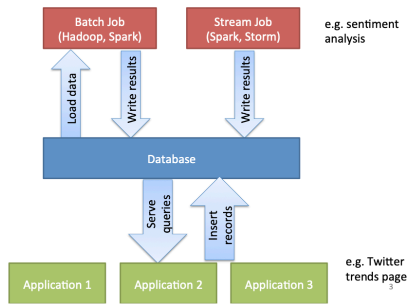
Large Scale Data Management

NoSQL

Silviu **Maniu**, LIG, Univ. Grenoble Alpes

Data is Central

Processing / Database Link



Data Depends on the Application

- Stock management
- Health insurance management
- Health records management
- Payroll
- Shopping
- Tweet news
- TikTok videos

…

Design questions

- **Structure** ? – schema ?
- **Access** ? – whole/part ?
- **Queries** ? – simple, complex ?
- **Volume** ? – centralized/ distributed ?
- **Evolution** ? – add attributes ?
- **Guarantees** ? – types ?

Common Patterns of Data Accesses

Large-scale data processing

- Batch processing: Hadoop, Spark, etc.
- Perform some computation/transformation over a full dataset
- Process all data

Selective query

- Access a specific part of the dataset
- Manipulate only data needed (1 record among millions)
- Main purpose of a database system

Types of Databases

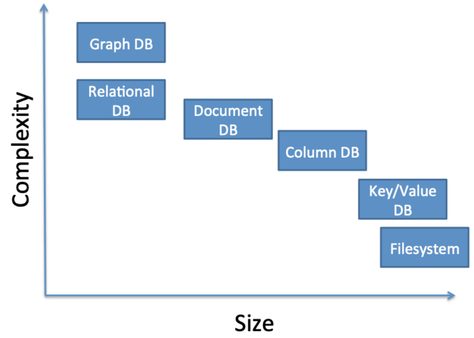
A file system can be seen as a very basic database

- Directories / files to organize data
- Very simple queries (file system path)
- Very good scalability, fault tolerance ...

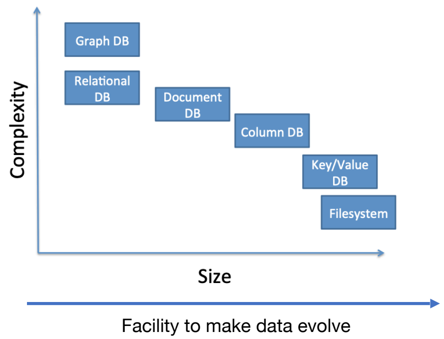
Other end of the spectrum: relational databases

- SQL query language, very expressive
- Limited scalability
- Very complex data evolution potential

Size / Complexity



Size / Complexity / Facility to Change Data



Motivations for Alternative Models

Limitations of Relational Databases

- Performance and scalability
  - Difficult to partition the data (in general run on a single server)
  - Need to scale up to improve performance
- Lack of flexibility
  - Will to easily change the schema
  - Need to express different relations
  - Not all data are well structured
- Few open source solutions
- Mismatch between the relational model and object-oriented programming model

