulty transaction.

- **Isolation** – The Isolation property handles the execution of concurrent transactions. It ensures that each transaction does not see changes made by other concurrent transactions. This guarantees that all transactions are isolated from each other. Depending on the isolation level or concurrency control method the serialization of concurrent transitions varies.

- **Durability** – After a transaction has been successfully completed, the changes are persisted and the stored data is available. So Durability ensures that written data is permanently stored even in the case of a system failure or power outage.

2.2.1 Multi-version concurrency control (MVCC)

Multi-version concurrency control [BG83] is a popular approach for NewSQL systems to handle concurrency control and maintain consistency in a distributed environment. This approach is non-standardized and each DBMS implements it slightly differently. MVCC is an alternative to the traditional read/write lock approach. The read/write lock approach has a fundamental problem. Writing transactions block reading transactions and reading transactions block writing transactions. This is, in fact, a well-known property of this approach, but it can lead to critical performance issues, because of the additional latency while waiting to acquire a lock.

MVCC solves this problem by eliminating this single resource that has to be locked. As the name suggests, each record in a DBMS which implements MVCC consist of multiple simultaneous copies or versions. In order to differentiate these versions, each version is associated with a timestamp. Updates on a record create a new version with a timestamp representing the creation time. This approach allows the DBMS to decide which version a transaction should read. The problem that the read/write lock approach faces does not apply to MVCC. Reading transactions do not block writing transactions, because writing transactions just add new versions and the reading transactions can still access all prior versions. Also, writing transactions do not block reading transactions, because the DBMS can decide which version should be read by the reading transactions. This can be used to implement snapshots of the database. For example, a transaction could only read values, which were visible at the time the transaction started. All updates from other transactions would be logically after that particular transaction and would not affect it. Transactions writing to a record contest over adding the newest version on top of the stack. This means that only if two or more transactions write to the same record, then the system has to resolve a conflict.

MVCC can be used to implement various isolation levels like in traditional DBMSs. Generally, isolation levels are used to control the level of consistency. More loose consistency levels usually provide better performance and vice versa. Most NewSQL systems using MVCC do not implement the same isolation levels as traditional DBMS, but the majority implements the semantics of the highest consistency level Serializable. Due to the use of time-stamped versions, the system can easily decide which version should be visible for which transaction. For example, at a low transaction level comparable to Read Uncommitted the system could allow the transaction to read uncommitted records, which could result in dirty reads if uncommitted records would be rolled back. On the other hand, the stricter consistency level Serializable can be achieved by restricting the visibility of the transaction to the versions, which already have been committed at the start of the transaction. This means that a transaction reads a snapshot of the database at the begin of the transaction.

As mentioned before, there is only one situation in which the system has to resolve a conflict and this is when a transaction wants to update a record, which is being updated by another transaction. At this point there are two options a system can choose from. Firstly, the system can cancel and usually restart the transaction. Secondly, the system can make the updating transaction wait until the other transaction completes and make a decision based on the outcome of the transaction. The waiting transaction would only succeed in its update if the preceding transaction rolled back. Alternatively, the waiting transaction gets cancelled.

The following examples will explain how concurrent read and write transactions are handled using MVCC. They will illustrate a single table with three records (R10, R11, R12), multiple versions of these records and how concurrent transactions affect the data.

The first example (Figure 2.5) contains two concurrent transactions (T1 and T2). T1 reads each record of the table once and concurrently T2 updates the record R12 before

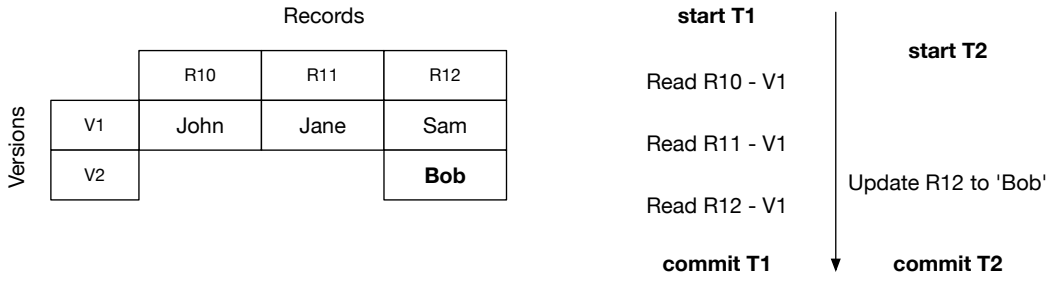

Figure 2.5: MVCC: Example 1

T1 reads the value of R12. In a lock-based system, T1 would have to lock each record or the whole table to consistently read the data from the table and T2 has to wait until T1 is finished in order to update the table. Considering a system using MVCC the writing transaction T2 would not be blocked by the reading transaction T1. Also, T1 would read the record R12 from version V1, because V2 was not committed at the time T1 started (considering a snapshot isolation level).

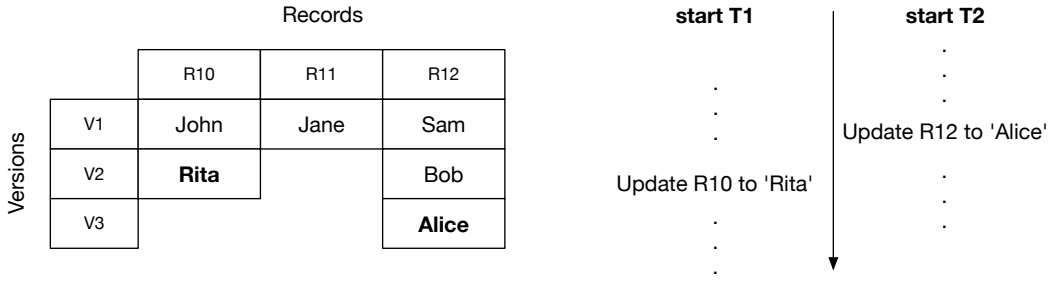

Figure 2.6: MVCC: Example 2

The second example (Figure 2.6) covers two transactions (T1 and T2) which write to distinct records. The transactions start at the same time. T1 updates R10 and T2 updates R12. Both transactions can write to the record and add a new version because the particular record is currently not updated by another transaction. No transaction is blocked and both transactions can continue. In a lock-based system the result would be the same.

In the third example, two concurrent transactions (T1 and T2) try to write to the same record, which results in an update conflict. Both T1 and T2 attempt to update record R12, but only one transaction will update the record first. Assuming that T2 updates the record first, which will add a new version to R10. While T2 is uncommitted the newly created version is also in an uncommitted state. After T2 added a new version, T1 also tries to update R10, but T1 detects that a transaction is currently updating this record indicated by the uncommitted state of the newest version.

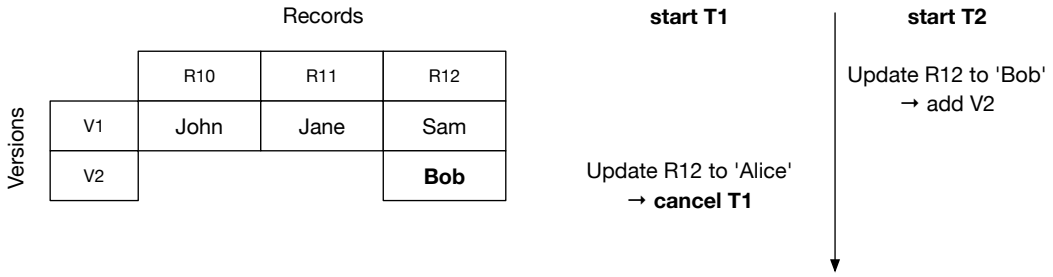

Figure 2.7: MVCC: Example 3 - Immediate cancel

The further behaviour depends on the particular system and the configuration of the isolation level. Generally, there are two options. The system can immediately cancel T1 after it detects that T1 attempts to update a record with an uncommitted version as illustrated in Figure 2.7.

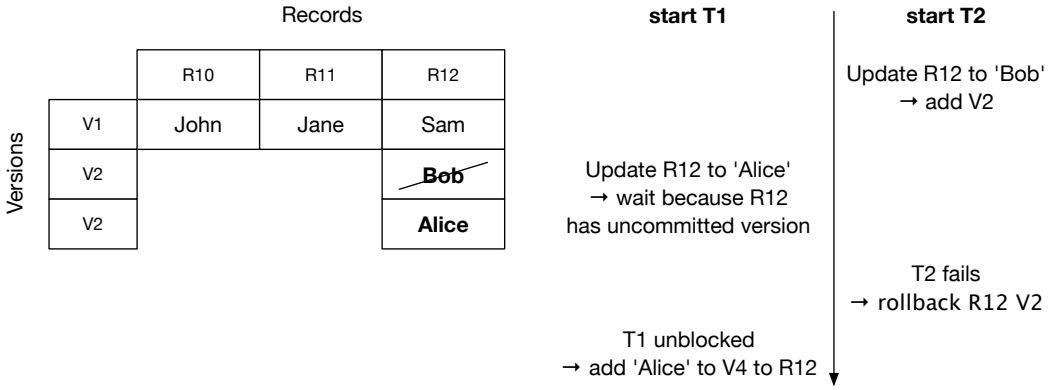

Figure 2.8: MVCC: Example 3 - Waiting - T2 fails

The second option would be to interrupt and pause T1 until T2 finishes. This option speculates that T2 will be cancelled and rolled back. Then the uncommitted version of T2 would become obsolete and T1 could update R12 (Figure 2.8). In the case T2 does not fail and commits successfully, T1 would fail anyway and would have waited unnecessarily (Figure 2.9). Both of the described options are valid approaches and most systems offer a configuration option, so the developer can choose an option depending on the requirements of the application.

2.3 CAP theorem

A common strategy used by many data stores is to distribute data sets across multiple server nodes to achieve parallel data processing. Additionally, replications of these nodes

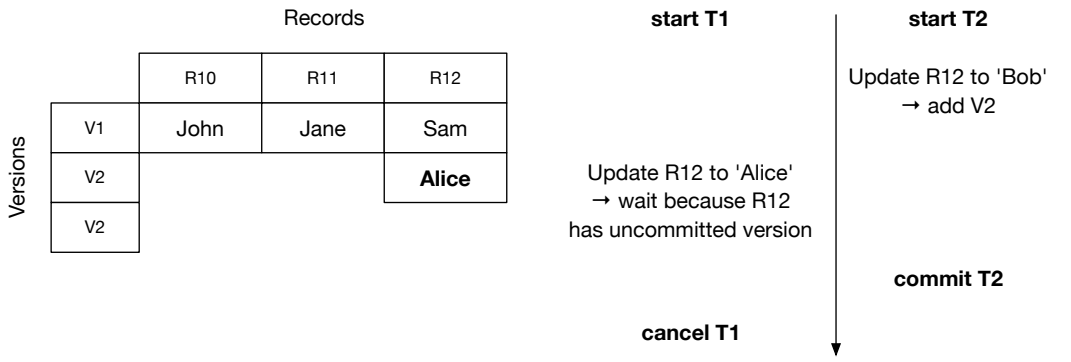

Figure 2.9: MVCC: Example 3 - Waiting - T2 succeeds

are created in order to maintain high availability in case of server failures. Nevertheless, these systems underlie restrictions, stated in the CAP Theorem [Bro09]. The CAP Theorem says that in a distributed system a maximum of two of the following three properties can be guaranteed at the same time:

- **Consistency** The system is consistent in the sense of the CAP theorem if all nodes of the distributed system read the same data at all times. As opposed to eventual consistency, the consistency term used here is also called immediate consistency or strong consistency. This property can be interpreted as equal to running the data store just on a single node.
- **Availability** The system is available if it responds to all requests, but without the guarantee that the response contains the most recent write. This means that every non-failing node responds to all read and write requests in a reasonable time. This requirement applies to all nodes, regardless of whether there is network partitioning or not.
- **Partition tolerance** Sets of nodes in a distributed system are partitioned if the communication between these sets is interrupted (all messages are lost). The system is partition tolerant if it still remains functional in the event of an interruption in communication between nodes in the network.

Note that the definition of the term consistency is different from the definition used in ACID. The consistency model in ACID relates to data integrity constraints. The DBMS ensures that all constraints are fulfilled. On the other hand, the consistency model in CAP promises that every replica in the distributed system corresponding to the same logical value has the same value at all times.

A system, which should fulfil all three characteristics, would therefore have to keep the disconnected nodes available, i.e. responding to requests and responding consistently if

the connection between nodes is interrupted. Now, if data changes on one node, obviously not all the remaining nodes can maintain consistency since some nodes are no longer reachable from the other nodes. The system can now either be in an inconsistent state or not respond, so it would have to give up either consistency or availability.

Depending on which of the properties are guaranteed, these three types of systems can be distinguished: CP (consistency and partition tolerance), AP (availability and partition tolerance) and CA (consistency and availability). Shown in Figure 2.10.

- **CP** systems respond to network partitioning by sacrificing full availability. The part of the network that can be kept consistent remains available, the rest not. Consistency and partition tolerance remain fulfilled and availability is not fulfilled. Once the connections between the nodes have been restored, the previously unavailable nodes are first restored to a consistent state before responding to requests.

- **AP** systems respond to network partitioning by sacrificing full consistency. All parts of the network respond to requests, but the answers are not necessarily consistent. After the network recovered from the partitioning, consistency is restored. Many AP systems including most NoSQL data stores systems go one step further and give up consistency during normal operation (without network partitioning) to increase their performance.

- **CA** systems try to ensure consistency and availability alike, but of course, they cannot prevent network partitioning from happening in a distributed system. Hence, they are forced to choose between consistency and availability in this case as well. Systems can only classify themselves as CA systems if network partitioning can be avoided, which is only possible in a non-distributed environment.

This classification is an idealized representation. In practice, particular systems do in most cases not fit in just a single category, it is of course also possible to create diverse forms between CP and AP systems. Also, many systems leave this decision to the administrator in form of a configuration option, so the system can be adjusted to certain needs.

The notion of network partitioning also has practical issues. First of all, there is no guarantee that all nodes of the system will react in the same way in case of a network partitioning. Some nodes may not be even aware of a connection issue. In addition, a connection failure is usually detected by exceeding timeouts, but not all nodes are necessarily affected equally.

Nevertheless, the CAP theorem describes an important issue, especially the need to balance these different requirements, most notably between consistency and availability.

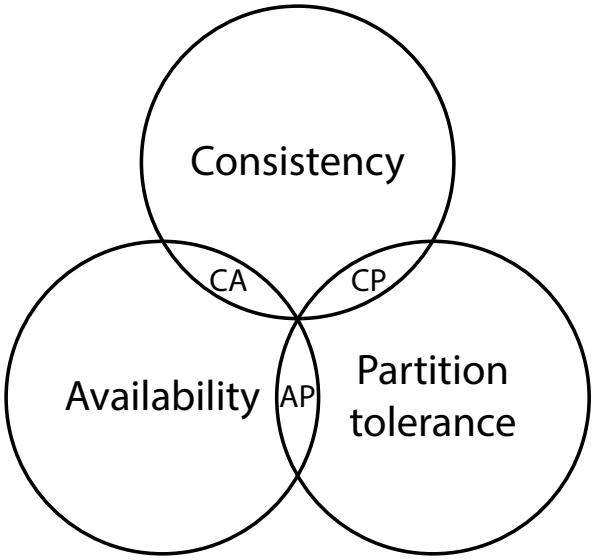

Figure 2.10: CAP diagram

As mentioned many systems are configurable in this respect and allow for a tailored solution for a given application.

[GHTC13, MBT16, MH13]

ACID and BASE The CAP theorem emphasis the fundamental decision between consistency and availability. This is also an elemental difference between NewSQL systems, which prefer consistency over availability, and NoSQL systems, which prefer availability over consistency. Consistency in NewSQL refers to the ACID properties of the transactional model. On the other hand, availability in NoSQL refers to the BASE concept. But in practice, this separation of characteristics is not as black and white as it seems. Many systems leave the database administrator the flexibility to mix different levels of availability and consistency. These combinations of consistency and availability usually result in a superior configuration. Table 2.1 summaries the differences between ACID and BASE. [Bre00]

ACID	**BASE**
NewSQL	NoSQL
Strong consistency	High availability
Prioritizes consistency	Weak consistency
Pessimistic	Optimistic
Planning and analysis	Best effort

Table 2.1: ACID vs. BASE

CHAPTER 3

Comparison

This chapter contains the comparison of the NoSQL and NewSQL databases. For NoSQL one representative of each NoSQL class was selected:

- Redis (Key-value store)
- Cassandra (Column store)
- MongoDB (Document store)
- Neo4j (Graph data store)

The selected NewSQL database systems:

- VoltDB
- CockroachDB
- NuoDB

These NewSQL databases are all three classified as New Architectures. The category of New Architectures contains the most promising and interesting NewSQL systems. These systems are built from scratch and aim to fulfil all modern requirements of distributed relational DBMSs.

The selected data stores are compared by the following key characteristics:

- Architecture and general principles
- Sharding

- Replication
- Querying
- Concurrency control

3.1 Architecture and general principles

Each data store follows certain design principles and characteristics. The combination of these principles gives each system an individual touch and influences the architectural decisions. The following section highlights these fundamentals to provide an overview of the systems. The subsequent sections will then elaborate the chosen implementations and techniques in detail.

3.1.1 Redis

Redis [red18a] is one of the most popular key-value stores. It is a open source, in-memory database and stores a mapping of keys to values. Also, Redis supports replication and sharding, this enables the system to scale read performance.

Redis does not satisfy ACID requirements instead Redis architecture uses the BASE approach, typical for NoSQL systems. In the context of the CAP Theorem, Redis is categorized in AP (availability and partition tolerance). This means if a Redis cluster recognizes a network partitioning, the focus lies on availability and not consistency. In this scenario, Redis takes the risk of serving inconsistent data, instead of not responding.

The in-memory design is one reason for Redis high-performance [MPS17], but it also offers disk-persistence. The in-memory data can automatically be saved to disk. Redis offers two modes of persistence for storing in-memory data on disk. The first mode persists data on disk on a timely basis when certain conditions are met. The other mode uses a log file, which writes all commands that modify the data to disk. This file can be used to restore the data in the event of a power failure.

Data stored in Redis does not follow a schema, so Redis data model is schema-less. A schema-less approach allows for flexibility, but also shifts more responsibility to the developers, designing and maintaining the data store.

Data in Redis can be stored using five data structures (strings, lists, hashes, sets and sorted sets) shown in Figure 3.1. Many key-value stores only support a simple string value. These features enable Redis to be used as a primary database or as a secondary database to support other data stores.

The `String` data type in Redis is comparable to `String` data types in other systems. `String` values can be stored, fetched and deleted given a key. Moreover, Redis provides a few commands to operate only on a part of the `String`.
`Lists` in Redis are implemented as a linked-list of `Strings` and support a large variety of data manipulation and retrieval commands, e.g. push/pop on the left or right side of

the list, insert and delete for a given index and trim to a certain size.
The `Hash` data type contains a mapping of keys to values. This data type can be seen as scaled-down key-value store itself, which can store `Strings` as values.
The simple `Set` data type is implemented as a hash table with no associated values, therefore the values of the Set are unordered and unique. Redis offers common set operations known from set theory including intersection, union and difference. In addition, Redis offers a `Zset` data type which represents an ordered set. As the simple `Set` the `Zset` is also implemented as a hash table, but the associated values are floating point numbers. These floating point numbers are used to determine the order of the elements in the set.

