ARIMA

EPIsciences

NoSQL databases: A survey

Digonaou KPEKPASSI[1] **and David FAYE**[1]

[1]Gaston Berger University, Senegal

*E-mail : kpekpassi.digonaou@ugb.edu.sn

DOI : 10.46298/arima.13970
Submitted on 23 July 2024 - Published on 2 December 2025
Volume : **43** - Year : **2025**

Editors : *Mathieu Roche, Clémentin Tayou Djamegni, Nabil Gmati*

Abstract

NoSQL data stores have introduced a new way of designing database systems to meet the recent needs of applications and services operating in areas such as the World Wide Web, Big Data, and Data Analytics. They offer a means to store and access high volumes of partially structured data by enhancing the flexibility of the data model and integrating distributed architecture at their core, thus providing better properties of high data availability and low data latency.
This paper reviews the various design approaches of NoSQL data stores, providing up-to-date information on their data models, request processing, scalability, storage management, data distribution modes, and use cases. It also addresses multi-model and cloud-oriented NoSQL stores and offer a comprehensive description of a wide range of NoSQL stores with the use of a rich taxonomy.

Keywords

Database; NoSQL; key-value; document; wide-column; graph

I INTRODUCTION

The advent of web and Big Data-oriented services has drastically increased the amount of data and, consequently, the processing loads on database systems. Due to this situation, relational database systems, initially designed for business-oriented applications but later popularized for other use cases, have progressively shown some limitations. These limitations stem from the inherent architecture and data model of relational databases. In response, NoSQL data stores have emerged, offering alternative data models and architectures to address these issues.
This work aims to provide a comprehensive overview of NoSQL databases, highlighting key properties that characterize them and demonstrating how these characteristics enhance their efficiency. To achieve this, our study is divided into several parts: First, we present a brief history of database systems up to the present day. Second, we discuss the characteristics of relational database systems as predecessors of NoSQL databases and outline the limitations they face. Third, we introduce the key characteristics and properties of NoSQL stores, mapping them to corresponding use cases, and providing an explicit description of a large number of NoSQL

databases. Fourth, we analyze the disadvantages and challenges encountered by NoSQL stores.

II RELATED SURVEYS ON NOSQL STORES AND THE PARTICULARITY OF OUR WORK

Among the numerous surveys encountered, a significant emphasis is placed on the importance and relevance of NoSQL databases as robust alternatives to traditional relational databases. These surveys highlight NoSQL stores capabilities in handling large-scale data, offering unparalleled scalability and flexibility. Some studies specifically provide comparative analyses between NoSQL and relational databases, focusing on their respective properties and advantages [67].
Conversely, several surveys delve into comparative analyses among various NoSQL stores themselves. These studies consider crucial factors such as scalability [77], performance, data modeling [87], consistency models [74, 88], and practical use cases [52, 81]. Additionally, some offer practical decision-making guidance [62, 82]. Some surveys place a strong emphasis on theoretical foundations and general overviews [72], while others dive deep into the intricacies of data modeling and querying [66, 91, 105].
Moreover, certain reviews focus on the application of NoSQL stores in specific domains such as Big Data [80], Cloud services [54, 68], and Web applications [68]. There are also notable works centered on performance evaluation benchmarks [69, 75].
Studies on specific data types, such as graph stores [43, 58, 107] and multi-model graph stores [94], have been reviewed. Additionally, some research has thoroughly investigated the challenges faced by NoSQL stores and the potential future directions they could take [63, 71, 77].
Taking into account the valuable contributions of these surveys, our work proposes several enhancements:

- Presenting a larger number of NoSQL stores compared to previous works (48 of them), including the most recent developments. These stores are well described and linked to practical use cases. This makes our work a comprehensive reference for understanding the current landscape of NoSQL databases. The rich description on each NoSQL store permit to have a good overview of their specificities and use cases.
- Utilizing subclassifications within the NoSQL data model classification, based on major observed characteristics. This approach provides a more nuanced understanding of the diverse range of NoSQL databases, allowing for better categorization and comparison based on their unique features and functionalities.
- Providing in addition dedicated sections on multi-model and cloud-based NoSQL data stores. These sections offer in-depth insights into these specific areas, addressing their unique challenges, advantages, and applications in the broader context of NoSQL databases.
- Given queries needs and data structure evolution, we have dedicated a section on the correspondance between NoSQL stores and the requirements on query patterns, use cases, properties requirements.

III THE DIFFERENT GENERATIONS OF DATABASE SYSTEMS

Multiple generations of database systems (table 1) have succeeded one another since the advent of data management systems. Here, we provide a brief overview of these generations to offer a general view of the history of database systems.

The file systems. Initially, in the mid-1960s, data was stored in raw format using file system-based data stores. These systems used a file-based structure to organize and manage data. Examples of these include ISAM [1] and VSAM [7].

The Hierarchical and Network databases systems. The hierarchical [12] and network [11] databases are the first generation of database systems. In these systems, data are stored as records linked together in hierarchical or network structures. To access data, applications must execute low-level procedural operations, navigating through the records to reach the desired data. This low-level access provides high performance and throughput, but it makes applications difficult to write due to the lack of data independence and the tedious nature of navigational access. Different usage patterns necessitate different optimal storage structures, requiring anticipation of future usage patterns before modeling the data. Examples of network databases include IDS [48], TOTAL [6], ADABAS [5], and IDMS [18], based on the CODASYL database model, while IMS and System 2000 are examples of hierarchical databases.

The Relational databases systems. Originating from the work of Codd [4], relational databases represent data as tuples grouped in collections called relations. In practice, these tuples are referred to as rows, and the collections are represented as tables. These tables have a fixed number of columns. Relational databases decouple the physical representation of data from their logical representation, enabling more independent logical data modeling over physical data structures, in contrast to hierarchical and network databases. This provided them an advantage over their predecessors by making them easier to use. They also reduce data redundancies through data normalization methods, thus optimizing storage usage.
Relational databases also benefit from the use of a common declarative language called SQL [8] for data queries. Relational database technologies, like each previous generation of database technology, were initially developed for conventional business data-processing applications, such as inventory control, payroll, and accounts.
Examples of relational databases include Oracle [14], SQL/DS [65], DB2 [16], and INGRES [10].

The Object-oriented databases systems. Object-oriented database systems [25, 27] emerged to address the limitations of relational databases in modeling data for complex applications such as computer-aided design and manufacturing systems (CAD, CAE, CASE, CAM), as well as to resolve the impedance mismatch with object-oriented programming languages.
Unlike relational databases, object-oriented databases offer semantic concepts such as generalization and aggregation relationships, allowing the representation of complex nested entities, such as design and engineering objects, and complex documents. They are based on the object-oriented paradigm, aiming to represent data as a collection of objects organized into classes with complex values associated with them. These classes can inherit properties from other classes.
Relational databases have evolved to incorporate concepts from object-oriented databases, leading to the development of Object-Relational Database (ORDB) systems. This evolution, along with their established market presence, has made it challenging for object-oriented databases to surpass relational databases in market share.
As example of object-oriented stores, we have Gemstone [24], ObjectStore [29], Versant [20], O2 [28], Iris [23].

The NoSQL databases systems. NoSQL database systems [51, 87] are a category of databases that are often schema-flexible, designed for scalability and handling large volumes of unstructured or semi-structured data. They appeared to tackle many issues encountered by relational stores such as scalability issues as the latter were build for vertical scaling (scaling by hardware upgrade) by design leading to limitations due to hardware limits. NoSQL were

Data store generation	Key specificity
File systems	Data stored in raw format in files
Hierarchical and Network data systems	Data accessed through path navigation over records
Relational databases	Data organized in rows and columns
Object oriented data stores	Data organized into classes and objects
NoSQL data store	Distributed data with flexible schema
NewSQL data stores	Distributed data with enforced ACID properties

Table 1: Key aspect of the different generations of database systems

designed with the aspect of horizontal scaling characterized by the distribution of data across multiple cheap server nodes. Also the massive apparition of very variable and evolving data structure for data representation, as in social networks, ecommerce data, IoT, leaded to need of less rigid schemas traditionnally encountered in relational store. The adoption of JSON based, columnar data, graph based data with NoSQL have permitted easy adaptation to those new kind of data. Also with applications requiring high velocity with high workload like , such as realtime analytics (realtime data), IoT sensors stream data there is a need to reduce bottleneck created by constraints like transaction, joins. Therefore many NoSQL store avoids the use of complex joins and multi-rows transactions. NoSQL data stores consist in various data model subcategories, including key-value, document, wide-column, and graph database systems. This work will further elaborate on the characteristics of these systems to understand their specificities.
Among them we have Amazon Dynamo [42], Google Bigtable [44], Cassandra[1].

The NewSQL database systems. [56, 98] As many services, who were facing scalability issue with high volume of data, were directing to NoSQL data stores, a part of them still require a strong consistency constraint. The compromise of NoSQL store in this context of consistency, being too big, alternatives such as NewSQL stores wich purposed to keep strong consistency aspect even though adopting the scalability mode of NoSQL stores. Some NewSQL stores(e.g., TiDB [92], MemSQL/SingleStore[2]) permit to run OLTP and OLAP workloads in the same cluster without ETL (HTAP (Hybrid Transactional/Analytical Processing)). Examples of NewSQL databases include Google's Spanner [60], [95], VoltDB[3], NuoDB[4].

IV THE RELATIONAL DATABASES

4.1 The key particularities of the relational databases

Relational databases have key characteristics that enable efficient operation. They are:

The predefined schema of the databases. The tables of relational databases have a predefined schema initialized at their creation. This helps optimize storage and request efficiency as the storage organization is predictable.

[1]https://cassandra.apache.org/doc/latest/
[2]https://www.singlestore.com/
[3]https://www.voltactivedata.com/
[4]https://doc.nuodb.com/nuodb/latest/release-notes/

The use of join operations. The rows of the tables can be linked by join operations when a query is performed. This is possible by using some columns to store keys referencing rows of other tables.

The ACID properties. The concept of transactions was introduced by Jim Gray [30] and described as a unit of work in database systems spanning a set of operations. It is based on four properties: Atomicity, Consistency, Isolation, and Durability (ACID properties), described as follows:

- **Atomicity**: A transaction is committed if all operations it contains succeed; otherwise, if one of the operations does not succeed, all the other operations are rolled back, and the transaction is aborted.
- **Consistency**: Ensures that from one transaction to another, the database system transitions from one consistent state to another. Integrity checks, such as foreign key integrity and primary key uniqueness checks, provide these guarantees.
- **Isolation**: Guarantees that two concurrent transactions will not see each other's new data updates until they commit them.
- **Durability**: Ensures that the updates of a committed transaction will be visible to subsequent transactions, independent of hardware or software errors, until they are overwritten by a new update.

A standardized and powerful query language SQL. Relational databases utilize a common standard query language called SQL, which is a declarative query language. It is powerful, allowing for the writing of complex queries, and it is a mature query language.

4.2 Limitations of the relational databases

Relational databases face several efficiency challenges in modern use cases:

General limitations. Originally optimized for business data processing, relational databases struggle with specialized workloads such as data warehousing, streaming, and text search. As shown in [41], specialized systems (e.g., Bigtable [44], Neo4J) outperform relational databases for these scenarios.

Scalability and latency. Relational databases traditionally scale vertically (hardware upgrades), which is costly and limited. Horizontal scaling (distributing data across servers) introduces shared-memory, shared-disk, and shared-nothing architectures. Shared-memory and shared-disk approaches suffer from bottlenecks and expensive coordination. Shared-nothing improves scalability but brings complex challenges:

- **Atomicity:** Distributed transactions require coordinated commit/rollback, increasing latency.
- **Consistency:** Maintaining foreign key integrity across nodes needs distributed locks, causing bottlenecks.
- **Isolation:** Synchronizing locks or versions (e.g., via Paxos [37]) adds overhead.
- **Durability:** Multi-node commits require protocols like two-phase commit [22].
- **Joins:** Cross-node joins are costly in bandwidth and latency.

Horizontal scaling in relational databases is possible but complex and often inefficient; NoSQL systems address these bottlenecks.

Schema inflexibility. Relational tables require a fixed schema, leading to:

- Inability to store records with varying structures.
- Costly and disruptive schema changes (table rebuilds, downtime).
- Storage waste from unused fields filled with nulls.

V THE NOSQL DATABASES

5.1 The precursors of the NoSQL movement

The most notable precursors of the NoSQL movement were Bigtable [44], Amazon Dynamo [42] and the CAP theorem [34, 36]. We will discuss about them in the following.

5.1.1 The Brewer's CAP theorem

The CAP theorem (figure 1) was conjectured by Fox and Brewer [34, 36] and proved by Gilbert and Lynch [39]. It states that, for any shared-data system (such as a distributed data store), it is only possible to achieve at most two of the three properties—consistency, availability, and partition tolerance—simultaneously.

These properties are defined as follows:

- **Consistency**: This property determines if a system guarantees linearizability [26]. We will describe the notion of consistency in more detail in subsection 5.10.
- **Availability**: This property denotes that every request sent by a client eventually receives a successful response within a finite time. However, this definition does not specify a limit on response time. Therefore, even if a successful response is received after many days, the system is still considered available. Consequently, latency is also an important factor.
- **Partition tolerance**: This property denotes that a system continues to respond to client requests even in the case of a network partition between nodes of a distributed system, provided it adheres to either the availability or consistency properties mentioned earlier. However, this property does not account for other failure types such as node failures or message loss.

From the CAP theorem, we can describe the three possible types of distributed storage systems with respect to the CAP properties:

- **CA systems (Consistency + Availability)**: These database systems provide availability and consistency but not partition tolerance. It is practically impossible for distributed data stores to fit this description as network partitions are unavoidable. This type is more associated with non-distributed data stores, such as traditional relational databases.
- **CP systems (Consistency + Partition tolerance)**: These systems maintain consistency in the presence of network partitions. Some nodes may not respond to avoid an inconsistent state due to a lack of synchronization with disconnected nodes. Databases with this property are typically used in networks with few network failures, such as within a single data center.

- **AP systems (Availability + Partition tolerance)**: These databases are found in distributed systems spread across geographically distributed data centers, where network failures and increased latency are more common. To reduce update failures, consistency constraints are relaxed, leading to potential conflicting data. Therefore, manual or automatic conflict resolution mechanisms are required.

As previously indicated, systems must fit into one of the three categories: CA, CP, or AP. However, Brewer suggested that systems can implement continuous CAP properties, allowing for progressive levels between consistency and availability. This enables better adaptability of distributed systems depending on the use case.

Also CAP does not account for the tradeoffs that occur during *normal operation*. In practice, most distributed databases replicate data to improve fault tolerance and scalability. Replication introduces another fundamental tradeoff:

- If updates are propagated synchronously, the system achieves strong **consistency** but suffers from higher **latency**.
- If updates are propagated asynchronously, the system achieves low **latency** but risks temporary inconsistency across replicas.

To capture both cases, Abadi proposed the **PACELC theorem** [57], which extends CAP. It states that *if there is a partition, the system must choose between availability and consistency; else, when the system is healthy, it must choose between latency and consistency.* Examples include:

- Dynamo, Cassandra, Riak: **PA/EL** (favor availability during partitions and low latency in normal operation).
- VoltDB, Megastore: **PC/EC** (preserve consistency at all times, paying in latency or availability).
- PNUTS: **PC/EL** (preserve consistency during partitions, but reduce it in normal operation to keep latency low).

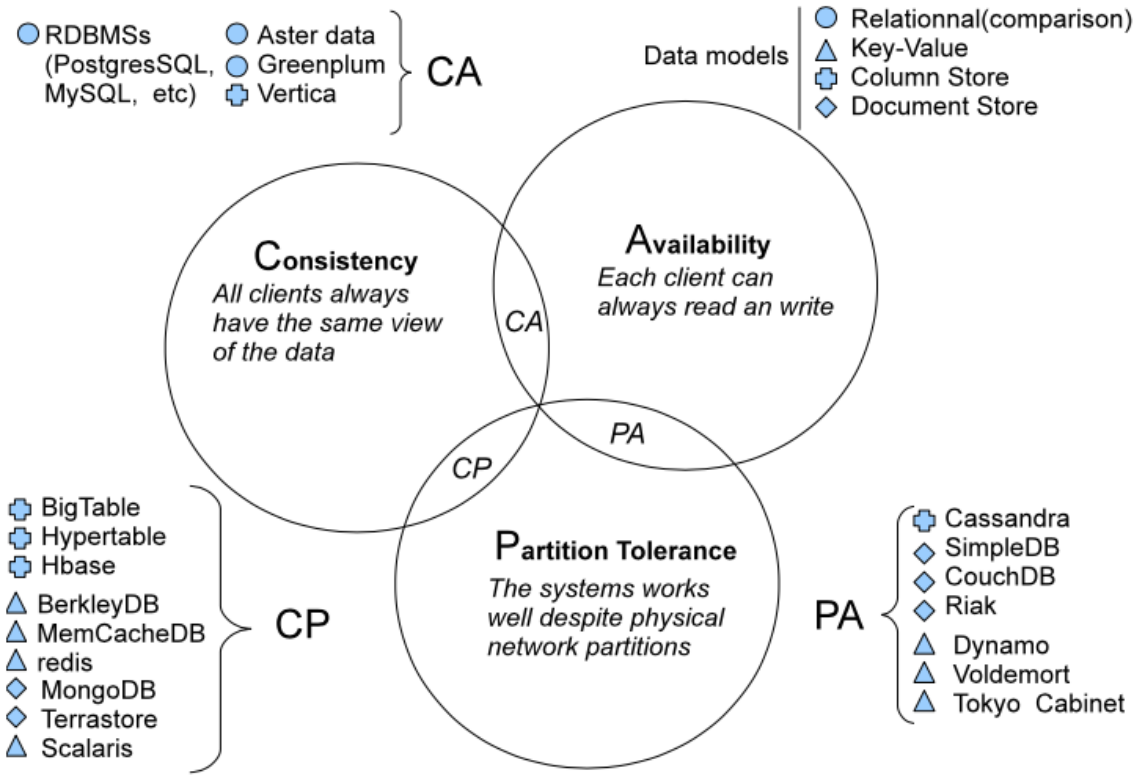

Figure 1: Some NoSQL proposals and te CAP theorem

5.1.2 Amazon Dynamo

Amazon Dynamo [42] [section 5.4.1] introduced the concept of eventual consistency and combined it with distributed hash tables [38] to build highly available and scalable peer-to-peer [38] data store architectures. The use of distributed hash tables helps to evenly distribute data among nodes in a decentralized manner, avoiding bottlenecks and enhancing scalability and fault tolerance. Eventual consistency, a type of relaxed consistency (will be studied further). The paper on Dynamo has inspired a new generation of NoSQL stores such as Riak, Voldemort, and Cassandra.

5.1.3 Google Bigtable

Google Bigtable [44][section 5.4.2] was the precursor to the wide-column stores as we know them today. It handles rows with a flexible number of columns. The use of LSM-tree [31] as the underlying data structure permits high write throughput. Additionally, the use of a distributed file system (GFS [40]), for scalability enables it to handle petabytes of data across thousands of commodity servers. These properties have encouraged the emergence of similar NoSQL stores such as Cassandra, HBase[5], and Accumulo[6].

[5]https://hbase.apache.org/
[6]https://accumulo.apache.org/

	Bigtable	**Amazon**
Distributed architecture	Centralized	Decentralized
Consistency	Single region strong consistency	Flexible consistency
Keys aspects	High throughput Low latency	High availability High scalability
CAP property	CP	CA

Table 2: Bigtable vs Amazon Dynamo

5.2 What distinguish the NoSQL databases?

The schemaless data model. Unlike relational database systems that require a predefined schema for data storage, NoSQL data stores are mostly schemaless, meaning it is not necessary to set a predefined structure for each record type. This results in the following advantages:

- Records with different schemas can be stored together.
- It is possible to easily alter the schema of records without constraints, which is advantageous for rapidly evolving projects.
- Applications can define more customized schemas on their side, leading to more flexible possibilities. This is typical for key-value databases that primarily store raw data without managing a schema.
- Records occupy only the space required by the fields that have defined values, unlike relational databases that may have empty fields with null values, leading to wasted storage space.

One disadvantage of schemaless collections is that, since the document has no fixed schema, the database cannot optimize the storage and retrieval of parts of the aggregate effectively.

The relaxed consistency (BASE). Many NoSQL databases relax distributed consistency constraints by adopting the BASE paradigm. BASE is an acronym for Basically Available, Soft State, Eventually Consistent, and contrasts with the ACID paradigm. BASE systems prioritize availability over consistency, offering replicated soft states. Replica synchronization is typically achieved through eventual consistency (an asynchronous replication mode), which we will explore further in subsection 5.10.

A native horizontal scalability mechanism with shared-nothing architecture. One reason for the success of NoSQL stores is their inherent ability to distribute data storage and processing across multiple servers, usually commodity servers. These servers do not share common hardware, thereby eliminating bottlenecks—a concept known as the shared-nothing architecture. The distribution process involves sharding and replication. Sharding dispatches records among multiple nodes, while replication mirrors data across multiple servers. These properties enhance data throughput and high availability. Further details about their implementation will be discussed in subsection 5.8.

