Database Systems Evolution

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Data Systems – some requirements

- **Reliability**: The system should continue performing the correct function at the desired performance,
  - even in the face of adversity (hardware or software faults, and even human error).
- **Scalability**: As the system grows (in data volume, traffic volume or complexity), there should be reasonable ways of dealing with that growth.
- **Maintainability**: Over time, many different people should all be able to work on it productively,
  - Engineering and operations, both maintaining current behavior and adapting the system to new use cases.

Database Models: Beyond RDBMS

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Transactions – ACID Properties

- **Atomic**
  - All of the work in a transaction completes (commit) or none of it completes
- **Consistent**
  - A transaction transforms the database from one consistent state to another consistent state. Consistency is defined in terms of constraints.
- **Isolated**
  - The results of any changes made during a transaction are not visible until the transaction has committed. Concurrent interactions behave as though they occurred serially
- **Durable**
  - The results of a committed transaction survive failures

Impedance Mismatch

- **Object** Orientation
  - based on software engineering principles
- **Relational** Paradigms
  - based on mathematics and set theory
- Mapping from one world to the other has problems

- To store data persistently in modern programs a single logical structure must be split up
  - The nice word is **normalised**

Normalization

- Why IDs (region_id, industry_id, ..) and not plain-text?
  - Consistent style and spelling across profiles,
  - Avoiding ambiguity, e.g. if there are several cities with the same name,
  - The name is stored only in one place, so it is easy to update,
  - Simplify translation into other languages,
- A database in which entities like region and industry are referred to by ID is called **normalized**.
- A database that duplicates the names and properties of entities on each document is **denormalized**.

RDBMS have fundamental issues

- In dealing with (horizontal) scale
  - Designed to work on single, large machines
  - Difficult to distribute effectively
- More subtle: An Impedance Mismatch
  - We create logical structures in memory
    - and then rip them apart to stick it in an RDBMS
  - The RDBMS data model often disjoint from its intended use
    - (Normalisation sucks sometimes)
  - Uncomfortable to program with (joins and ORM etc.)

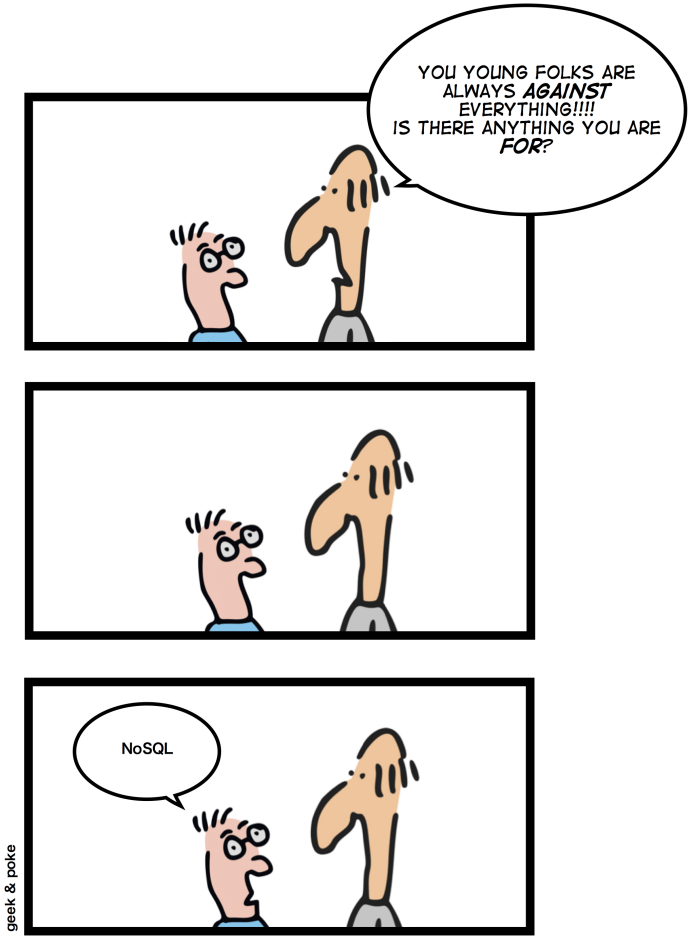
The NoSQL Movement

NoSQL

- The term NoSQL is unfortunate, since it doesn't refer to any technology
  - "Not only SQL"
- Nevertheless, the term struck a nerve, and quickly spread through the web startup community and beyond.
- Several interesting database systems are now associated with the #NoSQL hashtag.

The NoSQL movement

- Key attributes include:
  - **Non-Relational**
    - They can be, but aren't good at it
  - **Simple API**
    - No Join
  - **BASE** & **CAP** Theorem
    - No ACID requirements
  - **Schema-free**
    - Implicit schema, application side
  - **Inherently Distributed**
    - Some more so than others
  - **Open Source**
    - mostly



BASE Transactions

- Acronym contrived to be the opposite of ACID
  - Basic Availability
    - The database appears to work most of the time.
  - Soft-state
    - Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
  - Eventual consistency
    - Stores exhibit consistency at some later point (e.g., lazily at read time).
- Characteristics
  - Optimistic
  - Simpler and faster
  - Availability first
  - Best effort
  - Approximate answers OK

Brewer's CAP Theorem

- A distributed system can support only two of the following characteristics:
  - **Consistent**
    - writes are atomic, all subsequent requests retrieve the new value
  - **Available**
    - The database will always return a value so long as the server is running
  - **Partition Tolerant**
    - The system will still function even if the cluster network is partitioned (i.e. the cluster loses contact with parts of itself)
- The overly stated well cited issue is:
  - We can only ever build an algorithm which satisfies 2 of 3.

Storage and Retrieval

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Hash indexes

- Key-value stores are like a *dictionary* which is usually implemented as a hash map.
- A simple indexing strategy: keep an in-memory hash map where every key is mapped to a byte offset in the data file.
- This is essentially what some key-value databases do (e.g. Bitcask/Riak)
  - they offer high-performance reads and writes, if the hash map is kept completely in memory.

Managing disk space

- How do we avoid running out of disk space?
  - Creating segments (for better performance).
  - Each segment contains all the values written to the database during some period of time.
  - Performing compaction (throwing away duplicate keys in the log).
- The merging and compaction can be done in a background thread.
  - We can continue to serve read and write requests as normal, using the old segment files.

Other issues

- File format
  - CSV is not the best format for a log.
  - A binary format may be used here.
- Deleting records
  - To delete a key, we need to append a special deletion record to the data file (tombstone).
  - When log segments are merged, the process discards any previous values for the deleted key.
- Crash recovery
  - If the system is restarted, the in-memory hash maps are lost.
  - To speed up recovery, we may store a snapshot of each segment's hash map on disk.

Other issues

- Partially written records
  - The database may crash at any time, including halfway through appending a record to the log.
  - Checksums may allow such corrupted parts of the log to be detected and ignored.
- Concurrency control
  - As writes are appended in sequential order, we may have only one writer thread.
  - Data file segments are append-only and immutable, so they can be concurrently read by multiple threads.

Append-only log

- Append-only design seems wasteful at first glance.
  - why don't you update the file in place, overwriting the old value with the new value?
- But, it turns out to be good for several reasons:
  - Appending and segment merging are sequential write operations, which are generally much faster than random writes.
  - Concurrency and crash recovery are much simpler if segment files are append-only or immutable.
- However, it also has limitations:
  - The hash table must fit in memory. It is difficult to make an on-disk hash map perform well.
  - Range queries are not efficient. For example, search all keys between A01 and A99.

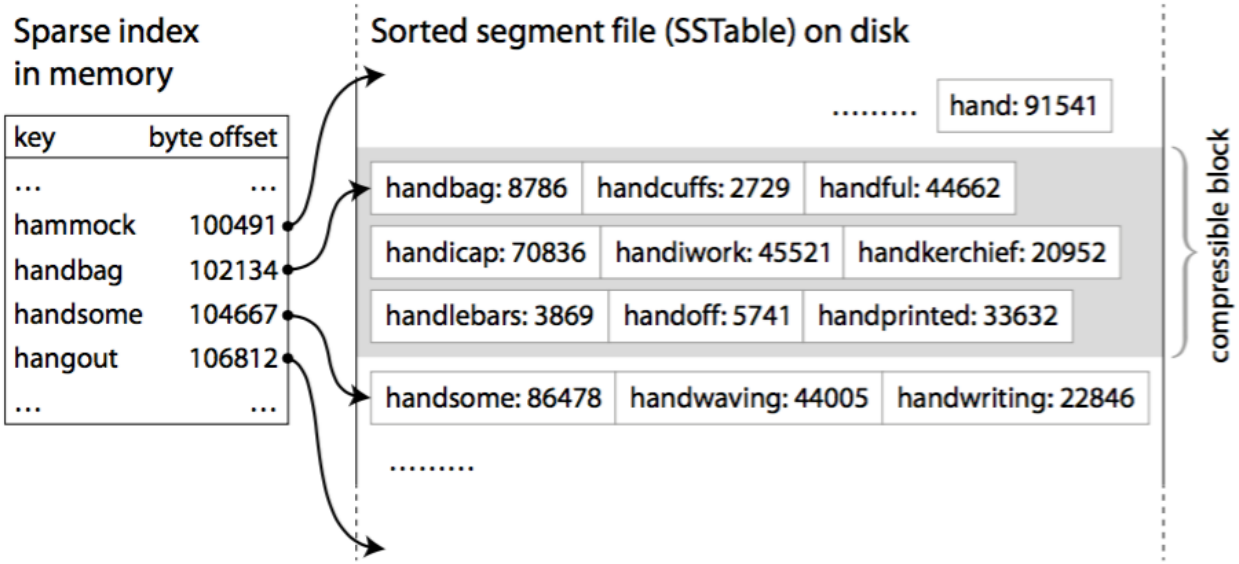
Sorted String Table (SSTable)

- In the previous examples, the key-value pairs appear in the order that they were written.
  - But, we can assure that the sequence of key-value pairs is *sorted by key*.
  - At first glance, that requirement seems to break our ability to use sequential writes.

- We can use *Sorted String Table*, or *SSTable* for short.
- We also require that each key only appears once within each merged segment file (the merging process already ensures that).

SSTable

- SSTables have several advantages over log segments with hash indexes:
  - Merging segments is simple and efficient, even if the files are bigger than the available memory (like the mergesort algorithm).
  - No need to keep an index of all the keys in memory (e.g. just "aa", "ab", "ac", ...).



SSTable – sorting

- When the memtable gets bigger than some threshold, write it out to disk as an SSTable file.
  - This can be done efficiently because the tree already maintains the key-value pairs sorted by key.
  - The new SSTable file becomes the most recent segment of the database.
  - When the new SSTable is ready, the memtable can be emptied.
- To serve a read request:
  - First search the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

SSTable and Log

- This scheme suffers from one problem:
  - if the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost.
- To avoid that problem, we can keep a separate log on disk to which every write is immediately appended.
  - Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

B-trees

- B-tree are the standard index implementation in almost all relational databases and many non-relational databases.
- Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookups and range queries.
- B-trees break the database down into fixed-size blocks or pages, traditionally 4 kB in size, and read or write one page at a time.
  - This corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.

B-trees

- The structure starts at the root page.
- Each page contains *k* keys and *k + 1* references to child pages
  - *k* would typically be in the hundreds.
- Each child is responsible for a continuous range of keys, and the keys in the root page indicate where the boundaries between those ranges lie.

B-trees – operations

- Adding a new key:
  - find the page whose range encompasses the new key, and add it to that page.
  - If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges.
- Updating the value for an existing key:
  - search for the leaf page containing that key, and change the value that page, and write the page back to disk (any references to that page remain valid).
- The tree remains *balanced*
- Time complexity (Search, Insert, Delete)
  - $O(\log_k n)$ for a B-tree with k entries per block and n keys

Update-in-place vs. append-only log

- The basic underlying write operation of a B-tree is to overwrite a page on disk with new data.
  - Log-structured indexes only append to files but never modify files in place.
- Some operations require several different pages to be overwritten.
- If there is a crash after writing only some of the pages, we end up with a corrupted index.
  - e.g. an orphan page which is not a child of any parent.
- To deal with crashes, it is normal to include a *write-ahead log* (WAL, or *redo log*).
  - An append-only file to which every B-tree modification must be written before it can be applied to the tree.

Update-in-place vs. append-only log

- An additional complication of updating pages in-place is that careful concurrency control is required.
  - if multiple threads are going to access the B-tree at the same time, a thread may see the tree in an inconsistent state.
- This is typically done by protecting the tree's data structures with latches (lightweight locks).
- In this case, log-structured approaches are simpler
  - The merging and swapping occur in background without interfering with incoming queries.

B-tree optimizations

- Copy-on-write scheme.
  - To deal with crash recovery, a modified page is written to a different location, and a new version of parent pages in the tree is created, pointing at the new location.
- We can save space in pages by not storing the entire key, but abbreviating it.
  - Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels.
- B-tree variants such as fractal trees borrow some log-structured ideas to reduce disk seeks.

B-tree versus LSM-trees

- LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads.
- A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes.
- In B-trees each key exists in exactly one place in the index, whereas a log-structured storage may have multiple copies in different segments.
  - This makes B-trees attractive in databases that want to offer strong transactional semantics.
  - In many relational databases, transaction isolation is implemented using locks on ranges of keys, and in a B-tree index, those locks can be directly attached to the tree.