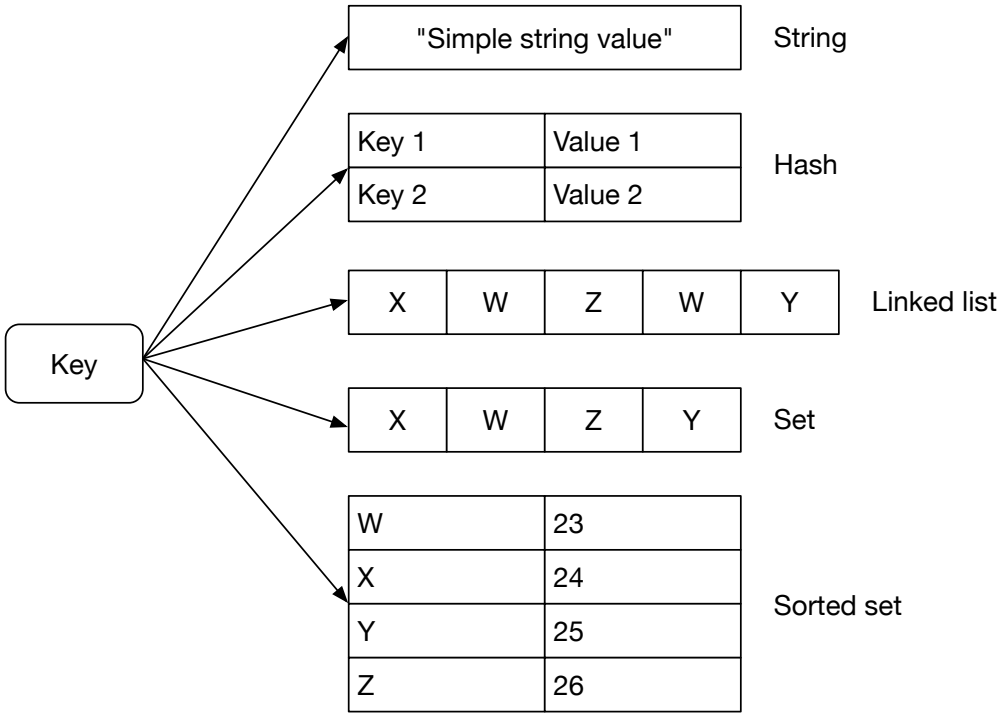

Figure 3.1: Redis: data model

Key-value stores are one of the best data stores regarding performance, because of their rather simple design and in-memory architecture. To even increase read-performance and guarantee availability in case of a server failure Redis supports master-slave replication. Slave servers connect to a designated master server and initially replicate the data stored on the master server. The master server then propagates every operation to the connected slave servers. Clients performing read operations on the data can now be distributed over the master server and associated slave servers to increase read performance.

Despite Redis is an in-memory data store it supports two methods to persist data on disk. First Redis is capable of dumping data to disk when certain conditions are met,

e.g. after a defined number of writes were executed. This database dump can also be executed by using a command. An append-only file is used by the second method. This file logs operations executed by Redis, it can be configured to log in different granularity from never, once a second or after execution of each operation.

[Car13]

3.1.2 Cassandra

Cassandra [cas18b] is one of the most successful NoSQL data stores. Cassandra is designed as a scalable, fault-tolerant, distributed system able to handle a huge amount of data and it is open source. Due to its architectural features, Cassandra is often used in Big Data projects, but can also be used for complex web applications in cooperation with an application server. Cassandra is a distributed database management system designed to manage very large volumes of semi-structured data and is optimized for high scalability and reliability in distributed systems.

Cassandra was developed by engineers at Facebook to overcome the inbox search problem. Facebook's social network contains millions of users distributed over thousands of servers worldwide. The inbox search feature of Facebook enables to user to search through their Facebook inbox. The problem which arises is that the system has to handle billions of writes per day and scale even further.

One of the core principles in Cassandra is distribution, which means that Cassandra is able to run on multiple servers while appearing as a single system. It is also possible to run Cassandra on a single server, but a single server setup would not utilize Cassandra's strengths. Unlike Cassandra, many systems have a designated master node, which handles the communication between Cassandra and the accessing clients and the organize the other nodes, called slaves. Cassandra is not only distributed, it is also decentralized, which means that all nodes in a Cassandra cluster are equal and there is no single point of failure. The nodes use a peer-to-peer protocol to communicate and organize the cluster decentralized.

The distributed and decentralized architecture enables Cassandra to scale horizontally. Cassandra also automates the process of adding and removing nodes. Cassandra calls this feature elastic scalability. There is no need to restart the process, rebalance the data or change the application queries. This is done automatically and transparent to the users. In distributed systems, the CAP theorem plays always an important role. Similar to other NoSQL data stores Cassandra is placed as AP (availability and partition tolerance).

In the default configuration, Cassandra trades consistency for availability, but Cassandra provides a feature called tunable consistency. It allows the administrator to configure a weak, causal or strict consistency model. All of these modes have different trade-offs between consistency and availability. Depending on the situation and requirements a more optimistic configuration provides high availability but weak consistency and if a

high level of consistency is necessary Cassandra can trade availability for consistency. [LCG18]

Cassandra's data structure contains at the utmost level a so-called `cluster`. Cassandra performs at its best utilizing multiple nodes combined in a cluster. If a particular system is designed to run Cassandra only on a single machine, the choice of the data store should be reconsidered. A cluster is also called a ring because the nodes within a cluster are arranged to form a ring structure. Clusters also take care of node replication. They can be configured to use a certain number of nodes as replicas if a machine goes down a replica can step in until the machine is back up.

The next level in Cassandra's data structure is a `keyspace`. A cluster can contain multiple keyspaces. Compared to a relational DBMS a keyspace is analogous to a relational database. As a relational database contains tables, a keyspace contains column families.

A `column family` contains an ordered collection of rows and each row contains an ordered collection of columns (Figure 3.2). A row has a unique key (so-called `row key`), which uniquely identifies a row like a primary key. These are the only similarities to a relational database, a column family is not a relational table.

The definition of a column family contains only two properties. The first is the name of the column family and the second is a comparator, which defines the order of columns when they are returned. Depending on the application's needs columns can be added to any column family at any time. So the data model of Cassandra can be seen as multidimensional hash tables (keyspace - column family - row - column).

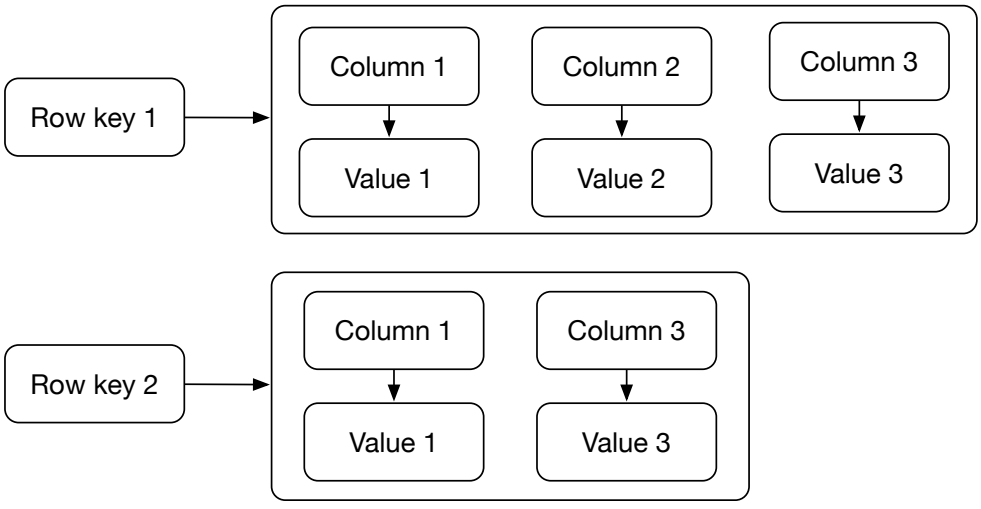

Figure 3.2: Cassandra: column family

A column family is also the unit of persistence. Each column family is stored in a separate file on disk, so related columns should be assigned to the same column family to optimize

storage operations. Beside column families, Cassandra offers super column families which allow column families to be nested in each other.

The actual data is stored in a `column`. A column is a triple containing a name, a value and a timestamp. Columns in Cassandra are not comparable to columns in relational tables. Columns are not defined in a schema, only column families are defined upfront. The data triple, which a column consists of, can be freely chosen for every column. The data types for the name and value are byte arrays.
In a relational table attributes with no value would be left blank and would wast memory. But in column family data stores like Cassandra blank values do not waste space, because a row contains only those columns which are needed.

[Hew10]

3.1.3 MongoDB

MongoDB [mon18a] is an open source database that uses a document-oriented data model. MongoDB is built with scalability and distribution in mind, so data stored in MongoDB can be distributed across multiple geographically distributed data centers. It also supports elastic scaling with no downtime and without the need to restart the system if a node joins or leaves the cluster. The system was designed to scale horizontally, so tasks, like rebalancing documents between nodes and routing queries to particular nodes, are done automatically.

MongoDB stores data in JSON-like documents without defining a schema. This means that stored documents do not have to conform to a fixed schema. Each document could have a different structure, but it is recommended to maintain a certain structure in the data model. A flexible data model allows for rapid prototyping and fast development.

In the context of the CAP theorem, MongoDB is placed as CP (consistency and partition tolerance), because the architectural design is focused more on consistency then availability. MongoDB uses a master-slave (primary-secondary) setup, where the clients can only execute read and write request on the primary. This indicates that MongoDB gives more importance to consistency than to availability and it is also a reason why MongoDB is considered as CP and not as AP in terms of the CAP theorem.

In-memory storage is one of the reasons why MongoDB provides high-performance, but in-memory storage also comes with the risk of losing data in case of a power failure. When MongoDB is set up as a cluster, MongoDB can utilize the benefits of in-memory and on-disk storage. So the primary uses in-memory storage, to efficiently process data and secondaries are configured to use on-disk storage to keep the data replicated on disk. [HEM18]

Unlike relational databases that use tables and rows, MongoDB relies on an architecture of collections and documents. On the top level of MongoDB's data structure stands a `database` similar to a database in relational DBMS. A database is a group of `collections`, like a relational table, is a group of rows. MongoDB is capable of

operating multiple databases at once. A database encapsulates data which usually belongs to one application. Like a relational DBMS, a MongoDB database manages its own permissions and its own files.

A `collection` in MongoDB is a group of documents. It can be compared to a table in a relational DBMS, but a collection does not have a fixed schema like a relational table. MongoDB offers the concept of dynamic schemas. This means that a collection can hold any kind of document, there is no restriction on the content of the document. It would be possible that each document in a collection has a different structure. And yet it is not recommended to store all different kinds of documents in the same collection.

First of all, to query one kind of a document from a collection with mixed documents, each document would have to be type checked to return the query result. If collections only contain one kind of document, it is much easier for developers to query the needed data and also increases the performance of such a query dramatically. The design of the data model offers a lot of flexibility like in most NoSQL data stores but misused it can quickly lead to complications.

Additionally, MongoDB offers the concept of `sub-collections`. Sub-collections are used for an organizational grouping of collections. For example the collection of employees named `company.employees` and the collection of departments named `company.departments` are sub-collections within the same namespace. This grouping does not indicate any relationship between the members, it only simplifies the administration of collections. Nevertheless, this concept is used by several tools provided by MongoDB to structure the data model. In order to uniquely identify a sub-collection named `company.employees` and a database named `organization`, these two strings can be joined using a `.` character.

Figure 3.3: MongoDB: embedded document

The core data structure of MongoDB is the `document`. A document is an ordered set of

key-value pairs and can be represented as JSON. Every document must have an `id` key, like a primary key. For document storage and data exchange, the data store uses the so-called BSON (Binary JSON) format, which provides a binary representation of JSON documents. The values of a document are not just unstructured data fields. The BSON specification offers multiple data types including arrays for the values. In addition, the value of a document can be an entire document, so-called embedded document shown in Figure 3.3. This enables documents to hold more rich and complex structures than simple key-value pairs.

[mon17, Cho13]

3.1.4 Neo4j

The Neo4j graph data store [neo18a] is a Java-based, scalable open source data store that supports ACID (for a single instance) and provides highly available clustering. It is an open source project with an enterprise version. Neo4j is a native graph database system. The database engine refers to data by linking data points to related data points, allowing for faster processing than relational joins or writing custom joins in a NoSQL database. The relationships between nodes are stored and processed as they actually occur. This results in a more responsive and agile system.

The high-availability (HA) cluster of Neo4j implements a master-slave architecture. Unlike other NoSQL systems, Neo4j allows reads and writes to master and slave nodes, but a write operation on a slave is not directly executed on the node. First, the operation is forwarded to the master and only if the transaction is successful, the operation is executed on the slave. In the context of the CAP theorem, Neo4j can be classified as an AP system, because the ACID properties cannot be satisfied in a distributed environment.

Furthermore, Neo4j keeps a full copy of the data on each node of the cluster. True data sharding is not available, but Neo4j implements a feature called cache sharding. Cache sharding manages data routing to nodes in the cluster. Requests are redirected to nodes, which have certain parts of the data cached in memory so that disk operations are kept at a minimum.

Neo4j runs on the property graph model invented by Neo4j developers. A property graph model differs from a simple graph model in that the nodes and edges consist of objects with embedded properties. Nodes and relations can be seen as objects with certain properties or as hash maps (key-value pairs).

To query graph data Neo4j provides a declarative graph query language called Cypher. It is designed to visually represent graph patterns using an ASCII-Art syntax. Based on these patterns Neo4j queries or manipulates matching graph data. Cypher can also be used for graph specific concepts e.g. shortest path between two nodes.

Like other NoSQL systems, Neo4j is also a schema-less data store. There is no need to define nodes or relations up front. Additionally, Neo4j does not restrict which node can be in a relationship with another node, so any node can be in relation with any other

node.
This flexibility enables application developers to easily expand or change the implicit data model. Just because Neo4j allows you to have no scheme does not mean that you should not use a data model in the background. The advantage of flexibility also brings more responsibility to data maintenance itself, as the system allows almost any kind of changes to the data.

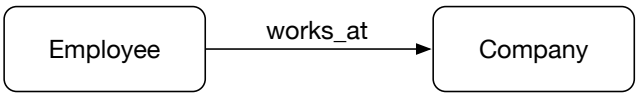

Figure 3.4: Neo4j: data model

Figure 3.4 shows a simple example to illustrate a Neo4j data model. The model contains the nodes `Employee` and `Company`, which would correspond to tables in a relational data model. The relation `belongs_to` between these two nodes would also be represented as a table in a relational data model. A node or relation could additionally contain properties.

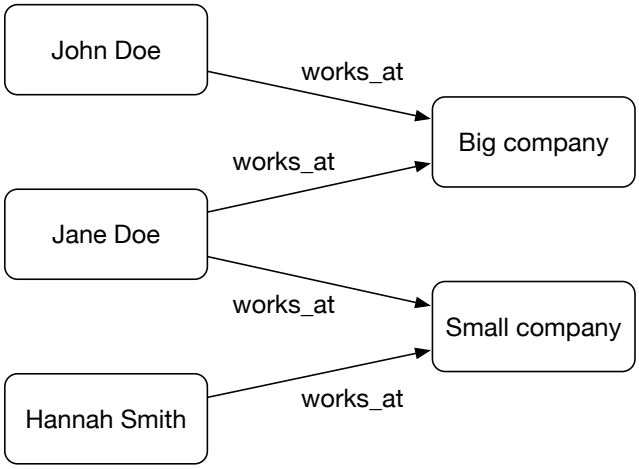

Figure 3.5: Neo4j: data model instance

An instance of the data model from Figure 3.4 could look like Figure 3.5. In a graph data model, all individual instances like employees and companies are represented as a single node connected by their relations. This representation is especially useful to discover which relations between which nodes are possible when designing a more complex graph data model from scratch.

[PVW15]

3.1.5 VoltDB

VoltDB [vol18b] is a scalable NewSQL data store, which aims to run entirely in-memory. VoltDB offers an open source community edition and an enterprise edition. It was designed for critical high-performance applications. To avoid disk access and increase performance, VoltDB utilizes the in-memory storage. In contrast to traditional systems, VoltDB serializes all data access. This enables additional performance increases because resource expensive techniques e.g. locking and transaction logs can be avoided. VoltDB was built to scale, so clustering, replication and sharding are core concepts of its architecture.

VoltDB is a relational database. Standard SQL can be used to define schemas, manipulate and access data. VoltDB fully supports ACID-compliant transactions and uses the ANSI standard SQL. Additionally, to the default SQL functionality VoltDB adds support for JSON columns inside the relational model. By using SQL extensions queries and indices can be used within JSON documents.

However, VoltDB is not intended to be a general purpose database. It focuses on the business-domain including fast and reliable applications e.g. industry applications, social media applications and financial applications. VoltDB is not optimized for all types of applications. Specifically, business intelligence and data warehouse applications are not recommended to be implemented with VoltDB, because they operate on a large set of historical data that has to be queried across multiple tables.

VoltDB's cluster implementation provides parallelism on two levels. First, data can be distributed across the cluster on table bases, but a single table can either be partitioned or replicated. Large and frequently manipulated tables should be partitioned and small frequently read tables should be replicated to increase performance. Second, processing of a task can also be distributed among all nodes in the cluster to achieve high parallelism. Due to VoltDB's consistency and cluster characteristics, it is classified as a CP (consistency and partition tolerance) system in the context of the CAP theorem.

The architecture of VoltDB is based on in-memory storage. Each node in the cluster runs a single-threaded execution engine which processes and stores its associated data in memory. The in-memory architecture minimizes time-consuming disk access, but for recovery and backup proposes disk access cannot be avoided. VoltDB uses data snapshots and command logs to persist and backup data.

Traditional DBMSs implement complex latching and locking mechanisms to handle the execution of multiple parallel transactions. VoltDB can avoid these complex multi-threaded mechanisms because each VoltDB execution engine is single-threaded and manages a transaction queue, so transaction requests are executed sequentially. VoltDB distinguishes transactions which operate on a single partition and transaction which operate on multiple partitions. If a transaction involves only a single partition, the engine operates autonomously. But if a transaction operates on multiple partitions one engine manages and coordinates the other engines.