Illustration of the Object-Relational Mismatch

Figure by M. Kleppmann

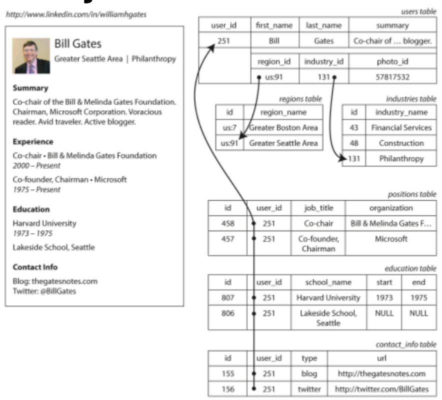


Figure: A CV in a relation database

Illustration of the Object-Relational Mismatch

Figure by M. Kleppmann

```
{
  "user_id":251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co—chair of the Bill & Melinda Gates; Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {"job_title": "Co—chair", "organization": "Bill & Melinda Gates
         Foundation"},
    {"job_title": "Co—founder, Chairman", "organization": "Microsoft"}
  ],
  "education": [
    {"school_name": "Harvard University", "start": 1973, "end": 1975},
    {"school_name": "Lakeside School, Seattle", "start": null, "end": null}
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

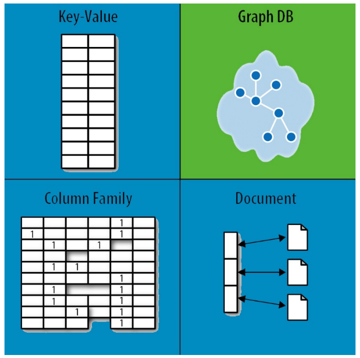
Figure: A CV in a JSON document

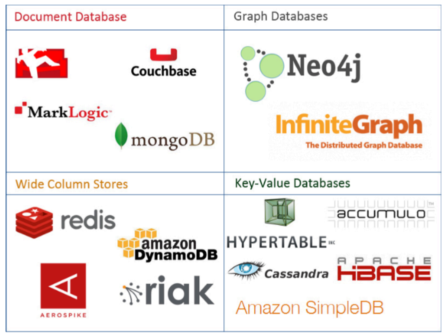
NoSQL

What is NoSQL?

- A hashtag
  - NoSQL approaches were existing before the name became famous
- No SQL
- New SQL
- Not only SQL
  - Relational databases will continue to exist alongside non-relational datastores

The NoSQL Jungle

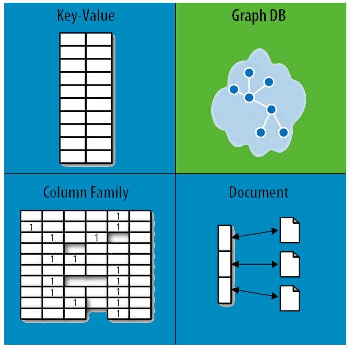




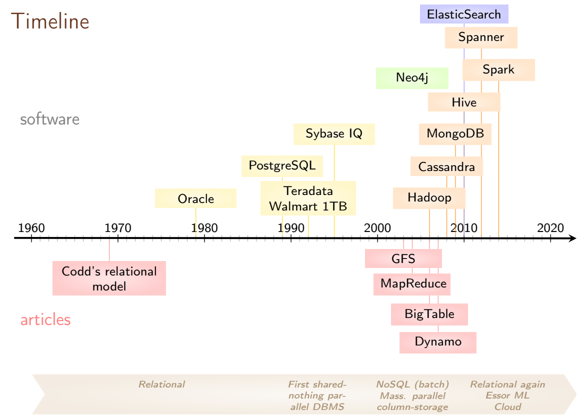
A variety of NoSQL solutions

Difference with relational databases

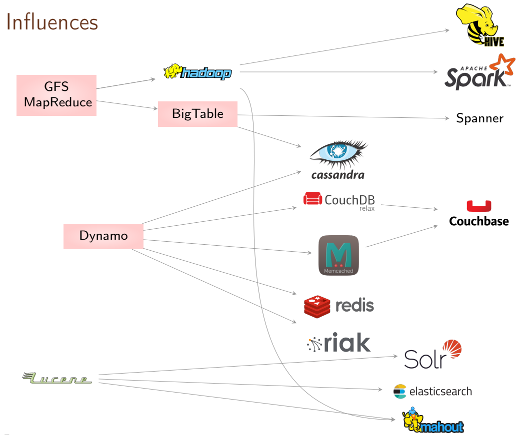
- Properties = guarantees
- Data models = data structure
- Underlying architecture = implementation and performance



A Timeline



Who Influenced Who?



Types of Parallelism

Parallelism in DBMS has a long history:

- inter-operator: every CPU computes a query operation (pipeline); volcano model – a query operation sends the output directly to the next operation.
- intra-operator: every CPU computes the entire query on a part of the data
- inter-query: several queries executed in parallel

Since then:

- large scale data: distributed computing on a large amount of computers.
- shared nothing architecture

Distributed System

A distributed computing system is a system including several computational entities where:

- Each entity has its own local memory
- All entities communicate by message passing over a network

Each entity of the system is called a node.

Distributed Databases

Why distribute?

- parallelism (=performance)
- scalability
- availability: accessibility and fault tolerance (cloud)
- optimize for different hardware, distribute geographically,...

Implementation challenges

- decentralized architecture; maintain coherence between copies, task and data partitioning
- shared nothing architecture (no shared disk, not shared memory pool); how to chose the partitioning

How to Distribute

How to distribute data: partitioning and replication

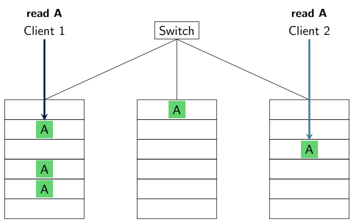
Replication

- Several nodes host a copy of the data
- Main goal: Fault tolerance
  No data lost if one node crashes

Partitioning

- Splitting the data into partitions
- Partitions are assigned to different nodes
- Main goal: Performance
  Partitions can be processed in parallel

Replication



Objectives: reliability, read performance.

Techniques:

- RAM+logs on disk: write-ahead logs (WAL)
- generally, asynchronous (eventual consistency)
- sometimes synchronized (but can have slow updates)
- versioning (vector clocks)
- network state (faults,...): gossip
- fault recovery: consensus (Paxos)

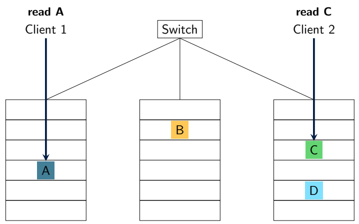
Ex: MongoDB: asynchronous, WAL.

In distributed DBMS:

PostgreSQL (WAL), MariaDB, Oracle (materialized views), SQL Server...

Often admin level choices (number of masters, synchronization,...)

Partitioning / Sharding



Objectives: performance by distributing the load

Main challenge: how to distribute the load (for reads or for writes?)

Challenges of Partitioning / Replicating

- good data partition/replication
- coherence (trade-off between performance and integrity when dealing with reads and writes)
- distributing computation tasks (to minimize data exchange)
- fault tolerance
- transaction control
- data privacy

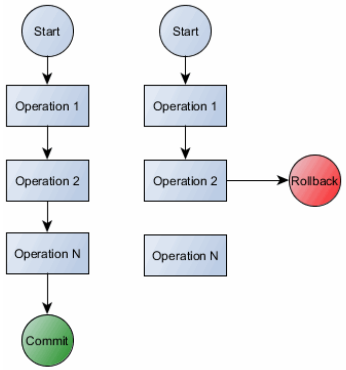
Distributed Architectures

Master-Slave: MongoDB, HDFS, BigTable

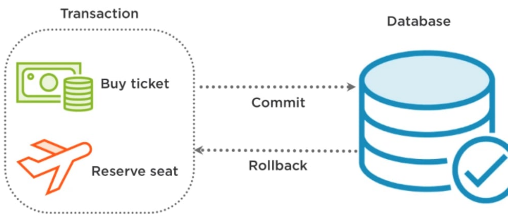
Decentralised: Dynamo, Cassandra

On Guarantees: Transactions

- The concept of transaction
  - Groups several read and write operations into a logical unit
  - A group of reads and writes are executed as one operation:
    - The entire transaction succeeds (commit)
    - or the entire transaction fails (abort, rollback)
- If a transaction fails, the application can safely retry



Example of a Transaction

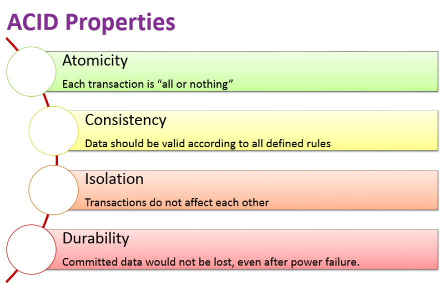


Why Transactions?

- Crashes may occur at any time
  - On the database side
  - On the application side
  - The network might not be reliable
- Several clients may write to the database at the same time

ACID Properties

- Having such properties make the life of developers easy, but:
  - ACID properties are not the same in all databases
  - It is not even the same in all SQL databases
- NoSQL solutions tend to provide weaker safety guarantees
  - To have better performance, scalability, etc.



Atomicity

Description

- A transactions succeeds completely or fails completely
  - If a single operation in a transaction fails, the whole transaction should fail
  - If a transaction fails, the database is left unchanged
- It should be able to deal with any faults in the middle of a transaction
- If a transaction fails, a client can safely retry

In the NoSQL context:

- Atomicity is still ensured

Consistency

Description

- Ensures that the transaction brings the database from a valid state to another valid state
  - All university staff is paid at the end of month
- It is a property of the **application**, not of the database

In the NoSQL context:

- Consistency is (often) not discussed

Durability

Description

- Ensures that once a transaction has committed successfully, data will not be lost
  - Even if a server crashes (flush to a storage device, replication)

In the NoSQL context:

- Durability is also ensured

Isolation

Description

- Concurrently executed transactions are isolated from each other
  - We need to deal with concurrent transactions that access the same data