Other indexing structures

- It is also very common to have secondary indexes.
  - The main difference is that keys are not unique, i.e. there might be many rows (documents, vertices) with the same key.
- This can be solved two ways:
  - either by making each value in the index a list of matching row identifiers (like a posting list in a full-text index), or
  - by making each key unique by appending a row identifier to it.
- Both B-trees and log-structured indexes can be used as secondary indexes.

Storing values within the index

- The key is the thing that queries search for, but the value could be one of two things:
  - the actual row (document, vertex) in question,
  - a reference to the row stored elsewhere.
- In the reference case, the place where rows are stored is known as a ***heap file***.
  - It avoids duplicating data when multiple secondary indexes are present.
- When updating a value, the heap file approach can be quite efficient.
- The record can be overwritten in-place, if the new value is not larger than the old value.

Storing values within the index

- In some situations, it can be desirable to store the indexed row directly within an index. This is known as a ***clustered index***.
  - The primary key of a table can be a clustered index, and secondary indexes refer to the primary key (rather than a heap file location).
- A compromise between a clustered and a nonclustered index is known as a ***covering index***.
  - It stores some table's columns within the index.
- These indexes can speed up reads, but
  - They require additional storage and overhead on writes.
  - Databases also need additional effort to enforce transactional guarantees, because of the duplication.

Multi-column indexes

- The indexes discussed so far only map a single key to a value.
  - That is not sufficient if we need to query multiple columns of a table (or multiple fields in a document) simultaneously.
- The most common type of multi-column index is called a concatenated index.
  - Combines several fields into one key by appending one column to another.
- This is like a phone book, which provides an index from (*lastname, firstname*) to phone number.
  - This index can be used to find all the people with a particular last name, or all the people with a particular lastname-firstname combination.

Fuzzy indexes

- All the indexes discussed so far assume that we query for exact values of a key, or a range of values of a key with a sort order.
  - How to search for similar keys, such as misspelled words.
- Some data stores (e.g. Lucene) allow searching text within a certain edit distance.
  - Lucene uses a SSTable-like structure for its term dictionary.
  - This structure tells queries at which offset in the sorted file they need to look for a key.
  - This is finite state automaton over the characters in the keys, like a trie, which supports efficient search for words within a given edit distance.

Keeping everything in memory

- For many datasets, it is feasible to keep them entirely in memory.
  - Potentially distributed across several machines.
- This led the development of in-memory databases.
- Some in-memory key-value stores, such as Memcached, are intended for caching use only.
  - Acceptable for data to be lost if a machine is restarted.
- But others aim for durability.
  - by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines.
  - Writing to disk also has operational advantages: files on disk can easily be backed up, inspected and analyzed by external utilities.

In-memory solutions

- The performance advantage of in-memory databases is not due to the fact that they don't need to read from disk.
  - Even a disk-based storage engine may never need to read from disk if you have enough memory.
- Rather, they can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk.
  - Besides performance, they provide data models that are difficult to implement with disk-based indexes.
  - For example, Redis offers a database-like interface to various data structures such as priority queues and sets.
  - By keeping all data in memory, its implementation is comparatively simple.

Transaction Processing and Transaction Analytics

Online Transaction Processing (OLTP)

- In the early days, a write to the database typically corresponded to a commercial **transaction**:
  - making a sale, placing an order with a supplier, paying an employee's salary, etc.
- Despite databases expanded into many other areas, the term *transaction* nevertheless stuck.
  - ***Transaction processing*** just means allowing clients to make low-latency reads and writes.
  - On the other hand, ***batch processing*** jobs run periodically, for example once per day.
- Application are typically interactive (insert, update, search, etc.).
  - This access pattern became known as ***online transaction processing*** (**OLTP**).

Online Analytic Processing (OLAP)

- Analytic queries are often written by business analysts.
  - and feed into reports that help the management of a company make better decisions (business intelligence).
- To differentiate this pattern of using databases from transaction processing, this are has been called ***online analytic processing*** (**OLAP**)

OLTP versus OLAP

Property	OLTP	OLAP
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

- RDBMS and SQL works well for OLTP-type queries as well as OLAP-type queries.
- However, OLAP normally use separate database than OLTP – **data warehouse**.

Data warehousing

- OLTP systems are expected to be highly available and to process transactions with low latency.
- A data warehouse is a separate database that analysts can query without affecting OLTP operations.
  - It typically contains a read-only copy of the data in all the various OLTP systems in the company.
- Data is extracted from OLTP databases, transformed, cleaned up, and then loaded into the data warehouse.
  - This process of getting data into the warehouse is known as *Extract-Transform-Load* (ETL).

Why Separate Data Warehouse?

- Why combine OLTP and OLAP?
  - Data warehouse are commonly relational, because SQL is also a good fit for analytic queries.
  - Many OLTP tools (querying, visualization, etc.).
- Why separate?
  - Different functions and different data:
    - missing data: Decision support requires historical data which operational DBs do not typically maintain.
    - data consolidation: Decision support requires consolidation (aggregation, summarization) of data from heterogeneous sources.
    - data quality: different sources typically use inconsistent data representations, codes and formats which must be reconciled.
- Many vendors now support either transaction processing or analytics workloads.

Schemas for analytics

- Many data warehouses use a ***star schema*** (or dimensional modeling).
- The main entity is the ***fact table***.
  - Each row of the fact table represents an event at a particular time (a purchase, a page view, etc.).
  - Some columns are attributes (e.g. price). Others are references (foreign keys) to other tables (dimension tables).
- A variation of the star template is the ***snowflake schema***
  - The dimensions are broken down into sub-dimensions (more normalized but more complex to work with).

Querying problem

- Problem:
  - We need to read the entire table to process one single column.
  - OLTP databases are typically row-oriented: all the values from one row of a table are stored next to each other.

- Solution:
  - ***Column-oriented storage***

Column-oriented storage

- The idea:
  - don't store all the values from one row together, but store all the values from each column together instead.
  - If each column is stored in a separate file, a query only needs to read and parse those columns.

- The column-oriented storage layout relies on each column containing the rows in the same order.
- To reassemble an entire row *n*, we need to take all the *n* entry from each of the individual column files.

Column compression

- Column-oriented storage often lends itself very well to compression.
- The sequences of values for each column are often repetitive.
- Depending on the data in the column, different compression techniques can be used.
- One technique that is particularly effective in data warehouses is a bitmap encoding.

Column sorting

- Normally, columns are stored by the insert order.
- However, we can choose another order, using one column values.
  - It serves as an indexing mechanism.
- A second column can determine the sort order of any rows which have the same value in the first sorting column.
  - E.g., if **date_key** is the first sort key, and **product_sk** the second, all sales for the same product on the same day are grouped together.
- Another advantage of sorted order is that it can help with compression of columns.

Writing problem

- Column-oriented storage, compression and sorting all help to make OLAP queries faster.

- **What about writing?**
  - As rows are identified by their position within a column, the insertion has to update all columns consistently.
- **What's the best structure?**
- An update-in-place approach, like B-trees use, is not possible with compressed columns.
  - If you wanted to insert a row in the middle of a sorted table, you would most likely have to rewrite all the column files.

Writing to column-oriented storage

- A good solution: LSM-trees (Log Structured Merge)

- All writes first go to an in-memory store, where they are added to a sorted structure, and prepared for writing to disk.
- When enough writes have accumulated, they are merged with the column files on disk, and written to new files in bulk.
- Queries need to examine both the column data on disk and in memory.

Materialized views

- Data warehouse queries often involve an aggregate function (COUNT, SUM, AVG, MIN, ...).
- Aggregates can be used by many different queries.
  - Repeating the data processing.
- Solution?
- Cache these aggregates in a ***materialized view***.
  - When the underlying data changes, a materialized view needs to be updated.
- A ***data cube*** or ***OLAP cube*** is a special case of a materialized view, which is a grid of aggregates grouped by different dimensions.

Data Formats

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Data encoding

- Software applications inevitably **change** over time.
  - In most cases, this also requires a change to data.
  - Old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time.
- For the system to continue running smoothly, we need to maintain compatibility in both directions:
  - **Backward compatibility** - newer code can read data that was written by older code.
  - **Forward compatibility** - older code can read data that was written by newer code. It requires older code to ignore additions made by a newer version of the code.

RDF – Resource Description Framework

- Language for representing information about resources in the World Wide Web
  - + a family of technologies, languages, specifications, …
  - Used in graph databases and in the context of the Semantic Web, Linked Data, …
- Developed by W3C
  - Started in 1997
- Versions: 1.0 and 1.1
- W3C recommendations
  - https://www.w3.org/TR/rdf11-concepts/
    - Concepts and Abstract Syntax
  - https://www.w3.org/TR/rdf11-mt/
    - Semantics

Statements

- RDF is based on the concept that every **resources** can have different **properties** which have **values**.
- Resource - Any real-world entity
  - **Referents** = resources identified by IRI (Internationalized Resource Identifier)
    - E.g. physical things, documents, abstract concepts, …

    `http://db.pt/movies/Marnoto`
    `http://db.pt/terms#actor`
    `mailto:somegirl@nowhere.com`
    `urn:issn:0167-6423`

  - **Values** = resources for literals
    - E.g. numbers, strings, …

Statements

- Example of a **statement** about a web page:

  http://www.example.org/index.html has an author whose name is Pete Maravich.

- A RDF statement is a **triple** that contains a:
  - **Resource**, the **subject** of a statement
  - **Property**, the **predicate** of a statement
  - **Value**, the **object** of a statement
- Several properties for this web page could be:

  http://www.example.org/index.html has an author whose name is Pete Maravich.

  http://www.example.org/index.html has a language which is English.

  http://www.example.org/index.html has a title which is Example_Title.

Protocol Buffers

- Protocol Buffers is a **binary** encoding library that require a **schema** for any data that is encoded.
- Extensible mechanism for serializing structured data
  - Used in communication protocols, data storage, …
- Developed (and widely used) by Google
  - Mostly for server-side communication
- Design goals
  - Language-neutral, platform-neutral
  - **Small**, **fast**, **simple**
- File extension: *.proto
- https://developers.google.com/protocol-buffers/

Key-Value Databases

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Key-Value Databases – Advantages

- Highly fault tolerant – always available.
- Schema-less offers easier upgrade path for changing data requirements
  - (Document stores provide even greater flexibility).
- Efficient at retrieving information about a particular object (bucket) with a minimum of disc operations.
- Very simple data model. Very fast to set up and deploy.
- Great at scaling horizontally across hundreds or thousands of servers.

Key-Value Databases – Advantages

- No requirement for SQL queries, indexes, triggers, stored procedures, temporary tables, forms, views, or the other technical overheads of RDBMS.
- Very high data ingest rates.
  - Favors write once, read many applications.
- Powerful offline reporting with very large data sets.
- Some vendors are offering advanced forms of KVs that approach the capabilities of document stores or column oriented stores.

Key-Value Databases – Disadvantages

- Not suitable for complex applications.
- Not efficient at updating records where only a portion of a bucket is to be updated.
- Not efficient at retrieving limited data from specific records.
  - For example, in an employee database returning only records of employees making between $40K and $60K.
- As the volume of data increases maintaining unique values as keys becomes more difficult
  - Some more complexity in generating character strings that will remain unique over a large set of keys.
- Generally needs to read all the records in a bucket or you may need to construct secondary indexes.

Query Patterns

- Basic **CRUD operations**
  - Only when a key is provided
  - The knowledge of the keys is essential
  - It might even be difficult for a particular database system to provide a list of all the available keys!
- **No searching by value**
  - But we could instruct the database how to parse the values
  - … so that we can fetch the intended search criteria
  - … and store the references within index structures
- **Batch / sequential processing**
  - MapReduce

Other Functionality

- **Expiration** of key-value pairs
  - After a certain interval of time key-value pairs are automatically removed from the database
  - Useful for user sessions, shopping carts etc.
- **Collections** of values
  - We can store not only ordinary values, but also their collections such as ordered lists, unordered sets etc.
- **Links** between key-value pairs
  - Values can mutually be interconnected via links
  - These links can be traversed when querying

- Particular functionality depends on the store.

Riak Key-Value Store

Data Model

- Instance (→ bucket types) → buckets → objects

- **Bucket** = collection of objects (logical, not physical collection)
  - Each object must have a unique key
  - Various properties are set at the level of buckets
    - E.g. default replication factor, read / write quora, …
- **Object** = key-value pair
  - Key is a Unicode string
  - Value can be anything (text, binary object, image, …)
    Each object is also associated with metadata
    - E.g. its content type (text/plain, image/jpeg, …),
    - and other internal metadata as well

Data Model

- How buckets, keys and values should be designed?

- Complex objects containing various kinds of data
  - E.g. one key-value pair holding information about all the actors and movies at the same time
- Buckets with different kinds of objects
  - E.g. distinct objects for actors and movies, but all in one bucket
  - Structured naming convention for keys might help
    - E.g. actor_trojan, movie_medvidek
- Separate buckets for different kinds of objects
  - E.g. one bucket for actors, one for movies

Redis
(REmote DIctionary Service)

Redis Overview

- Redis
  - **In-memory** key-value store
  - Open source, master-slave replication architecture, sharding, high availability, various persistence levels, …
- Developed by Redis Labs
- Implemented in C
- First release in 2009
- Available at http://redis.io/

Redis Overview

- Functionality
  - Standard key-value store
  - Support for structured values (e.g. lists, sets, …)
  - Time-to-live
  - Transactions

- Redis is not just a plain key-value store, but a data structures server, supporting different kind of values.

