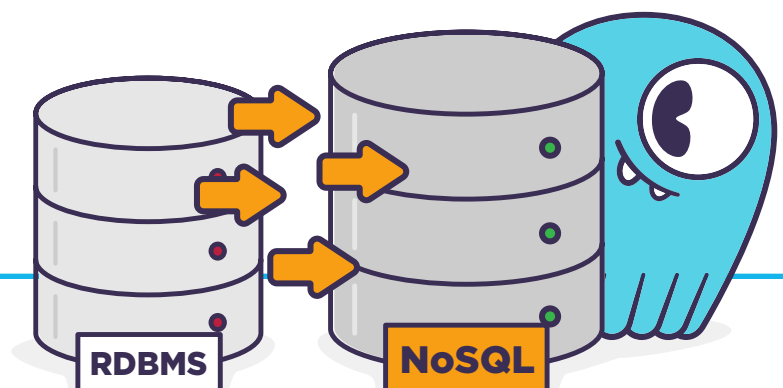


SQL to NoSQL: Architecture Differences and Considerations for Migration



CONTENTS

SQL VERSUS NOSQL BASICS	3
ARCHITECTURAL DIFFERENCES BETWEEN SQL AND NOSQL	4
ECONOMIES OF SCALE	4
REPLICAS OF DATA	5
APPLICATION-DRIVEN USE CASES	5
CONSISTENCY VERSUS AVAILABILITY	6
ACID VERSUS BASE CONSISTENCY	6
LIGHTWEIGHT TRANSACTIONS	7
QUERY LANGUAGES: SQL VERSUS CQL	8
JOINS	8
VALUES VERSUS OBJECTS	8
SCALING CHARACTERISTICS	8
CONSIDERATIONS FOR SQL TO NOSQL MIGRATIONS	8
DATA MODELS	8
QUERY PATTERNS	9
MATERIALIZED VIEWS	10
SECONDARY INDEXES	10
REFERENTIAL INTEGRITY	10
MIGRATION TO NOSQL	10
HYBRID CONVERSION TO NOSQL	10
DATA FORKLIFTING	10
DATA VALIDATION	10
SQL AND NOSQL DATABASE ADMINISTRATION	10
PERFORMANCE MONITORING AND TUNING	10
BACKUP AND RECOVERY	11
NODE REPAIR AND REPLACEMENT	12
SCYLLA NOSQL: SCALE-UP OF RDBMS AND HIGH AVAILABILITY OF NON-RELATIONAL	12

SQL and NoSQL: Understanding the Tradeoffs between Flexibility, Scale and Cost

When and how to migrate data from SQL to NoSQL are matters of much debate. It can certainly be a daunting task, but when your SQL systems hit architectural limits or your cloud provider expenses skyrocket, it's probably time to consider a move.

SQL VERSUS NOSQL BASICS

Since their invention in 1970 by Edgar Codd, relational databases have served as the default data store for almost every IT organization, large or small. Today, the most iconic and familiar relational databases include IBM DB2, Oracle Database, Microsoft SQL Server, PostgreSQL, and MySQL.

Structured Query Language, or SQL, was invented at IBM soon after the introduction of the relational database. Since its introduction, SQL has become the most widely used database language, used for querying data, data manipulation (insert, update and delete), data definition ([schema](#) creation and modification), and data access control. Though the terms refer to different technologies, 'SQL' and 'RDBMS' have become virtually interchangeable. Though some non-relational databases support SQL, the term "SQL database" generally means a relational database.

During the decades in which relational databases proliferated, data entry was largely a manual process. Times have changed. The advent of smartphones, the 'app economy,' and cloud computing in the late 2000s caused a seachange in the workloads, query types, and traffic patterns needed to support a global user base.

Fast forward to 2020, when people, smart-devices, sensors, and machines emit continuous streams of data, such as user activity, IoT and machine-generated data, and metadata that encompasses geolocation and telemetry. As early as 2013, one researcher noted that 90% of the world's data had been generated over the previous two years. This trend has only accelerated. In response to this torrent of data,