- Serializability
  - High level of isolation where each transaction executes as if the transactions are applied serially, one after the other

In the NoSQL context:

- Let us have a look at the CAP theorem

The CAP "Theorem" (E. Brewer, 2000)

3 properties of databases

Consistency

- What guarantees do we have on the value returned by a read operation?
  - It strongly relates to Isolation in ACID (and not to consistency)

Availability

- The system should always accept updates

Partition tolerance

- The system should be able to deal with a partitioning of the network

The CAP Theorem Statement

It is impossible to have a system that provides Consistency, Availability, and Partition tolerance at the same time.

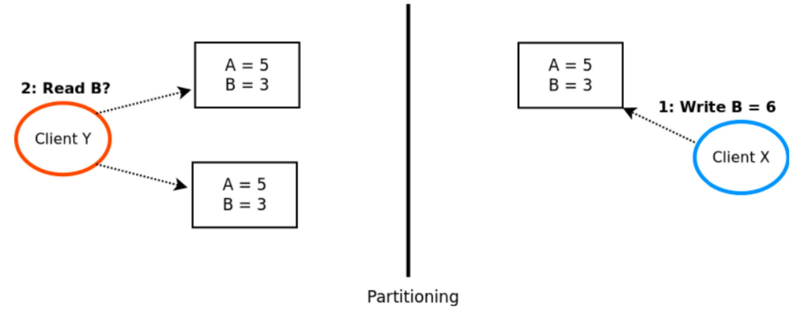
Partitioning (failures) are inevitable in a large scale distributed setting => need to **choose between availability and consistency**

In the CAP theorem:

- Consistency is meant as serializability (the strongest consistency guarantee)
- Availability is meant as total availability

In practice, different trade-offs can be provided

The Intuition Behind CAP



The impact of CAP on ACID for NoSQL

The main consequence

- **No NoSQL database with strong Isolation**

The other ACID properties?

- Atomicity
  - Each side should ensure atomicity
- Durability
  - Should never be compromised

Key-Value Store

- Data are stored as key-value pairs
  - The value can be a data structure (eg, a list)
- In general, only support single-object transactions
  - In this case, key-value pairs
- Examples:
  - Redis
  - Amazon DynamoDB
- Use cases
  - Scalable cache for data
  - Client sessions
  - ...
- Note that some solutions ensure durability by writing data to disk

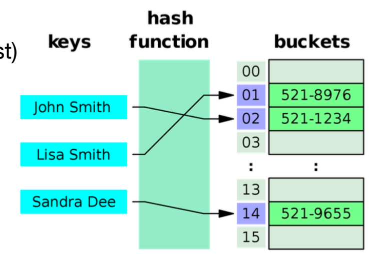
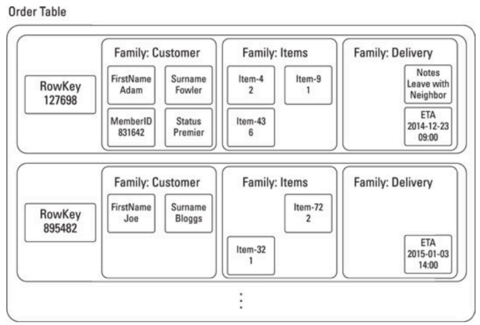


Image by J. Stolfi

Column Family Stores

- Data are organized in rows and columns (Tabular data store)
  - The data are arranged based on the rows
  - Column families are defined by users to improve performance
  - The idea is to group related columns together
- Only support single-object transactions
  - In this case, a row
- Examples:
  - BigTable/HBase
  - Cassandra
- Use case:
  - Data with some structure with the goal of achieving scalability and high throughput
  - Provide more complex lookup operations than KV stores

Column Family Stores



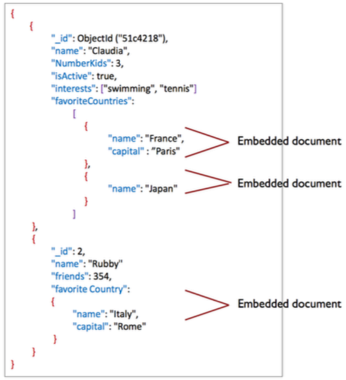
Note that a row does not need to have entries for all columns

Document Databases

- Data are organized in Key-Document pairs
  - A document is a nested structure with embedded metadata
  - No definition of a global schema
  - Popular formats: XML, JSON
- Only support single-object transactions
  - In this case, a document or a field inside a document
- Examples:
  - MongoDB
  - CouchDB
- Use case:
  - An alternative to relational databases for structured data
  - Offer a richer set of operations compared to KV stores:
    - Update, Find, etc.

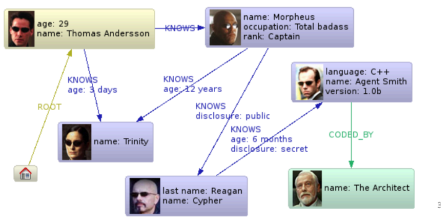
Document Databases

A document can have one or more documents inside.



