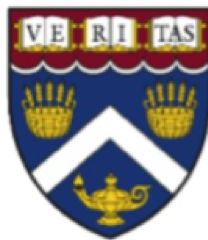


CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova



Lecture 6 NoSQL DBs

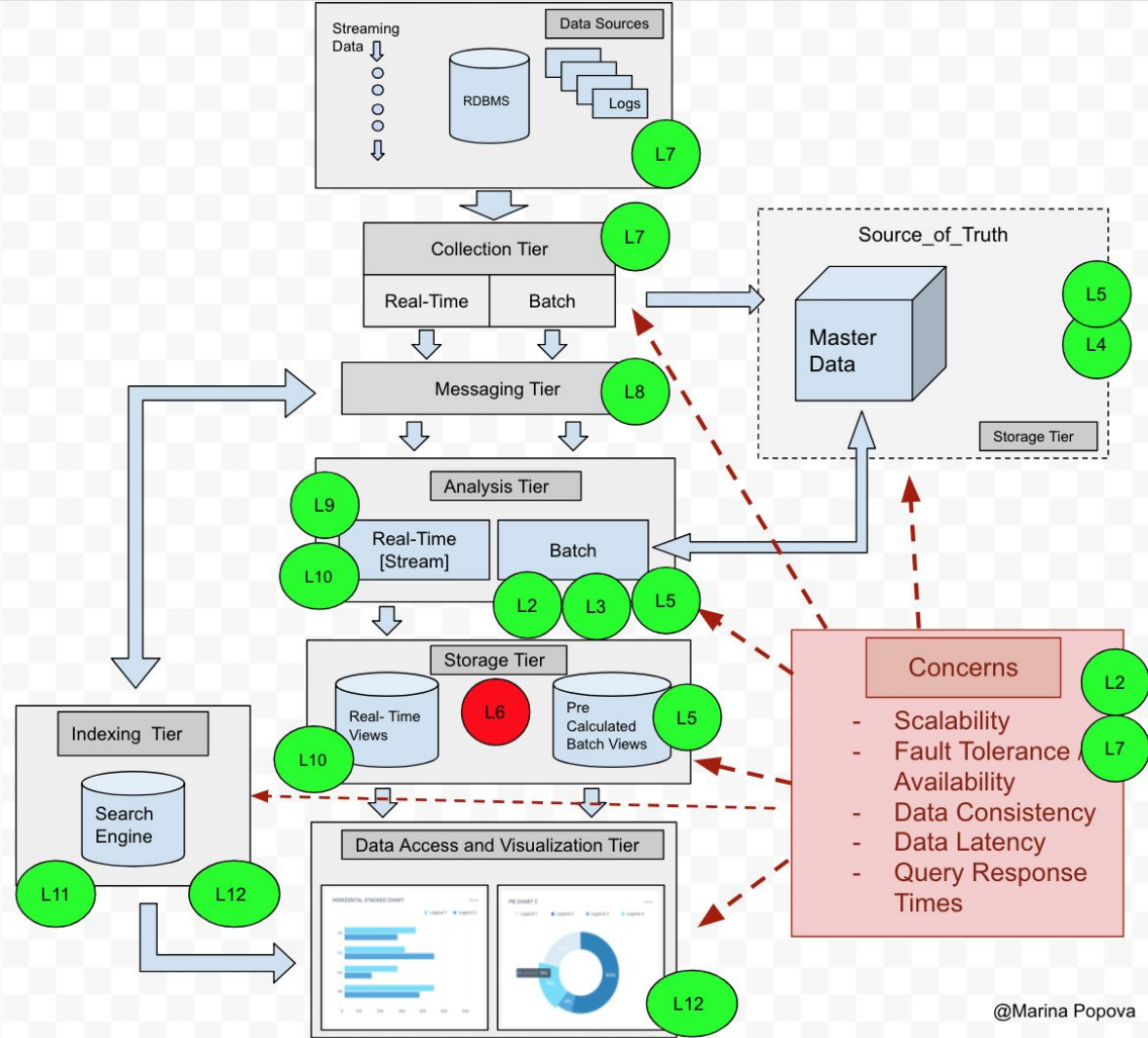
@Marina Popova

Agenda

Big Data Storage

- a more holistic view on data storage scaling
- NoSQL DB concepts and classification
- Batch Views storage options
- HBase as an option for Batch Views Storage

Where Are We?

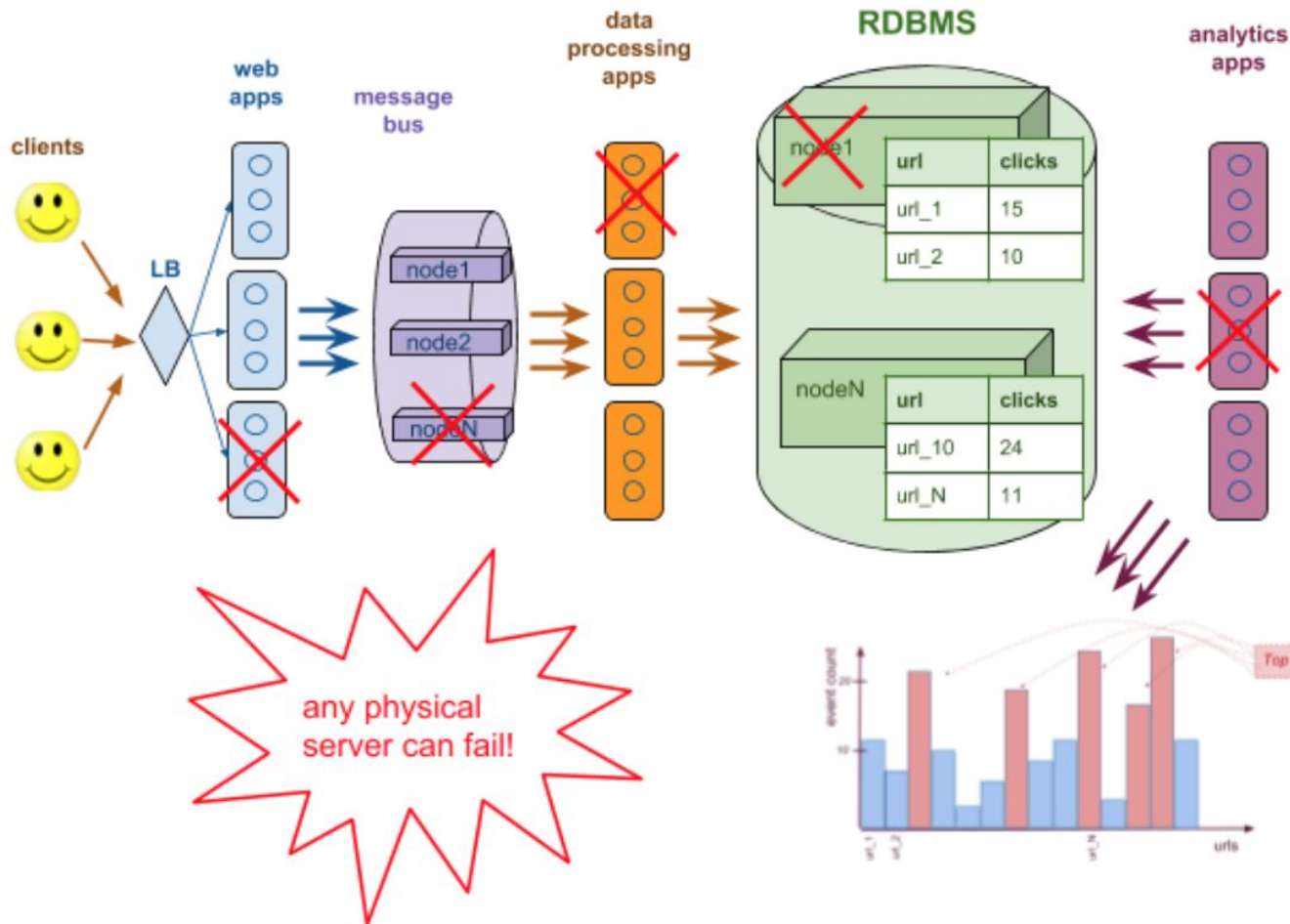


Scaling: recap

We've scaled the Data Processing at this point (MR !!)

We've identified critical issues when scaling Storage tier:

- single row contention
- indexes growing too large
- mutable objects are not handled well with data replication
- atomic multi-step operations cause performance problems



Scaling Data Storage

why cannot I use my beloved RDBMS, again ??

Traditional relational database management systems (RDBMSs) provide powerful mechanisms to store and query structured data under **strong consistency and transaction guarantees** and have reached an unmatched level of reliability, stability and support through decades of development.

They provide these capabilities by implementing and guaranteeing a specific set of properties when handling transactions.



Scaling Data Storage

Why are RDBMS systems getting slow under huge load?
What causes the single row contention issues?

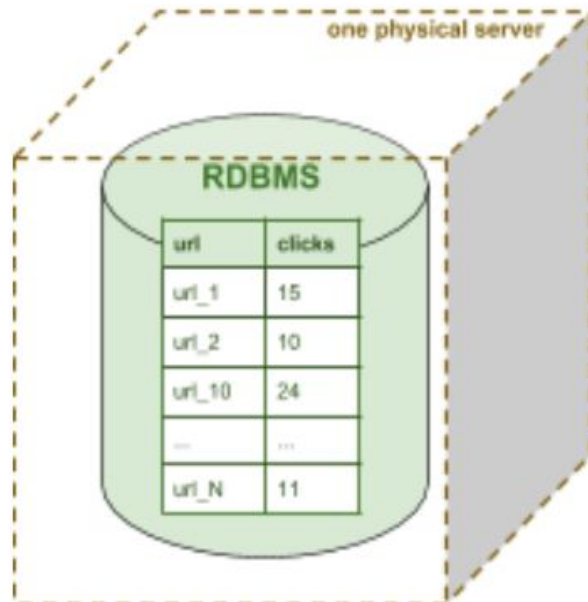
it's ACID !!

Transaction is a single logical unit of work which accesses and possibly modifies the contents of a database.

Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called **ACID** properties.

- **Atomicity** - entire transaction takes place at once or doesn't happen at all - "All Or Nothing Rule"
- **Consistency** - any defined checks and constraints are satisfied after any transaction
- **Isolation** - events within a transaction must be hidden from other transactions running concurrently
- **Durability** - once TX finished, it's results now become permanent and are stored in a non-volatile memory



Scaling Data Storage - recap

How did we attempt to solve the issue with slow DB reads/writes ?

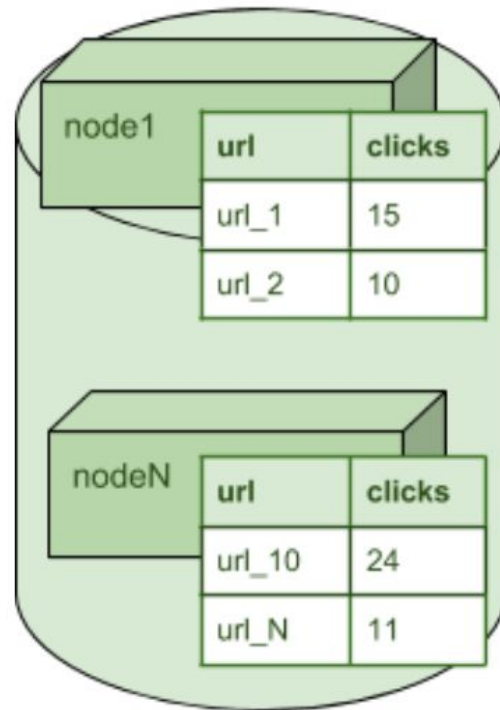
By **distributing our DB across multiple physical servers**

- This is called **horizontal sharding**
- this requires a new approach to storing data: rows of tables have to be split across servers based on their keys (PKs)
- have to compute **hash code** of a key based on the number of shard[servers] available in the cluster
- Your DB has to be smart enough to know which key is on which server - this is a very expensive functionality

What about queries?

- get count per URL?
- get Top 5 URLs?

RDBMS



Scaling Data Storage

Just sharding data is not enough - why?

What's the solution?

Same core techniques as in Data Processing:

- Data partitioning
- Data replication

! Same core requirements have to be met: !

- **Scalability**
- **Fault Tolerance**
- **Data Consistency**
- **Data Latency**
- **Query Performance**

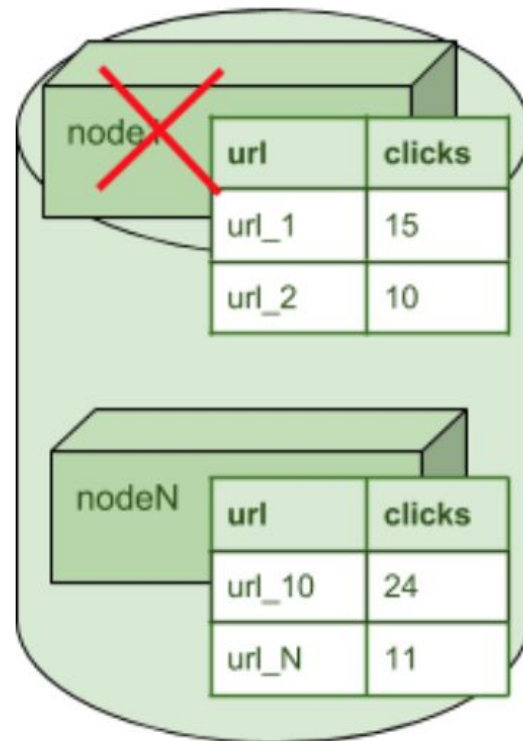


as well as ACID ...

- **Atomicity**
- **Consistency**
- **Isolation**
- **Durability**

... can this truly be done on multi-node DBs?

RDBMS



Big Data Scaling → NoSQL

This is when it is usually time to move to NoSQL Storage solutions! **why??**

- horizontally scaling means that: the same ACID properties would have to be guaranteed when potentially **many servers are involved in one transaction.**
- however, any server can die or become separated due to network connectivity issues == "**network partitions**"
- traditional ACID systems cannot operate in the same ACID mode under those conditions anymore
- .. but a different type of systems, called NoSQL (Not Only SQL), can, although with different properties:

Most NoSQL systems offer **horizontal scalability and higher availability than relational databases by sacrificing querying capabilities and consistency guarantees.**

If ACID properties cannot be guaranteed by NoSQL solutions -
then what properties **can** they have?

NoSQL Classification

Meet **CAP properties** - a set of properties used to describe different types of availability and consistency behaviors of distributed systems:

Consistency (C): Reads and writes are always executed atomically and are strictly consistent (linearizable). Put differently, all clients have the same view of the data at all times

Availability (A): Every non-failing node in the system can always accept read and write requests by clients and will eventually return with a correct response (not with an error message)

Partition-tolerance (P): The system upholds the previously displayed consistency guarantees and availability in the presence of message loss between the nodes or partial system failure.

