NoSQL

Source: NoSQL Databases

Christof Strauch

www.christof-strauch.de/nosqldbs.pdf

NoSQL

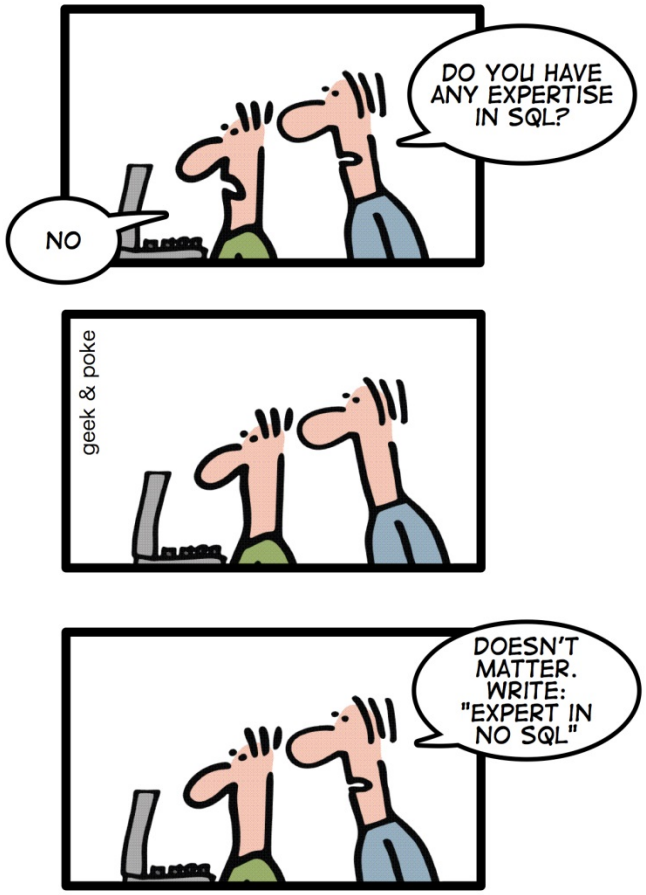
- The term NoSQL was first used in 1998 for a relational database that omitted the use of SQL
- The term was picked up again in 2009 and used for conferences of advocates of non-relational databases
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they 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 !=

HOW TO WRITE A CV



Leverage the NoSQL boom

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 sharding 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
- **Complexity and Cost of Setting up Database Clusters**
- **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:** Shard 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 deliver this data to clients and therefore disburden database servers significantly.

Main memory

- As—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** and especially high 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

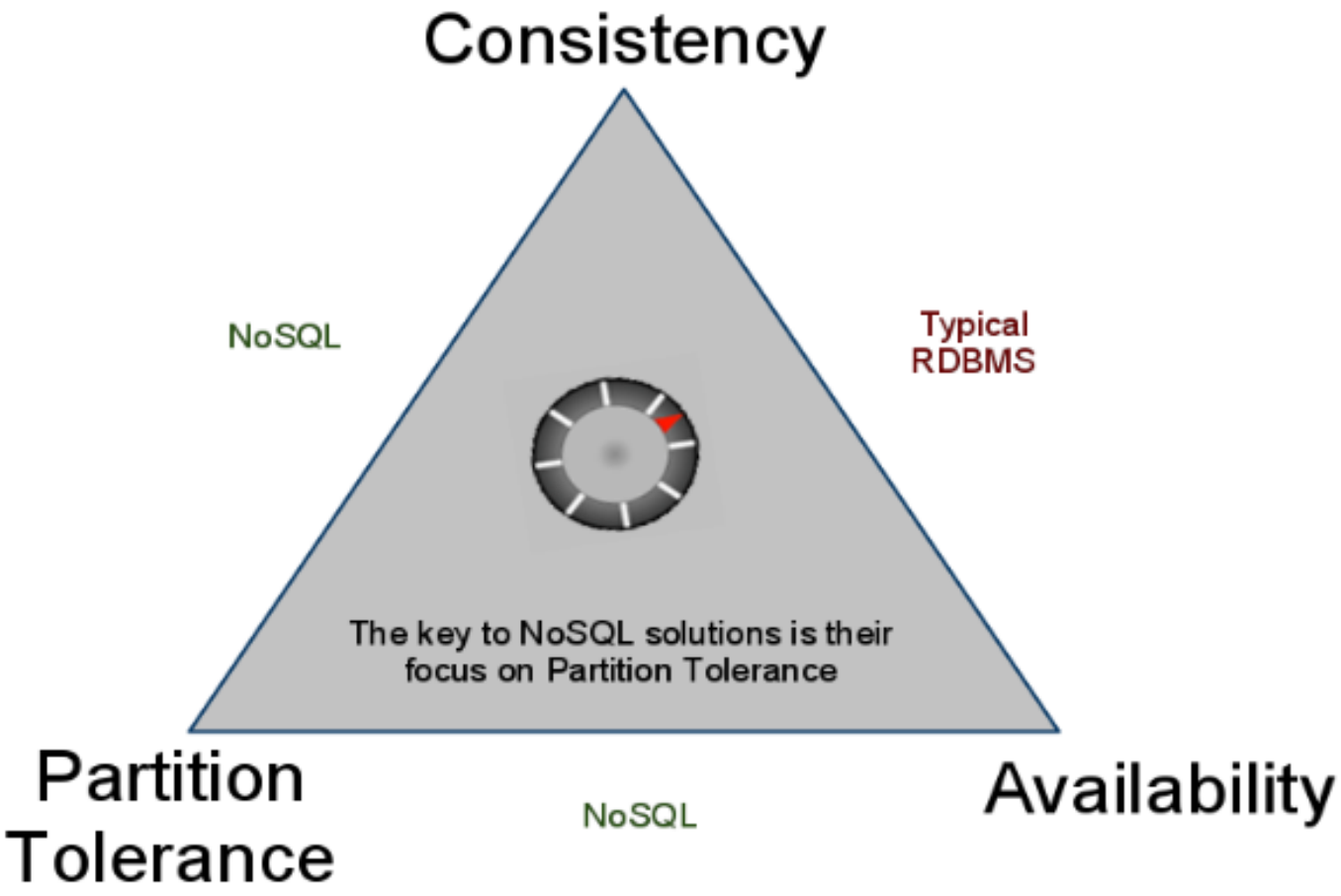
CAP Theorem

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

Versioning of Datasets in Distributed Scenarios

- If datasets are distributed among nodes, they can be read and altered on each node and no strict consistency is ensured by distributed transaction protocols
- Questions arise on how “concurrent” modifications and versions are processed and to which values a dataset will eventually converge to.

Solutions to versioning

- **Timestamps** seem to be an obvious solution for developing a chronological order. However, timestamps “rely on synchronized clocks and don’t capture causality”
- **Optimistic Locking** implies that a unique counter or clock value is saved for each piece of data. When a client tries to update a dataset it has to provide the counter/clock-value of the revision it likes to update
- **Vector Clocks** are an alternative approach to capture order and allow reasoning between updates in a distributed system

Solutions to versioning

- **Multiversion Storage** means to store a timestamp for each table cell. These timestamps “don’t necessarily need to correspond to real life”, but can also be some artificial values that can be brought into a definite order.
- For a given row multiple versions can exist concurrently.
- Besides the most recent version a reader may also request the “most recent before T” version.

