NoSQL

Source: NoSQL Databases

Christof Strauch

www.christof-strauch.de/nosqldbs.pdf

NoSQL

- The term NoSQL was first used in 1998 for a relational database that omitted the use of SQL
- The term was picked up again in 2009 and used for conferences of advocates of non-relational databases
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties

NoSQL

- Stands for **N**ot **O**nly **SQL**

NoSQL !=

NoSQL

- “NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favor of more efficient and cheaper ways of managing data.”

  Computerworld magazine

- Web 2.0 startups have begun their business without Oracle and even without MySQL
- Instead, they built their own datastores influenced by Amazon’s Dynamo and Google’s Bigtable in order to store and process huge amounts of data like they appear e.g. in social community or cloud computing applications

NoSQL

- Most of these datastores became open source software.
- For example, Cassandra originally developed for a new search feature by Facebook is now part of the Apache Software Project.

NoSQL features

- **Avoidance of Unneeded Complexity:** Relational databases provide a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases.
- **High Throughput:** Some NoSQL databases provide a significantly higher data throughput than traditional RDBMSs
- **Horizontal Scalability and Running on Commodity Hardware**: Machines can be added and removed (or crash) without causing the same operational efforts to perform distribution in RDBMS cluster-solutions

NoSQL features

- **Avoidance of Expensive Object-Relational Mapping:** Most of the NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented programming languages compared to relational data structures
- **Compromising Reliability for Better Performance**
- **The Current "One size fit's it all" Databases Thinking Was and Is Wrong**

NoSQL features

- **The Myth of Effortless Distribution and Partitioning of Centralized Data Models:** data models originally designed with a single database in mind often cannot easily be partitioned and distributed among database servers
- **Movements in Programming Languages and Development Frameworks:** provide abstractions for database access trying to hide the use of SQL and relational databases

NoSQL Features

- **Requirements of Cloud Computing:** two major requirements of datastores in cloud computing environments
  1. High until almost ultimate scalability—especially in the horizontal direction
  2. Low administration overhead

NoSQL Features

- **The RDBMS plus Caching-Layer Pattern/Workaround vs. Systems Built from Scratch with Scalability in Mind:** distribute MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together.
- Memchached: partitioned—though transient— in-memory database
- It replicates most frequently requested parts of a database to main memory, rapidly delivers this data to clients and therefore disburdens database servers significantly.

Main memory

- Compared to the 1970s, enormous amounts of main memory have become cheap and available
- “The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing [. . .] quite slowly”
- Such databases are “capable of main memory deployment now or in near future”. Stonebraker et al.
- The OLTP market a main memory market even today or in near future.

CAP Theorem

- **Consistency** meaning if and how a system is in a consistent state after the execution of an operation.
- A distributed system is typically considered to be consistent if after an update operation of some writer all readers see his updates in some shared data source.
- **Availability** meaning that a system is designed and implemented in a way that allows it to continue operation (i.e. allowing read and write operations) if e.g. nodes in a cluster crash or some hardware or software parts are down due to upgrades.

CAP Theorem

- **Partition Tolerance** understood as the ability of the system to continue operation in the presence of network partitions. These occur if two or more “islands” of network nodes arise which (temporarily or permanently) cannot connect to each other