However, no single system can have all CAP properties ... Why??

NoSQL Classification - CAP Theorem Classes

Meet **The CAP Theorem**, presented by Eric Brewer at PODC 2000 and later proven by Gilbert and Lynch

It is one of the truly influential impossibility results in the field of distributed computing, because it places an ultimate upper bound on what can possibly be accomplished by a distributed system.

The CAP Theorem states that a **sequentially consistent read/write register that eventually responds to every request cannot be realised in an asynchronous system that is prone to network partitions.**

In other words, it can guarantee at most two of the three CAP properties at the same time

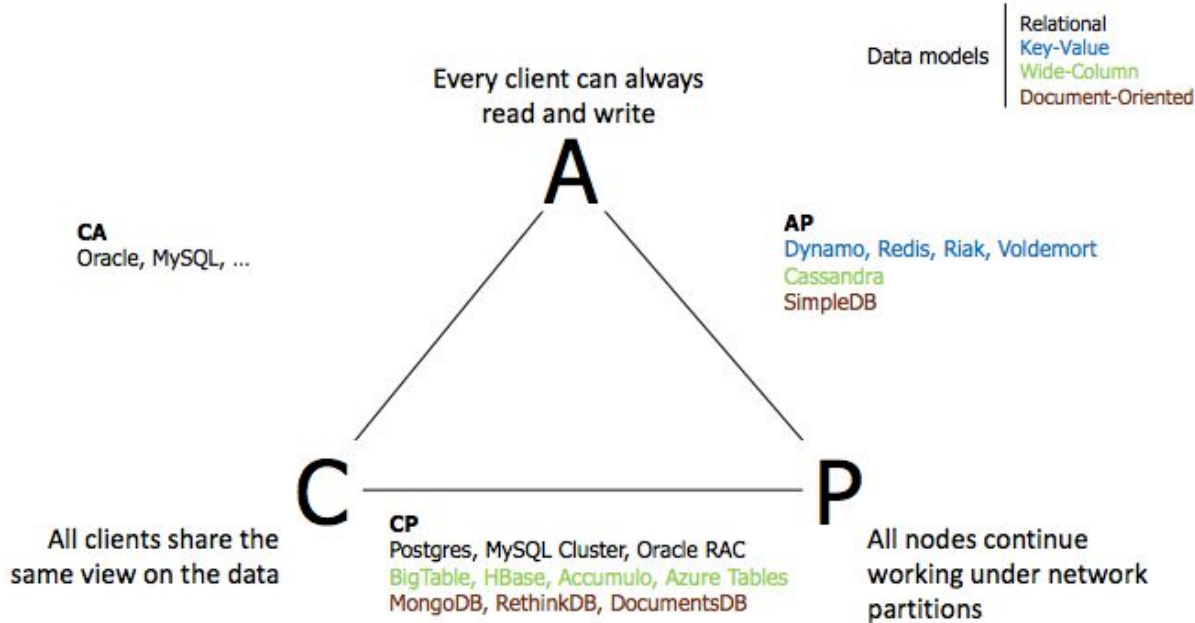
NoSQL Classification

Brewer states that a system can be **both available and consistent** in normal operation, but in the presence of a network partition, this is **not possible**:

- If the system continues to work in spite of the partition, there is some non-failing node that has lost contact to the other nodes and thus has to decide to either continue processing client requests to preserve availability (AP, eventual consistent systems) or to reject client requests in order to uphold consistency guarantees (CP)
- The first option violates consistency, because it might lead to stale reads and conflicting writes, while the second option obviously sacrifices availability.

There are also systems that usually are available and consistent, but fail completely when there is a partition (CA), for example single-node systems - like traditional RDBMS systems

NoSQL Triangle



- every distributed database system has to make this **A-vs-C trade-off**
- the huge number of different NoSQL systems shows that there is a wide spectrum between the two paradigms



Nathan Hurst: Visual Guide to NoSQL Systems
<http://blog.nahurst.com/visual-guide-to-nosql-systems>

NoSQL Classification

However:

CAP Theorem **does not state anything on normal operation**; it merely tells us whether a system favors availability or consistency **in the face of a network partition**.

PACELC theorem (by Daniel Abadi): addresses the trade-off between latency and consistency during normal operations

In case of a **P**artition, there is an **A**vailability-**C**onsistency trade-off;
Else, i.e. in normal operation, there is a **L**atency-**C**onsistency trade-off

This classification offers two possible choices for the partition scenario (A/C) and also two for normal operation (L/C)

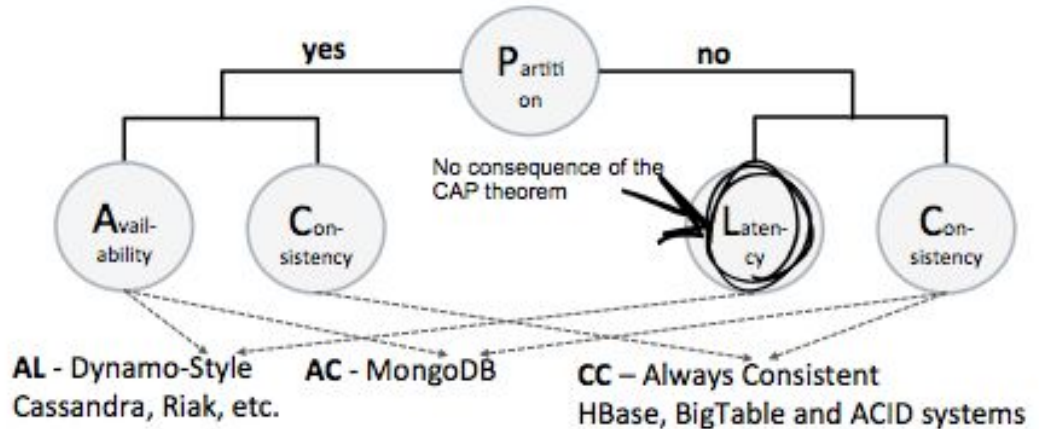
NoSQL

The two theorems, CAP and PACELC, can be used to categorize database systems by their position in the Availability-vs-Consistency and Latency-vs-Consistency spectrum

some systems cannot be assigned exclusively to one single PACELC class

PACELC – an alternative CAP formulation

- ▶ Idea: Classify systems according to their behavior during *network partitions*



Abadi, Daniel. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story."

NoSQL - CAP classification

Is CAP or PACELC - based classification perfect? NO

<https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html>

- CAP uses very narrow definitions - often interpreted loosely
 - "Consistency" - is really a linearizability, applied to a single read/write register
 - *"If operation B started after operation A successfully completed, then operation B must see the the system in the same state as it was on completion of operation A, or a newer state."*
 - it does NOT covers multi-object/key transactions
 - "Availability" in CAP is defined as *"every request received by a non-failing [database] node in the system must result in a [non-error] response"*. It's not sufficient for *some* node to be able to handle the request: *any* non-failing node needs to be able to handle it. Many so-called "highly available" (i.e. low downtime) systems actually do not meet this definition of availability.

NoSQL - CAP classification

- Many systems are neither AP nor CP - or can be either via configuration

*For example, take any replicated database with a **single leader**, which is the standard way of setting up replication in most relational databases. In this configuration, if a client is partitioned from the leader, it cannot write to the database. Even though it may be able to read from a follower (a read-only replica), the fact that it cannot write means any single-leader setup is **not CAP-available**.*

*If you allow the application to make **reads from a follower**, and the replication is asynchronous (the default in most databases), then a follower may be a little behind the leader when you read from it. In this case, your reads will not be linearizable, i.e. **not CAP-consistent**.*

NoSQL - CAP classification

So, why bother then? Is it totally useless? NO



- CAP/PACELC-based analyses shows the behaviour with:
 - default settings OR
 - most commonly used settings/ usage patterns
- Metrics (Latency/Availability/Consistency) are extremely important to understand
 - for your specific use cases
 - for corner cases
- The same trade-offs have to be made - no matter what; it makes one be aware of the choices that have to be made

NoSQL Classification: by Data Models and Types

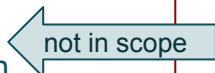
Distributed Storage systems can be classified by:

- ✓ CAP categories
- data models and storage types

Data Model of NoSQL systems determines the way they store and allow access to data

Conceptual Types:

- ✓ File-based
 - ✓ S3, HDFS, Google Cloud Storage
- **Key-Value**
 - **Key: Document**
 - **Key: Wide-Column**
- Graph DBs
 - Neo4J, JanusGraph, Giraph
- Document/Lucene-based - Search Engines
 - ES, Solr, Splunk



Hybrids:

- NoSQL - SQL: hybrid between NoSQL and Lucene-based
 - Kudu, CockroachDB, BigQuery, Presto, Impala
- In-memory DBs
 - Aerospike, VoltDB, Redis
- Time Series DBs
 - InfluxDB, Prometheus, TimescaleDB
- many more specialized ones ...

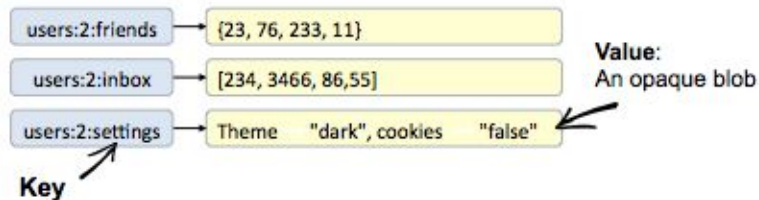
NoSQL - Data Models - Key-Value Stores

Key-Value Stores - consists of a set of key-value pairs with unique keys

- **schemaless:**
 - no knowledge of the stored data structure - no data definition language (no schema-on-write)
 - application logic is responsible for data interpretation (schema-on-read)
- ==> only supports **get** and **put** operations (simple CRUD)

Key-Value Stores

- Data model: (key) -> value
- Interface: CRUD (Create, Read, Update, Delete)



- Examples: Amazon Dynamo (AP), Riak (AP), Redis (CP)

Advantages: easy to partition and query the data ⇒ database system can achieve low latency as well as high throughput

Disadvantages: complex operations, e.g. range queries, are not efficient (or not supported at all)

NoSQL Data Models: Document Store

Document Store is a key-value store that **restricts values to semi-structured formats** such as JSON documents

- values can have nested structure
- can have secondary indices

Advantages:

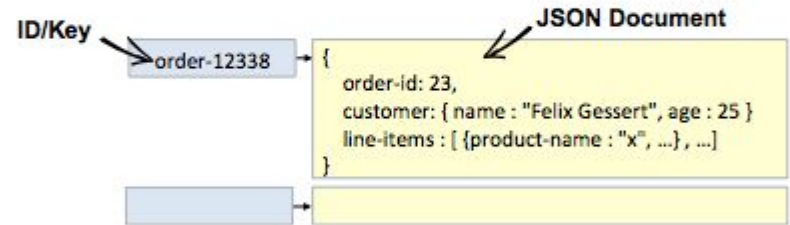
- greater flexibility in accessing the data:
 - can retrieve only parts of a document
 - can execute queries like aggregation, query-by-example or even full-text search

Disadvantages

- same: complex queries are not efficient

Document Stores

- ▶ Data model: (collection, key) -> document
- ▶ Interface: CRUD, Querys, Map-Reduce



- ▶ Examples: CouchDB (AP), RethinkDB (CP), MongoDB (CP)

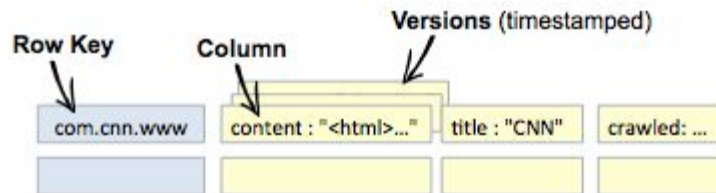
NoSQL Data Models: Wide-Column Stores

Wide-Column Store: analogous to a **distributed multi-level sorted map**

- the first-level keys identify rows which themselves consist of key-value pairs.
- the first-level keys are called **row keys**, the second-level keys are called **column keys**
- each cell is timestamped (has a version)
- this design enables support for **arbitrarily many columns** - null values do not take any space
- the set of all columns is partitioned into **column families** to co-locate columns on disk

Wide-Column Stores

- ▶ Data model: (rowkey, column, timestamp) -> value
- ▶ Interface: CRUD, Scan



- ▶ Examples: Cassandra (AP), Google BigTable (CP), HBase (CP)

NoSQL Data Models: Wide-Column Stores

Wide-Column Stores - continue

- On disk, ALL data from each row is NOT co-located:
 - instead, **values of the same column family** and from the same row are co-located
 - ==> a single row **cannot be retrieved by one single lookup** as in a document store, but has to be joined together from the columns of all column families
- this enables **highly efficient data compression** and makes retrieving only a portion of an entity very efficient
- rows are further stored sorted (lexicographic order) by their keys - so that data that are accessed together are physically co-located, enabling **efficient range queries**
- all rows are distributed into contiguous ranges (so-called tablets) among different tablet servers ==>
 - **row scans** only involve few servers and thus **are very efficient**

We will dive into more details of this type of storage later on with Goggle BigTable and HBase

NoSQL Classification

We now understand how diverse database systems can be

Every significantly successful database is designed for a particular class of applications, or to achieve a specific combination of desirable system properties.

The simple reason why there are so many different database systems is that it is **not possible for any system to achieve all desirable properties at once**.

Traditional SQL databases such as PostgreSQL have been built to provide the **full functional package**: a very flexible data model, sophisticated querying capabilities including joins, global integrity constraints and transactional guarantees.

On the other end of the design spectrum, there are key-value stores like Dynamo that scale with data and request volume and offer high read and write throughput as well as low latency, but **barely any functionality apart from simple lookups**.

how to determine which of the databases is good for some specific use case?