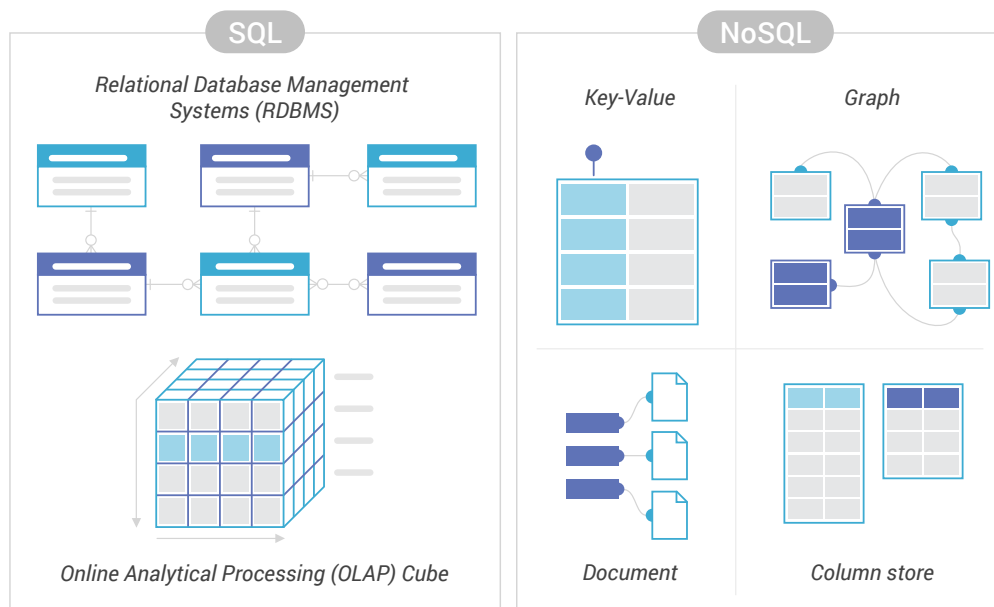
many organizations have been reevaluating their use of traditional relational databases.

Cloud computing exposed many limitations of relational databases. RDBMSs proliferated in an age when databases were isolated islands with relatively stable user bases, running in a traditional client-server configuration. RDBMSs support arbitrary reshaping and joining of data, but performance can be variable and unpredictable. In restricted environments, such variable performance characteristics could be managed. The shift to a mobile, globally dispersed user base caught many organizations off guard.

During the same time period, consumer demands shifted radically. Today, users expect low-latency applications that deliver an extremely responsive experience, regardless of the user's location. Apps that are slow and unresponsive contribute significantly to customer churn. Predictable performance became more important than the semantic flexibility afforded by RDBMSs.

Latency issues can be addressed by shifting data closer to the customer. To meet this need, data must be replicated across different geographic locations. Such geographical replication turned out to be a struggle for RDBMSs. While RDBMSs are not fit for distributed deployments, non-relational databases are designed specifically to support such topologies.

A number of alternative non-relational database systems have been proposed, including Google's Bigtable (2006) and Amazon's Dynamo (2007). The papers for these projects paved the way for Cassandra (2008) and MongoDB (2009).



SQL vs NoSQL

Today, a range of mature NoSQL databases are available to help organizations scale big data applications.

Yet, despite their origins in a long-forgotten technology cycle, relational SQL databases are by no means ‘legacy’ technology. Some SQL databases, notably PostgreSQL and MySQL, have experienced a recent resurgence in popularity. A new generation of NewSQL databases, notably Google Spanner and CockroachDB, leverage SQL as a query language and offer a distributed architecture similar to that of NoSQL databases yet provide full transactional support.

ARCHITECTURAL DIFFERENCES BETWEEN SQL AND NOSQL

ECONOMIES OF SCALE

Database administrators add capacity to RDBMS and NoSQL databases in very different ways. Typically, the only way to add capacity in a relational system is to add expensive hardware, faster CPUs, more RAM, and more advanced networking components. This is often referred to as ‘vertical’ scale, or ‘scaling up.’

In contrast, NoSQL databases are designed for low latency and high resilience, being built from the ground up to run across clusters of distributed nodes. This architecture is often referred to as ‘horizontal scale,’ or ‘scaling out.’ To add capacity to a NoSQL database, administrators simply add more nodes, a very simple process in modern cloud environments.

In a NoSQL cluster, nodes are easy to add and remove according to demand, providing ‘elastic’ capacity. This feature enables organizations to align their data footprint with the needs of the business while maintaining availability even in the face of seasonal demand spikes, node failures, and network outages.