Vector clocks

- A vector clock is defined as a tuple *V* [0], *V* [1], ...,*V* [*n*] of clock values
- In a distributed scenario node i maintains such a tuple of clock values, which represent the state of itself and the other (replica) nodes' state it is aware about at a given time (*Vi*[0] for the clock value of the first node, *Vi*[1] for the clock value of the second node, . . . *Vi*[*i*] for itself, . . . *Vi*[*n*] for the clock value of the last node).
- Clock values may be real timestamps derived from a node's local clock, version/revision numbers or some other ordinal values.

Vector clocks

- As an example, the vector clock on node number 2 may take on the following values:
- $V2[0] = 45$, $V2[1] = 3$, $V2[2] = 55$
- This reflects that from the perspective of the second node, the following updates occurred to the dataset the vector clock refers to:
  - an update on node 1 produced revision 3
  - an update on node 0 lead to revision 45
  - the most recent update is encountered on node 2 itself which produced revision 55.

Vector clocks updates

- Vector clocks are updated in a way defined by the following rules
  - If an internal operation happens at node i, this node will increment its clock *Vi*[*i*]. This means that internal updates are seen immediately by the executing node
  - If node i sends a message to node k, it first advances its own clock value *Vi*[*i*] and attaches the vector clock *Vi* to the message to node k. Thereby, he tells the receiving node about his internal state and his view of the other nodes at the time the message is sent.

Vector clocks updates

– If node i receives a message from node j, it first advances its vector clock $Vi[i]$ and then merges its own vector clock with the vector clock *Vmessage* attached to the message from node j so that:
  - $Vi = max(Vi, Vmessage)$

To compare two vector clocks *Vi* and *Vj* in order to derive a partial ordering, the following rule is applied:

- $Vi > Vj$, *if* $\forall k\ Vi[k] > Vj[k]$

If neither $Vi > Vj$ nor $Vi < Vj$ applies, a conflict caused by concurrent updates has occurred and needs to be resolved by e.g. a client application.

Vector clocks for consistency

- Vector clocks can be utilized to resolve consistency between writes on multiple replicas
- Replica nodes do typically not maintain a vector clock for clients but clients participate in the vector clock scenario in such a way that they keep a vector clock of the last replica node they have talked to and use this vector clock depending on the client consistency model that is required; e.g. for monotonic read consistency a client attaches this last vector clock it received to requests and the contacted replica node makes sure that the vector clock of its response is greater than the vector clock the client submitted. This means that the client can be sure to see only newer versions of some piece of data

Advantages of vector clocks

- Compared to the alternative approaches mentioned above (timestamps, optimistic locking with revision numbers, multiversion storage) the advantages of vector clocks are:
  - No dependence on synchronized clocks
  - No total ordering of revision numbers required
  - No need to store and maintain multiple revisions of a piece of data on all nodes

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.

Memory Caches

- The memcached protocol whose implementation is available in different programming languages to be used in client applications provides a simple key-/value-store API.
- It stores objects placed under a key into the cache by hashing that key against the configured memcached-instances

Clustering

- **Clustering** of database servers is another approach to partition data which strives for transparency towards clients who should not notice talking to a cluster of database servers instead of a single server.
- While this approach can help to scale the persistence layer of a system to a certain degree many criticize that clustering features have only been added on top of DBMSs that were not originally designed for distribution

Separating Reads from Writes

- Write-operations for all or parts of the data are routed to master(s)
- A number of replica-servers satisfy read requests (slaves).
- If the master replicates to its clients asynchronously there are no write lags but if the master crashes before completing replication to at least one client the write-operation is lost
- If the master replicates writes synchronously the update does not get lost, but write lags cannot be avoided. If the master crashes the slave with the most recent version of data can be elected as the new master.

Separating Reads from Writes

- The master-/slave-model works well if the read/write ratio is high.
- The replication of data can happen either by transfer of state (i.e. copying of the recent version of data or delta towards the former version) or by transfer of operations which are applied to the state on the slaves nodes and have to arrive in the correct order

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 at least parts of 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

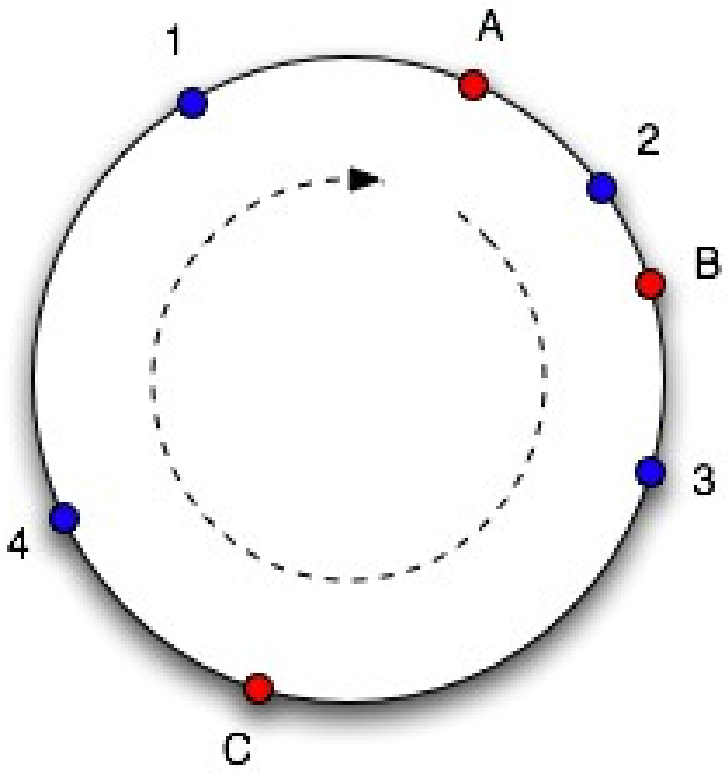
Consistent Hashing

- 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

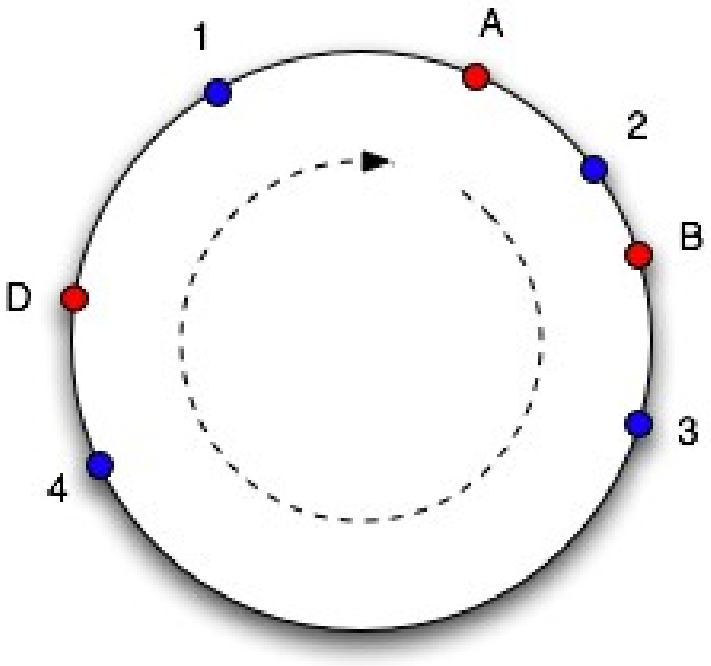
- The consistent hashing approach has the advantage that client applications can calculate which node to contact in order to request or write a piece of data and there is no metadata server necessary as in systems like the the Google File System (GFS) which has such a central (though clustered) metadata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