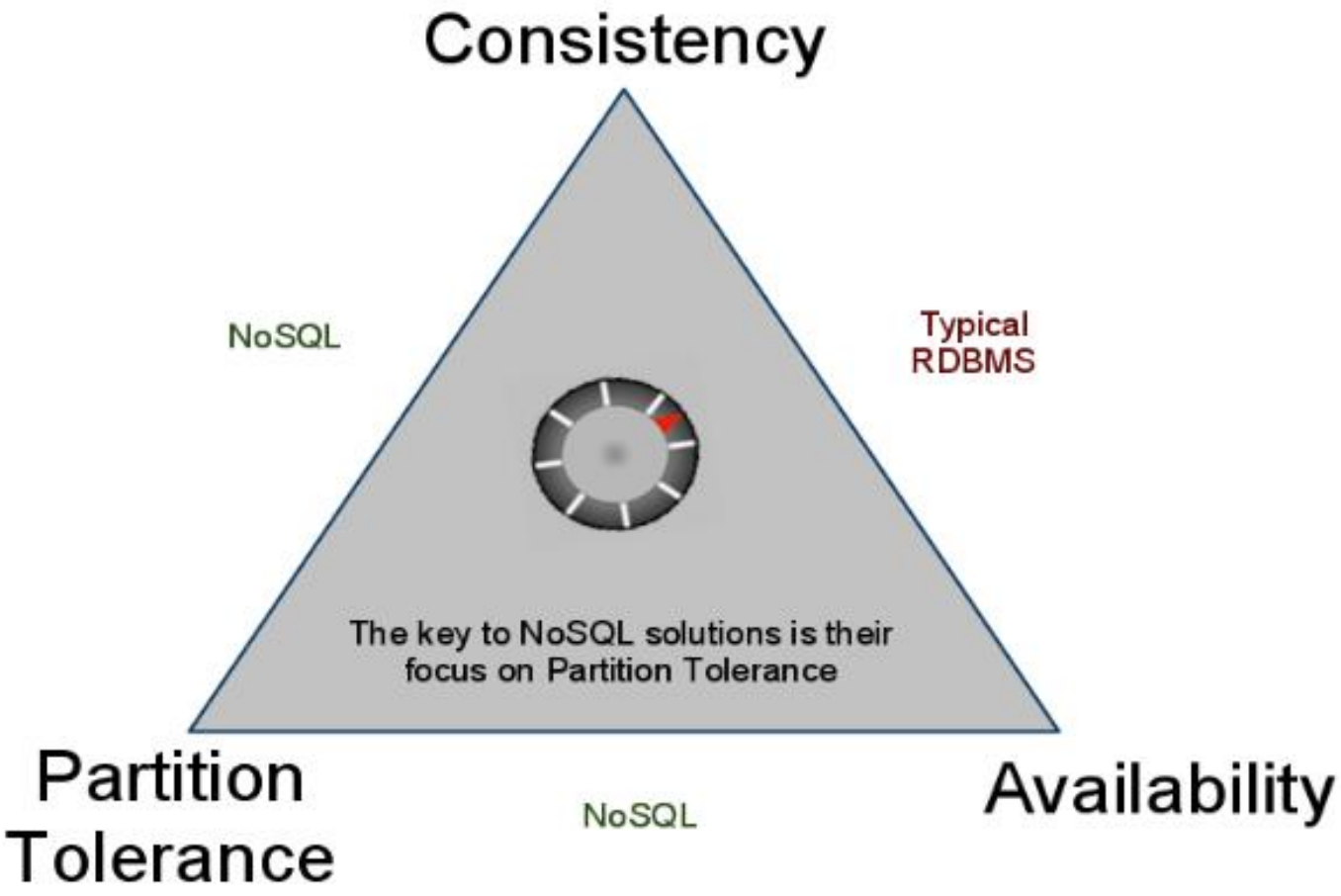
CAP Theorem

It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees

- **C**onsistency: all nodes see the same data at the same time
- **A**vailability: every request receives a response about whether it was successful or failed
- **P**artition Tolerance: the system continues to operate despite arbitrary message loss

You have to choose only two. In almost all cases, you would choose availability over consistency

CAP Theorem


Consistency
NoSQL
Typical
RDBMS
The key to NoSQL solutions is their
focus on Partition Tolerance
Partition
Tolerance
NoSQL
Availability

ACID vs. BASE

- The internet with its wikis, blogs, social networks etc. creates an enormous and constantly growing amount of data needing to be processed, analyzed and delivered.
- Companies, organizations and individuals offering applications or services in this field have to determine their individual requirements regarding performance, reliability, availability, consistency and durability
- For a growing number of applications and use-cases (including web applications, especially in large and ultra-large scale, and even in the e-commerce sector), availability and partition tolerance are more important than strict consistency.

BASE

- The BASE approach forfeits the ACID properties of consistency and isolation in favor of “availability, graceful degradation, and performance”
- The acronym BASE is composed of the following characteristics:
  - **B**asically **a**vailable
  - **S**oft-state
  - **E**ventual consistency
- An application works basically all the time (basically available), does not have to be consistent all the time (soft-state) but will be in some known state eventually (eventual consistency)

Strict Consistency

- All read operations must return data from the latest completed write operation, regardless of which replica the operations went to
- This implies that either read and write operations for a given dataset have to be executed on the same node or that strict consistency is assured by a distributed transaction protocol (like two-phase-commit or Paxos).
- As we have seen above, such a strict consistency cannot be achieved together with availability and partition tolerance according to the CAP-theorem

Eventual Consistency

- Readers will see writes, as time goes on
- In a steady state, the system will eventually return the last written value
- Clients therefore may face an inconsistent state of data as updates are in progress.
- For instance, in a replicated database updates may go to one node which replicates the latest version to all other nodes that contain a replica of the modified dataset so that the replica nodes eventually will have the latest version.