The horizontal scale of NoSQL brings tradeoffs of its own. Adding commodity hardware to a cluster can be cheap in terms of software licenses and subscriptions. However, as more and more nodes are added in the pursuit of higher throughput and lower latency, operational overhead and administrative costs spike. Big clusters of small instances demand more attention and generate more alerts than small clusters of large instances.

(Notably, some next-generation NoSQL databases like Scylla are able to overcome this tradeoff, scaling out in a way that can take

advantage of the powerful hardware in modern servers and ultimately running in smaller, though still distributed, clusters of fewer nodes.)

REPLICAS OF DATA

Replicating data across multiple nodes allows databases to achieve higher levels of resilience. In the RDBMS world, it's not trivial to replicate data across multiple instances. Relational databases do not support replication. Instead, they rely on external tools to extract and update copies of datasets. These tools run batch processes that often take hours to complete. As a result, there is no way to ensure real-time synchronization of data among the copies of data.

While non-relational databases provide native support for data replication, they follow three basic models: multi-master databases, such as DynamoDB, master-slave architectures, such as MongoDB, and masterless, such as Scylla. Given their reliance on master nodes, both multi-master and master-slave architecture introduce a point of failure. When a master goes down, the process of electing a new master introduces a brief downtime. Even though the delay may be minimal, measured in milliseconds, that delay can still cause SLA violations.

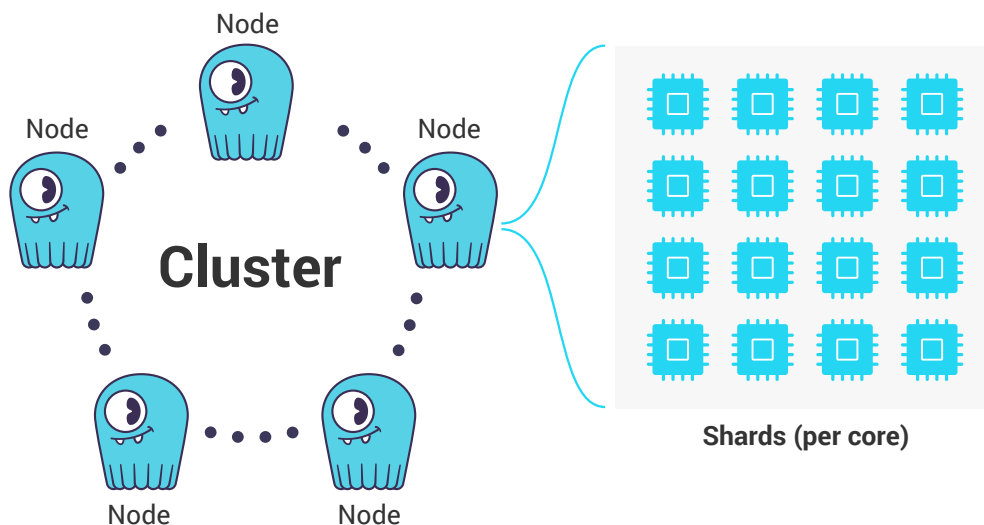
A masterless architecture addresses this limitation. In these databases, data is replicated across multiple nodes, all of which are equal.

In a masterless architecture, no single node can bring down an entire cluster. A typical masterless topology involves three or more replicas for each dataset. Adopting a NoSQL database that implements a masterless architecture provides yet another layer of resilience for high-volume, low-latency applications.

APPLICATION-DRIVEN USE CASES

The rise in popularity of NoSQL databases paralleled the adoption of agile development and DevOps practices. Unlike RDBMSs, NoSQL databases encourage 'application-first' or API-first development patterns. Following these models, developers first consider queries that support the functionality specific to an application, rather than considering the data models and entities. This developer-friendly architecture paved the path to the success of the first generation of NoSQL databases.

In contrast, relational databases impose fairly rigid, schema-based structures to data models; tables consisting of columns and rows, which can be joined to enable 'relations' among entities. Each table typically defines an entity. Each row in a table holds one entry, and each column contains a specific piece of information for that record. The relationships among tables are clearly defined and usually enforced by schemas and database rules.