Consistent Hashing



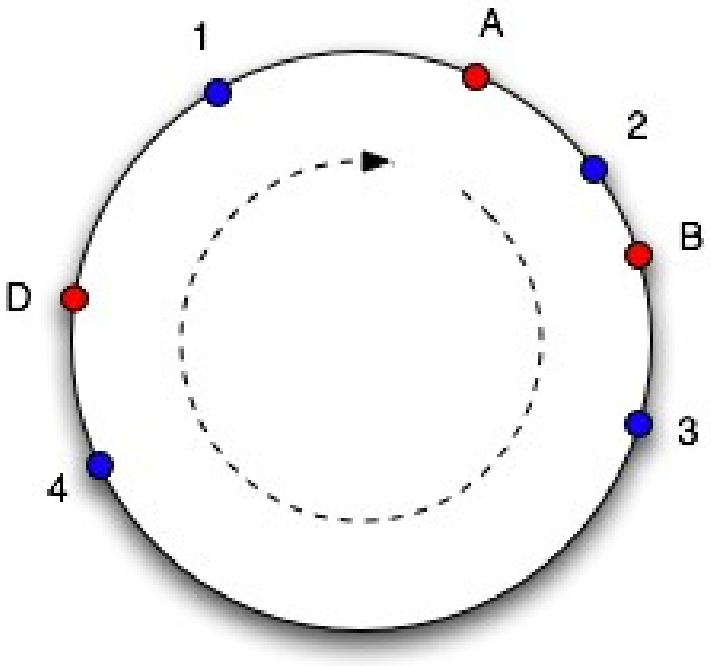
- Three red colored nodes A, B and C and four blue colored objects 1–4 are mapped to a hash-function's result range pictured as a ring.
- Objects are mapped by moving clockwise
- objects 4 and 1 are mapped to node A, object 2 to node B and object 3 to node C.

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

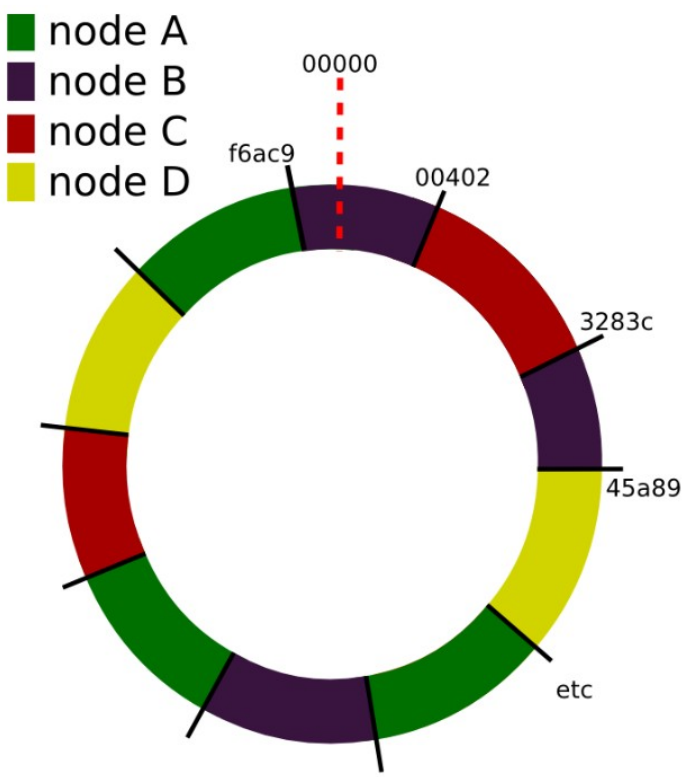


- Node C left and node D entered the system, so that now objects 3 and 4 will get mapped to node D
- By changing the number of nodes not all objects have to be remapped to the new set of nodes but only part of the objects.

Virtual Nodes

- Issues with this procedure: at first, the distribution of nodes on the ring is actually random as their positions are determined by a hash function and the intervals between nodes may be “unbalanced” which in turn results in an unbalanced distribution of cache objects on these nodes
- Solution: hash a number of representatives/ replicas—also called virtual nodes—for each physical node onto the ring

Virtual Nodes

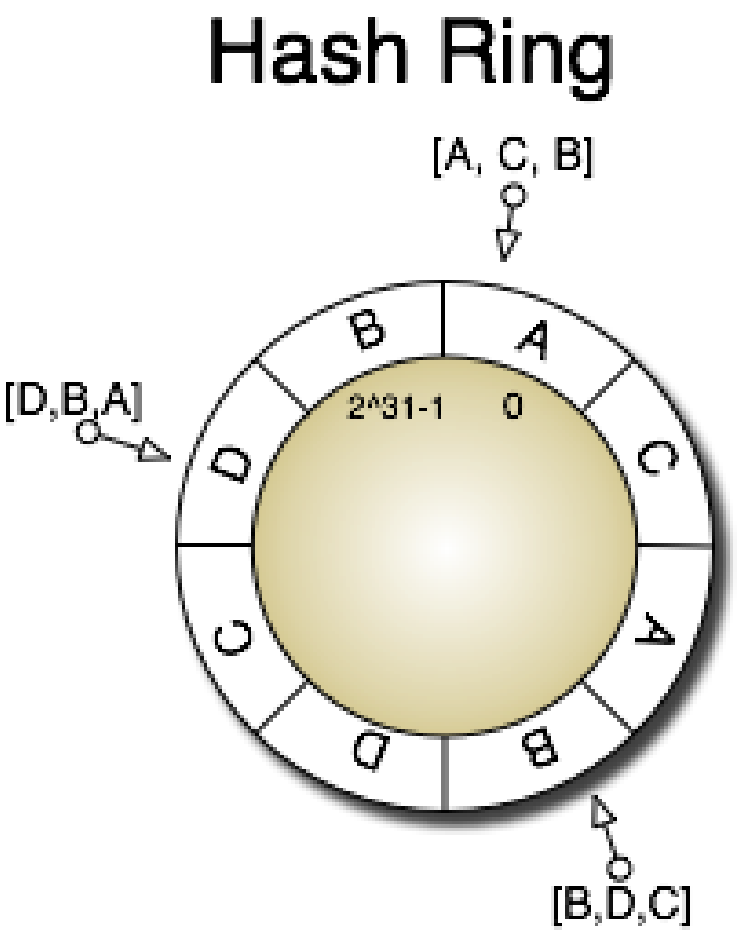


- The number of virtual nodes for a physical can be defined individually according to its hardware capacity (cpu, memory, disk capacity) and does not have to be the same for all physical nodes.
- By appending e.g. a replica counter to a node's id which then gets hashed, these virtual nodes should distribute points for this node all over the ring.