Eventual Consistency

- An eventually consistent system may provide more differentiated, additional guarantees to its clients
- **Read Your Own Writes (RYOW) Consistency** signifies that a client sees his updates immediately after they have been issued and completed, regardless if he wrote to one server and in the following reads from different servers.
- Updates by other clients are not visible to him instantly

Partitioning

- Assuming that data in large scale systems exceeds the capacity of a single machine and should also be replicated to ensure reliability and allow scaling measures such as load-balancing, ways of partitioning the data of such a system have to be thought about.
- Approaches:
  - **Memory Caches**

Memory Caches

- Can be seen as partitioned—though transient—in-memory databases as they replicate most frequently requested parts of a database to main memory, rapidly deliver this data to clients and therefore disburden database servers significantly (e.g. memcached).
- In the case of memcached the memory cache consists of an array of processes with an assigned amount of memory that can be launched on several machines in a network and are made known to an application via configuration.

Sharding

- **Sharding** means to partition the data in such a way that data typically requested and updated together resides on the same node and that load and storage volume is roughly evenly distributed among the servers
- Data shards may also be replicated for reasons of reliability and load-balancing and it may be either allowed to write to a dedicated replica only or to all replicas maintaining a partition of the data.
- To allow such a sharding scenario there has to be a mapping between data partitions (shards) and storage nodes that are responsible for these shards.

Sharding

- This mapping can be static or dynamic, determined by a client application, by some dedicated “mapping-service/component” or by some network infrastructure between the client application and the storage nodes
- The downside of sharding scenarios is that joins between data shards are not possible, so that the client application or proxy layer inside or outside the database has to issue several requests and postprocess (e.g. filter, aggregate) results instead.

Sharding

- In a partitioned scenario knowing how to map database objects to servers is key. An obvious approach may be a simple hashing of database-object primary keys against the set of available database nodes in the following manner:
- *partition = hash(o) mod n with o = object to hash, n = number of nodes*
- The downside of this procedure is that the data have to be redistributed whenever nodes leave and join

Sharding

- In a setting where nodes may join and leave at runtime (e.g. due to node crashes, temporal unattainability, maintenance work) a different approach such as consistent hashing has to be found

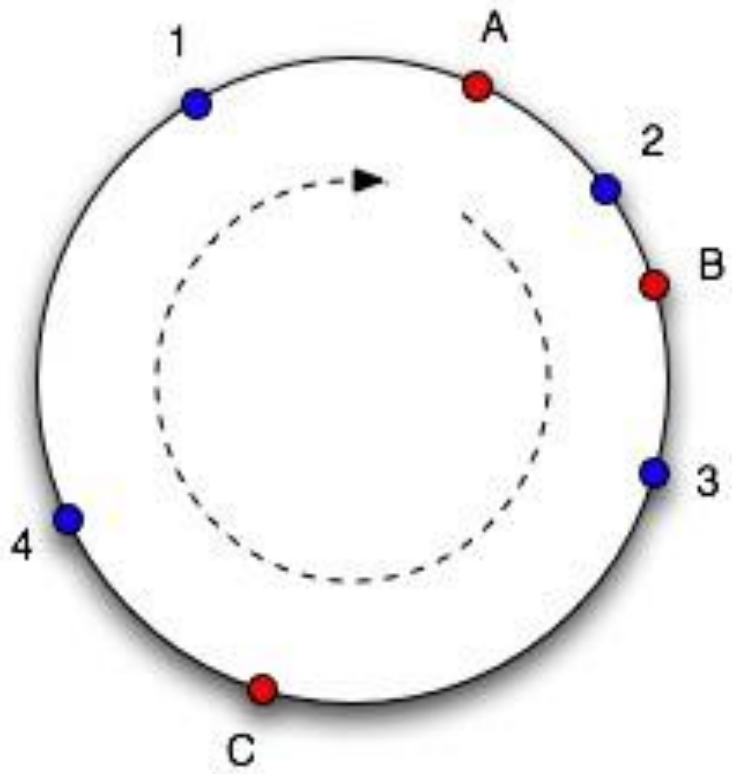
Consistent Hashing

- The basic idea behind the consistent hashing algorithm is to hash both objects and nodes using the same hash function
- Not only hashing objects but also machines has the advantage that machines get an interval of the hash-function's range and adjacent machines can take over parts of the interval of their neighbors if those leave and can give parts of their own interval away if a new node joins and gets mapped to an adjacent interval