- Real-world users
  - Twitter, GitHub, Pinterest, StackOverflow, Flicker, …

Data Model

- Structure
  - Instance → databases → objects

- **Database** = collection of objects
  - Databases do not have names, but integer identifiers
- **Object** = key-value pair
  - Key is a string (i.e. any binary data)
  - Values can be…
    - Atomic: string
    - Structured: list, set, ordered set, hash

Data Types

- **String**
  - The only atomic data type
  - May contain any binary data
    (e.g. string, integer counter, PNG image, …)
  - Maximal allowed size is 512 MB
- **List**
  - Ordered collection of strings
  - Elements should preferably be read / written at the head / tail

Data Types

- **Set**
  - Unordered collection of strings
  - Duplicate values are not allowed
- **Sorted set**
  - Ordered collection of strings
  - The order is given by a score (floating number value) associated with each element (from the smallest to the greatest score)
- **Hash**
  - Associative map between string fields and string values
  - Field names have to be mutually distinct

Interface

- **Command line client**
  - redis-cli
- Two modes are available…
- Basic
  - Commands are passed as standard command line arguments
    - E.g. redis-cli PING, redis-cli -n 16 DBSIZE
  - Batch processing is possible as well
    - E.g. cat script.txt | redis-cli
- Interactive
  - Users type database commands at the prompt redis-cli
- **RESP** (REdis Serialization Protocol)

Basic Commands

- **SET** key value
  - inserts / replaces a given string
- **GET** key
  - returns a given string
- **HELP** command
  - Provides basic information about Redis commands
- **CLEAR**
  - Clears the terminal screen
- **FLUSHDB**
  - Deletes all the keys of the currently selected database
- **BGSAVE**
  - Saves the current dataset (on background)

Strings Operations

- **STRLEN** key
  - returns a string length
- **APPEND** key value
  - appends a value at the end of a string
- **GETRANGE** key start end
  - returns a substring Both the boundaries are considered to be inclusive
  - Positions start at 0;
  - Negative offsets for positions starting at the end
- **SETRANGE** key offset value
  - replaces a substring
  - Binary 0 are padded when the original string is not long enough

Counter Operations

- **INCR** key
- **DECR** key
  - Increments / decrements a value by 1
- **INCRBY** key increment
- **DECRBY** key increment
  - Increments / decrements a value by a given amount

Handling Keys

- **EXISTS** key
  - determines whether a key exists
- **KEYS** pattern
  - finds all the keys matching a pattern (*, ?, …)
  - E.g. KEYS *
- **DEL** key …
  - removes a given object / objects
- **RENAME** key newkey
  - changes the key of a given object
- **TYPE** key – determines the type of a given object
  - Types: integer, string, list, set, zset and hash

Volatile Keys

- Keys with limited time to live
  - When a specified timeout elapses, a given object is removed
  - Works with any data type

- **EXPIRE** key seconds
  - Sets a timeout for a given object, i.e. makes the object volatile
  - Can be called repeatedly to change the timeout
- **TTL** key
  - Returns the remaining time to live for a key
- **PERSIST** key
  - Removes the existing timeout

Lists

- **LPUSH** key value
- **RPUSH** key value
  - Adds a new element to the head / tail (Left / Right)

- **LINSERT** key BEFORE | AFTER pivot value
  - Inserts an element before / after another one

- **LPOP** key
- **RPOP** key
  - Removes and returns the first / last element (Left / Right)

Lists

- **LINDEX** key index
  - gets an element by its index
    - The first item is at position 0;
- **LRANGE** key start stop
  - gets a range of elements
- **LREM** key count value
  - Removes a given number of matching elements from a list
    - Positive / negative = moving from head to tail / tail to head
    - 0 = all the items are removed
- **LLEN** key
  - gets the length of a list

Sets

- **SADD** key value …
  - Adds an element / elements into a set
- **SREM** key value …
  - Removes an element / elements from a set
- **SISMEMBER** key value
  - Determines whether a set contains a given element
- **SMEMBERS** key
  - gets all the elements of a set
- **SCARD** key
  - gets the number of elements in a set
- **SUNION / SINTER / SDIFF** key …
  - Calculates and returns a set union / intersection / difference of two or more sets

Hashes

- **HSET** key field value
  - sets the value of a hash field
- **HGET** key field
  - gets the value of a hash field

Batch alternatives

- **HMSET** key field value … …
  - Sets values of multiple fields of a given hash
- **HMGET** key field …
  - Gets values of multiple fields of a given hash

Hashes

- **HEXISTS** key field
  - determines whether a given field exists
- **HGETALL** key
  - gets all the fields and values
- **HKEYS** key
  - gets all the fields in a given hash
- **HVALS** key
  - gets all the values in a given hash
- **HDEL** key field
  - Removes a given field / fields from a hash
- **HLEN** key
  - returns the number of fields in a given hash

Sorted Sets

Basic operations

- **ZADD** key score value … …
  - Inserts one element / multiple elements into a sorted set
- **ZREM** key value …
  - Removes one element / multiple elements from sorted set

Working with score

- **ZSCORE** key value
  - Gets the score associated with a given element
- **ZINCRBY** key increment value
  - Increments the score of a given element

Document Databases

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Document vs. Relational databases

- The main arguments in favour of the document data model are:
  - simpler application code, schema flexibility, and better performance due to locality.

- May be used for data with a document-like structure,
  - i.e. a tree of one-to-many relationships, where typically the entire tree is loaded at once.
  - When splitting a document-like structure into multiple tables can lead to unnecessarily complicated application code.
    - Event logging, content management systems, blogs, web analytics, e-commerce applications, …

Documents for schema flexibility

- The *schemaless* approach is **advantageous if the data is heterogeneous**
  - i.e. the items in the collection don't all have the same structure.
- For example, because:
  - there are many different types of objects, and it is not practical to put each type of object in its own table, or
  - Data structure determined by external systems, over which we have no control, and which may change at any time.
- In situations like these, a schema may hurt more

Documents for schema flexibility

- A document is usually stored as a **single continuous string**, encoded as JSON, XML or a binary variant thereof (such as MongoDB's BSON).
  - If one needs to access the entire document, there is a performance advantage to this storage locality.
- The **locality advantage** only applies if you need large parts of the document at the same time.
  - The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents.
  - On updates to a document, the entire document usually needs to be re-written.
- Recommended to keep documents small.

Document vs. Relational databases

- When not to use Document
  - Set operations involving multiple documents
  - Design of document structure is constantly changing
    - i.e. when the required level of granularity would outbalance the advantages of aggregates
- If the application does use **many-to-many relationships**, the document model becomes less appealing (no joins).
  - We may denormalize the database, or joins can be emulated in application code by making multiple requests to the database.
  - But… the problems of managing denormalization and joins may be greater than the problem of object-relational mismatch.

Document Stores

- Data model
  - **Documents**
    - Self-describing
    - Hierarchical tree structures (JSON, XML, …) – Scalar values, maps, lists, sets, nested documents, …
    - Identified by a unique identifier (key, …)
  - **Collections** – a set of documents
- Query patterns (CRUD)
  - **C**reate, **U**pdate or **D**elete a document
  - **R**ead/retrieve documents according to complex query conditions
  - Extended key-value stores where the value part is examinable

MongoDB Document Database

Data Model

- Structure
  - Instance → databases → collections → documents

- **Database**
  - Set of Collections
- **Collection**
  - Set of Documents, usually of a similar structure
- **Document**
  - MongoDB document = one JSON object
  - Internally stored as BSON
  - Each document…
    - belongs to exactly one collection
    - has a unique identifier `_id`

```
{
   name: "martin",
   age: 22,
   interests: [ sports, CBD ]
}
```

Data Model – Primary Keys

- **_id** is reserved for a primary key
  - Unique within a collection
  - Immutable (cannot be changed once assigned)
  - Can be of any type other than an array
- Possible values
  - Natural identifier (e.g. a key)
    - Must be unique!
  - UUID (Universally unique identifier)
    - 16-byte number (ISO/IEC 11578:1996, RFC 4122)
  - **ObjectId**
    - Special 12-byte BSON type (default option)
    - Small, likely unique, fast to generate, ordered, based on a timestamp, machine id, process id, and a process-local counter

Data Model – Denormalized

- **Embedded documents**
  - Related data in a single structure with subdocuments
  - Suitable for one-to-one or one-to-many relationships
  - Brings ability to read / write related data in a single operation
    - i.e. better performance, less queries need to be issued

```
> db.redwine.insert( {
   winepack: "Dinner",
   bottles: [
      { name: "Cartuxa", year: 2012 },
      { name: "Evel", year: 2010 },
      { name: "EA", year: 2016 }
      ]
   })
```

Data Model – Normalized

References

- Directed links between documents, expressed via identifiers
  - Idea analogous to foreign keys in relational databases
  - Suitable for **many-to-many relationships**
  - Embedding in this case would result in data duplication
- References provide more flexibility than embedding
  - But follow up queries are needed

```
> db.redwine.insert( {
    winepack: "Dinner",
    bottles: [
        { "$id" : "1" },
        { "$id" : "3" }
        ]
    })
```

```
{ _id: "1",
   name: "Cartuxa",
   year: 2012 }
{ _id: "2",
   name: "Evel",
   year: 2010 }
{ _id: "3",
   name: "EA",
   year: 2016 }
```

  - The $id field contains the value of the _id field in the referenced document.

Selection operators

- Comparison
  - **`$eq, $ne`**
    - Tests the actual field value for equality / inequality
  - **`$lt, $lte, $gte, $gt`**
    - Less than / less than or equal / greater than or equal / greater
  - **`$in`**
    - Equal to at least one of the provided values
  - **`$nin`**
    - Negation of `$in`
- Logical
  - **`$and, $or`**
  - **`$nor`**
    - returns all documents that fail to match both clauses.
  - **`$not`**

Selection operators

- Element operators
  - **`$exists`**
    - tests whether a given field exists / not exists
  - **`$type`**
    - selects documents if a field is of the specified type.

- Evaluation operators
  - **`$regex`**
    - tests whether the field value matches a regular expression (PCRE)
  - **`$text`**
    - performs text search (text index must exists)

Selection operators

- Array query operators
  - `$all`
    - Matches arrays that contain all elements specified in the query.
  - `$elemMatch`
    - Selects documents if an element in the array field matches all the specified $elemMatch conditions.
  - `$size`
    - Selects documents if the array field is a specified size.

Indexes

- **Motivation**
  - Full collection scan must be performed when searching for the documents, unless an appropriate index exists
- **Primary** index
  - MongoDB creates a unique index on the *_id* field during the creation of a collection
- **Secondary** indexes
  - Created manually for a given key field / fields
  - To create an index, use *db.collection.createIndex()* or a similar method from your driver.

```
db.<collection>.createIndex(keys, options)
```

  - MongoDB indexes use a B-tree data structure.

Index Types

- **Single** Field
  - Ascending/descending indexes on a single field.
- **Compound** Index
  - Indexes on multiple fields
    - The order of fields listed in a compound index has significance
    - e.g. { userid: 1, score: -1 }, sort by userid ASC and then, by score DESC.
- **Multikey** Index
  - To index a field that holds an array value.
- **Text** Indexes
- **Hashed** Indexes
- **Geospatial** Index

Index Types

- **1, -1** – standard ascending / descending value indexes

```
db.<collection>.createIndex( { field: -1 } )
```

- **hashed** – hash values of a single field are indexed

```
db.<collection>.createIndex( { _id: "hashed" } )
```

- **text** – basic full-text index

```
db.<collection>.createIndex( { comments: "text" } )
```

- **2d** – points in planar geometry

```
db.<collection>.createIndex( { <location field> : "2d" , <additional field> : <value> } , { <index-specification options> } )
```

- **2dsphere** – points in spherical geometry

```
db.<collection>.createIndex( { <location field> : "2dsphere" } )
```

MapReduce

- Data processing paradigm for condensing large volumes of data into useful *aggregated* results.
- Both map and reduce functions are implemented as ordinary JavaScript functions
  - **Map** function: current document is accessible via this, emit(key, value) is used for emissions
  - **Reduce** function: key and array of values are provided as arguments, reduced value is published via return
- Beside others, **query**, **sort** or **limit** options are accepted
  - **out** option determines the output (e.g. a collection name)

Column Databases

UA.DETI.CBD

José Luis Oliveira / Carlos Costa

Concept

- **Store and process data by column instead of row**

- Origin in analytics and business intelligence
  - usually consist of aggregation queries
  - operating in a shared-nothing massively parallel processing architecture to build high-performance applications
- Usually described as “sparse, distributed, persistent multidimensional sorted map”
- Main inspiration for column-oriented datastores is Google’s Bigtable

Columnar databases

- Good for
  - Queries that involve only a few columns
  - Aggregation queries against vast amounts of data
  - Column-wise compression

- Not so good
  - Incremental data loading
  - Online Transaction Processing (OLTP) usage
  - Queries against only a few rows

(Dis)Advantages explained…

↑ Some queries could become really fast
  - aggregation queries
  - function over fields, e.g. average age of users

↑ Better data compression
  - when running the algorithms on each column (similar data)
  - accentuated as your dataset becomes larger

↓ Aggregation is great, but some applications need to show data for each individual record
  - columnar databases are generally not great for these types of queries

↓ Writing new data could take more time
  - inserting a new record into a row-oriented database is a simple write operation
  - updating many values in a columnar database could take much more time

Usage

- Suitable use cases
  - event logging, content management systems, blogs, …
    - i.e. for structured flat data with similar schema
  - batch processing via map reduce