3.1.6 CockroachDB

CockroachDB [coc18a] is a scalable consistent open-source NewSQL data store built to survive. Although the system is considered as a relational database, it is built on top of a key-value store. CockroachDB consists of multiple layers and each layer communicates and interacts with those directly above and below it. Each layer is responsible for a core concept in CockroachDB:

- SQL layer: The SQL layer is the first layer of the hierarchy and exposes an SQL API to the clients. CockroachDB supports a major part of the ANSI SQL standard. Additionally, CockroachDB developed a Distributed SQL optimization tool (DistSQL). It is used to increase the performance of distributed queries which target multiple nodes in the cluster. The layer is responsible for parsing, planning and executing the incoming requests. The SQL statements are then converted to key-value operations that are passed to the transaction layer.
- Transaction layer: The transaction layer implements support for ACID transactions by coordinating concurrent operations. CockroachDB treats consistency as the most important feature of a database. All SQL statements are handled as transactions, also single statements. The transaction layer implements a two-phase commit process to support transactions across the whole cluster including cross-table transactions. In terms of concurrency control, CockroachDB supports the strongest isolation level (serializable). All other isolation levels are automatically upgraded to serializable. This layer also controls the key-value operations sent to the distribution layer.
- Distribution layer: CockroachDB uses a so-called monolithic sorted map structure to manage distribution. This structure describes the data stored in the cluster and where it is located. It is cached by each node to perform quick lookups to locate a certain record in the cluster. Data is automatically split into contiguous data chunks and distributed across all nodes. It also manages the rebalancing of the cluster when a new node joins the cluster or when a node is removed from the cluster. The distribution layer uses gRPC [grp18] for communication between the nodes.
- Replication layer: The replication layer is responsible for copying data between nodes and establishes consistency between these replicas by using the Raft consensus protocol [raf18]. It is responsible for the distribution of the data when new nodes join the cluster or certain nodes become unavailable. Replicas are used to increases availability and data safety in the case of a node failure. CockroachDB offers a very fine granular feature to manage the location and number of replicas for a specific data set. This feature is called replications zones and also applies to the rebalancing of the cluster.
- Storage layer: Data in CockroachDB is stored in an embedded key-value store called RocksDB [roc18, OAG17] . It is treated as a black box to persist key-value pairs.

RocksDB is also used to store meta data of the system e.g. log files. CockroachDB uses multi-version concurrency control (MVCC) to process concurrent requests and guarantee consistency. MVCC is implemented in the storage layer, but it is also used in the transaction layer to enforce consistency. MVCC enables CockroachDB to implement time-travel queries according to the SQL standard.

CockroachDB implements a special multi-master replication strategy, called Multi-Active Availability. It is a consensus-based strategy which allows CockroachDB to accept read and write operations on all nodes in the cluster, which are responsible for the data.

Consistency is one of the core concepts in CockroachDB. The term consistency applies to two concepts: ACID transactions and CAP theorem. CockroachDB implements support for ACID transactions. This functionality is implemented by the transaction layer. Furthermore, consistency is also satisfied considering the CAP theorem. Like other NewSQL systems, CockroachDB can be characterized as a CP system in the context of the CAP theorem.

To guarantee consistency across the distributed system the individual components have to agree on one current state. Therefore, consensus is another core concept of CockroachDB. This is achieved by satisfying a quorum, so a majority of nodes which are responsible for a certain data set have to agree on the current state of the data, to safely persist the data. Without a consensus, no change can be persisted to prevent an inconsistent state of the data store.

CockroachDB implements synchronous replication to keep consistent copies of data. Synchronous replication requires all replicas to agree on the data set and before it can be committed. The commit is only acknowledged if the replicas successfully persist the commit. Without these regulations, data could become temporarily inconsistent and would lead to eventual consistency.

3.1.7 NuoDB

NuoDB [nuo18a] is an elastic NewSQL system designed to scale horizontally and run on distributed data centers. It provides ACID-compliant transactions and uses SQL as its querying language. NuoDB is closed source and offers multiple editions including a free community edition. NuoDB focuses on consistency in a distributed environment by using its own two-tier architecture. NuoDB splits its architecture between a transactional tier and a storage tier. In addition to these two tiers NuoDB contains an administration component, which has access to both tiers and offers an interface for the system administrator. By separating transactional and storage management NuoDB is able to scale its relational model. Traditional DBMSs implement a tight coupling of processing and storage management. NuoDB eliminates this barrier to scale out and distribute its data. The separation also allows NuoDB to manage failures independently.

The transactional tier is responsible for three of the four ACID properties: atomicity, consistency and isolation. All data in this layer is kept in-memory to keep performance

high. Durability is managed by the storage tier which is responsible for data access and committing data to disk. These two tiers operate on multiple nodes in a cluster. Each node can be configured as one of these two roles: Transaction Engine or Storage Manager. So either a node is a Transaction Engine or a Storage Manager. The transaction and storage tiers are represented by the corresponding nodes. There is no central coordinator, hence there is no single point of failure. The nodes communicate using an encrypted peer-to-peer protocol.

In the case of a network portioning NuoDB sacrifices availability to keep data consistent. So NuoDB will not allow separated groups of nodes to operate concurrently. This could lead to diverging states of the database. NuoDB will remove one of the groups from the cluster to prevent inconsistent data. Therefore, NuoDB can be classified as a CP (consistency and partition tolerance) system in the context of the CAP Theorem.

The Transaction Engines accept incoming SQL requests from clients. The requests are parsed and queries are executed against cached data in the Transaction Engine. If it cannot answer the query by using its local cache, it can ask any of the other nodes. Either it can ask a Transaction Engine, that has the data in memory, or a Storage Manager, which performs a lookup against the durable storage.

If a request arrives at a Transaction Engine, the transaction is propagated to all corresponding Transaction Engines that are associated with the corresponding data. NuoDB supports a tunable commit protocol. It can be configured to forward the transaction asynchronously or synchronously to the other Transaction Engines. The asynchronous configuration is the fastest approach but can cause inconsistency. The synchronous approach allows the developer to configure a number of Transaction Engines which have to acknowledge the transaction before the commit can be considered successful.

The smallest configuration of NuoDB consists of one node that runs one Transaction Engine and one Storage Manager. Transaction Engines or Storage Managers can be added or removed at any time. They will automatically synchronize with the running instances before starting operations. The number of Transaction Engines and Storage Managers does not necessarily have to be the same. If the transaction throughput has to be increased, additional Transaction Managers and can be added to the system. This also applies to Storage Managers. Additional Storage Managers can be added to the cluster in order to increase the number of durable copies of the database.

Beneath the SQL layer data is stored in so-called Atoms. All data is stored and managed through Atoms. Atoms are self-coordinating objects and all information in the system, like user data or metadata, is represented by Atoms. The Transaction Engines are responsible for mapping SQL data to Atoms. The ACID principles atomicity, consistency and isolation are applied to Atoms by the Transaction Engines. The size of Atoms is not fixed. It can be adapted by the system to maximize efficiency depending on the current state of the database. Atoms have unique identifiers and are stored as key-value pairs. NuoDB provides a lookup service, also based on Atoms, to resolve other Atoms across the system. On disk, Atoms are persisted in an archive and additionally, all changes are

stored in a journal. The journal is a simple append-only file that is used to efficiently replay changes.

3.1.8 Summary

NoSQL and NewSQL systems follow different design strategies. What makes NoSQL architectures so special is the choice of the data model. The underlying data model influences many factors of the system, starting with the way how data is physically stored, up to data presentation. Besides, NoSQL data stores follow the same architectural principles. They are built to scale and perform at a high level. This decision also involves stepping down in terms of consistency. NoSQL data stores prefer to implement an eventual or weak consistency model. In general NoSQL data stores follow a more relaxed approach. This also holds true for the data models and schemas.

The compared NewSQL data stores implement new architectures by separating operational and storage management in order to scale horizontally. They have in common that they implement a major part of the ANSI SQL standard to provide the user with a known interface. Underneath that each data store maps the SQL statement to a custom internal data structure. This abstraction helps the systems to perform tasks like data sharding and distribution. Consistency has the highest priority for all compared NewSQL data stores. Unlike NoSQL data stores NewSQL data stores offer configurations, which allow the developer to tune the system's consistency settings. So the developer can decide what degree of consistency should be used. Overall each data store offers an individual design tailored for its specific uses cases.

3.2 Sharding

Sharding is a technique for distributing data across multiple nodes and it is an essential tool regarding the scalability of NoSQL or NewSQL data stores. This is achieved by splitting the data into parts, called shards or partitions. There exist various methods to perform this split. Generally, data can be split up horizontally or vertically. This distinction has its origin in the tabular view of a database. A vertical split separates a table in multiple tables that contain fewer columns, but the same number of rows. A horizontal split divides the table by row, which reduces the number of rows per shard but keeps the number of columns the same. Sharding refers to a horizontal split of the data. Due to the distribution of the data, the workload and resources consumption of the system is also distributed, which can greatly improve the overall performance of the system.

Based on the data store and the underlying data model, different approaches and techniques of sharding are possible. A common approach is, to use the key of a particular row as a decision criterion, to evaluate where the row should be stored. The assignment of rows to shards is usually not fixed, because modifications of the data set, like inserts, updates and deletes, influence the distribution of the data. Balancing and redistribution of the data is an important problem, which data stores have to address to keep the data evenly distributed across all shards.

In general, sharding increases the complexity of the database system. Since the data set is distributed across multiple machines, the system has to handle the internal communication between the machines and the communication with the clients accessing the database. The following sections describe, how the selected data stores address these challenges.

3.2.1 Redis

Redis supports the concept of sharding on two levels. First, Redis is able to use its data types to shard data across multiple Redis instances. Second, Redis supports clustering of server instances by assigning each instance a set of data.

Redis Hashes, Sets and Zsets can be used to shard data. If these data types are used for sharding they are limited in their functionality, because not all functions can be applied to distributed data sets. Redis uses Lua scripting [lua18] to introduce sharded data types. Lua is a lightweight scripting language, which is used to create scripted extensions in Redis. Especially Hashes and Sets are optimized for a distributed use case because almost all operations of these data types support sharding.

To shard a Hash data type, the keys of the hash are used to distribute the data structure. Typically, a hash function is used to partition the keys, but it is up to the developer to decide how the keys are distributed across the shards. Various hash functions can be used. Also, the calculation of the hash depends on the data type of the particular key. Depending on how many keys a shard should hold and how many keys in total should be

stored, the number of shards can be calculated. With these obtained values the shard ID for a specific key can be determined.

The sharding of Sets in Redis works similar to Hashes. Instead of using the hash key to determine the shard ID the values of the set are used. Again, it must be paid attention to, which type the elements of the list have, to calculate a suitable hash value.

The second level of sharding which Redis supports is Redis Cluster. Redis Cluster uses the concept of hash slots. Each key is assigned to a specific hash slot. A Redis Cluster has 16384 hash slots. To assign a key to a hash slot, a hash value is calculated for each key using the CRC16 (16-bit cyclic redundancy check) function and the resulting value is taken modulo the number of hash slots (16384). This ensures that each key is assigned to one of the hash slots.

A Redis Cluster is a composition of nodes. Each node in the cluster is responsible for a subset of hash slots.

For example, a Redis Cluster could be setup with three nodes A, B and C.

- Node A is responsible for slot 0 to 6000.
- Node B is responsible for slot 6001 to 1200.
- Node C is responsible for slot 1201 to 16383.

The association of hash slots to a node does not cause an interruption of the node's operation. Therefore, the addition and removal of nodes in a cluster can be done anytime. If we want to remove a node, e.g. node A, from the cluster, its associated hash slots can be distributed evenly among the remaining nodes B and C. After the node has distributed all its hash slots to the remaining nodes in the cluster, it can be removed from the cluster. Also if a node D joins the cluster, the nodes A, B and C would hand over hash slots to node D.

[Car13]

3.2.2 Cassandra

Cassandra is designed as a distributed system that spreads its data among a group of nodes. Data in Cassandra is stored by column families, which contain rows. Each row has a partition key which the row is identified by.

A cluster in Cassandra uses the concept of consistent hashing to partition or shard data across all nodes. A consistent hash function is a hash function that minimizes the number of reassignments. Reassignments occur whenever nodes are added or removed. Cassandra uses a partitioner to distribute the data sets for each cluster. A partitioner is responsible for generating hash values from the partition keys of the rows. The generated hash values are used to determine to which node a row will be assigned.

Each cluster has a global configuration for the partitioner. Cassandra provides three partitioners:

- **RandomPartitioner**: The RandomPartitioner is used to distribute data across the system using an MD5 hash. It was the default partitioner prior to Cassandra 1.2. Partition range: 0 to $2^{127} - 1$.
- **Murmur3Partitioner**: The Murmur3Partitioner is faster than its predecessor the RandomPartitioner and is now the default partitioner in Cassandra. It uses the MurmurHash function, which is a non-cryptographic hashing function [App08]. It is also recommended to use this partitioner for all new clusters. Partition range: -2^{63} to $+2^{63} - 1$.
- **ByteOrderedPartitioner**: The ByteOrderedPartitioner is used for ordered partitioning. It uses the hexadecimal representation of the keys to order them lexically. This means the partitioned values can be scanned like a sorted list of values. Other partitioners do no support this kind of functionality, but it can also be achieved by using a table index.

However, the usage of an ordered partitioner brings more disadvantages than advantages and is therefore not recommended. Especially, because the order of the keys leads to an uneven distribution of the data. This means that data has to be rearranged by calculating partition ranges based on their estimates of the partition key distribution, which results in a high administrative workload.

Furthermore, ordered hash values can easily lead to sequential read and write operations. Sequential reads and writes of rows are in most cases executed on a single node and are not distributed over multiple nodes. This leads to a high workload of a single machine and does not utilize the capacities of the cluster.

[cas18a]

3.2.3 MongoDB

MongoDB supports sharding through MongoDB sharded clusters. A sharded cluster consists of three types of components, illustrated in Figure 3.6:

- **Shard**: A shard is the main component involved in sharding. It contains a subset of the sharded data and is the unit of replication. All shards together hold the whole data set. A MongoDB cluster consists of two or more shards.
- **Mongos**: Mongos act as interface and router between the client applications and the shards. They manage the communication, query execution and write operations. Shards are generally not directly accessed by the client applications. A MongoDB cluster consists of one or more mongos.

- **Config servers**: Config servers support the cluster by hosting configuration settings and metadata for the cluster. This data includes information about how the data is distributed across the shards.

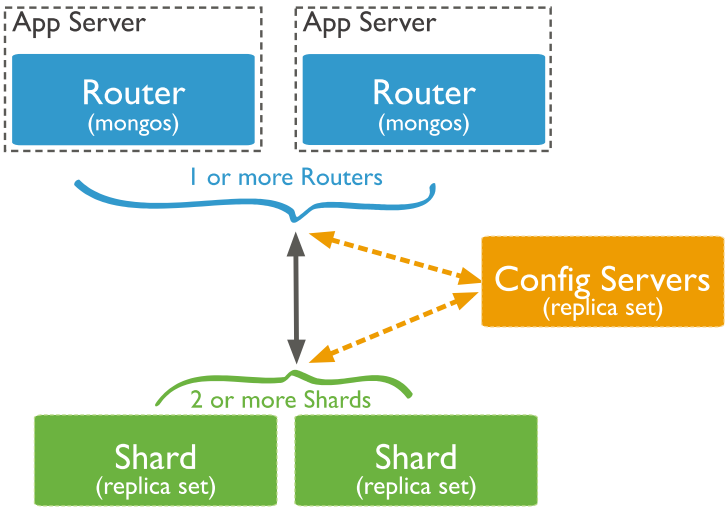

Figure 3.6: MongoDB sharded cluster
[mon18e]

MongoDB organizes data in collections of documents. Data sharding in MongoDB takes place on collection level. In order to enable sharding for a particular collection of documents, each document has to contain a shard key. A shard key consists of one or more immutable fields, but a collection can only have one shard key. The shard key cannot be changed after the sharding and a sharded collection cannot be unsharded.

MongoDB supports sharded and unsharded collections of documents in the same cluster. A cluster always has a designated shard which stores all unsharded collections called the primary shard.

As in other systems, it is crucial to select a suitable shard key because the choice of the shard key influences many aspects of the cluster like scalability, performance and how the data is distributed.

The data parts, which are distributed across shards in a MongoDB cluster, are called chunks. Each chunk has a lower and upper range based on the shard key. A sharded cluster balancer is responsible for distributing the chunks across the shards. It keeps the number of shards at a specific threshold, to evenly distribute data by migrating chunks between shards.

MongoDB offers two sharding strategies to distribute data across a cluster:

- **Hashed sharding**: Based on a hash function MongoDB computes a hash value from the shard key's value. This value is used to assign a document to a chunk and each chunk has a range of values, which it is responsible for. The ranges are calculated based on the possible hash values of the shard key. Figure 3.7 illustrates the distribution of documents using a hash function.

  The hash function leads to equally distributed documents across the cluster, even if the values of the shard key are similar. The favourable distribution of the data leads to a disadvantage when considering query performance because a range-based query is very likely to include multiple shards, which increase the response time and workload of the cluster. When performing queries MongoDB automatically computes the hash values to access the desired shards and chunks.

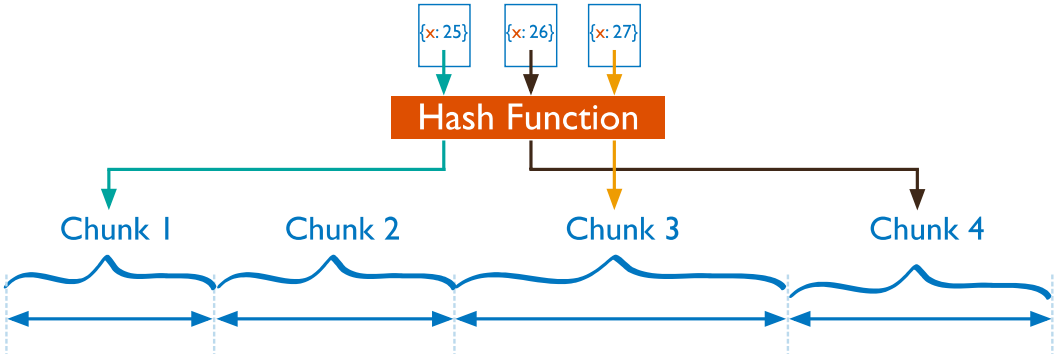

Figure 3.7: MongoDB hashed sharding
[mon18e]

- **Ranged sharding**: Similar to hashed sharding ranged sharding assigns each chunk a range based on the shard key's values. Ranged sharding uses the plain values of the shard key, instead of the hashed values. Figure 3.8 illustrates the split of the keyspace in contiguous chunks. Each chunk has an assigned range, that is used to distribute the documents.

  In contrast to hashed sharding, documents with shard keys which are close to each other will generally be stored in the same chunk. This sharding strategy depends even more on a good sharding key because a bad choice can cancel out the benefits of data sharding.