Replication Factor

- If a node has left the scene, data stored on this node becomes unavailable, unless it has been replicated to other nodes before
- In the opposite case of a new node joining the others, adjacent nodes are no longer responsible for some pieces of data which they still store but not get asked for anymore as the corresponding objects are no longer hashed to them by requesting clients.
- Solution: a replication factor (*r*) is introduced: not only the next node but the next *r* (physical!) nodes in clockwise direction become responsible for an object

Replication Factor



- The uppercase letters represent storage nodes and the circles with arrows represent data objects which are mapped onto the ring at the depicted positions.
- r=3 so for every data object three physical nodes are responsible which are listed in square brackets in the figure.

Read and write operations

- Introducing replicas in a partitioning scheme—besides reliability benefits—also makes it possible to spread workload for read requests that can go to any physical node responsible for a requested piece of data.

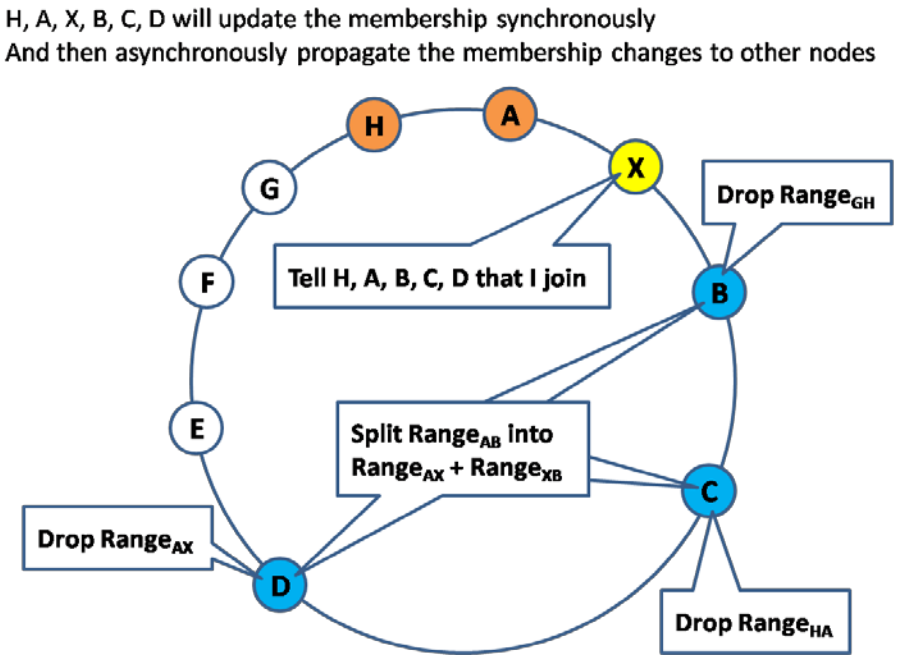
Membership Changes

- In a partitioned database where nodes may join and leave the system at any time without impacting its operation all nodes have to communicate with each other, especially when membership changes.

New node

- When a new node joins the system the following actions have to happen
  1. The newly arriving node announces its presence and its identifier to adjacent nodes or to all nodes via broadcast.
  2. The neighbors of the joining node react by adjusting their object and replica ownerships.
  3. The joining node copies datasets it is now responsible for from its neighbors. This can be done in bulk and also asynchronously.
  4. If, in step 1, the membership change has not been broadcasted to all nodes, the joining node is now announcing its arrival

New node

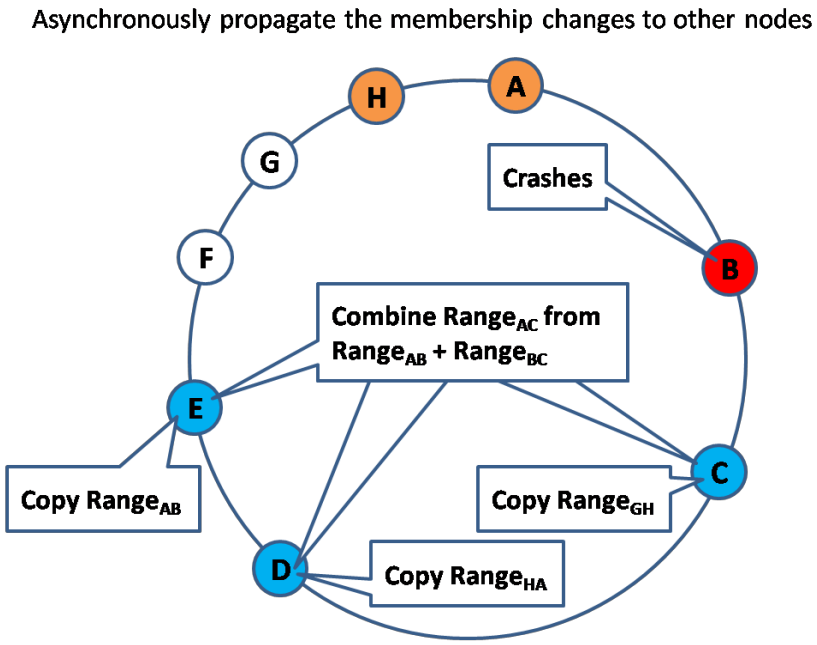


- Node X joins a system for which r=3
- It is hashed between A and B, so that the nodes H, A and B transfer data to the new node X and after that the nodes B, C and D can drop parts of their data for which node X is now responsible as a third replica (in addition to nodes H, A and B).

Node leaving

- When a node leaves the system the following actions have to occur
- Nodes within the system need to detect whether a node has left as it might have crashed and not been able to notify the other nodes of its departure. It is also common in many systems that no notifications get exchanged when a node leaves.
- If a node's departure has been detected, the neighbors of the node have to react by exchanging data with each other and adjusting their object and replica ownerships.

Node leaving



- Node B leaves the system. Nodes C, D and E become responsible for new intervals of hashed objects and therefore have to copy data from nodes in counterclockwise direction and also reorganize their internal representation of the intervals

Cluster management

- Internal nodes may need to find each other
- Since nodes may fail and recover, a configuration file doesn't really suffice
- We need a way of keeping some kind of consistent view of the cluster state

Omniscient Master

- When nodes join/leave or change state, they talk to a master
- That master holds the authoritative view of the world
- Pros: simplicity, single consistent view of the cluster
- Cons: potential Single Point of Failure (SPOF) unless master is made highly available. Not partition-tolerant.

Gossip

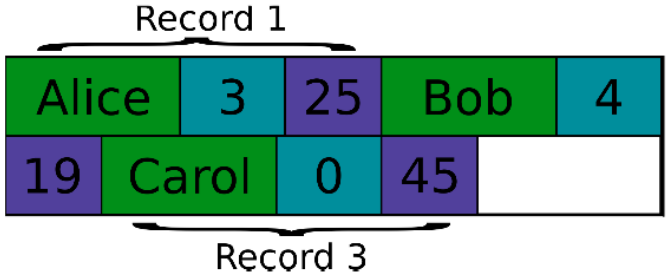
- Gossip is one method to propagate a view of cluster status
  - Every t seconds, on each node:
  - The node selects some other node to chat with.
  - The node reconciles its view of the cluster with its gossip buddy
  - Each node maintains a “timestamp" for itself and for the most recent information it has from every other node
- Information about cluster state spreads in O(log n) rounds (eventual consistency)
- Scalable and no SPOF, but state is only eventually consistent

Storage Layout

- It determines how the disk is accessed and therefore directly implicate performance.
- Furthermore, the storage layout defines which kind of data (e.g. whole rows, whole columns, subset of columns) can be read en bloque