Simple query languages without join operations. NoSQL data stores typically do not support join operations in their core functionality. They are oriented toward storing denormalized data, and if a client application requires a join operation, it must perform the join after retrieving the necessary data. Data stored in a denormalized format is designed to resemble the result of

a join operation, anticipating the types of joins applications are likely to require, thus avoiding the need for join operations by the client application.

NoSQL uses a denormalized format, a form of data aggregates where data typically distributed among different records of the same or different tables in the relational model are merged to form an aggregated record. These aggregated records gather data usually requested together in the same storage area. Document-oriented, key-value, and wide-column data stores are aggregate-oriented data systems, where this is a key concept. Aggregates can be computed on servers using technologies such as MapReduce, which collect data from different sources to perform operations. The aggregate format has the following advantages:

- Reduces cross-server data transfer as related data is less dispersed.
- Eliminates the need for foreign key integrity checks across servers.
- Avoids the use of distributed lock mechanisms or distributed version control systems required by ACID transactions, as related data is typically stored together. Therefore, NoSQL databases usually support only ACID transactions applied record-by-record rather than spanning multiple records, as in relational databases.
- Allows client applications to access the aggregated mode of data, avoiding the need for join operations to obtain the desired data format, thus reducing additional processing time and costs and enhancing latency properties.

NoSQL databases provide predictable latency because they do not require complex join operations or locks that could slow down request responses.

Easy node failure management. Designed to handle a high number of server nodes, NoSQL data stores consider hardware node failures as a natural occurrence. They maintain high data availability and durability even in the event of major server failures through inherent failover and backup capabilities.
Table 3 summarizes the key differences between SQL and NoSQL databases.

Table 3: Key differences between SQL and NoSQL databases

Criterion	**SQL (Relational)**	**NoSQL**
Data Model	Tables with rows & fixed columns	Key–Value, Document (JSON/BSON), Wide-Column, Graph
Schema	Fixed schema, must `ALTER TABLE` to change	Schema-less / schema-on-read, flexible fields per record
Query Language	ANSI SQL (`SELECT`, `JOIN`, `GROUP BY`)	Model-specific (JSON queries, graph traversals, key lookups) — may have SQL-like dialect but not fully relational
Joins	Built-in, core feature	Usually absent or limited — joins emulated at app level or via aggregation framework

Transactions	Strong ACID by default	Often relaxed (BASE: Basically Available, Soft state, Eventually consistent) — ACID may be optional or limited
Scalability	Scale-up (bigger servers)	Scale-out (horizontal sharding/-partitioning) by design
Consistency	Strong consistency by default	Tunable or eventual consistency (CAP theorem trade-offs)
Distribution	Single-node primary, clustering is add-on	Distributed-first architecture, replication/sharding native
Use Case Fit	General-purpose, structured data	Specialized: real-time analytics, large-scale writes, flexible data structures, high-relationship queries

5.3 Taxonomy of NoSQL stores

It is not easy to categorize NoSQL stores as they have a variety of design modes. However, based on frequently encountered characteristics, we can categorize them according to the following criteria:

- **Data Model** (section 5.4) defines how data is structured, stored, and accessed in the database. The common models include key-value, document, wide-column, and graph data models.
- **Consistency Model** (section 5.10) defines the guarantees provided regarding the visibility and ordering of updates. This includes models such as eventual, strong, causal, and tunable consistencies.
- **Scalability Type** describes the ability to handle growth in data load and user access. NoSQL stores typically adopt horizontal scaling, though vertical scaling is also a consideration.
- **Data Partitioning** (section 5.8.1) involves dividing data into segments distributed across multiple nodes. This can include sharding (partitioning by rows), vertical partitioning (partitioning by columns), and traversal-oriented partitioning (used in graph stores).
- **Data Replication Strategy** (section 5.8.2) manages how data copies are handled across nodes. Methods include Primary-secondary and multi-master (or peer-to-peer) replication, with replication being either synchronous or asynchronous.
- **Transaction Support** defines how transactions are managed. Databases can be ACID compliant (section 4.1) or BASE compliant (section 5.2).
- **Storage Mode** (section 5.9) describes how data is physically stored. Data can be stored in memory, on persistent storage (disk, SSD), or in a hybrid manner.
- **Query Model and Capability** defines how data can be queried. This can be done through a declarative or procedural query language, and access can be provided through APIs.
- **Use Cases** defines the specific scenarios or applications for which a given data store is well suited. Use cases include real-time applications, content management, IoT, and big data applications.
- **Geographic Distribution** describes the deployment scope and how data is managed across different regions, including single-region and multi-region deployments.

There are other criteria that have not been detailed here. We will further describe most of the cited criteria in the following sections of this work.

5.4 The NoSQL stores data models

NoSQL stores fall into various data models types, each adapted to specific use cases. We can identify four principal ones such as the key-value, document, wide-column and graph data models

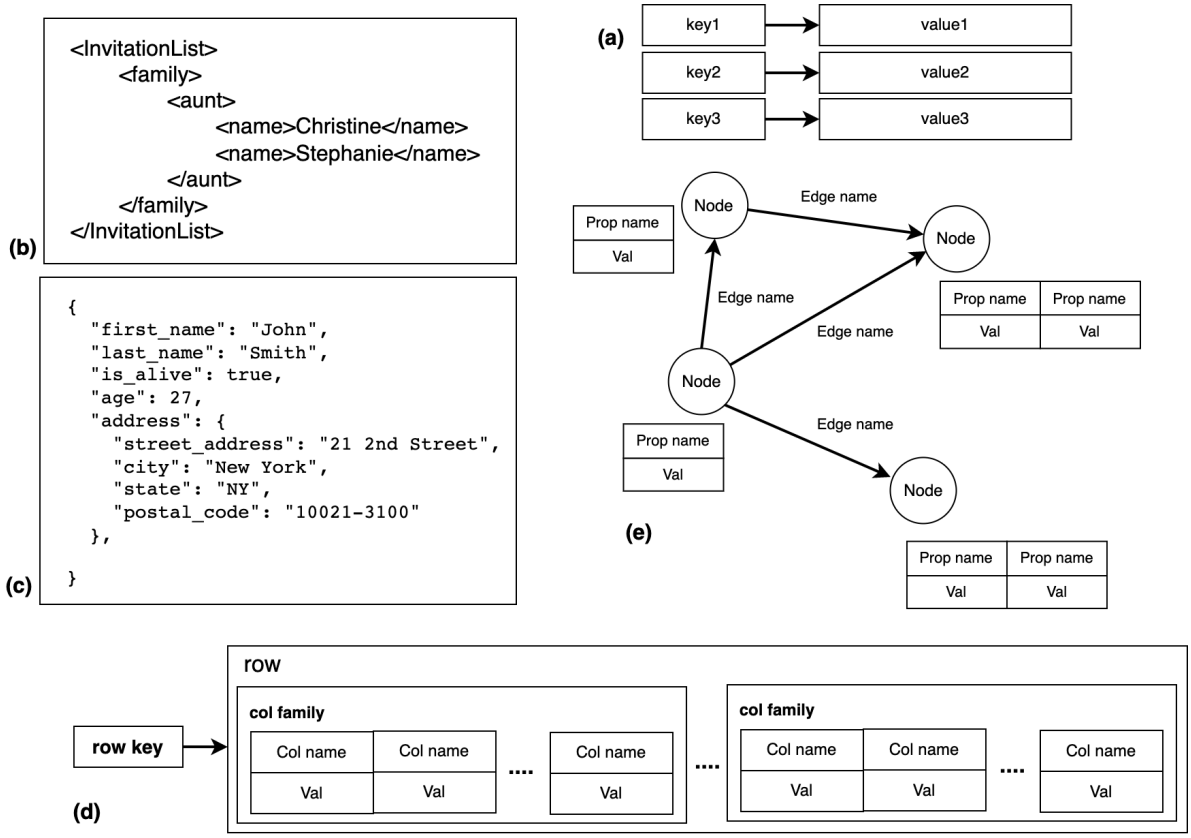

Figure 2: The NoSQL data models

We will further study their specificities in the following sub-sections.

5.4.1 Key-value stores

Key-value stores have been around for a while, usually as single-server oriented stores (table 4). However, the paper of Dynamo [42] popularized the concept of distributed key-value stores based on distributed hash tables, offering high scalability and eventual consistency.

The data model. The key-value data model (figure 2a) is the simplest data model encountered in NoSQL databases [94]. It can be compared to associative arrays, dictionaries, or hashes. Each record in the key-value data model stores an arbitrary value identified by its unique key. The values are usually opaque to the system and are stored as blob data. This blob data is deserialized/serialized on the client application side.
Since the data store generally does not analyze the value content, the advantage is the very fast access response and the flexibility for record values to have diverse internal structures. However, even if the client needs part of a value, it must fetch the entire value. There are cases where key-value stores can accept basic data structures to compose values. Examples include

lists and sets used in Redis[7], Aerospike[8], tabular structures used in IBM Spinnaker [53], HyperDex [61], Yahoo Pnuts [46], and document structures used in Riak KV[9], but with only basic operations on them. the use of additional data structures have blurred the boundaries between key-value data models and other data models.

The query mode. Key-value stores essentially offer simple operations such as get, put, and delete. For key-value stores with values having low-level data structures like Redis, operations on basic data structures include value increments and usual operations on lists, strings, and maps. Riak also offers the possibility to perform link traversals between records using metadata attached to values and to use regular expressions and text search engines. Many key-value stores provide REST API access.

The storage layout and indexing. Many key-value stores use classical indexing modes such as B-trees for good local range queries or hash tables for good local random access to data. Also on distributed aspect we have data structures such as distributed hash tables. We can see more details concerningn those aspects in section 5.5.
Some store data primarily in memory, such as key-value stores used for caching purposes like Memcached[10]. Others offer the possibility to persist data to disk, like BerkeleyDB[11] and Amazon Dynamo [42].

The use cases. Key-value databases can be used for cache storage, message queuing, and brokering. Some can be used as embedded storage in applications or as storage engines in other databases.

Key-value stores categorized according to their main features. Depending on their main features, key-value stores are used as cache stores, memory-based stores, high-availability oriented stores, data-grid oriented stores, and embedded data stores. We will describe notable key-value stores for each feature (see further categorizations in Table 4).

- **Memory cache-oriented stores:** Memory cache-oriented stores are used to improve performance by reducing the need for repetitive and resource-intensive data retrievals from more complex data store systems. Distributed cache stores distribute cached data across multiple nodes, pooling the dedicated memory from different nodes. Notable data stores dedicated to caching include Memcached, Ehcache[12], and Ncache[13].

  Memcached[14] provides distributed caching using a consistent hashing mechanism to distribute keys across nodes, without persisting data to disk. Applications interact with Memcached servers through client libraries, following a client-server architecture. Communication between clients and servers occurs via a simple text-based protocol. Memcached supports cache eviction policies such as LRU and time-to-live (TTL). Widely used in web environments, particularly CMS (Content Management Systems), it is employed on platforms such as YouTube, Reddit, and Facebook.

  Ehcache[15], unlike Memcached, functions as a cache storage embedded in applications

[7] https://redis.io/docs/latest/
[8] https://aerospike.com/docs/
[9] https://docs.riak.com/riak/kv/latest/index.html
[10] https://github.com/memcached/memcached/wiki/Home
[11] https://www.oracle.com/database/technologies/related/berkeleydb.html
[12] https://www.ehcache.org/documentation/
[13] https://www.alachisoft.com/resources/docs/ncache/
[14] https://github.com/memcached/memcached/wiki/Home
[15] https://www.ehcache.org/documentation/

as a library. Supporting distributed caching via Java RMI (Remote Method Invocation), Ehcache distributes cache data across multiple nodes, with the application implementing the key distribution strategy. It allows a client-server architecture with client and server libraries and integrates seamlessly with Java EE technologies, making it suitable for Java Enterprise applications, Spring, and Hibernate. In contrast to Memcached, Ehcache can persist data to disk and supports replication. It offers cache change notifications, enabling applications to react to cache changes. Developers can easily customize and configure Ehcache's components through XML or Java annotations. Ehcache supports diversified eviction policies such as Least Recently Used (LRU), Least Frequently Used (LFU), Time To Live (TTL), FIFO, or custom policies.

Ncache[16], like Ehcache, is an embedded library supporting cache distribution. Similar to Memcached, consistent hashing is used for key distribution. NCache enables a client-server architecture with specific client or server libraries and employs a binary protocol for node communication. Cache dependency is supported for invalidating or updating cache items based on changes in the underlying data source, and cache notifications can send alerts to applications when specific cache events occur. Specifically designed for .NET applications, NCache integrates easily with ASP.NET, WCF (Windows Communication Foundation), and others. It includes a SQL-like query language for efficient data retrieval based on specific criteria and supports features such as transactions, item versioning, and caching of various data types like objects and files. Additionally, NCache supports cache eviction policies such as LRU, LFU, priority eviction, TTL, or custom policies.

Synthesis: We can say that the n-memory data stores are designed for speed and transient storage. Memcached is minimalist, in-memory only, and ideal for high-traffic web caching. Ehcache and NCache add persistence, replication, and notifications, integrating tightly with Java and .NET ecosystems respectively. They evolve simple caching into programmable, enterprise-ready cache layers, whereas Memcached remains a lightweight distributed cache.

- **In-memory data stores:** These are used to store and retrieve data primarily in main memory (RAM) rather than on disk. This allows for extremely fast read and write operations, enabling low-latency access to data. Additionally, they generally allow the value part of the data to be represented in basic forms such as integers, strings, and various data structures like lists, sets, maps, and large binary objects (blobs). Some even offer the capability to store JSON-based data, which can be queried with simplified query languages.
  Among them, we have Redis[17], Aerospike[18], KeyDB[19], Tarantool[20], and Kyoto Tycoon[21].

  Redis features a simple API with a clear set of commands, easily integrated into applications, which contributes to its popularity among developers. This versatile data store supports various data structures, including lists, sets, hashes, bitmaps, and HyperLogLogs, allowing developers to choose the most suitable one for their specific use cases. JSON data storage is enabled through the RedisJSON extension, which supports querying and extracting data using JSONPath. Although primarily an in-memory store, Redis offers persistence options to ensure data durability and recovery from server restarts or failures. JSON data

[16] https://www.alachisoft.com/resources/docs/ncache/
[17] https://redis.io/docs/latest/
[18] https://aerospike.com/docs/
[19] https://docs.keydb.dev/docs/
[20] https://www.tarantool.io/en/doc/latest/
[21] https://ktlib.readthedocs.io/en/latest/

can also be indexed. While Redis is a single-threaded application, multithreaded alternatives like KeyDB provide nearly identical features. Consistent hashing is used for data distribution, with replication following an eventual consistency model within a Primary-secondary architecture. Alternatives like Tarantool provide similar data structures but include transactions and ACID compliance. Redis includes a publish/subscribe (pub/sub) mechanism suitable for real-time messaging systems, allowing clients to subscribe to channels and receive messages. Lua scripting is supported for performing complex operations on the server side. Redis is also utilized for caching, offering eviction policies based on LRU, LFU, and TTL, and finds applications in real-time analytics, session storage, messaging systems, leaderboards, task queues, and geospatial indexing.

Aerospike utilizes a hybrid approach of combining memory and disk storage. Frequently accessed data is kept in memory for quick access, while less frequently accessed data is stored on disk for persistence. Optimized for flash storage technologies (SSD), Aerospike supports enhanced data types including lists, maps, geospatial data, large lists of elements, and large volume data types such as CLOB (Character Large Object). It also supports secondary indexes. Aerospike offers strong consistency and ACID transactions, differing from Redis by providing automatic data distribution and rebalancing. It is well-suited for real-time applications, including real-time bidding platforms, product recommendations, analytics, gaming, time-sensitive financial transactions, fraud detection, risk analysis, portfolio management, telecommunication subscriber data management, activity record-keeping, and IoT data collection and analysis across a vast number of devices.

Oracle NoSQL Database[22] integrates seamlessly with the Oracle ecosystem. It uses a SQL-like query language and supports complex data types including maps, arrays, and records. JSON document types can also be used as values, with fields that can be indexed and queried. Oracle NoSQL Database supports ACID transactions and provides tunable consistency levels, from eventual consistency to causal consistency. It features automatic sharding and employs indexing mechanisms like B-trees and LSM-trees for efficient data retrieval.

Synthesis: We can say that in-memory data stores go beyond caching by supporting persistence, clustering, and rich data models. Redis dominates with versatile structures and ecosystem support; KeyDB extends it with multithreading. Aerospike and Oracle NoSQL target enterprise-scale analytics with stronger consistency and secondary indexes. Tarantool adds ACID + SQL, making it suitable for transactional workloads. In essence, Redis is the general-purpose leader, while Aerospike/Oracle serve enterprise-grade real-time systems.

- **Highly-available key-value stores:** These are designed to provide a high level of data accessibility even in the face of failures or disruptions. Many NoSQL stores offer these characteristics, including key-value stores such as Amazon Dynamo [42], Riak KV, and Voldemort[23].
  Amazon Dynamo [42] is a key-value store described by Amazon, designed for scenarios where high availability, fault tolerance, and scalability are critical, such as the Amazon e-commerce platform. It employs automatic data distribution among nodes using consistent hashing and a quorum-based replication mechanism, where reads and writes require the participation of a specified number of nodes. The consistency can be tuned from eventual

[22] https://docs.oracle.com/en/database/otherdatabases/nosqldatabase/
[23] https://www.project-voldemort.com/voldemort/

consistency to strong consistency. Dynamo propagates node membership information and data updates through a gossip-based protocol and uses active anti-entropy to maintain data consistency and repair inconsistencies.

Riak KV is an open-source implementation of the Dynamo paper, offering consistent hashing and tunable consistency. Additionally, it supports MapReduce [47] for data processing and analysis across Riak nodes. It supports various data types such as sets, maps, and binary data. Riak can store JSON documents and uses Apache Solr for indexing and searching within those documents. Riak can be used for session storage, user profiles and preference data, content delivery and caching, IoT data storage, logging and event data, and collaborative applications.

Voldemortis another open-source implementation of the Dynamo paper and supports custom storage engines such as Berkeley DB and RocksDB[24]. It stores data in binary form.

Etcd[25] is a high-availability key-value store designed to provide strong consistency guarantees, primarily for managing and coordinating distributed systems. It stores distributed system configuration data, which necessitates strong consistency, high availability, and resilience to network failures and node partitions. Etcd employs the Raft Consensus Algorithm [93] to achieve consensus among nodes in the cluster, ensuring that all nodes agree on the order of operations. It is used for service discovery in microservices, storing information about available services and their locations. Etcd supports leader election and a watch mechanism, enabling clients to monitor data changes and respond accordingly. It is actively used with the Kubernetes[26] project for storing and managing cluster state but can also be used for traditional data storage.

Synthesis: We can say that highly-available data stores are built for resilience and distributed reliability. Dynamo, Riak, and Voldemort prioritize availability with tunable consistency, making them well-suited for large-scale, fault-tolerant applications. Etcd takes the opposite path, ensuring strong consistency via Raft, which makes it essential for cluster management (e.g., Kubernetes). The former emphasize scalability and uptime, while Etcd ensures coordination and correctness.

- **Embedded key-value stores:** Embedded data stores are designed to be directly embedded within applications and even used as storage engines for other database systems. They eliminate the need for a separate database server for applications. These stores are suitable for scenarios where applications need a simple database system with minimal storage configuration requirements, and where the dataset is not large enough to exceed the available system resources. They have the advantage of being lightweight with minimal dependencies, offering a simple API with basic operations and high performance by avoiding the overhead associated with network communication. They also offer flexibility for some database systems that can propose various database engines characterized by these embedded data stores, allowing selection of the appropriate storage engine for the corresponding data workload type. They are also suitable for applications that need to manage local caches. Mobile and edge devices can easily integrate them. Examples

[24]https://rocksdb.org/docs/getting-started.html
[25]https://etcd.io/docs/
[26]https://kubernetes.io/docs/home/

of embedded-oriented data stores include BerkeleyDB, LevelDB[27], RocksDB[28], LMDB[29], Badger[30], BoltDB[31], and Kyoto Cabinet[32].

BerkeleyDB originated as part of the UNIX BSD operating system from the University of California, Berkeley, before being acquired by Sleepycat and eventually by Oracle. The underlying data structure is a B-tree. Known for performance, reliability, and simplicity, BerkeleyDB is written in C, with a Java version for seamless integration into Java applications. Initially released in 1991, BerkeleyDB preceded the NoSQL movement but supports the storage of flexible schemas, making it suitable for NoSQL applications. Memory-mapped files are used for efficient data storage and access, leveraging the operating system's virtual memory feature. Support for transactions is provided, offering ACID properties. Concurrency is managed using Multi-Version Concurrency Control (MVCC) [22] for concurrent multi-process and multi-thread access. Employing a Primary-secondary replication mode, where the master is chosen through an election process, replication is typically asynchronous but can be configured to be synchronous. BerkeleyDB does not implement a key distribution mechanism for managing data distribution across cluster nodes, requiring applications to handle the distribution mechanism themselves.

LevelDB uses the LSM Tree data structure (Log-Structured Merge-Tree), offering efficient write performance by combining memory and disk-based structures. With the LSM Tree, data is stored in order by keys, enabling efficient range queries. Badger, an implementation of the LSM Tree written in Go, offers similar capabilities and integrates well with Go applications.