- When not to use
  - ACID transactions are required
  - Complex queries: joining, …
  - Early prototypes
    - i.e. when database design may change

Apache Cassandra

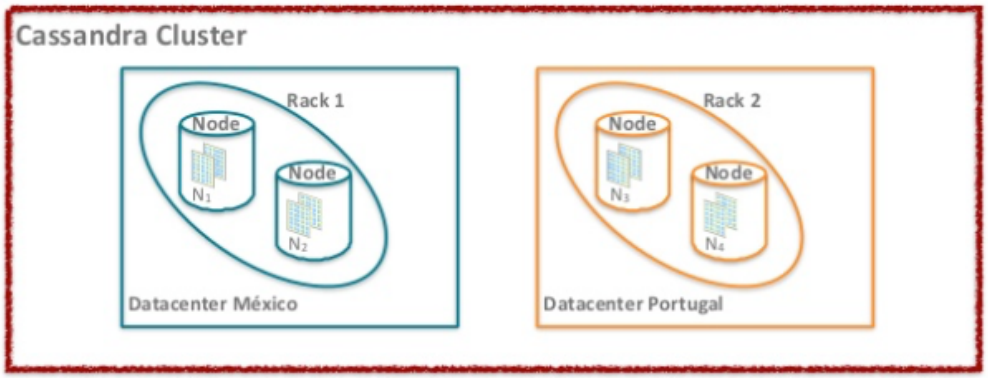
Cassandra Overview

- Features:
  - open-source, high availability, linear scalability, sharding (spanning multiple datacenters), peer-to-peer configurable replication, tunable consistency, MapReduce support
- Implemented in Java
- Cross-platform
- Originally developed by Facebook
  - open-sourced in 2008
- Adopted by Twitter, Rackspace, Netflix, eBay, GitHub, Instagram, etc.
- Developed by Apache Software Foundation
  - http://cassandra.apache.org/

Motivations

- High Availability
- High Write Throughput
  - while not sacrificing read efficiency
- Fault Tolerance
- High and incremental scalability
- Reliability at massive scale

Clusters, Data Centers, Nodes



- Node
  - a machine where Cassandra is running
- Data Center
  - A collection of related nodes
  - Synonymous of replication group
    - replication is set by data center
    - a grouping of nodes configured together for replication purposes
  - Using separate data centers allows:
    - dedicating each data center for different processing tasks
    - satisfying requests from a data center close to client
- Cluster
  - A cluster is a collection of data centers
    - the same data is written in all data center

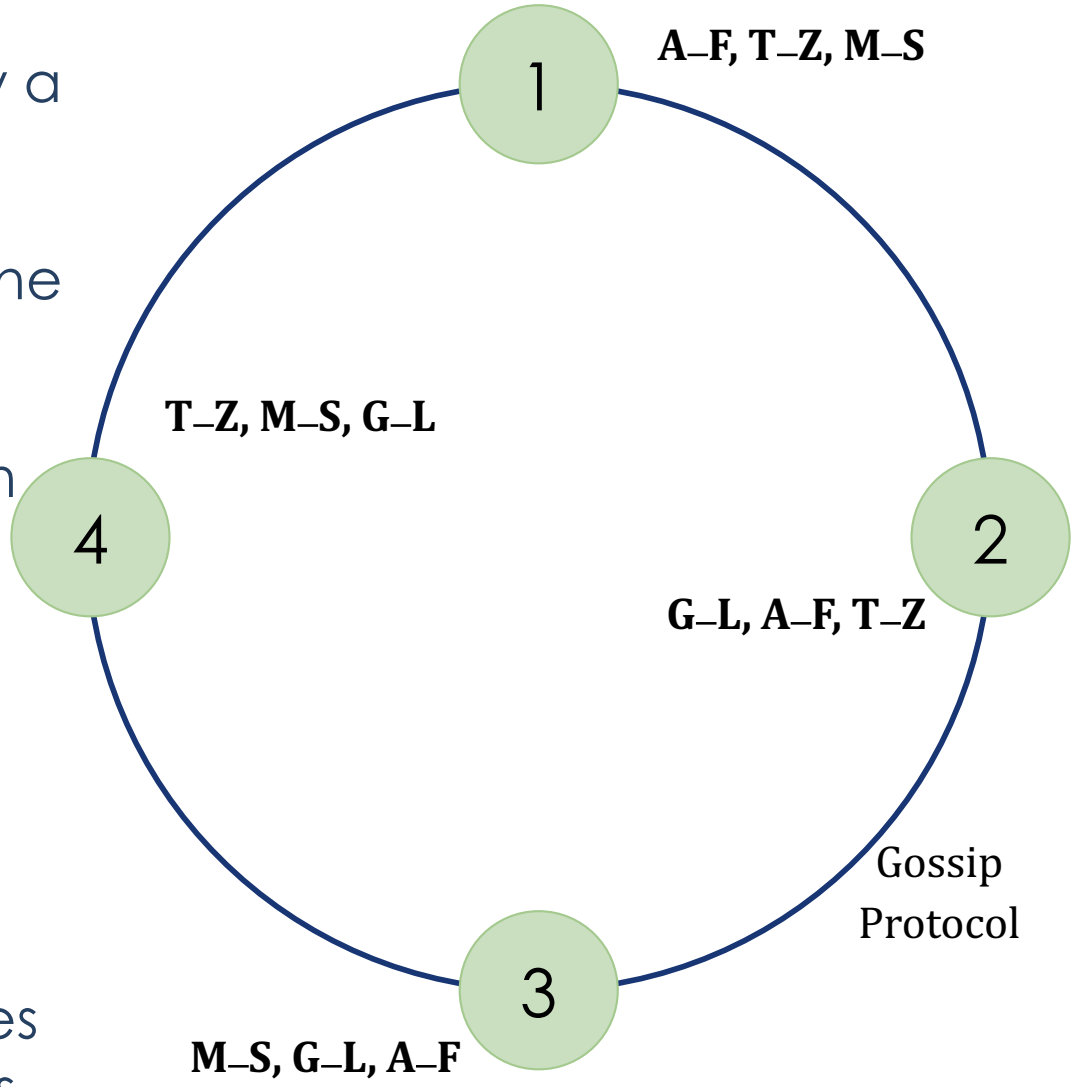
Cassandra – System Architecture

- Cluster Membership
  - how nodes are added, deleted to the cluster
- Partitioning
  - how data is partitioned across nodes
  - nodes are logically structured in Ring Topology
  - hashed value of key associated with data partition is used to assign it to a node in the ring
- Replication
  - how data is duplicated across nodes
  - each data item is replicated at N (replication factor) nodes

Note: Those topics will be discussed more in detail in posterior lessons. In this phase, we will be only focusing in the data model.

Network Nodes Topology

- Cluster Data managed by a ring of nodes
- Each node has a part of the database
- Rows distribution based on primary key
  - row lookups are fast
- Multiples nodes have the same data to ensure both availability and durability
- No master node – all nodes can perform all operations



Data Model (keyspaces → tables → rows → columns)

- Keyspace
  - a **namespace** that defines data replication on nodes
  - a cluster contains one keyspace per node
- Table (column family)
  - collection of (similar) rows
  - a multi dimensional map indexed by key (row key)
  - table schema must be specified but can be modified later
  - 2 types: simple or super (nested Column Families)
- Row
  - collection of columns
  - rows in a table do not need to have the same columns
  - each row is **uniquely identified** by a **primary key**
- Column
  - name-value pair + additional data

Cassandra API

- CQLSH
  - interactive command line shell
  - bin/cqlsh
  - uses CQL (Cassandra Query Language)

- Client drivers
  - provided by the community
  - available for various languages
    - Java, Python, Ruby, PHP, C++, Scala, Erlang, …

Table Primary Key

Primary key has two parts:

- Compulsory **partition key**
  - single column or multiple columns
  - describes how table rows are distributed among partitions
- Optional **clustering columns**
  - defines the clustering order, i.e. how table rows are locally stored within a partition

```
CREATE TABLE groups (
    groupname text,
    username text,
    email text,
    age int,
    PRIMARY KEY (groupname, username)
)
```

PRIMARY KEY has two components: **groupname**, which is the **partitioning key**, and **username**, which is called the **clustering key**. This will give us one partition per groupname. Within a particular partition (group), rows will be ordered by username.

Keys Roles

- **Partition Key**
  - responsible for data partitioning across database nodes
- **Clustering Key**
  - responsible for data sorting within the partition
- **Primary Key**
  - **Partition Key + Clustering Key**
  - is equivalent to the **Partition Key** in a single-field-key table
- **Composite Key**
  - partition and clustering key can have multiple-columns

```
create table mytable (
      k_part_one text,
      k_part_two int,
      k_clust_one text,
      k_clust_two int,
      k_clust_three uuid,
      data text,
      PRIMARY KEY((k_part_one,k_part_two), k_clust_one, k_clust_two, k_clust_three)
  );
```

Select – FROM Clause

- Defines a **single table** to be queried
  - from the current / specified keyspace
  - joining of multiple tables is not possible
- Supports:
  - **distinct** to remove duplicate rows
  - (user-defined) **aggregate functions**
  - * to select all columns; and attributes **alias** (AS)
  - WRITETIME (**timestamp**) and TTL (**time-to-live**) of a column
    - cannot be used in WHERE clause

```
SELECT name, occupation FROM users WHERE userid IN (199, 200, 207);
SELECT JSON name, occupation FROM users WHERE userid = 199;
SELECT name AS user_name, occupation AS user_occupation FROM users;

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
  AND time > '2011-02-03'
  AND time <= '2012-01-01'

SELECT COUNT (*) AS user_count FROM users;
```

```
select videoname, ttl(videoname), writetime(videoname) from videos;

 videoname                  | ttl(videoname) | writetime(videoname)
----------------------------+----------------+----------------------
   Ondas gigantes na barra! |           null |      1509294890781000
           Aviões de papel! |           null |      1509294888607000
```

Select – WHERE Clause

- **Similar syntaxes**: **CQL** and **SQL**

- **Several differences** due the fact that Cassandra is dealing with distributed data and aims to prevent inefficient queries
  - rows are spread around the cluster based on the hash of the partition keys
  - clustering key columns are used to cluster the data of a partition, allowing a very efficient retrieval of rows

- Partition key, clustering and normal columns support different sets of restrictions within the WHERE clause

Select – WHERE: Examples 1

```
CREATE TABLE numberOfRequests (
    cluster text,
    date text,
    datacenter text,
    hour int,
    minute int,
    numberOfRequests int,
    PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

```
/*  Data will be stored like this:
{datacenter: US_WEST_COAST {hour: 0 {minute: 0 {numberOfRequests: 130}} {minute: 1 {numberOfRequests: 125}} …
                                    {minute: 59 {numberOfRequests: 97}}}
                           {hour: 1 {minute: 0 …
*/
```

```
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND datacenter = 'US_WEST_COAST'
    AND hour = 14
    AND minute = 00;
```

```
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour = 14
    AND minute = 00;
```

```
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND hour = 14
    AND minute = 00;
```

```
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour IN (14, 15)
    AND minute = 0;

SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) IN ((14, 0), (15, 0));

-- multi-column IN restrictions can be applied to any set of clustering columns.
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND (datacenter, hour) IN (('US_WEST_COAST', 14), ('US_EAST_COAST', 17))
    AND minute = 0;
```

Select – WHERE: Examples 2

>, >=, <= and < restrictions
Single column slice restrictions are allowed only on the last clustering column being restricted.
Multi-column slice restrictions are allowed on the last set of clustering columns being restricted.