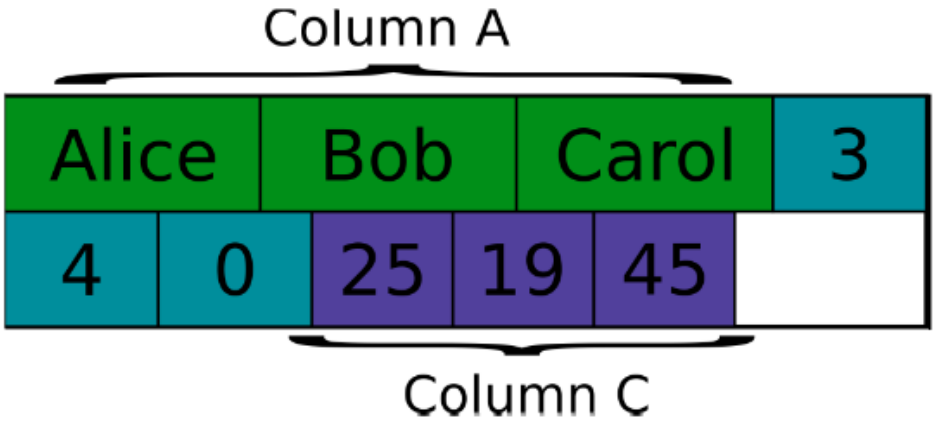
Row-Based Storage Layout

- Means that a table of a relational model gets serialized as its lines are appended and flushed to disk
- The advantages of this storage layout are that at first whole datasets can be read and written in a single IO operation and that secondly one has a good locality of access (on disk and in cache) of different columns
- On the downside, operating on columns is expensive as a considerable amount data (in a naïve implementation all of the data) has to be read.



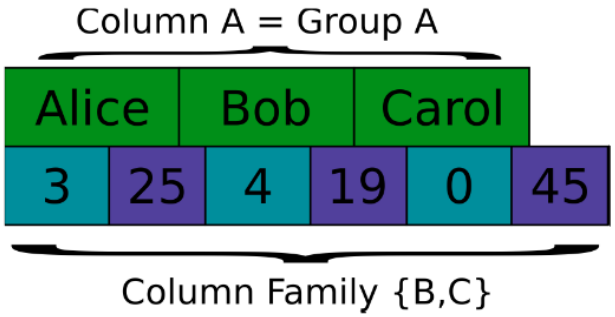
Columnar Storage Layout

- Serializes tables by appending their columns and flushing them to disk
- Therefore operations on columns are fast and cheap while operations on rows are costly and can lead to seeks in a lot or all of the columns.
- A typical application field for this type of storage layout is analytics where an efficient examination of columns for statistical purposes is important.



Columnar Storage Layout with Locality Groups

- Is similar to column-based storage but adds the feature of defining so called locality groups that are groups of columns expected to be accessed together by clients.
- The columns of such a group may therefore be stored together and physically separated from other columns and column groups
- The idea of locality groups was introduced in Google's Bigtable paper.



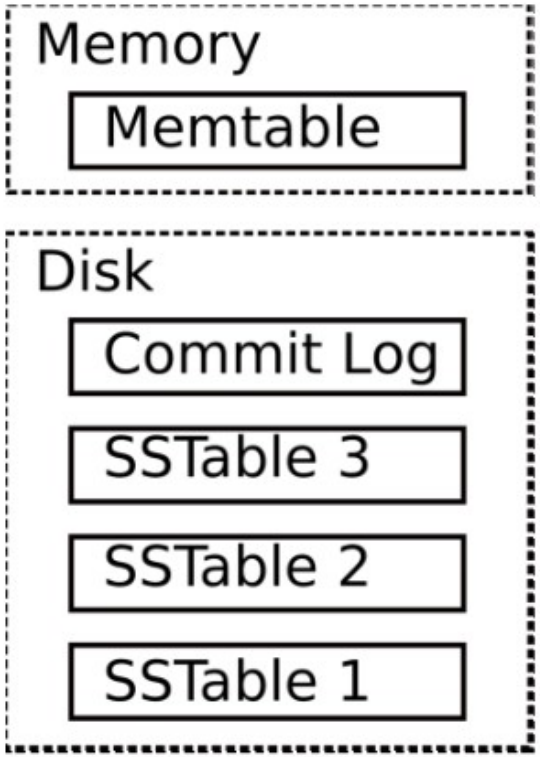
Log Structured Merge Trees

- (LSM-trees aka «The BigTable model») in contrast to the storage layouts explained before do not describe how to serialize logical datastructures (like tables, documents etc.) but how to efficiently use memory and disk storage in order to satisfy read and write requests in an efficient, performant and still safely manner.
- The idea is to hold chunks of data in memory (in so called Memtables), maintaining on-disk commit-logs for these in-memory data structures and flushing the memtables to disk from time to time into so called SSTables

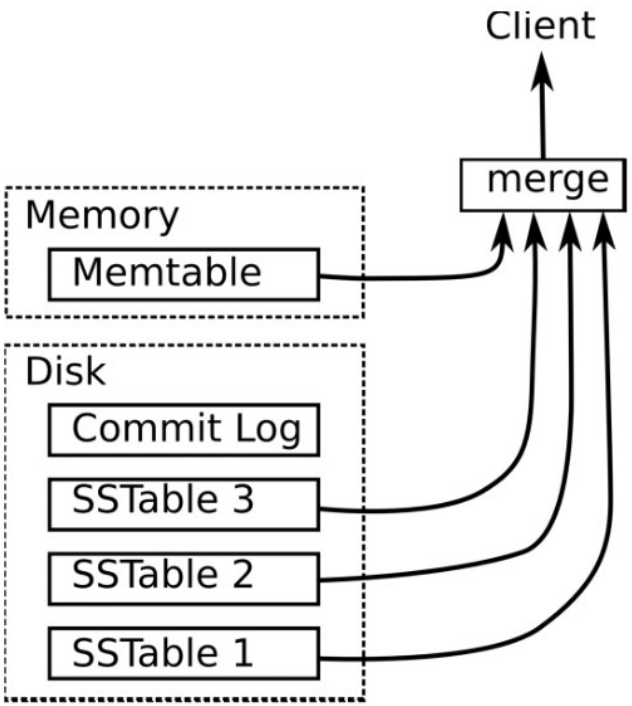
Log Structured Merge Trees

- Random IO for writes is bad (and impossible in some distributed file systems)
- LSM Trees convert random writes to sequential writes
- Writes go to a commit log and in-memory storage (Memtable)
- The Memtable is occasionally flushed to disk (SSTable)
- The disk stores are periodically compacted