Consistent Hashing

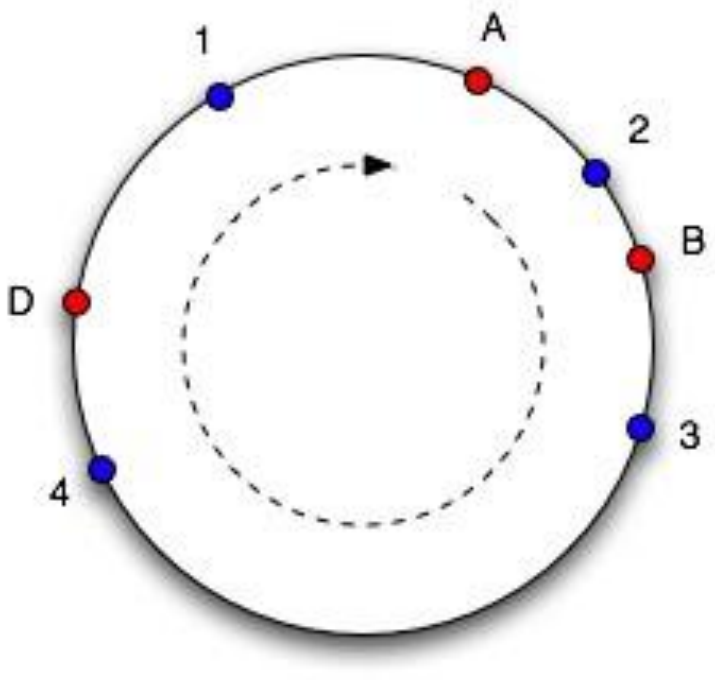
- The consistent hashing approach has the advantage that client applications can calculate which node to contact in order to request or write a piece of data and there is no metadata server necessary as in systems like the the Google File System (GFS) which has a central (though clustered) metadata server that contains the mappings between storage servers and data partitions

Consistent Hashing



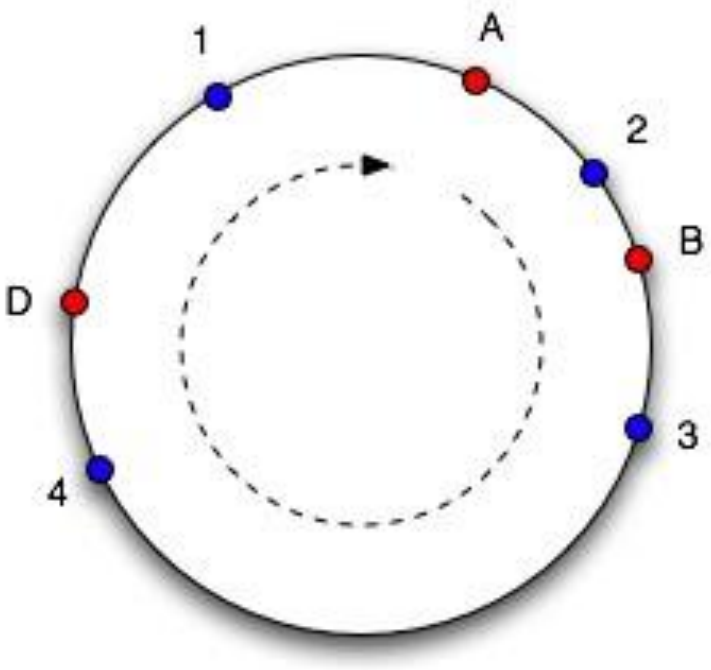
- Three red colored nodes A, B and C and four blue colored objects 1–4 are mapped to a hash-function's result range pictured as a ring.
- Objects are mapped by moving clockwise
- objects 4 and 1 are mapped to node A, object 2 to node B and object 3 to node C.

Consistent Hashing



- When a node leaves the system, objects will get mapped to their adjacent node (in clockwise direction) and when a node enters the system it will get hashed onto the ring and will overtake objects

Consistent Hashing



- Node C left and node D entered the system, so that now objects 3 and 4 will get mapped to node D and only 1 to A
- By changing the number of nodes not all objects have to be remapped to the new set of nodes but only part of the objects.