[mon18e]

3.2.4 Neo4j

Neo4j does not support the common sharding technique, where each node in the cluster stores a different set of data, like other NoSQL data stores, do. Each node in a Neo4j

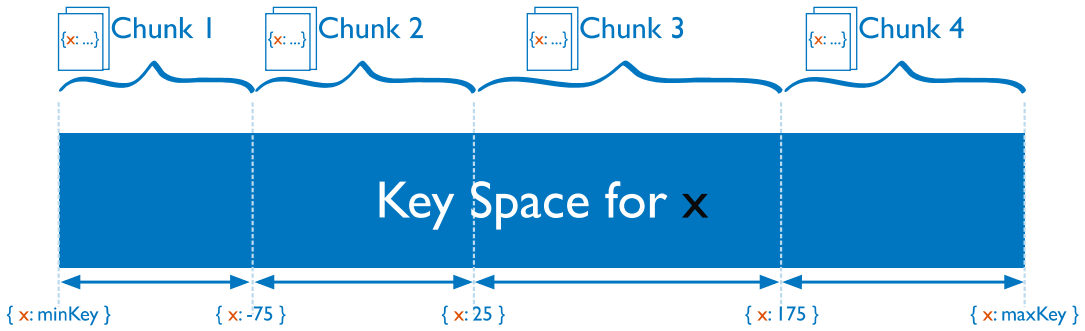

Figure 3.8: MongoDB ranged sharding [mon18e]

cluster stores the whole data graph. This means that if the data set gets too big, a node cannot hold the whole graph in memory, but only a part of it. In order to keep this part in memory as long as possible and not having to constantly load new data from the hard disk, Neo4j uses a technique known as cache sharding.

Cache sharding is not a physical data partitioning technique. It manages the data access for all nodes in the cluster. By using a load-balancing mechanism Neo4j makes sure that certain requests are routed to the same nodes in the cluster.

For example, a Neo4j cluster contains two Nodes A and B and the data graph contains information about employees. The load balancer routes all requests regarding employees with name A to O to node A and from P to Z to node B. Therefore, the nodes are able to hold their data longer in memory and do not have to load a different data set from disk and replace it with the current one. This increases the performance of the cluster because hard disk operations can be kept at a minimum.

To enable cache sharding a load balancer has to be set up. Neo4j recommends using HAProxy [hap17] as a load balancer. Depending on the data domain different routing strategies can be implemented:

- Sticky sessions: The load balancer ensures that the client always connects to the same server, regardless of the query or action.
- Characteristic-based routing: The destination server is selected based on the request, so the load balancer has to know which server is responsible for which query or action.
- Geography-based routing: This strategy takes the geographical origin into account and routes requests or actions from the same area to the same server.

The problem of optimally sharding a graph across multiple nodes in a cluster is NP-complete [KR04]. In general solutions of this kind of problems are derived using heuristics and approximations.

The main benefit of Neo4j and graph data stores, in general, is the power of local graph queries. By partitioning graph data across multiple nodes, most queries would traverse a large part of the nodes in the cluster. That in turn, would result in very high response times.

[PVW15, RF18]

3.2.5 VoltDB

The sharding feature of VoltDB is a core feature of the data store. This feature separates the database tables into independent parts. The sharding is done automatically by the system based on a partitioning column which can be specified by the developer. Figure 3.9 illustrates three tables A, B and C are distributed over three partitions X, Y and Z. Each partition holds a distinct set of rows e.g. Partition X holds A', B', C'.

There is no need to manually manage the shards. The sharded tables can be distributed across multiple servers or on a single server. Furthermore, VoltDB also uses this sharding to split up the processing regarding distributed queries and other operations which accesses the data. The sharding by itself, without distributing the shards across multiple machines, can result in a performance increase in certain scenarios. This increase is particularly clear when a large amount of data is loaded at once, due to the parallelization of the reading process.

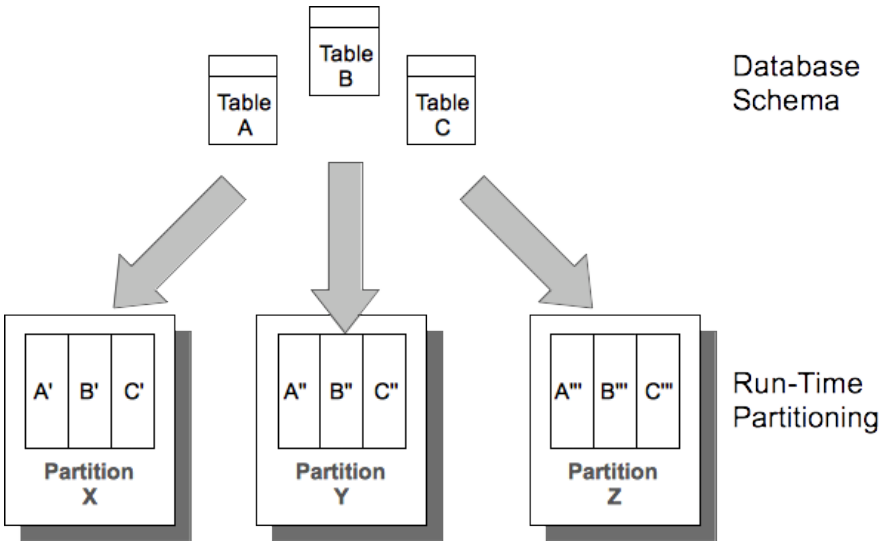

Figure 3.9: VoltDB sharded tables [vol18c]

VoltDB provides a partition statement to specify a partition column, which can be used within the schema definition of the table. Every time a new record is inserted in a partitioned table the system decides on which partition the new data entry should be stored based on the new value of the partition key.

The choice of the partition key affects not only the distribution of the data but also the processing. Hence, both factors should be considered when choosing a partition key.

The name property of an employee would be a decent choice for the partition key regarding data distribution. But if the employee table is often used to compare departments which employees work in, then the employees of one department would be very likely distributed across all shards. The query which aggregates the employees of that department would need to access all these data shards. If the name of the department is the partition key and not the name of employee itself, then the system would probably only need to access one shard, because all employees of the same department would also reside on the same shard.

[vol18c]

3.2.6 CockroachDB

CockroachDB supports automatic data sharding based on the underlying key-value data structure. Data is sharded into so-called ranges. Ranges split the whole key space into contiguous data chunks. A range is the unit of distribution and replication in CockroachDB, so ranges are distributed across multiple nodes in the cluster. Each node stores a set of ranges.

Under the hood, CockroachDB uses a so-called monolithic sorted map structure to manage ranges in the cluster. This map is used for two essential tasks:

- First, the map stores meta ranges which represent the location of the actual data in the cluster and other information about the cluster. CockroachDB stores meta ranges in a two-level index. The meta ranges are replicated and accessed like every other element in the key-value data structure. These special ranges are for example used by queries to locate the responsible node for a certain key.
- Second, the map holds the actual data of the cluster. As mentioned before, the data is divided into contiguous chunks. The default maximum size of a range is 64 MB. CockroachDB claims that 64 MB is the optimal size, small enough to migrate the data range between nodes and large enough to store a relevant set of data.

If a range exceeds the threshold of 64 MB, the range is split up into two ranges. Consequently, each range covers a contiguous part of the key space. There is also a lower threshold for the size of a range, if ranges fall below this threshold they are merged back together. The aim of CockroachDB is to keep the ranges relatively small to easily

distribute the data chunks across multiple nodes in the cluster. It also automatically manages the rebalancing of ranges between nodes in the cluster. The nodes communicate using a peer-to-peer gossip protocol to exchange information about the stored ranges, capacity and other meta information.

The meta ranges are stored and cached by each node to keep up with the current state of the cluster. If a node receives a request, it is able to look up the responsible node by comparing the keys in the request with the keys in the meta ranges and send the request to the node. If the requested data moved to a different location, the node replies with the new location.

[coc18a]

3.2.7 NuoDB

Since version 2.6 NuoDB fully supports the SQL standard for partitioned tables. These partitions are used to distribute data across multiple servers. NuoDB uses two mappings to represent its sharded data structure. The first mapping is between a table partition and a storage group (SG) and the second mapping is between a storage group and a storage manager (SM). On top of that resides an in-memory layer, which contains transaction engines (TE). In Figure 3.10 three storage groups are distributed and replicated over three storage managers. This represents the mapping

The logical unit of storage in NuoDB is a storage group. A storage group contains one or more table partitions and each table partition is assigned to exactly one storage group. These storage groups can be assigned to multiple storage managers. A storage manager represents a physical storage location. Typically, one storage manager runs on each node in the cluster. This means that the assignment of a storage group to a storage manager determines the physical location of the data set.

NuoDB offers two policies to horizontally split a table into multiple partitions:

- Partition by range: This partitioning policy allows the developer to specify value ranges for a given column of a table. This is implemented using the VALUES LESS THAN clause. This clause evaluates to true if the input value is less than the specified value. Each row is assigned to a partition based on the evaluation of these clauses. The clauses are sorted from lowest to highest and the keyword MAXVALUE is used to assign rows to a partition which do not match prior clauses. Each row must be assigned to a partition otherwise the system throws an error.

```
PARTITION BY RANGE (employee_id)
(
  PARTITION employee_1 VALUES LESS THAN (1000),
  PARTITION employee_2 VALUES LESS THAN (2000),
  PARTITION employee_3 VALUES LESS THAN (3000),
  PARTITION employee_4 VALUES LESS THAN (MAXVALUE)
```

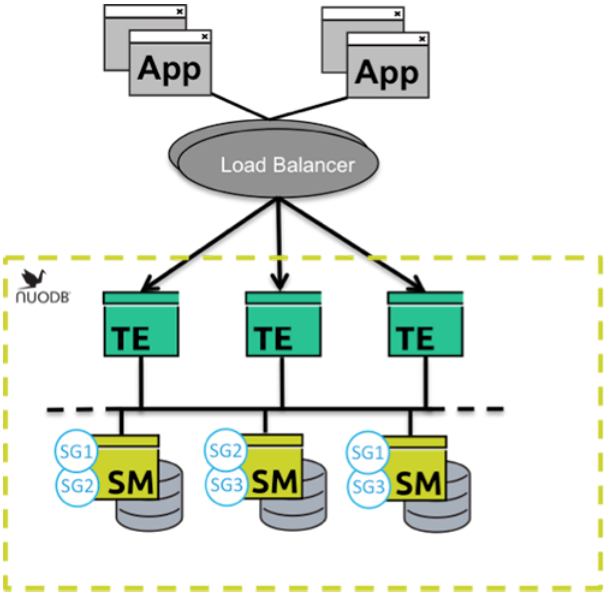

Figure 3.10: NuoDB sharded architecture [nuo18c]

```
);
```

- Partition by list: The list partitioning policy works similar to the range partitioning policy, but instead of the VALUES LESS THAN clause it uses the VALUES IN clause. This clause evaluates to true if the input value is contained in a specified list. Each row of the table is assigned to partition based on these statements. There is also the DEFAULT keyword which assigns rows to a default partition if no other clause matches. As with range partitioning, the system throws an error if a row cannot be mapped to a partition.

```
PARTITION BY LIST (department_name)
(
  PARTITION employee_1 VALUES IN ('Sales', 'Marketing', 'HR'),
  PARTITION employee_2 VALUES IN ('Research', 'Development'),
  PARTITION employee_3 VALUES IN ('Accounting', 'Finance'),
  PARTITION employee_4 VALUES IN (DEFAULT)
);
```

[nuo18c]

3.2.8 Summary

The individual databases pursue very different approaches due to their diverse data structures. What they all have in common is that every record is assigned to a shard

based on some shard key, the hash-based sharding approach is very popular, mostly because it evenly distributes data across the nodes.

The sharding implementation of the NoSQL data stores greatly depends on the underlying data structure of the system. Especially Redis provides for each data structure a sharding implementation. Also, the sharding implementations of the NewSQL data stores are not as similar as one would aspect. For example, the implementation of CockroachDB is based on the underlying key-value store and on the other hand NuoDB uses the SQL standard for partitioned tables to shard and distribute the records.

The data stores are torn between flexibility and automation in terms of sharding. For example, CockroachDB handles the sharding process, including the rebalancing of nodes, automatically without the need to interfere manually. On the other hand, Cassandra offers different sharding strategies, which can lead to performance increases, depending on the data set. This kind of flexibility requires a profound understanding of the data set itself, without this knowledge the sharding can result in an uneven distribution, which would cancel out benefits of sharding.

Neo4j is the only data store that does not support the common sharding approach of physically splitting the data across multiple machines. It uses cache sharding. Each node in Neo4j holds the whole data set and only a part (shard) of the data is held in memory. The remaining data is stored on disk and only accessed if necessary.

In most cases, sharding is closely coupled with replication because the data shards are used as the base for replicating data. For example, NuoDB uses the data shards (storage groups) to distribute parts of the data to multiple servers and it uses the same storage groups to replicate the data by assigning the same storage group to multiple servers. VoltDB has a more restricted relation to replication because tables in VoltDB can either be sharded or replicated.

In conclusion, all reviewed data stores provide on the first sight very similar sharding techniques, but the approaches vary greatly based on the underlying data structure of the system.

3.3 Replication

Replication refers to a technique of keeping multiple copies of the same data on different nodes in the cluster. This approach has multiple benefits: The most prominent benefit is the fact that the data is backed up on multiple even physically separated machines, to prevent data loss in a disaster scenario. But in general, replication cannot completely replace traditional backup methods, because it only replicates the current state of the data. Replication does not keep a history of old data records.

The main reason why modern systems deploy replication is horizontal scalability. Horizontal scalability can be divided into the read-scalability and write- scalability. Scaling reads enables the system to serve many concurrent users data from the data store and scaling writes enables the data store to perform many concurrent writes. Read-scalability can be achieved if multiple replicas are allowed to accept and answer read requests from clients and this applies analogously to write requests. Which form of scalability is available depends on the data store.

Another aspect that needs to be differentiated is the way in which updates are propagated and acknowledged by replicas. When using synchronous replication updates are only acknowledged to the client, if the replicas also acknowledge the update. Synchronous replication is used by NewSQL stores to ensure strong consistency within the system. If an update fails on one replica, the other nodes have to roll the update back, to keep the data consistent.
Asynchronous replication is used by most NoSQL data stores because it provides higher performance. If a node receives an update and it is successful, without knowing the result of the other replicas, the receiving node acknowledges the update. This asynchronous fashion of update propagation goes hand in hand with the eventual consistency model used by NoSQL data stores, because it cannot be guaranteed that all datasets of the replicas are identical. This means that a read request, that is sent to two replicas at the same time, can return two different results.

To decide which replicas are allowed to answer which type of queries and how to propagate updates to other replicas, the data stores implement different master-slave and multi-master architectures. The following sections describe these different replication approaches:

3.3.1 Redis

Redis uses a master-slave setup for replication. One node is the designated master and multiple slave nodes are connected to it. The goal of a slave is always to represent an exact copy of the master server. This replication approach does not only provide data safety in case of emergency (e.g. hard disk failure), it also allows the system to scale read operations because slaves are able to answer queries as well. Write requests are only accepted by the master. In case of a connection loss or timeout, the slaves will automatically try to reconnect to the master. This is achieved by covering these three scenarios:

- If the connection between the master and the slave is stable, the master sends a stream of commands to its assigned slaves. The slaves execute these commands in order to replicate the master's data set.
- In the case of connection loss or timeout between the master and a slave, the slave tries to reconnect to the master. If the reconnection is successful, a partial resynchronization is initiated. A partial resynchronization attempts to retrieve the lost commands and execute them to restore the state of the data set.
- If a partial resynchronization fails or is not possible, the slave initiates a full resynchronization. A full resynchronization does not obtain the missing commands, but it retrieves a full snapshot of the data set. This snapshot is created by the master and applied by the slave. After this process, the master continuous sending the command stream.

If a user manipulates the data set of the master the command is forwarded to its slaves to replicate the change. The master does not wait for the slaves to acknowledge the execution of the command back to the user. Additionally, the slaves periodically acknowledge the amount of data received from the master. This means that Redis uses asynchronous replication by default. However synchronous replication can be achieved using the WAIT command, but the WAIT command can only ensure that there is a certain number of acknowledged copies on the other nodes.

The asynchronous replication concept is also reflected in a way that the replication tasks are non-blocking for the master. This means that the master continues to answer queries and requests during synchronization tasks like initial synchronization and partial/full resynchronization. The asynchronous nature of Redis reflects the relaxed approach of NoSQL data stores regarding consistency.

Also, the replication tasks on the slaves are considered non-blocking. During synchronization tasks, the slave is able to answer queries, but the slave can only access the old dataset to answer these requests. However, the slave nodes can be configured to respond to requests during these time slots with an error, in order to avoid the delivery of outdated data.

Furthermore, Redis supports a replication structure where slaves are able to connect with other slaves and create a tree structure. The sub-slaves will receive the same command stream as the directly connected slaves.

[red18b]

3.3.2 Cassandra

Cassandra distributes replicas over several nodes in the cluster based on the replication factor. The unit of replication in Cassandra is a row and the replication factor determines the number of replicas in the system. A replications factor of 1 means that there is no

additional copy of the row in the system, only the original row itself. A replications factor of n would indicate that there are n copies of the row in the system and each copy is stored on a different node. This means that the replication factor should not exceed the number of nodes in the cluster. The replication concept of Cassandra does not identify one replica as a primary or master replica, all replicas are equally important so they are replicas of each other.

The placement of replicas can be configured using a replication strategy. Cassandra provides an interface, which can be used to implement a custom replication strategy. Furthermore, two replication strategies are already implemented and fit the majority of use cases.

The choice of the right replication strategy is crucial because the strategy influences which node is responsible for which key range. This, in turn, affects which nodes receive the write operations, which can greatly affect the efficiency of the system. If your system includes two geographically dispersed data centers and replicas spread across those data centers, the write operations would also be distributed, which in turn would lead to performance loss. The flexibility to choose a replication strategy or implement a custom strategy allows the system to tackle this problem and place the replicas based on the underlying network topology of the cluster.

Cassandra offers the following two strategies:

- **Simple Strategy**: This strategy should only be used if Cassandra operates on a single data center and one rack because this strategy is unaware of the network topology of the cluster. The nodes in a Cassandra cluster are organized in a ring and replicas are just placed clockwise in the ring. This means that the next node in the ring could be in a different data center.

- **Network Topology Strategy**: This replication strategy is tailored for a distributed scenario in which the data set is spread across multiple data centers and racks. First, the number of replicas in each data center has to be specified. An asymmetrical replication between the data centers is also possible. For example, the data center, which is used for production, can have three replicas and another data center, which is used for development, can have only one replica. To choose a destination node for a replica, the nodes in the ring are iterated clockwise until reaching the first node in another rack, but still in the same data center. Replicas in the same data center should be placed in different racks because there is a high probability that they share the same properties like brand and date of purchase, so machines in the same rack often fail at the same time.

[Hew10, cas18e]

3.3.3 MongoDB

Replication in MongoDB is handled by a so-called replica set which consists of multiple mongod processes. A mongod process is the primary daemon process of MongoDB. MongoDB uses replica sets to provide high availability and redundancy, in case of failure and supports multiple replicas on different nodes.

