NoSQL Database Systems: A Survey and Decision Guidance

Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter

Universität Hamburg, Germany
{gessert, wingerath, friedrich, ritter}@informatik.uni-hamburg.de

Abstract. Today, data is generated and consumed at unprecedented scale. This has lead to novel approaches for scalable data management subsumed under the term "NoSQL" database systems to handle the ever-increasing data volume and request loads. However, the heterogeneity and diversity of the numerous existing systems impede the well-informed selection of a data store appropriate for a given application context. Therefore, this article gives a top-down overview of the field: Instead of contrasting the implementation specifics of individual representatives, we propose a comparative classification model that relates functional and non-functional requirements to techniques and algorithms employed in NoSQL databases. This NoSQL Toolbox allows us to derive a simple decision tree to help practitioners and researchers filter potential system candidates based on central application requirements.

1 Introduction

Traditional relational database management systems (RDBMSs) provide powerful mechanisms to store and query structured data under strong consistency and transaction guarantees and have reached an unmatched level of reliability, stability and support through decades of development. In recent years, however, the amount of useful data in some application areas has become so vast that it cannot be stored or processed by traditional database solutions. User-generated content in social networks or data retrieved from large sensor networks are only two examples of this phenomenon commonly referred to as **Big Data** [35]. A class of novel data storage systems able to cope with Big Data are subsumed under the term **NoSQL databases**, many of which offer horizontal scalability and higher availability than relational databases by sacrificing querying capabilities and consistency guarantees. These trade-offs are pivotal for service-oriented computing and as-a-service models, since any stateful service can only be as scalable and fault-tolerant as its underlying data store.

There are dozens of NoSQL database systems and it is hard to keep track of where they excel, where they fail or even where they differ, as implementation details change quickly and feature sets evolve over time. In this article, we therefore aim to provide an overview of the NoSQL landscape by discussing employed concepts rather than system specificities and explore the requirements typically posed to NoSQL database systems, the techniques used to fulfil these requirements and the trade-offs that have to be made in the process. Our focus lies on key-value, document and wide-column stores, since these NoSQL categories

cover the most relevant techniques and design decisions in the space of scalable data management.

In Section 2, we describe the most common high-level approaches towards categorizing NoSQL database systems either by their data model into key-value stores, document stores and wide-column stores or by the safety-liveness trade-offs in their design (CAP and PACELC). We then survey commonly used techniques in more detail and discuss our model of how requirements and techniques are related in Section 3, before we give a broad overview of prominent database systems by applying our model to them in Section 4. A simple and abstract decision model for restricting the choice of appropriate NoSQL systems based on application requirements concludes the paper in Section 5.

2 High-Level System Classification

In order to abstract from implementation details of individual NoSQL systems, high-level classification criteria can be used to group similar data stores into categories. In this section, we introduce the two most prominent approaches: data models and CAP theorem classes.

2.1 Different Data Models

The most commonly employed distinction between NoSQL databases is the way they store and allow access to data.

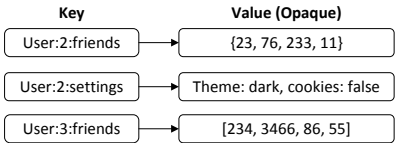

Fig. 1: Key-value stores offer efficient storage and retrieval of arbitrary values.

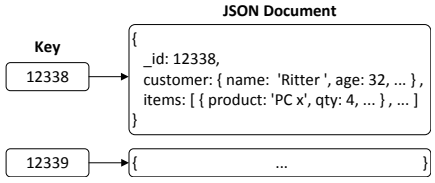

Fig. 2: Document stores are aware of the internal structure of the stored entity and thus can support queries.

Key-Value Stores. A key-value store consists of a set of key-value pairs with unique keys. Due to this simple structure, it only supports get and put operations. As the nature of the stored value is transparent to the database, pure key-value stores do not support operations beyond simple CRUD (Create, Read, Update, Delete). Key-value stores are therefore often referred to as **schemaless** [44]: Any assumptions about the structure of stored data are implicitly encoded in the application logic (*schema-on-read* [31]) and not explicitly defined through a data definition language (*schema-on-write*).

The obvious advantages of this data model lie in its simplicity. The very simple abstraction makes it easy to partition and query the data, so that the database system can achieve low latency as well as high throughput. However, if an application demands more complex operations, e.g. range queries, this data model is not powerful enough. Figure 1 illustrates how user account data and settings might be stored in a key-value store. Since queries more complex than simple lookups are not supported, data has to be analyzed inefficiently in application code to extract information like whether cookies are supported or not (`cookies: false`).

Document Stores. A document store is a key-value store that restricts values to semi-structured formats such as JSON[1] documents. This restriction in comparison to key-value stores brings great flexibility in accessing the data. It is not only possible to fetch an entire document by its ID, but also to retrieve only parts of a document, e.g. the age of a customer, and to execute queries like aggregation, query-by-example or even full-text search.

Wide-Column Stores inherit their name from the image that is often used to explain the underlying data model: a relational table with many sparse columns. Technically, however, a wide-column store is closer to a distributed multi-level[2] sorted map: The first-level keys identify rows which themselves consist of key-value pairs. The first-level keys are called **row keys**, the second-level keys are called **column keys**. This storage scheme makes tables with arbitrarily many columns feasible, because there is no column key without a corresponding value. Hence, null values can be stored without any space overhead. The set of all columns is partitioned into so-called **column families** to colocate columns on disk that are usually accessed together. On disk, wide-column stores do not colocate all data from each row, but instead values of the same column family *and* from the same row. Hence, an entity (a row) cannot be retrieved by one single lookup as in a document store, but has to be joined together from the columns of all column families. However, this storage layout usually enables highly efficient data compression and makes retrieving only a portion of an entity very efficient. The data are stored in lexicographic order of their keys, so that data that are accessed together are physically co-located, given a careful key design. As all rows are distributed into contiguous ranges (so-called **tablets**) among different **tablet servers**, row scans only involve few servers and thus are very efficient.