Masterless Architecture in Scylla

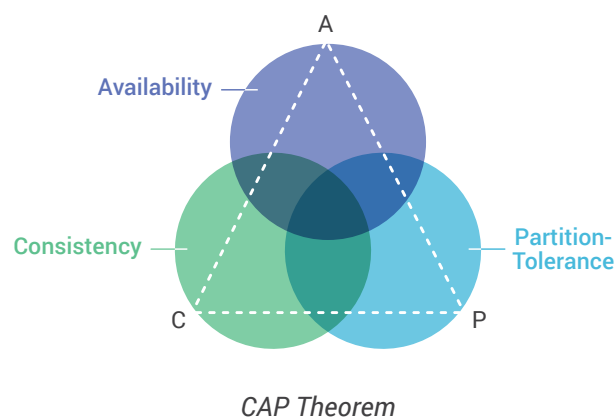
Relational data models enforce uniformity, whereas non-relational models do not. NoSQL databases permit multiple 'shapes' of data objects to coexist, which is more flexible but can also be more error prone. In the world of relational databases, the schemas that support uniformity are usually managed by database administrators. This can sometimes introduce friction between administrators and development teams, resulting in long, non-agile application development lifecycles. Such highly structured data requires normalization to reduce redundancy. Since the data model is based on the entity being represented; query patterns are a secondary consideration.

NoSQL inverts this approach, placing more power in the hands of the developer and often decentralizing control over data structures. Non-relational data models are flexible, and schema management is often delegated to application developers, who are relatively free to adapt data models independently. Such a decentralized approach can accelerate development cycles and provide a more agile approach to addressing user requirements.

CONSISTENCY VERSUS AVAILABILITY

A consideration of the architectural differences between relational and non-relational databases would not be complete without the CAP theorem. The CAP theorem was formulated by Eric Brewer in 2000, as a way of expressing the key tradeoffs in distributed systems. The CAP theorem states that it is impossible for a distributed data store to provide more than two of the following three guarantees:

- **Consistency:** Every read receives either the most recent write or an error.
- **Availability:** Every request receives a response that is not an error, but with no guarantee that it contains the most recent write.
- **Partition Tolerance:** The system continues to operate even when an arbitrary number of messages are delayed, dropped or reordered among nodes.



Another way of putting this is that the CAP theorem dictates that any data store brings with it a fundamental trade-off. As such, many databases are referred to as CP (consistent and partition tolerance, but not available) or AP (available and partition-tolerant, but not consistent). In CAP terms, the critical trade-off that distinguishes relational and non-relational data stores is between availability and consistency. SQL data stores sacrifice availability in favor of data consistency. NoSQL data stores sacrifice consistency in favor of availability.

It is important to note that the CAP theorem has come under significant criticism. Martin Kleppmann, in particular, has written a comprehensive [Critique of the CAP Theorem](#). So, it is important to keep in mind that the theorem is merely a simplified model for understanding a very complex topic.

ACID VERSUS BASE CONSISTENCY

One of the defining tradeoffs between relational and non-relational datastores is in the type of consistency that they provide. In simple terms, RDBMS provides strong consistency, while NoSQL databases provide a weaker form. Consistency in general refers to a database's ability to process concurrent transactions while preserving the integrity of the data. Somewhat confusingly, 'consistency' as defined in the CAP theorem has a different, though related, meaning than the consistency discussed in this section. The definition used by Brewer in the CAP theorem derives from distributed systems theory, while the definition used in this section derives from database theory.

In simple terms, consistency is a guarantee that a read should return the result of the latest successful write. This seems simple, but such a guarantee is incredibly difficult to deliver without impacting the performance of the system as a whole. In a relational database, a single data item is actually split across independent registers that must agree with one another. Thus, a single database write is actually decomposed into several small writes to these registers, which must be completed and visible when the read is executed. With concurrent operations running against the database, the semblance of order between the group of sub-operations needs to be maintained; the concurrent operations must be atomic. ACID consistency means the rules of relations must be satisfied. In a globally distributed database topology, which involves multiple clusters each containing many nodes the problem of consistency becomes exponentially more complex.

In general, relational databases that support 'strong consistency' provide 'ACID guarantees.' ACID is an acronym designed to capture the essential elements of a strongly consistent database. The components of the ACID are as follows:

- **Atomicity:** Guarantees that each transaction is treated as a single "unit", which either succeeds completely or fails completely.
- **Consistency:** Guarantees that each transaction only changes affected data in permitted ways.
- **Isolation:** Guarantees that the concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- **Durability:** The transactions results are permanent, even in the event of system failure.

ACID compliance is a complex and often contested topic. In fact, one popular system of analysis, the [Jepsen test](#), is dedicated to verifying vendor consistency claims.