Data models

- key-/value-stores
- document databases
- column-oriented databases

Key-/value-stores

- Simple data model: a map/dictionary, allowing clients to put and request values per key.
- Besides the data-model and the API, modern key-value stores favor high scalability over consistency and therefore most of them also omit rich ad-hoc querying and analytics features (especially joins and aggregate operations are set aside)
- Often, the length of keys to be stored is limited to a certain number of bytes while there is less limitation on values
- A large number of this class of NoSQL stores has been heavily influenced by Amazon's Dynamo

Other Key-/Value-Stores

- Tokyo Cabinet and Tokyo Tyrant
- Redis
- Memcached and MemcacheDB
- Scalaris

Document Databases

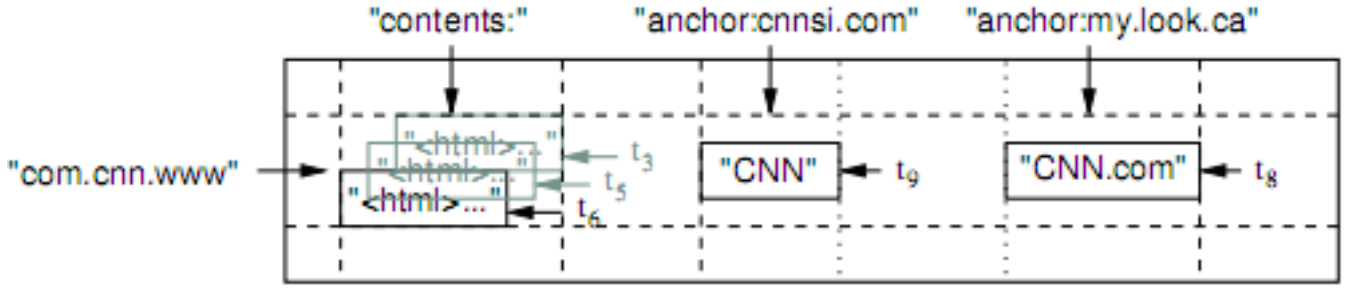
- They allow to encapsulate key-/value-pairs in documents.
- There is no strict schema documents have to conform to which eliminates the need of schema migration efforts
- The two major representatives for the class are
  - Apache CouchDB
  - MongoDB

Column-Oriented Databases

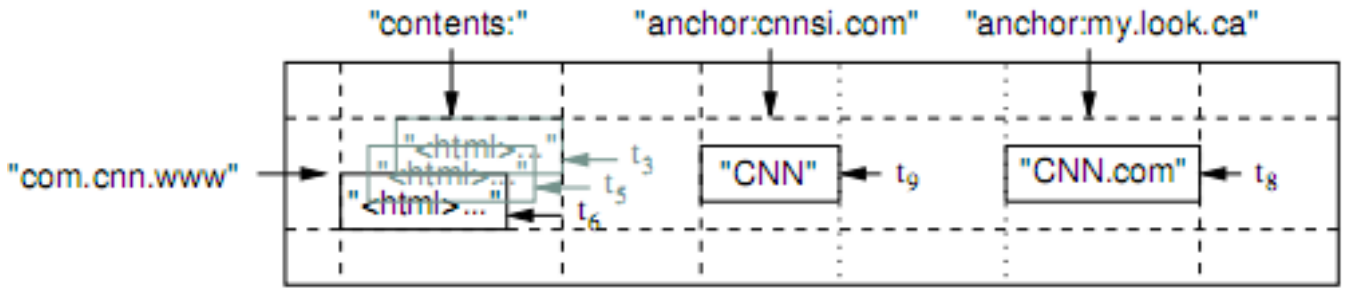
- The approach to store and process data by column instead of row has its origin in analytics and business intelligence where column-stores operating in a shared-nothing massively parallel processing architecture can be used to build high-performance applications.
- The class of column-oriented stores is seen as less puristic, also subsuming datastores that integrate column- and row-orientation
- The main inspiration for column-oriented datastores is Google's Bigtable
- Cassandra is inspired by Bigtable

Google's Bigtable

- The data structure provided and processed by Google's Bigtable is described as "a sparse, distributed, persistent multidimensional sorted map".
- Values are stored as arrays of bytes which do not get interpreted by the data store. They are addressed by the triple (row-key, column-key, timestamp)
- Example of a Bigtable storing information a web crawler might emit



Bigtable



- The map contains a non-fixed number of rows representing domains read by the crawler as well as a non-fixed number of columns:
  - the first of these columns (contents:) contains the page contents
- the others (anchor:<domain-name>) store link texts from referring domains—each of which is represented by one dedicated column