A replica set can contain multiple nodes, which can have one of these three roles: primary, secondary or arbiter. There is one primary node, multiple secondary nodes and one optional arbiter node. Illustrated in Figure 3.11

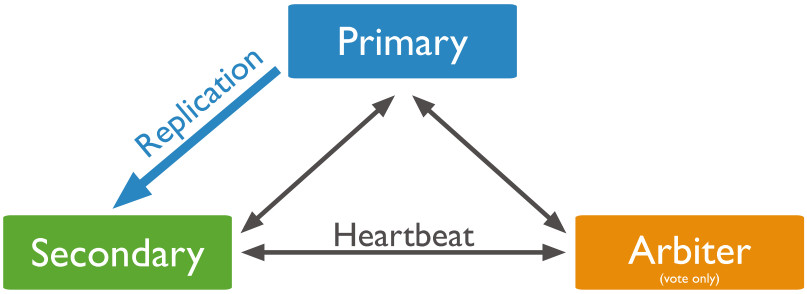

Figure 3.11: MongoDB replication architecture
[mon18d]

- **Primary**: A replica set has exactly one primary node. The primary is the only node which receives write operations from the clients. These write operations are applied to the primary's data set and stored in an operation log. This log is used for replication. All nodes including the primary are able to accept read operations, but the read operations are by default forwarded to the primary node. In the case that the primary node fails or becomes unavailable, an election process is started to decide which secondary node becomes the new primary node.

- **Secondary**: A replica set can contain one or more secondary nodes. The primary's operation log, in which all write operations are stored, is replicated to all secondary nodes. The secondary nodes apply the operations from the log to replicate the data set of the primary in an asynchronous process. Nodes in a replica set ping each other every two seconds to check the availability of the other nodes, this is called heartbeat. If a node does not reply in a certain time, the node will be marked as inaccessible. This information is used in the election process.

- **Arbiter**: This node is optional in a replica set. It does not hold a replica of the data and cannot become a primary node, but it has a vote in the election process. The main purpose of an arbiter node is to manage a quorum. This allows MongoDB to have an uneven number of voting nodes without the overhead of a secondary node.

The replication approach in MongoDB provides an automatic failover in case a primary node does not respond to a member of the replica set. The first secondary node which notices this starts an election, see Figure 3.12. The election can be influenced by the priority of the secondary nodes.

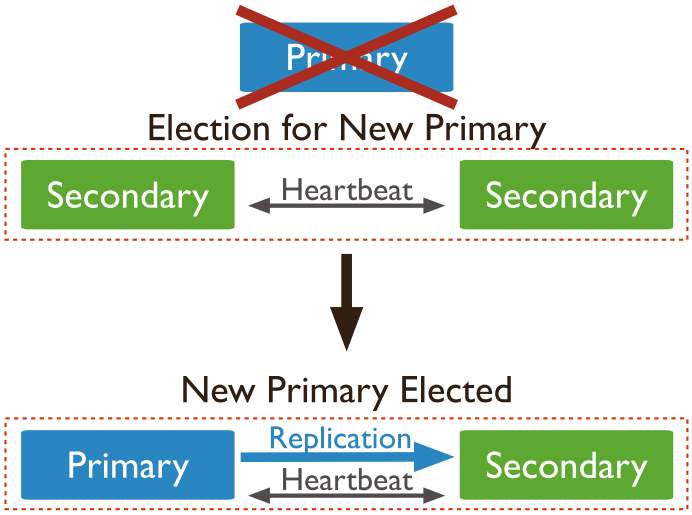

Figure 3.12: MongoDB election
[mon18d]

In the case of a network partitioning the nodes keep track of the number of nodes which are still reachable in their partition. If the primary detects that it is located in a minority partition it automatically steps down and becomes a secondary node. On the other side, if a node is able to reach a majority of the replica set, it starts an election and becomes the new primary. During an election there is no primary and MongoDB does not accept any write operations, but read operations are still accepted.

The asynchronous nature of MongoDB's replication concept implies that there is a time window in which a read operation on a secondary node returns outdated data, which does not reflect the data set of the primary node. This is why MongoDB forwards by default all read requests to the primary node. This behaviour can be changed in the settings.

[mon18d]

3.3.4 Neo4j

Neo4j's high availability (HA) cluster implements a master-slave architecture. A Neo4j cluster contains one master and zero or more slave nodes. Each node in the cluster contains a full copy of the original data set. Neo4j provides a special slave node which is called arbiter. It does communicate with other nodes in the cluster, but it does not hold a replica of the data set. The master is the primary instance which handles write

transactions from the clients. Surprisingly, Neo4j not only allows read operations but also write transactions to slave nodes.

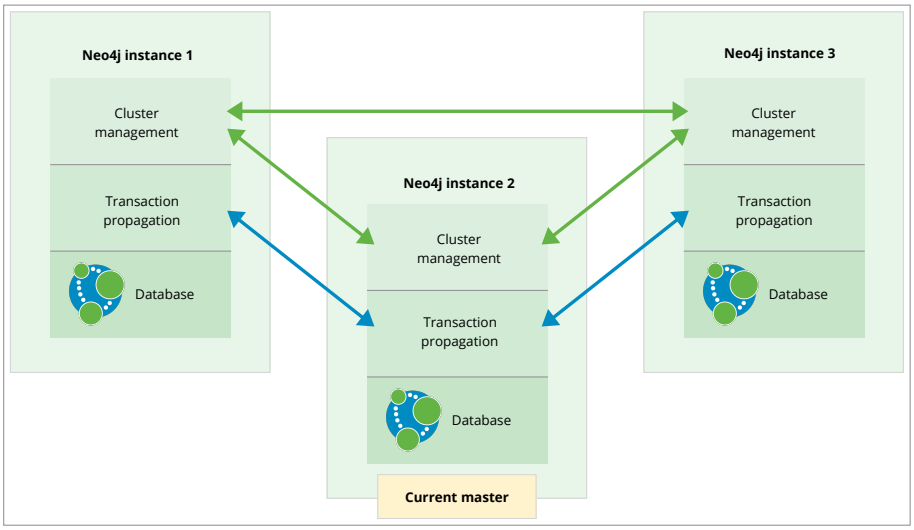

Figure 3.13: Neo4j master-slave cluster [neo18b]

Figure 3.13 illustrates a master-slave setup with one master node and two slave nodes. The green arrows represent the communication between all nodes of the cluster regarding election management and the actual communication about data replication is represented by the blue arrows.

If a write transaction is executed directly on the master, it will execute the transaction and manipulate its data set. If the transaction succeeds it is forwarded to the assigned slave nodes. After the transactions were successfully executed on the master, without knowing if it will be successfully executed on the slaves, the client is notified that the transaction was successful. This means that Neo4j uses an asynchronous replication approach. This optimistic approach is very common for NoSQL data stores.

When a transaction is executed on a slave instead of the master, each write operation will be synchronized with the master. This process includes the usage of locks on the master and slave. First, the transaction is forwarded to the master and only if the transaction is successfully executed, it is executed on the slave. Although Neo4j allows writing access to slaves, the increased communication overhead and the additional use of locks increase the workload on the cluster.

When nodes in the cluster observe that a node is unavailable it will be marked as temporarily failed. After the node becomes available again, it will automatically synchronize to recover the current state of the master. If the master becomes unavailable a new master will be elected by the remaining slaves.

In order to elect a new master, a cluster must have a quorum. Neo4j defines a quorum as follows: more than 50% of active cluster members. This means that if a cluster should be able to handle n node failures a minimum of $2n + 1$ nodes is necessary to satisfy the quorum. To handle a single node failure, a cluster would consist of three nodes to satisfy the quorum and elect a new master. After the election was successful the new master broadcasts its new role to the cluster until this time all write operations from clients are rejected.

3.3.5 VoltDB

VoltDB supports two levels of replication. The first level is based on database tables and the second level affects the whole database. VoltDB uses an ACID compliant approach, which means that a transaction is not completed until all replicas have successfully applied the changes. A SQL statement has to be either visible to all replicas or to none.

Replicated tables in VoltDB are distributed to all nodes in the cluster. In VoltDB tables can either be partitioned or replicated across the cluster, VoltDB offers no possibility to do both. Figure 3.14 illustrates a database schema consisting of multiple partitioned tables (Table A, B and C) and a replicated table (Table D). Partitioning and replication of tables always target all nodes in the cluster. Additionally, you have to decide for each table if it will be partitioned or replicated. If a partition column is set, the table will be partitioned, otherwise, the table will be replicated. VoltDB recommends replicating small and read-only tables, which are often used for joins. This allows the system to execute certain queries only on one node, without the overhead of communicating with other nodes.

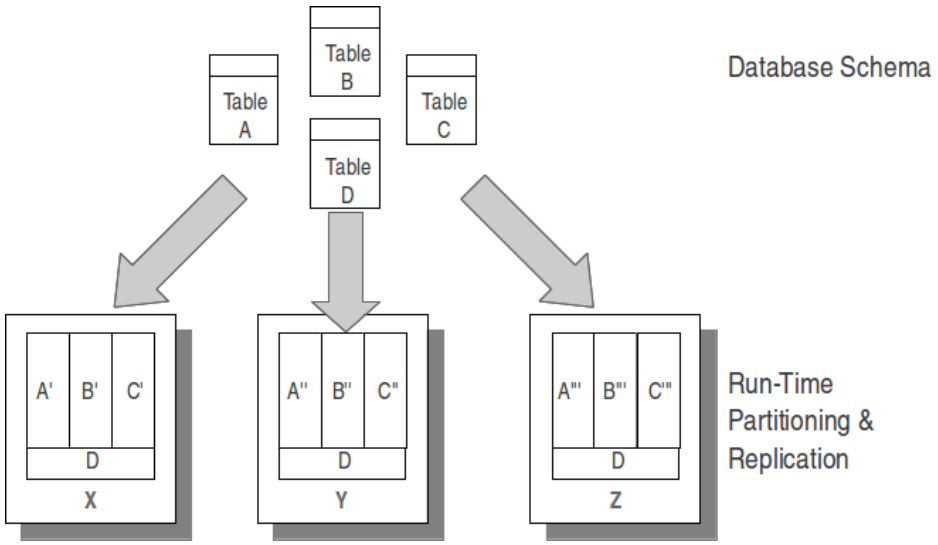

Figure 3.14: VoltDB replication [vol18d]

In addition to table replication, VoltDB also offers the option of replicating the entire database cluster. VoltDB offers two options for database replication:

- Passive replication (One-way): The passive replication approach implements a master-slave replication model on a data center level. One way means that replication only happens from the master to the slaves. The master data center is the only data center which accepts write operations from the clients and the slave data centers are read-only. To create a copy of the master data center, all changes are forwarded to the slave data centers. The slaves only allow transactions from the master to modify their database.
- Cross data center replication (Two-way): The cross data center replication (XDCR) can also be called active replication or two-way replication because all data centers are allowed to accept write operations from the clients and the changes from each data center are broadcasted to the other participating data centers. When a client modifies a record in one data center the changes are committed locally first, then forwarded to the other data centers. This can result in a conflict if two or more data centers modify the same record.

3.3.6 CockroachDB

CockroachDB implements a special multi-master replication strategy, called Multi-Active Availability. It is a consensus-based strategy which allows CockroachDB to accept read and write operations on all nodes in the cluster, but only for a specific range. This is the difference to a regular multi-master replication where all nodes can receive all read and write requests.

The consensus algorithm CockroachDB uses requires a quorum to accept the incoming changes for a specific range. More than half of the responsible nodes must agree to achieve a majority before those changes are committed. Hence, the smallest cluster which is able to achieve a quorum must have three nodes. The replication factor x determines the number of replications for a data set, so a cluster can handle $(x-1)/2$ node failures.

When a new node joins the cluster CockroachDB rebalances the replicas in the cluster to utilize the new resources. In the event of a node failure, the system automatically realizes that a certain node is unavailable and redistributes replicas to maximize data survivability.

One of the core components is the consensus protocol Raft. It is responsible for the distribution of the data when new nodes join the cluster or certain nodes become unavailable. But most importantly, Raft ensures that the nodes in the cluster agree on the current state of the data. This is done by dividing the nodes into so-called Raft groups. One node in the group is the leader and the others are followers. The leader is responsible for the Raft log, which contains the operations the nodes have agreed on. This log can be used to replay the last operations if a node becomes temporally

unavailable and has to catch up to the current state of the database. When a follower receives a write operation it is proposed to the group leader. If the operation is accepted by the quorum it is appended to the log and the changes are replicated on the other nodes of the group.

If a new node joins the cluster, it would take too much time and resources to replay the whole Raft log and get the new node up to date. Therefore, each node can create a snapshot of its data, which can be sent to a new node. After the new node has loaded the snapshot, any remaining changes are applied using the Raft log.

CockroachDB offers a very fine granular feature to manage the location and number of replicas for a specific data set. This feature is called replication zones and also applies to the rebalancing of the cluster. The developer can create detailed replication zones on multiple levels and CockroachDB also offers pre-configured replication zones.

Replication zones can be configured on four levels:

- Cluster
- Database
- Table
- Row

When a certain data set is replicated, the system uses the most specific zone available. So if a certain row is modified the system uses the replication zone for this row if it is specified, otherwise, it uses the replication zone for its table, then CockroachDB looks for a database and cluster replication zone.

[coc18c]

3.3.7 NuoDB

NuoDB implements an active-active (multi-master) replication strategy, which allows the clients to perform read and write operations on all nodes in the cluster. By default, each storage manager on a node keeps a complete copy of the database. New storage managers will automatically synchronize with another storage manager in the cluster.

The storage groups, which result when tables are sharded, can also be used for replication. A storage group can be assigned to multiple storage managers to replicate the data across multiple nodes in the cluster.

The distributed architecture of NuoDB allows the system to operate one database across multiple data centers. Figure 3.15 shows that the storage manager (SM) keep the replicas in different data centers to provide durability in the event of a data center outage. If the connection between the data centers fails, then the components in one data center will

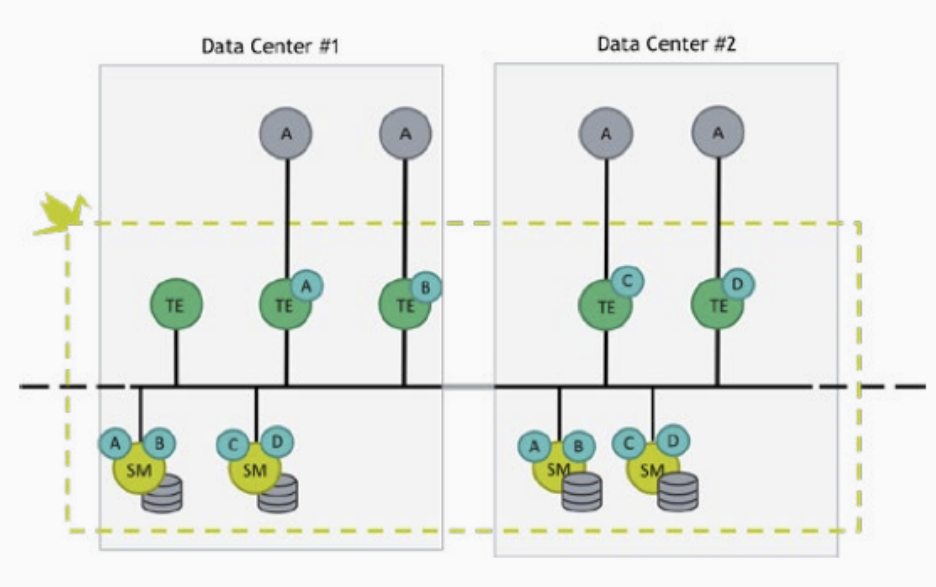

Figure 3.15: NuoDB data center replication [nuo18c]

be automatically shut down to prevent inconsistency. This feature can also be disabled, but NuoDB cannot automatically fix the potential inconsistencies.

By default, NuoDB uses synchronous replication, but the system can also be configured to support different strategies. This strategy can be adapted using different commit database options:

- Safe: This is the default option. When a transaction is executed the transaction manager sends a pre-commit message to the storage manager hosting a replica of the corresponding data. The actual acknowledgement will not be sent to the client until all storage managers acknowledge their pre-commit message.
- Local: The transaction manager sends a commit message to all involved storage managers, so they can replicate the changes, but does not wait for their acknowledgements. This means that the transaction manager only waits for the transaction to complete locally, before sending the acknowledgement to the client.
- Remote: This commit option takes a parameter n which represents the number of storage managers which have to acknowledge the commit. Hence, the transaction manager waits until n storage managers confirm the commit, before acknowledging the transaction to the client.
- Region: This commit option works similar to the commit option remote, but instead of waiting for n storage managers the transaction manager waits for n regions,

before acknowledging the transaction to the client. A region represents a geographic location of a database.

[nuo17, nuo18b]

3.3.8 Summary

The research confirms that NoSQL databases rely more on performance and less on consistency. This is also confirmed by the choice of the architectures of the individual systems. The replication architectures are mainly focused on asynchronous communication, which in turn highlights again the eventual consistency model.

Most NoSQL data stores implement master-slave architectures, where writes are only allowed on the master and reads are allowed on all replicas, rather than a multi-master architecture. This emphasizes the performance goal of NoSQL systems because a multi-master architecture would imply a higher communication effort, which would decrease the overall performance.

The NewSQL data stores, on the other hand, implement multi-master architectures. This architectural approach allows the clients to perform read and write operations on all nodes in the cluster. Writes to multiple nodes can lead to conflicts in the system, to resolve these issues most NewSQL data stores implement a consensus mechanism to agree on a solution. In order to support the ACID properties, the NewSQL data stores use synchronous replication, which implies a higher communication overhead, but they accept this overhead to achieve consistency.

But the line between replication strategies is not always that strict. Most data stores offer the possibility to configure the degree of synchronous and asynchronous replication. For example, the commit options of NuoDB allow for a very granular configuration between synchronous and asynchronous replication. In general, it is easier to step down from synchronous replication to asynchronous replication as the other way round.

In conclusion, there is always a trade-off between consistency and performance. NoSQL data stores prefer performance over consistency and NewSQL data stores prefer consistency over performance. This does not mean that there is no consistency in NoSQL systems or that NewSQL systems are slow, but it is necessary to know the requirements and goals, which an application should achieve, before choosing a data store.

3.4 Querying

Generally, querying is a key task for all kinds of data stores. NoSQL and NewSQL data stores implement very contrasting techniques to retrieve their stored data.

As NoSQL data stores were designed, querying was not the priority, they are optimized for flexibility, scalability and performance, with exception of graph data stores. Essentially, most NoSQL data stores lack querying mechanisms compared to traditional DBMSs.

The most influencing factor, how querying mechanisms work in NoSQL data stores, is the data model. The data model is closely tied to the query method. This means that there is no standard, like SQL, connecting NoSQL data stores. Almost all NoSQL data stores provided a proprietary API to access the stored data according to their data model. The NoSQL evaluations of this section will focus on how the querying mechanisms work according to the specific data models.

On the other side, NewSQL data stores prioritize querying more than NoSQL data stores. NewSQL systems by definition support SQL as their main interface for data retrieval and manipulation. They are not as diverse as NoSQL data stores in terms of querying, because all of them support SQL, but not every NewSQL data store supports SQL to the same extent. The NewSQL part of this section will focus on the limitations considering the degree of SQL compliance, rather than listing all supported standard SQL statements.