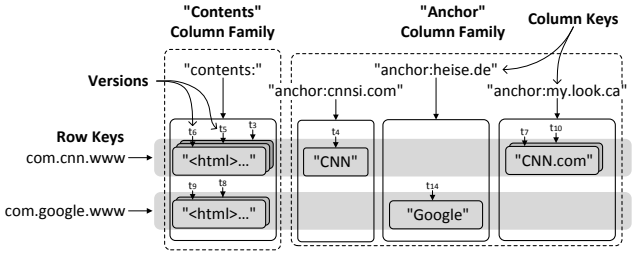

Fig. 3: Data in a wide-column store.

Bigtable [9], which pioneered the wide-column model, was specifically developed to store a large collection of webpages as illustrated in Figure 3. Every row in the webpages table corresponds to a single webpage. The row key is a concatenation of the URL components in reversed order and every column key is composed of the column family name and a column qualifier, separated by a colon. There are two column families: the "contents" column family with only one column holding the actual webpage and the "anchor" column family holding

[1] The JavaScript Object Notation is a standard format consisting of nested attribute-value pairs and lists.

[2] In some systems (e.g. Bigtable and HBase), multi-versioning is implemented by adding a timestamp as third-level key.

links to each webpage, each in a separate column. Every cell in the table (i.e. every value accessible by the combination of row and column key) can be versioned by timestamps or version numbers. It is important to note that much of the information of an entity lies in the keys and not only in the values [9].

2.2 Consistency-Availability Trade-Offs: CAP and PACELC

Another defining property of a database apart from how the data are stored and how they can be accessed is the level of consistency that is provided. Some databases are built to guarantee strong consistency and serializability (ACID[3]), while others favour availability (BASE[4]). This trade-off is inherent to every distributed database system and the huge number of different NoSQL systems shows that there is a wide spectrum between the two paradigms. In the following, we explain the two theorems CAP and PACELC according to which database systems can be categorised by their respective positions in this spectrum.

CAP. Like the famous FLP Theorem [19], the CAP Theorem, presented by Eric Brewer at PODC 2000 [7] and later proven by Gilbert and Lynch [21], is one of the truly influential impossibility results in the field of distributed computing, because it places an ultimate upper bound on what can possibly be accomplished by a distributed system. It states that a sequentially consistent read/write register[5] that eventually responds to every request cannot be realised in an asynchronous system that is prone to network partitions. In other words, it can guarantee at most two of the following three properties at the same time:

- **Consistency (C)**: Reads and writes are always executed atomically and are strictly consistent (linearizable [26]). Put differently, all clients have the same view on the data at all times.
- **Availability (A)**: Every non-failing node in the system can always accept read and write requests by clients and will eventually return with a meaningful response, i.e. not with an error message.
- **Partition-tolerance (P)**: The system upholds the previously displayed consistency guarantees and availability in the presence of message loss between the nodes or partial system failure.

Brewer argues that a system can be both available and consistent in normal operation, but in the presence of a system partition, this is not possible: If the system continues to work in spite of the partition, there is some non-failing node that has lost contact to the other nodes and thus has to decide to either continue processing client requests to preserve availability (AP, **eventual consistent systems**) or to reject client requests in order to uphold consistency guarantees (CP). The first option violates consistency, because it might lead to stale reads and conflicting writes, while the second option obviously sacrifices availability. There are also systems that usually are available and consistent, but fail completely when there is a partition (CA), for example single-node systems. It has been shown that the CAP-theorem holds for any consistency property

[3] ACID [23]: **A**tomicity, **C**onsistency, **I**solation, **D**uration

[4] BASE [42]: **B**asically **A**vailable, **S**oft-state, **E**ventually consistent

[5] A read/write register is a data structure with only two operations: setting a specific value (`set`) and returning the latest value that was set (`get`).

that is at least as strong as causal consistency, which also includes any recency bounds on the permissible staleness of data (Δ-atomicity) [37]. Serializability as the correctness criterion of transactional isolation does not require strong consistency. However, similar to consistency, serializability can also not be achieved under network partitions [15].

The classification of NoSQL systems as either AP, CP or CA vaguely reflects the individual systems' capabilities and hence is widely accepted as a means for high-level comparisons. However, it is important to note that the CAP Theorem actually does not state anything on normal operation; it merely tells us whether a system favors availability or consistency *in the face of a network partition*. In contrast to the FLP-Theorem, the CAP theorem assumes a failure model that allows arbitrary messages to be dropped, reordered or delayed indefinitely. Under the weaker assumption of reliable communication channels (i.e. messages always arrive but asynchronously and possibly reordered) a CAP-system is in fact possible using the Attiya, Bar-Noy, Dolev algorithm [2], as long as a majority of nodes are up[6].

PACELC. This lack of the CAP Theorem is addressed in an article by Daniel Abadi [1] in which he points out that the CAP Theorem fails to capture the trade-off between latency and consistency during *normal* operation, even though it has proven to be much more influential on the design of distributed systems than the availability-consistency trade-off in failure scenarios. He formulates PACELC which unifies both trade-offs and thus portrays the design space of distributed systems more accurately. From PACELC, we learn that in case of a **P**artition, there is an **A**vailability-**C**onsistency trade-off; **E**lse, i.e. in normal operation, there is a **L**atency-**C**onsistency trade-off.

This classification basically offers two possible choices for the partition scenario (A/C) and also two for normal operation (L/C) and thus appears more fine-grained than the CAP classification. However, many systems cannot be assigned exclusively to one single PACELC class and one of the four PACELC classes, namely PC/EL, can hardly be assigned to any system.