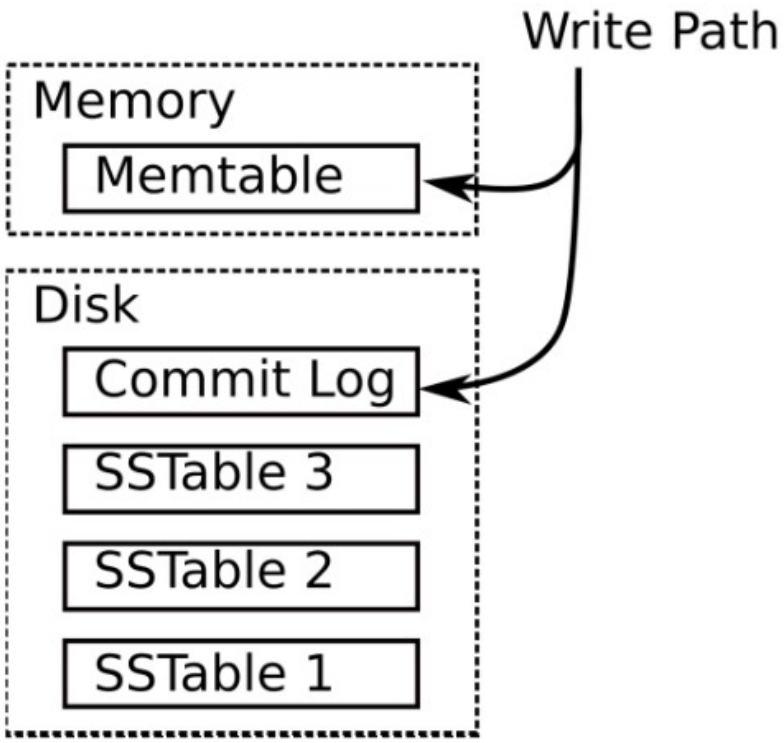
LSM Data Layout



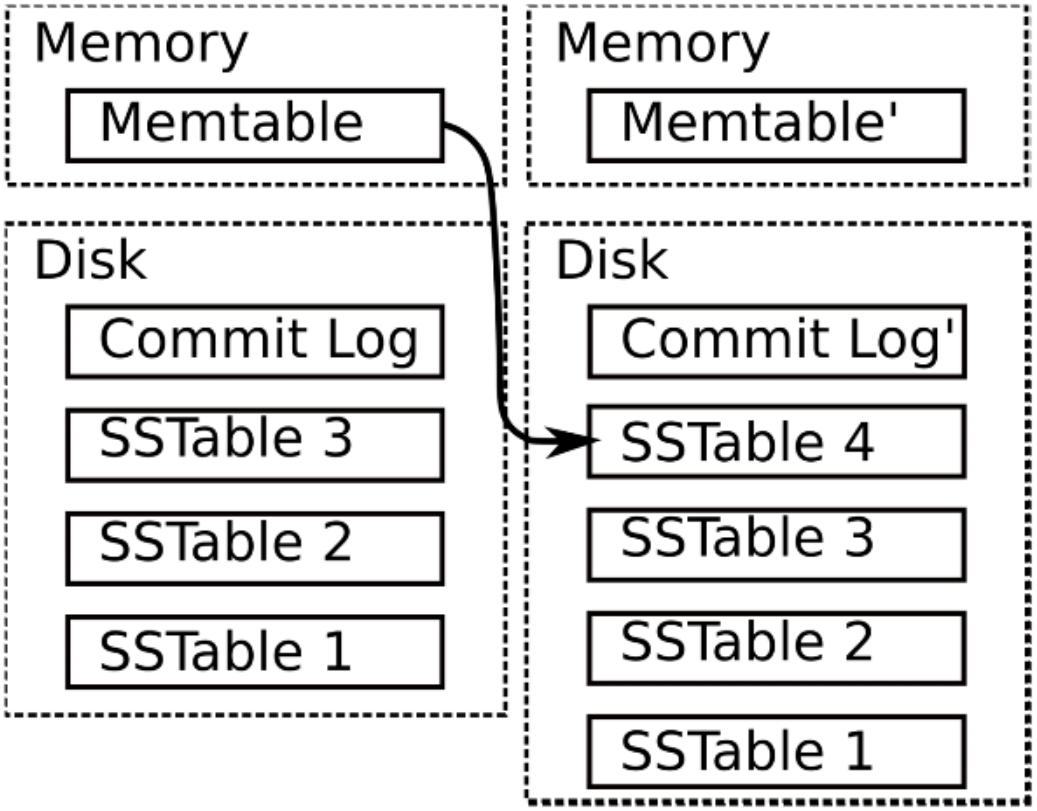
Read path


Client
merge
Memory
Memtable
Disk
Commit Log
SSTable 3
SSTable 2
SSTable 1

Write path

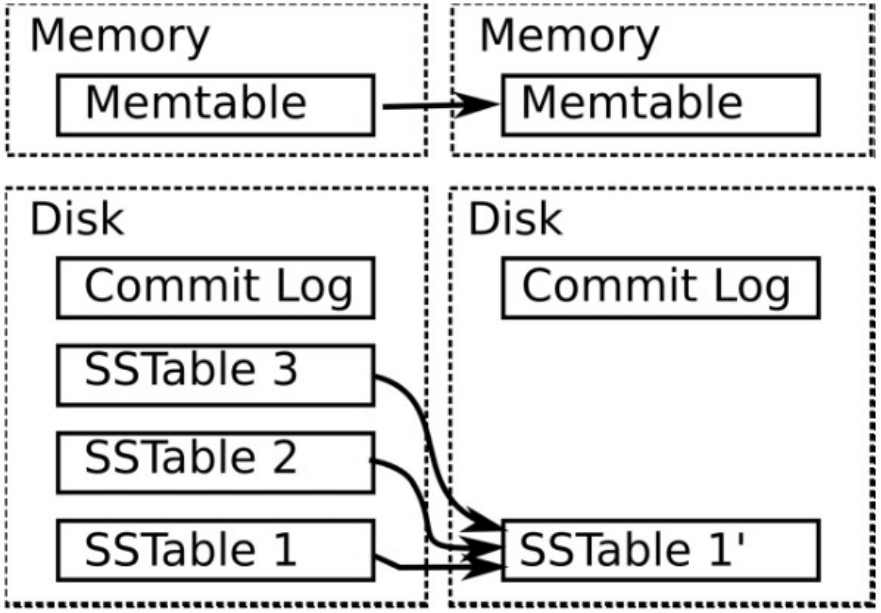

Write Path
Memory
Memtable
Disk
Commit Log
SSTable 3
SSTable 2
SSTable 1

LSM Memtable Flush



LSM Compaction

SSTables are immutable and get compacted over time by copying the compacted SSTable to another area of the disk while preserving the original SSTable and removing the latter after the compactation process has happened



Query Models

- Substantial differences in the querying capabilities the different NoSQL datastores offer
- Whereas key/value stores by design often only provide a lookup by primary key or some id field and lack capabilities to query any further fields, other datastores like the document databases CouchDB and MongoDB allow for complex queries
- This is not surprising as in the design of many NoSQL databases rich dynamic querying features have been omitted in favor of performance and scalability

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

Amazon's Dynamo

- The interface Dynamo provided to client applications consists of only two operations:
  - get(key), returning a list of objects and a context
  - put(key, context, object), with no return value
- The get-operation may return more than one object if there are version conflicts for objects stored under the given key.
- It also returns a context, in which system metadata such as the object version is stored, and clients have to provide this context object as a parameter in addition to key and object in the put operation.