RocksDB, built on LevelDB's foundation, is customized for distributed systems. Supporting tiered storage configuration for LSM Trees, RocksDB allows customization of storage levels, types of storage devices for different levels, and the option to activate compression for selected levels. The tiered storage configuration can be dynamically adjusted, adapting to changing workloads and storage requirements. It enables the separation of frequently and less frequently accessed data across different storage levels. While LevelDB uses a single-threaded compaction mechanism, RocksDB improves write throughput with a multi-threaded compaction mechanism, leveraging parallel processing.

LMDB is notable for memory efficiency, simple design, and reliability. Utilizing a B+Tree data structure and a zero-copy architecture, LMDB achieves low overhead, making it optimized for read-intensive workloads. Durability is ensured through the use of write-ahead logging. Supporting transactions and offering ACID compliance, LMDB is intended for single-process use, without support for multi-process or multi-threaded modes. This memory-efficient design has led to its popularity in embedded systems, IoT, and applications that require fast and reliable data storage.

BoltDB is akin to BerkeleyDB but employs a B+Tree and is written in Go, enhancing compatibility with Go applications. Memory-mapped files are used for storage, facilitating efficient read and write operations for multi-goroutine access. Data storage follows a page-based architecture (with data divided into fixed-sized pages), optimizing read and write efficiency. BoltDB serves as the storage engine for the etcd key-value store. One

[27]https://github.com/google/leveldb
[28]https://rocksdb.org/docs/gettingstarted.html
[29]http://www.lmdb.tech/doc/
[30]https://dgraph.io/docs/badger/
[31]https://github.com/boltdb/bolt
[32]https://dbmx.net/kyotocabinet/

notable fork of BoltDB is Bbolt.

Kyoto Cabinet is an efficient key-value storage system written in C. Offering multiple storage backends, Kyoto Cabinet allows the use of hash tables, B+Trees, and fixed-length record-based tables. It supports memory-based and cache-based storage, using a B-tree structure for persistent storage and a cache mechanism to store frequently accessed data in memory. The append-only storage option ensures new data is added only to the end of the file. Users can fine-tune various parameters for optimal performance based on specific requirements. Kyoto Cabinet supports ACID transactions, memory-mapped files, and compression, enabling concurrent read and write operations. In addition to being used as an embedded store, it is suitable for caching, custom indexing (e.g., combining B-trees and hash tables), fast access through hash tables, and handling write-intensive workloads with append-only storage.

Synthesis: We can say that Embedded key-value stores are lightweight, process-embedded stores with trade-offs in storage engines. BerkeleyDB is the mature, general-purpose option with ACID and replication. LevelDB and RocksDB (LSM-based) excel at write-heavy workloads, RocksDB adding compression and multi-threading. LMDB and BoltDB are simple, memory-mapped, and ACID, optimized for read-heavy or Go-native use cases. Kyoto Cabinet stands out for backend flexibility (hash, B+Tree, append-only). Overall, embedded stores trade clustering for local speed and efficiency.

Table 4: Comparative Overview of Key-Value Stores

Category	Store	Architecture	Data Persistence	Distribution / Replication	Data Structures / Querying	Special Features	Typical Use Cases
Memory Cache-oriented	Memcached	Client-server, distributed cache	No persistence (in-memory only)	Consistent hashing, replication by app logic	Key-value only	LRU, TTL eviction; simple text protocol	Web caching (e.g., YouTube, Reddit, Facebook)
	Ehcache	Embedded (Java), can be distributed	Optional disk persistence	Distribution via Java RMI, replication supported	Custom eviction, cache notifications	XML/Java config; Spring/Hibernate integration	Java EE, Hibernate/Spring caching
	NCache	Embedded + client-server for .NET	Yes (file/object caching)	Consistent hashing, replication	SQL-like query, objects/files caching	Cache dependencies, notifications, transactions	ASP.NET apps, enterprise .NET caching
In-memory Data Stores	Redis	Client-server	Optional persistence (RDB/AOF)	Primary-secondary, clustering, eventual consistency	Strings, lists, sets, hashes, streams, JSON (via RedisJSON)	Pub/sub, Lua scripting, eviction policies	Sessions, analytics, queues, leaderboards
	Aerospike	Hybrid: RAM + SSD	Yes, optimized for flash	Automatic sharding, strong consistency, ACID	Maps, lists, geospatial, CLOB	Secondary indexes, auto rebalancing	Real-time bidding, fraud detection, IoT
	KeyDB	Redis-compatible (multi-threaded)	Optional persistence	Primary-secondary replication	Same as Redis	Multi-threaded performance	Redis alternative with higher throughput
	Tarantool	In-memory + optional persistence	Yes	Sharding supported	Data structures + SQL + transactions	ACID compliance	High-performance OLTP, caching
	Oracle NoSQL DB	Client-server (Oracle ecosystem)	Yes (B-trees + LSM trees)	Automatic sharding, tunable consistency	JSON, maps, arrays, SQL-like querying	ACID transactions, Oracle integration	Enterprise apps, Oracle-based analytics
Highly-available KV Stores	Amazon Dynamo	Decentralized, peer-to-peer	Yes	Consistent hashing, quorum-based replication	Key-value only	Gossip protocol, anti-entropy	Amazon-scale e-commerce

Continued on next page

Category	Store	Architecture	Data Persistence	Distribution / Replication	Data Structures / Querying	Special Features	Typical Use Cases
	Riak KV	Dynamo-inspired	Yes	Consistent hashing, tunable consistency	Sets, maps, JSON docs + Solr indexing	MapReduce support	Session storage, IoT, logging
	Voldemort	Dynamo-inspired	Yes (BerkeleyDB, RocksDB)	Consistent hashing	Binary data only	Pluggable storage engines	Large-scale web apps
	Etcd	Strongly-consistent cluster (Raft)	Yes	Raft consensus replication	Simple key-value, watch API	Leader election, service discovery	Kubernetes cluster state, config management
Embedded KV Stores	BerkeleyDB	Embedded, library	Yes (B-tree, mmap)	Primary-secondary replication	Flexible schemas	ACID, MVCC	Embedded DB engine, local caches
	LevelDB	Embedded, library	Yes (LSM Tree)	No built-in replication	Sorted key-value (range queries)	Efficient writes	Local storage, simple DB engine
	RocksDB	Embedded (LevelDB-based)	Yes (tiered LSM)	No built-in replication	Range queries	Multi-threaded compaction, tiered storage	Distributed systems needing fast writes
	LMDB	Embedded	Yes (B+Tree, mmap)	No replication	Key-value only	Zero-copy, ACID, read-optimized	Read-heavy workloads, embedded apps, IoT
	BoltDB	Embedded (Go)	Yes (B+Tree, mmap)	No replication	Page-based storage	ACID, Go ecosystem	Go apps, used by etcd
	Kyoto Cabinet	Embedded	Yes (B+Tree, hash tables)	No replication	Flexible storage backends	Append-only option, ACID, compression	Caching, indexing, write-intensive workloads

Table 4: Comparative Overview of Key-Value Stores

5.4.2 *The wide-column stores*

Wide-column databases (table 5) are principally inspired by Google's paper on Bigtable [44], which is used in their production environment for large data storage.

Data Model. In wide-column databases (figure 2d), data is represented in rows, with each row uniquely identified by a row key and composed of columns that can vary from one row to another.
Columns in each row are organized into one or more groups called column families, which are defined when the database is created.
Column families help categorize and physically group together columns that are logically related to each other. The column families can be physically distributed across different nodes.
Broadly speaking, wide-column databases can be considered a type of key-value store where the value is structured hierarchically, with the key as the root, column families as children, and these column families containing columns as their children.
Wide-column stores can store multiple versions of a column value using timestamps generated either automatically by the store system or manually by the client application.

Query Model. Queries in wide-column databases primarily perform key lookups. As previously explained, data in wide-column databases is represented in an aggregate format, with rows viewed as data hierarchies. Aggregate records in wide-column databases can be constructed using MapReduce [47] frameworks, which enable parallelized calculations over very large datasets across multiple machines. MapReduce is a low-level procedural language that can be challenging to maintain. Therefore, data analytics platforms such as Pig and Hive bridge the gap between declarative query languages and procedural MapReduce jobs by offering a high-level query language that can be compiled into a sequence of MapReduce jobs. While column family databases offer SQL-like queries, they are limited in expressivity, as only row keys and indexed values can be used in WHERE clauses. Currently, there is no common query language available for column-family databases.

Storage and Indexing Model. Typical wide-column databases use an LSM-tree data structure [31](figure 3). Data is first written to a commit log to prevent loss in case of failure, then stored in a memory buffer (referred to as MemTable in Bigtable [44]) in lexicographical order. When the buffer exceeds a certain size threshold, the data is persisted to storage as a file (referred to as SSTable in Bigtable) and cannot be altered afterwards. Since values in the stored file cannot be directly changed, updating a value involves creating a new version, and deleting a value involves creating a deletion marker to indicate that the value is considered deleted. Stored files (SSTables) are periodically compacted into a single file. This process groups together values belonging to a row, reconciles versions, and removes values marked as deleted. LSM-trees are efficient for random read operations, as reads are first applied to the buffer. Additionally, stored files are searched from the latest to the oldest, allowing access to the latest versions of values. To avoid searching for a non-existent key in a stored file, Bloom filters [3] are used. LSM-trees are more suitable for applications where insert and delete operations are more frequent than find operations. We have to precise that also key-value stores used LSM-trees as storage engine (like RocksDB, LevelDB, etc...) but they are not wide-column stores.

Use Cases. The flexibility, scalability, and high performance of wide-column databases, combined with MapReduce, make them suitable for Big Data storage and processing, real-time data processing, time series data storage, sensor data storage, and log data storage.

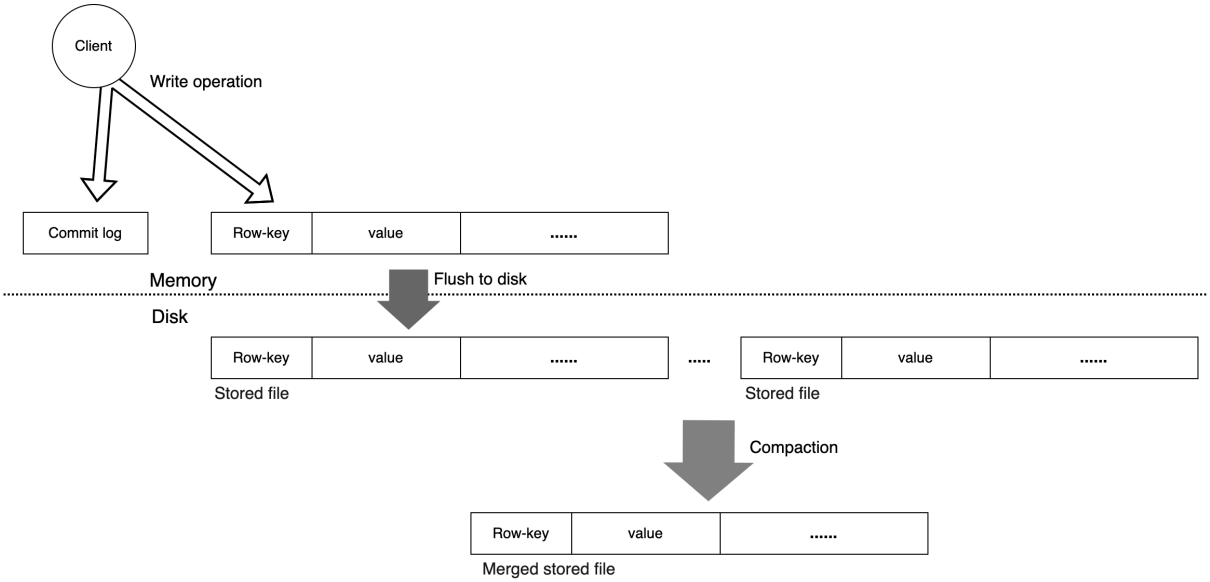

Figure 3: LSM-tree

Wide-columns stores categorization according to their main feature. We can globally classify wide-column stores into those that use a distributed file system to manage data distribution and those that handle data distribution independently (see Table 5).

- **Distributed Filesystem based Data Stores:** These are characterized by an underlying third-party distributed data store to which they delegate the functions of data distribution and replication. Examples include Bigtable [44] and HBase[33].

  Bigtable [44] was designed by Google to meet the requirements for a scalable, high-performance, and fault-tolerant storage system capable of handling massive amounts of structured data, as existing storage systems were insufficient. This system is capable of managing petabytes of data across thousands of commodity servers. Data is stored in SSTables (Sorted String Tables), and a distributed lock service, also developed by Google, manages metadata and coordination tasks. Built on top of the Google File System (GFS) [40], Bigtable uses a distributed master server for metadata and tablet assignments and has been used for applications such as Google Earth and Google Analytics.

  HBase, an open-source implementation of the Bigtable concept, plays a key role in the Hadoop ecosystem, seamlessly integrating with other Hadoop components such as Apache Hive and Apache Spark. Utilizing the concept of MemStore for in-memory data and HFiles for data stored in HDFS[34], HBase has a Java-based API and employs HDFS for its underlying distributed storage. It leverages Hadoop MapReduce for processing and analyzing large datasets and Apache Zookeeper for distributed coordination, leader election, and metadata maintenance about the cluster. While HBase does not have a specific query language, it automatically partitions and distributes tables into regions, each served by a RegionServer, and as data grows, regions are automatically split to maintain even data distribution and load.

  Accumulo[35], derived from the Bigtable design concept, emphasizes strong security, pro-

[33]https://hbase.apache.org/book.html
[34]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[35]https://accumulo.apache.org/

viding cell-level security access control that is more fine-grained than HBase's ACLs at the table and column family levels. Like HBase, Accumulo uses HDFS for distributed storage and supports ACID transactions. It integrates well with MapReduce and offers a powerful feature for customizing data processing called server-side iterators, allowing data modification during scans, such as transformation or aggregation with the dynamic creation of additional columns for newly processed data. Accumulo provides greater flexibility in data compaction strategies, allowing different levels of compaction for individual tables, making it a popular choice for applications requiring strong security features and fine-grained access control, such as those in government agencies and the defense sector.

Hypertable[36], also inspired by Google's Bigtable, uses HDFS as a storage backend. Unlike HBase, Hypertable employs a schema-based design where the schema is predefined. It features its own query language, HQL, and initially provided a Thrift-based API with language bindings for various programming languages. Additionally, Hypertable supports the use of Hadoop MapReduce jobs for processing and analyzing data stored within it.

Synthesis: We can say therefore that Distributed Filesystem based wide-column data Stores rely on a third-party distributed storage backend, such as GFS or HDFS, for data distribution and replication. They typically use a master-region server architecture (or tablet servers) for metadata management and table partitioning. Bigtable pioneered this approach, providing scalable, high-throughput storage for structured data, while HBase offers an open-source Hadoop-integrated alternative. Accumulo builds on HBase by adding cell-level security and fine-grained access control, making it suitable for sensitive applications. Hypertable emphasizes predefined schemas and provides its own query language (HQL), often used with Hadoop MapReduce. These systems excel in analytics and large-scale read/write workloads where fault-tolerant, coordinated storage is essential.

- **Standalone-based Data Stores:** These are characterized by having their own storage backend independent of a third-party distributed file storage system.

  Cassandra employs a peer-to-peer architecture where all nodes in the cluster have equal roles. Drawing inspiration from Google's Bigtable and Amazon's Dynamo papers, Cassandra combines the wide-column data model with Amazon's consistent hashing method, leading to a decentralized architecture. This design emphasizes high availability and partition tolerance, while offering tunable consistency levels to balance availability and consistency. Featuring a SQL-like query language called CQL (Cassandra Query Language), Cassandra supports multi-datacenter replication and integrates effectively with Apache Spark[37] for real-time data processing and analytics.

  ScyllaDB[38] is a wide-column data store similar to Cassandra but written in C++ for enhanced efficiency. It claims higher throughput and lower latencies compared to Cassandra on identical hardware. Using a shard-per-core architecture, each CPU core manages a shard or subset of data, optimizing for parallelism and multi-core architectures. Supporting the Cassandra Query Language (CQL), ScyllaDB facilitates seamless migration from Cassandra to ScyllaDB.

  Synthesis: We therefore can say that standalone-based wide-columns stores implement their own storage and distribution mechanisms, independent of a filesystem. They follow

[36] https://hypertable.com/
[37] https://spark.apache.org/docs/latest/
[38] https://docs.scylladb.com/stable/

a peer-to-peer, masterless architecture with consistent hashing and tunable consistency, inspired by Bigtable and Dynamo. Cassandra supports multi-datacenter replication and offers a SQL-like query language (CQL), balancing high availability, partition tolerance, and eventual consistency. ScyllaDB improves on Cassandra by implementing a shard-per-core, C++ design, reducing latency and increasing throughput, while remaining compatible with Cassandra's ecosystem. These systems are ideal for real-time, high-volume workloads requiring low-latency operations and horizontal scalability.

Table 5: Comparison of Wide-column Stores

Category	Store	Architecture	Storage Back-end	Distribution / Coor-dination	Query / API	Special Features	Typical Use Cases
Filesystem-based	Bigtable	Master + tablet servers	GFS + SSTables	Distributed lock service (Chubby) for metadata	Proprietary APIs (internal at Google)	Highly scalable, petabyte-scale, used internally at Google	Google Earth, Google Analytics, Search indexing
	HBase	Master + Region-Servers	HDFS (HFiles + MemStore)	ZooKeeper for coor-dination	Java API, integrated with Hadoop, no SQL	Auto-sharding, re-gion splits, MapRe-duce integration	Hadoop ecosystem, analyt-ics, real-time reads/writes
	Accumulo	Tablet servers (Bigtable-like)	HDFS	ZooKeeper + server-side iterators	Java API, Hadoop integration	Cell-level security, flexible compaction, ACID support	Government, defense, ap-plications requiring fine-grained access control
	Hypertable	Master + Range-Servers	HDFS	Hadoop MapReduce support, metadata services	HQL (Hypertable Query Language), APIs (Thrift-based)	Predefined schemas, custom language bindings	High-throughput apps, structured workloads
Standalone-based	Cassandra	Peer-to-peer (no master)	Native storage engine	Consistent hashing + gossip protocol, tunable consistency	CQL (SQL-like)	Multi-datacenter replication, tunable CAP trade-offs	Large-scale web apps, IoT, time-series, analytics
	ScyllaDB	Peer-to-peer, shard-per-core	Native storage engine (C++ optimized)	Same as Cassandra (gossip + partition-ing)	CQL (Cassandra-compatible)	High through-put, low latency, hardware-efficient	Drop-in Cassandra replace-ment, low-latency apps

5.4.3 *The Document stores*

The document-oriented data storage approaches (table 6) can be viewed as an implementation of the semi-structured data models described in [32]. Therefore, we will first explain the concept of semi-structured data before discussing the document data model, which is one of its implementations.

The Data Model. Semi-structured data (figure 2b,c) is represented as a self-describing hierarchical structure that can include nested objects, lists, and attributes. In this model, the schema is stored within the data itself, using structural or semantic tags to define the data structure. This approach is based on the idea of representing data without an explicit and separate schema definition.
Semi-structured data has the following properties:

- An irregular structure;
- An implicit structure (similar to HTML tags), which can be interpreted by the application or directly by the database;
- A partial structure, meaning parts of the data can be stored outside the database;
- A schema that is a-posteriori: Unlike traditional databases where the schema is predefined, in semi-structured data the schema is derived after the data exists. This is advantageous for integrating data from different sources with varying schemas. Until a fixed common schema is agreed upon, the a-posteriori schema allows for initial use of loose structures for different sources before finalizing a structure in the future [32];
- Rapid schema evolution: The schema can be updated flexibly and frequently, which is different from classical databases where schema updates are rare and costly. This is useful in scenarios where experimental techniques evolve, causing the resulting data schemas to change gradually.

The document structure is suited for storing data aggregates, meaning it groups together data usually represented in multiple records related by one-to-one or one-to-many relationships in the relational data model.
Depending on the database system, documents can be represented in formats such as JSON (e.g., in CouchDB, CouchBase), BSON (e.g., in MongoDB), or XML (e.g., in MarkLogic, Virtuoso).
In XML format [33], elements are delimited by tags that can contain simple text, sub-elements, or a combination of both.
XML was initially used for managing content in applications such as healthcare, science, and digital libraries, and for data exchange between nodes. JSON [70], defined as JavaScript Object Notation, originates from the object data type representation in the JavaScript programming language.
JSON-based NoSQL stores have gained popularity over XML due to their compactness, simplicity, and alignment with modern programming languages. They are commonly used in interactive and dynamic web applications.

The Query Model. The query languages for document stores are generally procedural. Depending on the data store technology, operations such as aggregation, grouping, MapReduce, and regular expressions filtering are available.

Storage and Indexing Mode. Document stores can represent data physically on drive in JSON, BSON and be stored in B-tree / B+ tree and also indexed locally B+ trees. However, there is a trend towards the use of Log-Structured Merge (LSM) trees for data storage. For distrbuted aspect, we can see section 5.5.