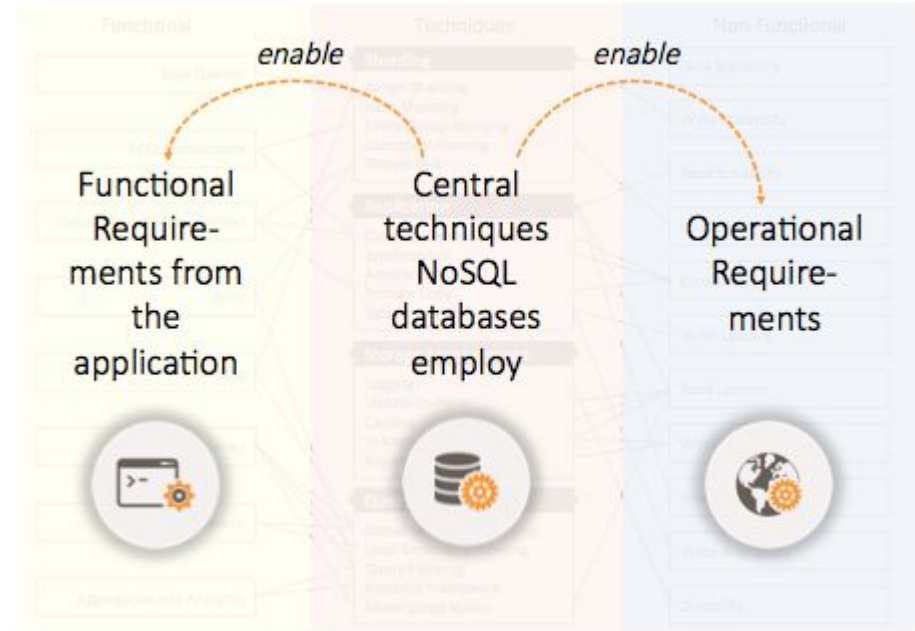
NoSQL Classification

The research by Felix Gessert and his team proposes an approach and a decision-supporting tool to do just that:

<https://medium.bagend.com/nosql-databases-a-survey-and-decision-guidance-ea7823a822d>

They have identified the most important distributed systems **implementation techniques** (such as sharding, replication, storage management and query processing) and how they are related to different **functional and non-functional properties** (goals) of data management systems.

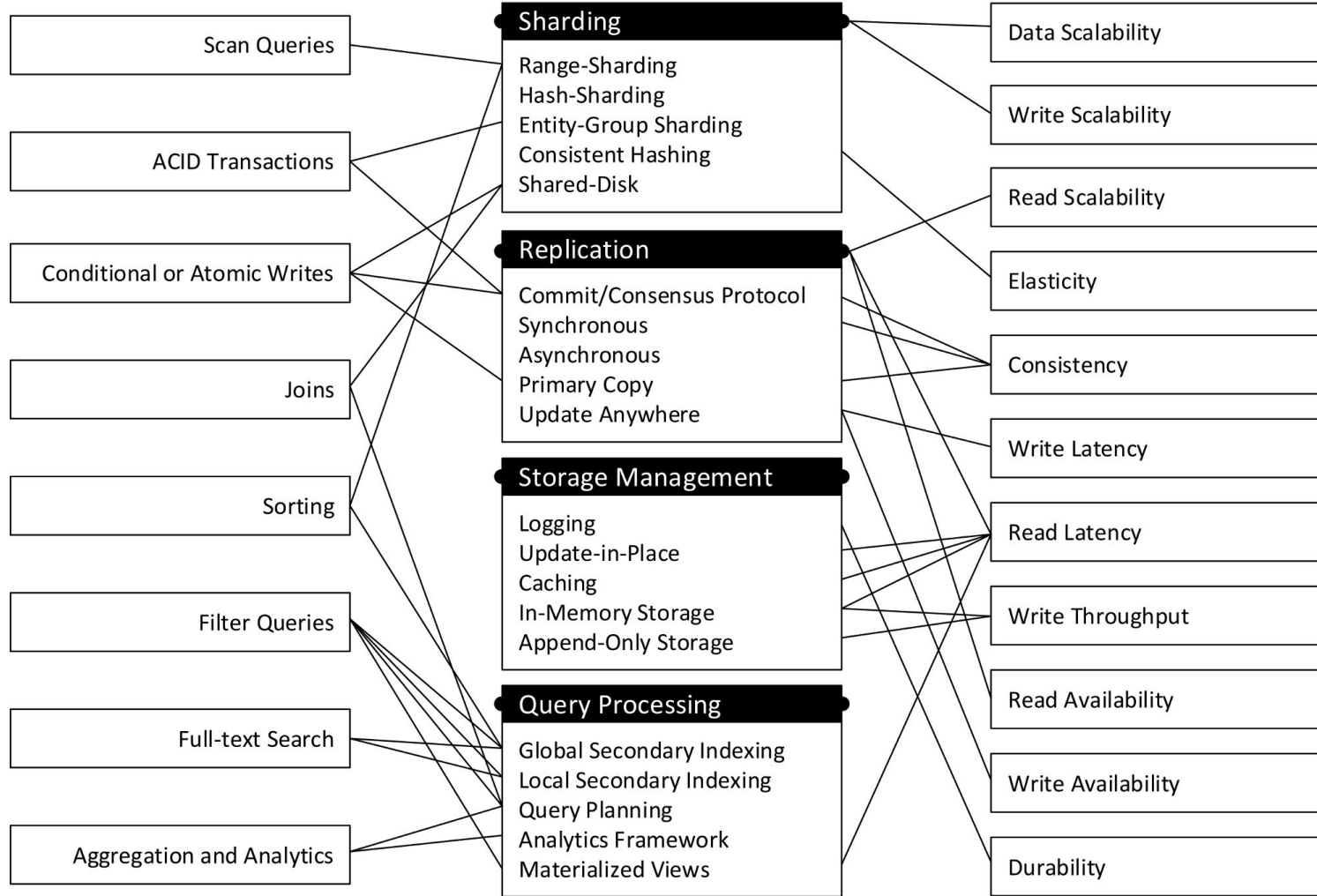
In order to illustrate what techniques are suitable to achieve which system properties, they also provide the NoSQL Toolbox (see below) where each technique is connected to the functional and non-functional properties it enables (positive edges only).



Functional

Techniques

Non-Functional



NoSQL Techniques: too overwhelmed? :)

This is a bit overwhelming :)

We will review a few groups of techniques in details
and leave others for self-study or later discussions

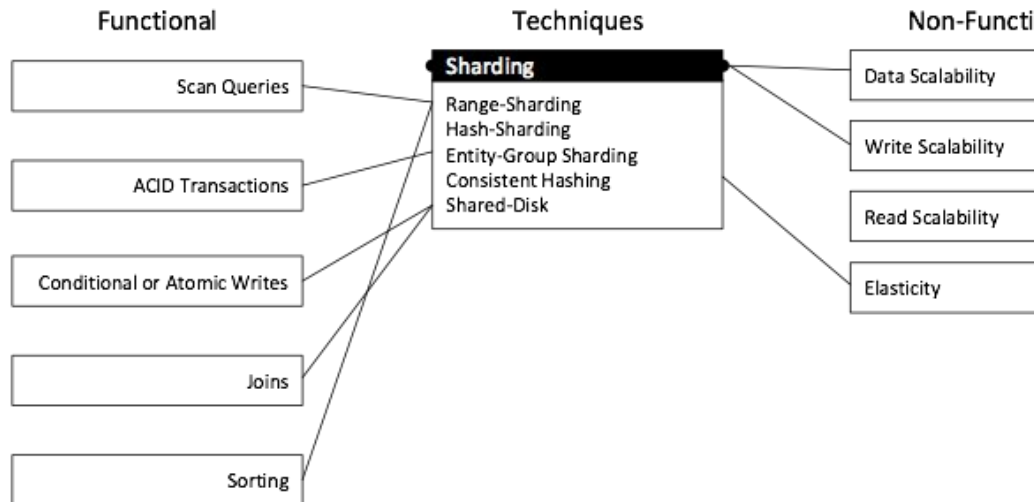


NoSQL Techniques: Sharding

Lets start with "sharding"

Several distributed relational database systems such as Oracle RAC or IBM DB2 pureScale rely on a **shared-disk architecture** where all database nodes access the same central data repository (e.g. a NAS or SAN).

Thus, these systems provide consistent data at all times, but are also inherently difficult to scale.



NoSQL Techniques: Sharding

In contrast, the (NoSQL) database systems we are considering here are built upon a **shared-nothing architecture**, meaning each system consists of many servers with private memory and private disks that are connected through a network.

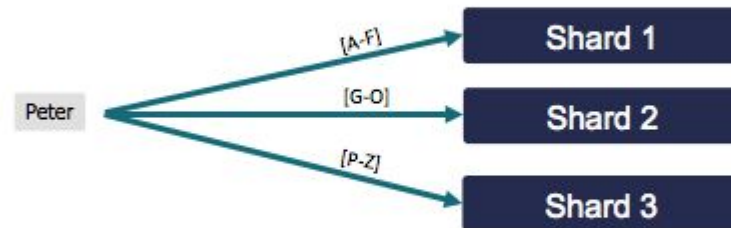
Thus, high scalability in throughput and data volume is achieved by sharding (partitioning) data across different nodes (shards) in the system.

Sharding is also one of the core techniques used by Distributed Search/Indexing systems, like ElasticSearch, Solr, Presto, etc.

Sharding (aka Partitioning, Fragmentation)

Scaling Storage and Throughput

- Horizontal distribution of data over nodes



- Partitioning strategies: Hash-based vs. Range-based
- Difficulty: Multi-Shard-Operations (join, aggregation)

NoSQL Techniques: Sharding

There are three basic distribution techniques: range-sharding, hash-sharding and entity-group sharding.

Range-sharding

- data is partitioned into ordered and contiguous value ranges
- ranges are defined over fields (shard keys) and assigned to partitions

Pros:

- efficient scans

Cons:

- repartitioning/balancing is required; why?

Implemented in

BigTable, HBase, DocumentDB
Hypertable, MongoDB,
RethinkDB, Espresso

It requires some coordination through a master that manages assignments. To ensure elasticity, the system has to be able to detect and resolve **hotspots** automatically by further splitting an overburdened shard.

NoSQL Techniques: Sharding

Hash-sharding

- every data item is assigned to a shard server according to some hash value built from the primary key
- does not require a coordinator and also guarantees the data to be evenly distributed across the shards, as long as the used hash function produces an even distribution

Pros:

- Even data distribution among servers
- Very scalable horizontally

Cons:

- only allows lookups; scans are unfeasible
- No data locality

Implemented in

MongoDB, Riak, Redis,
Cassandra, Azure Table,
Dynamo

NoSQL Techniques: Sharding

Hash-sharding - continue

- the shard server that is responsible for a record can be determined as $serverid = hash(id) \% servers$, for example.
- this hashing scheme requires all records to be reassigned every time a new server joins or leaves, because it changes with the number of shard servers (servers)
- this makes it impossible to use in elastic systems like **Dynamo**, **Riak** or **Cassandra**, which allow additional resources to be added on-demand and again be removed when not needed anymore.

For increased flexibility, elastic systems typically use **consistent hashing** where records are not directly assigned to servers, but instead to logical partitions which are then distributed across all shard servers.

This means that only a fraction of the data have to be reassigned upon changes in the system topology.

NoSQL Techniques: Sharding

Entity-group sharding

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

Pros:

- enables ACID transactions

Cons:

- Partitioning is not easily changeable

Implemented in

G-Store, MegaStore,
Relational Cloud, Cloud SQL
Server

NoSQL Techniques: Replication

Lets move to the next Technique - Replication

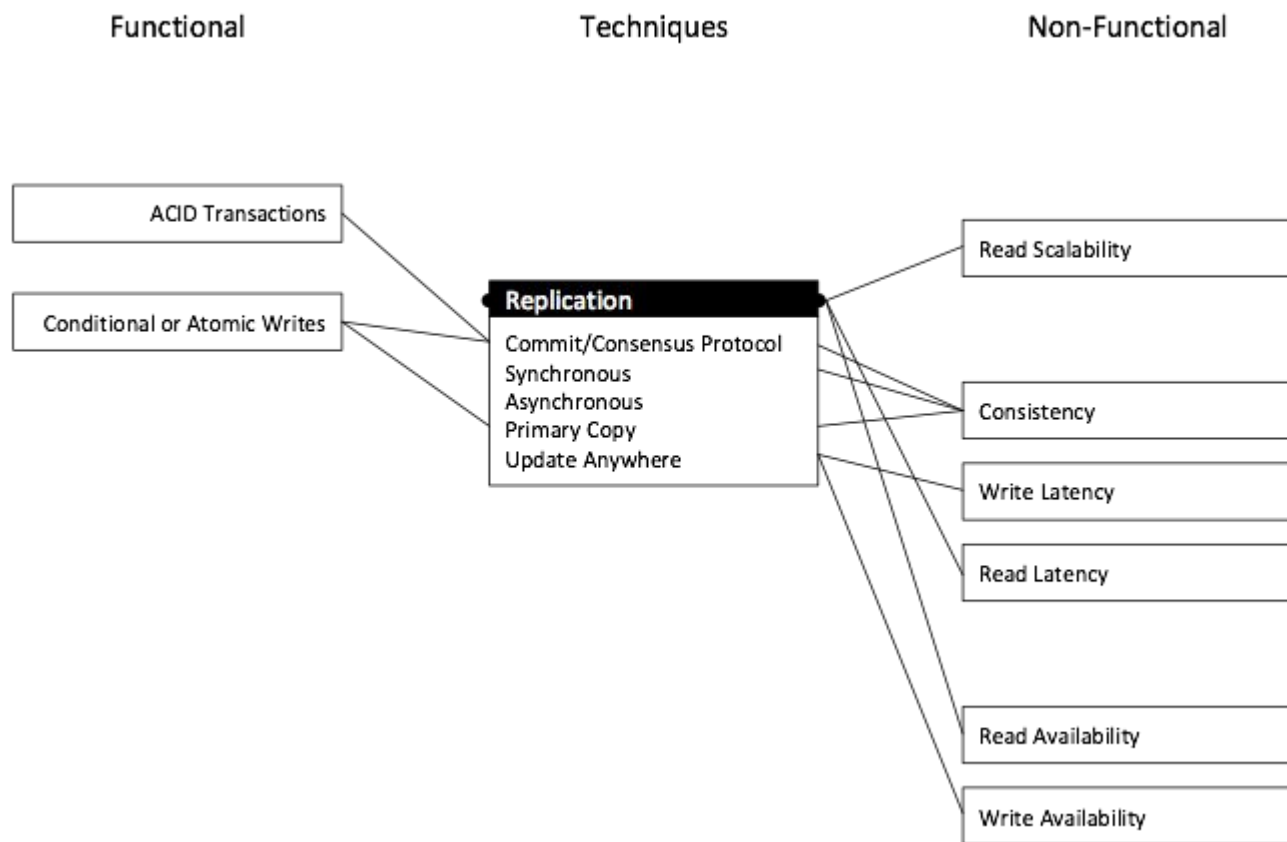
In terms of CAP, conventional RDBMSs are often CA systems run in single-server mode: The entire system becomes unavailable on machine failure. And so system operators secure data integrity and availability through expensive, but reliable high-end hardware.