3.4.1 Redis

The querying options in Redis are very limited. A value can only be queried by its key depending on the data type:

- String: A string can be retrieved from the key-value store using its key.
- Hash: The whole hash can be accessed by its key. Additionally, all values or keys of a hash can also be queried and the values of the hash itself can be accessed separately using the key of the hash and the key of the particular value.
- List: A list or a defined range of elements in the list can be retrieved by the key.
- Set: The set can also be accessed by its key. The set datatype offers additional methods to query the intersect, difference or union of multiple sets.
- Sorted set: Sorted sets can be queried like normal sets in Redis. Furthermore, each element of a sorted set has a score as a sorting criterion, this score can be used to query a range of elements.

It is possible to query all keys matching a pattern, but this command should not be used in production environments. The time complexity of this command is $O(N)$ with N

being the number of keys in the data store because Redis iterates all keys and tries to match the pattern.

To extend the rather limited querying capabilities, Redis' data structure can be used to create secondary indices. In most cases, the sorted set data type is a very useful data structure to create a secondary index. For example, the following Redis hash represents employees of a company. The HMSET command allows us to set multiple key-value pairs on a hash given a key.

```
HMSET emp:1 id 1 givenname John surname Doe department 45
HMSET emp:2 id 2 givenname Jane surname Doe department 45
HMSET emp:3 id 3 givenname Sam surname Smith department 12
```

The ZADD command is used to add an item with a score to a sorted set. This data structure represents the secondary index. This index has to be manually created and maintained, so the developer has to update the index if the employee data changes. The following commands create a sorted set with the employee identifiers as values and the department identifiers as scores. The sorted set can now be used to query all employee identifiers given a range of department identifiers (score).

```
ZADD emp.department.index 45 1
ZADD emp.department.index 45 2
ZADD emp.department.index 12 3
```

The sorted set data type has another useful feature regarding indices. The score of a sorted set is a floating point number, so it can only be used to index numbers. If items in the set have the same score, the items are sorted lexicographically and Redis offers a command to query ranges in a lexicographical way. This feature is very similar to a b-tree, which is used in a variety of data stores to implement indices and can be used to create complex indices in Redis.

[red18c]

3.4.2 Cassandra

Cassandra provides a query language similar to SQL called CQL (Cassandra Query Language). It is the primary language used in Cassandra to access and manipulate resources in the data store. Prior to CQL, the main interface for data management was Thrift [thr18]. SQL and CQL share in most cases the same syntax and functionality, but CQL does not offer the same set of query functionality like SQL. Most of these restrictions are due to the fact that CQL operates on a distributed NoSQL data store and complex queries would greatly reduce the performance of the cluster. The strength of Cassandra and CQL is the query performance but not the expressiveness of the queries.

Additionally, CQL provides support for JSON. CQL can be used to return tables formatted in JSON. This feature not only applies to the SELECT statement, but also to the INSERT statement, so data can be inserted and accessed using JSON.

The main restrictions compared to SQL are:

- CQL does not support joins.
- CQL does not support subqueries.
- A WHERE condition can only be applied to a column, if a secondary index exists for this column, so all attributes used in a WHERE clause must have an associated secondary index, besides the primary key.
- CQL provides no OR or NOT operators, only the AND operator is available in the WHERE clause.

After defining a table, a secondary index can be created for a column. As mentioned before, secondary indices are essential to query tables using a column, which would not be queryable otherwise. An index can also be created on a collection column. A collection is an embedded data structure with a limited size. CQL contains the set, list and map collection data types.

Secondary indices can have a great impact on the performance of the cluster. Cassandra stores the index table on each node in the cluster and it is recommended to create indices only on specific columns. Especially, indices should not be created on columns with a high-cardinality. The cardinality of a column is high if it contains many distinct values. So a column which contains mostly unique values, should not be used to create a secondary index. A bad example would be the address of an employee because this column contains many distinct values and a good example would be the name of the department the employee works in. Furthermore, indices should not be created on columns that are frequently updated or deleted.

[cas18d, cas18c]

3.4.3 MongoDB

MongoDB provides a rich set of possibilities regarding data retrieval, starting with the find operator. The find operator is the primary tool to query documents from a single collection. It is based on the query-by-example concept, as the name suggests, the find operator queries all documents of a collection which match the provided example document.

```
db.employees.find({ "surname" :  "Doe", "givenname" : "John" })
```

This example would return a subset of the employees collection, where the *surname* field is equal to *Doe* and the *givenname* field is equal to *John*. This is a rather simple query that illustrates the query-by-example concept, but the find operator allows for more complexity. MongoDB provides many operators to go beyond the exact matching of documents:

- Logical operators: $or (OR), $not (NOT). The logical AND is implicitly provided by the query-by-example concept.
- Comparison operators: $lt (<), $lte (<=), $gt (>), $gte (>=), $ne (not equal)
- Include operators: $in (in), $nin (not in)

MongoDB provides many more operators. These are the primary operators, which can be compared to SQL statements. A detailed list of operators can be found in the MongoDB documentation [mon18c].

Aggregation pipeline Besides the find operator, which queries only a single collection, MongoDB offers the lookup operator to combine documents from two collections. The lookup operator performs a left outer join, so the input documents are extended by an array of matching documents from the joined collection if the specified fields match. For example, a data store could have two collections: a collection that contains departments of a company and a collection with employees. Each department contains a list of identifiers of employees, who work in this department. The lookup operator can be used to join the referenced employees to the corresponding department so that the department contains a list of employee documents.

The lookup operator can be used together with a set of other operators to perform a complex chain of operations. This concept in MongoDB is called aggregation pipeline and it can also operate on a sharded collection. The chaining of operators illustrated in Figure 3.16 corresponds to a SQL statement (WHERE, SELECT, JOIN and GROUP BY)

Figure 3.16 contains following operators:

- Match: Filters documents
- Project: Adds, removes, renames or computes fields
- Lookup: Left-outer join
- Group: Combines documents

Furthermore, MongoDB also provides a map-reduce function. The map-reduce function has two main phases, the map and the reduce phase. First, the map part is executed for each document in the collection and emits one or more objects. Then the reduce phase combines the result of the map phase. Both phases use a custom JavaScript function to perform the map-reduce operation. A map-reduce function can also operate on a sharded collection like an aggregation pipeline. The JavaScript function provides great flexibility for the developer, but due to the use of JavaScript, the performance of the map-reduce function is not as efficient as the aggregation pipeline.

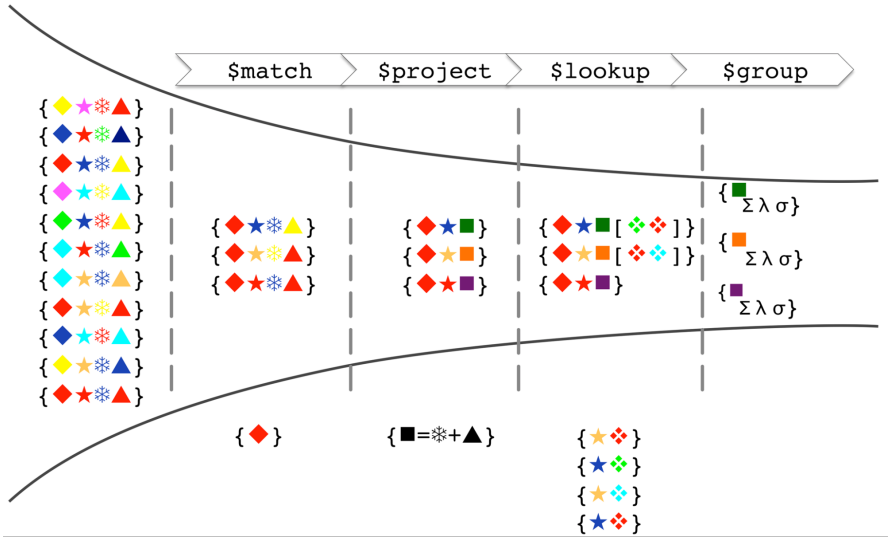

Figure 3.16: MongoDB: Aggregation pipeline [mon18b]

[mon18b]

Indices in MongoDB are used to increase the performance of queries. If a query accesses a non-indexed field of a document, MongoDB has to perform a full collection scan. The collection scan iterates over all documents in the collection, to check if that document matches the query. The index reduces the number of documents, which have to be scanned, hence the query performance increases.

The _id field of a MongoDB collection is always associated with an index and it ensures that the identifier of a document in a collection is unique. This special index cannot be deleted. Besides this simple index, MongoDB provides many different indices and most of them are built for special use cases or data types. These are some examples of available indices in MongoDB: Single Field Indices, Multikey Indices, Text Indices, 2d Indices, geoHaystack Indices and Hashed Indices.

3.4.4 Neo4j

Neo4j, like other graph data stores, use well-established concepts from graph theory to efficiently query data. One of the key concepts is graph traversal. Graph traversal refers to the process of visiting nodes in the graph by moving from one node to another node using relationships connecting these nodes. This technique is unique to graph data stores and is fundamental to data retrieval. The main difference between traditional querying, like SQL, and graph traversal is, that querying data with graph traversal only takes data that is required for the query. In contrast to SQL operations, like joins, mostly perform expensive data grouping and selection on the whole data set. Generally, the query performance for SQL-based DBMSs degrades with the depth of the query. In

contrast, Neo4j knows which nodes have already been visited and can efficiently traverse the graph. Neo4j will simply traverse and collect nodes until there are no more nodes to visit.

To perform these special queries, Neo4j provides its own query language called Cypher. Cypher is a declarative query language for graph data. It uses a matching mechanism to express patterns, which are used to select data from the graph. It is designed to visually represent graph patterns using an ASCII-Art syntax in the form `(node_a) -[:RELATIONSHIP]-> (node_b)`. `(node_a)` and `(node_b)` represent nodes in the graph and `node_a` and `node_b` are variable names, which can be used to ref to them later. The arrow `-[:RELATIONSHIP]->` between them represents the relationship connecting these two nodes. Both nodes and relationships can be further refined to match a more specific structure in the graph. Additionally, Cypher supports a SQL-like WHERE clause to filter relevant elements.

These expressions are building blocks for more complex patterns. A query can consist of multiple patterns, the resulting nodes have to match all containing patterns. It is possible to create chains of multiple nodes and relationships to describe expressive structures. For example:

```
(employee)-[:WORKS_WITH]-(co_worker)-[:WORKS_ON]-(projects)
```

This example matches the projects of the co-workers of an employee. It is also possible to query a node by ID. This node can be used as a starting point for a pattern. In the previous example, the employee could be queried by id to match the project of the co-worker for a specific employee. Generally, querying in Neo4j follows this approach, first find one or multiple starting nodes and after that perform a graph traversal.

Cypher is inspired by many query languages, like SQL and SPARQL [spa18]. Most of the keywords especially WHERE and ORDER BY are from SQL. Cypher queries are built using clauses. Clauses produce intermediate result sets, which can be chained together. The main clauses are:

- MATCH - The MATCH clause uses graph patterns to select a subset of the graph.
- WHERE - The WHERE clause is a part of the MATCH clause and filters the result and adds constraints to a pattern.
- RETURN - This clause defines, which parts of the patterns should be returned. Nodes, relationships and properties can be used.

Patterns can also be used to update the graph. Following clauses can be used:

- CREATE / DELETE - These clauses can be used to create/delete nodes and relationships.

- SET - Sets and adds properties on nodes.
- REMOVE - Removes properties from nodes.
- MERGE - Match existing nodes or create new nodes and patterns.

Neo4j provides multiple methods to find a starting node. First, as mentioned, a node can be looked up directly by ID. In the same way, multiple nodes can be retrieved from the graph by providing a list of IDs. But it is also possible to find nodes by using the properties of a node. The recommended way to lookup nodes is by using Lucene indices. Lucene is the default implementation of Neo4j's index management. An index in Neo4j is a redundant copy of information that is used to retrieve data in an efficient way. Indices are created at the cost of additional space and slower write. The choice of what to index is very important because if they are used in a wrong way, it can lead to performance issues. In Neo4j indices can be created for a property of a node or a relationship. The index contains all references to nodes or relationships containing this property. An index is identified by unique string value and can be accessed using a designated index manager. After an index is created it is managed by the system, this includes automatic updates if the graph changes.

3.4.5 VoltDB

Generally, VoltDB uses SQL as its main query and data manipulation interface, but as most database systems VoltDB does not fully support the SQL standard. The following section lists the main limitations of VoltDB's implementation according to their documentation:

- Partitioned tables can only be joined using the partitioning column. Joining two partitioned tables on non-partitioned columns or on a range of values is not supported. This restriction does not apply to replicated tables.
- Subqueries in the ORDER BY, GROUP BY or HAVING clauses are not supported. Statements of the data manipulation language (INSERT, UPDATE, DELETE and UPSERT) also do not support subqueries.
- If subqueries are used in selection or boolean expressions, then partitioned tables are not supported in these subqueries. In the FROM clause of a select statement both, replicated and partitioned tables can be used.
- The logical OR operator is supported syntactically, but it is not optimized and can lead to performance issues.
- Triggers are not supported.

- CREATE INDEX: The uniqueness constraint is enforced separately within each partition. This means that only indices on partitioned tables containing the partitioning column or replicated tables can ensure global uniqueness.
- CREATE TABLE: CHECK and FOREIGN KEY constraints are not supported, also VoltDB does not support AUTO INCREMENT.
- Partitioning columns must not contain NULL values. A partitioning column without a NOT NULL constraint will result in an error.

In addition to SQL VoltDB supports following interfaces and APIs: C#, C++, Erlang, Go, Java (packaged with VoltDB), JDBC (packaged with VoltDB), JSON HTTP (packaged with VoltDB), Node.js, PHP and Python.

3.4.6 CockroachDB

CockroachDB supports a large portion of the ANSI SQL standard. The following list contains the main limitations of CockroachDB's SQL implementation. CockroachDB also provides a full list of supported SQL features [coc18b].

- Multiple indices for a single query to filter the table's values is not supported.
- Full-text indices are not supported.
- Prefix/Expression and geospatial indices are not supported.
- A column using a SET or ENUM can only contain defined values.
- Correlated subqueries are not supported, but non-correlated subqueries are supported.
- The current version of CockroachDB (Version 2.0) is undergoing major changes to improve the performance of queries using joins and subqueries.
- Inner joins are generally processed more efficiently than outer joins.
- Joins where no index can be used to satisfy a join, loads all rows in memory. This can cause performance issues.
- Scalar subqueries disable the distributed execution of a query.
- The results of scalar subqueries are loaded entirely during the execution of the surrounding query. This can lead to memory issues if the subqueries return a large result.

Additionally, CockroachDB supports the PostgreSQL wire protocol. Many drivers such as PostgreSQL ORMs, GORM (Go) and Hibernate (Java) support this protocol. Furthermore, CockroachDB supports the following client languages: Go, Python, Ruby, Java, Node.js, C, C++, C# (.NET), Clojure, PHP and Rust.

DistSQL CockroachDB developed an optimization tool called DistSQL [dis18] used for distributed queries. In a non-distributed environment, the node receiving a query has all the information it needs to return the corresponding result. DistSQL performs certain operations on each node to reduce network traffic. The coordinating node, which receives the query, delegates the workload to the nodes, which contain parts of the data. Each node does computations on the rows it contains and instead of returning the entire rows only the computed results are returned. The coordinating node then collects and aggregates the results and sends the response to the client.

3.4.7 NuoDB

NuoDB is ANSI SQL92 compliant with SQL99 extensions and works with many popular ORM tools such as Hibernate, PDO, Active Record and supports a set of JDBC/ODBC drivers. With Release 2.0, NuoDB also supports SQL and Java-based Stored Procedures.

In contrast to the other NewSQL database, NuoDB has no considerable limitations regarding the SQL standard!

NuoDB supports many client languages including JDBC, Python, PHP PDO and NuoDB's C++ API and C API. In addition, NuoDB provides a Hibernate dialect and a Java class to do the mapping, which sits on top of NuoDB's JDBC driver.

3.4.8 Summary

The querying capabilities and techniques of the inspected data stores, especially the NoSQL data stores, are as diverse as their data models. The first evaluated data store was Redis. Redis uses a rather simple data model compared to the other NoSQL data stores. Mostly all operations revolve around accessing data by key. It offers a simple but very efficient API. In contrast to Redis, Cassandra implements its own query language, called CQL. It is inspired by SQL but not as powerful as SQL. A drawback of most NoSQL data stores, including Cassandra, is the lack of a join operator to create complex queries. MongoDB surprises with very rich querying capabilities for a NoSQL data store. On the one side, MongoDB supports queries on non-key fields, which go beyond exact matching of values. More important, MongoDB's aggregation pipeline offers powerful querying possibilities similar to SQL. Like Cassandra, Neo4j also provides its own querying language, called Cypher. Cypher is built to query graph data using a pattern-based approach. In conclusion, each NoSQL data store offers very unique querying capabilities tailored to their individual data model.

The NewSQL data stores don't differ in their querying capabilities like the NoSQL data stores. All of them support SQL as their main querying interface, but they do not support SQL to the same degree. Although, all of them support the main SQL statements (SELECT, INSERT, UPDATE and DELETE). VoltDB has the most limited SQL implementation compared to CockroachDB and NuoDB. Most of these limitations revolve around partitioned tables and subqueries. CockroachDB has just some minor SQL limitations and performance issues in some edge cases. Most noticeable, NuoDB

has no considerable limitations regarding the SQL standard. Both CockroachDB and NuoDB provide a detailed list of all supported SQL features.

3.5 Concurrency control

Concurrency control in NoSQL and NewSQL data stores addresses conflicts caused by simultaneous access and manipulation of data. They implement mechanisms to coordinate multiple concurrent transactions or users accessing the same resources and try to maintain consistency and integrity of the data store.

Generally, NoSQL data stores follow a weaker approach to consistency compared to NewSQL systems. Therefore, NoSQL data stores implement in most cases looser concurrency control mechanisms. Weaker concurrency control mechanisms allow NoSQL data stores to focus on performance and availability because concurrency control typically implies higher latency due to the coordination of multiple transactions. A widespread approach for NoSQL systems is read/write locking.

NewSQL systems are ACID compatible and follow a stronger approach to consistency. This means that NewSQL systems also implement strong concurrency control mechanisms to avoid inconsistency. In comparison to NoSQL data stores, most NewSQL systems provide multiple isolation levels to offer a more granular control to the isolation of concurrent transactions. A popular approach for NewSQL systems is MVCC.

3.5.1 Redis

As most NoSQL data stores, Redis supports elementary features regarding concurrency control. A transaction in Redis allows for an execution of multiple commands in a single operation. Transactions in Redis have the following properties:

- Commands in a transaction are executed sequentially. It is not possible that a command from another client is executed in the middle of a transaction. This means that the commands of a transaction are serialized and executed as a single isolated operation.
- Transactions are atomic, either all commands of a transaction are executed or none of them.
- Redis does not support rollbacks. If an error occurs during a transaction, Redis will still execute the rest of the transaction.