3 Techniques

Every significantly successful database is designed for a particular class of applications, or to achieve a specific combination of desirable system properties. The simple reason why there are so many different database systems is that it is not possible for any system to achieve all desirable properties at once. Traditional SQL databases such as PostgreSQL have been built to provide the full functional package: a very flexible data model, sophisticated querying capabilities including joins, global integrity constraints and transactional guarantees. On the other end of the design spectrum, there are key-value stores like Dynamo that scale with data and request volume and offer high read and write throughput as well as low latency, but barely any functionality apart from simple lookups.

[6] Therefore, consensus as used for coordination in many NoSQL systems either natively [4] or through coordination services like Chubby and Zookeeper [28] is even harder to achieve with high availability than strong consistency [19].

In this section, we highlight the design space of distributed database systems, concentrating on sharding, replication, storage management and query processing. We survey the available techniques and discuss how they are related to different functional and non-functional properties (goals) of data management systems. In order to illustrate what techniques are suitable to achieve which system properties, we provide the **NoSQL Toolbox** (Figure 4) where each technique is connected to the functional and non-functional properties it enables (positive edges only).

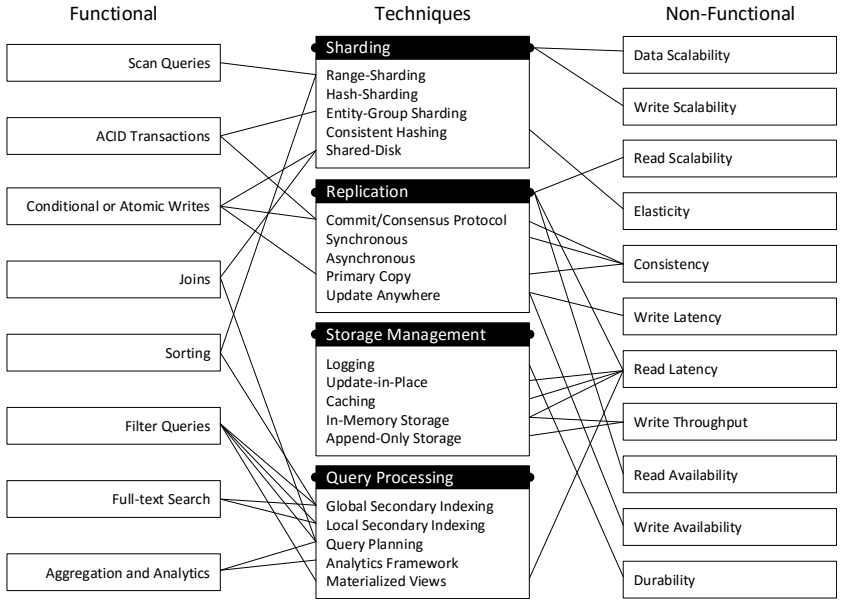

Fig. 4: The NoSQL Toolbox: It connects the techniques of NoSQL databases with the desired functional and non-functional system properties they support.

3.1 Sharding

Several distributed relational database systems such as Oracle RAC or IBM DB2 pureScale rely on a **shared-disk architecture** where all database nodes access the same central data repository (e.g. a NAS or SAN). Thus, these systems provide consistent data at all times, but are also inherently difficult to scale. In contrast, the (NoSQL) database systems focused in this paper are built upon a **shared-nothing architecture**, meaning each system consists of many servers with private memory and private disks that are connected through a network. Thus, high scalability in throughput and data volume is achieved by **sharding** (partitioning) data across different nodes (**shards**) in the system.

There are three basic distribution techniques: range-sharding, hash-sharding and entity-group sharding. To make efficient scans possible, the data can be partitioned into ordered and contiguous value ranges by **range-sharding**. However, this approach requires some coordination through a master that manages assignments. To ensure elasticity, the system has to be able to detect and resolve hotspots automatically by further splitting an overburdened shard.

Range sharding is supported by wide-column stores like BigTable, HBase or Hypertable [49] and document stores, e.g. MongoDB, RethinkDB, Espresso [43] and DocumentDB [46]. Another way to partition data over several machines is **hash-sharding** where every data item is assigned to a shard server according to some hash value built from the primary key. This approach does not require a coordinator and also guarantees the data to be evenly distributed across the shards, as long as the used hash function produces an even distribution. The obvious disadvantage, though, is that it only allows lookups and makes scans unfeasible. Hash sharding is used in key-value stores and is also available in some wide-coloumn stores like Cassandra [34] or Azure Tables [8].

The shard server that is responsible for a record can for example be determined as $server_{id} = hash(id)\ mod\ servers$. However, this hashing scheme requires all records to be reassigned every time a new server joins or leaves, because it changes with the number of shard servers (*servers*), so that it is actually not used in elastic systems like Dynamo, Riak or Cassandra, which allow additional resources to be added on-demand and again be removed when dispensable. Instead, elastic systems use **consistent hashing** [30] where only a fraction of the data have to be reassigned upon such system changes.

Entity-group sharding is a data partitioning scheme with the goal of enabling single-partition transactions on co-located data. The partitions are called entity-groups and either explicitly declared by the application (e.g. in G-Store [14] and MegaStore [4]) or derived from transactions' access patterns (e.g. in Relational Cloud [13] and Cloud SQL Server [5]). If a transaction accesses data that spans more than one group, data ownership can be transferred between entity-groups or the transaction manager has to fallback to more expensive multi-node transaction protocols.

3.2 Replication

In terms of CAP, conventional RDBMSs are often CA systems run in single-server mode: The entire system becomes unavailable on machine failure. And so system operators secure data integrity and availability through expensive, but reliable high-end hardware. In contrast, NoSQL systems like Dynamo, BigTable or Cassandra are designed for data and request volumes that cannot possibly be handled by one single machine, and therefore they run on clusters consisting of thousands of servers[7]. Since failures are inevitable and will occur frequently in any large-scale distributed system, the software has to cope with them on a daily basis [24]. In 2009, Google fellow Jeff Dean stated [16] that a typical new cluster at Google encounters thousands of hard drive failures, 1,000 single-machine failures, 20 rack failures and several network partitions due to expected and unexpected circumstances in its first year alone. Many more recent cases of network partitions and outages in large cloud data centers have been reported [3]. Replication allows the system to maintain availability and durability in the face of such errors. But storing the same records on different machines (**replica servers**) in the cluster introduces the problem of synchronization between them

[7] Low-end hardware is used, because it is substantially more cost-efficient than high-end hardware [27, Section 3.1].

and thus a trade-off between consistency on the one hand and latency and availability on the other.