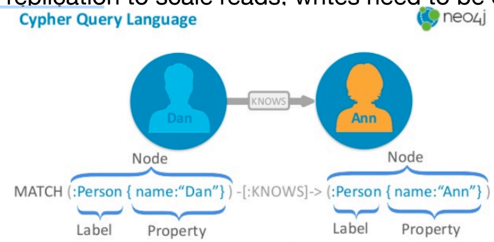
Graph Databases

- Represent data as graphs
  – Nodes / relationships with properties as K/V pairs



Graph DB: Neo4j

- Rich data format
  - Query language as pattern matching
  - Limited scalability: replication to scale reads, writes need to be done to every replica



Relationships in Data

- Many-to-one
  - Example: Many people went to the same university
- One-to-Many: An item may have several entries of the same kind
  - Example: One person may have had several positions during her career
  - Document DB allow storing such information easily and allow simple read operations
- Many-to-Many
  - Example: Several persons may have worked in the same company.
  - Graph DB

Many-to-One

Relational vs Document DB

Relational databases use a foreign key

- Consistency and low memory footprint (normalization)
- Easy updates and support for joins
- Difficult to scale

Document databases duplicate data

- Efficient read operations
- Easy to scale
- Higher memory footprint and updates are more difficult (risk of consistency issues)
  - Transactions on multiple objects could be very useful in this case
- Join operations have to be implement by the application

Google BigTable

- Column family data store
- Data storage system used by many Google services: Youtube,Google maps, Gmail, etc.
  - Paper published by Google in 2006 (F. Chang et al)
- Now available as a service on Google Cloud
- Many ideas reused in other NoSQL databases

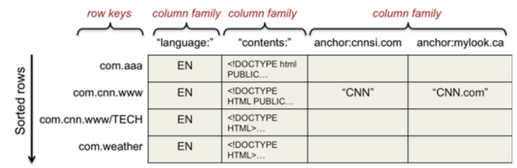
Motivations

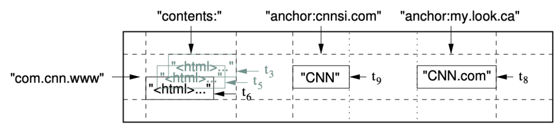
- A system that can store very large amounts of data
  - TB or PB of data
  - A very large number of entries
  - Small entries (each entry is an array of bytes)
- A simple data model
  - Key-value pairs (A key identifies a row)
  - Multi-dimensional data
  - Sparse data
  - Data are associated with timestamps
- Works at very large scale
  - Thousands of machines
  - Millions of users

About the Data Model

- Rows are identified by keys (arbitrary strings)
  - Modifications on one row are atomic
  - Rows are maintained in lexicographic order
- Columns are grouped in columns families
  - Columns can be sparse
  - Clients can ask to retrieve a column family for one row
- Each cell can contain multiple versions indexed by a timestamp
  - Assigned by BigTable or by the client
  - Most recent versions are accessed first
  - GC politics: Keep last n versions or Keep all new-enough versions

About the Data Model

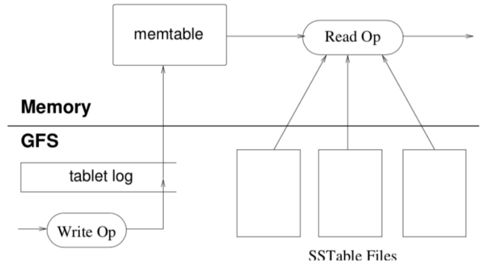




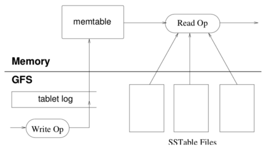
Building Blocks of BigTable

- A master
  - Assign tables parts (tablets) to servers
  - With the help of a locking service
- Tablet servers
  - Store the tables (divided in tablets)
  - Process client requests
- Tablets
  - Stored as SSTables (Sorted String Tables)
  - Stored in the Google File System for durability

Implementation of Tablets



Write Operation



- Data stored in memory (Memtable)
  - Any update is written to a commit log on GFS for durability
  - The log is shared between all hosted tablets