The Use Cases. Document stores are well-suited for various use cases, including Content Management Systems, Product Catalogs, User Profiles and Personalization, Event Logging and Time Series Data, and Internet of Things (IoT) Data.

Document stores categorized according to their main feature.

- **Standard Document Stores:** They are used for standard contents data access.

  MongoDB[39] became popular for web applications and projects requiring quick iteration and agile development. Storing data in a JSON-like format called BSON (Binary JSON), MongoDB supports a query language with syntax similar to JSON, making it easy to read and write, and thus intuitive and familiar to developers. A powerful aggregation framework is included for performing complex data transformations and computations. Various types of indexes are supported, such as single-field, compound, text, geospatial, and hashed indexes, using a B-tree data structure. MongoDB features automatic sharding and replication and includes a specification for storing and retrieving large files (called GridFS [40]) in a scalable manner. Initially using the MMAPv1 storage engine, MongoDB now employs WiredTiger as the default storage engine. MMAPv1 relies on memory-mapped files for storage, while WiredTiger enhances document-level concurrency control and supports on-the-fly compression of data on disk. Using an LSM-tree for data storage and a B-tree for primary and secondary indexing, WiredTiger is designed to handle mixed workloads efficiently, such as read-heavy and write-heavy scenarios. Multi-document ACID transactions are supported, and MongoDB benefits from a large and active community.

  RavenDB[41], similar to MongoDB but written in C#, uses a LINQ-based query language called RQL, resembling SQL and well-suited for .NET developers familiar with LINQ. Full-text search capabilities and ETL features for moving and transforming data between different databases and systems are included. RavenDB supports Map-Reduce [47] to perform complex data transformations and aggregations. Offering indexes such as MapReduce indexes, which provide real-time aggregation, means that as new documents are added or existing ones are updated, the index is incrementally updated, keeping results current. These indexes allow efficient querying of aggregated data without the need for costly and slow aggregations at query time, making it suitable for real-time analysis or reporting. Additionally, RavenDB supports standard static indexes and dynamic indexes, which adapt to varying query requirements. Multi-map indexes can combine data from multiple document types into a single index, and automatic indexing based on analysis of query patterns reduces the need for explicit index creation and maintenance. Geospatial and time series-based indexes are also supported. Using a storage engine called Voron, which relies on a B+tree data structure for organizing and storing data on disk and employs memory-mapped files for managing data storage, RavenDB supports ACID transactions with snapshot isolation and storage compression.

  Synthesis: We can say therefore that Standard document stores like MongoDB and RavenDB

[39] https://www.mongodb.com/docs/
[40] https://www.mongodb.com/docs/manual/core/gridfs/
[41] https://ravendb.net/

are optimized for web applications and general-purpose document management. They support JSON-like documents, indexing, and ACID transactions, with MongoDB emphasizing horizontal scaling and mixed workloads, while RavenDB integrates tightly with .NET and offers real-time indexing and aggregation.

- **Realtime Data Stores:** are better suited for applications requiring real-time data access.

  CouchDB[42] offers bi-directional replication suitable for decentralized data stores. Providing a RESTful HTTP API, CouchDB allows storage of attachments such as images and files along with documents. Supporting eventual consistency, CouchDB enables client usage in disconnected mode. Views can be created based on data processing with MapReduce functions, and CouchApps—web applications that can be served directly from the database—are supported.

  Couchbase[43] combines the document-oriented features of CouchDB with the distributed caching capabilities of Membase, resulting from the merger of CouchOne and Membase. Adopting a memory-first architecture, Couchbase prioritizes in-memory storage, representing data in JSON documents. Features for edge and mobile devices are available through Couchbase Lite, an embeddable data store for mobile devices, and Couchbase Sync Gateway, which enables real-time synchronization between Couchbase Lite nodes and the server. Couchbase Lite can act as a synchronized replica of selected server data, storing it locally on mobile devices, which is convenient for offline mode. It also synchronizes data from other Couchbase Lite nodes, using a conflict resolution mechanism to handle conflicts that may arise when disconnected from the Couchbase Sync Gateway. Couchbase offers primary and secondary indexes, utilizing local secondary indexes to limit indexing to specific nodes and Global Secondary Indexes to index data across all nodes. Covered index queries are supported, where the index contains all necessary information to satisfy the query, eliminating the need to access the concerned documents and improving performance. Partial indexes cover a subset of documents based on specific conditions. Automatic index creation optimizes queries, and geospatial indexing is supported. Indexes can be partitioned among nodes to distribute workloads, aiding horizontal scalability. Couchbase uses N1QL, a SQL-based query language, allowing powerful queries including joins and aggregations, and supports full-text search and multi-dimensional scaling by allowing independent scaling of services such as data, query, index, search, analytics, and eventing. The analytics service enables analytics and reporting based on the Couchbase Analytics service, using the SQL++ query language. The eventing service allows creating event-driven functions and triggers based on changes to data. Couchbase provides tunable consistency, allowing a balance between performance and consistency levels, specified for each query. This flexibility aids in scaling depending on different workload types. Consistent hashing is used for data distribution, and Couchbase supports multi-master replication, allowing write operations on both the primary copy and replicas. Cross-datacenter replication (XDCR) enables data replication to different locations, offering low-latency access to data regardless of user location.

  RethinkDB[44] is known for providing live, up-to-date query results in real-time, simplifying the development of real-time and collaborative applications.

[42] https://couchdb.apache.org/
[43] https://www.couchbase.com/
[44] https://rethinkdb.com

Mnesia[45], developed as part of the OTP (Open Telecom Platform) framework, is used in Erlang/OTP applications, particularly in telecommunication and distributed systems where fault tolerance, scalability, and real-time capabilities are critical. Primarily designed as a distributed, in-memory oriented storage, Mnesia also offers options for persistent storage on disk. Built on the Erlang VM, Mnesia is highly concurrent and supports ACID transactions. It is designed to support hot code upgrades in Erlang/OTP systems, allowing continuous operation during updates. Mnesia enables powerful queries based on Erlang's pattern matching capabilities and provides event-handling mechanisms, allowing applications to be notified of data changes in real-time, making it advantageous for reactive and event-driven systems.

Synthesis: We can say as synthesis that Realtime document stores such as CouchDB, Couchbase, RethinkDB, and Mnesia provide low-latency data access and replication. CouchDB enables offline/disconnected mode, Couchbase adds distributed caching and mobile/edge support, RethinkDB offers live query updates, and Mnesia targets high-concurrency Erlang applications with ACID compliance and event-handling.

- **Embedded Data Stores:** These are suitable for use in resource-constrained environments, making lightweight and easy-to-use embedded databases useful.

  LiteDB[46] is designed for various .NET environments, including desktop, mobile, and web applications. Supporting LINQ query types, LiteDB allows the storage of file attachments within the database, making it suitable for scenarios requiring the association of documents with external files. By default, data is automatically indexed to optimize query performance, and LiteDB is easily extensible.

  UnQLite[47] provides both key-value and document storage modes. Written in C, UnQLite integrates well with C applications. The query language uses a JSON-like syntax and supports basic CRUD (Create, Read, Update, Delete) operations along with full-text search capabilities. Leveraging memory-mapped files improves performance, and UnQLite supports extensions and customizations to meet specific use cases.

  Synthesis: We can say that Embedded document stores like LiteDB and UnQLite are lightweight, file-based systems suitable for resource-constrained environments. They provide simple query mechanisms, ACID compliance, and often support both document and key-value modes, enabling local storage in desktop, mobile, or embedded applications.

[45] https://www.erlang.org/doc/apps/mnesia/mnesia.html
[46] https://www.litedb.org/
[47] https://unqlite.symisc.net/

Table 6: Comparison of Document Stores

Category	Store	Architecture	Storage Backend	Distribution / Coordination	Query / API	Special Features	Typical Use Cases
Standard Document Stores	MongoDB	Client-server	WiredTiger (default), MMAPv1 (legacy), BSON	Automatic sharding and replication	JSON-like query language, aggregation framework, B-tree indexes	ACID transactions, GridFS [48] for large files, mixed workloads	Web apps, rapid development
	RavenDB	Embedded / Client-server	Voron (B+Tree + mmap)	Replication, distributed coordination	LINQ-based RQL, Map-Reduce, dynamic/static indexes, geospatial/time-series	ACID transactions, automatic indexing, real-time aggregation	.NET apps, ETL, reporting
Realtime Data Stores	CouchDB	Client-server, RESTful HTTP API	Attachments + JSON documents	Bi-directional replication, offline mode	MapReduce views	Eventually consistent, CouchApps	Decentralized apps, offline-capable apps
	Couchbase	Memory-first, distributed caching	JSON documents	Multi-master replication, XDCR	N1QL (SQL-like), primary/secondary/-global indexes, full-text search	Tunable consistency, mobile/edge support via Couchbase Lite, analytics, eventing	Real-time apps, mobile apps, analytics dashboards
	RethinkDB	Client-server, in-memory + disk	JSON documents	Automatic push updates to clients	Realtime queries, live feeds	Live query results for clients	Collaborative apps, dashboards, notifications
	Mnesia	Distributed in-memory (Erlang/OTP)	In-memory with optional disk persistence	Fault-tolerant, hot code upgrades	Pattern matching queries	ACID transactions, event-handling	Telecom, distributed systems, reactive apps
Embedded Data Stores	LiteDB	Embedded (.NET)	Single file database	N/A	LINQ queries, automatic indexing	Stores attachments, lightweight	Desktop, mobile, web apps
	UnQLite	Embedded (C)	Memory-mapped files	N/A	JSON-like CRUD queries, full-text search	Key-value + document storage, lightweight	C applications, embedded systems

[48] https://www.mongodb.com/docs/manual/core/gridfs/

5.4.4 *The graph-oriented stores*

Graph databases [43, 79, 86] (table 7) are database systems designed to store, query, and manage graph-structured data. Instead of using rigid tables (as in relational DBs), graph databases natively store vertices (nodes) and edges (relationships), often enriched with labels and properties. They excel in handling highly connected data, such as social networks, knowledge graphs, or recommendation systems.

The advantages of adopting graph data stores are:

- **More natural modeling:** Graph structures are visible to the user, with all information about an entity kept in a single node, facilitating easy access to information about all its connections.
- **Queries directly referring to the graph structure:** Specific graph operations, such as searching for shortest paths and identifying subgraphs, can be integrated into the query language algebra. Graph operations allow for high-level query abstraction, unlike navigational data models such as hierarchical and network models that require knowledge of low-level physical operations for navigation.
- **Dedicated data structures for graph storage and efficient graph algorithms:** These are designed for graph-oriented operations.

There are multiple variations of graph databases, including classical graph databases, graph streaming frameworks, and graph processing systems, described as follows:

- **Classical graph databases,** also known as Transactional graph stores, focus on dynamic graph-based data and are suited for queries typically applied to a local part of the data.
- **Graph processing systems,** also known as Analytical graph stores, focus on static and simple graphs with an emphasis on global graph analytics (e.g., PageRank [35]). Graph processing is used in diverse areas such as machine learning, computational sciences, medical applications, social network analysis, and more.
- **Graph streaming frameworks** input graphs as a stream of data, simplifying the addition and removal of edges. They focus on simple graph models where edges or vertices may have weights and, in some cases, simple additional properties such as timestamps [107].

Some graph stores are hybrid, offering both transactional and analytical capabilities. This text focuses on classical graph databases. Modern graph databases in production tend to be large and highly dynamic, often experiencing regular structural changes. Graph stores face challenges in distributing components of graphs among multiple servers because data must be stored to minimize cross-node graph traversals [84, 102]. They also lack a standardized query language. Some consider RDF databases as graph databases, but the primary model used for practical graph storage is the Labeled Property Graph (LPG) data model [83]. Representing RDF-based data in LPG graphs is challenging because in RDF, the predicate can be a subject for another triple, requiring the predicate to be converted into a node in the LPG graph.

Data Model. Graph-oriented stores (figure 2e) draw inspiration from graph theory and are efficient in storing data requiring advanced manipulation of links between data, especially where the density of links is higher than in relational stores, necessitating special storage and access modes for efficient traversal among data. The graph data model represents data as nodes (vertices) connected by links (edges). Graphs can have directed or undirected edges, with edges and nodes being labeled or unlabeled.

Several proposals for graph data model variants exist, but the predominant one is the Labeled Property Graph (LPG) data model, consisting of nodes and edges categorized with labels and having one or many properties represented by key-value pairs.

The query model. Data manipulation in graph databases is performed using graph-oriented operators. Graph databases can be queried in two general ways: graph pattern matching and graph traversal.

- **Graph pattern matching:** Involves finding paths that match a given graph pattern.
- **Graph traversal:** Involves traversing the graph according to a given description, which can use either breadth-first or depth-first traversal methods.

Most graph databases can be accessed through REST APIs. Unlike other types of NoSQL databases, several standard query languages are shared among graph databases, such as Gremlin[49], Cypher [85], and SPARQL[50] [49], which have graph pattern matching capabilities. SPARQL is a declarative query language designed to query RDF graphs, while Cypher is a declarative query language designed for labeled property graph databases. Gremlin [76], on the other hand, is a graph traversal query language whose style is more imperative and functional.

Storage and Indexing Methods. Graphs can be represented using native methods or by employing other database types, such as relational or NoSQL-based data models.

There are different representations of graphs, such as:

- **Adjacency matrices:** Simple representation of vertex connections, efficient for dense graphs but memory-intensive for large sparse graphs.
- **Compressed Sparse Row (CSR):** Stores neighbor lists compactly in arrays, widely used for sparse graphs due to space efficiency and fast traversal.
- **Compressed adjacency lists:** Variants that reduce storage using delta encoding, re-pairing, or reference-based compression, suitable for large graphs with locality in vertex IDs.
- **K^2-trees:** Succinct representation of adjacency matrices using recursive partitioning, enabling compact storage of very large sparse graphs (e.g., web graphs).
- **Labeling schemes for reachability:** Store additional information with nodes (e.g., 2-hop labeling) to efficiently answer reachability queries without full graph traversal.
- **Structural summaries:** Techniques such as simulation or bisimulation that group equivalent nodes or subgraphs, both compressing the graph and accelerating pattern queries.

For connectivity between records (vertices), two methods are possible:

- **Store direct pointers to the concerned records:** This provides fast access to records without needing an index to find their physical location, making it efficient for graph traversal.

[49] https://tinkerpop.apache.org/docs/current/reference/
[50] https://www.w3.org/TR/sparql11-query/

- **Store unique IDs of the concerned records:** This requires indexing these IDs to look up the physical location of the records. While this method allows for changes in the physical location of records without changing the unique ID, it slows down access due to the need for index lookups before accessing the records. This is efficient when the referred records frequently change location with updates.

Sometimes, graph data is stored directly in an index, such as in triple stores (RDF). An example is Jena TDB.

We can categorize different types of indexes based on their role:

- **Neighborhood-Based Indexes:** Used to speed up access to adjacency lists to accelerate traversal queries.
- **Data Indexes:** Designed to index data for quick retrieval and processing.

Use Cases. Graph databases are suitable for highly interconnected data and are efficient in traversing relationships. They are used in complex network-based applications such as social networks, telecommunications, biological networks, recommendation systems, pattern recognition, dependency analysis, pathfinding solutions for navigation systems, web, geographical systems, transportation, social and biological networks.

Graph stores categorized according their main feature.

- **Classical graph stores:** they represent standard graph stores designed to efficiently store, manage, and query graph structures.

  Neo4j[51] was developed to handle interconnected data, such as social networks and recommendation engines. Originating as a native graph storage system, Neo4j is currently the most representative among graph data stores. Based on the Labeled Property Graph (LPG) data model, Neo4j uses the declarative query language Cypher, which has become a standard for some other graph data stores. Cypher supports query types including pattern matching, traversal, and filtering, and provides aggregation functions and sorting capabilities. Real-time subscriptions to data changes for specific queries are also supported. Neo4j offers various types of indexing, such as simple, composite, spatial, and full-text indexing, and supports ACID transactions. Neo4j separates nodes, relationships, labels, properties, and indexes into different stores. It supports two clustering modes: shared file system clustering, with a designated master node for write/read operations and replication nodes restricted to read-only; and causal clustering, a newer mechanism based on the Raft consensus algorithm, treating all nodes as equal replicas that support both read and write operations with strong consistency guarantees.

  JanusGraph[52]supports the graph property data model and is built on top of Apache TinkerPop, a graph computing framework. Designed for high scalability and distribution, JanusGraph supports automatic sharding of data across multiple nodes and uses the TinkerPop Gremlin query language, an imperative graph traversal language. It supports ACID transactions, indexing mechanisms, and full-text search via Apache Lucene and Elasticsearch. With a masterless architecture using consistent hashing, JanusGraph supports multiple storage backends, such as Apache Cassandra, Apache HBase, and Google Cloud Bigtable, allowing users to choose the backend that best fits their specific requirements.

[51] https://neo4j.com/
[52] https://docs.janusgraph.org/

It can parallelize query execution across multiple nodes with partition-aware query planning. Vertex-centric indexing supports scalable and distributed indexing, and JanusGraph also supports graph analytics algorithms like community detection, centrality analysis, and pathfinding.

Dgraph[53] is a property graph-based data store with a distributed architecture at its core. Using GraphQL±, a graph query language that extends GraphQL, Dgraph represents graph data using RDF triples. Unlike classical RDF, Dgraph allows RDF predicates linking a subject and object to have special properties called facets. An optional schema specifies types, predicates (properties), and indexes. Supporting ACID transactions, Dgraph uses snapshots to provide a consistent view of data in transactions and queries. Faceted search and filtering are supported, and integration with the full-text search engine Bleve enhances search capabilities. Dgraph offers two key-value stores as storage engines: Badger and RocksDB. Both use an LSM (Log-Structured Merge) tree structure, efficient for write-intensive workloads. Badger, closely integrated with Dgraph, provides efficient storage and retrieval, while RocksDB offers more options for handling data persistence effectively with the Write-Ahead-Log mechanism. Badger is suitable for high-speed operations (adapted to SSDs), and RocksDB for large datasets (adapted to disk storage). Dgraph automatically indexes node properties and supports exact and full-text indexing. The Raft consensus algorithm ensures fault tolerance and consistency in a distributed environment, allowing reads and writes from clients on any node. Write operations on replicas are coordinated by a leader node elected through the Raft consensus mechanism. Distributed transactions are supported, and predicate-based sharding is used to distribute data across nodes, associating predicates (edge labels or property names) with a set of shards. Dgraph's query planner analyzes queries to determine the most efficient execution method across the distributed cluster and employs techniques to reduce network overhead, such as selective data fetching and efficient results aggregation.

Synthesis: We therefore can say that classical Graph Stores like Neo4j, JanusGraph, and Dgraph are optimized for general-purpose graph workloads. Neo4j is a native graph database, ideal for connected data and real-time queries with ACID compliance. JanusGraph focuses on scalability and supports multiple storage backends, making it suitable for large distributed graphs. Dgraph provides a fully distributed property graph system with predicate-based sharding and GraphQL± for high-throughput analytics.

- **In-memory graph stores:** These systems keep the entire graph (or its active working set) in main memory rather than on disk. This design enables extremely fast traversals and queries, with very low latency and high throughput. However, it comes at the cost of higher memory consumption and often limited dataset size, making them best suited for real-time analytics, recommendation engines, or scenarios where speed is more critical than persistence.

  Memgraph[54] is a property graph data store optimized for in-memory storage and processing, effectively handling high-performance graph queries and real-time analytics. Memgraph supports the Cypher query language, based on Neo4j's Cypher. Features include automatic sharding, distributed query parallelization, and dynamic scaling, suitable for scenarios requiring rapid data processing and analysis.

[53] https://dgraph.io/docs/
[54] https://memgraph.com/docs

- **Graph Semantics-oriented graph stores:** are design to support data wich rich semantics. They often utilize standards such as RDF and support SPARQL query languages.

  Ontotext GraphDB[55] is a semantic graph database based on the RDF data model. As a native RDF database, Ontotext GraphDB supports SPARQL, OWL (Web Ontology Language) reasoning, and rule-based inference, enabling users to define custom rules to infer new facts from existing data. The database also supports the SeRQL (Sesame RDF Query Language) and stores data in a proprietary binary format, facilitating the use of Named Graphs and optimizing large datasets. Using B+Tree indexes for RDF data indexing and searching, it can also utilize bitmap indexes for specific filtering operations. Separate indexes are maintained for subjects, predicates, and objects, known as SPO indexes. Integration with Lucene provides textual indexing, and geo-spatial indexing and clustered indexes are supported, allowing indexes to be distributed across nodes. Relevance ranking assigns scores to query results based on relevance to search criteria. Users can customize indexing settings based on specific requirements. Ontotext GraphDB employs in-memory caching, supports Snapshot Isolation for consistency, and MVCC for concurrent transaction handling. Integration with other Ontotext platform components, including text analytics and semantic annotation tools, provides a comprehensive solution for structured and unstructured data. Graph analytics on stored RDF data can be performed, and support for Linked Data principles enables data interlinking across different web sources. Data distribution across multiple nodes for large-scale datasets is supported, and users can choose between strong and eventual consistency. Versioning manages changes to the graph over time, and for distribution, various sharding strategies, such as hash-based and range-based, are supported. The Primary-secondary replication mechanism ensures queries are routed to the appropriate nodes containing relevant data. Commonly used in applications involving semantic web technologies and linked data.