Redis concurrency control features are based on the following commands. MULTI, EXEC, DISCARD and WATCH. MULTI and EXEC are used to declare the scope of a transaction. MULTI marks the start of a transaction and the following commands are queued for later execution. The EXEC command closes the scope of a transaction and executes all queued commands. The DISCARD command can be used to clear the queued commands and abort the execution of the transaction.

As mentioned before, rollbacks are not supported by Redis. Commands that fail during a transaction do not cancel the following commands or rollback the transaction. If a

transaction is started, all commands are executed, even if commands fail. Redis argues that this is not as severe as it seems, because commands can only fail if called with a wrong syntax or if called with a wrong data type. These kinds of errors can be easily detected and fixed during development. Not supporting rollbacks enables Redis to use a simplified transaction engine, which results in a better overall performance.

Along with version 2.2, Redis introduced optimistic locking using a check-and-set mechanism. This feature is implemented by the WATCH command. The WATCH command is used to monitor particular keys. If at least one watched key is modified by another client, the following transaction will be cancelled, so the WATCH command makes the execution of a transaction conditional.

For example, an increment of an integer value can be done in three steps, read the value, increment the value and write the incremented value back. An unfortunate scheduling of commands issued by multiple clients can lead to a race condition and an unintended result. By using the check-and-set mechanism, illustrated in Listing 3.1, the value can be incremented without the risk of other clients interfering in between the operations.

```
WATCH key
value = GET key
value = value + 1
MULTI
SET key $value
EXEC
```

Listing 3.1: Example: Redis check-and-set

3.5.2 Cassandra

Similar to Redis, Cassandra's concurrency control mechanisms are very limited compared to traditional RDBMSs. As a NoSQL data store, Cassandra follows the approach of eventual consistency. Within this paradigm, Cassandra implements atomic, isolated and durable transactions.

A write operation in Cassandra is atomic, but only on the partition level. Insertions, updates and deletions of multiple rows in the same partition are executed as one write operation. If a data item is replicated over multiple nodes, the write operations are forwarded to the replicas. If a write operation on one of the replicas fails, the other successful write operations are not rolled back automatically. This leads to inconsistent values among the replicas.

If multiple clients update the same column, client-side timestamps are used to determine the most recent update. For concurrent writes to the same value, the write operation with the latest timestamp wins.

Write and delete operations in Cassandra are isolated on row-level, but this only applies to a row within a single partition on a single node. This means that a write operation

in this scope is only visible to the client performing the operation until it is completed. Cassandra also provides batch operations. Batch operations within the same partition also have these properties, but if a batch operation includes more than one partition it is not isolated.

Furthermore, write operations in Cassandra are durable. Before a replica acknowledges a write operation it is stored in memory and in a commit log on disk. The in-memory data is periodically persisted on disk. If a system failure occurs before the data could be persisted to disk, the commit log is used to replay the lost write operations to recover lost data.

To counteract eventual consistency, Cassandra runs repair operations to minimize inconsistent data in the distributed system. Because Cassandra implements a looser consistency model, it is able to provide higher availability and performance. Furthermore, Cassandra offers the possibility to configure this trade-off between consistency and availability to some extent. Therefore, two consistency types are supported:

- **Tuneable consistency** - The tuneable consistency model offers multiple read and write consistency levels. The consistency levels can be configured globally or separately for each operation. For reading operations, the read consistency level determines the number of replicas, which have to respond and acknowledge the read operations before the result can be returned successfully. If a read operation exposes inconsistent datasets between replicas, Cassandra initiates a repair job to adjust the issue. The write consistency level determines the number of replicas, which have to respond to the request, in order to consider the operation successful. These consistency levels enable the developer to choose a proper consistency level depending on the requirements of the application. The higher the consistency level, the more replicas have to respond to the request. This means higher consistency results in higher latency. This trade-off between availability and consistency has its origin in the CAP theorem. Independent of the consistency level, Cassandra forwards write operations to all replicas responsible for the particular partition. However, Cassandra is an AP system in the context of the CAP theorem and it cannot be tuned to a CA System.
- **Linearizable consistency** - Compared to RDBMSs, Cassandra does not use traditional techniques like locking to control concurrent operations. Cassandra employs a mechanism, called lightweight transactions. Lightweight transactions are used to execute a sequence of operations. It is not possible that another client's operation is executed in-between operations of this sequence. Cassandra achieves lightweight transactions using the Paxos consensus protocol [GL06]. Paxos is used to establish consensus in a distributed network, which uses an unreliable communication medium, so Cassandra uses Paxos to agree on one result among a group of replicas. A series of four phases is implemented to achieve this: Prepare/Promise, Read/Results, Propose/Accept, Commit/Acknowledge. These four phases require four round trips between a node proposing a transaction and a group of replicas.

During this communication between the proposing node and the replicas Cassandra tries to establish a consensus. If all conditions are met, the transaction can be committed and acknowledged.

3.5.3 MongoDB

MongoDB puts a higher focus on consistency than availability compared most NoSQL systems. MongoDB uses locking as its main concurrency control mechanism. It ensures that an operation on a single document is atomic. Atomicity for a single document is in most cases sufficient because a document can contain embedded documents and arrays to describe relationships between records in a single document. But MongoDB also offers the possibility to perform transactions on multiple documents against replica sets. Multi-document transactions can also be used over multiple collections and databases. MongoDB also guarantees that all operations in multi-document transactions are executed or none of them are. If an operation in the transaction fails, the transaction is cancelled and all changes are rolled back, without any pre-committed data being visible to other clients. In addition, no changes are visible to other clients until the transaction is committed successfully. Furthermore, it is to mention that multi-document transactions have a greater impact on performance than single document writes. MongoDB recommends avoiding multi-document transaction. Multi-document transactions should not be a replacement for a suitable schema design.

MongoDB uses read and write locks to allow multiple clients concurrent read access and exclusive write access to a resource. In summary, MongoDB offers four types of lock modes:

- Shared lock
- Exclusive lock
- Intent shared lock
- Intent exclusive lock

The shared lock is used to grant shared read access and the exclusive lock is used for write operations. In addition to these two lock modes, MongoDB provides the intent shared and the intent exclusive lock mode. These modes indicate an intent to read or write a resource with a finer granularity. If a resource is locked using a shared or exclusive lock all higher level resources are automatically locked with the corresponding intent lock. For example, if a collection is locked exclusively the containing database and the global level are locked in intent exclusive mode. Resources can be simultaneously locked in intent shared and intent exclusive mode, an intent shared lock can coexist with a shared lock, but an exclusive lock cannot coexist with any other locks.

MongoDB also handles concurrent operations on distributed collections. As mentioned before, an operation on a single document is atomic. This means that such an operation

concerns only a single shard, so sharding improves concurrency because multiple mongod instances can perform operations in parallel. Locks are applied to each individual shard and not to the whole system. Operations on one instance do not block the operations on any other instance, because mongod instances are independent of each other and each instance uses its own locks.

In combination with replica sets a write operation to a collection on a primary also triggers a new log entry. This log is a special collection in the local database. For that reason, MongoDB locks both the database of the collection and the local database at the same time to keep the data consistent. The log in the local database is used to keep the secondaries up-to-date. During the update phase, secondaries do not allow read operations to prevent inconsistent data from being read.

MongoDB provides so-called read concern levels to control the consistency and isolation level of data read from shards and replica sets. For write operations, MongoDB provides multiple write concern levels which control the level of acknowledgement. Both, read and write concern levels are especially relevant if MongoDB uses replica sets or a sharded cluster. Generally, these concern levels adjust the trade-off between availability and consistency, stronger consistency requirements imply higher latency and therefore lower availability.

The lowest read concern level returns data without the guarantee that the data has been written to a majority of the replica set. Without this guarantee, data may be rolled back after it has been read. Higher read concern levels increase the consistency level by using data that has been approved by a majority of the replica set, so the read data is considered durable even in the event of failure.

For the write concern level, a specific number of instances can be specified, which have to acknowledge the write operation, before it can be considered successful. A write operation can be acknowledged after applying the write in memory or after writing to the on-disk log. Additionally, MongoDB provides an option to use the majority of the replica set.

Combing all these options, MongoDB provides extensive configuration possibilities regarding concurrency control. Compared to other NoSQL systems MongoDB's default configuration is more aimed to consistency than to availability.

3.5.4 Neo4j

Neo4j is one of the few NoSQL data stores which is ACID compliant if operated on a single instance. Therefore, if Neo4j is configured without a cluster, it provides the same guarantees as a traditional DBMS. If operated on a single instance Neo4j offers transactions, to manipulate multiple graph entities like nodes and relationships at once. New transactions in Neo4j are executed on the current thread. Therefore, it is crucial to properly finish each transaction. Alternatively, transactions executed on the same thread may affect each other and cause inconsistent data. If a transaction in Neo4j completes

successfully, the changes can be considered durable, meaning that the changes are written to disk. Not every transaction is immediately persisted on disk, but a transaction log records all mutating operations to replay them in an event of failure.

In Neo4j all database operations which access the data graph, schema or indices must be performed in a transaction. Transactions can also be nested in so-called nested transactions. Each nested transaction is added to the scope of the top level transaction. A failure in a nested transaction can lead to a rollback in the top level transaction if configured.

As many other data stores, Neo4j uses locks to manage concurrent access to the same resource. Read locks are used to lock a resource, without any other client being able to modify the resource. They can be acquired on every resource if there is no write lock currently active. Furthermore, read locks are not mutually exclusive, multiple read locks can be concurrently active on the same resource. Write locks cannot be combined with other locks and a write lock can only be acquired on a resource if there is no active lock. By default, transactions do not use read locks automatically. This means that transactions see the latest committed state of a resource. On the other hand, write locks are acquired automatically and they are held until the transaction finishes. Compared to the isolation levels traditional DBMSs use, this behaviour is similar to the read committed isolation level. In order to achieve a higher level of consistency, locks have to be managed manually by the developer.

Locks can also be managed by the developer, not only by the system, this allows for a flexible lock configuration. Depending on the requirements locks can be used to trade off consistency for availability and vice versa. The careless use of locks can result in deadlocks. To counteract this problem, Neo4j provides a built-in detection mechanism for deadlocks. If the mechanism detects a deadlock, an exception in the particular transaction is thrown. This exception cancels and rolls back the transaction.

As mentioned before, these concurrency control mechanisms are only valid for a setup with a single Neo4j instance. If Neo4j is configured to operate as a cluster, the consistency model weakens and becomes eventual consistency. In a distributed environment Neo4j is not ACID compliant.

3.5.5 VoltDB

VoltDB uses a unique approach to concurrency control. Operations in VoltDB are serialized on a single thread. More specific, each instance in a distributed setup can act independently, is single threaded and contains a queue of transaction requests. This means that there is no need for concurrency control and VoltDB avoids time-consuming concurrency control mechanism like locking. All requests are buffered in a queue and executed sequentially and exclusively against the data, so there are no concurrent transactions.

A transaction that only accesses a single partition, enables each VoltDB engine in a cluster to act autonomously. Transactions that access multiple partitions are more complex.

One particular VoltDB engine distributes and manages the assigned transaction for the other engines.

Transactions and stored procedures are one and the same thing in VoltDB. For each stored procedure VoltDB automatically provides ACID guarantees. Therefore, stored procedures either succeed or automatically roll back in an event of failure. When data is manipulated using a stored procedure, it is guaranteed to stay consistent, because all stored procedures modify the database isolated from each other. If a transaction succeeds it is guaranteed that the changes are durable and visible for the next transaction.

Stored procedures in VoltDB are written in Java. To execute a stored procedure, the Java class containing the stored procedure has to be compiled and the resulting jar file has to be loaded into the database. In order to maintain consistency and durability, a stored procedure has to be deterministic. Given specific input parameters, a stored procedure has to return a consistent and predictable output. This property is crucial for VoltDB because this determinism makes it possible to run multiple redundant copies of a partition without sacrificing performance. This also applies to SQL queries used in stored procedures. For example, SQL queries without an ORDER BY clause are not guaranteed to be in a specific order. Hence, SQL queries used in a stored procedure should always use an ORDER BY clause, to avoid a non-deterministic behaviour. Transactions rely on a rollback mechanism if an exception arises. As stored procedures are implemented in Java, VoltDB uses Java's exception handling to perform rollbacks.

VoltDB avoids time-consuming concurrency control mechanisms by using this single-threaded approach. This unique concept takes its toll on usability. There are restrictions for many aspects in VoltDB, such as schema design, sharding and querying to provide consistency across the whole cluster.

3.5.6 CockroachDB

CockroachDB attaches great importance to consistency and concurrency control. To support this features CockroachDB implements full support for the ACID properties. CockroachDB's architecture contains a designated layer, the transaction layer, to manage concurrent transactions and consistency.

CockroachDB implements an isolation level called Serializable Snapshot. It is an optimistic concurrency control system which is based on MVCC. It guarantees that the execution of the concurrent transactions is equivalent to a serial execution. Furthermore, the system is recoverable, which means that a set of aborted transactions has no effect on the state of the database and a two-phase commit system makes sure that individual transactions are recoverable. Additionally, the CockroachDB is able to recover any combination of transactions. This is achieved without the use of locks. Furthermore, CockroachDB‘s concurrency control is organized without a central coordinator and no single point of failure.

All these features regarding concurrency control are based on MVCC in combination with HLC (hybrid logical clock) [KDM+14] timestamps. Hybrid logical clocks were

introduced to order events in an asynchronous distributed system. They captured the causality relationship like logical clocks and enable identification of consistent snapshots in distributed systems. This is possible by combining physical and logical clocks. Every transaction in CockroachDB is assigned with a HLC timestamp when it passes the gateway node. This HLC timestamp is used to track versions of records for MVCC. To guarantee consistency in a cluster, CockroachDB needs a certain level of clock synchronization to keep data consistent. If a node detects that its clock is out of sync given a certain threshold, it crashes immediately. This precaution is necessary to avoid the risk of inconsistent data.

All operations in CockroachDB are executed as transactions, also single statements. In order to support cluster wide transactions, CockroachDB uses a two-phase commit process.

The first phase covers read and write operations. A write transaction is not immediately persisted on disk. Instead, two temporary objects are created to manage the distributed execution of the transaction: A transaction record and a write intent. First, a transaction record is created on the shard where the first write operation occurs. It represents the transactions current state. When a transaction has started, it is in the state pending and after it is finished, it is either in the state committed or aborted. A write intent is essentially the same as an MVCC record, but additionally has a pointer to the transaction record. When a write intent is created, CockroachDB restarts the transaction if newer committed values exist. After the write operations are successfully processed, read operations are executed. Both read and write operations can encounter conflictual data. If this is the case CockroachDB initiates a conflict resolution process.

The second phase is essentially a check if any of the running transaction has been aborted. If a transaction is in the state aborted, it is restarted. If it is not in the state aborted, then the state is set to committed and the client is notified that the transaction completed successfully. The write intents of committed transactions are then converted to MVCC values. After that, the two-phase commit process is completed and CockroachDB initiates a third asynchronous phase which cleans up the temporary objects.

Compared to the ANSI isolation levels (Read Uncommitted, Read Committed, Repeatable Read and Serializable) CockroachDB only supports the highest isolation level Serializable and lower isolation levels are automatically upgraded to Serializable. CockroachDB does not support weaker isolation levels to prevent concurrency-based attacks. Warszawski and Bailis demonstrate in [WB17] that weaker isolation levels are vulnerable to concurrency-based attacks.

The isolation level Serializable Snapshot is the default isolation level in CockroachDB, it is the correspondent to the ANSI isolation level Serializable. It enforces the client to retry transactions if serializability constraints are at risk. Additionally, CockroachDB provides a weaker isolation level called Snapshot. This isolation level aims to reduce retries of transactions. Data is read using the initial transaction timestamp, but at this isolation level, CockroachDB allows the transaction to push the commit timestamp forward in the

event of a transaction conflict.

3.5.7 NuoDB

NuoDB provides ACID-compliant transactions and focuses on consistency in a distributed environment by using its own two-tier architecture. The transactional layer of its two-tier architecture manages atomicity, consistency and isolation. It is a pure in-memory layer and it has no effect on durability. The storage layer is responsible for data durability. NuoDB uses MVCC to provide ACID-compliant transactions and manage conflicts between concurrent transactions. All update and delete operations add a new version to a database record. The transaction engines hold these multi-versioned records in a cache until they are persisted by the storage layer. Generally, the transaction engines do not provide durability and operate only as a cache. Until a transaction is completed successfully new versions of a record are pending. This information is used to detect conflicts of concurrent transactions.

Transactions in NuoDB implement the default behaviour. If a transaction fails it is automatically rolled back. In NuoDB each statement starts its own transaction. By default, all statements in NuoDB are executed as transactions. This transaction mode is called implicit. It is not necessary to explicitly start and commit a transaction. NuoDB also implements an explicit transaction mode for more flexibility.

Based on MVCC NuoDB implements three isolation levels to provide a consistent view of the stored data. Because NuoDB uses MVCC instead of a locking technique, reading transactions do not block writing transactions and writing transactions do not block reading transactions. The weakest consistency level is Read Committed. This isolation level provides no isolation guarantees. A transaction using the Read Committed isolation level always reads the most recently committed version of a record. The system takes a snapshot of the database at the beginning of each statement in the transaction. This means that transactions using this isolation level can read changes committed by other transactions because a new snapshot is read before each execution of a new statement.

The strongest consistency level is Consistent Read. Compared to the Read Committed isolation level a transaction configured with Consistent Read does not read a snapshot before each statement, but once at the start of the transaction. This means that the transactions see a snapshot of the database independently of other transactions. If a transaction encounters a record which has been recently updated by another transaction and the version is still pending, then the transaction waits until the other transaction completes. Only if the transaction, which updated the record first, fails then the pending version of the record becomes obsolete and the waiting transaction can succeed in its update. Alternatively, the pending version becomes commit and the waiting transaction gets an error.

NuoDB also provides a third isolation level, between Read Committed and Consistent Read, called Write Committed. This isolation level provides the same behaviour as

Consistent Read for reading operations and the same behaviour as Read Committed for write operations. The release of NuoDB 3.2 deprecated this isolation level.

The different isolation levels provide a clear visibility model for NuoDB. The isolation level Consistent Read offers a consistent view of the database captured at the moment the transaction started. Multiple transactions see overlapping snapshots of the database depending on the time the transaction started. In a distributed environment without a central coordinator, this approach provides clear visibility and isolation between concurrent transactions.