Amazon's Dynamo Implementation

- Key and object values are not interpreted by Dynamo but handled as "an opaque array of bytes". The key is hashed by the MD5 algorithm to determine the storage nodes responsible for this key-/value-pair.
- To provide incremental scalability, Dynamo uses consistent hashing to dynamically partition data across the storage hosts that are present in the system at a given time
- To determine conflicting versions, perform syntactic reconciliation and support client application to resolve conflicting versions Dynamo uses the concept of vector clocks

Project Voldemort

- A key-/value-store initially developed for and still used at LinkedIn.
- API consisting of:
  - get(key), returning a value object
  - put(key, value)
  - delete(key)
- Both, keys and values can be complex, compound objects such as lists and maps

Project Voldemort

- The simple data structure and API of a key-value store does not provide complex querying capabilities: joins have to be implemented in client applications while constraints on foreign-keys are impossible; besides, no triggers and views may be set up.
- Project Voldemort allows namespaces for key-/value-pairs called “stores“, in which keys are unique.
- Each key is associated with exactly one value

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

Apache CouchDB

- The main abstraction and data structure in CouchDB is a document.
- Documents consist of named fields that have a key/name and a value.
- A fieldname has to be unique within a document and its assigned value may a string (of arbitrary length), number, Boolean, date, an ordered list or an associative map
- Documents may contain references to other documents (URIs, URLs) but these do not get checked or held consistent by the database
- A further limitation is that documents in CouchDB cannot be nested

Apache CouchDB

- A wiki article may be an example of such a document:

"Title" : "CouchDB",

"Last editor" : "172.5.123.91" ,

"Last modified": "9/23/2010" ,

"Categories": ["Database", "NoSQL", "Document Database"],

"Body": "CouchDB is a ...",

"Reviewed": false

Apache CouchDB

- Besides fields, documents may also have attachments
- CouchDB maintains some metadata such as a unique identifier and a revision number for each document
- The document id is a 128 bit value (so a CouchDB database can store 3.4×10^{38} different documents)
- The revision number is a 32 bit value determined by a hash-function

Apache CouchDB

- Documents do not correspond to a fixed schema (schema-free) but have some inner structure known to applications as well as the database itself.
- Compared to key-/value-stores data can be evaluated more sophisticatedly
- In the web application field there are a lot of document-oriented applications which CouchDB addresses as its data model fits this class of applications and the possibility to iteratively extend or change documents can be done with a lot less effort compared to a relational database

Apache CouchDB

- Each CouchDB database consists of exactly one flat/non-hierarchical namespace that contains all the documents which have a unique identifier (consisting of a document id and a revision number aka sequence id)
- CouchDBs way to query, present, aggregate and report the semi-structured document data are views
- A typical example for views is to separate different types of documents (such as blog posts, comments, authors in a blog system) which are not distinguished by the database itself as all of them are just documents to it

Apache CouchDB

- Views are defined by JavaScript functions which neither change nor save or cache the underlying documents but only present them to the requesting user or client application.
- Therefore documents as well as views (which are in fact special documents, called *design-documents*) can be replicated and views do not interfere with replication.

Views

- Views and are calculated on demand.
- There is no limitation regarding the number of views in one database or the number of representations of documents by views
- The JavaScript functions defining a view are called map and reduce which have similar responsibilities as in Google’s MapReduce approach

Map function

- The map function gets a document as a parameter, can do any calculation and may emit arbitrary data for it if it matches the view's criteria; if the given document does not match these criteria the map function emits nothing.
- Examples of emitted data for a document are the document itself, extracts from it, references to or contents of other documents (e.g. semantically related ones like the comments of a user in a forum, blog or wiki).

Map and reduce functions

- The data structure emitted by the map function is a triple consisting of the document id, a key and a value which can be chosen by the map function.
- After the map function has been executed its results get passed to an optional reduce function which can do some aggregation on the view
- As all documents of the database are processed by a view's functions this can be time consuming and resource intensive for large databases
- Therefore a view is not created and indexed when write operations occur but on demand (at the first request directed to it) and updated incrementally when it is requested again

Apache CouchDB

- CouchDB databases are addressed via a RESTful HTTP interface that allows to read and update documents
- The CouchDB project also provides libraries providing convenient access from a number of programming languages as well as a web administration interface

Interface

- CouchDB documents are requested by their URL according to the RESTful HTTP paradigm (read via HTTP GET, created and updated via HTTP PUT and deleted via HTTP DELETE method).
- A read operation has to go before an update to a document as for the update operation the revision number of the document that has been read and should be updated has to be provided as a parameter.
- To retrieve document urls—and maybe already their data needed in an application—views can be requested by client applications (via HTTP GET).

MongoDB

- MongoDBs name is derived from the adjective *humongous*
- It is a schema-free document database
- MongoDB databases reside on a MongoDB server that can host more than one of such databases which are independent and stored separately by the MongoDB server.
- A database contains one or more collections consisting of documents.
- In order to control access to the database a set of security credentials may be defined for databases

MongoDB

- The documents within a collection may be heterogeneous although the MongoDB manual suggests to create “one database collection for each of your top level objects”
- Once the first document is inserted into a database, a collection is created automatically and the inserted document is added to this collection
- JavaScript is used by the interactive MongoDB shell

Collections

- Collections may also be created explicitly by the createCollection-command:

db. createCollection (<name >, {< configuration parameters >})

Documents

- The abstraction and unit of data storable in MongoDB is a document, a data structure comparable to an XML document, a Python dictionary, a Ruby hash or a JSON document.
- In fact, MongoDB persists documents by a format called BSON which is very similar to JSON but in a binary representation for reasons of efficiency and because of additional datatypes compared to JSON
- Documents in MongoDB are limited in size by 4 megabytes

Documents

- As an example, a document representing a wiki article may look like the following in JSON notation:

```
{
title: "MongoDB",
last_editor: "172.5.123.91" ,
last_modified: new Date ("9/23/2010") ,
body: "MongoDB is a...",
categories: [" Database", "NoSQL", "Document Database "] ,
reviewed: false
}
```

Documents

- To add such a document into a MongoDB collection the insert function is used:

db.<collection >. insert({ title: "MongoDB", last_editor : ... });

- Once a document is inserted it can be retrieved by matching queries issued by the find operation and updated via the save operation:

db.<collection >. find({ categories: ["NoSQL", "Document Databases"] });

db.<collection >. save({ ... });

MongoDB

- MongoDB does not provide a foreign key mechanism so that references between documents have to be resolved by additional queries issued from client applications.
- References may be set manually by assigning some reference field the value of the _id field of the referenced document
- The MongoDB points out that although references between documents are possible there is the alternative to nest documents within documents. The embedding of documents is “much more efficient” according to the MongoDB manual as “[data] is then colocated on disk”.

Queries

- **Selection** Queries in MongoDB are specified as *query objects*, BSON documents containing selection criteria, and passed as a parameter to the find operation which is executed on the collection to be queried

queried db.<collection >. find({ title: "MongoDB");

- The selection criteria given to the find operation can be seen as an equivalent to the WHERE clause in SQL statements

Selection

- In the selection criteria passed to the find operation a lot of operators are allowed—besides equality comparisons as in the example above.
- These have the following general form:

<fieldname >: {$<operator >: <value >}

<fieldname >: {$<operator >: <value >, $<operator >: value} // AND -junction

Selection

- Examples of allowed operators
- Non-equality: $ne
- Numerical Relations: $gt, $gte, $lt, $lte (representing >, ≥, <, ≤)
- Equality-comparison to (at least) one element of an array: $in with an array of values as comparison operand, e.g.