  Stardog[56] serves as an enterprise knowledge graph platform. Supporting SPARQL for queries and OWL reasoning, Stardog also accommodates the property graph data model and the Gremlin query language. Additionally, it supports the GraphQL query language, which can map to RDF or property graph data models. A notable feature is data virtualization, allowing users to query and access data from different sources as if stored in a single, unified database. This includes relational databases, NoSQL databases, web services, and other storage solutions, enabling organizations to build knowledge graph capabilities using their existing infrastructures. Stardog provides a unified view of diverse data, supports temporal reasoning for modeling and querying time-dependent information, and integrates full-text search and geo-spatial indexing capabilities. Utilizing triple indexes that combine hash-based and tree-based indexing, Stardog also supports composite indexes by combining multiple RDF triple components. Various storage backends, such as local disks, networked file systems, or cloud-based solutions, are supported. Consistent hashing distributes data across nodes, with replication being Primary-secondary and either synchronous or asynchronous. Stardog is primarily used for enterprise knowledge graph and data integration scenarios.

  AllegroGraph[57] is similar to Ontotext GraphDB but includes support for GeoTemporal reasoning, allowing querying of data with spatial and temporal dimensions. AllegroGraph supports the RDF data model and SPARQL query language, integrates well with Common

[55] https://graphdb.ontotext.com
[56] https://docs.stardog.com/
[57] https://allegrograph.com/

Lisp (a powerful and extensible programming language), and includes a built-in Prolog interpreter for rule-based reasoning and inference.

Blazegraph[58] is designed for massively parallel data processing, known for high throughput and low latency. Supporting GPU acceleration for parallel processing on certain graph algorithms and analytics, Blazegraph efficiently handles large-scale graphs and complex queries. The RDF data model is used, with engineering for high performance and scalability. Inferences based on both RDFS and OWL are supported, along with the SPARQL query language. GeoSPARQL enables storage, querying, and analysis of geospatial information. Blazegraph supports read and write operations on multiple versions of a graph for scenarios requiring historical or versioned data. Graph analytics features offer advanced computations on graph data, such as graph traversal and pattern identification, with integration capabilities for Apache Spark for large-scale analytics and processing. Distribution is supported using consistent hashing, and the Primary-secondary-based replication mode facilitates geo-replication and cluster deployment in different geographical locations.

Synthesis: We can say that graph Semantics-oriented Stores (Ontotext GraphDB, Stardog, AllegroGraph, Blazegraph) emphasize RDF and semantic data management. They support SPARQL, reasoning with OWL or Prolog rules, and advanced features like temporal or geospatial queries. These stores are tailored for enterprise knowledge graphs, linked data, and applications requiring rich semantic querying and inference, with strong consistency and distributed capabilities.

[58] https://github.com/blazegraph/database/wiki/Main_Page

Table 7: Comparison of Graph Stores

Category	Store	Architecture	Storage Backend	Distribution / Coordination	Query / API	Special Features	Typical Use Cases
Classical Graph Stores	Neo4j	Native graph storage, Labeled Property Graph	Separate stores for nodes, relationships, properties	Shared file system clustering or causal clustering (Raft)	Cypher	ACID transactions, indexing (simple, composite, spatial, full-text), real-time subscriptions	Social networks, recommendation engines
	JanusGraph	Distributed, masterless, property graph	Multiple storage backends (Cassandra, HBase, Bigtable)	Automatic sharding, partition-aware query planning	TinkerPop Gremlin	Vertex-centric indexing, full-text search via Lucene/Elasticsearch, ACID transactions	Large-scale distributed graphs, analytics
	Dgraph	Distributed property graph, RDF triples	Badger / RocksDB (LSM trees)	Raft consensus, predicate-based sharding, distributed transactions	GraphQL±	Faceted search, automatic indexing, ACID transactions, high-speed writes	Scalable graph applications, real-time queries
In-Memory Graph Stores	Memgraph	In-memory, property graph	In-memory storage	Automatic sharding, distributed query parallelization	Cypher	Real-time analytics, high-performance queries	Real-time graph analytics, streaming data applications
Graph Semantics-Oriented	Ontotext GraphDB	RDF triple store	B+Tree indexes, SPO indexes, bitmap indexes	Primary-secondary replication, sharding (hash/range)	SPARQL, SeRQL	OWL reasoning, rule-based inference, full-text search, geospatial indexing	Semantic web, linked data, enterprise knowledge graphs
	Stardog	Knowledge graph platform, RDF + property graph	Triple indexes, hash/tree-based, composite indexes	Consistent hashing, Primary-secondary replication	SPARQL, Gremlin, GraphQL	Data virtualization, temporal reasoning, full-text and geospatial indexing	Enterprise knowledge graphs, data integration
	AllegroGraph	RDF triple store with geotemporal reasoning	Triple indexes	Primary-secondary replication	SPARQL	Prolog-based inference, GeoTemporal reasoning	Semantic web, spatio-temporal analytics

Continued on next page

Table 7 – continued from previous page

Category	Store	Architecture	Storage Backend	Distribution / Coordination	Query / API	Special Features	Typical Use Cases
	Blazegraph	RDF triple store, high-performance	LSM-based storage	Consistent hashing, Primary-secondary replication	SPARQL	GPU acceleration, RDFS/OWL inference, GeoSPARQL, graph analytics	Large-scale RDF analytics, complex queries, historical graph data

For further information about the properties of the different graph stores, see the table **??**.

5.4.5 Multi-model NoSQL data stores

The multi-model data stores (table 8) categorized here are designed to natively manage multiple data models. Table 8 describes some multi-model oriented data stores.

Virtuoso[59] [50] is known for its versatility, supporting both SQL and SPARQL while managing data models such as relational data, the RDF data model for the semantic web, and document data models based on XML or JSON, as well as key-value data models. It includes full-text search capabilities and supports various types of indexes, including full-text indexes, geospatial indexes, clustered indexes (which organize data physically based on the order of the index), uniqueness indexes, filtered indexes, and bitmap indexes. Virtuoso also offers linked data features. It maintains a Primary-secondary replication system that can be either synchronous or asynchronous. Various sharding strategies are allowed, including range-based partitioning, hash-based partitioning, and other custom strategies. Geospatial data can be queried with GeoSPARQL. Virtuoso's data virtualization feature allows the integration of external relational databases and web services.

MarkLogic[60] is well-known as a document-centric data store but also supports the RDF triple data model and the property graph data model. It permits the storage of geospatial data. Documents, triples, graphs, and geospatial data can coexist in the same database instance. MarkLogic provides full-text search capabilities and supports ACID transactions, ensuring data integrity and consistency. It also allows data integration from various sources.

ArangoDB[61] [106] is primarily a document-oriented data store but also supports graph and key-value data models. With ACID transactions, it ensures data integrity and consistency. It supports a declarative query language called AQL (ArangoDB Query Language), enabling joins between collections and multi-collection transactions. ArangoDB includes a microservices framework called Foxx, allowing users to develop and deploy JavaScript-based microservices within the database. It supports clustering based on a Primary-secondary architecture and has a shard coordinator to distribute data across shards. The shard key is a document attribute or a set of attributes chosen by the user. It also supports full-text search and geospatial indexing.

OrientDB[62] supports multiple data models, including document, graph, key-value, and object-oriented models. It uses a SQL-like query language and supports both schema-less and schema-full modes. OrientDB allows polymorphic queries, enabling data querying across different classes that share common properties. It supports ACID transactions and employs a multi-master replication strategy. For distribution, shard keys can be based on one or multiple properties. In terms of indexing, it supports full-text indexing and geospatial indexing. OrientDB also supports triggers and functions, allowing the definition of logic based on certain events or conditions.

Synthesis: We can say therefore that the multi-model databases have the purpose to combine the different strengh of the data models. Virtuoso is versatile, handling relational, RDF, document, and key-value data, with strong indexing, sharding, and data integration features. MarkLogic focuses on documents and semantic data, offering full-text search, geospatial queries,

[59] https://docs.openlinksw.com/virtuoso/
[60] https://www.progress.com/marklogic
[61] https://arangodb.com/
[62] https://orientdb.dev/

and ACID transactions. ArangoDB combines document and graph models, supports multi-collection queries with AQL, ACID transactions, clustering, and microservices. OrientDB supports document, graph, key-value, and object-oriented models, allowing flexible queries, multi-master replication, triggers, and schema options.

Table 8: Comparative summary of multi-model data stores including architecture, storage backend, special features, and typical use cases

Store	**Architecture**	**Storage Backend**	**Data Models**	**Query / API**	**Special Features**	**Typical Use Cases**
Virtuoso	Multi-process, shared-nothing	Disk-based	Relational, RDF, Document (XML/JSON), Key-Value	SQL, SPARQL	Linked data, multiple sharding strategies, data virtualization	Linked data, semantic web, integration of multiple data sources
MarkLogic	Server-based, document-centric	Disk-based	Document, RDF triples, Property Graph, Geospatial	Built-in query API, SPARQL	Full-text search, multi-model coexistence, geospatial support	Multi-model storage, geospatial data, enterprise integration
ArangoDB	Clustered Primary-secondary	Disk-based, sharded	Document, Graph, Key-Value	AQL (joins, multi-collection), Foxx microservices	Microservices framework, multi-collection transactions, shard coordinator	Multi-model applications, microservices, complex queries
OrientDB	Multi-master cluster	Disk-based, sharded	Document, Graph, Key-Value, Object-Oriented	SQL-like, polymorphic queries	Triggers, functions, schema-less/schema-full support	Polymorphic queries, schema-less/schema-full applications, multi-model enterprise use

5.4.6 *Cloud managed NoSQL data stores*

Cloud-managed data stores (table 9), similar to standalone NoSQL data stores, offer the advantage of eliminating the complexity of manually deploying a data store infrastructure.
Cloud-based storage allows resources to be scaled based on demand, with users paying only for the throughput and storage consumed. Cloud providers typically handle maintenance tasks such as backups, updates, and security patches. These services often provide high availability and disaster recovery options, ensuring data durability and minimizing downtime. They also offer integration with other cloud services, such as analytics, machine learning, and monitoring tools.

Azure Cosmos DB[63] is a multi-model, globally distributed database service. It supports multiple data models, including document (using JSON), key-value, graph (with Gremlin), and wide-column (with Cassandra). It enables users to replicate and distribute data across multiple regions. Indexing is automatic and instantaneous as data is ingested. It supports SQL-like queries. Multiple consistency models are offered, including strong, bounded stateless, session, and eventual consistency, depending on the requirements. It also supports APIs from different data models, such as MongoDB, Cassandra, Gremlin, and Azure Table Storage. It provides automatic scaling. In terms of replication, it supports multi-master replication, allowing writes to any region. The change feed mechanism allows users to track data changes in real-time. In terms of compatibility, it integrates with various Azure services, including Azure Functions, Azure Logic Apps, and Azure Synapse Analytics.

Amazon DynamoDB[64] is a key-value and document database cloud service from AWS (Amazon Web Services). It allows cross-region replication and offers consistent and predictable performance with low-latency response times. It integrates seamlessly with other AWS services such as AWS Lambda, Amazon S3, AWS CloudTrail, and Amazon CloudWatch. Using DynamoDB Streams, it captures changes to the data in real-time, enabling the creation of event-driven architectures by triggering AWS Lambda functions. It is suitable for applications with unpredictable workloads.

Google Cloud Bigtable[65] is designed to handle large volumes of data with low-latency reads and writes. It is therefore suited for applications that require high-throughput and real-time access to large datasets. It is a wide-column store that is horizontally scalable and supports global replication and automatic sharding. It is compatible with the Apache HBase API and can be used for storing and analyzing time-series data. It offers good integration with other Google Cloud services, such as BigQuery, Cloud Storage, Cloud Pub/Sub, and Dataflow.

Google Cloud Firestore[66] stores data in a JSON-like format and is a fully managed service provided by Google Cloud. It provides real-time synchronization and updates for data changes, allowing clients to subscribe to changes. It scales horizontally and supports multi-region data distribution. It supports multiple platforms, including mobile and server environments, and supports atomic transactions. One of its particular features is support for offline data access, allowing mobile and web applications to continue functioning even when offline. It is part of the Firebase platform, which includes services such as Firebase Authentication, Cloud Functions, Hosting, and more. It can also be integrated with other Google Cloud services.

[63] https://learn.microsoft.com/enus/azure/cosmosdb/
[64] https://docs.aws.amazon.com/dynamodb/
[65] https://cloud.google.com/bigtable/docs
[66] https://cloud.google.com/firestore/docs

Amazon Neptune[67] is a fully managed graph database provided by AWS (Amazon Web Services). It supports both property-graph and RDF data models. It is scalable and offers automatic replication. It supports ACID transactions. Languages such as Gremlin and SPARQL can be used for queries. It provides automatic indexing and real-time graph queries. It supports data ingestion from various sources, such as Amazon S3 and AWS Data Pipeline, and integrates well with other AWS services, such as AWS Lambda, Amazon S3, and Amazon CloudWatch.

Azure Table Storage[68] stores data in a key-value format and is fully managed. The data content is stored in tables represented as collections of properties with a flexible schema. It is scalable and sharded using a composite key composed of a partition key and a row key, which are automatically indexed. It provides low-latency access to data, making it suitable for applications that require quick retrieval and storage of key-value pairs. It supports geo-replication and can be accessed through a RESTful API. It integrates with other Azure services, such as Azure Functions, Azure Logic Apps, and Azure Event Grid.

Synthesis: Therefore, we can say that cloud-managed data stores simplify infrastructure management while offering scaling and global distribution. Their strengths differ by data model support, performance, replication, and platform integration. Azure Cosmos DB stands out for multi-model support, global distribution, and flexible consistency, making it suitable for applications needing versatile workloads and strong replication. Amazon DynamoDB excels in predictable low-latency key-value/document operations and event-driven architectures, ideal for highly variable workloads. Google Cloud Bigtable is optimized for massive datasets and real-time analytics, especially time-series and high-throughput applications. Google Cloud Firestore provides real-time synchronization, offline support, and seamless mobile/web integration, making it strong for interactive applications. Amazon Neptune focuses on graph workloads, supporting property-graph and RDF models with automatic replication and real-time queries, suited for relationship-centric analytics. Azure Table Storage is simple, low-latency key-value storage with automatic sharding, best for lightweight scalable applications.

[67] https://docs.aws.amazon.com/neptune/
[68] https://learn.microsoft.com/en-us/azure/storage/tables/

Table 9: Comparative summary of cloud-managed NoSQL data stores including data models, architecture, replication/distribution, special features, and typical use cases

Store	Data Models	API / Query	Architecture	Replication / Distribution	Special Features	Typical Use Cases
Azure Cosmos DB	Document, Key-Value, Graph, Column-Family	SQL-like, MongoDB, Cassandra, Gremlin, Azure Table	Globally distributed, multi-master, automatic scaling	Multi-region replication	Automatic indexing, multiple consistency models, change feed, integrates with Azure services	Multi-model, globally distributed applications, real-time change tracking
Amazon DynamoDB	Key-Value, Document	DynamoDB API	Cloud-managed	Cross-region replication	Low-latency reads/writes, real-time change capture via DynamoDB Streams, integrates with AWS services	Event-driven apps, unpredictable workloads, scalable key-value/document applications
Google Cloud Bigtable	Wide-column	HBase API compatible	Horizontally scalable, cloud-managed	Global replication, automatic sharding	High-throughput analytics, time-series optimized	Large-scale analytics, time-series data, real-time high-throughput applications
Google Cloud Firestore	JSON-like documents	Firestore API	Horizontally scalable, cloud-managed	Multi-region distribution	Real-time synchronization, offline support, part of Firebase	Mobile/web apps with real-time updates, offline-first applications
Amazon Neptune	Property-graph, RDF	Gremlin, SPARQL	Fully managed, scalable	Automatic replication	Real-time graph queries, automatic indexing, integration with AWS services	Relationship-centric applications, graph analytics, knowledge graphs
Azure Table Storage	Key-Value	RESTful API	Fully managed, cloud-based	Sharded with partition + row key, geo-replication	Flexible schema, low-latency access	Lightweight key-value workloads, scalable table storage

5.5 Efficient data structures for storage and access optimizations

This section describes the physical organization of data in NoSQL stores, focusing on how storage and distribution strategies enable optimal query performance. NoSQL systems rely on both local and distributed data structures to balance efficiency, scalability, and fault tolerance.

For efficient local range queries, B+ trees are commonly used, while hash tables are preferred for fast random access. On the distributed side, NoSQL stores employ global data structures such as Distributed Hash Tables (DHTs) [38] for partitioning (e.g., Amazon Dynamo, Apache Cassandra, Riak), Log-Structured Merge Trees (LSM-Trees) for write-optimized storage (e.g., Google Bigtable, HBase, RocksDB, LevelDB), and distributed skip lists for ordered data access (e.g., MemSQL/SingleStore).

Local Data Structures in NoSQL Stores. Local data structures are **internal data representations** used within a single node or shard to manage storage and retrieval efficiently. They are not inherently distributed; each node maintains them independently.

Examples:

- **B-Trees / B$^+$-Trees** → used in MongoDB (MMAPv1 engine) and many storage engines for indexing.
- **LSM-Trees (Log-Structured Merge Trees)** → used in Cassandra, HBase, RocksDB, LevelDB; optimized for write-heavy workloads.
- **Hash Tables** → used in Redis, DynamoDB internals, and some key-value stores for fast key lookups.
- **Tries / Radix Trees** [2] → used in Redis for prefix search and memory efficiency.
- **Skip Lists** → used in Redis for sorted sets (ZSET).
- **In-memory data structures** → Redis offers lists, sets, sorted sets, and bitmaps as *local structures* per node.

Role: Optimize local performance (read/write efficiency, indexing, compression, caching).
Limitation: Operate only within a node; require coordination mechanisms for global consistency.

Managing Local Data Structures Globally in NoSQL. NoSQL systems combine local data structures (e.g., LSM-Trees, B-Trees, Hash Tables) into a global distributed system using several strategies:

1. **Partitioning (Sharding):** Keys are distributed across nodes using consistent hashing or range partitioning. *Examples: Cassandra (DHT [38]), HBase/Bigtable (range partitioning).*
2. **Replication:** Local structures are duplicated across nodes for fault tolerance and availability. *Consistency: eventual (Cassandra), tunable, or strong (Spanner).*
3. **Coordination:** Metadata services (e.g., ZooKeeper, Gossip protocols) track cluster state and guide query routing.
4. **Global Indexing:** Local indexes are combined into distributed ones for search or secondary lookups. *Examples: Elasticsearch, MongoDB.*
5. **Compaction and Repair:** Local compaction plus global repair keep data consistent and balanced.
6. **Query Routing:** Coordinators direct queries to the appropriate nodes transparently. *Examples: Dynamo-style coordinators, Bigtable masters.*

Key Insight

NoSQL stores do not invent new global data structures; instead, they:

- **Partition** the keyspace,
- **Replicate** local structures, and
- **Coordinate and route queries** to provide a unified distributed system.

It is important to clearly distinguish between global data distribution mechanisms and local data access optimization techniques in NoSQL environments. Distributed storage systems such as **DHT (Distributed Hash Table)** [38], **GridFS** [69], and **HDFS** implement *sharding strategies*, which partition and distribute data across multiple nodes to ensure scalability and fault tolerance. In contrast, data structures such as **LSM-Trees**, **hash indexes**, and **radix trees** are used for *local indexing*, aiming to optimize query performance and data retrieval within individual storage nodes. Thus, while sharding addresses the *global organization and distribution* of data, local indexing focuses on the *internal efficiency of data access* at the node level.

Distributed Hash Tables (DHTs) [38] distribute keys evenly across nodes using consistent hashing, offering excellent scalability, fault tolerance, and $O(1)$ lookups. They are efficient for key-value access but do not preserve ordering, making range queries impossible. Log-Structured Merge-Trees (LSM-Trees) [31] are write-optimized structures that buffer updates in memory and flush them sequentially to disk, providing very high write throughput, compression efficiency, and support for range queries. Their drawbacks are slower reads due to multiple SSTables and costly compaction overhead. Distributed Skip Lists maintain keys in sorted order with probabilistic linked levels, offering predictable $O(\log n)$ operations, good concurrency, and natural support for range queries. However, they are less write-optimized than LSM-Trees, consume more memory, and lack the strong fault tolerance of DHT-based approaches.

The following Tables 11 and 12 summarize the performance of these data structures for different query types.