By their nature, ACID-compliant databases are generally slow, difficult to scale, and expensive

to run and maintain. It should be noted some RDBMS systems enable performance to be improved by relaxing ACID guarantees. Still, all SQL databases are ACID compliant to varying degrees, and as such, they all share this downside. The practical effect of ACID compliance is to make it extraordinarily difficult and expensive to achieve resilient, distributed SQL database deployments.

In contrast to RDBMS' ACID guarantees, NoSQL databases provide so-called 'BASE guarantees.' BASE enables availability and relaxes the stringent consistency. The acronym BASE designates:

- **Basic Availability:** Data is available most of the time, even during a partial system failure.
- **Soft state:** Individual data items are independent and do not have to be consistent with each other.
- **Eventual consistency:** Data will become consistent at some unspecified point in the future.

As such, NoSQL databases sacrifice a degree of consistency in order to increase availability. Rather than providing strong consistency, NoSQL databases generally provide eventual consistency. A data store that provides BASE guarantees can occasionally fail to return the result of the latest write, providing different answers to applications making requests. Developers building applications against eventually consistent data stores often implement consistency checks in their application code.

Lightweight transactions

In a traditional SQL RDBMS, a "transaction" is a logical unit of work — a group of tasks that provides the ACID guarantees discussed above. To compensate for relaxed consistency, some NoSQL databases offer 'lightweight transactions' (LWTs).

Lightweight transactions are limited to a single conditional statement, which enables an atomic "compare and set" operation. Such an operation checks whether a condition is true before it conducts the transaction. If the condition is not

met, the transaction is not executed. (For this reason, LWTs are sometimes called ‘conditional statements’). LWTs do not truly lock the database for the duration of the transaction; they only ‘lock’ a single cell or row. LWTs leverage a consensus protocol such as Paxos to ensure that all nodes in the cluster agree the change is committed. In this way, LWTs can provide sufficient consistency for applications that require the availability and resilience of a distributed database.

QUERY LANGUAGES: SQL VERSUS CQL

As we’ve noted, relational databases are defined in part by their use of the Structured Query Language (SQL). In contrast, NoSQL databases employ a host of alternative query languages that have been designed to support diverse application use cases. A partial list includes MongoDB Query Language (MQL), Couchbase’s N1QL, Elasticsearch’s Query DSL, Microsoft Azure’s Cosmos DB query language, and Cassandra Query Language (CQL).

In this paper, we will focus on the most widely used NoSQL query language, CQL. While CQL is the primary language for communicating with Apache Cassandra, it is also supported by a range of familiar NoSQL databases. Common CQL-compliant databases include Scylla, DataStax Enterprise, Microsoft’s cloud-native Azure Cosmos DB, and Amazon Keyspaces.

CQL’s similarity to SQL enables developers to move between the languages with relative ease. A few distinctions between SQL and CQL include:

Joins

SQL and CQL share similar statements to store and modify data, such as Create, Alter, Drop, and Truncate commands, but unlike SQL, CQL is not designed to support joins between tables. In CQL, relations are implemented within the application, rather than within the database query.

Values versus objects

Query results are also returned differently. SQL natively returns data-typed values, usually to be read into an object one field at a time. In contrast, CQL natively returns complete objects, often serialized in extensible markup language (XML) or Javascript object notation (JSON). This makes applications responsible for parsing these objects to obtain the desired result of a query.

Scaling characteristics

In NoSQL, data is stored across nodes in a cluster based on a token range, which is a hashed value of the primary key. By using token ranges, NoSQL databases enable objects to be stored on different nodes. CQL queries are inherently more scalable than SQL queries, having been specifically designed to query across a horizontally distributed cluster of servers, rather than a single database at a time.