In contrast, NoSQL systems like Dynamo, BigTable or Cassandra are designed for data and request volumes that cannot possibly be handled by one single machine, and therefore they run on clusters consisting of thousands of low-end servers. Thus, server failures are inevitable and frequent

Solution? **Replication**

But storing the same records on different machines (replica servers) in the cluster introduces the problem of synchronization between them and thus a **trade-off between consistency on the one hand and latency and availability on the other.**

NoSQL



NoSQL Techniques: Replication

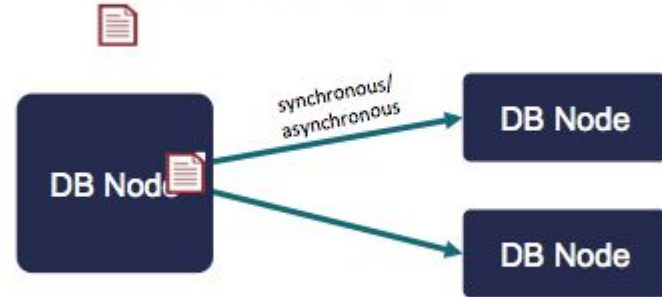
Gray et al. proposed a two-tier classification of different replication strategies according to:

- **Tier one: when** updates are propagated to replicas (consistency model)
- **Tier two: where** updates are accepted (coordination)


Replication

Read Scalability + Failure Tolerance

- › Stores N copies of each data item



- › Consistency model: synchronous vs asynchronous
- › Coordination: Multi-Master, Master-Slave

 Ōsu, M.T., Valduriez, P.: Principles of distributed database systems. Springer Science & Business Media (2011)

NoSQL Techniques: Replication

There are two possible consistency models on tier one (“when”)

- **eager (synchronous)** replication propagates incoming changes synchronously to all replicas before a commit can be returned to the client
 - Pros: consistency among replicas
 - Cons:
 - high write latency
 - needs a commit protocol
 - Impaired availability under some partitions
- **lazy (asynchronous)** replication applies changes only at the receiving replica and passes them on asynchronously.
 - Performed through log shipping or update propagation
 - Pros:
 - very fast writes, since it allows replicas to diverge
 - no controller needed
 - Cons: data may be served stale

@Marina Popova

Implemented in

BigTable, HBase, Accumulo,
CouchBase, MongoDB,
RethinkDB

Implemented in

Dynamo , Riak, CouchDB,
Redis, Cassandra, Voldemort,
MongoDB, RethinkDB

NoSQL Techniques: Replication

On the second tier (“where updates are accepted”) two different approaches/ **coordination protocols** are possible

- **master-slave (primary copy)** protocols:
 - changes can only be accepted by one replica (the master)
 - concurrency control is not more complex than in a distributed system without replicas, but the entire replica set becomes unavailable, as soon as the master fails
- **multi-master (update anywhere)** protocols:
 - every replica can accept writes
 - require complex mechanisms for prevention or detection and reconciliation of conflicting changes
 - Techniques typically used for these purposes are versioning, vector clocks, gossiping and read repair (e.g. in Dynamo) and convergent or commutative datatypes (e.g. in Riak).

NoSQL Techniques: Replication

all four combinations of the two-tier classification are possible

Distributed relational systems usually perform **eager master-slave** replication to maintain strong consistency.

Eager update anywhere replication suffers from a heavy communication overhead generated by synchronisation and can cause distributed deadlocks which are expensive to detect.

Example: Google's Megastore

NoSQL database systems typically rely on **lazy replication**:

- either in combination with the **master-slave** (CP systems, e.g. HBase and MongoDB)
- or the **update anywhere** approach (AP systems, e.g. Dynamo and Cassandra).

Many NoSQL systems leave the choice between latency and consistency to the client:

- i.e. for every request, the client decides whether to wait for a response from any replica to achieve minimal latency or for a certainly consistent response (by a majority of the replicas or the master) to prevent stale data

NoSQL Techniques: Data Storage

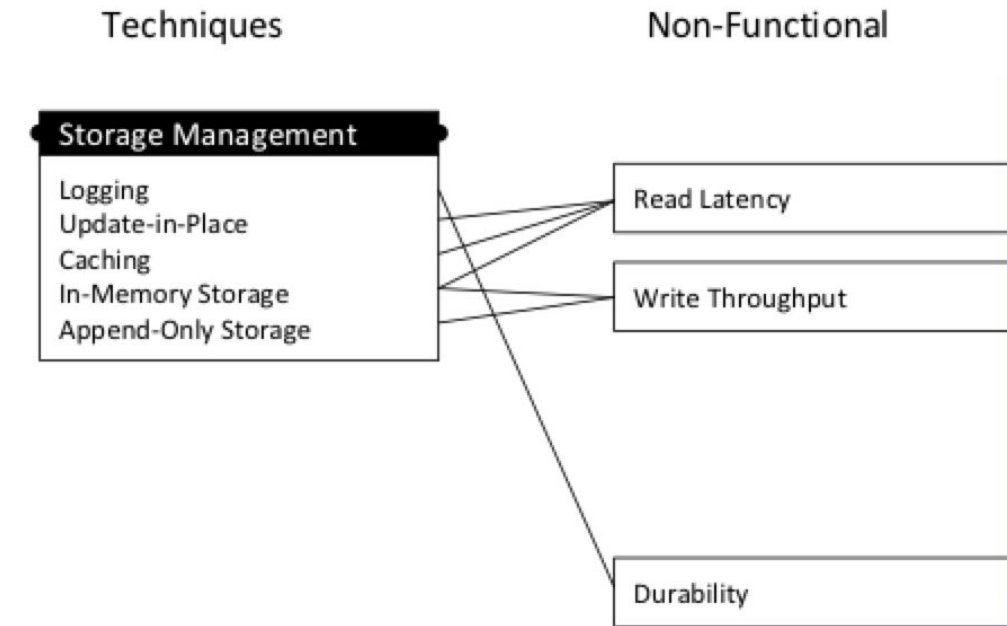
For best performance, database systems need to be optimized for the storage media they employ to serve and persist data.

These are typically:

- main memory (RAM)
- solid-state drives (SSDs)
- spinning disk drives (HDDs)

that can be used in any combination.

Unlike RDBMSs in enterprise setups, distributed NoSQL databases avoid specialized shared-disk architectures in favor of shared-nothing clusters based on commodity servers



NoSQL Techniques: Data Storage

Storage management can be categorized by two dimensions:

- **spatial dimension** (where to store data) - techniques of organizing data are:
 - Update-in-place (disk and memory)
 - update-only-IO (disk)
 - RAM (memory)
- **temporal dimension** (when to store data)
 - logging is a temporal technique that decouples main memory and persistent storage and thus provides control over when data is actually persisted

NoSQL Techniques: Data Storage

Data Storage options in a Nutshell

- large performance improvements can be expected if **RAM is used as primary storage (in-memory databases)**.
- The downside are high storage costs and lack of durability—a small power outage can destroy the database state.
- This can be solved in two ways: the state can be replicated over N in-memory server nodes protecting against N-1 single-node failures (e.g. HStore, VoltDB) or by **logging** to durable storage (e.g. Redis or SAP Hana)
- SSDs and HDDs are very different from RAM in terms of read and write performance:
 - **random writes [RW]** to an **SSD** are roughly an order of magnitude slower than **sequential writes [SW]**
 - **random reads [RR] on SSD**, on the other hand, can be performed without any performance penalties
 - There are some database systems (e.g. Oracle Exadata, Aerospike) that are explicitly engineered for these performance characteristics of SSDs.
 - In **HDDs**, both random reads and writes are 10–100 times slower than sequential access
 - ⇒ thus, **logging** (sequential writes), works great for both SSDs and HDDs since they both offer a significantly higher throughput for sequential writes

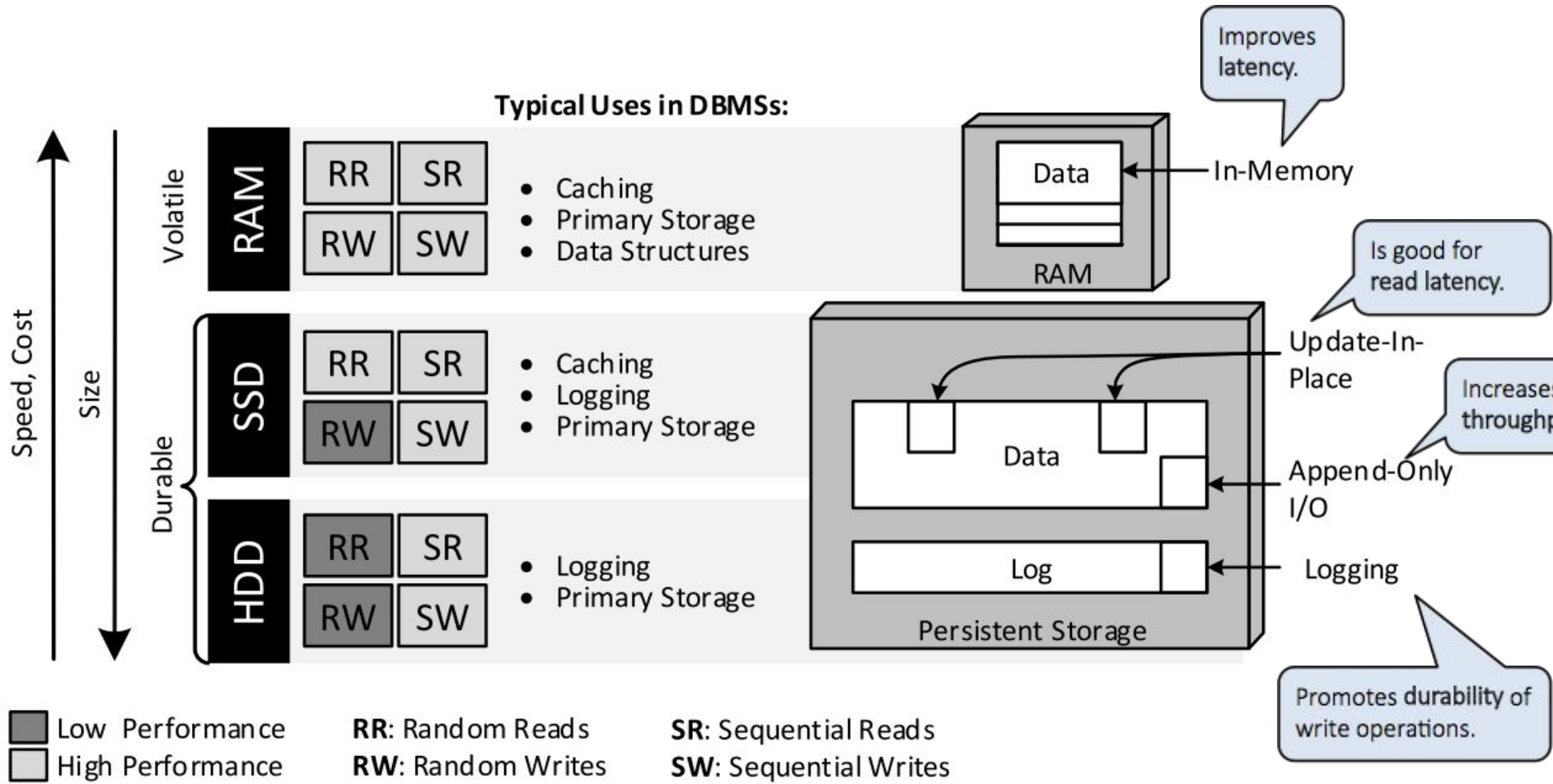
NoSQL Techniques: Data Storage

update-in-place access pattern:

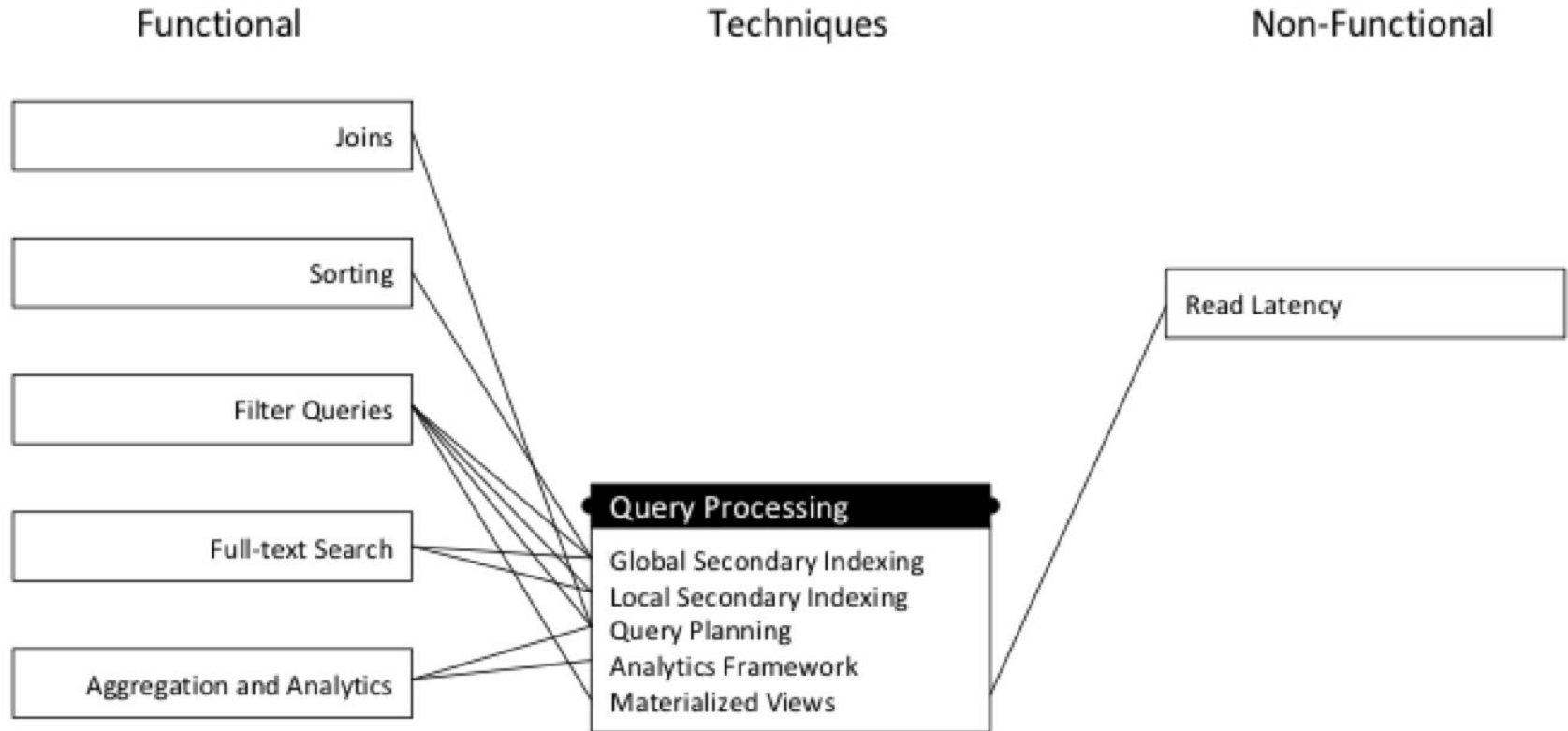
- Ideal for RAM storage: random writes to RAM are essentially equally fast as sequential ones
- For persistent storage: to mitigate the **slow random access** to persistent storage, main memory is usually used as a **cache** and complemented by **logging** to guarantee durability

append-only storage (also referred to as **log-structuring**):

- tries to maximize throughput by writing sequentially
- Is often implemented using the Log-Structured Merge (LSM) trees consisting of an in-memory cache, a persistent log and immutable, periodically written storage files (example: BigTable)
- LSM trees and variants like Sorted Array Merge Trees (SAMT) and Cache-Oblivious Look-ahead Arrays (COLA) have been applied in many NoSQL systems (Cassandra, CouchDB, LevelDB, Bitcask, RethinkDB, WiredTiger, RocksDB, InfluxDB, TokuDB)



NoSQL Techniques: Query Processing



NoSQL Techniques: Query Processing

The querying capabilities of a NoSQL database mainly follow from its distribution model, consistency guarantees and data model.