- Periodic writes to disk
  - When the Memtable becomes too big:
    - Copied as a new SSTable to GFS
    - Multiple SSTables are created if locality groups are defined (based on column families)
    - Reduces the memory footprint and reduces the amount of work to do during recovery
    - SSTables are immutable (no problem of concurrency control)

Read Operation

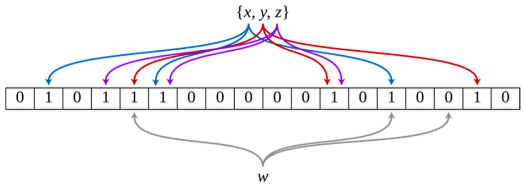
- The state of the tablet = the Memtable + all SSTables
  - A merged view needs to be created
  - The Memtable and the SSTables may contain delete operations
- Locality groups help improving the performance of read operations
- Major compaction
  - When the number of SSTables becomes too big, merge them into a single SSTable
  - Allow reclaiming resources for deleted data
  - Improve the performance of read operations

Bloom Filters and Reads

- During a read operation, potentially several SSTables need to be read
- How to avoid reading all SSTables when not needed?
  - Use of Bloom filters (1970 !)
  - Data structure that allows us to know if a SStable contains an entry for a given key-column pair
- Bloom filter
  - Implements a membership function (is X in the set?)
  - If the bloom filter answers no: it is guaranteed that X is not present
  - If the bloom filter answers yes: the element is in the set with a high probability
  - Good trade-off between accuracy and memory footprint

About bloom filters

- A vector of n bits and k hash functions
- On insert:
  - Compute the k hash values
  - Set the corresponding bits to 1 in the vector
- On lookup:
  - Compute the k hash values
  - Test whether all bits are set to 1

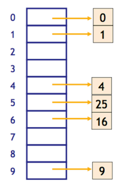


Apache Cassandra

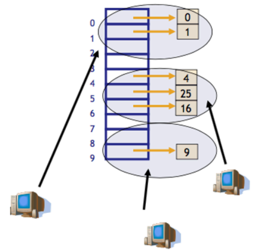
- Column family data store
- Paper published by Facebook in 2010 (A. Lakshman and P. Malik)
  - Used for implementing search functionalities
  - Released as open source
- Build on top of several ideas introduced by BigTable
  - Warning: Many changes in the design have been made sinc... version of Cassandra

Partioning in Cassandra

Ideas from DHT = Distributed Hash Tables

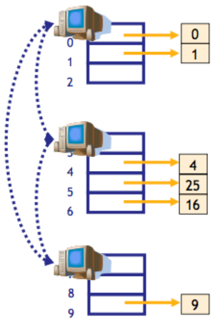


- Hash function: hash(x) = x mod 10
- Insert numbers 0, 1, 4, 9, 16, and 25
- Easy to find if a given key is present in the table



DHT: Principle

- In a DHT, each node is responsible for one or more hash buckets
  - As nodes join and leave, the responsibilities change
- Nodes communicate among themselves to find the responsible node
  - Scalable communications make DHTs efficient
- DHTs support all the normal hash table operations



Lectures of **Prof. Jussi Kangasharju,**
http://www.cs.helsinki.fi/u/jakangas/

Partioning in Cassandra

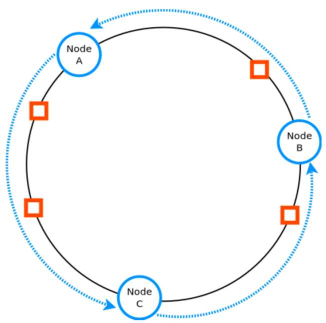
Partitioning based on a hashed name space

- Data items are identified by keys
- Data are assigned to nodes based on a hash of the key
- Tries to avoid hot spots

Namespace represented as a ring

- Allows increasing incrementally the size of the system
- Each node is assigned a random identifier
  - Defines the position of a node in the ring