3.5.8 Summary

To conclude, the inspected NoSQL data stores provide various concurrency control techniques: Redis provides only basic mechanisms to manage concurrent requests, as the atomic execution of multiple commands and optimistic locking. Transactions in Redis guarantee atomicity and the sequential execution of commands, but Redis does not support rollbacks. If a command fails, the subsequent transactions are still executed. Cassandra provides a flexible approach to concurrency control. Write operations in Cassandra are atomic, but only on the partition level. So if a transaction targets multiple partitions, then atomicity is not guaranteed, which can lead to temporary inconsistent data. This applies generally to NoSQL data stores. They provide stronger concurrency control mechanisms for transactions which operate only on one instance. Casandra also offers a tuneable consistency model with multiple read and write consistency levels. They can be used to control the trade-off between consistency and availability. MongoDB uses a flexible locking mechanism to manage concurrent operations with four different lock types. Also, MongoDB offers a possibility to control the level of consistency by configuring the read and write concern levels. MongoDB's default configuration is more focused on consistency than on availability. Neo4j claims to be fully ACID compliant, but this only holds true if Neo4j is operated on a single instance. If Neo4j is configured as a cluster, the consistency model becomes eventual consistency. In this setup, Neo4j is not ACID compliant.

NoSQL data stores are not only about availability, all of the inspected data stores provide some kind of concurrency control mechanism. Also, the concept of a transaction is widely used by NoSQL data stores, but in most cases, it can not be compared to traditional DBMSs or NewSQL systems, because frequently they redefine the concept of a transaction.

The compared NewSQL database systems are not as different as the NoSQL systems. All of them provide ACID compliant transactions in a distributed environment. Both CockroachDB and NuoDB use MVCC to handle concurrent transactions. Additionally, CockroachDB uses hybrid logical locks to order events in a distributed system. Compared to NuoDB, CockroachDB implements only one isolation level, but this isolation level is equal to the strongest ANSI isolation level Serializable. NuoDB implements three isolation levels, Read Committed, Write Committed and Consistent Read. Consistent

Read is the strongest consistency level and equal to the ANSI isolation level Serializable. VoltDB uses a unique approach to concurrency control. It avoids concurrency control mechanisms by using a single threaded approach. This means that all transactions are serialized on one thread and there is no need for concurrency control. But the approach of VoltDB has negative side effects in terms of schema design, sharding and querying.

CHAPTER 4

Conclusions

NoSQL and NewSQL data stores provide powerful new features and approaches. This thesis analyzed the methods and algorithms used by NoSQL and NewSQL systems. The resulting comparison provides guidance for application developers searching for a fitting data store.

At the beginning of the thesis, new challenges of traditional DBMSs have been discussed. Modern applications have high requirements for performance, scalability and consistency. Traditional DBMSs cannot keep up with these requirements. Today's database landscape offers great diversity and many systems are tailored for specific requirements. NoSQL and NewSQL systems face these challenges and using various approaches and design principles.

The NoSQL domain comprises a huge number of distinct systems. They focus on scalability, flexibility, performance and new data models. Generally, they follow the BASE principles: basically available, soft state and eventual consistency. These properties emphasise the focus on availability and less on consistency. They are built to operate in a distributed environment and utilize multiple server instances. Most of them follow novel approaches based on specific data models. They do not use fixed schemas like relational DBMSs. This enables NoSQL data stores to operate with semi-structured and schema-less data. They are categorised into four NoSQL families: Key-value stores, column stores, document stores and graph data stores.

The class of NewSQL systems questions the design decisions and assumptions of traditional DBMSs. While NoSQL systems do not use SQL, NewSQL systems use SQL as their main querying language. Furthermore, NewSQL systems use a relational data model and support the ACID (Atomicity, Consistency, Isolation and Durability) properties. They support a similar feature set as traditional DBMSs. The big difference is that NewSQL databases are built to scale like NoSQL systems while maintaining consistency and data integrity. The NewSQL domain does not offer as many distinct systems as the NoSQL

domain. NewSQL systems can be grouped into three categories: New Architectures, Transparent Sharding Middleware, Database-as-a-Service. New architectures represent the most innovative category of NewSQL systems.

Four NoSQL (Redis, Cassandra, MongoDB and Neo4j) and three NewSQL (VoltDB, CockroachDB and NuoDB) data stores were selected for the comparison. The following five key criteria were used, to compare the selected NoSQL and NewSQL data stores: Architecture and general principles, Sharding, Replication, Querying, and Concurrency control. These key criteria refer to the first research question: *What are the key criteria to compare NoSQL and NewSQL systems?* The second and third research questions (*In what manner do particular NoSQL and NewSQL systems compare according to these criteria?, Which different methods and algorithms are used by NoSQL and NewSQL systems to cope with these key criteria?*) were answered in Chapter 3.

The chapter started with a comparison of the architectures and general principles used by the data selected data stores. The architecture of a NoSQL system is greatly influenced by the underlying data model. The data models promote flexibility and influence many other design aspects. NewSQL data stores implement multiple abstraction levels to separate different aspects like SQL, sharding and concurrency control. The consistency requirements of NewSQL systems are deeply rooted in their architectures.
The sharding approaches used by the selected systems are as different as their data models. The possibilities to shard data across multiple nodes heavily rely on how data is modelled and stored. Most NoSQL data stores implement master-slave replication strategies, where read operations are allowed on masters and slaves. This emphasizes the performance goal of NoSQL systems. NewSQL systems use in the most cases a multi-master replication approach and synchronous replication to keep data consistent. Generally, almost all NoSQL and NewSQL systems offer configuration options to influence consistency and availability.
In terms of querying capabilities, NoSQL systems do not provide as rich possibilities as NewSQL systems. Many systems use a proprietary query language to access the data store. On the other hand, NewSQL systems use SQL as their main querying interface. They support the majority of available SQL statements. VoltDB has the most limited SQL implementation compared to CockroachDB and NuoDB.
NoSQL systems implement relative basic concurrency control mechanisms compared to NewSQL systems. However, NoSQL systems provide configuration options to increase consistency in the exchange for availability. Most NoSQL concurrency control approaches are based on read/write locks. NewSQL systems use lock-free concurrency control mechanisms. VoltDB uses a unique single-threaded approach to avoid concurrency control. The other NewSQL systems use multi-version concurrency control (MVCC).

Table 4.1 provides an overview of all compared data stores and key criteria.

NoSQL and NewSQL data stores provide powerful features and there are many systems to choose from. This rich set of options presents a huge challenge to developers choosing a fitting data store. Knowing the requirements of a system is essential because there is no general-purpose DBMS, which fits all needs. This thesis presented multiple NoSQL

and NewSQL data stores providing a comprehensive comparison in multiple dimensions. The sheer number of data stores and comparison possibilities leaves enough room for further research in this area. Especially, NewSQL systems are constantly developing and new systems will be released. Especially, performance evaluations [KS17] under different workloads would add more information to ease the challenge of choosing a fitting data store.

	Architecture	Sharding	Replication	Querying	Concurrency control
NoSQL					
Redis	Key-value store, in-memory	Per data type sharding (Hash, Set and Zset)	Master-slave setup	Key-based querying	Atomic transactions, optimistic locking
Cassandra	Column store, wide-columns	Consistent hashing based on partitioner	Replication-factor, replication strategies	CQL (Cassandra Query Language)	Time-stamp based
MongoDB	Document store, JSON-based documents	Hashed sharding, ranged sharding	Replica set (Primary, Secondar, Arbieter)	Query by key or value, Aggregation pipeline	Read/write locks
Neo4j	Graph store, property graph model	Cache sharding (no physical data distribution)	Master-slave setup	Cypher (pattern-based query language)	Read/write locks
NewSQL					
VoltDB	SQL based, single-threaded	Table partitioning by key	Replicated tables, database replication	SQL (join and subquery restrictions)	Single-threaded (no concurrency control)
CockroachDB	SQL based, layered-architecture	Automatic sharding of ranges (data chunks)	Multi-master setup	SQL, DistSQL (optimization tool for distributed queries)	MVCC
NuoDB	SQL based, two-tier architecture	Table partitioning by range or by list	Multi-master setup	ANSI SQL92 compliant with SQL99 extension	MVCC

Table 4.1: Overview of all data stores and key criteria

List of Figures

List of Tables

Bibliography

[agi18] AgilData Scalable Cluster. http://www.agildata.com/scalable-cluster-for-mysql/, 2018. [Online; accessed 14.07.2018].

[App08] Austin Appleby. Murmurhash. *URL https://sites. google. com/site/murmurhash*, 2008.

[Asl11] Matthew Aslett. How will the database incumbents respond to NoSQL and NewSQL. *San Francisco, The*, 451:1–5, 2011.

[aur18] Amazon Aurora. https://aws.amazon.com/rds/aurora/, 2018. [Online; accessed 14.07.2018].

[BG83] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[Bre00] Eric A. Brewer. Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[Bro09] Julian Browne. Brewer's CAP theorem. *J. Browne blog*, 2009.

[Car13] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[cas18a] Apache Cassandra Documentation. https://docs.datastax.com/en/cassandra/latest/index.html, 2018. [Online; accessed 16.03.2018].

[cas18b] Cassandra. http://cassandra.apache.org, 2018. [Online; accessed 23.02.2018].

[cas18c] Cassandra: Indexing. https://docs.datastax.com/en/cql/3.3/cql/cql_using/usePrimaryIndex.html, 2018. [Online; accessed 19.04.2018].

[cas18d] Cassandra: Querying tables. https://docs.datastax.com/en/cql/3.3/cql/cql_using/useQueryDataTOC.html, 2018. [Online; accessed 19.04.2018].

[cas18e] Cassandra Replication. https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html, 2018. [Online; accessed 06.04.2018].

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[Cho13] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2013.

[cle18] ClearDB. http://w2.cleardb.net/aws/, 2018. [Online; accessed 14.07.2018].

[clu18] Clustrix. https://www.clustrix.com/, 2018. [Online; accessed 14.07.2018].

[coc18a] CockroachDB. https://www.cockroachlabs.com/, 2018. [Online; accessed 30.03.2018].

[coc18b] CockroachDB: Detailed SQL Standard Comparison. https://www.cockroachlabs.com/docs/stable/detailed-sql-support.html, 2018. [Online; accessed 20.07.2018].

[coc18c] CockroachDB replication. https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html, 2018. [Online; accessed 08.04.2018].

[cou18a] Couchbase. https://www.couchbase.com/, 2018. [Online; accessed 13.07.2018].

[cou18b] CouchDB. http://couchdb.apache.org/, 2018. [Online; accessed 13.07.2018].

[dbe18] DB-Engines. https://db-engines.com/, 2018. [Online; accessed 09.02.2018].

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[dis18] CockroachDB: DistSQL. https://www.cockroachlabs.com/docs/stable/architecture/sql-layer.html#distsql, 2018. [Online; accessed 20.07.2018].

[dyn18] The Raft Consensus Algorithm. https://aws.amazon.com/dynamodb/, 2018. [Online; accessed 13.07.2018].

[Eva09] Eric Evans. NOSQL. hhttp://blog.sym-link.com/2009/05/12/nosql_2009.html, 2009.

[GHTC13] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: advances, systems and applications*, 2(1):22, 2013.

[gir18] Apache Giraph. http://giraph.apache.org/, 2018. [Online; accessed 13.07.2018].

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[goo18] Google Spanner. https://cloud.google.com/spanner/, 2018. [Online; accessed 14.07.2018].

[gra18] GraphX. https://spark.apache.org/graphx/, 2018. [Online; accessed 13.07.2018].

[grp18] gRPC. https://grpc.io/, 2018. [Online; accessed 03.05.2018].

[hap17] HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org/, 2017. [Online; accessed 21.03.2018].

[hba18] HBase. https://hbase.apache.org/, 2018. [Online; accessed 17.07.2018].

[HEM18] Moshik Hershcovitch, Revital Eres, and Adam McPadden. Pm aware storage engine for mongodb. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR '18, pages 123–123, New York, NY, USA, 2018. ACM.

[Hew10] E Hewitt. *Cassandra-The Definitive Guide: Distributed Data at Web Scale. Definitive Guide Series.* O'Reilly, 2010.

[KDM+14] Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa, A Bharadwaj, and M Leone. Logical physical clocks and consistent snapshots in globally distributed databases. *State University of New York at Buffalo, Computer Science and Engineering Technical Report*, 4, 2014.

[KR04] Andreev Konstantin and H Räcke. Balanced graph partitioning. *SPAA'04*, pages 120–124, 2004.

[KS17] K. Kaur and M. Sachdeva. Performance evaluation of NewSQL databases. In *2017 International Conference on Inventive Systems and Control (ICISC)*, pages 1–5, Jan 2017.

[LCG18] Jordi Lladós, Fernando Cores, and Fernando Guirado. Scalable consistency in t-coffee through apache spark and cassandra database. *Journal of Computational Biology*, 25(8):894–906, 2018. PMID: 30004242.

[lua18] Lua. `https://www.lua.org/`, 2018. [Online; accessed 16.03.2018].

[mar18] MariaDB MaxScale. `https://mariadb.com/products/technology/maxscale`, 2018. [Online; accessed 14.07.2018].

[MBT16] José Maria Monteiro, Angelo Brayner, and Júlio Alcântara Tavares. What Comes After NoSQL? NewSQL: A New Era of Challenges in DBMS Scalable Data Processing. *Tópicos em Gerenciamento de Dados e Informações. Minicursos do XXXI Simpósio Brasileiro de Banco de Dados (SBBD)*, pages 27–56, 2016.

[MH13] ABM Moniruzzaman and Syed Akhter Hossain. NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[mon17] MongoDB Architecture Guide. `https://www.mongodb.com/collateral/mongodb-architecture-guide`, 2017. [Online; accessed 10.02.2018].

[mon18a] MongoDB. `https://www.mongodb.com`, 2018. [Online; accessed 23.02.2018].

[mon18b] MongoDB. `https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-pa` 2018. [Online; accessed 20.04.2018].

[mon18c] MongoDB query documents. `https://docs.mongodb.com/manual/tutorial/query-documents/`, 2018. [Online; accessed 20.04.2018].

[mon18d] MongoDB Replication. `https://docs.mongodb.com/manual/replication/`, 2018. [Online; accessed 06.04.2018].

[mon18e] MongoDB Sharding. `https://docs.mongodb.com/manual/sharding/`, 2018. [Online; accessed 07.04.2018].

[MPS17] Hu Meng, Yongsheng Pan, and Lang Sun. Application and Implementation of Batch File Transfer in Redis Storage. In Zhongzhi Shi, Ben Goertzel, and Jiali Feng, editors, *Intelligence Science I*, pages 228–233, Cham, 2017. Springer International Publishing.

[neo18a] Neo4j. https://neo4j.com, 2018. [Online; accessed 23.02.2018].

[neo18b] Neo4j High Availability. https://neo4j.com/docs/operations-manual/current/clustering/high-availability/architecture/, 2018. [Online; accessed 07.04.2018].

[nuo17] From Disaster Recovery to active-active: NuoDB and Multi-Data center deployments. http://go.nuodb.com/rs/139-YPK-485/images/DR-Ebook_FINAL.pdf, 2017. [Online; accessed 08.04.2018].

[nuo18a] NuoDB. https://www.nuodb.com/, 2018. [Online; accessed 30.03.2018].

[nuo18b] NuoDB commit options. http://doc.nuodb.com/Latest/Default.htm#Description-of-Values-for-commit-Database-Option.htm%3FTocPath%3DDatabase%2520Administration%7CChoosing%2520a%2520Commit%2520Protocol%7C_____2, 2018. [Online; accessed 08.04.2018].

[nuo18c] NuoDB Docs. http://doc.nuodb.com/Latest/Default.htm, 2018. [Online; accessed 30.03.2018].

[OAG17] Keren Ouaknine, Oran Agra, and Zvika Guz. Optimization of RocksDB for Redis on Flash. In *Proceedings of the International Conference on Compute and Data Analysis*, ICCDA '17, pages 155–161, New York, NY, USA, 2017. ACM.

[ora18] Oracle. https://www.oracle.com/, 2018. [Online; accessed 07.03.2018].

[PA16] Andrew Pavlo and Matthew Aslett. What's Really New with NewSQL? *ACM Sigmod Record*, 45(2):45–55, 2016.

[Pok13] Jaroslav Pokorny. NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

[pos18] PostgreSQL. https://www.postgresql.org/, 2018. [Online; accessed 07.03.2018].

[PVW15] Jonas Partner, Aleksa Vukotic, and Nicki Watt. *Neo4j in action*. Manning, 2015.

[raf18] The Raft Consensus Algorithm. https://raft.github.io/, 2018. [Online; accessed 19.04.2018].

[red18a] Redis. https://www.redis.io/, 2018. [Online; accessed 23.02.2018].

[red18b] Redis Replication. https://redis.io/topics/replication, 2018. [Online; accessed 06.04.2018].

[red18c] Secondary indexing with Redis. https://redis.io/topics/indexes, 2018. [Online; accessed 18.04.2018].

[RF18] Syed Zain R. Rizvi and Philip W. L. Fong. Efficient authorization of graph database queries in an attribute-supporting rebac model. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 204–211, New York, NY, USA, 2018. ACM.

[RFG+18] Romulo Alceu Rodrigues, Lineu Alves Lima Filho, Gildarcio Sousa Gonçalves, Lineu F. S. Mialaret, Adilson Marques da Cunha, and Luiz Alberto Vieira Dias. Integrating NoSQL, Relational Database, and the Hadoop Ecosystem in an Interdisciplinary Project involving Big Data and Credit Card Transactions. In Shahram Latifi, editor, *Information Technology - New Generations*, pages 443–451, Cham, 2018. Springer International Publishing.

[ria18] Riak KV. http://basho.com/products/riak-kv/, 2018. [Online; accessed 12.07.2018].

[roc18] RocksDB. https://rocksdb.org/, 2018. [Online; accessed 09.05.2018].

[sca18] ScaleArc. http://www.scalearc.com/, 2018. [Online; accessed 14.07.2018].

[SF12] Pramod J Sadalage and Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2012.

[SH18] Krishnarajanagar G. Srinivasa and Srinidhi Hiriyannaiah. Chapter five - comparative study of different in-memory (no/new) sql databases. In Pethuru Raj and Ganesh Chandra Deka, editors, *A Deep Dive into NoSQL Databases: The Use Cases and Applications*, volume 109 of *Advances in Computers*, pages 133 – 156. Elsevier, 2018.

[SMA+07] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 1150–1160. VLDB Endowment, 2007.

[spa18] SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/, 2018. [Online; accessed 15.07.2018].

[thr18] Thrift. https://thrift.apache.org/, 2018. [Online; accessed 19.04.2018].

[UO18] Yelda Unal and Halit Oguztuzun. Migration of data from relational database to graph database. In *Proceedings of the 8th International Conference on Information Systems and Technologies*, ICIST '18, pages 6:1–6:5, New York, NY, USA, 2018. ACM.

[vol18a] Project Voldemort. https://www.project-voldemort.com/voldemort/, 2018. [Online; accessed 12.07.2018].

[vol18b] VoltDB. https://www.voltdb.com/, 2018. [Online; accessed 07.04.2018].

[vol18c] VoltDB Documentation. https://docs.voltdb.com/, 2018. [Online; accessed 22.03.2018].

[vol18d] VoltDB Replication. https://docs.voltdb.com/UsingVoltDB/IntroHowVoltDBWorks.php#IntroReplicate, 2018. [Online; accessed 07.04.2018].

[WB17] Todd Warszawski and Peter Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 5–20. ACM, 2017.