{ categories: {$in: ["NoSQL", "Document Databases"]} }

…

Projection

- A second parameter can be given to the find operation to limit the fields that shall be retrieved—analogous to the projection clause of a SQL statement
- These fields are again specified by a BSON object consisting of their names assigned to the value 1:

db.<collection >. find({<selection criteria >}, {<field_1 >:1, ...});

Result Processing

- The results of the find operation may be processed further by arranging them using the sort operation, restricting the number of results by the limit operation and ignoring the first n results by the skip operation:

db.<collection >. find(...).sort({<field >: <1| -1 >}).limit(<number >).skip(<number >);

Inserts

- Documents are inserted into a MongoDB collection by executing the insert operation which simply takes the document to insert as an argument:

db.<collection >. insert(<document >);

- MongoDB appends the primary key field _id to the document passed to insert.
- Alternatively, documents may also be inserted into a collection using the save operation:

db.<collection >. save(<document >);

- The save operation comprises inserts as well as updates: if the _id field is not present in the document given to save it will be inserted; otherwise it updates the document with that _id value in the collection

Updates

- The save operation can be used to update documents.
- However, there is also an explicit update operation with additional parameters and the following syntax:

db.<collection >. update(<criteria >, <new document >);

Deletes

- To delete documents from a collection, the remove operation has to be used which takes a document containing selection criteria as a parameter:

db.<collection >. remove({ <criteria > });

- Selection criteria has to be specified in the same manner as for the find operation

The eval-operation

- To execute arbitrary blocks of code locally on a database server, the code has to be enclosed by an anonymous JavaScript function and passed to MongoDB's generic eval operation:

db.eval(function(<formal parameters >) { ... }, <actual parameters >);

Implementation

- MongoDB supports horizontal scaling via an automatic sharding architecture to distribute data across “thousands of nodes” with automatic balancing of load and data as well as automatic failover
- MongoDB uses read/write locks for many operations with any number of concurrent read operations allowed, but typically only one write operation. The acquisition of write locks is greedy and, if pending, prevents subsequent read lock acquisitions
- Replication is asynchronous

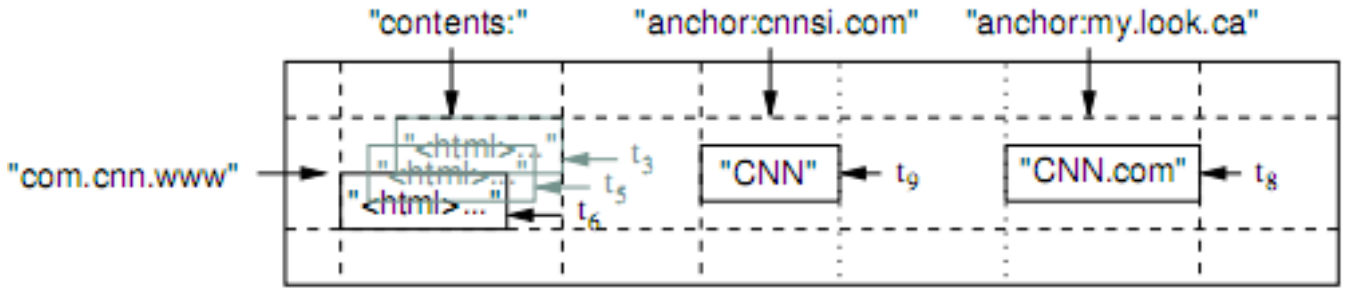
Column-Oriented Databases

- 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 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

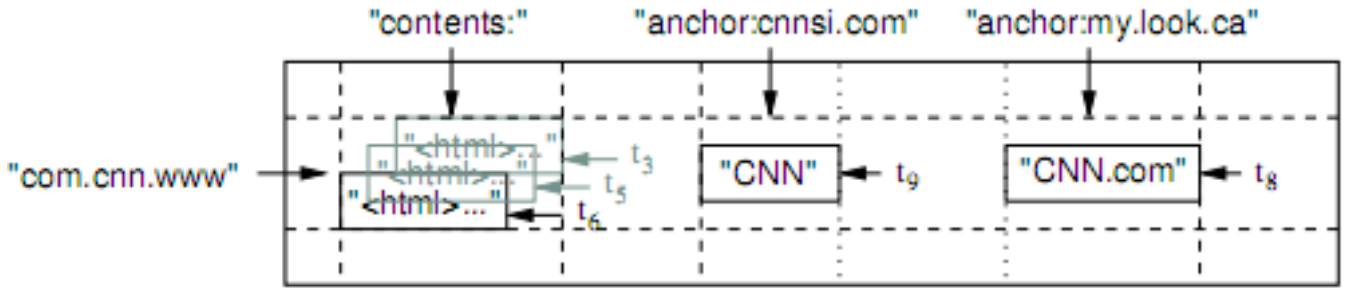
Google's Bigtable

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

Rows

- Every value also has an associated timestamp (*t*3, *t*5, *t*6 for the page contents, *t*9 for the link text from *CNN Sports Illustrated*, *t*8 for the link text from *MY-look*).
- **Row** keys in Bigtable are strings of up to 64KB size.
- Rows are kept in lexicographic order and are dynamically partitioned by the datastore into so called ***tablets***, “the the unit of distribution and load balancing” in Bigtable.
- Client applications can exploit these properties by wisely choosing row keys as the ordering of row-keys directly influences the partitioning of rows into tablets

Rows

- Row ranges with a small lexicographic distance are probably split into only a few tablets, so that read operations will have only a small number of servers delivering these tablets
- In the example the domain names used as row keys are stored hierarchically descending (from a DNS point of view), so that subdomains have a smaller lexicographic distance than if the domain names were stored reversely (e.g. com.cnn.blogs, com.cnn.www in contrast to blogs.cnn.com, www.cnn.com).

Columns

- The number of **columns** per table is not limited.
- Columns are grouped by their key prefix into sets called *column families*.
- Column families are an important concept in Bigtable as they have specific properties and implications
  - They “form the basic unit of access control”,
  - They are expected to store the same or a similar type of data.
  - Their data gets compressed together by Bigtable.
  - They have to be specified before data can be stored into a column contained in a column family.

Columns

- The example shows two column families: content and anchor.
- The content column family consists of only one column whose name does not have to be qualified further.
- In contrast, the anchor column family contains two columns qualified by the domain name of the referring site.

Timestamps

- **Timestamps**, represented as 64-bit integers, are used in Bigtable to discriminate different reversion of a cell value.
- The value of a timestamp is either assigned by the datastore (i.e. the actual timestamp of saving the cell value) or chosen by client applications (and required to be unique).
- Bigtable orders the cell values in decreasing order of their timestamp value “so that the most recent version can be read first”.
- In order to disburden client applications from deleting old or irrelevant revisions of cell values, an automatic garbage-collection is provided

API

- **Read Operations** include the lookup and selection of rows by their key, the limitation of column families as well as timestamps (comparable to projections in relational databases) as well as iterators for columns.
- **Write Operations for Rows** cover the creation, update and deletion of values for a column of the particular row.
- **Write Operations for Tables and Column Families** include their creation and deletion.
- **Administrative Operations** allow to change “cluster, table, and column family metadata, such as access control rights”.

API

- **Server-Side Code Execution** is provided for scripts written in Google's data processing language Sawzall
- **MapReduce Operations** may use contents of Bigtable maps as their input source as well as output target.
- Transactions are provided on a single-row basis: "Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row"