Gray et al. [22] propose a two-tier classification of different replication strategies according to *when* updates are propagated to replicas and *where* updates are accepted. There are two possible choices on tier one ("when"): **Eager** (*synchronous*) replication propagates incoming changes synchronously to all replicas before a commit can be returned to the client, whereas **lazy** (*asynchronous*) replication applies changes only at the receiving replica and passes them on asynchronously. The great advantage of *eager* replication is consistency among replicas, but it comes at the cost of higher write latency due to the need to wait for other replicas and impaired availability [22]. *Lazy* replication is faster, because it allows replicas to diverge; as a consequence, stale data might be served. On the second tier ("where"), again, two different approaches are possible: Either a **master-slave** (*primary copy*) scheme is pursued where changes can only be accepted by one replica (the master) or, in a **update anywhere** (*multi-master*) approach, every replica can accept writes. In *master-slave* protocols, concurrency control is not more complex than in a distributed system without replicas, but the entire replica set becomes unavailable, as soon as the master fails. Multi-master protocols require complex mechanisms for prevention or detection and reconciliation of conflicting changes. Techniques typically used for these purposes are versioning, vector clocks, gossiping and read repair (e.g. in Dynamo [18]) and convergent or commutative datatypes [45] (e.g. in Riak).

Basically, all four combinations of the two-tier classification are possible. Distributed relational systems usually perform *eager master-slave* replication to maintain strong consistency. *Eager update anywhere* replication as for example featured in Google's Megastore suffers from a heavy communication overhead generated by synchronisation and can cause distributed deadlocks which are expensive to detect. NoSQL database systems typically rely on *lazy* replication, either in combination with the master-slave (CP systems, e.g. HBase and MongoDB) or the update anywhere approach (AP systems, e.g. Dynamo and Cassandra). Many NoSQL systems leave the choice between latency and consistency to the client, i.e. for every request, the client decides whether to wait for a response from any replica to achieve minimal latency or for a certainly consistent response (by a majority of the replicas or the master) to prevent stale data.

An aspect of replication that is not covered by the two-tier scheme is the distance between replicas. The obvious advantage of placing replicas near one another is low latency, but close proximity of replicas might also reduce the positive effects on availability; for example, if two replicas of the the same data item are placed in the same rack, the data item is not available on rack failure in spite of replication. An alternative technique for latency reduction is used in Orestes [20], where data is cached close to applications using web caching infrastructure and cache coherence protocols.

Geo-replication can protect the system against complete data loss and improve read latency for distributed access from clients. *Eager* geo-replication, as implemented in Google's Megastore [4], Spanner [12], MDCC [32] and Mencius

[38] achieve strong consistency at the cost of higher write latencies (typically 100ms [12] to 600ms [4]). With *lazy* geo-replication as in Dynamo [18], PNUTS [11], Walter [47], COPS [36], Cassandra [34] and BigTable [9] recent changes may be lost, but the system performs better and remains available during partitions. Charron-Bost et al. [10, Chapter 12] and Öszu and Valduriez [41, Chapter 13] provide a comprehensive discussion of database replication.

3.3 Storage Management

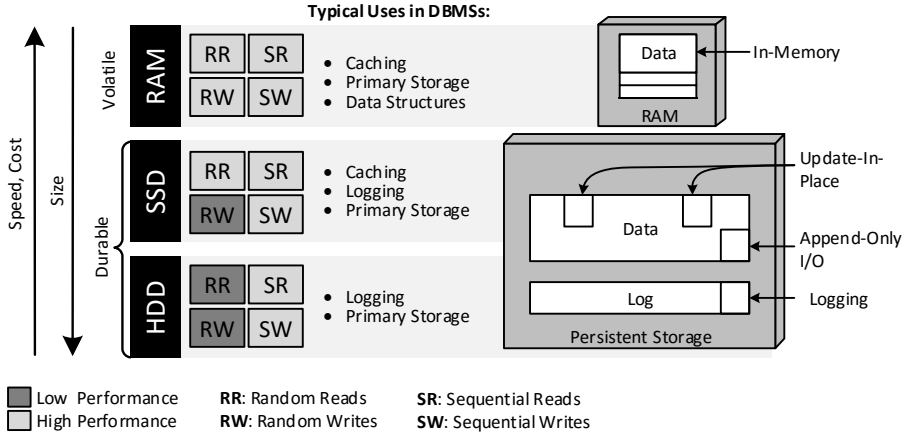

Fig. 5: The storage pyramid and its role in NoSQL systems.

For best performance, database systems need to be optimized for the storage media they employ to serve and persist data. These are typically main memory (RAM), solid-state drives (SSDs) and spinning disk drives (HDDs) that can be used in any combination. Unlike RDBMSs in enterprise setups, distributed NoSQL databases avoid specialized shared-disk architectures in favor of shared-nothing clusters based on commodity servers (employing commodity storage media). Storage devices are typically visualized as a "storage pyramid" (see Figure 5) [25]. There is also a set of transparent caches (e.g. L1-L3 CPU caches and disk buffers, not shown in the Figure), that are only implicitly leveraged through well-engineered database algorithms that promote data locality. The very different cost and performance characteristics of RAM, SSD and HDD storage and the different strategies to leverage their strengths (storage management) are one reason for the diversity of NoSQL databases. Storage management has a spatial dimension (where to store data) and a temporal dimension (when to store data). Update-in-place and append-only-IO are two complementary spatial techniques of organizing data; in-memory prescribes RAM as the location of data, whereas logging is a temporal technique that decouples main memory and persistent storage and thus provides control over when data is actually persisted.