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

NoSQL Techniques: Query Processing

To circumvent the inefficiencies of $O(n)$ scans, **secondary indexes** can be used, of two types:

- **local secondary indexes** that are managed in each partition
 - global queries over local secondary indexes can only be targeted to a subset of partitions if the query predicate and the partitioning rules intersect. Otherwise, results have to be assembled through scatter-gather
- **global secondary indexes** that index data over all partitions
 - Since the global index itself has to be distributed over partitions, consistency becomes a problem due to slow and potentially unavailable commit protocols
 - Therefore in practice, most systems only offer **eventual consistency** for these indexes (e.g. Megastore, Google AppEngine Datastore, DynamoDB) or **do not support them at all** (e.g. HBase, Azure Tables).
 - A special case of global secondary indexing is **full-text search**, where selected fields or complete data items are fed into either a database-internal inverted index (e.g. MongoDB) or to an external search platform such as ElasticSearch or Solr (Riak Search, DataStax Cassandra)

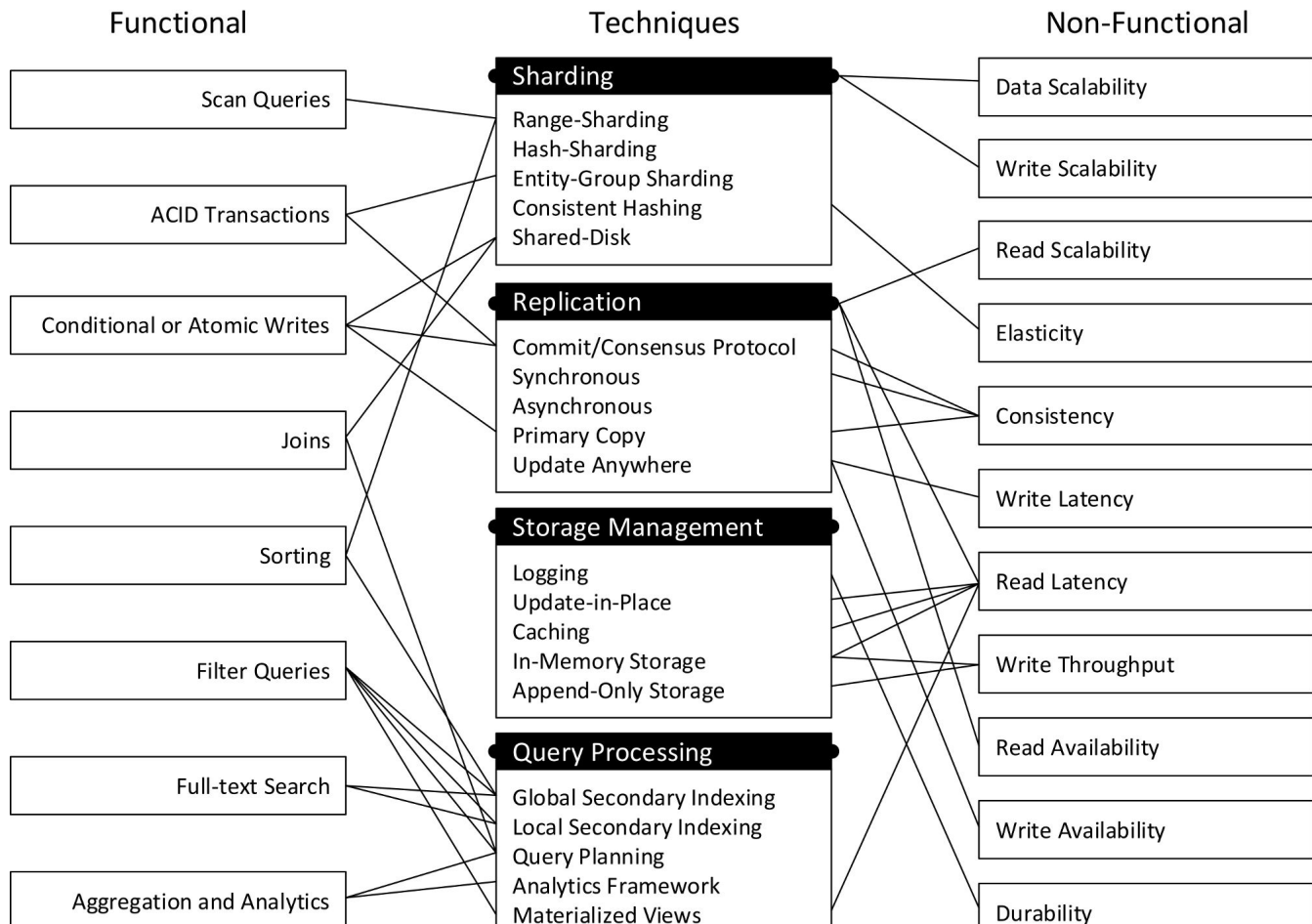
NoSQL Techniques: Query Processing

In-database analytics:

- can be performed either natively (e.g. in MongoDB, Riak, CouchDB)
- or through external analytics platforms such as Hadoop, Spark and Flink (e.g. in Cassandra and HBase)
- The prevalent native batch analytics abstraction exposed by NoSQL systems is **MapReduce**. (An alternative to MapReduce are generalized data processing pipelines, where the database tries to optimize the flow of data and locality of computation based on a more declarative query language, e.g. MongoDB's aggregation framework.)
 - Due to I/O, communication overhead and limited execution plan optimization, these batch- and micro-batch-oriented approaches have high response times
- **Materialized views** are an alternative with lower query response times. They are declared at design time and continuously updated on change operations (e.g. in CouchDB and Cassandra)
 - However, similar to global secondary indexing, view consistency is usually relaxed in favor of fast, highly-available writes, when the system is distributed.

As only few database systems come with built-in support for ingesting and querying unbounded streams of data, near-real-time analytics pipelines commonly implement either the Lambda Architecture or the Kappa Architecture

NoSQL Classification Toolbox - Summary



NoSQL Toolbox: example classification of 5 DBs

A direct comparison of functional requirements, non-functional requirements and techniques among MongoDB, Redis, HBase, Riak, Cassandra and MySQL according to the NoSQL Toolbox

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

NoSQL

This comparison demonstrates how SQL and NoSQL databases are designed to fulfill very different needs:

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

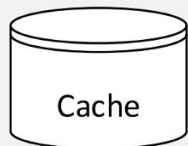
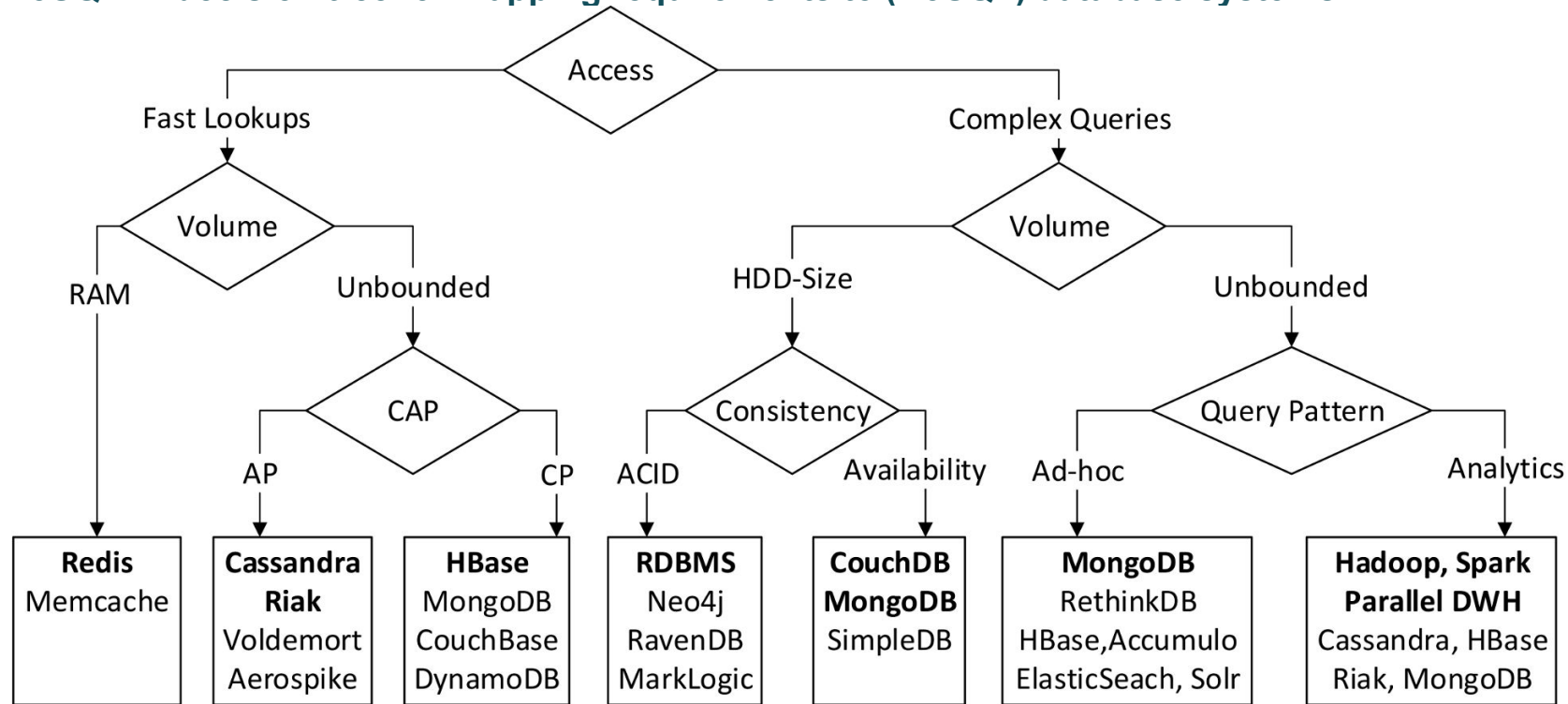
NoSQL

Choosing a database system always means **to choose one set of desirable properties over another !!**

To break down the complexity of this choice, the following binary decision tree was created, that maps trade-off decisions to example applications and potentially suitable database systems.



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

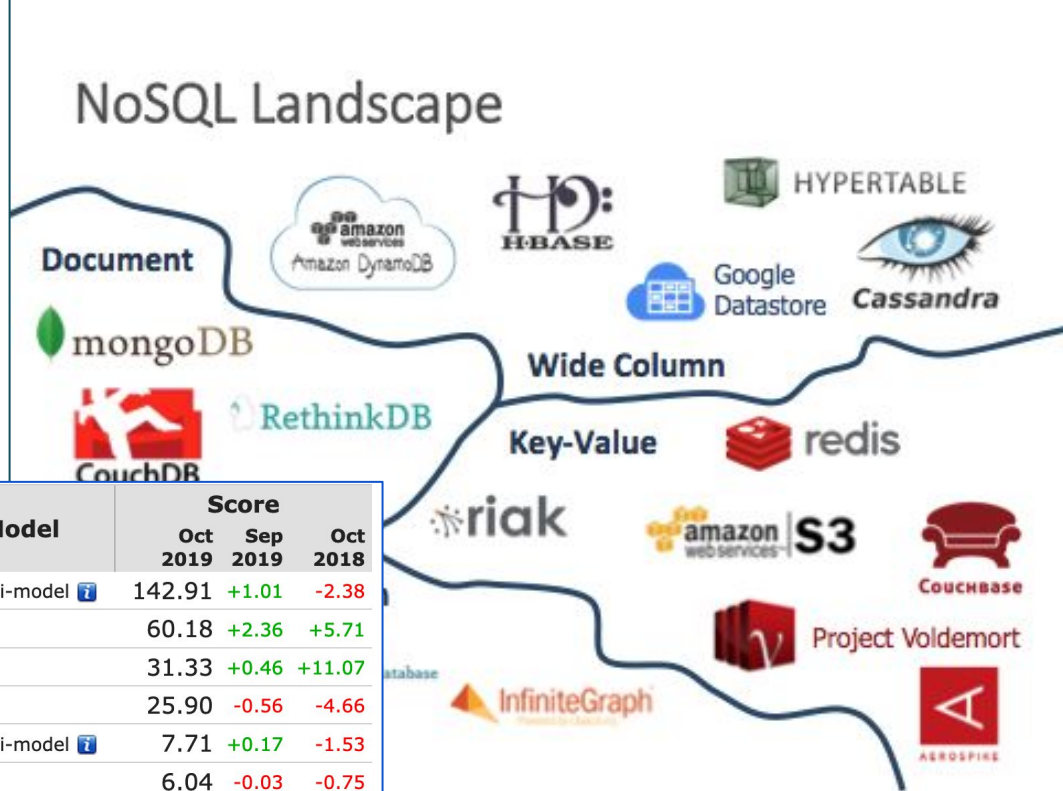


Example Applications

NoSQL DBs

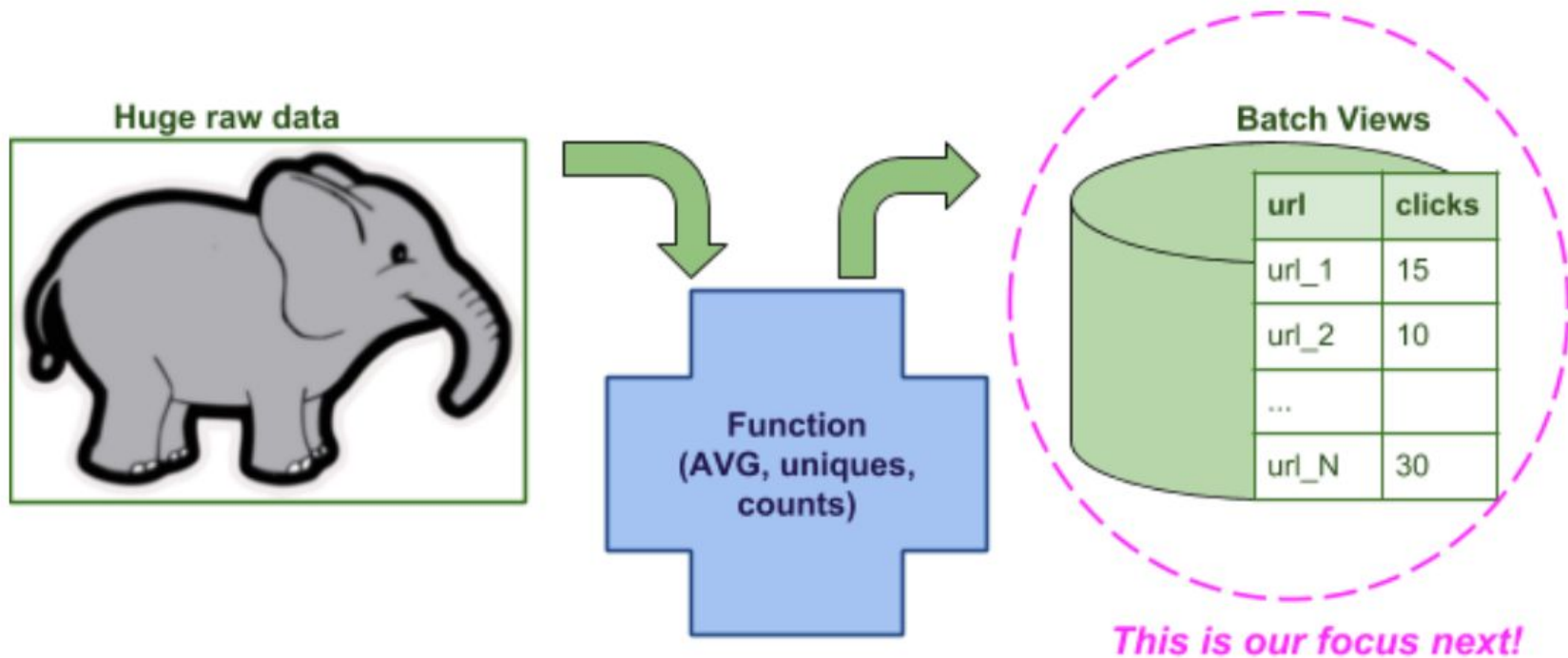
Categorizations are very subjective!