CONSIDERATIONS FOR SQL TO NOSQL MIGRATIONS

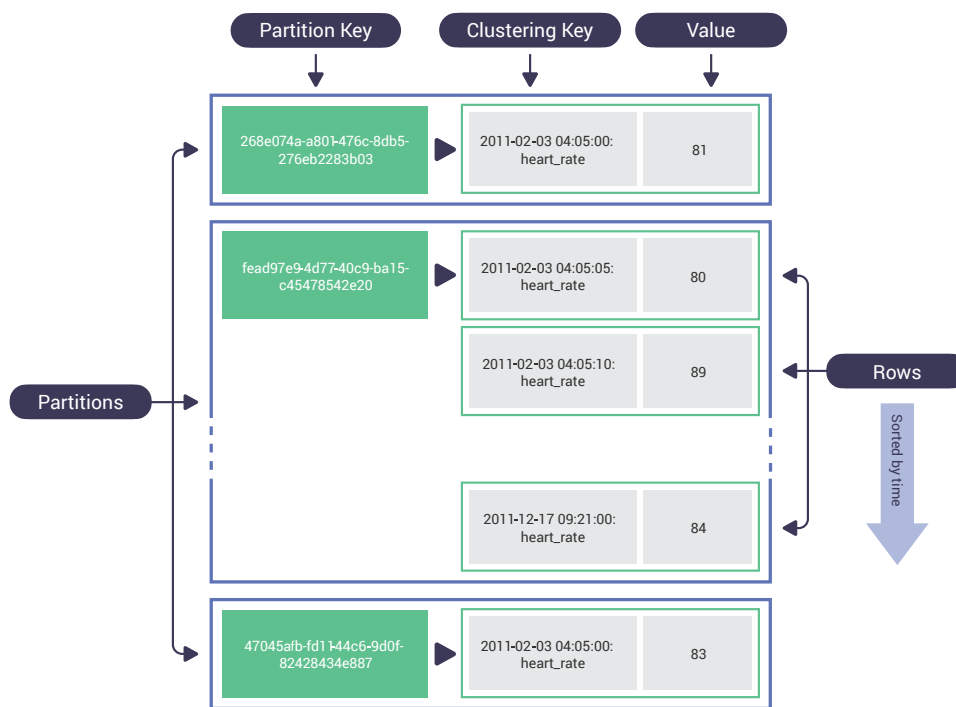
Data models

SQL data models follow a normalized design; different but related pieces of information are stored in ‘relations,’ which are separate logical tables connected by joins. NoSQL databases use denormalized data models, in which redundant copies of data are added as needed by the consuming applications. The point of denormalization is to increase performance and lower latency since the joins involved in normalized data models can introduce significant performance overhead, especially in distributed topologies.

When migrating from SQL to NoSQL, the primary key in the relational table becomes the partition key in the NoSQL table. If the RDBMS table must be joined to additional tables to retrieve the business object, those closely related tables should combine into a single NoSQL table. The NoSQL cluster ordering key determines the physical order of records, so it should be a unique value (often a composite value) that would be useful for searching.

Project Code	Project Name	Project Manager	Project Budget	Employee No.	Employee Name	Department No.	Department Name	Hourly Rate
PC010	Reservation System	Mr. Ajay	120500	\$100	Mohan	D03	Database	21.00
PC010	Reservation System	Mr. Ajay	120500	\$101	Vipul	D02	Testing	16.50
PC010	Reservation System	Mr. Ajay	120500	\$102	Riyaz	D01	IT	22.00
PC011	HR System	Mrs. Charu	500500	\$103	Pavan	D03	Database	18.50
PC011	HR System	Mrs. Charu	500500	\$104	Jitendra	D02	Testing	17.00
PC011	HR System	Mrs. Charu	500500	\$315	Pooja	D01	IT	23.50
PC012	Attendance System	Mr. Rajesh	710700	\$137	Rahul	D03	Database	21.50
PC012	Attendance System	Mr. Rajesh	710700	\$218	Avneesh	D02	Testing	15.50
PC012	Attendance System	Mr. Rajesh	710700	\$109	Vikas	D01	IT	20.50

Denormalized data



Example of Partition

Of course, partition keys and cluster ordering keys are not the only way data is queried. Additional indexes on the relational table provide the basis for secondary indexes or materialized views, in order to support an application's search and filtering requirements.

QUERY PATTERNS

Relational databases are organized around data structures and relationships. In contrast, NoSQL databases are organized around query patterns. As noted above, the NoSQL partition key can be mapped to a primary key in an RDBMS.

Secondary keys and indexes can be added later. A UNIQUE constraint in a SQL database becomes a good candidate for a cluster ordering key in NoSQL.

Materialized views

Common, frequent queries against a database can become expensive. When the same query is run again and again, it makes sense to ‘virtualize’ the query. Materialized views address this need by enabling common queries to be represented by a database object that is continuously updated as data changes.

Secondary indexes

[Secondary indexes](#) enable queries to run against the main table using indexed values, as in an RDBMS, but it is actually implemented as a materialized view. The application is isolated from having to query the secondary index directly.