[69] https://www.mongodb.com/docs/manual/core/gridfs/

Table 11: Comparison of Data Structures for Local Query Types in NoSQL Stores

Local Query Type	Efficient Data Structure	Efficiency	Reason for Efficiency	Key-Value Stores	Document Stores	Wide-Column Stores	Graph Stores
Point Lookups	Hash Table	High	$O(1)$ key-to-value retrieval	Redis, DynamoDB	–	–	–
	B-Tree (Primary Key)	High	Balanced search trees allow logarithmic lookup	–	MongoDB, Couchbase	HBase, Cassandra (row key)	Neo4j (node index)
	LSM Tree (Memtables)	High	Fast writes + sorted runs support efficient key access	RocksDB, LevelDB	MongoDB WiredTiger	Cassandra, HBase	–
Range Queries	B-Tree	High	Sorted structure supports ordered range scans efficiently	FoundationDB	MongoDB, Couchbase	HBase, Cassandra clustering keys	Neo4j (indexed filters)
	LSM Tree	Medium	Range scans slower due to SSTable merging	RocksDB	MongoDB WiredTiger	Cassandra	–
	Skip List	Medium	Ordered linked list allows sequential scans, but higher pointer overhead	Redis Skiplist	–	–	–
Full-Text Search	Inverted Index	High	Token-to-document mapping accelerates keyword search	RedisSearch	Elasticsearch, MarkLogic	Cassandra+Solr	Neo4j full-text
	Distributed Inverted Index	High	Parallel partitioning scales term lookups across nodes	–	Elasticsearch, CouchDB, ArangoDB	HBase+Lucene	JanusGraph+ES
Aggregations	Columnar Storage	High	Columnar layout reduces I/O for aggregation scans	–	–	Cassandra+Spark, HBase MapReduce	–
	Aggregation Trees	Medium	Pre-aggregated trees accelerate grouped queries but require overhead	–	MongoDB Pipeline, Couchbase N1QL	–	Neo4j APOC
	MapReduce	Medium	Distributes computation across cluster, but higher latency	DynamoDB Streams	Couchbase Views	HBase jobs	JanusGraph OLAP
Spatial Queries	R-Tree	High	Hierarchical partitioning of space enables efficient bounding-box search	RedisGeo	MongoDB 2dsphere	–	Neo4j Spatial

Local Query Type	Data Structure	Efficiency	Reason for Efficiency	Key-Value Stores	Document Stores	Wide-Column Stores	Graph Stores
	KD-Tree	Medium	Works for low dimensions; degrades with high dimensions	–	Elasticsearch, MarkLogic	–	–
	GeoHash Index	High	Preserves locality in keyspace, efficient range bounding	Aerospike Geo	MongoDB Geo, Couchbase GeoJSON	Cassandra extensions	TigerGraph Geo
Time-Series Queries	Append-only Log	High	Sequential writes optimize ingestion speed	RedisTimeSeries	–	Cassandra, HBase WAL	–
	Time-partitioned Tables	High	Partition by time buckets allows efficient chronological queries	–	MongoDB Time-Series	Cassandra, HBase	–
	LSM Trees	High	Sequential SSTables handle temporal writes effectively	RocksDB	MongoDB WiredTiger	Cassandra, HBase	–
Event/Stream Queries	Append-only Log	High	Fast ingestion, durable sequential writes	Redis Streams	–	HBase WAL, Cassandra CDC	–
	Pub/Sub Buffers	High	In-memory event propagation enables low-latency streaming	Redis Pub/Sub	Couchbase Eventing, MongoDB Change Streams	–	Neo4j Streams, Kafka integration
Joins	Secondary Index	Medium	Enables foreign key lookups, but overhead in scaling	–	MongoDB, Couchbase N1QL, ArangoDB	Cassandra limited index	Neo4j Cypher, Gremlin
	Query Planner + MapReduce	Low	Expensive distributed join computation	–	MarkLogic, CouchDB Views	HBase, Spark joins	JanusGraph OLAP
Graph Queries	Adjacency List	High	Constant-time edge lookup per node	–	OrientDB, ArangoDB	–	Neo4j, TigerGraph
	Edge-centric Storage	High	Optimized for traversals, storing edges as first-class citizens	–	OrientDB, ArangoDB	–	JanusGraph, Neo4j
	Native Graph Index	High	Specialized index accelerates pattern matching and traversals	–	ArangoDB, OrientDB	–	TigerGraph, Neo4j native

Table 12: Mapping of global query types to distributed mechanisms, efficiency, and NoSQL store implementations across data models.

Global Query Type	Distributed Mechanism	Efficiency	Reason for Efficiency	Key-Value Stores	Document Stores	Wide-Column Stores	Graph Stores
Point Lookups	Distributed Hash Table (DHT) [38]	High	Keys deterministically mapped to nodes, enabling $O(1)$ access	Dynamo, Riak	–	Cassandra, ScyllaDB	–
	Consistent Hashing	High	Balances load while minimizing re-sharding on cluster change	DynamoDB, Voldemort	Couchbase partitions	Cassandra, ScyllaDB	–
Range Queries	Range Partitioning	High	Maintains key order across shards for efficient scans	–	MongoDB shard ranges	HBase regions, Cassandra clustering	–
	Distributed Skip List	Medium	Allows ordered scans across partitions but with higher latency	–	–	FoundationDB, CockroachDB	–
Full-Text Search	Distributed Inverted Index	High	Parallel term→document lookup across shards	RedisSearch Cluster	Elasticsearch, MarkLogic	Cassandra+Solr, HBase+Lucene	JanusGraph+ES
Aggregations	MapReduce	Medium	Parallel computation across nodes, but higher latency	DynamoDB Streams	CouchDB Views, MongoDB MapReduce	HBase jobs, Cassandra+Spark	JanusGraph OLAP, Neo4j GDS
	Columnar Distribution	High	Parallel scans reduce I/O for analytics workloads	–	–	Cassandra, HBase with SparkSQL	–
Spatial Queries	GeoSharding	High	Preserves spatial proximity in partitioning	Aerospike Geo	MongoDB geoshards, Elastic-Search Spatial	Cassandra+GeoMesa, HBase Spatial	TigerGraph
	Distributed R-Tree	Medium	Distributed bounding-box search across partitions	–	Elasticsearch Spatial	HBase Spatial Index	Neo4j Spatial
Time-Series Queries	Time-based Sharding	High	Efficient chronological scans per shard	–	MongoDB time-series sharding	Cassandra time-buckets, ScyllaDB	–
	Append-Log Replication	High	Sequential ordering across nodes ensures durability	Redis Streams Cluster	Couchbase Eventing	Cassandra CommitLog	–

Global Query Type	Distributed Mechanism	Efficiency	Reason for Efficiency	Key-Value Stores	Document Stores	Wide-Column Stores	Graph Stores
Event/Stream Queries	Pub/Sub Cluster	High	Low-latency propagation of events across nodes	Redis Cluster Pub/Sub	MongoDB Change Streams, Couchbase Eventing	–	Neo4j Streams
	Distributed Log (Kafka-style)	High	Sequential distributed log supports replay and parallel consumers	–	–	Cassandra+Kafka, HBase+Kafka	–
Joins	Global Secondary Index	Medium	Enables cross-shard lookup, but adds overhead	–	Couchbase N1QL, MongoDB Atlas Index	Cassandra Global Indexes, HBase+Hive	ArangoDB multi-shard joins
	Query Federation	Low	Joins simulated across nodes; high network cost	–	MarkLogic Federation	HBase+Hive joins	Gremlin OLAP joins
Graph Queries	Graph Partitioning	Medium	Data locality improves traversal, but cross-partition edges costly	–	ArangoDB, OrientDB	–	Neo4j Fabric, TigerGraph
	Parallel Graph Processing	High	Distributed traversal engines support analytics at scale	–	–	–	JanusGraph+Spark, Neo4j GDS

Table 13: Mapping of database properties to NoSQL characteristics and best store implementations across data models.

Database Property	NoSQL Characteristic	Key-Value	Document	Columnar	Graph
Low Latency Reads/Writes	In-memory storage, caching, B-Trees	Redis (RAM-based, microsecond latency)	MongoDB (WiredTiger B-Tree + cache)	ScyllaDB (C++ reimplementation of Cassandra, ultra-low latency)	Neo4j (optimized for traversals in-memory)
High Throughput Writes	LSM Trees, log-structured storage	Riak (write-heavy workloads)	Couchbase (high ingest + caching)	Cassandra (LSM + distributed writes)	JanusGraph (with Cassandra backend)
Horizontal Scalability	Sharding, partitioning	DynamoDB (auto-sharding, global scale)	MongoDB (sharded clusters)	Cassandra (peer-to-peer architecture)	JanusGraph (scales via Cassandra/HBase backend)
High Availability / Fault Tolerance	Replication, masterless design, quorum protocols	Riak (Dynamo-inspired, fault tolerant)	CouchDB (multi-master replication)	Cassandra (tunable consistency, no single master)	Cosmos DB (Gremlin API) (multi-region, highly available)
Strong Consistency	Consensus protocols, tunable consistency	Etcd / FoundationDB (KV)	MongoDB (transactions since v4.0)	HBase (linearizable reads/writes)	TigerGraph (ACID transactions for graph workloads)
Flexible Schema	Schema-less / schema-on-read	RedisJSON (flexible key schema)	MongoDB (schema-less JSON/BSON)	Bigtable (wide, sparse columns)	Neo4j (property graph with dynamic schema)
Complex Querying	Secondary indexes, inverted indexes	Aerospike (secondary indexes)	MongoDB (rich indexes, queries, aggregations)	Cassandra (SASI indexes)	Neo4j (Cypher query language)

Continued on next page

Table 13 – continued from previous page

Database Property	NoSQL Characteristic	Key-Value	Document	Columnar	Graph
Analytics & Aggregations	Columnar layout, MapReduce integration	Redis Streams (light analytics)	Couchbase (N1QL + MapReduce)	HBase + Hive or Cassandra + Spark	TigerGraph (parallel graph analytics)
Full-Text Search	Inverted indexes, Lucene integration	KeyDB + RediSearch	Couchbase FTS, MongoDB Atlas Search	Cassandra + Elasticsearch	JanusGraph + Elasticsearch
Graph Traversals	Adjacency lists, property graph model	—	ArangoDB (multi-model, doc+graph)	—	Neo4j, TigerGraph, JanusGraph
Global Distribution	Multi-master replication, CRDTs, geo-sharding	DynamoDB Global Tables	MongoDB Atlas Global Clusters	Cassandra (multi-dc replication)	Cosmos DB (multi-region Gremlin API)
Security	Encryption, RBAC, audit logging	DynamoDB (IAM integration)	MongoDB Atlas (RBAC, auditing, encryption)	HBase (Kerberos, fine-grained ACLs)	Neo4j Enterprise (RBAC, LDAP integration)
Ease of Operations	Managed services, automation	DynamoDB (serverless, fully managed)	MongoDB Atlas, Couchbase Capella	Cassandra Astra (managed)	Neo4j Aura, Cosmos DB (Graph API)
Integration with Analytics/ML	Spark/Hadoop connectors, streaming APIs	RedisAI (ML integration)	MongoDB + Spark Connector	Cassandra + Spark / Presto	TigerGraph + Graph ML libraries

5.6 Efficient data stores designs for specific application needs

This section maps common application requirements to NoSQL characteristics and highlights the best-suited NoSQL stores across different data models. Table 13 summarizes how key database properties and application requirements map to NoSQL characteristics and the most suitable store implementations for each data model. For example, low-latency and high-throughput workloads are best served by in-memory key-value stores like Redis or log-structured columnar stores like Cassandra/ScyllaDB. Flexible schema and rich querying are strengths of document stores such as MongoDB and Couchbase. Wide-column stores excel in analytics and scalable ingestion, while graph stores like Neo4j and TigerGraph are optimized for complex relationship traversal and graph analytics. The table also highlights which stores offer strong consistency, global distribution, security, and integration with analytics or machine learning, helping practitioners select the right NoSQL solution based on specific requirements.

5.7 A brief classification of the best NoSQL stores

Best considered NoSQL stores and why for each category:

Key-Value Stores.

- **Redis**: Optimized for in-memory speed (microsecond latency). Better than others for caching, real-time analytics, pub/sub, and now supports JSON and Streams. Its persistence (AOF, RDB) makes it versatile beyond a pure cache.
- **Amazon DynamoDB**: Best for global-scale KV workloads (serverless, auto-scaling, global tables). It inherits Dynamo's design (high availability, partition-tolerance) and surpasses self-managed KV stores in ease of operations.
- **Riak**: Strong in eventual consistency and CRDTs, making it better than others in KV when conflict-free replicated data types are required.

Document Stores.

- **MongoDB**: Best for general-purpose workloads with a rich query language, secondary indexes, and aggregation pipeline. It balances flexibility and performance better than most.
- **Couchbase**: Adds distributed SQL-like querying (N1QL), integrated caching, and multi-dimensional scaling (query, index, data services separated), making it stronger in mixed workloads.
- **CouchDB**: Best for sync and offline-first apps (replication via master-master and PouchDB integration), giving it a unique advantage in mobile and edge scenarios.

Columnar / Wide-Column Stores.

- **Cassandra**: Excels in high write throughput and linear scalability. Its tunable consistency makes it more flexible than HBase, and its decentralized design (no single master) avoids single points of failure.
- **HBase**: Strong in tight Hadoop ecosystem integration (Hive, Spark, MapReduce). Better suited if you want analytics and storage together.
- **ScyllaDB**: A modern reimplementation of Cassandra in C++ with much lower latency and better hardware efficiency than Cassandra.

Graph Stores.

- **Neo4j**: The most mature graph database, with the Cypher query language (widely adopted). Optimized for graph traversals, better for OLTP-style graph queries.
- **JanusGraph**: Best for scalable distributed graphs, built on Cassandra, HBase, or Scylla with Elasticsearch. Stronger in large-scale graph analytics than Neo4j.
- **TigerGraph**: Optimized for deep link analytics (10+ hops traversal). Stronger when graph queries are computationally heavy, such as fraud detection or recommendation workloads.

5.8 The data distribution

The data distribution enhances scaling capabilities. Scaling is significantly impacted by partitioning, replication, consistency criteria, and concurrency control strategies [64]. This subsection explores the data partitioning and replication modes commonly encountered in NoSQL stores.

5.8.1 Data Partitioning

Data partitioning involves splitting data into disjoint partitions and distributing them among different storage nodes. Partitioning helps achieve scalability in read, write, and data storage operations. It balances the request load among multiple nodes, allows parallel request processing, and places data closer to where it is frequently accessed.

There are various partitioning methods, including horizontal partitioning, vertical partitioning, and traversal-oriented partitioning. These methods are explored further below:

Horizontal Partitioning. Horizontal partitioning, also known as sharding, divides data at the row level into partitions. It involves storing sets of rows or records in different segments (or shards), which may be located on different servers. Sharding can be static or dynamic. In dynamic sharding, the partitions are adjusted based on node overload to balance the request and storage load. In static sharding, there are two approaches: range-based sharding and hash-based sharding.

- **Range-based sharding:** Range-based sharding distributes data among nodes by key range. It is effective for range-based queries but can create hot spots and load balancing issues if a particular range of data is more requested than others. This may require coordination from a master node to set node assignments and resolve hot spots by splitting shards causing workload or storage imbalance. It also requires a routing server to maintain range partitions and direct clients to the correct server based on the request, which can create a bottleneck and increase latency. Databases using range-based sharding include Google Bigtable, Apache HBase, MongoDB.
- **Hash-based sharding:** In hash-based sharding, keys are hashed to distribute data among nodes. The process of assigning records to nodes in hash-based sharding can be done using a hash modulo operation with the number of nodes to find the destination node (modulo based sharding). It is also possible to use a circular hash function where each node is identified by a hash value on the ring, and each node is responsible for data whose key hash values are between the node's hash and the next node's hash in ascending order (consistent hashing). It is possible that a circular hash function maps multiple hash values for each node, making each node responsible for multiple virtual nodes identified by the hash values (virtual node hashing).

Hash-based sharding has the advantage of fast calculation of the object's location without the need for a mapping service, unlike range partitioning. However, it negatively impacts range queries because neighboring keys are distributed across different nodes. Among the stores applying it are Amazon DynamoDB, Apache Cassandra, Couchbase, Rya, Redis Cluster,Yahoo PNUTS, IBM Spinnaker.

Vertical Partitioning. Vertical partitioning distributes item columns into partitions so that parts of items are dispatched among different nodes. It is useful when some columns are accessed more frequently than others and can be handled by more nodes. This method is mostly used in wide-column databases, which can partition their items by column families. The column families can then be localized on different nodes. In Bigtable, column families can be grouped into locality groups, each stored in an SSTable, which is the unit of data distribution. Frequently accessed column families can be gathered into a separate locality group.

Traversal-Oriented Partitioning. Traversal-oriented partitioning is used in graph databases to minimize cross-partition traversal. Partitioning can be edge-based or vertex-based. In edge-based partitioning, the edges are balanced among the partitions. Vertices can be replicated across multiple partitions if they are endpoints of edges in different partitions. This method is efficient for traversing relationships and in scenarios involving high-degree vertices (those vertices can have their edges balanced among partitions). However, it has issues in terms of maintaining the consistency of replicated vertices and storage overhead due to vertices replication.
In vertex-based partitioning, the vertices are balanced among the partitions, and an edge is stored with either the source or destination vertex. This method is suitable for vertex-centric queries and situations where vertices are modified more frequently than edges. As vertices are not replicated, they are easier to manage and reduce storage overhead. However, it can lead to communication overhead for edges connecting vertices in different partitions and can result in load imbalance if there are many high-degree vertices.

Challenges. Data partitioning faces challenges such as minimizing multi-partition requests, balancing data processing, request, and storage load, minimizing data transfer during data redistribution, and managing directories that map data to its hosting node.

5.8.2 The data replication

Data replication enables the use of multiple copies of data across different nodes, enhancing read/write scalability and allowing better handling of request loads. It plays a crucial role in scaling read and write requests as they are distributed among multiple nodes. Additionally, it improves system reliability and fault tolerance since data remains accessible even if some servers fail, and it enhances data durability.

Replication of data can be done either synchronously or asynchronously:

- In **synchronous mode**, changes are propagated to all replicas before the write operation is confirmed as successful to the client. This ensures that all copies of the data are updated simultaneously, providing strong consistency. However, synchronous replication can be slower because it requires all nodes to complete the write operation before proceeding.
- In **asynchronous mode**, write operations are confirmed to the client even if some replicas have not yet received the update. This allows for faster write operations and is suitable for long-distance data transfer and unreliable network connections. However, it can result in stale data, leading to consistency issues.

Most NoSQL database systems use asynchronous replication as their primary approach to provide better latency, even at the cost of weaker consistency. The choice of replication model is strongly related to the consistency level defined in the data store.

In term of replication architectures, we distinguish Primary-secondary replication, multi-master (peer-to-peer) replication.

Primary-secondary Replication. In Primary-secondary replication, any node can handle read requests, but only one node is designated as the master to handle write operations. The master node writes data, which is then replicated to the other nodes, known as slave nodes. Primary-secondary replication is suited for environments with heavy read loads since all nodes can process read requests. However, it is less suitable for heavy write loads because write operations can only be processed on the master node, potentially creating a bottleneck. If the master fails, read operations can continue on slave nodes, but global write operations are disabled until the master is restored or the master role is transferred to another node (dynamically or manually). This architecture can also serve as a resilience architecture where read and writes are only performed on the master, with slaves serving as backups. Additionally, slow read operations requiring significant resources can be offloaded to replicas. Data stores such as MongoDB and Redis can use Primary-secondary replication.

Multi-Master (Peer-to-Peer) Replication. In multi-master replication, all nodes storing data replicas have equivalent roles for data write and read operations. They can therefore easily scale not only read requests but also write requests. Protocols such as the Gossip protocol [59] can be used to replicate data efficiently in a multi-master setup. Data stores like Dynamo, Riak, Voldemort, and Cassandra support peer-to-peer replication.

Some replication technologies operate at different layers of the system architecture. Below, we compare three representative replication technologies: HDFS, GridFS, and DHT, which operate at the file system level, database layer, and network layer, respectively.

HDFS. The Hadoop Distributed File System (HDFS) represents a *centralized, file-system-level* approach to replication. It partitions large files into fixed-size blocks (typically 128 MB) and distributes these blocks across multiple nodes. Each block is replicated a configurable number of times (usually three), and the placement of replicas is managed by a single master node, the *NameNode*. This centralized orchestration guarantees fault tolerance and high throughput but introduces potential bottlenecks in metadata management. HDFS is therefore designed for environments prioritizing throughput and durability over strong real-time consistency.

GridFS. GridFS operates at the *database layer* and extends MongoDB to support the storage of large files by fragmenting them into smaller chunks stored as documents. Replication within GridFS is not managed by the file storage logic itself but instead by MongoDB's *replica set* architecture. This design delegates replication and fault tolerance to the database layer, offering a balance between consistency and availability depending on the configured write concern. GridFS thus provides an abstraction for large object storage that integrates with database-level consistency guarantees.

DHT. A Distributed Hash Table (DHT) constitutes a fundamentally *decentralized, algorithmic* approach to replication. It distributes key–value pairs across participating nodes according to a consistent hashing scheme, with no central coordination. Replication is achieved by maintaining redundant copies of keys on multiple neighboring nodes in the identifier space. This model favors scalability and fault tolerance over strict consistency, achieving eventual convergence through autonomous coordination among peers. DHTs form the basis of many peer-to-peer and large-scale distributed systems such as BitTorrent, IPFS, Cassandra, and Dynamo.

Synthesis.. Although all three systems employ replication to enhance reliability and availability, they differ in architectural layering and control philosophy. HDFS reflects a *centrally managed, infrastructure-level* design, GridFS represents a *database-integrated, semi-centralized* model, and DHTs embody a *decentralized, peer-driven* paradigm. Together, these illustrate the continuum of replication strategies across distributed systems, from fully controlled to fully autonomous coordination mechanisms.