<https://db-engines.com/en/ranking/key-value+store>



Rank			DBMS	Database Model	Score		
Oct 2019	Sep 2019	Oct 2018			Oct 2019	Sep 2019	Oct 2018
1.	1.	1.	Redis +	Key-value, Multi-model	142.91	+1.01	-2.38
2.	2.	2.	Amazon DynamoDB +	Multi-model	60.18	+2.36	+5.71
3.	3.	4.	Microsoft Azure Cosmos DB +	Multi-model	31.33	+0.46	+11.07
4.	4.	3.	Memcached	Key-value	25.90	-0.56	-4.66
5.	5.	5.	Hazelcast	Key-value, Multi-model	7.71	+0.17	-1.53
6.	6.	6.	Ehcache	Key-value	6.04	-0.03	-0.75
7.	8.	9.	Aerospike +	Key-value	5.77	+0.11	+0.42
8.	7.	7.	Riak KV	Key-value	5.56	-0.12	-1.14
9.	9.	8.	OrientDB	Multi-model	5.14	+0.07	-0.55
10.	10.	10.	ArangoDB	Multi-model	4.88	+0.52	+0.63
11.	11.	11.	Ignite	Multi-model	4.45	+0.45	+0.43
12.	12.	12.	Oracle NoSQL +	Key-value, Multi-model	3.39	+0.27	+0.47
13.	13.	13.	InterSystems Caché	Multi-model	3.09	+0.04	+0.24

Moving On ... to Batch Views



Batch Views

What exactly are the Batch Views?

It is a storage which holds pre-computed values of the queries we want to ask, over a period of time.

What do they give us?

A much faster access to the results of such queries.

Next:

- how to store? - models
- how to compute?
- where to store?

Batch Views Modeling

Example:

raw data in the following format, for the past 2 years:

<uuid> <timestamp> <url> <userID>

[uuid0]	[19/Jun/2015 11:05:00 +0200]	url1	user01
[uuid1]	[19/Jun/2015 11:44:17 +0200]	url1	user01
[uuid2]	[19/Jun/2015 11:45:05 +0200]	url2	user02
[uuid3]	[19/Jun/2015 12:04:17 +0200]	url1	user01
[uuid4]	[14/Jul/2015 01:11:00 +0200]	url1	user01
[uuid5]	[14/Jul/2015 01:44:00 +0200]	url2	user02
[uuid6]	[14/Jul/2015 02:44:17 +0200]	url1	user02

Query: get count of events per URL per hour

To get an answer - query plan:

- scan full master dataset
- group data by hour
- group by unique URLs in the hour buckets - get the count of events

Important questions:

- How many hour buckets are in a 2 year period? 17520
- How many events can be in each hour bucket? ... could be Millions or Billions....
- Does the result of computation change? No - it is the total number of clicks, not de-duped across windows; change in the semantic meaning is very unlikely ...

This means we can pre-compute results for this query over the past 2 years and store them for a faster access later

Batch Views

To store the results of this query we could use a Batch View like this:

hour	URL	count
19/Jun/2015 11	url1	2
19/Jun/2015 11	url2	1
19/Jun/2015 12	url1	1
14/Jul/2015 01	url1	1
14/Jul/2015 01	url2	1
14/Jul/2015 02	url1	1

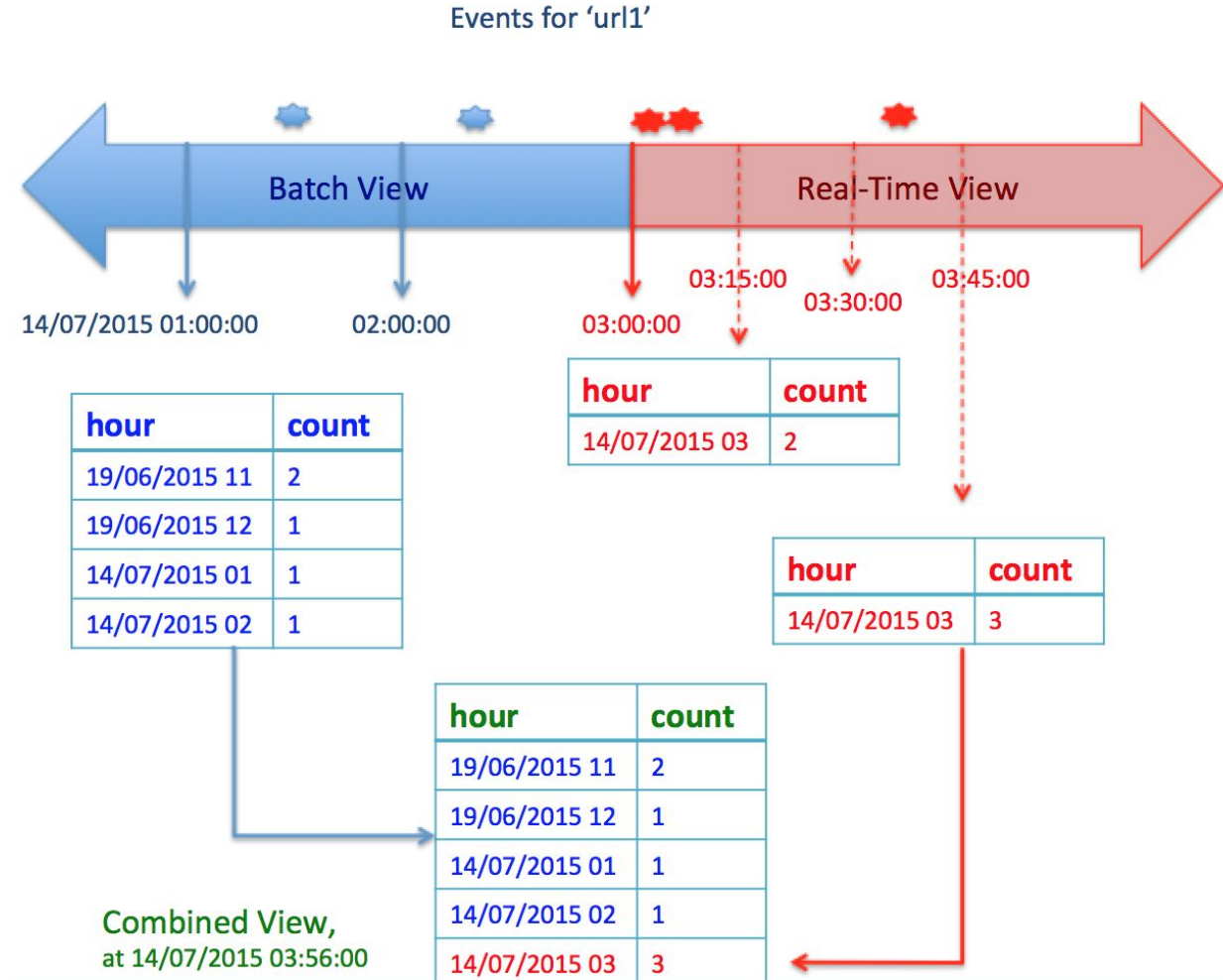
How would we use it to get the final real-time query result?

Query A: Get count of events per 1 hr for url = 'url1' for the past 2 years

Lets assume right now is 14/Jul/2015 03:56:00

Lets assume we have a streaming layer that reads latest events from Kafka and does windowed 1-hr event count aggregation by URL. Which means it calculates exactly the same data as in the table, but only for **the last 1 hour**

Batch Views



Batch Views Modeling

what if we want to also run Queries like:

- get a count of events per URL per each 2 hrs?
- per each 24 hrs?
- per any time period specified by the user?

Implementation Options:

Option 1:

run each query over the full dataset of 2 years -
this requires a full master dataset scan - can be very slow

Option 2:

use pre-computed 1hr counts and combine the results
for different time periods

Ref: "Big Data" by Nathan Martz

2-hr derived Batch View:
(same goes for other time intervals)

2-hour interval	URL	count
19/Jun/2015 11-01	url1	3
19/Jun/2015 11-01	url2	1
14/Jul/2015 01-03	url1	2
14/Jul/2015 01-03	url2	1

How will this approach work over time?

Since we can always combine the values of any time interval - we only need to keep computing the query results for the new, incoming data. Once a full hour of data is computed - we can store the final value into the Batch View and use it as a read-only value going forward

Batch Views Modeling

This is what is called an **Incremental algorithm** approach:

- batch view results are calculated **once only**
- streaming layer calculates the same query results over the latest in-coming data
- once the next time interval has finished, and results for it are calculated - they are added, **incrementally**, to the Batch View storage
- querying 1-hr Batch Views is **much faster** than querying the full master dataset

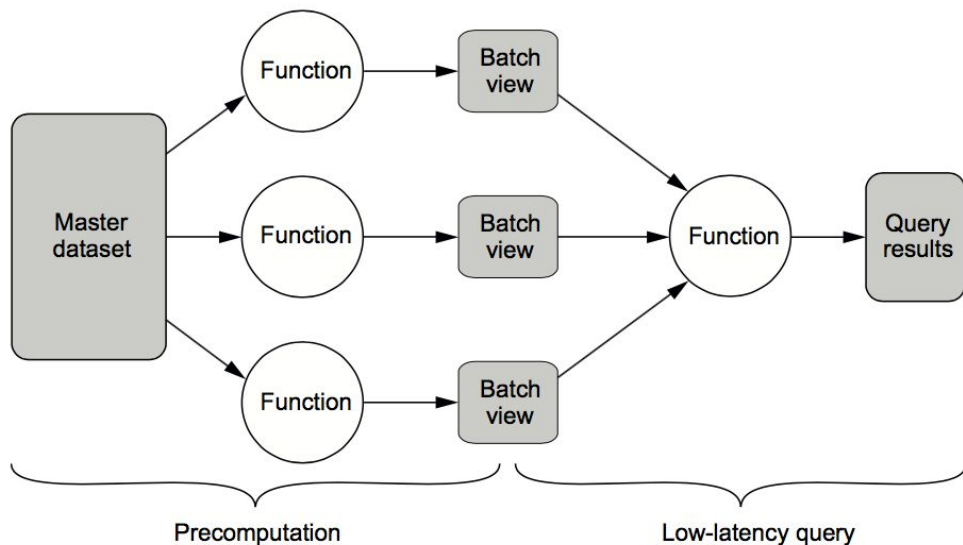


Figure 6.3
Splitting a
query into pre-
computation and
on-the-fly compo-
nents

Ref: "Big Data" by
Nathan Martz

Batch Views Modeling

Ok, this worked great! Why not just pre-compute everything we need and keep updating the batch views, incrementally, as time goes by? What exactly is "everything" ??

Lets add **time-to-load** in ms, to each request:

Raw data:

[uuid0]	[19/Jun/2015 11:05:00 +0200]	url1	user01	5.1
[uuid1]	[19/Jun/2015 11:44:17 +0200]	url1	user01	6.1
[uuid1_2]	[19/Jun/2015 11:45:00 +0200]	url1	user02	4.1
[uuid2]	[19/Jun/2015 11:45:05 +0200]	url2	user02	15.5
[uuid3]	[19/Jun/2015 12:04:17 +0200]	url1	user01	5.0
[uuid4]	[14/Jul/2015 01:11:00 +0200]	url1	user01	6.0
[uuid5]	[14/Jul/2015 01:44:00 +0200]	url2	user02	10.0
[uuid6]	[14/Jul/2015 02:01:17 +0200]	url1	user02	5.0
[uuid7]	[14/Jul/2015 02:05:00 +0200]	url1	user02	7.5
[uuid8]	[14/Jul/2015 02:44:01 +0200]	url1	user02	10.2

Query B: Get an average time-to-load per URL

Batch View - V1: AVG TTL per 1-hr

hour	URL	AVG time-to-load
19/Jun/2015 11	url1	5.1
19/Jun/2015 11	url2	4.1
19/Jun/2015 12	url1	5.0
14/Jul/2015 01	url1	6.0
14/Jul/2015 01	url2	10.0
14/Jul/2015 02	url1	7.56

Batch Views Modeling

How to get AVG TTL per 2 (X) hours?