```
-- OK
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour= 12
    AND minute >= 0 AND minute <= 30;

-- OK
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour >= 12;

-- OK
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter > 'US';


-- NOK
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND hour >= 12 AND minute = 0;
```

```
CREATE TABLE numberOfRequests (
    cluster text,
    date text,
    datacenter text,
    hour int,
    minute int,
    numberOfRequests int,
    PRIMARY KEY ((cluster, date), datacenter, hour, minute))
```

```
-- OK
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) >= (12, 0) AND (hour, minute) <= (14, 0)

-- OK
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) >= (12, 30) AND (hour) <= (14)


-- NOK: the restrictions must start with the same column
SELECT * FROM numberOfRequests
    WHERE cluster = 'cluster1'
    AND date = '2015-06-05'
    AND datacenter = 'US_WEST_COAST'
    AND (hour, minute) >= (12, 30) AND (minute) <= (30)
```

Select – WHERE: Secondary Index

- Direct queries on secondary indices support only **=**, **CONTAINS** or **CONTAINS KEY** restrictions

```
CREATE TABLE contacts (
    id int PRIMARY KEY,
    firstName text,
    lastName text,
    phones map<text, text>,
    emails set<text>
);
```

```
select * from contacts
where firstname = 'Maria';
```

```
/*
    Solution: Secondary Index
*/
CREATE INDEX ON contacts (firstName);
-- Using the keys function to index the map keys
CREATE INDEX ON contacts (keys(phones));
CREATE INDEX ON contacts (emails);
```

```
SELECT * FROM contacts WHERE firstname = 'Benjamin';
SELECT * FROM contacts WHERE phones CONTAINS KEY 'office';
SELECT * FROM contacts WHERE emails CONTAINS 'Benjamin@oops.com';
```

Select – Group and Order By Examples

```
CREATE TABLE contacts (
    type text,
    id int,
    firstName text,
    lastName text,
    phones map<text, text>,
    emails set<text>,
    PRIMARY KEY(type, id)
);
```

```
select * from contacts
where id = 33
order by id;
```

```
select * from contacts
where type = 'personal'
and id = 33
order by id;
```

```
select * from contacts
where type = 'personal'
group by type;
```

```
select * from contacts
where type = 'personal'
group by id;
```

only the first record is returned

```
select count(*) from contacts
group by type;
```

```
select count(*) from contacts
group by id;
```

Select – User Defined Functions

- User-Defined Functions (UDF)
  - allow the execution of user-provided code (Java or JavaScript)
  - Statements: CREATE (or REPLACE) /DROP FUNCTION

```
CREATE FUNCTION IF NOT EXISTS akeyspace.fname(someArg int)
    CALLED ON NULL INPUT
    RETURNS text
    LANGUAGE java
    AS $$
        // some Java code
    $$;
```

```
CREATE FUNCTION IF NOT EXISTS div (n counter, d counter)
   CALLED ON NULL INPUT
   RETURNS double
   LANGUAGE java AS '
     return Double.valueOf(n/d);
     ';

select rating_counter, div(rating_total, rating_counter)
from ...
```

Select – Aggregates

- Native
  - COUNT(column), MIN(column), MAX(column), SUM(column) and AVG(column)
- User-Defined Aggregate Function (UDA)
  - creation of custom aggregate functions

1

```
CREATE TABLE team_average (
   team_name text,
   cyclist_name text,
   cyclist_time_sec int,
   race_title text,
   PRIMARY KEY (team_name, race_title,cyclist_name)
);
```

2

```
-- UDF: adds all the race times together and counts the number of entries.
CREATE OR REPLACE FUNCTION avgState ( state tuple<int,bigint>, val int )
CALLED ON NULL INPUT
RETURNS tuple<int,bigint>
LANGUAGE java AS
$$ if (val !=null) {
      state.setInt(0, state.getInt(0)+1);
      state.setLong(1, state.getLong(1)+val.intValue());
      }
   return state; $$
;
```

3

```
-- UDF: computes the average of the values passed to it from the state function
CREATE OR REPLACE FUNCTION avgFinal ( state tuple<int,bigint> )
CALLED ON NULL INPUT
RETURNS double
LANGUAGE java AS
  $$ double r = 0;
     if (state.getInt(0) == 0) return null;
     r = state.getLong(1);
     r/= state.getInt(0);
     return Double.valueOf(r); $$
;
```

4

```
-- UDA: calculate the average
--      value in the column
CREATE AGGREGATE average(int)
SFUNC avgState
STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0,0);
```

5

```
-- Test the function using a select statement
SELECT average(cyclist_time_sec) FROM team_average
WHERE team_name='UnitedHealthCare'
AND race_title='Amgen Tour';
```

Select – ALLOW FILTERING

- Option used to explicitly allow (some) queries that require filtering
- By default, only non-filtering queries are allowed
  - i.e. queries where the number of rows read ~ the number of rows returned
    - such queries have predictable performance
    - execution time that is proportional to the amount of data returned

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
)

CREATE INDEX ON users(birth_year);
```

```
SELECT * FROM users;
SELECT * FROM users WHERE birth_year = 1981;
```

```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR';
```

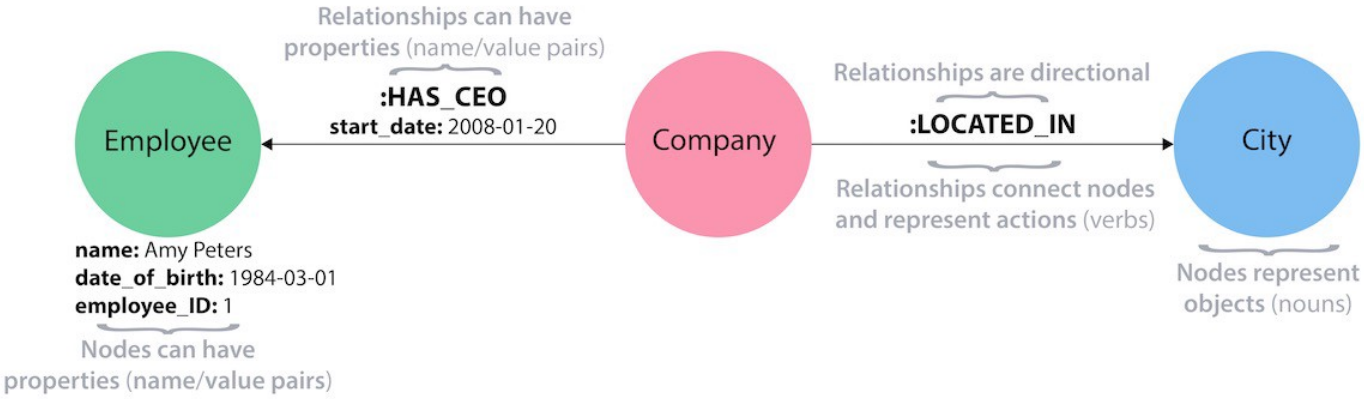
```
SELECT * FROM users WHERE birth_year = 1981 AND country = 'FR' ALLOW FILTERING;
```

Graph Databases

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

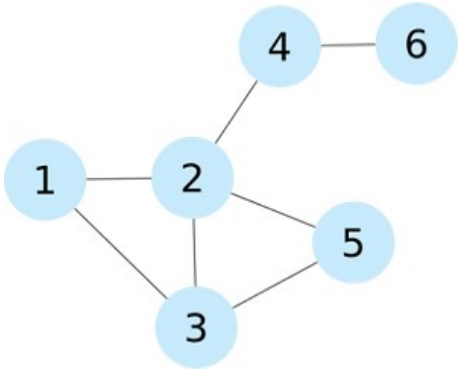
Types of Graphs

- **Single-relational**
  - Edges are homogeneous in meaning
    - e.g., all edges represent friendship
- **Multi-relational** (property) graphs
  - Edges are typed or labelled
    - e.g., friendship, business, communication
  - Vertices and edges maintain a set of key/value pairs
    - Representation of non-graphical data (properties)
    - e.g., name of a vertex, the weight of an edge



Adjacency Matrix

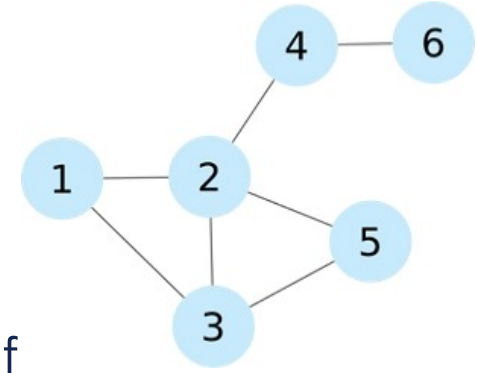
- Bi-dimensional **array** *A* of *n* x *n* Boolean values
  - Indexes = node identifiers
  - *Aij* indicates whether the two nodes *i, j* are connected
- **Pros**:
  - Checking if two nodes are connected
  - Adding/removing edges
- **Cons**:
  - Quadratic space with respect to *n*
    - We usually have sparse graphs (lots of 0)
  - Addition of nodes is expensive
  - Retrieval of all the neighbouring - *O(n)*
- Other variants:
  - Directed graphs, Weighted graphs, …



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Adjacency List

- A **set of lists** where each accounts for the neighbours of one node
  - A vector of *n* pointers to adjacency lists
- Undirected graph:
  - An edge connects nodes *i* and *j* => the list of neighbours of *i* contains the node *j* and vice versa



- **Pros**:
  - Obtaining the neighbours of a node
  - Cheap addition of nodes to the structure
  - Compact representation of sparse matrices
- **Cons**:
  - Checking an edge between two nodes

N1 → {N2, N3}

N2 → {N1, N3, N5}

N3 → {N1, N2, N5}

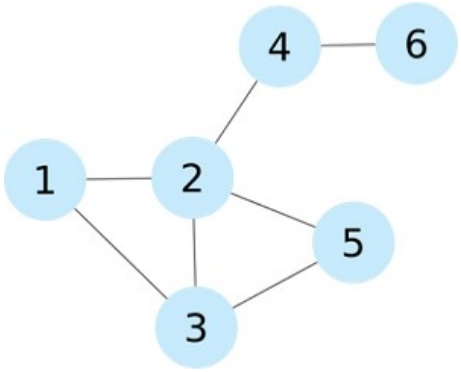
N4 → {N2, N6}

N5 → {N2, N3}

N6 → {N4}

Incidence Matrix

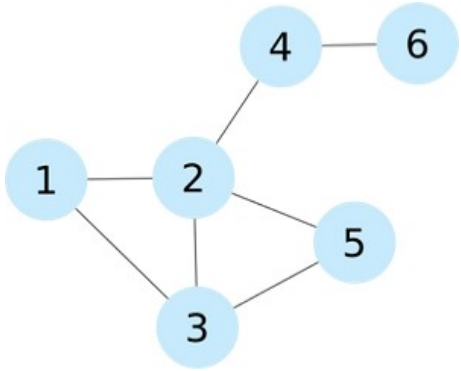
- Bi-dimensional Boolean matrix of *n* rows and *m* columns
  - A **column** represents an **edge**
    - Nodes that are connected by a certain edge
  - A **row** represents a **node**
    - All edges that are connected to the node
- **Pros**:
  - For representing hypergraphs, where one edge connects an arbitrary number of nodes
- **Cons**:
  - Requires *n* x *m* bits



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Laplacian Matrix

- Bi-dimensional **array** of *n x n* integers
  - Diagonal of the Laplacian matrix indicates the **degree** of the node
  - The rest of positions are set to -1 if the two vertices are connected, 0 otherwise
  - **L = D – A**
    - where D is degree matrix of graph G and A is the adjacency matrix



- **Pros & Cons:**
  - = Adjacency Matrix
    - But, it retains more information
    - Allows analyzing the graph structure by means of spectral analysis

$$\begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 3 & 0 & -1 & 0 \\ 0 & -1 & 0 & 2 & 0 & -1 \\ 0 & -1 & -1 & 0 & 2 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$$

Graph Traversals

- Single step **traversal** from element *i* to element *j*, where $i,j \in (V \cup E)$
- Expose explicit **adjacencies** in the graph
  - e_{out} : traverse to the outgoing edges of the vertices
  - e_{in} : traverse to the incoming edges of the vertices
  - v_{out} : traverse to the outgoing vertices of the edges
  - v_{in} : traverse to the incoming vertices of the edges
  - e_{lab} : allow (or filter) all edges with the label
  - $\in$: get element property values for key r
  - e_p : allow (or filter) all elements with the property s for key r
  - $\in=$: allow (or filter) all elements that are the provided element

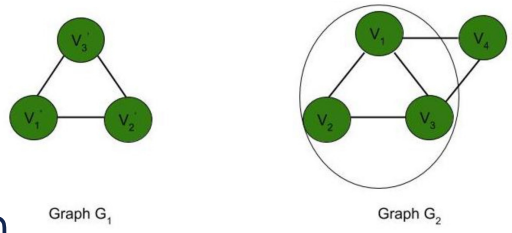
Graph Traversals

- Single step traversals can **compose complex traversals** of arbitrary length
- e.g., find all friends of Alberto
  - Traverse to the outgoing edges of vertex i (representing Alberto),
  - then only allow those edges with the label friend,
  - then traverse to the incoming (i.e. head) vertices on those friend-labelled edges.
  - Finally, of those vertices, return their name property.

$$f(i) = (\in^{name} \circ v_{in} \circ e^{friend}_{lab} \circ e_{out})(i)$$

Transactional Graph Databases

Types of Queries



- **Sub-graph** queries
  - More general type: sub-graph isomorphism
  - Searches for a specific pattern in the graph database
  - A small graph or a graph, where some parts are uncertain
    - e.g., vertices with wildcard labels
- **Super-graph** queries
  - Searches for the graph database members of which their whole structures are contained in the input query
- **Similarity** (approximate matching) queries
  - Finds graphs which are similar, but not necessarily isomorphic to the input query

Graph-oriented Database

Property graphs

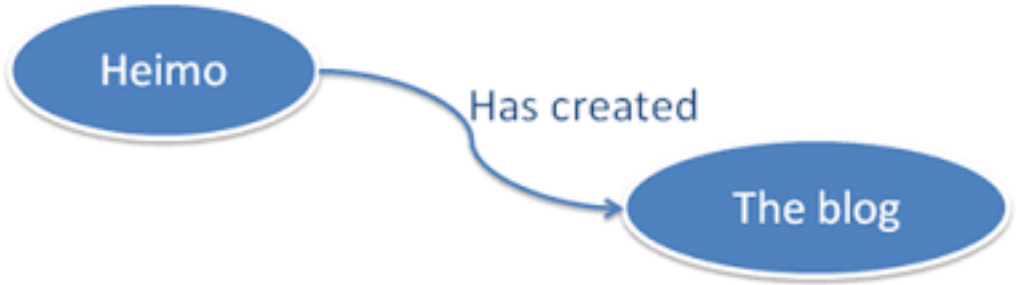
- Each **vertex** consists of:
  - a unique identifier,
  - a set of outgoing edges,
  - a set of incoming edges, and
  - a collection of properties (key-value pairs).
- Each **edge** consists of:
  - a unique identifier,
  - the vertex at which the edge starts (the tail vertex),
  - the vertex at which the edge ends (the head vertex),
  - a label to describe the type of relationship between the two vertices, and
  - a collection of properties (key-value pairs).