In their seminal paper "the end of an architectural era" [48], Stonebraker et al. have found that in typical RDBMSs, only 6.8% of the execution time is spent on "useful work", while the rest is spent on:

- buffer management (34.6%), i.e. caching to mitigate slower disk access

- latching (14.2%), to protect shared data structures from race conditions caused by multi-threading
- locking (16.3%), to guarantee logical isolation of transactions
- logging (11.9%), to ensure durability in the face of failures
- hand-coded optimizations (16.2%)

This motivates that large performance improvements can be expected if RAM is used as primary storage (**in-memory** databases [50]). The downside are high storage costs and lack of durability – a small power outage can destroy the database state. This can be solved in two ways: The state can be replicated over n in-memory server nodes protecting against $n-1$ single-node failures (e.g. HStore, VoltDB [29]) or by **logging** to durable storage (e.g. Redis or SAP Hana). Through logging, a random write access pattern can be transformed to a sequential one comprised of received operations and their associated properties (e.g. redo information). In most NoSQL systems, the commit rule for logging is respected, which demands every write operation that is confirmed as successful to be logged and the log to be flushed to persistent storage. In order to avoid the rotational latency of HDDs incurred by logging each operation individually, log flushes can be batched together (group commit) which slightly increases the latency of individual writes, but drastically improves throughput.

SSDs and more generally all storage devices based on NAND flash memory differ substantially from HDDs in various aspects: "(1) asymmetric speed of read and write operations, (2) no in-place overwrite – the whole block must be erased before overwriting any page in that block, and (3) limited program/erase cycles" [40]. Thus, a database system's storage management must not treat SSDs and HDDs as slightly slower, persistent RAM, since random writes to an SSD are roughly an order of magnitude slower than sequential writes. Random reads, on the other hand, can be performed without any performance penalties. There are some database systems (e.g. Oracle Exadata, Aerospike) that are explicitly engineered for these performance characteristics of SSDs. In HDDs, both random reads and writes are 10-100 times slower than sequential access. Logging hence suits the strengths of SSDs and HDDs which both offer a significantly higher throughput for sequential writes.

For in-memory databases, an **update-in-place** access pattern is ideal: It simplifies the implementation and random writes to RAM are essentially equally fast as sequential ones, with small differences being hidden by pipelining and the CPU-cache hierarchy. However, RDBMSs and many NoSQL systems (e.g. MongoDB) employ an update-in-place update pattern for persistent storage, too. To mitigate the slow random access to persistent storage, main memory is usually used as a cache and complemented by logging to guarantee durability. In RDBMSs, this is achieved through a complex buffer pool which not only employs cache-replace algorithms appropriate for typical SQL-based access patterns, but also ensures ACID semantics. NoSQL databases have simpler buffer pools that profit from simpler queries and the lack of ACID transactions. The alternative to the buffer pool model is to leave caching to the OS through virtual memory (e.g. employed in MongoDB's MMAP storage engine). This simplifies the database

architecture, but has the downside of giving less control over which data items or pages reside in memory and when they get evicted. Also read-ahead (speculative reads) and write-behind (write buffering) transparently performed with OS buffering lack sophistication as they are based on file system logics instead of database queries.

Append-only storage (also referred to as log-structuring) tries to maximize throughput by writing sequentially. Although log-structured file systems have a long research history, append-only I/O has only recently been popularized for databases by BigTable's use of Log-Structured Merge (LSM) trees [9] consisting of an in-memory cache, a persistent log and immutable, periodically written storage files. LSM trees and variants like Sorted Array Merge Trees (SAMT) and Cache-Oblivious Look-ahead Arrays (COLA) have been applied in many NoSQL systems (Cassandra, CouchDB, LevelDB, RethinkDB, RocksDB, InfluxDB, TokuDB) [31]. Designing a database to achieve maximum write performance by always writing to a log is rather simple, the difficulty lies in providing fast random and sequential reads. This requires an appropriate index structure that is either permanently updated as a copy-on-write (COW) data structure (e.g. CouchDB's COW B-trees) or only periodically persisted as an immutable data structure (e.g. in BigTable-style systems). An issue of all log-structured storage approaches is costly garbage collection (compaction) to reclaim space of updated or deleted items.

3.4 Query Processing

The querying capabilities of a NoSQL database mainly follow from its distribution model, consistency guarantees and data model. **Primary key lookup**, i.e. retrieving data items by a unique ID, is supported by every NoSQL system, since it is compatible to range- as well as hash-partitioning. **Filter queries** return all items (or projections) that meet a predicate specified over the properties of data items from a single table. In their simplest form, they can be performed as *filtered full-table scans*. For hash-partitioned databases this implies a *scatter-gather* pattern where each partition performs the predicated scan and results are merged. For range-partitioned systems, any conditions on the range attribute can be exploited to select partitions.

To circumvent the inefficiencies of $O(n)$ scans, secondary indexes can be employed. These can either be **local secondary indexes** that are managed in each partition or **global secondary indexes** that index data over all partitions [4]. As the global index itself has to be distributed over partitions, consistent secondary index maintenance would necessitate slow and potentially unavailable commit protocols. Therefore in practice, most systems only offer eventual consistency for these indexes (e.g. Megastore, Google AppEngine Datastore, DynamoDB) or do not support them at all (e.g. HBase, Azure Tables). When executing global queries over local secondary indexes the query can only be targeted to a subset of partitions if the query predicate and the partitioning rules intersect. Otherwise, results have to be assembled through scatter-gather. For example, a user table with range-partitioning over an age field can service queries that have an equality condition on age from one partition whereas queries over names need

to be evaluated at each partition. A special case of global secondary indexing is full-text search, where selected fields or complete data items are fed into either a database-internal inverted index (e.g. MongoDB) or to an external search platform such as ElasticSearch or Solr (Riak Search, DataStax Cassandra).