Adding up two 1-hr values and dividing by 2? NO

Solution?

store additional information - the number of events that the AVG was computed over

This is a very good solution for this query:

- We are only adding one more field to store in the Batch View (event count)
- We can take advantage of pre-computed batch view results for all previous hours/years
- This means that the Incremental approach is good for this example

Batch View V2:
1HR:



1 hour	URL	AVG TTL	count
19/Jun/2015 11	url1	5.1	3
19/Jun/2015 11	url2	4.1	1
19/Jun/2015 12	url1	5.0	1
14/Jul/2015 01	url1	6.0	1
14/Jul/2015 01	url2	10.0	1
14/Jul/2015 02	url1	7.56	3

Derived 2HR View

2 hour interval	URL	AVG time-to-load	count
19/Jun/2015 11-1	url1	5.075 $[5.1 \times 3 + 5.0 \times 1] / [3 + 1]$	4
19/Jun/2015 11-1	url2	4.1	1
14/Jul/2015 01-03	url1	7.17 $[6.0 \times 1 + 7.56 \times 3] / [1+3]$	4
14/Jul/2015 01-03	url2	10.0	1

Batch Views Modeling

Lets consider another example:

same raw data:

[uuid0]	[19/Jun/2015 11:05:00 +0200]	url1	user01
[uuid1]	[19/Jun/2015 11:44:17 +0200]	url1	user01
[uuid1_2]	[19/Jun/2015 11:45:00 +0200]	url1	user02
[uuid2]	[19/Jun/2015 11:45:05 +0200]	url2	user02
[uuid3]	[19/Jun/2015 12:04:17 +0200]	url1	user01
[uuid4]	[14/Jul/2015 01:11:00 +0200]	url1	user01
[uuid5]	[14/Jul/2015 01:44:00 +0200]	url2	user02
[uuid6]	[14/Jul/2015 02:01:17 +0200]	url1	user02
[uuid7]	[14/Jul/2015 02:05:00 +0200]	url1	user02
[uuid8]	[14/Jul/2015 02:44:01 +0200]	url1	user02

Query C: **Get count of UNIQUE users per URL**

Batch View V1: 1HR

1-hour	URL	# of unique users
19/Jun/2015 11	url1	2
19/Jun/2015 11	url2	1
19/Jun/2015 12	url1	1
14/Jul/2015 01	url1	1
14/Jul/2015 01	url2	1
14/Jul/2015 02	url1	1



Derived 2HR View

2-hour	URL	# of unique users
19/Jun/2015 11-01	url1	3
19/Jun/2015 11	url2	1
14/Jul/2015 01-02	url1	2
14/Jul/2015 01	url2	1

Batch Views Modeling

Batch View V2: 1HR + list of unique user IDs

1-hour	URL	# of unique users	List of unique users
19/Jun/2015 11	url1	2	user01, user02
19/Jun/2015 11	url2	1	user02
19/Jun/2015 12	url1	1	user01
14/Jul/2015 01	url1	1	user01
14/Jul/2015 01	url2	1	user02
14/Jul/2015 02	url1	1	user01



Derived 2HR View

2-hour window	URL	# of unique users
19/Jun/2015 11-01	url1	2
19/Jun/2015 11	url2	1
14/Jul/2015 01-03	url1	1
14/Jul/2015 01	url2	1

Is this a good solution/approach for this query? It depends!

- Lets consider one extreme: each event has a unique userID.
 - It means that we will be storing values of userIDs per EACH event - the number of stored data points is almost the same as the size of the Master dataset !
- What's a solution then?
 - pre-compute for a fixed set of time intervals - say 1hr, 12 hr and 24hr only
 - re-compute for any other time interval

Batch Views Modeling

What if

- add aggregation by "OS" ?
- .. by "deviceType" ?
- parsing/lookup algorithm for GEO location improved?



Have to **recompute** Batch Views over the Full master dataset again

This means we need to have **BOTH** incremental and re-computing algorithms for the Batch Layer !

Because you cannot pre-compute EVERYTHING !

Batch Views: Incremental vs Re-computing?

Summary : when can you use **Incremental algorithms**?

- Results of queries over two time intervals **can be combined** by a simple operation (like addition)
- If results are a function of the data - it can be applied to a combination of the pre-computed results without accuracy loss and **all required information is also available**:
 - Given $f(\text{interval1})$ and $f(\text{interval2})$: $f(\text{interval1 and interval2}) == F(f(\text{interval1}) \text{ and } f(\text{interval2}))$
 - Example: average:
 - Average time-to-load for hour [15-16] = (Total: 110 / number of events: 20) == 5.5 ms
 - Average time-to-load for hour [16-17] = (Total: 240 / number of events: 30) == 8.0 ms
 - Average time-to-load for hours [15-17] = $(5.5 \text{ ms} \times 20) + (8 \text{ ms} \times 30) / (20+30) = 7.0 \text{ ms}$
- Results **do not depend on uniqueness criteria** - that cannot be realized via the batch views data alone
- Results **do not depend on input data de-duplication**
- **No additional aggregation criteria are added** (like additional dimensions to aggregate by - browser version, request status, etc.)
- There is **no change in the semantic meaning** of the data

In all other cases - you need to use **Re-computing algorithms**

Batch Views Modeling

But wait.... does not it look familiar???

Is not storing counts ---
what we said we should NOT do earlier?

Master vs Batch-Views Storage!!

.... and do we really need it at all ??

@Marina Popova



Batch Layer - really needed or not?

Why is batch layer needed?

- So, why not combine Type3 and Type4 systems into one type?
- Basically, why cannot we do both - historical queries and real-time queries in one component?

Answer: you can, sometimes
-- meet Kappa Architecture:

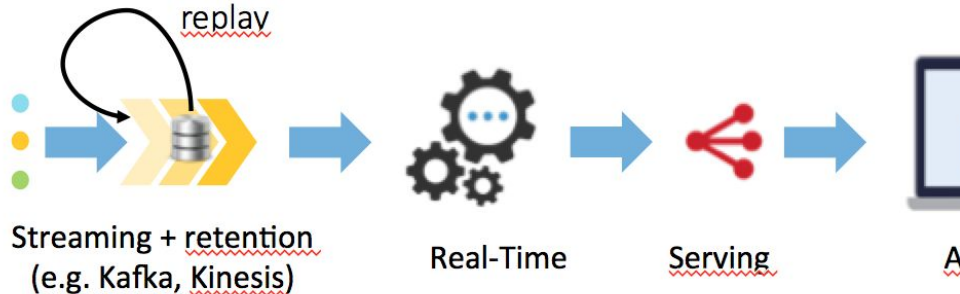
Benefits:

- Simpler than Lambda Architecture
- Data retention for relevant portion of history

Reasons to forgo Kappa:

- Legacy batch system that is not easily migrated
- **Purely incremental algorithms**

Kappa Architecture

$$\text{Stream}(D_{\text{all}}) = \text{Batch}(D_{\text{all}})$$


Jay Kreps, *Questioning the Lambda Architecture* (2014)

<https://www.oreilly.com/ideas/questioning-the-lambda-architecture>

Batch Views: Storage

Next:

- ✓ how to store? - models
- ✓ how to compute?
- where to store?

What DB to use to store results of batch views calculations?

Many options - depending on exact latency vs. CAP requirements

Based on our understanding of the usage patterns for Batch Views ...

Summary of requirements:

- very fast random reads
- **random writes not required**
- horizontal scalability
- Latency:
 - Low for reads
 - High for writes

Batch Views

This lands us into this category of requirements:

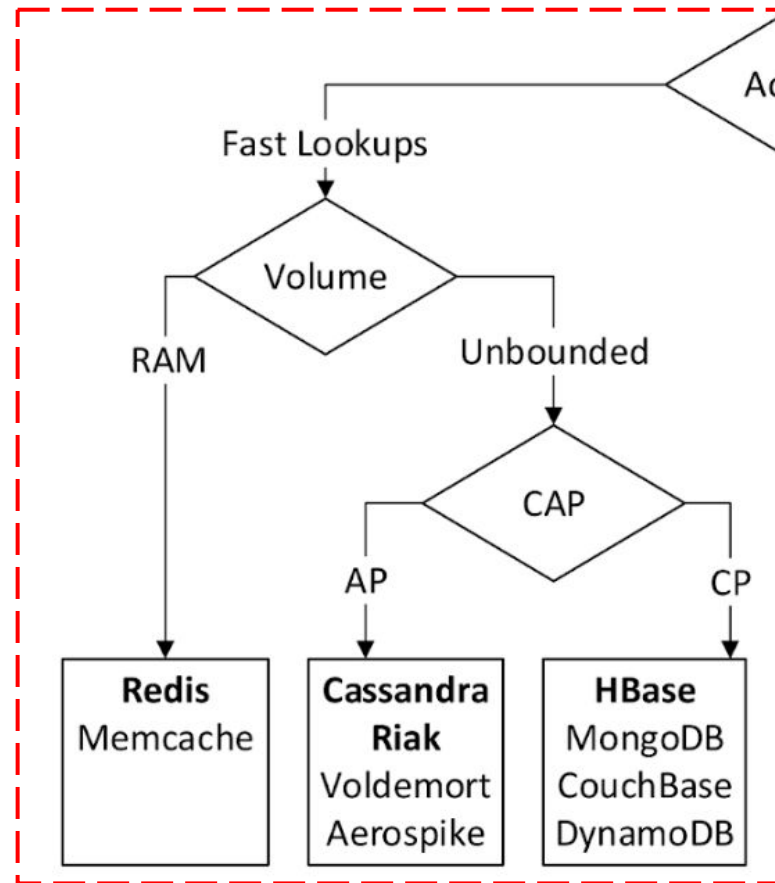
Next: your choice of:

- size VS.
- ~AP VS.
- ~CP

Great example of a real-life project that had to make these decisions and pick options:

<https://medium.com/@BoxeverTech/blue-green-databases-read-only-cassandra-6b6d83e07217>

- Hive
- HBase
- Cassandra
- Redis
- ...




Google BigTable: Grandfather of all ...

Google BigTable (CP)

- ▶ Published by Google in 2006
- ▶ Original purpose: storing the Google search index

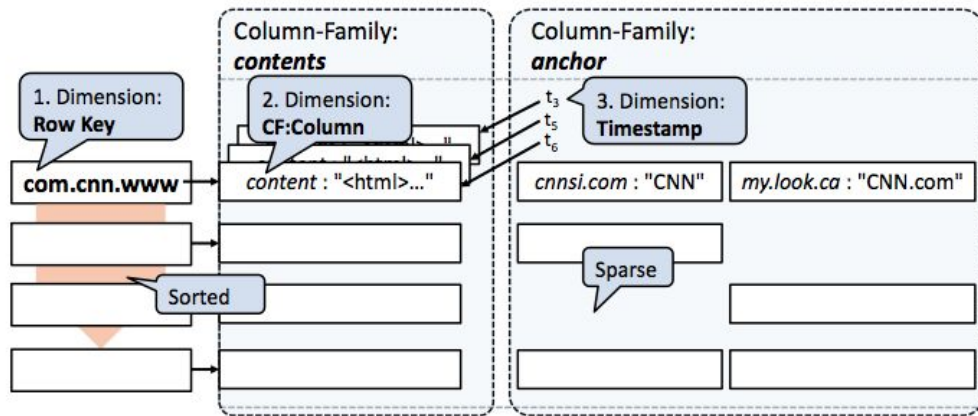
A Bigtable is a sparse, distributed, persistent multidimensional sorted map.

- ▶ Data model also used in: HBase, Cassandra, HyperTable, Accumulo

 Chang, Fay, et al. "Bigtable: A distributed storage for structured data."

Wide-Column Data Modelling

- ▶ Storage of crawled web-sites („Webtable“):

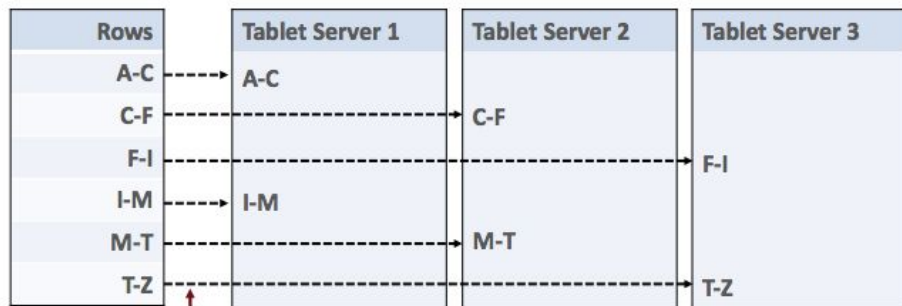


Batch Views Options: Google BigTable - like

Range-based Sharding

BigTable Tablets

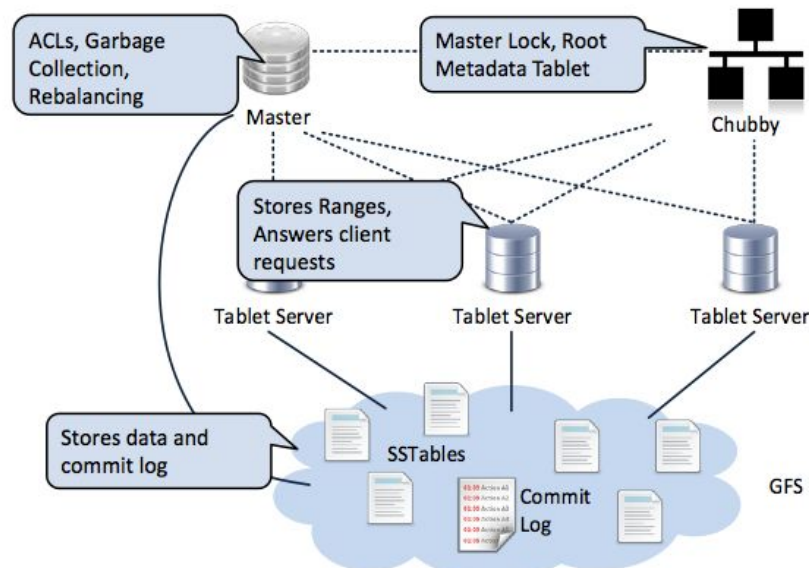
Tablet: Range partition of ordered records



Master

Controls Ranges, Splits, Rebalancing

Architecture

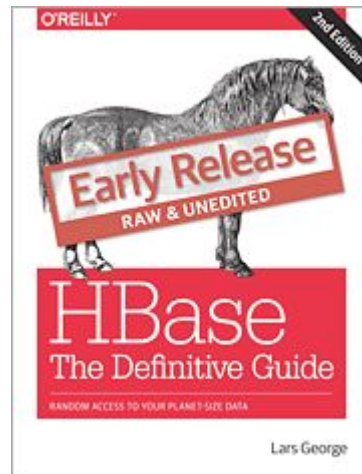


Apache HBase

Apache HBase Reference Guide: https://hbase.apache.org/book.html#getting_started

The Definitive Guide: <https://www.oreilly.com/library/view/hbase-the-definitive/9781492024255/>