- The nodes is responsible for all the keys in the range between its identifier and the one of the previous node.

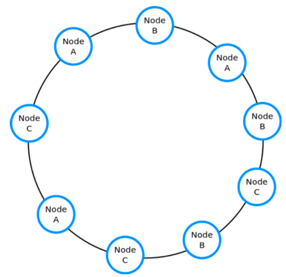
Partioning in Cassandra



- Limits : High risk of imbalance
- Some nodes may store more keys than others
- Nodes are not necessarily well distributed on the ring, especially true with a low number of nodes
- Issues when nodes join or leave the system
- When a node joins, it gets part of the load of its successor
- When a node leaves, all the corresponding keys are assigned to the successor

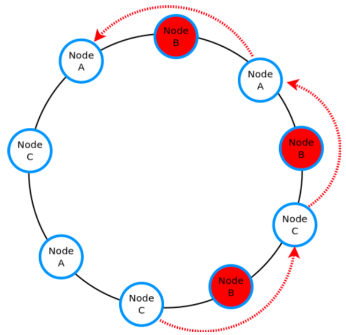
Partitioning and Virtual Nodes

- Concept of virtual nodes
- Assign multiple random positions to each node



The key space is better distributed between the nodes

Partitioning and virtual nodes



If a node crashes, the load is redistributed between multiple nodes

Partitioning and Replication

Items are replicated for fault tolerance.

- Simple strategy
  - Place replicas on the next R nodes in the ring
- Topology-aware placement
  - Iterate through the nodes clockwise until finding a node meeting the required condition
  - For example a node in a different rack

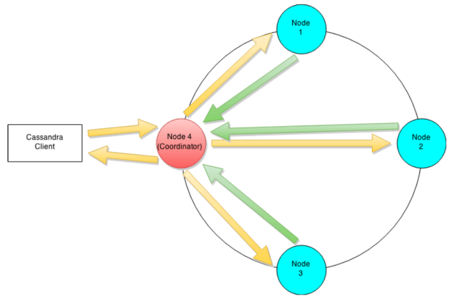
Replication in Cassandra

Replication is based on **quorums**

- A read/write request might be sent to a subset of the replicas
  - To tolerate f faults, it has to be sent to f + 1 replicas
- Consistency
  - The user can choose the level of consistency
  - Trade-off between consistency and performance (and availability)
- Eventual consistency
  - If an item is modified, readers will eventually see the new value

A Read/Write request

Figure from https://dzone.com/articles/introduction-apache-cassandras



- A client can contact any node in the system
- The coordinator contacts all replicas
- The coordinator waits for a specified number of responses before sending an answer to the client

Consistency Levels

ONE (default level)

- The coordinator waits for one ack on write before answering the client
- The coordinator waits for one answer on read before answering the client
- Lowest level of consistency
  - Reads might return stale values
  - We will still read the most recent values in most cases

QUORUM

- The coordinator waits for a majority of acks on write before answering the client
- The coordinator waits for a majority of answers on read before answering the client
- High level of consistency
  - At least one replica will return the most recent value

References

- Bigtable: A Distributed Storage System for Structured Data., F. Chang et al., OSDI, 2006.
- Cassandra: a decentralized structured storage system ., A. Lakshman et al., SIGOPS OS review, 2010.
- http://martin.kleppmann.com/2015/05/11/ please-stop-calling-databases-cp-or-ap.html, M. Kleppmann, 2015.
- https://jvns.ca/blog/2016/11/19/ a-critique-of-the-cap-theorem/,
  J. Evans, 2016.

References

- Lecture notes of V.Leroy
- Lecture notes of F.Zanon Boito
- Lecture notes of T.Ropars
- Lecture notes of B. Groz
- *Designing Data-Intensive Applications* by Martin Kleppmann
  - Chapters 2 and 7