Query planning is the task of optimizing a query plan to minimize execution costs [25]. For aggregations and joins, query planning is essential as these queries are very inefficient and hard to implement in application code. The wealth of literature and results on relational query processing is largely disregarded in current NoSQL systems for two reasons. First, the key-value and wide-column model are centered around CRUD and scan operations on primary keys which leave little room for query optimization. Second, most work on distributed query processing focuses on OLAP workloads that favor throughput over latency whereas single-node query optimization is not easily applicable for partitioned and replicated databases. However, it remains an open research challenge to generalize the large body of applicable query optimization techniques especially in the context of document databases[8].

In-database analytics can be performed either natively (e.g. in MongoDB, Riak, CouchDB) or through external analytics platforms such as Hadoop, Spark and Flink (e.g. in Cassandra and HBase). The prevalent native batch analytics abstraction exposed by NoSQL systems is MapReduce [17]. Due to I/O, communication overhead and limited execution plan optimization, these batch- and micro-batch-oriented approaches have high response times. **Materialized views** are an alternative with lower query response times. They are declared at design time and continuously updated on change operations (e.g. in CouchDB and Cassandra). However, similar to global secondary indexing, view consistency is usually relaxed in favor of fast, highly-available writes, when the system is distributed. As only few database systems come with built-in support for ingesting and querying unbounded streams of data, **near-real-time analytics** pipelines commonly implement either the **Lambda Architecture** [39] or the **Kappa Architecture** [33]: The former complements a batch processing framework like Hadoop MapReduce with a stream processor such as Storm [6] and the latter exclusively relies on stream processing and forgoes batch processing altogether.

4 System Case Studies

In this section, we provide a qualitative comparison of some of the most prominent key-value, document and wide-column stores. We present the results in strongly condensed comparisons and refer to the documentations of the individual systems for in-detail information. The proposed NoSQL Toolbox (see Figure 4, p. 6) is a means of abstraction that can be used to classify database systems along three dimensions: functional requirements, non-functional requirements and the techniques used to implement them. We argue that this classification characterizes many database systems well and thus can be used to meaningfully contrast different database systems: Table 1 shows a direct comparison

[8] Currently only RethinkDB can perform general θ-joins. MongoDB's aggregation framework has support for left-outer equi-joins in its aggregation framework and CouchDB allows joins for pre-declared map-reduce views.

of MongoDB, Redis, HBase, Riak, Cassandra and MySQL in their respective default configurations. A more verbose comparison of central system properties is presented in Table 2 (see p. 15).

	Funct. Req.								Non-Funct. Req.										Techniques																					
	Scan Queries	ACID Transactions	Conditional Writes	Joins	Sorting	Filter Queries	Full-Text Search	Analytics	Data Scalability	Write Scalability	Read Scalability	Elasticity	Consistency	Write Latency	Read Latency	Write Throughput	Read Availability	Write Availability	Durability	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared-Disk	Transaction Protocol	Sync. Replication	Async. Replication	Primary Copy	Update Anywhere	Logging	Update-in-Place	Caching	In-Memory Storage	Append-Only Storage	Global Indexing	Local Indexing	Query Planning	Analytics Framework	Materialized Views
MongoDB	x		x		x	x	x	x	x	x	x		x	x	x	x		x	x	x						x	x	x		x		x	x	x		x	x	x		
Redis	x	x	x								x		x	x	x	x	x		x									x	x		x		x							
HBase	x		x		x			x	x	x	x	x	x	x		x			x	x							x		x		x		x		x				
Riak							x	x	x	x	x	x		x	x	x	x	x	x		x		x					x		x	x	x			x	x		x		
Cassandra	x		x		x		x	x	x	x	x	x		x		x	x	x	x		x		x					x		x		x		x	x	x			x
MySQL	x	x	x	x	x	x	x	x			x		x						x					x				x	x		x	x	x			x	x		

Table 1: A direct comparison of functional requirements, non-functional requirements and techniques among MongoDB, Redis, HBase, Riak, Cassandra and MySQL according to our NoSQL Toolbox.

The comparison elucidates how SQL and NoSQL databases are designed to fulfill very different needs: RDBMSs provide an unmatched level of functionality whereas NoSQL databases excel on the non-functional side through scalability, availability, low latency and/or high throughput. However, there are also large differences among the NoSQL databases. Riak and Cassandra, for example, can be configured to fulfill many non-functional requirements, but are only eventually consistent and do not feature many functional capabilities apart from data analytics and, in case of Cassandra, conditional updates. MongoDB and HBase, on the other hand, offer stronger consistency and more sophisticated functional capabilities such as scan queries and – only MongoDB: – filter queries, but do not maintain read and write availability during partitions and tend to display higher read latencies. Redis, as the only non-partitioned system in this comparison apart from MySQL, shows a special set of trade-offs centered around the ability to maintain extremely high throughput at low-latency using in-memory data structures and asynchronous master-slave replication.

5 Conclusions

Choosing a database system always means to choose one set of desirable properties over another. To break down the complexity of this choice, we present a binary decision tree in Figure 6 that maps trade-off decisions to example applications and potentially suitable database systems. The leaf nodes cover applications ranging from simple caching (left) to Big Data analytics (right). Naturally, this view on the problem space is not complete, but it vaguely points towards a solution for a particular data management problem. The first split in the tree is along the access pattern of applications: They either rely on fast lookups only (left half) or require more complex querying capabilities (right half). The fast lookup applications can be distinguished further by the data volume they process: If the main memory of one single machine can hold all the data, a single-node system like Redis or Memcache probably is the best choice,

depending on whether functionality (Redis) or simplicity (Memcache) is favored. If the data volume is or might grow beyond RAM capacity or is even unbounded, a multi-node system that scales horizontally might be more appropriate. The most important decision in this case is whether to favor availability (AP) or consistency (CP) as described by the CAP theorem. Systems like Cassandra and Riak can deliver an always-on experience, while systems like HBase, MongoDB and DynamoDB deliver strong consistency.