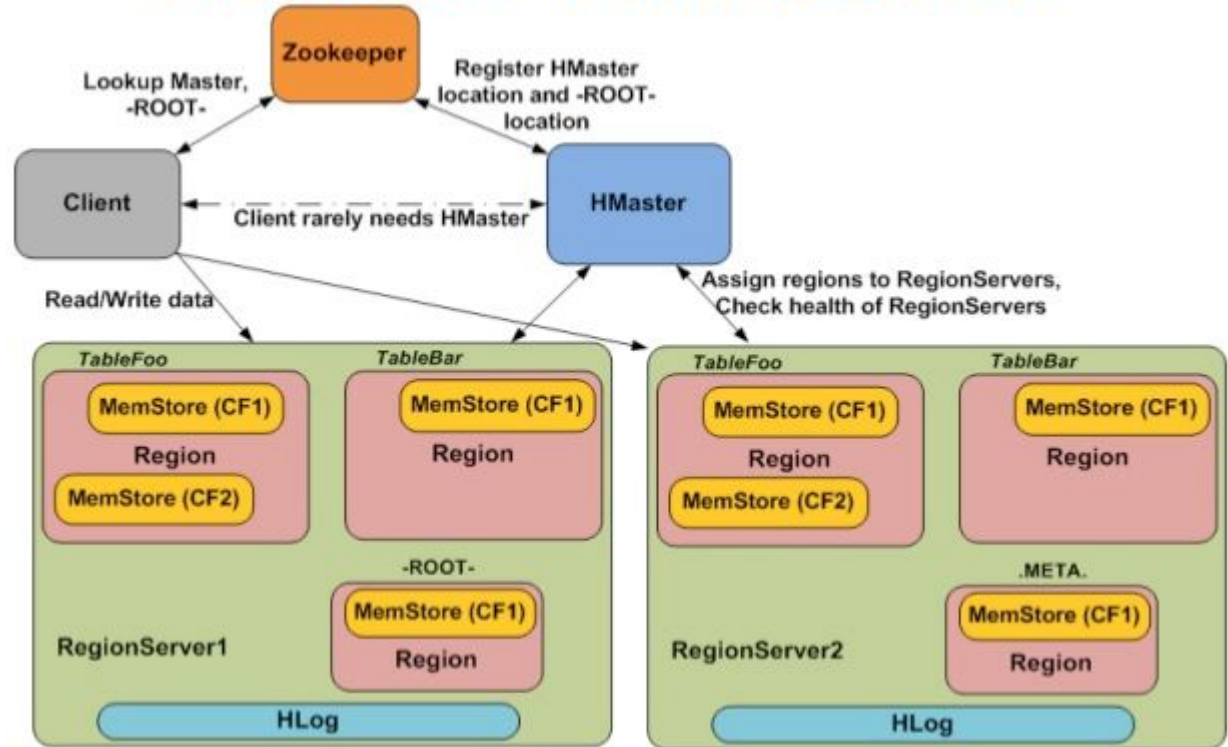
- OpenSource implementation of Google BigTable!
- written in Java
- designed to provide low-latency random reads and writes on top of HDFS
- Integration with Hadoop: uses ZK and HDFS
- used as input/output for MR jobs



Apache HBase

- single HBase master node (HMaster)
- several slaves i.e. region servers
- when a client sends a write request, HMaster receives the request and forwards it to the corresponding region server
- tables are dynamically distributed across Regions by the system whenever they become too large to handle (Auto Sharding)

HBase Architecture



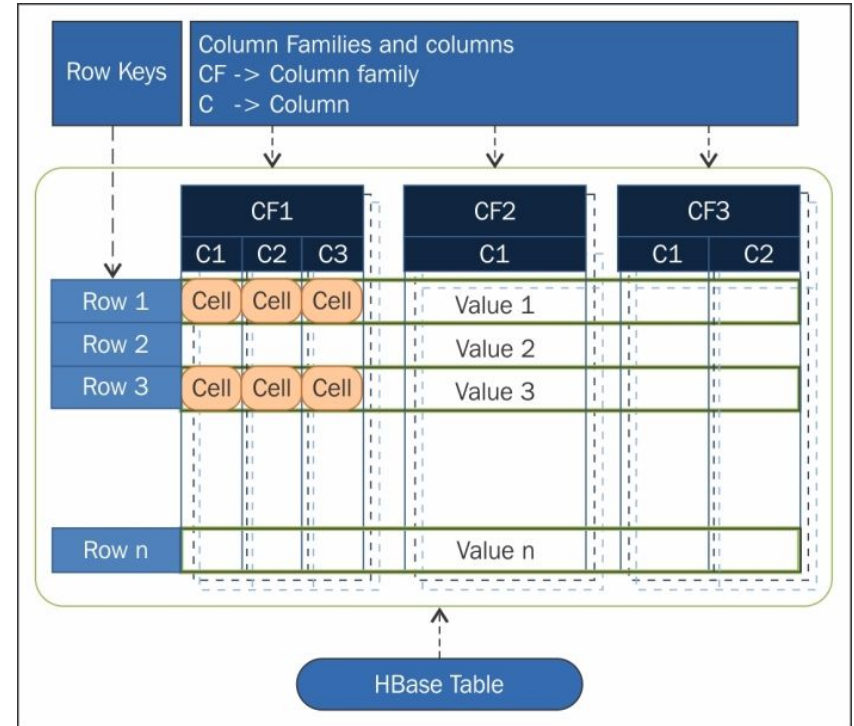
Apache HBase Data Model Concepts

Ref:

https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781784396688/5/ch05lvl1sec44/the-hbase-data-model

HBase Data Model Main Concepts:

- **Table** - collection of Rows, partitioned into **Regions**
- **Rows** - logical representation of data, collection of Columns, identified by **RowKey**
- **Columns** - arbitrary number of attributes, grouped into Column Families (CFs)
- **Column Families** - groups of columns stored into a single low-level storage file, **HFile**
- **Cells** - contain the actual values with timestamps
- **versions/ timestamps**

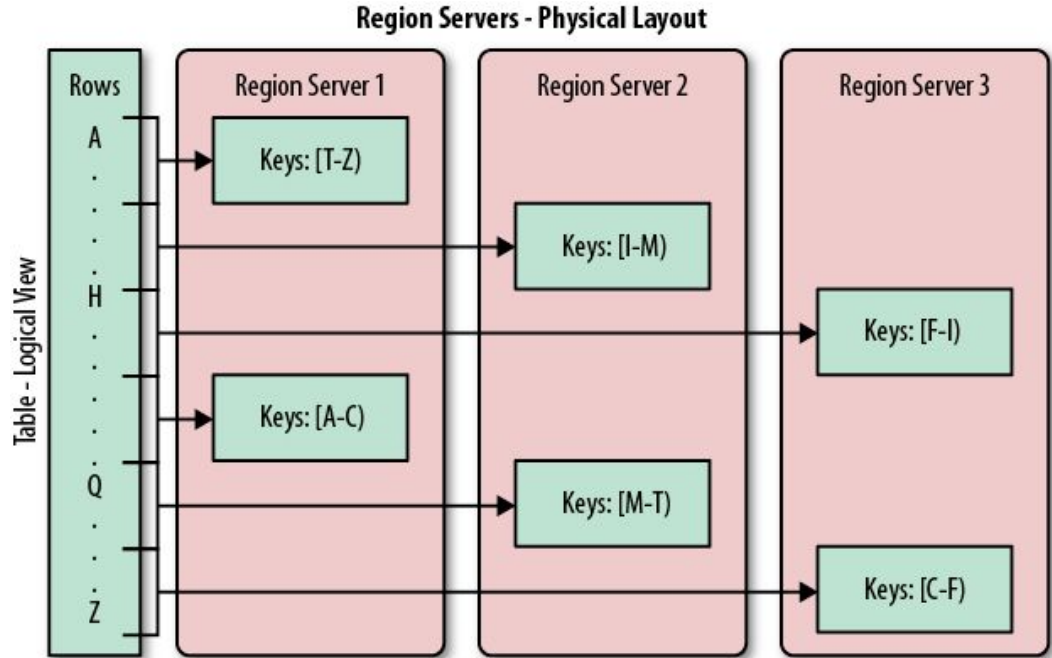


Apache HBase - Architecture

Ref: "*HBase: the Definitive Guide*"

HBase Regions and Auto-Sharding:

- Tables are divided into contiguous sets of rows stored together, called Regions
- The HBase regions are equivalent to *range partitions* as used in database sharding
- basic unit of scalability and load balancing
- Tables and Regions are dynamically split by the system when they become too large
- Each region is assigned to and served by exactly one **Region Server**, and each of these servers can serve many regions at any time.



Region Serv

Figure: Region Server Components



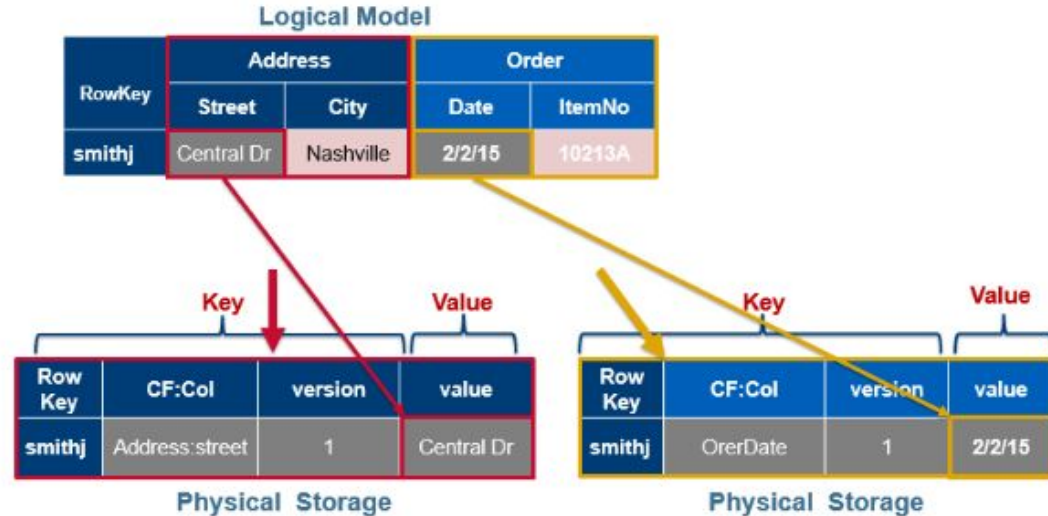
HBase Data Model Details

Ref:
<http://www.corejavaguru.com/bigdata/hbase-tutorial/data-model>

HBase data model: is a sparse, distributed, persistent, multidimensional map, which is indexed by row key, column key, and a timestamp

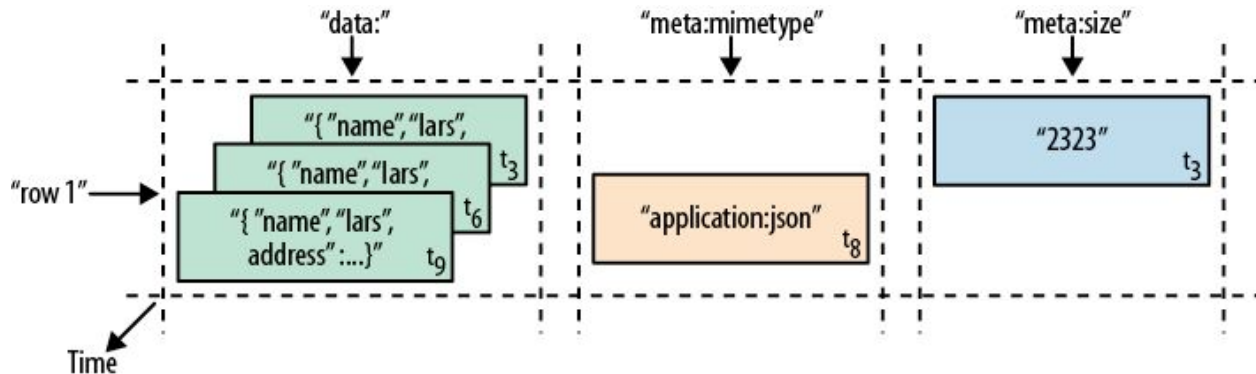
- HBase Raw Cells hold data values
- cells are identified by:
 - row key
 - column key = CF + Column + Timestamp
 - version (optional)
- **cells are accessed by Keys only - no indices!**
Key design is very important!
- all columns in a CF are stored together in the same storage file, **HFile**
- CF is also used as a grouping unit for compression and encoding

Cell Coordinates= Key				Value
Row key	Column Family	Column Qualifier	Timestamp	Value
Smithj	Address	city	1391813876369	Nashville



HBase Data Model Details - Versions

- every column value, or *cell*, either is timestamped implicitly by the system or explicitly by the user
- each cell can have multiple *versions* of a value
- you can query **ranges of values** by those timestamps



more on Cell values:

- stored as uninterpreted byte content (no schema)
- application is responsible for serialization
- can interface with other SerDe frameworks like Avro, Thrift

Row Key	Time Stamp	Column "data:"	Column "meta:"		Column "counters:" "updates"
			"mimetype"	"size"	
"row1"	t ₃	"{"name": "lars", "address": ...}"		"2323"	"1"
	t ₆	"{"name": "lars", "address": ...}"			"2"
	t ₈		"application/json"		
	t ₉	"{"name": "lars", "address": ...}"			"3"

HBase Data Model Operations

Main Operations:

- GET
 - returns all attributes (columns) for a specified row
 - can get specific CFs/Columns/Cells by specifying full lookup keys
 - full cell key: [TableName, RowKey, CF, Column, Timestamp, Version] --> Value
- PUT
 - creates new rows for new keys
 - updates (not real update, versioned) values for existing keys
- SCAN - iterate over multiple rows for specified attributes
- DELETE
 - does NOT delete data but rather puts a tombstone marker on
 - eventually removed by major compactions
- **no updates in place!**

HBase Data Model Details - More on Rows


- Row keys are **sorted lexicographically**:
- Access to one row is atomic - and it includes all columns for reads and writes
- One can have an atomic operation (get/put) over multiple rows - **as long as they are in the same region !**
- There is no guarantee of transactional feature that spans multiple rows across regions, or across tables

Example 1-1. The sorting of rows done lexicographically by their key

```
hbase(main):001:0> scan 'table1'
ROW                                COLUMN+CELL
row-1                             column=cf1:, timestamp=1297073325971 ...
row-10                             column=cf1:, timestamp=1297073337383 ...
row-11                             column=cf1:, timestamp=1297073340493 ...
row-2                             column=cf1:, timestamp=1297073329851 ...
row-22                             column=cf1:, timestamp=1297073344482 ...
row-3                             column=cf1:, timestamp=1297073333504 ...
row-abc                           column=cf1:, timestamp=1297073349875 ...
7 row(s) in 0.1100 seconds
```

Classification: HBase

Techniques

 Sharding	Range-Sharding	Hash-Sharding	Entity-Group Sharding	Consistent Hashing	Shared Disk
 Replication	Transaction Protocol	Sync. Replication	Async. Replication	Primary Copy	Update Anywhere
 Storage Management	Logging	Update-in-Place	Caching	In-Memory	Append-Only Storage
 Query Processing	Global Index	Local Index	Query Planning	Analytics	Materialized Views