Table 14: Compact Comparison of HDFS, GridFS, and DHT Replication Models

System	Layer / Architecture	Replication Control	Consistency Model	Design Philosophy
HDFS	File-system level, centralized	Managed by master node (*NameNode*)	Strong consistency per block, relaxed global consistency	Centralized orchestration and high throughput
GridFS	Database layer (MongoDB)	Managed by database replica sets	Configurable (strong or eventual, depending on write concern)	Integrated storage abstraction with database replication
DHT	Network / peer-to-peer layer	Fully decentralized, algorithmic placement via consistent hashing	Eventual consistency	Autonomous coordination and scalability

5.9 Data persistence and durability

Data in databases can be stored on various types of storage such as volatile memory like RAM and persistent storage like Solid State Drives (SSD) and Hard Disk Drives (HDD). Volatile memory offers very fast I/O speed but loses data on node failure. In contrast, SSDs and HDDs have slower I/O speeds but retain data even in the event of a node failure.

Memory storage is useful for real-time data like IoT data, gaming data, and for ephemeral and dynamic data like session management and e-commerce shopping cart information. On the other hand, hard drive storage is used for more durable data like product information, payment processing, and so on. NoSQL storage systems utilize both types of storage depending on their use case.

NoSQL In-Memory Data Stores. Many key-value stores, such as Memcached and Redis, are used for data caching. This involves storing recently retrieved or computed data from a persistent data store in memory so that future requests for the same data can be served from memory instead of having to retrieve or recompute the data from the persistent store.

NoSQL Persistent Data Stores. All categories of NoSQL stores include data stores that use persistent storage to securely keep data and store data that are too large to fit in memory.

NoSQL Hybrid Data Stores. Some NoSQL stores offer hybrid storage. For instance, wide-column databases use a hybrid storage mode where data updates are initially stored in a sorted memory buffer (referred to as a Memtable in BigTable). Before storing data in the Memtable, it is appended to a commit log for recovery purposes in case of node failure. When the memory buffer reaches a threshold, its content is converted to block storage (called an SSTable in BigTable) and persisted on a storage disk (such as GFS for BigTable). The advantage of this

hybrid mode is that more recent data are stored in memory and can be quickly retrieved when requested. Additionally, the data sorting process can be done faster in memory since wide-column databases store data in sorted order by key. If the required data is not found in memory, the lookup can be performed on the storage disk. Some key-value stores, such as Redis, also have the option to persist data to disk if necessary after a certain period of time.

The Use of Replication to Increase Durability. To enhance durability, ensuring data recovery even in the event of complete node hard disk failure is essential. Replicating data across multiple nodes increases data persistence. Configurations can be set for synchronous or asynchronous replication modes. The more durable mode is synchronous replication, where the node commits the data only when replication is confirmed to have succeeded on a certain number of nodes. In contrast, asynchronous replication increases the possibility of data loss since the node commits the data without waiting for confirmation of successful replication.

5.10 The data consistency

The concept of consistency [78] determines the effect of concurrent operations on the system as perceived by different clients. This consistency is distinct from ACID consistency, which ensures that a transaction brings the database from one valid state to another. In this section, we will describe commonly encountered types of consistency.

Linearizability [26] is the strongest form of consistency in distributed data systems. It ensures that all operations appear to execute atomically and in a single sequence, even though the data is distributed across multiple replicas. This allows a total ordering of all transactions, such that a read operation always returns the most recent write. However, achieving linearizability typically involves complex and costly strategies such as the two-phase commit or Paxos protocols for update commits, or strict two-phase locking for pessimistic concurrency control. These methods can significantly impact availability and performance, especially in wide-area networks. A weaker form of consistency is sequential consistency, described below.

Sequential Consistency [15] implies that operations occur in some global order, consistent with the order of operations on each individual node. Once a process A has observed an operation from process B, it will never observe a state of B prior to the last observed state of B. This condition, combined with the total ordering property, provides a sufficiently strong model for programmers, as it ensures a consistent order of operations across all processes.

Eventual Consistency [42] allows replicas to converge to the same value over time, despite potentially observing different orders of updates at any given moment. Here, the system uses a quorum-based approach, where a read or write quorum refers to the minimum number of replicas that must respond to a read or write request for it to be considered successful. If R is the read quorum, W is the write quorum, and N is the total number of replicas of a record, strong consistency is achieved if $W + R > N$, meaning that the sets of reads and writes overlap sufficiently for at least one read to obtain the latest write version. High quorums can impact latency. Eventual consistency requires conflict detection and resolution mechanisms to reconcile updates, such as client-side automatic reconciliation, timestamp-based reconciliation, and vector clock-based reconciliation. Databases like Amazon Dynamo, Cassandra, Voldemort, and Riak use eventual consistency.

Causal Consistency [13] It ensures that operations which are causally related are seen by all processes in the same order, while concurrent operations (those without a causal relationship) may be seen in different orders by different processes. In practice, if operation A happens before operation B (for example, a user reads a value written by another user and then writes a new value), then every process in the system will observe A before B. However, if two operations are independent and do not influence each other, their order of visibility is not guaranteed. Causal

consistency is weaker than sequential consistency but stronger than eventual consistency, and is useful in collaborative or social applications where the order of related actions matters but strict global ordering is not required.

Per-Object Timeline Consistency [45] maintains a total order of all operations on each record, respecting the order as issued by each process. This model was initially used in Yahoo! PNUTS data store. It is based on the observation that web applications typically perform operations on single records rather than multiple records. Each record has a primary replica, which may differ from one record to another. Serialization of operations on the record is coordinated at the primary replica, where all write operations are directed before confirmation. The updated record can then be propagated from the primary replica to other replicas, even asynchronously.

Parallel Snapshot Isolation (PSI) [55] is a distributed approach to snapshot isolation, which simulates a database snapshot on which transactions operate from start to finish. With snapshot isolation, the system checks before committing a transaction whether the values to be updated have been externally modified; if not, the updates are committed, otherwise the transaction is canceled. PSI, as applied in geo-replicated data stores, relaxes basic snapshot isolation by not enforcing a global ordering of transactions but instead preserving local snapshot isolation within each data center. In the case of cross data center transactions, PSI maintains causal ordering. Snapshot isolation is often combined with Multi-Version Concurrency Control (MVCC) [22] for enhanced concurrency management. Some NoSQL stores, such as Walter, enforce PSI over transactions.

5.11 The concurrency control

The concurrency control in distributed stores can be implemented using two main approaches: pessimistic concurrency control and optimistic concurrency control.

Pessimistic Concurrency Control [9, 21, 22] prevents situations where two or more concurrent users try to update the same record simultaneously. This approach is useful when updates can be delayed until the previous update is complete, making it well-suited for short-interval updates.

Optimistic Concurrency Control [17] Optimistic concurrency control assumes that conflicts are possible but rare. Instead of locking the record, the data store checks at the end of the update process, before committing the operation, to determine whether concurrent users have attempted to modify the same record. If conflicts are detected, various resolution mechanisms can be used, such as canceling the concerned operations or retrying them.

This approach can be combined with MVCC (multi-version concurrency control) [19], which involves creating a new version of the updated data and marking the old version as obsolete. During read operations, the user views the data as it was when the reading began, even if the data was updated or deleted in the meantime by other users. Many NoSQL data stores, such as Voldemort, Riak, HBase, CouchDB, Clustrix, and NuoDB, implement optimistic concurrency control using MVCC.

VI NOSQL STORES CHOICE PER USE CASES

Various use cases are encountered in new-generation applications and services, many of which require properties such as flexibility, high availability, low latency, high throughput, and scalability. Consequently, each use case has specific requirements, necessitating a thorough analysis before selecting the appropriate data store. The table 15 outlines commonly encountered use cases, their characteristic requirements, and examples of NoSQL stores suited for each purpose.

Use Case	Required Characteristics	Key-Value Store	Document Store	Wide-Column Store	Graph Store
Real-Time Applications	Low latency, high throughput, real-time updates	Redis	Couchbase	-	-
Content Management Systems (CMS)	Flexibility, rich querying, schema-less design	-	MongoDB, MarkLogic	-	-
Big Data Applications	Scalability, high-throughput processing, low-latency access	-	-	Google Bigtable, Apache Cassandra	-
Internet of Things (IoT)	Time-series handling, efficient data ingestion, horizontal scalability	-	-	InfluxDB, Amazon Timestream	-
E-Commerce	ACID transactions, flexible schema, high availability	Redis	MongoDB	-	-
Financial Services	ACID compliance, strong consistency, high availability	-	Couchbase	HBase	-
Healthcare	Compliance, data integrity, schema flexibility	-	RavenDB, CouchDB	-	-
Geospatial Applications	Geospatial indexing, complex queries	-	MongoDB	-	-
Personalization and Recommendation Engines	Graph traversal, real-time updates, relationship management	-	-	-	Neo4j, TigerGraph
Search Engines	Full-text search, indexing, high throughput	-	Elasticsearch, Solr	-	-
Mobile Applications	Offline access, real-time sync, flexible schema	-	Couchbase Lite, Firestore	-	-
Graph-Based Use Cases	Graph traversal, efficient queries, dynamic schema	-	-	-	Neo4j, Dgraph
Microservices Architecture	Decentralized management, high availability	etcd	-	-	-
Configuration Management	High availability, fast reads, consistency	etcd	-	-	-
Event and Logging Systems	High write throughput, event-driven architecture	-	-	Apache Cassandra	-
Session Management	Low-latency access, in-memory storage, high availability	Redis, Memcached	-	-	-
Gaming Applications	Low-latency access, real-time data processing	Redis	Couchbase	-	-
Social Networks	Efficient relationship management, real-time updates	-	-	-	Neo4j, JanusGraph
Recommendation Systems	Fast querying, real-time data processing, flexible schema	-	MongoDB	-	Neo4j
Data Warehousing and Analytics	Scalability, high-throughput, complex querying	-	-	Apache HBase	-
Edge Computing	Offline capabilities, real-time sync, lightweight storage	Redis	Couchbase Lite	-	-
Knowledge Graphs and Enterprise Knowledge Management	Semantic querying, graph traversal, data integration	-	-	-	Stardog, AllegroGraph
Customer 360 Views	Data integration, flexible schema, real-time data access	-	MongoDB	-	Neo4j
Blockchain and Ledger Systems	Immutable data storage, transaction support	-	-	-	BigchainDB
Logistics and Supply Chain Management	Real-time tracking, scalable storage, transaction support	-	Couchbase	Apache Cassandra	-

Table 15: Use Cases and corresponding NoSQL data stores

VII THE NOTION OF POLYGLOT PERSISTENCE

Polyglot persistence refers to the idea that different types of data within an application might be best served by data stores with different data models, depending on the data type. This approach encourages the use of multiple databases, each excelling in specific scenarios. It can lead to enhanced efficiency in terms of performance, scalability, and so on. Among polystores, we have ESTOCADA [90], BigDAWG [73], PAM [115], Abstra [99] and [111].
For example, a service might manage user account information with a relational store, store product information in a document store, and manage social network interactions between users with a graph store. Similarly, different microservices might use appropriate data stores for their specific needs.
However, this approach presents challenges such as data synchronization, consistency, and effective integration between the different databases involved.

VIII CHALLENGES IN NOSQL STORES

Despite their clear advantages in scalability and flexibility, NoSQL databases continue to face a range of technical and operational challenges. This section consolidates major limitations discussed in the literature and highlights corresponding research efforts addressing them.

- **Data modeling and schema evolution.** The schema-less nature of many NoSQL systems offers flexibility at the expense of data integrity and maintainability. Over time, applications accumulate heterogeneous document structures and redundant data, complicating queries and migrations. Recent work proposes several mitigation strategies: cost-based denormalization models that optimize performance and storage trade-offs [120], adaptive document migration pipelines for evolving schemas [112], taxonomies of schema-change operations such as aggregation, reference creation, or structural variants [101]. Newer directions include NLP-assisted schema inference that automatically extracts latent schema patterns using contextual embeddings [110], Monte Carlo simulations to predict the performance impact of schema evolution [97], and static code analysis tools that detect schema patterns from application code to recommend refactoring [117].
- **Consistency versus availability.** NoSQL systems typically follow the BASE model (Basically Available, Soft state, Eventually consistent), prioritizing availability and partition tolerance. Developers must handle stale reads, conflict resolution, and reconciliation manually. Comparative studies of concurrency-control mechanisms [108] and CAP–PACELC trade-offs [113] reveal that achieving tunable consistency at scale introduces performance costs. Broader reviews highlight the continued difficulty of achieving low-latency, consistent reads in distributed NoSQL clusters [89].
- **Query and indexing limitations.** While SQL systems provide mature query optimizers, NoSQL query languages remain system-specific and often limited. Complex joins, nested field queries, and spatial or vector data remain performance bottlenecks. Research into advanced index structures, such as localized R-tree variants [103] and adaptive multidimensional indexing [114], improves spatial and analytical workloads. Efforts like code-based schema inference [117] also aim to improve optimizer awareness of implicit data relationships.
- **Scalability versus operational complexity.** Horizontal scaling is a defining feature of NoSQL systems, but operational management of large clusters introduces new complexities. Sharding, replication, and cross-partition queries require fine-tuned strategies to avoid

data hotspots and latency spikes. Systematic reviews like [89] highlight how adaptive partitioning, rebalancing, and self-tuning orchestration frameworks can mitigate these issues. Simulation-based work such as [97] quantifies migration latency and cost under evolving workloads, while others examine model evolution for column-family databases [119].

- **Multi-model convergence.** Recent years have seen the emergence of unified engines supporting document, key–value, graph, and column-family models within one system. Although these multi-model databases enhance flexibility, they often sacrifice optimization for specialized workloads. Studies [100, 114] indicate that physical storage and execution layers must be dynamically tuned for each model type to avoid generalization overheads.
- **Security and compliance.** Early NoSQL databases prioritized scalability and simplicity over robust security. Despite progress, inconsistencies persist in access control granularity, auditability, and encryption defaults. The NIST guidance on NoSQL access control [116] emphasizes the adoption of uniform RBAC/ABAC models and field-level encryption. Complementary reviews [104] identify common misconfigurations and propose best practices for securing production deployments.
- **Cloud-native pressures.** The shift toward managed services such as DynamoDB, MongoDB Atlas, and Cosmos DB simplifies operations but raises concerns over cost visibility and vendor lock-in. Financial and environmental cost modeling frameworks [120] and architectural comparisons between SQL and NoSQL deployments [109] advocate for sustainability-aware design and multi-cloud portability.
- **Integration with analytics and AI.** Bridging transactional NoSQL systems with analytical and AI workloads remains challenging due to differing performance and schema requirements. Traditional ETL pipelines introduce latency and duplication. Research explores columnar and vector-aware NoSQL extensions [118], hybrid OLTP–OLAP designs, and big-data storage tools that combine NoSQL backends with analytical frameworks [96]. These innovations bring analytics closer to operational data but increase system complexity and maintenance overhead.

The following table 16 summarizes the principal challenges observed across the major NoSQL families (key–value, document, column-family and graph). It highlights operational, consistency, querying and research issues to help guide selection and further study.

Challenge Area	**Key–Value Stores** (DynamoDB, Riak, Redis)	**Document Stores** (MongoDB, Couchbase)	**Column-Family Stores** (Cassandra, HBase)	**Graph Stores** (Neo4j, JanusGraph)
Data Modeling & Querying	Very limited queries (mainly GET/PUT), poor secondary indexing	Semi-structured docs, richer queries but limited joins/aggregations	Wide tables, schema evolution painful, query support awkward (CQL-like)	Flexible graph model, but query languages (Gremlin, Cypher) lack standardization
Consistency & Transactions	Single-key atomicity only, multi-key hard	Recently added multi-doc transactions, but costly	Lightweight transactions exist but expensive; tunable consistency confusing	ACID often limited to single-node; distributed transactions very costly
Scalability & Performance	Excellent horizontal scaling, but hotspot risk with skewed keys	Sharding introduces routing complexity, queries slow if poorly indexed	Designed for linear scale, but hotspots & tail latencies are common	Hard to scale traversal queries across distributed clusters
Operational Complexity	Easier to operate, but replication conflict resolution (CRDTs) is complex	Index maintenance, rebalancing shards, schema migrations are heavy	Repairs, rebalancing, and anti-entropy mechanisms are operationally difficult	Distributed graph partitioning and rebalancing are unsolved problems
Interoperability & Portability	Vendor-specific APIs, poor portability	Some SQL-like APIs, but still non-standard	SQL-like CQL helps portability but semantics diverge from SQL	No universal graph query standard (Cypher, Gremlin, GQL emerging but fragmented)
Security & Compliance	Often basic auth, limited fine-grained access control	Better auth, but row-/field-level security limited	Role-based control exists, but auditing/logging weaker than RDBMS	Security often an afterthought, fine-grained control rare
Theoretical & Research Challenges	Conflict resolution semantics still evolving	Balancing expressive queries with distributed performance	Strong consistency at scale without losing availability remains open	Efficient distributed graph processing and HTAP integration still immature

Table 16: Open challenges across NoSQL families

IX CONCLUSION

This survey has presented a concise, up-to-date overview of NoSQL databases, covering their evolution, taxonomy, and core data models—key-value, document, wide-column, graph, multi-model, and cloud-managed. NoSQL systems address the scalability, flexibility, and performance needs of modern applications, with each data model suited to specific use cases such as real-time analytics, content management, and graph-based recommendations.

NoSQL databases have transformed data management by enabling horizontal scaling, handling semi-structured data, and supporting high availability. The choice of data model should be guided by application requirements like query complexity and schema flexibility. Multi-model and cloud-managed solutions further extend NoSQL capabilities, supporting polyglot persistence and simplifying deployment.

Despite their advantages, NoSQL systems face challenges in consistency, query standardization, operational complexity, and security. As demand grows for real-time analytics and machine learning, NoSQL platforms are evolving to support hybrid workloads, though this increases system complexity.

Future developments will focus on stronger consistency, better security, improved analytics integration, and multi-model convergence. NoSQL databases are now essential for scalable, resilient, and adaptable data management, and their ongoing evolution will continue to drive innovation in data-driven applications.

REFERENCES

[1] IBM. *IBM Indexed Sequential Access Method (ISAM)*. Original documentation and technical manuals from IBM detailing ISAM. IBM. 1960.

[2] D. R. Morrison. "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM* 15.4 (Oct. 1968), pages 514–534. ISSN: 0004-5411, 1557-735X.

[3] B. H. Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (July 1970), pages 422–426. ISSN: 0001-0782, 1557-7317.

[4] E. F. Codd. "A relational model of data for large shared data banks". In: *Commun. ACM* 13.6 (June 1, 1970), pages 377–387. ISSN: 0001-0782.

[5] E. E. Rump. "ADABAS: Generalized Database System at Large". In: *IEEE Transactions on Computers* C-20.11 (1971), pages 1380–1382.

[6] D. D. Schmidt and L. S. Rowan. "A Total Approach to Database Management". In: *AFIPS Conference Proceedings* 39 (1971), pages 1019–1030.

[7] IBM. *IBM Virtual Storage Access Method (VSAM)*. Technical documentation on the implementation and use of VSAM by IBM. IBM. 1973.

[8] D. D. Chamberlin and R. F. Boyce. "SEQUEL: A structured English query language". en. In: *Proceedings of the 1976 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control - FIDET '76*. Not Known: ACM Press, 1976, pages 249–264.

[9] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System". In: *Communications of the ACM* 19.11 (1976), pages 624–633.

[10] M. Stonebraker, E. Wong, P. Kreps, and G. Held. "The Design and Implementation of INGRES". In: *ACM Transactions on Database Systems* 1.3 (1976), pages 189–222.

[11] R. W. Taylor and R. L. Frank. "CODASYL Data-Base Management Systems". en. In: *ACM Computing Surveys* 8.1 (Mar. 1976), pages 67–103. ISSN: 0360-0300, 1557-7341.

[12] D. C. Tsichritzis and F. H. Lochovsky. "Hierarchical Data-Base Management: A Survey". en. In: *ACM Computing Surveys* 8.1 (Mar. 1976), pages 105–123. ISSN: 0360-0300, 1557-7341.

[13] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (1978), pages 558–565.

[14] L. Ellison, R. Miner, and E. Oates. "Oracle Database: A Relational Database Management System". In: *Proceedings of the International Conference on Management of Data*. ACM, 1979.

[15] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". In: *IEEE Transactions on Computers* C-28.9 (1979), pages 690–691.

[16] D. D. Chamberlin et al. "A History and Evaluation of System R". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1981, pages 456–476.

[17] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control". In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pages 213–226.

[18] C. Software. "IDMS: Comprehensive System for Large Database Management". In: *Proceedings of the National Computer Conference*. AFIPS Press, 1982, pages 101–110.

[19] P. A. Bernstein and N. Goodman. "Multiversion Concurrency Control—Theory and Algorithms". In: *ACM Transactions on Database Systems (TODS)* 8.4 (1983), pages 465–483.