Referential integrity

Referential integrity ensures that no references between tables are broken, as occurs when a foreign key references a non-existent entry. A lack of referential integrity in a database can result in incomplete query responses, usually failing quietly, with no indication of an error. Relational databases are designed to enforce referential integrity. NoSQL databases shift the responsibility for making sure that objects are complete and correct to the API, which checks entities when loading or saving them.

MIGRATION TO NOSQL

Hybrid conversion to NoSQL

Rather than migrating an entire RDBMS to a NoSQL database, some applications benefit from leaving some data on a relational database, while moving a subset of data to a NoSQL database. A hybrid solution that spans two database types can offer the best of both worlds. For example, in some deployments, customer account information, which is infrequently updated, will be stored in an RDBMS, while transactional or streaming data, such as IoT sensor data or telemetry, might be stored in a NoSQL database. Large, growing

tables and storage for streaming data, especially in the context of event-driven architecture (EDA), are good candidates to migrate to NoSQL.

Data forklifting

Tools like Apache Kafka can facilitate the process of migrating existing data from an RDBMS to NoSQL. Depending on the complexity of the conversion, more comprehensive operations may be needed. Tools such as [Apache Spark](#), a lightning-fast unified analytics engine for big data and machine learning, can be used to enable such data conversions.

For key-value migrations, the forklifting process is trivial. For a document model with hierarchical entities (order lines, for example) the process of building the new value from the old table joins can become much more involved.

Data validation

Following a migration, teams need to validate that data was migrated in the correct form. Missing or truncated data, which has the potential to degrade system capabilities, may not be immediately obvious, making comprehensive testing essential.

During the validation process, the old and new data stores can be run in parallel; new data is added or updated in both datastores simultaneously. Reports can be run against both systems and compared for accuracy. Running some queries across a large range of data will help to find differences that could be magnified across aggregated data.

SQL AND NOSQL DATABASE ADMINISTRATION

Performance Monitoring and Tuning

As development teams become smaller and more agile, they are also increasingly sensitive to database maintenance and administrative overhead. On application-focused teams, database experts are becoming less and less common. Traditional database administration and maintenance responsibilities are often rolled into ‘full-stack’ developer and DevOps positions. Operational overhead is often in direct competition with product development efforts.

For these reasons, the choice of a database must take into account the expertise of the organization and the need or desire to build up internal expertise around a given technology.

The ongoing maintenance of a database requires close monitoring and frequent performance tuning. As datasets grow and application traffic increases, administrators need to keep a close eye on disk space, CPU consumption, memory allocation, and index fragmentation. Performance adjustments are proprietary to each database and often require significant dedicated expertise.

A database administrator never wants to see database utilization spike over 100%. Therefore, administrators must provide a buffer against traffic spikes by 'overprovisioning' hardware. The degree to which hardware must be overprovisioned depends on the scaling characteristics of the database. In general, NoSQL databases have a flatter and more predictable performance curve. Therefore, NoSQL databases tend to require administrators to minimize overprovisioning without compromising safety.

Performance tuning can be used to minimize overprovisioning, but it can only go so far in preventing full utilization. When performance tuning hits a wall, the database must be scaled; the RDBMS administrator has two choices. First, the dataset can be 'sharded,' such that a subset of the data is stored on each node. Second, the administrator can add more powerful hardware, increasing the capacity of hardware by adding more powerful CPUs, more storage, and faster networking components.

Often, teams do both, sharding and scaling, which adds both complexity and cost. The vertical scale adds significant cost at each step and eventually runs up against the physical limits of the network.

NoSQL databases make it easier for administrators to monitor and manage database deployments. First, they tend to be capable of running at higher levels of utilization than most RDBMSs. Second, capacity can be increased

by adding new nodes running on inexpensive commodity servers. But within the family of NoSQL databases, these two capabilities vary considerably.

Some NoSQL databases also require expert administrators with detailed knowledge of proprietary tuning settings. Others adopt a more automated approach that minimizes tricky manual tuning parameters, enabling non-specialties to administer and operate the database.

Likewise, some NoSQL databases take a horizontal scale to an extreme, often requiring huge clusters to achieve the required performance targets and maintain SLAs. Sometimes these clusters run into the tens of thousands of nodes. While providing a frictionless path to scale, this approach also increases operational overhead. The ideal non-relational database can efficiently use powerful modern hardware, while also enabling clusters to grow and shrink elastically with minimal administrator intervention.