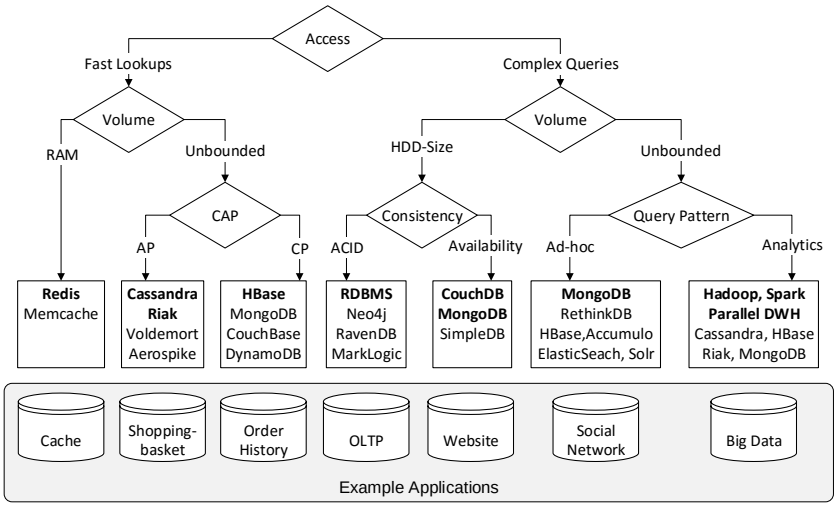

Fig. 6: A decision tree for mapping requirements to (NoSQL) database systems.

The right half of the tree covers applications requiring more complex queries than simple lookups. Here, too, we first distinguish the systems by the data volume they have to handle according to whether single-node systems are feasible (HDD-size) or distribution is required (unbounded volume). For common OLTP workloads on moderately large data volumes, traditional RDBMSs or graph databases like Neo4J are optimal, because they offer ACID semantics. If, however, availability is of the essence, distributed systems like MongoDB, CouchDB or DocumentDB are preferrable.

If the data volume exceeds the limits of a single machine, the choice of the right system depends on the prevalent query pattern: When complex queries have to be optimised for latency, as for example in social networking applications, MongoDB is very attractive, because it facilitates expressive ad-hoc queries. HBase and Cassandra are also useful in such a scenario, but excel at throughput-optimised Big Data analytics, when combined with Hadoop.

In summary, we are convinced that the proposed top-down model is an effective decision support to filter the vast amount of NoSQL database systems based on central requirements. The NoSQL Toolbox furthermore provides a mapping from functional and non-functional requirements to common implementation techniques to categorize the constantly evolving NoSQL space.

Dimension	MongoDB	HBase	Cassandra	Riak	Redis
Model	Document	Wide-Column	Wide-Column	Key-Value	Key-Value
CAP	CP	CP	AP	AP	CP
Scan Performance	High (with appropriate shard key)	High (only on row key)	High (using compound index)	N/A	High (depends on data structure)
Disk Latency per Get by Row Key	∼ Several disk seeks	∼ Several disk seeks	∼ Several disk seeks	∼ One disk seek	In-Memory
Write Performance	High (append-only I/O)	High (append-only I/O)	High (append-only I/O)	High (append-only I/O)	Very high, in-memory
Network Latency	Configurable: nearest slave, master ("read preference")	Designated region server	Configurable: R replicas contacted	Configurable: R replicas contacted	Designated master
Durability	Configurable: none, WAL, replicated ("write concern")	WAL, row-level versioning	WAL, W replicas written	Configurable: writes, durable writes, W replicas written	Configurable: none, periodic loging, WAL
Replication	Master-slave, synchronicity configurable	File-system-level (HDFS)	Consistent hashing	Consistent hashing	Asynchronous master-slave
Sharding	Hash- or range-based on attribute(s)	Range-based (row key)	Consistent hashing	Consistent hashing	Only in Redis Cluster: hashing
Consistency	Master writes, with quorum reads linearizable and else eventual	Linearizable	Eventual, optional linearizable updates ("lightweight transactions")	Eventual, client-side conflict resolution	Master reads: linearizable, slave reads: eventual
Atomicity	Single document	Single row, or explicit locking	Single column (multi-column updates may cause dirty writes)	Single key/value pair	Optimistic multi-key transactions, atomic Lua scripts
Conditional Updates	Yes (mastered)	Yes (mastered)	Yes (Paxos-coordinated)	No	Yes (mastered)
Interface	Binary TCP	Thrift	Thrift or TCP/CQL	REST or TCP/Protobuf	TCP/Plain-Text
Special Data Types	Objects, arrays, sets, counters, files	Counters	Counters	CRDTs for counters, flags, registers, maps	Sets, hashes, counters, sorted Sets, lists, HyperLogLogs, bitvectors
Queries	Query by example (filter, sort, project), range queries, MapReduce, aggregation	Get by row key, scans over row key ranges, project CFs/columns	Get by Partition Key and filter/sort over cluster key, FT-search	Get by ID or local secondary index, materialized views, MapReduce, FT-search	Data Structure Operations
Secondary Indexing	Hash, B-Tree, geo-spatial indexes	None	Local sorted index, global secondary index (hash-based), search index (Solr)	Local secondary indexes, search index (Solr)	Not explicit
License	GPL 3.0	Apache 2	Apache 2	Apache 2	BSD