[20] M. J. Carey, D. J. DeWitt, J. Richardson, and E. Shekita. *Object and File Management in the EXODUS Extensible Database System*. Technical Report. Accepted: 2012-03-15T16:41:43Z. University of Wisconsin-Madison Department of Computer Sciences, 1986.

[21] C. Mohan, B. Lindsay, and R. Obermarck. "Transaction Management in the R* Distributed Database Management System". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1986, pages 103–110.

[22] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5.

[23] D. H. Fishman, D. Beech, H. P. Cate, E. Lyngbæk, J. A. Shoens, and S. Zdonik. "Iris: An Object-Oriented Database Management System". In: *ACM Transactions on Office Information Systems* 5.1 (1987), pages 48–69.

[24] R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. "The Gem–Stone data management system". In: *Object-oriented concepts, databases, and applications*. New York, NY, USA: Association for Computing Machinery, Jan. 3, 1989, pages 283–308. ISBN: 978-0-201-14410-9.

[25] C. Beeri. "A formal approach to object-oriented databases". en. In: *Data & Knowledge Engineering* 5.4 (Oct. 1990), pages 353–382. ISSN: 0169023X.

[26] M. P. Herlihy and J. M. Wing. "Linearizability: a correctness condition for concurrent objects". en. In: *ACM Transactions on Programming Languages and Systems* 12.3 (July 1990), pages 463–492. ISSN: 0164-0925, 1558-4593.

[27] W. Kim. "Object-Oriented Databases: Definition and Research Directions". In: *IEEE Trans. on Knowl. and Data Eng.* 2.3 (Sept. 1, 1990), pages 327–341. ISSN: 1041-4347.

[28] O. Deux. "The O2 System". In: *Communications of the ACM* 34.10 (1991), pages 34–48.

[29] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore Database System". In: *Communications of the ACM* 34.10 (1991), pages 50–63.
[30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.
[31] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (June 1, 1996), pages 351–385. ISSN: 1432-0525.
[32] S. Abiteboul. "Querying semi-structured data". In: (1997). Edited by F. Afrati and P. Kolaitis, pages 1–18.
[33] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible markup language (XML) 1.0*. 1998.
[34] A. Fox and E. A. Brewer. "Harvest, Yield, and Scalable Tolerant Systems". In: Hot Topics in Operating Systems, Workshop on. ISSN: 1530-1621. IEEE Computer Society, Mar. 1, 1999, pages 174–174. ISBN: 978-0-7695-0237-3.
[35] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Stanford InfoLab, Nov. 1999.
[36] E. A. Brewer. "Towards robust distributed systems (abstract)". In: PODC '00 (July 16, 2000), page 7.
[37] L. Lamport. "Paxos Made Simple". en-US. In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (Dec. 2001), pages 51–58.
[38] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A scalable peer-to-peer lookup service for internet applications". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (Aug. 27, 2001), pages 149–160. ISSN: 0146-4833.
[39] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". en. In: *ACM SIGACT News* 33.2 (June 2002), pages 51–59. ISSN: 0163-5700.
[40] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. SOSP '03. New York, NY, USA: Association for Computing Machinery, Oct. 19, 2003, pages 29–43. ISBN: 978-1-58113-757-6.
[41] M. Stonebraker and U. Cetintemel. ""One size fits all": an idea whose time has come and gone". en. In: *21st International Conference on Data Engineering (ICDE'05)*. Tokoyo, Japan: IEEE, 2005, pages 2–11. ISBN: 978-0-7695-2285-2.
[42] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 14, 2007), pages 205–220. ISSN: 0163-5980.
[43] R. Angles and C. Gutierrez. "Survey of graph database models". en. In: *ACM Computing Surveys* 40.1 (Feb. 2008), pages 1–39. ISSN: 0360-0300, 1557-7341.
[44] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". en. In: *ACM Transactions on Computer Systems* 26.2 (June 2008), pages 1–26. ISSN: 0734-2071, 1557-7333.
[45] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proceedings of the VLDB Endowment*. Volume 1. 2. VLDB Endowment, 2008, pages 1277–1288.

[46] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". en. In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pages 1277–1288. ISSN: 2150-8097.

[47] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pages 107–113. ISSN: 0001-0782, 1557-7317.

[48] C. W. Bachman. "The Origin of the Integrated Data Store (IDS): The First Direct-Access DBMS". In: *IEEE Annals of the History of Computing* 31.4 (Oct. 2009), pages 42–54. ISSN: 1934-1547.

[49] J. Pérez, M. Arenas, and C. Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Transactions on Database Systems* 34.3 (Aug. 2009), pages 1–45. ISSN: 0362-5915, 1557-4644.

[50] O. Erling and I. Mikhailov. "Virtuoso: RDF Support in a Native RDBMS". In: *Semantic Web Information Management*. Edited by R. De Virgilio, F. Giunchiglia, and L. Tanca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 501–519. ISBN: 978-3-642-04328-4 978-3-642-04329-1.

[51] R. Cattell. "Scalable SQL and NoSQL data stores". en. In: *ACM SIGMOD Record* 39.4 (May 2011), pages 12–27. ISSN: 0163-5808.

[52] R. Hecht and S. Jablonski. "NoSQL evaluation: A use case oriented survey". In: *2011 International Conference on Cloud and Service Computing*. 2011 International Conference on Cloud and Service Computing (CSC). Hong Kong, China: IEEE, Dec. 2011, pages 336–341. ISBN: 978-1-4577-1637-9 978-1-4577-1635-5 978-1-4577-1636-2.

[53] J. Rao, E. J. Shekita, and S. Tata. "Using Paxos to build a scalable, consistent, and highly available datastore". en. In: *Proceedings of the VLDB Endowment* 4.4 (Jan. 2011), pages 243–254. ISSN: 2150-8097.

[54] S. Sakr, A. Liu, D. M. Batista, and M. Alomari. "A Survey of Large Scale Data Management Approaches in Cloud Environments". In: *IEEE Communications Surveys & Tutorials* 13.3 (2011), pages 311–336. ISSN: 1553-877X.

[55] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional Storage for Geo-Replicated Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, 2011, pages 385–400.

[56] M. Stonebraker. *New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps – Communications of the ACM*. blogcacm. June 16, 2011. URL: https://cacm.acm.org/blogcacm/new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/ (visited on 12/04/2025).

[57] D. Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story". In: *Computer* 45.2 (Feb. 2012), pages 37–42. ISSN: 1558-0814.

[58] R. Angles. "A Comparison of Current Graph Database Models". In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. 2012 IEEE International Conference on Data Engineering Workshops (ICDEW). Arlington, VA, USA: IEEE, Apr. 2012, pages 171–177. ISBN: 978-0-7695-4748-0 978-1-4673-1640-8.

[59] K. Birman, D. Freedman, Q. Huang, and P. Dowell. "Overcoming CAP with Consistent Soft-State Replication". en. In: *Computer* 45.2 (Feb. 2012), pages 50–58. ISSN: 0018-9162.

[60] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd,

S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. "Spanner: Google's Globally-Distributed Database". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, 2012, pages 261–264. ISBN: 978-1-931971-96-6.

[61] R. Escriva, B. Wong, and E. G. Sirer. "HyperDex: a distributed, searchable key-value store". In: SIGCOMM '12 (Aug. 13, 2012), pages 25–36.

[62] C. JMTauro, A. S, and S. A.B. "Comparative Study of the New Generation, Agile, Scalable, High Performance NOSQL Databases". In: *International Journal of Computer Applications* 48.20 (June 30, 2012), pages 1–4. ISSN: 09758887.

[63] S. Lombardo, E. Di Nitto, and D. Ardagna. "Issues in Handling Complex Data Structures with NoSQL Databases". In: *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). Timisoara, Romania: IEEE, Sept. 2012, pages 443–448. ISBN: 978-1-4673-5026-6.

[64] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz. "Data management in cloud environments: NoSQL and NewSQL data stores". In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (Dec. 2013), page 22. ISSN: 2192-113X.

[65] H. Harris. "SQL/DS: IBM's First RDBMS". In: *IEEE Annals of the History of Computing* 35.2 (2013), pages 69–71.

[66] K. Kaur and R. Rani. "Modeling and querying data in NoSQL databases". In: *2013 IEEE International Conference on Big Data*. 2013 IEEE International Conference on Big Data. Silicon Valley, CA, USA: IEEE, Oct. 2013, pages 1–7. ISBN: 978-1-4799-1293-3.

[67] A. Nayak, A. Poriya, and D. Poojary. "Type of NOSQL Databases and its Comparison with Relational Databases". In: *International Journal of Applied Information Systems* 5.4 (Mar. 5, 2013), pages 16–19.

[68] J. Pokorny. "NoSQL databases: a step to database scalability in web environment". In: *International Journal of Web Information Systems* 9.1 (Mar. 29, 2013), pages 69–82. ISSN: 1744-0084.

[69] V. Abramova, J. Bernardino, and P. Furtado. "Experimental Evaluation of NoSQL Databases". In: *International Journal of Database Management Systems* 6.3 (June 30, 2014), pages 01–16. ISSN: 09755985, 09755705.

[70] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 7159. Num Pages: 16. Internet Engineering Task Force, Mar. 2014.

[71] V. N. Gudivada, D. Rao, and V. V. Raghavan. "NoSQL Systems for Big Data Management". In: *2014 IEEE World Congress on Services*. 2014 IEEE World Congress on Services (SERVICES). Anchorage, AK, USA: IEEE, June 2014, pages 190–197. ISBN: 978-1-4799-5069-0 978-1-4799-5068-3.

[72] S. D. Kuznetsov and A. V. Poskonin. "NoSQL data management systems". In: *Programming and Computer Software* 40.6 (Nov. 2014), pages 323–332. ISSN: 0361-7688, 1608-3261.

[73] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. "The BigDAWG Polystore System". In: *ACM SIGMOD Record* 44.2 (Aug. 12, 2015), pages 11–16. ISSN: 0163-5808.

[74] D. Ganesh Chandra. "BASE analysis of NoSQL database". In: *Future Generation Computer Systems* 52 (Nov. 2015), pages 13–21. ISSN: 0167739X.

[75] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser. "Performance Evaluation of NoSQL Databases: A Case Study". In: PABS '15 (Feb. 1, 2015), pages 5–10.

[76] M. A. Rodriguez. "The Gremlin graph traversal machine and language (invited talk)". In: *Proceedings of the 15th Symposium on Database Programming Languages*. DBPL 2015. New York, NY, USA: Association for Computing Machinery, Oct. 27, 2015, pages 1–10. ISBN: 978-1-4503-3902-5.

[77] F. Gessert and N. Ritter. "Scalable data management: NoSQL data stores in research and practice". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 2016 IEEE 32nd International Conference on Data Engineering (ICDE). Helsinki, Finland: IEEE, May 2016, pages 1420–1423. ISBN: 978-1-5090-2020-1.

[78] P. Viotti and M. Vukolić. *Consistency in Non-Transactional Distributed Storage Systems*. June 29, 2016.

[79] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. *Foundations of Modern Query Languages for Graph Databases*. Sept. 26, 2017.

[80] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino. "Persisting big-data: The NoSQL landscape". In: *Information Systems* 63 (Jan. 2017), pages 1–23. ISSN: 03064379.

[81] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter. "NoSQL database systems: a survey and decision guidance". In: *Computer Science - Research and Development* 32.3 (July 2017), pages 353–365. ISSN: 1865-2034, 1865-2042.

[82] A. Gupta, S. Tyagi, N. Panwar, S. Sachdeva, and U. Saxena. "NoSQL databases: Critical analysis and comparison". In: *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*. 2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN). Gurgaon: IEEE, Oct. 2017, pages 293–299. ISBN: 978-1-5386-0627-8.

[83] R. Angles. "The Property Graph Database Model". In: Alberto Mendelzon Workshop on Foundations of Data Management. 2018.

[84] D. Fernandes and J. Bernardino. "Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB:" in: *Proceedings of the 7th International Conference on Data Science, Technology and Applications*. 7th International Conference on Data Science, Technology and Applications. Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2018, pages 373–380. ISBN: 978-989-758-318-6.

[85] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. "Cypher: An Evolving Query Language for Property Graphs". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, May 27, 2018, pages 1433–1445. ISBN: 978-1-4503-4703-7.

[86] A. Bonifati and S. Dumbrava. "Graph Queries: From Theory to Practice". In: *ACM SIGMOD Record* 47.4 (May 17, 2019), pages 5–16. ISSN: 0163-5808.

[87] A. Davoudian, L. Chen, and M. Liu. "A Survey on NoSQL Stores". In: *ACM Computing Surveys* 51.2 (Mar. 31, 2019), pages 1–43. ISSN: 0360-0300, 1557-7341.

[88] M. Diogo, B. Cabral, and J. Bernardino. "Consistency Models of NoSQL Databases". In: *Future Internet* 11.2 (Feb. 14, 2019), page 43. ISSN: 1999-5903.

[89] S. Ramzan, I. S. Bajwa, R. Kazmi, and Amna. "Challenges in NoSQL-Based Distributed Data Storage: A Systematic Literature Review". In: *Electronics* 8.5 (May 2019). Publisher: Multidisciplinary Digital Publishing Institute, page 488. ISSN: 2079-9292.

[90] R. Alotaibi, B. Cautis, A. Deutsch, M. Latrache, I. Manolescu, and Y. Yang. "ESTOCADA: towards scalable polystore systems". In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pages 2949–2952. ISSN: 2150-8097.

[91] P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone. "Data modeling in the NoSQL world". In: *Computer Standards & Interfaces* 67 (Jan. 2020), page 103149. ISSN: 09205489.

[92] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. "TiDB: a Raft-based HTAP database". In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), pages 3072–3084. ISSN: 2150-8097.

[93] D. Huang, X. Ma, and S. Zhang. "Performance Analysis of the Raft Consensus Algorithm for Private Blockchains". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 50.1 (Jan. 2020), pages 172–181. ISSN: 2168-2232.

[94] J. Lu and I. Holubová. "Multi-model Databases: A New Journey to Handle the Variety of Data". en. In: *ACM Computing Surveys* 52.3 (May 2020), pages 1–38. ISSN: 0360-0300, 1557-7341.

[95] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. "CockroachDB: The Resilient Geo-Distributed SQL Database". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS '20: International Conference on Management of Data. Portland OR USA: ACM, June 11, 2020, pages 1493–1509. ISBN: 978-1-4503-6735-6.

[96] A. Faridoon and M. Imran. "Big Data Storage Tools Using NoSQL Databases and Their Applications in Various Domains: A Systematic Review". In: *Computing and Informatics* 40.3 (Nov. 30, 2021), pages 489–521. ISSN: 2585-8807.

[97] A. Hillenbrand, U. Störl, S. Nabiyev, and S. Scherzinger. "MigCast in Monte Carlo: The Impact of Data Model Evolution in NoSQL Databases". In: *ArXiv* (Apr. 23, 2021).

[98] P. Valduriez, R. Jimenez-Peris, and M. T. Özsu. "Distributed Database Systems: The Case for NewSQL". In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLVIII*. Edited by A. Hameurlain and A. M. Tjoa. Volume 12670. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pages 1–15. ISBN: 978-3-662-63518-6 978-3-662-63519-3.

[99] N. Barret, I. Manolescu, and P. Upadhyay. "Abstra: Toward Generic Abstractions for Data of Any Model". In: *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. CIKM '22: The 31st ACM International Conference on Information and Knowledge Management. Atlanta GA USA: ACM, Oct. 17, 2022, pages 4803–4807. ISBN: 978-1-4503-9236-5.

[100] C. J. F. Candel, D. Sevilla Ruiz, and J. J. García-Molina. "A unified metamodel for NoSQL and relational databases". In: *Information Systems* 104 (Feb. 1, 2022), page 101898. ISSN: 0306-4379.

[101] A. H. Chillón, M. Klettke, D. S. Ruiz, and J. G. Molina. *A Taxonomy of Schema Changes for NoSQL Databases*. 2022. arXiv: 2205.11660 [cs.DB].

[102] W. Fan. "Big graphs: challenges and opportunities". In: *Proc. VLDB Endow.* 15.12 (Aug. 1, 2022), pages 3782–3797. ISSN: 2150-8097.

[103] A. Karras, C. Karras, D. Samoladas, K. C. Giotopoulos, and S. Sioutas. "Query Optimization in NoSQL Databases Using an Enhanced Localized R-tree Index". In: *Information Integration and Web Intelligence*. Edited by E. Pardede, P. Delir Haghighi, I. Khalil, and G. Kotsis. Volume 13635. Series Title: Lecture Notes in Computer Science.

Cham: Springer Nature Switzerland, 2022, pages 391–398. ISBN: 978-3-031-21046-4 978-3-031-21047-1.

[104] S. Sicari, A. Rizzardi, and A. Coen-Porisini. "Security&privacy issues and challenges in NoSQL databases". In: *Computer Networks* 206 (Apr. 7, 2022), page 108828. ISSN: 1389-1286.

[105] H. Vera-Olivera, R. Guo, R. C. Huacarpuma, A. P. B. Da Silva, A. M. Mariano, and M. Holanda. "Data Modeling and NoSQL Databases - A Systematic Mapping Review". In: *ACM Computing Surveys* 54.6 (July 31, 2022), pages 1–26. ISSN: 0360-0300, 1557-7341.

[106] R. Belgundi, Y. Kulkarni, and B. Jagdale. "Analysis of Native Multi-model Database Using ArangoDB". In: *Proceedings of Third International Conference on Sustainable Expert Systems*. Edited by S. Shakya, V. E. Balas, and W. Haoxiang. Singapore: Springer Nature, 2023, pages 923–935. ISBN: 978-981-19-7874-6.

[107] M. Besta, R. Gerstenberger, E. Peter, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. *Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries*. Sept. 15, 2023.

[108] S. Kanungo and R. D. Morena. "Concurrency versus consistency in NoSQL databases". In: *Journal of Autonomous Intelligence* 7.3 (Dec. 28, 2023). ISSN: 2630-5046.

[109] W. Khan, T. Kumar, C. Zhang, K. Raj, A. M. Roy, and B. Luo. "SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review". In: *Big Data and Cognitive Computing* 7.2 (June 2023). Publisher: Multidisciplinary Digital Publishing Institute, page 97. ISSN: 2504-2289.

[110] S. Belefqih, A. Zellou, and M. Berquedich. "Semantic Schema Extraction in NoSQL Databases using BERT Embeddings | Data Science Journal". In: (Jan. 12, 2024).

[111] L. El Ahdab, I. Megdiche, A. Peninou, and O. Teste. "Unified Models and Framework for Querying Distributed Data Across Polystores". In: *Research Challenges in Information Science*. Edited by J. Araújo, J. L. De La Vara, M. Y. Santos, and S. Assar. Volume 513. Series Title: Lecture Notes in Business Information Processing. Cham: Springer Nature Switzerland, 2024, pages 3–18. ISBN: 978-3-031-59464-9 978-3-031-59465-6.

[112] S. Fedushko, R. Malyi, Y. Syerov, and P. Serdyuk. "NoSQL document data migration strategy in the context of schema evolution". In: *Data & Knowledge Engineering* 154 (Nov. 2024), page 102369. ISSN: 0169023X.

[113] S. Ferreira, J. Mendonça, B. Nogueira, W. Tiengo, and E. Andrade. "Impacts of data consistency levels in cloud-based NoSQL for data-intensive applications". In: *Journal of Cloud Computing* 13.1 (Nov. 27, 2024), page 158. ISSN: 2192-113X.

[114] C. Gajiwala. "NoSQL Technologies in Big Data Ecosystems: A Comprehensive Review of Architectural Paradigms and Performance Metric". In: *IJFMR - International Journal For Multidisciplinary Research* 6.5 (Oct. 31, 2024). Publisher: IJFMR. ISSN: 2582-2160.

[115] S. He, Y. Yang, L. Tang, C. Dong, and X. Jiang. "A Polystore Approach for Post-processing CAE Data Management". In: *Proceedings of the 5th International Conference on Computer Information and Big Data Applications*. CIBDA 2024: 5th International Conference on Computer Information and Big Data Applications. Wuhan China: ACM, Apr. 26, 2024, pages 1208–1212. ISBN: 979-8-4007-1810-6.

[116] V. C. Hu. *Access control on NoSQL databases*. NIST IR 8504. Gaithersburg, MD: National Institute of Standards and Technology (U.S.), May 7, 2024, NIST IR 8504.

[117] C. J. F. Candel, A. Cleve, and J. G. Molina. "Towards the Automated Extraction and Refactoring of NoSQL Schemas from Application Code". In: *ArXiv* abs/2505.20230 (2025).

[118] M. Mouhiha and A. Mabrouk. "NoSQL data warehouse optimizing models: A comparative study of column-oriented approaches". In: *Big Data Research* 40 (May 2025), page 100523. ISSN: 22145796.

[119] P. Suárez-Otero, M. J. Mior, M. J. Suárez-Cabal, and J. Tuya. "Data migration for column family database evolution". In: *Information and Software Technology* 187 (Nov. 1, 2025), page 107834. ISSN: 0950-5849.

[120] J. Mali, S. Ahvar, F. Atigui, A. Azough, and N. Travers. "DaMoOp: A global approach for optimizing denormalized schemas through a multidimensional cost model". In: *Information Systems* 136 (Feb. 2026), page 102598. ISSN: 03064379.