Backup and Recovery

In both RDBMS and NoSQL worlds, data can become corrupt due to hardware issues, software bugs, and user errors. The resilient architecture of NoSQL databases typically provides a buffer against data loss. Still, administrators need to be able to restore the data to a known 'good state.' A backup and recovery plan is essential, being built around two core targets: Recovery Point Objective (RPO) and Recovery Time Objective (RTO).

- RPO is defined by the age of data in backup storage needed to resume normal operations after a failure.
- RTO defines the time needed to restore the system to a normal state.

A classic database restore plan might include a single daily backup along with differential backups every hour to support a one-hour RPO. For a large database, the recovery time for a full restore can take hours to days, and every backup takes additional storage space.

Node Repair and Replacement

Given the distributed nature of NoSQL clusters, nodes occasionally fall out-of-sync. To address this issue, NoSQL databases provide tools to bring out-of-sync nodes up-to-date using a repair procedure. Repairs populate the node to match the data on the other replicas. Sometimes a node can fall so far out-of-sync with the cluster that it needs to be replaced. As they are bootstrapped into the cluster, fresh nodes must stream a copy of the whole dataset; for large datasets, such a refresh can take an inordinate amount of time. NoSQL databases perform such operations using a variety of algorithms, some of which are more efficient than others. Thus, some NoSQL databases recover more quickly and predictably than others.

SCYLLA NOSQL: SCALE-UP OF RDBMS AND HIGH AVAILABILITY OF NON-RELATIONAL

In this document, we have discussed a set of trade offs between SQL and NoSQL databases. If your use case requires ACID guarantees, then NoSQL might not be an option. But

many modern, cloud-native applications are better suited to databases that support high availability and a developer-centric data model. The decision is based on business considerations: how important is each transaction? Where the aggregate scale and speed of all transactions outweighs the specific correctness of any single query, then NoSQL is the best fit.

With this fundamental tradeoff in mind, one database, Scylla, has been designed from the ground-up to overcome one of the key limitations of the first generation of NoSQL databases. Using a unique, close-to-the-hardware design, Scylla combines the scale up capabilities of traditional RDBMSs with the high availability and resilience of non-relational databases. The result is a database that extracts maximum performance from modern hardware to deliver predictable, low latency, while also minimizing operational overhead and significantly reducing TCO.

Many IT organizations have followed the principles in this paper and have migrated successfully from RDBMS to the Scylla NoSQL database.

	SQL	NoSQL
Orientation	Relational	Generally non-relational
Schema	Strict and rigid schema design and data normalization	Loose and more varied designs for unstructured and semi-structured data; data is generally denormalized
Language	Structured Query Language (SQL) for defining, reading and manipulating data. Supports JOIN statements to relate data across tables.	There are different languages for querying, some quite similar to SQL, such as Cassandra Query Language (CQL) for wide column databases, or others radically different, such as using object-oriented JSON for document databases.
Scalability	Vertically scalable. Loads on a single server can be increased with CPU, RAM or SSD.	Generally designed for horizontal scalability. Increased traffic can be handled by adding more servers in the database. This is useful for large and frequently changing datasets.
Structure	Table-based, which is efficient for applications using multi-row transactions or systems that were built with a relational structure.	NoSQL database structure is variable, and can be based on documents, key-value pairs, graph structures or wide-column stores.

ABOUT SCYLLADB

Scylla is the real-time big data database. API-compatible with Apache Cassandra and Amazon DynamoDB, Scylla embraces a shared-nothing approach that increases throughput and storage capacity as much as 10X. Comcast, Discord, Disney+ Hotstar, Grab, Medium, Starbucks, Ola Cabs, Samsung, IBM, Investing.com and [many more leading companies](#) have adopted Scylla to realize order-of-magnitude performance improvements and reduce hardware costs. Scylla's database is available as an open source project, an enterprise edition and a fully managed database as a service. ScyllaDB was founded by the team responsible for the KVM hypervisor. For more information: [ScyllaDB.com](https://scylladb.com)

SCYLLADB.COM



SCYLLA

United States Headquarters
2445 Faber Place, Suite 200
Palo Alto, CA 94303 U.S.A.
Email: info@scylladb.com

Israel Headquarters
11 Galgalei Haplada
Herzeliya, Israel



Copyright © 2020 ScyllaDB Inc. All rights reserved. All trademarks or registered trademarks used herein are property of their respective owners.