Table 2: A qualitative comparison of MongoDB, HBase, Cassandra, Riak and Redis.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer 45(2), 37–42 (2012)
2. Attiya, H., Bar-Noy, A., Dolev, D., other: Sharing memory robustly in message-passing systems. JACM 42(1), 124–142 (1995)
3. Bailis, P., Kingsbury, K.: The network is reliable. Commun. ACM 57(9), 48–55 (2014)
4. Baker, J., Bond, C., Corbett, J.C., other: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR. pp. 223–234 (2011)
5. Bernstein, P.A., Cseri, I., Dani, N., other: Adapting microsoft sql server for cloud computing. In: 27th ICDE. pp. 1255–1263. IEEE (2011)
6. Boykin, O., Ritchie, S., O'Connell, I., Lin, J.: Summingbird: A framework for integrating batch and online mapreduce computations. VLDB 7(13), 1441–1451 (2014)
7. Brewer, E.A.: Towards robust distributed systems. (2000)
8. Calder, B., Wang, J., Ogus, A., other: Windows azure storage: a highly available cloud storage service with strong consistency. In: 23th SOSP. ACM (2011)
9. Chang, F., Dean, J., Ghemawat, S., other: Bigtable: A distributed storage system for structured data. In: 7th OSDI. pp. 15–15. USENIX Association (2006)
10. Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice, Lecture Notes in Computer Science, vol. 5959. Springer (2010)
11. Cooper, B.F., Ramakrishnan, R., Srivastava, U., other: Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment 1(2), 1277–1288 (2008)
12. Corbett, J.C., Dean, J., Epstein, M., other: Spanner: Google's globally-distributed database. In: Proceedings of OSDI. pp. 251–264. USENIX Association (2012)
13. Curino, C., Jones, E., Popa, R.A., other: Relational cloud: A database service for the cloud. In: 5th CIDR (2011)
14. Das, S., Agrawal, D., El Abbadi, A., other: G-store: a scalable data store for transactional multi key access in the cloud. In: 1st SoCC. pp. 163–174. ACM (2010)
15. Davidson, S.B., Garcia-Molina, H., Skeen, D., other: Consistency in a partitioned network: a survey. SUR 17(3), 341–370 (1985)
16. Dean, J.: Designs, lessons and advice from building large distributed systems (2009), keynote talk at LADIS 2009
17. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
18. DeC andia, G., Hastorun, D., other: Dynamo: amazon's highly available key-value store. In: 21th SOSP. pp. 205–220. ACM (2007)
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (Apr 1985)
20. Gessert, F., Schaarschmidt, M., Wingerath, W., Friedrich, S., Ritter, N.: The cache sketch: Revisiting expiration-based caching in the age of cloud data management. In: BTW. pp. 53–72 (2015)
21. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (June 2002)
22. Gray, J., Hell and, P., other: The dangers of replication and a solution. SIGMOD Rec. 25(2), 173–182 (Jun 1996)
23. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. 15(4), 287–317 (Dec 1983)
24. Hamilton, J.: On designing and deploying internet-scale services. In: 21st LISA. pp. 18:1–18:12. USENIX Association (2007)

25. Hellerstein, J.M., Stonebraker, M., Hamilton, J.: Architecture of a database system. Now Publishers Inc (2007)
26. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
27. Hoelzle, U., Barroso, L.A.: The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines. Morgan and Claypool Publishers (2009)
28. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: USENIXATC. USENIX Association (2010)
29. Kallman, R., Kimura, H., Natkins, J., other: H-store: a high-performance, distributed main memory transaction processing system. VLDB Endowment (2008)
30. Karger, D., Lehman, E., Leighton, T., other: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: 29th STOC. pp. 654–663. ACM (1997)
31. Kleppmann, M.: Designing data-intensive applications. O Reilly, to appear (2016)
32. Kraska, T., Pang, G., Franklin, M.J., other: Mdcc: Multi-data center consistency. In: 8th EuroSys. pp. 113–126. ACM (2013)
33. Kreps, J.: Questioning the lambda architecture (2014), accessed: 2015-12-17
34. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (Apr 2010)
35. Laney, D.: 3d data management: Controlling data volume, velocity, and variety. Tech. rep., META Group (February 2001)
36. Lloyd, W., Freedman, M.J., Kaminsky, M., other: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: 23th SOSP. ACM (2011)
37. Mahajan, P., Alvisi, L., Dahlin, M., other: Consistency, availability, and convergence. University of Texas at Austin Tech Report 11 (2011)
38. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machines for wans. In: OSDI. vol. 8, pp. 369–384 (2008)
39. Marz, N., Warren, J.: Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications Co. (2015)
40. Min, C., Kim, K., Cho, H., other: Sfs: random write considered harmful in solid state drives. In: FAST. p. 12 (2012)
41. "Ozsu, M.T., Valduriez, P.: Principles of distributed database systems. Springer Science & Business Media (2011)
42. Pritchett, D.: Base: An acid alternative. Queue 6(3), 48–55 (May 2008)
43. Qiao, L., Surlaker, K., Das, S., other: On brewing fresh espresso: Linkedin's distributed data serving platform. In: SIGMOD. pp. 1135–1146. ACM (2013)
44. Sadalage, P.J., Fowler, M.: NoSQL distilled : a brief guide to the emerging world of polyglot persistence. Addison-Wesley, Upper Saddle River, NJ (2013)
45. Shapiro, M., Preguicc a, N., Baquero, C., other: A comprehensive study of convergent and commutative replicated data types. Ph.D. thesis, INRIA (2011)
46. Shukla, D., Thota, S., Raman, K., other: Schema-agnostic indexing with azure documentdb. Proceedings of the VLDB Endowment 8(12), 1668–1679 (2015)
47. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: 23th SOSP. pp. 385–400. ACM (2011)
48. Stonebraker, M., Madden, S., Abadi, D.J., other: The end of an architectural era: (it's time for a complete rewrite). In: 33rd VLDB. pp. 1150–1160 (2007)
49. Wiese, L., other: Advanced Data Management: For SQL, NoSQL, Cloud and Distributed Databases. Walter de Gruyter GmbH & Co KG (2015)
50. Zhang, H., Chen, G., Ooi, B.C., other: In-memory big data management and processing: A survey. TKDE (2015)