Property graphs

- Any vertex can have an edge connecting it with any other vertex.
  - There is no schema that restricts which kinds of things can or cannot be associated.
- Given any vertex,
  - We can efficiently find both incoming and outgoing edges.
  - Traverse the graph.
- Different labels for different kinds of relationship
  - Allow storing several different kinds of information in a single graph, while still maintaining a clean data model.

Triple-stores

- The **triple-store model** is mostly equivalent to the property graph model
  - using different words to describe the same ideas.
- Information is stored in the form of very simple three-part statements:
  - **subject**, **predicate**, **object**.



Triple-stores

- The **subject** of a triple is equivalent to a vertex in a graph.
- The **object** is one of two things:
  - a value in a primitive datatype, such as a string or a number.
    - In that case, the predicate and object of the triple are equivalent to the key and value of a property on the subject vertex.
    - For example, (lucy, age, 33) is like a vertex lucy with properties {"age":33}.
  - another vertex in the graph.
    - In that case, the **predicate** is an edge in the graph, the subject is the tail vertex and the object is the head vertex.
    - For example, in (lucy, marriedTo, alain).

Graph Databases

- **Suitable** use cases
  - Social networks, routing, dispatch, and location-based services,
  - recommendation engines, chemical compounds, biological pathways, linguistic trees, …
  - i.e. simply for graph structures

- **When not** to use
  - Extensive batch operations are required
    - Multiple nodes / relationships are to be affected
  - Only too large graphs to be stored
    - Graph distribution is difficult or impossible at all

Neo4j Graph Database

Features of Neo4j

- **Data model (flexible schema)**
  - Neo4j follows a data model named native **property graph model**.
  - The graph contains **nodes** (entities) and these nodes are connected with each other (depicted by **relationships**). Nodes and relationships store data in key-value pairs known as **properties**.
  - In Neo4j, there is no need to follow a fixed schema.
- **ACID properties**
  - Neo4j supports full ACID (Atomicity, Consistency, Isolation, and Durability) rules.

Features of Neo4j

- **Scalability and reliability**
  - You can scale the database by increasing the volume without affecting the query processing speed and data integrity.
  - Neo4j also provides support for **replication** for data safety and reliability.
- **Cypher Query Language**
  - Neo4j provides a powerful declarative query language known as Cypher.
  - It uses ASCII-art for depicting graphs.
  - Cypher is easy to learn and can be used to create and retrieve relations between data without using the complex queries like Joins.

Features of Neo4j

- **Built-in web application**
  - Neo4j provides a built-in **Neo4j Browser** web application. Using this, we can create and query any graph data.
- **Drivers** – Neo4j can work with
  - It supports two kinds of Java API: Cypher API and Native Java API to develop Java applications.
  - REST API to work with programming languages such as Java, Spring, Scala etc.
  - Java Script to work with UI MVC frameworks such as Node JS.
- **Indexing** – Neo4j supports Indexes by using Apache Lucene.

Data Model

- Database system structure
  - **Instance → single graph**

- Property graph = directed labelled multigraph
  - Collection of vertices (nodes) and edges (relationships)

- Graph **node**
  - Has a unique (internal) identifier
  - Can be associated with a set of **labels**
    - Allow us to categorize nodes
  - Can also be associated with a set of **properties**
    - Allow us to store additional data together with nodes

Data Model

- Graph **relationship**
  - Has a unique (internal) identifier
  - Has a **direction**
    - Relationships are equally well traversed in either direction!
    - Directions can be ignored when querying
  - Always has a start and end node
    - Can be recursive (i.e. loops are allowed)
  - Is associated with exactly one type
  - Can also be associated with a set of **properties**

Data Model

- Node and relationship **properties**
- Key-value pairs
  - Key is a string
  - Value is an atomic value of any primitive data type, or an array of atomic values of one primitive data type
- Primitive **data types**
  - **`boolean`** – boolean values **`true`** and **`false`**
  - **`byte`**, **`short`**, **`int`**, **`long`** – integers (1B, 2B, 4B, 8B)
  - **`float`**, **`double`** – floating-point numbers (4B, 8B)
  - **`char`** – one Unicode character
  - **`String`** – sequence of Unicode characters

Cypher

- Declarative graph query language
  - Allows for expressive and efficient querying and updates
  - Inspired by SQL (query clauses) and SPARQL (pattern matching)

- Clauses
  - E.g. MATCH, RETURN, CREATE, …
  - Clauses are (almost arbitrarily) chained together
  - Intermediate result of one clause is passed to a subsequent one

Cypher – Selection

MATCH

```
MATCH (node1)-[rel:TYPE]->(node2)
RETURN rel.property
```

- Generic format, from node1 to node2.

```
MATCH (n) RETURN n
```

- all nodes

```
MATCH (me:Person) WHERE me.name="My Name" RETURN me.name
MATCH (me:Person {name:"My Name"}) RETURN me.name
```

```
MATCH (movie:Movie)
WHERE movie.title = "Mystic River"
SET movie.released = 2003
RETURN movie.title AS title, movie.released AS released
```

title	released
Mystic River	2003

Cypher – Filtering

- **WHERE**

```
MATCH (tom:Person)-[:ACTED_IN]->()<-[:ACTED_IN]-(actor:Person)
WHERE tom.name="Tom Hanks" AND actor.born < tom.born
RETURN DISTINCT actor.name AS Name
```

  - ??

```
MATCH (gene:Person)-[:ACTED_IN]->()<-[:ACTED_IN]-(other:Person)
WHERE gene.name="Gene Hackman" AND exists( (other)-[:DIRECTED]->() )
RETURN DISTINCT other
```

  - ??

```
MATCH (gene:Person {name:"Gene Hackman"})-[:ACTED_IN]->(movie:Movie),
  (other:Person)-[:ACTED_IN]->(movie),
  (robin:Person {name:"Robin Williams"})
WHERE NOT exists( (robin)-[:ACTED_IN]->(movie) )
RETURN DISTINCT other
```

  - ??

Cypher – Ordering

- **ORDER BY**, **LIMIT**, **SKYP**, **DISTINCT**

- Return the five oldest people in the database

```
MATCH (person:Person)
RETURN person
ORDER BY person.born
LIMIT 5;
```

- List all actors, ordered by age

```
MATCH (actor:Person)-[:ACTED_IN]->()
RETURN DISTINCT actor
ORDER BY actor.born
```

Variable Length Paths

```
MATCH (node1)-[*]-(node2)
```

- Relationships that traverse any depth are:
  `(a)-[*]->(b)`
- Specific depth of relationships
  `(a)-[*depth]->(b)`
- Relationships from one to four levels deep
  `(a)-[*1..4]->(b)`
- Relationships of type KNOWS at 3 levels distance:
  `(a)-[:KNOWS*3]->(b)`
- Relationships of type KNOWS or LIKES from 2 levels distance:
  `(a)-[:KNOWS|:LIKES*2..]->(b)`

Aggregation

- Cypher provides support for a number of aggregate functions
  - **count(x)** Count the number of occurrences
  - **min(x)** Get the lowest value
  - **max(x)** Get the highest value
  - **avg(x)** Get the average of a numeric value
  - **sum(x)** Sum up values
  - **collect(x)** Collect all the values into an collection

```
MATCH (person:Person)-[:ACTED_IN]->(movie:Movie)
RETURN person.name, count(movie)
ORDER BY count(movie) DESC
LIMIT 10;
```

- Top ten actors who acted in more movies

Parallel and Distributed Databases

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Biggest Database Problem

- Large volume of data ⇒ use disk and large memory
- **Bottlenecks**
  - Speed(disk) << speed(RAM) << speed(microprocessor)
- Evolution
  - Processor speed growth (with multicore): 50 % per year
  - DRAM capacity growth: 4 × every three years
  - Disk throughput: 2 × in the last ten years
- **Biggest bottleneck: I/O**
- Solution to increase the I/O bandwidth
  - parallel data access
  - data partitioning

Parallel and Distributed DBMS

- **Parallel** database (Parallel DBMS)
  - A "Centralized" DB with multiple resources such as CPUs and disks in parallel.
    - It supports parallel operations such as query processing and data loading.

- **Distributed** databases (DDBMS)
  - Data is stored in multiple places (each is running a DBMS)
  - New notion of distributed transactions
  - DBMS functionalities are now distributed over many machines

Why Parallel Databases?

- Processing 1 Terabyte?
  - at 10MB/s => ~1.2 days to scan
  - 1000 x parallel => 1.5 minute to scan
- **Divide a big problem** into many smaller ones to be solved in parallel
- Data may be stored in a distributed fashion
  - But the distribution is governed solely by **performance** considerations.
- **Large-scale parallel database** systems increasingly used for:
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing

Why Distributed Databases?

- **Scalability**
  - If data volume, read load or write load grows bigger than a single machine can handle, you can potentially **spread the load across multiple machines**.
- **Fault tolerance / High availability**
  - Multiple machines can provide **redundancy**. When one fails, another one can take over.
- **Latency**
  - Applications are by nature distributed. With users around the world, and DB servers at various locations worldwide, **users can be served from a closer datacenter**.

Parallel Databases

- Parallel databases improve processing and I/O speeds by using **multiple CPUs and disks in parallel**
  - data can be partitioned across multiple disks
  - each processor can work independently on its own partition
- Exploit the parallelism in data management in order to deliver **high-performance**, **high-availability** and **extensibility**
  - support very large databases with very high loads
- Different **queries** can be **run** in **parallel.**

Parallel vs Distributed Databases

Although the **basic principles** of **parallel DBMS** are the **same** as in **distributed DBMS**, the **techniques are** fairly **different**

typically...

Parallel DB

- Fast interconnect
- Homogeneous software
- High performance is goal
- Transparency is goal

Distributed DB

- Geographically distributed
- Data sharing is goal
- Disconnected operation possible

Parallel vs Distributed Databases

- **Distributed processing can use parallel processing**
  - Parallel processing on a single machine (not the opposite)

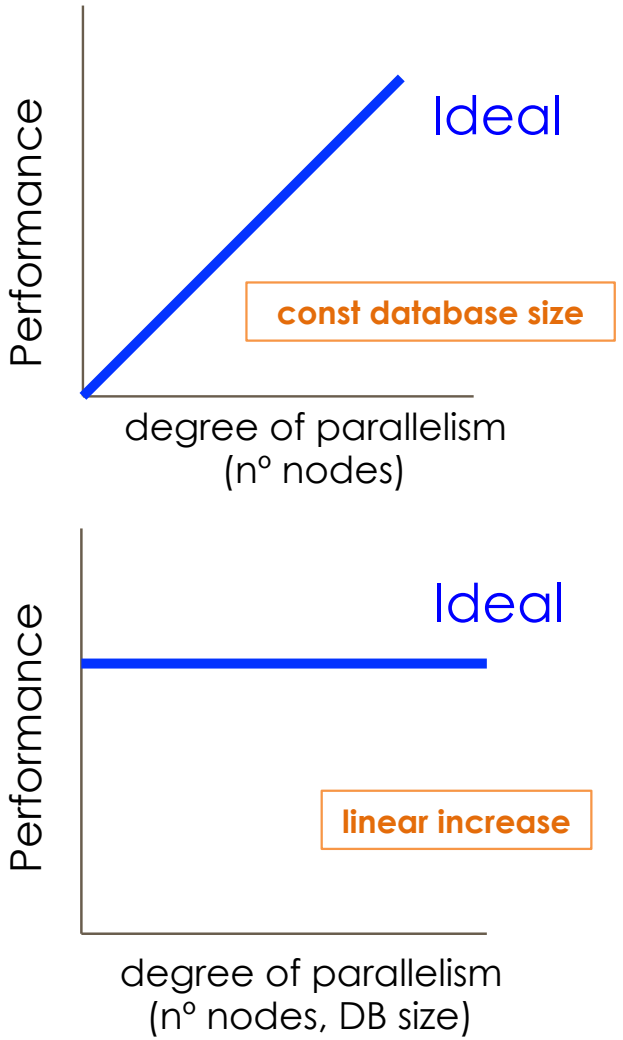
- **Parallel** Databases
  - Machines are physically close (e.g. same server room)
  - .. and connects with dedicated high-speed LANs
  - Communication cost is assumed to be small
  - Architecture: can be **shared-memory, shared-disk or shared-nothing**
- **Distributed** Databases
  - Machines can be in distinct geographic locations
  - .. and connected using public-purpose network, e.g., Internet
  - Communication cost and problems cannot be ignored
  - Architecture: usually **shared-nothing**

Parallel DBMS – Main Goals

- **High-performance** through parallelization of various operations
  - **High throughput** with inter-query parallelism
  - **Low response time** with intra-operation parallelism
  - **Load balancing** is the ability of the system to divide a given workload equally among all processors
- **High availability** by exploiting data replication
- **Extensibility** with the ideal goals
  - Linear speed-up
  - Linear scale-up

Ideal Extensibility Scenario

- **Speed-Up**
  - refers to a **linear increase** in **performance** for a **constant database size** while the **number of nodes** (i.e. processing and storage power) are **increased linearly**
  - more resources means proportionally less time for given amount of data
- **Scale-Up**
  - refers to a **sustained performance** for a **linear increase** in both **database size** and **number** of **nodes**
  - if resources increased in proportion to increase in data size, time is constant



Barriers to Parallelism

- **Startup**
  - The time needed to start a parallel operation may dominate the actual computation time
- **Interference**
  - When accessing shared resources, each new process slows down the others (hot spot problem)
- **Skew**
  - The response time of a set of parallel processes is the time of the slowest one

- Parallel data management techniques intend to overcome these barriers

Database Architectures

… to scale to higher loads:

- **Multiprocessor architecture**
  - Shared memory (SM)
  - Shared disk (SD)
  - Shared nothing (SN)

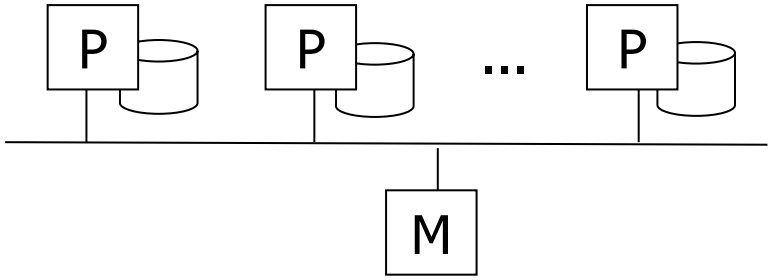
Also called vertical scaling (or scaling up)
Simplest approach - buy a more powerful machine

Aka horizontal scaling (or scaling out)

- **Hybrid architectures**
  - Non-Uniform Memory Architecture (NUMA)
  - Cluster (or hierarchical)

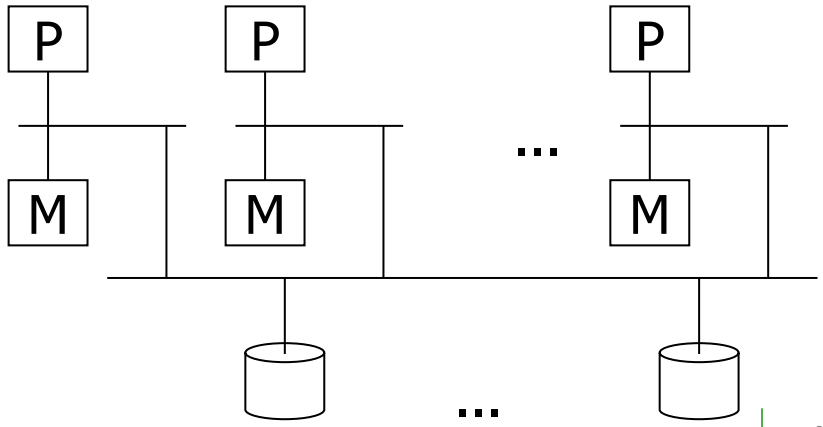
Shared Memory

- Multiple processors share the main memory (RAM) space, but each processor has its own disk (HDD)
  - provide communications among them and avoid redundant copies
- **Bottlenecks**
  - cost is super-linear: a machine with twice resources (CPU, RAM, disk) typically costs significantly more than twice
  - a machine twice the size cannot necessarily handle twice the load
  - offer limited fault tolerance



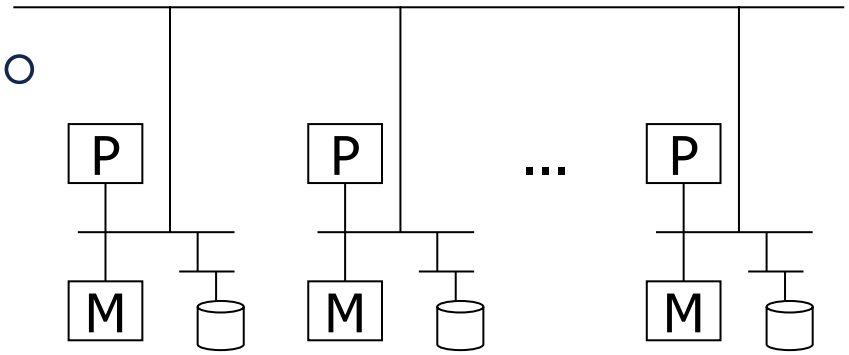
Shared Disk

- Uses several machines with independent CPUs and RAM, but **stores data on an array of disks** that is shared between the machines, connected via a fast network
- Used for some data warehousing workloads
- **Advantages** over shared memory
  - each processor has its own memory - is not a bottleneck
  - a simple way to provide a degree of fault tolerance.
- **Disadvantages**
  - I/O contention
  - limited scalability



Shared Nothing

- Each machine running the database software (**node**) uses its CPUs, RAM and disks independently
- Any coordination between nodes is done at the software level, using a conventional network
- Most common architecture nowadays
- **Advantages**:
  - best price/performance ratio
  - extensibility
  - availability
  - reduce latency



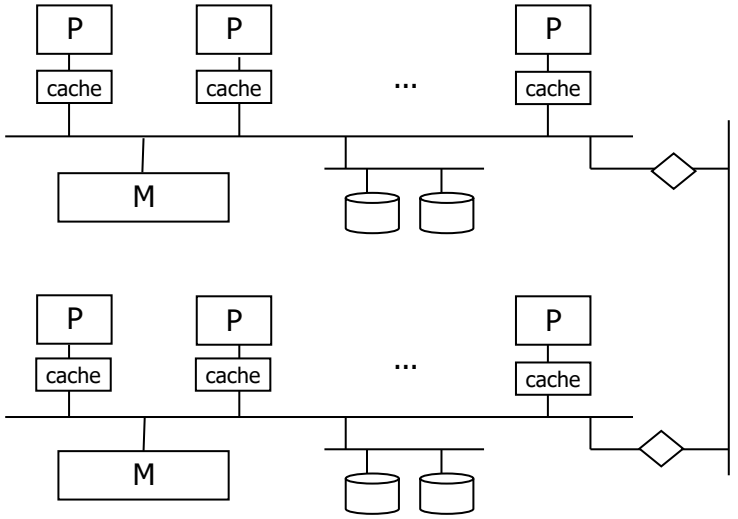
- **Disadvantages**:
  - complexity, difficult load balancing

NUMA (non-uniform memory access)

- Shared Memory vs. Distributed Memory
  - mixes two different aspects:
    - addressing: single address space *and* multiple address spaces
    - physical memory: central *and* distributed
- NUMA uses **single address space on distributed physical memory**
  - eases application portability
  - extensibility
- Cache Coherent NUMA (CC-NUMA)
  - the most successful

CC-NUMA

- Principle: main **memory distributed** as with shared-nothing. However, any **processor has access to all other processors**' memories
  - remote memory access very efficient, only a few times (typically between 2 and 3 times) the cost of local access
- Different processors can access the same data in a conflicting update mode
  - a global cache consistency protocols are needed



Parallel & Distributed DBMS Techniques

- **Data placement**
  - Physical placement of the DB onto multiple nodes
  - Static vs. Dynamic
- **Parallel data processing algorithms**
  - Select is easy
  - Join (and all other non-select operations) is more difficult
- **Parallel query optimization**
  - Choice of the best parallel execution plans
  - Automatic parallelization of the queries and load balancing
- **Distributed Transaction management**

Distributed Data Storage

Two common ways of distribute data across nodes:

- **Partitioning**
  - splitting a big database into smaller subsets called partitions
  - different partitions can be assigned to different nodes
- **Replication**
  - keeping a copy of the same data on several different nodes; potentially in different locations
  - provides redundancy; if some nodes are unavailable, the data can still be served from the remaining nodes
  - can also help improve performance
- Replication and Partitioning can be combined

I/O Parallelism

- In horizontal partitioning, tuples of a relation are divided among many disks

Partitioning techniques (number of disks = n):

- **Round-robin**
  - send the i^{th} tuple inserted in the relation to the disk: i mod n.
- **Hash partitioning**
  - apply a hash function to one or more attributes that range 0…n - 1
- **Range partitioning**
  - associates a range of key attribute(s) to every partition

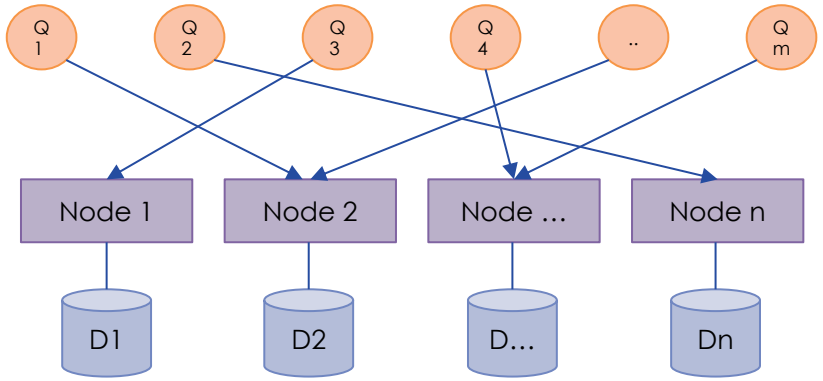
Comparison of Partitioning Techniques

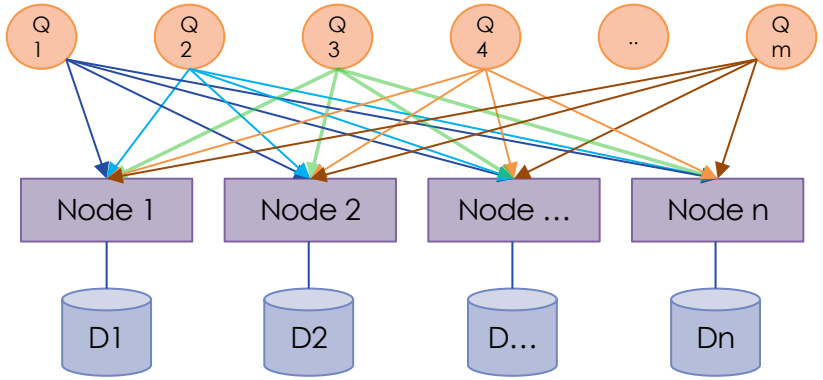
- Evaluate how well partitioning techniques support the following types of data access:
  - Scanning the entire relation.
  - Locating a tuple associatively – point queries.
    - E.g., r.A = 25.
  - Locating all tuples such that the value of a given attribute lies within a specified range – range queries.
    - E.g., $10 \leq r.A < 25$.

	Round Robin	Hashing	Range
Sequential Scan	Best/good parallelism	Good	Good
Point Query	Difficult	Good for hash key	Good for range vector
Range Query	Difficult	Difficult	Good for range vector

Query Parallelism

- **Interquery** Parallelism
  - Parallel execution of multiple queries generated by concurrent transactions
- **Intraquery** Parallelism
  - Split the execution of a single query in parallel on multiple nodes





Interquery Parallelism

- To **increase** the **transactional throughput**
  - used primarily to scale up a transaction processing system to support a **larger number of transactions per second**
- **Easiest** form of parallelism to support
  - particularly in a **shared-memory** parallel database
- More **complicated** on **shared-disk** or **shared-nothing**
  - locking and logging must be coordinated by passing messages between processors
  - data in a local buffer may have been updated at another processor
  - cache-coherency has to be maintained: reads and writes of data in buffer must find latest version of data

Intraquery Parallelism

- The **same query** is **executed** by **many processors**, **each** one **working** on a **subset** of the **data**
  - for speeding up long-running queries
- Two complementary forms of intraquery parallelism:
  - **Intra-operation**:
    - break up a query into multiple parts within a single database partition and execute these parts at the same time.
  - **Inter-operation**:
    - break up a query into multiple parts across multiple partitions of a partitioned database on a single server or between multiple servers
- Intra-operation scales better with increasing parallelism because the number of tuples processed by each operation is typically more than the number of operations in a query

Parallel Data Processing

The following discussion assumes:

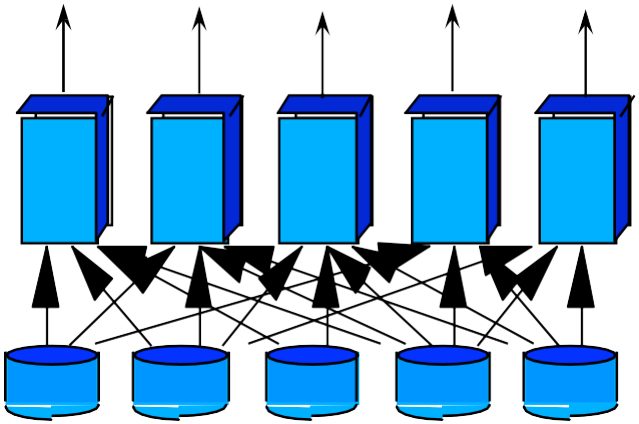
- **Read-only** queries
- **Shared-nothing** architecture
  - shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems
- ***n* processors** (P_0, ..., P_{n-1}) and *n* disks (D_0, ..., D_{n-1}) where disk D_i is associated with processor P_i
  - if a processor has multiple disks they can simply simulate a single disk D_i
- We will focus on **sort**, **select** and **join** operators
  - other binary operators (such as union) can be handled in similar way to join

Parallel Selection - $\sigma_c(R)$

- Relation **R** is **partitioned** over **m machines**
  - each partition of R is around |R|/m tuples
- **Each machine scans** its own **partition** and applies the **Selection** condition **c**
- Data Partitioning impact:
  - **round robin** or a **hash function** (over the entire tuple)
    - relation is expected to be well distributed over all nodes
    - ➢ **all partitions will be scanned**
  - **range** or hash-based (on the selection column)
    - relation can be clustered on few nodes
    - ➢ **few partitions need to be touched**
-

Parallel Sorting

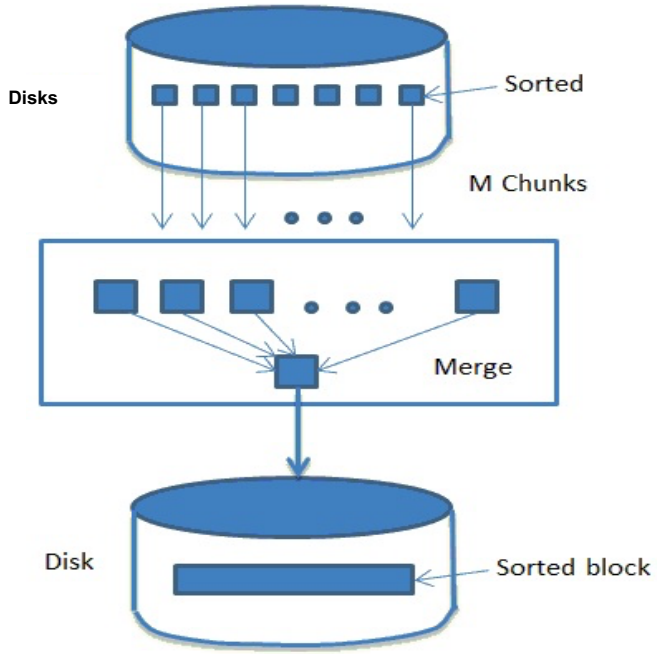
1. **Range-Partitioning Sort**
   - **Choose processors** P_0, ..., P_m, where $m \leq n - 1$ **to do sorting**
   - **Re-partition R based on ranges (on the sorting attributes) into m partitions**
     - this step **requires I/O** and **communication overhead**
   - Machine *i* receives all i^{th} partitions from all machines and sort that partition, without any interaction with the others
     - P_i stores the tuples it received temporarily on disk D_i
   - **Final merge** operation is trivial
     - range-partitioning ensures that, for $i < j < m$, the key values in processor P_i are all less than the key values in P_j
   - Skewed data is an issue
     - ranges can be of different width
     - apply sampling phase first



Parallel Sorting (Cont.)

2. Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks D_0, ..., D_{n-1} (in whatever manner).
- **Each node sorts its own data**
- **All nodes** start **sending** their **sorted data** (one block at a time) to a **single machine**
- This **machine** applies **merge-sort** technique as data come



Parallel Join

- The join operation **requires pairs** of **tuples** to be **tested** to see if they satisfy the join condition
  - If tuples satisfy the join condition, the pair is added to the join output
- Steps…

1. **Parallel join algorithms** attempt to **split the pairs-testing** over **several processors**
2. **Each processor** then **computes part** of the **join** locally
3. **Results** from each processor are **collected** together to **produce** the **final result**

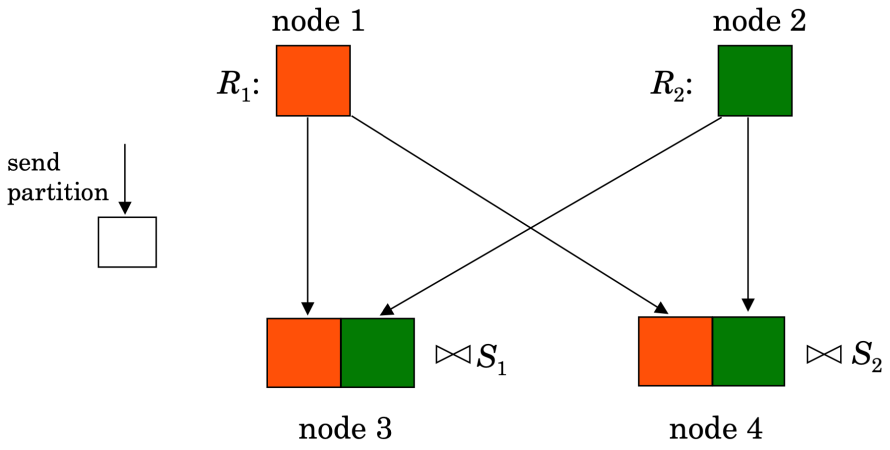
Join Algorithms

- Three basic parallel join algorithms for partitioned databases
  - **Parallel Nested Loop** (PNL)
  - **Parallel Associative Join** (PAJ)
  - **Parallel Hash Join** (PHJ)
- All previous **algorithms** are **intra-operator parallelism**
- They also apply to other complex operators such as duplicate elimination, union, intersection, etc. with minor adaptation

- Next Examples:
  - join of two **relations *R*** and ***S*** that are partitioned over ***m*** and ***n* nodes**, respectively

Parallel Nested Loop Join

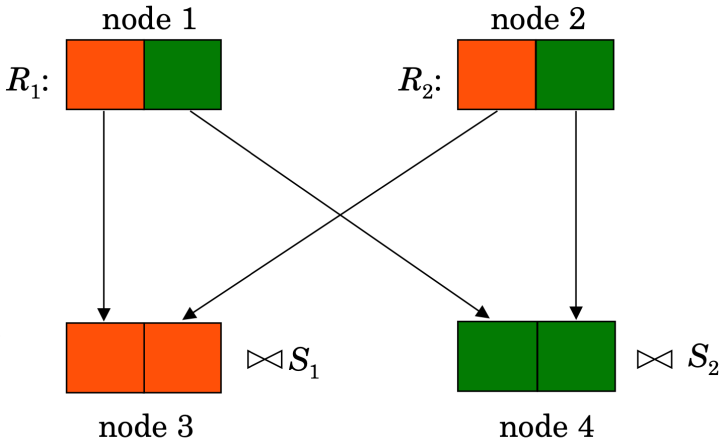
- **Cartesian product** *R* x *S* of relations ***R*** and ***S,*** in parallel.
  - **Simplest** and most **general** method
- **Algorithm phases**:
  1. each fragment of *R* (outer relation) is send and compared to each fragment of *S* (inner relation)
     - this phase is done in parallel by *m* nodes
  2. each *S*-node *j* receives relation *R* entirely, and locally joins *R* with fragment *Sj*.
     - join processing may start as soon as data are received
- **Optimization**:
  - R<<S
  - If S has **index** on the join attribute at each partition, this is pretty fast



$$R \bowtie S \rightarrow \cup_{i=1,n}(R \bowtie S_i)$$

Parallel Associative Join

- Applies **only** to **equijoin** with **one** of the **operand relations partitioned** according to the **join attribute**
- Assume
  - **equijoin** predicate is on **attribute *A*** from ***R***, and ***B*** from ***S***
  - ***S*** *is* **partitioned** according to **hash function applied** to attribute ***B***
    - tuples of *S* that have the same *h*(*B*) value are placed at the same node
  - **no knowledge** of how ***R*** is **partitioned**
- Algorithm phases:
  1. relation ***R*** is **sent** associatively to the ***S*-nodes based** on the **hash function** *h* applied to **attribute *A***
  2. **each *S*-node** *j* **receives** in parallel from the different *R*-nodes the relevant subset of *R* (i.e., *Rj*) and **joins** it **locally** with the fragments *Sj*

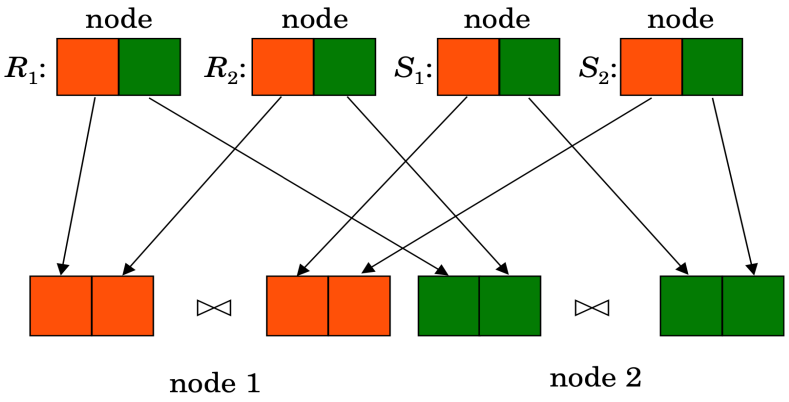


$$R \bowtie S \rightarrow \cup_{i=1,n}(R_i \bowtie S_i)$$

Parallel Hash Join

- Generalization of parallel associative join algorithm
  - Does not require any specific partitioning of the operand relations
- **Basic idea**:
  - **partition** of ***R*** and ***S*** into the same number distinct ***p* fragments**
    - *R*1,*R*2,...,*Rp*, and *S*1,*S*2,...,*Sp*,
  - The ***p* nodes** may be **selected** at **run time** based on the load of the system
- **Algorithm phases**:
  1. *build*: **hashes *R*** on the **join attribute**, **sends** it to the **target *p* nodes** that build a hash table for the incoming tuples
  2. probe: **sends *S* associatively** to the **target *p* nodes** that probe the hash table for each incoming tuple
  3. join each p node and merge all

$$R \bowtie S = \bigcup_{i=1}^{p} (R_i \bowtie S_i)$$



Parallel Query Optimization

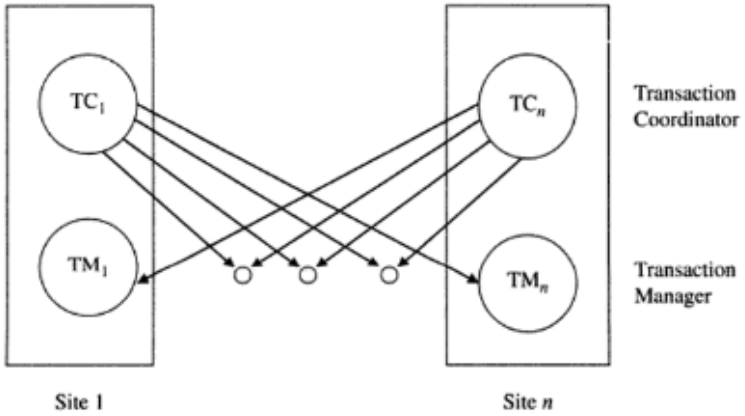
- The **objective** is to **select the "best" parallel execution plan** for a query using the following components:
  - Search space
    - Models alternative execution plans as operator trees
    - Left-deep vs. Right-deep vs. Bushy trees
  - Search strategy
    - Dynamic programming for small search space
    - Randomized for large search space
  - Cost model (abstraction of execution system)
    - Physical schema info. (partitioning, indexes, etc.)
    - Statistics and cost functions
- **Target**: **minimize** the **movement of data** among machines

Load Balancing

- **Balancing** the **load** of **different transactions** and **queries among different nodes** is essential to **maximize throughput**
- **Problems** arise for **intra-operator parallelism** with ***skewed* data distributions**
  - attribute data skew (AVS)
    - inherent to dataset (e.g., there are more citizens in Paris than in Aveiro).
  - tuple placement skew (TPS)
    - introduced when the data are initially partitioned (e.g., with range partitioning)
  - selectivity skew (SS)
    - introduced when there is variation in the selectivity of select predicates on each node
  - redistribution skew (RS)
    - occurs in the redistribution step between two operators (similar to TPS)
  - join product skew (JPS)
    - occurs because the join selectivity may vary between nodes
- **Solutions**
  - sophisticated parallel algorithms that deal with skew
  - dynamic processor allocation (at execution time)

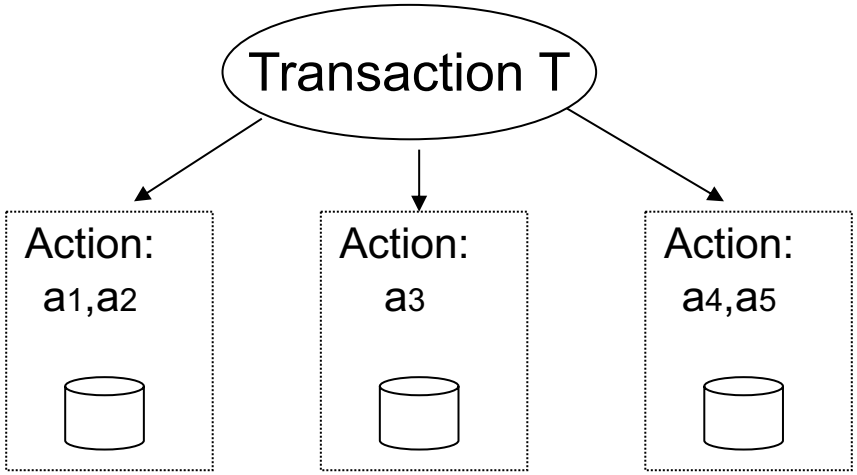
Distributed Transactions

- Transaction may access data at several sites
- Each site has a **transaction coordinator** for:
  - **starting** the execution of every transactions at that site
  - **Breaking transaction and distributing** sub-transactions
  - coordinating the **termination** of each transaction
    - may result in the transaction being committed or aborted at all sites
- Each site has a **transaction manager** responsible for:
  - maintaining a log for recovery purposes
  - coordinating the concurrent execution of the transactions executing at that site



Distributed commit problem

- **Commit** must be **atomic**…
- How a distributed transaction that has components at several sites can execute atomically?



- **Solution**: Two-phase commit (2PC), Centralized 2PC, Distributed 2PC, Linear 2PC, etc.

Example: Two-phase commit protocol

- **First phase** - **coordinator collecting a vote** (commit or abort) **from each participant**
  - Participant stores partial results in permanent storage before voting
- **Second phase** - **coordinator** makes a **decision**
  - **if** all participants want to commit and no one has crashed, coordinator multicasts "commit" message
    - everyone commits
    - if participant fails, then on recovery, can get commit msg from coordinator
  - **else** if any participant has crashed or aborted, coordinator multicasts "abort" message to all participants
    - everyone aborts

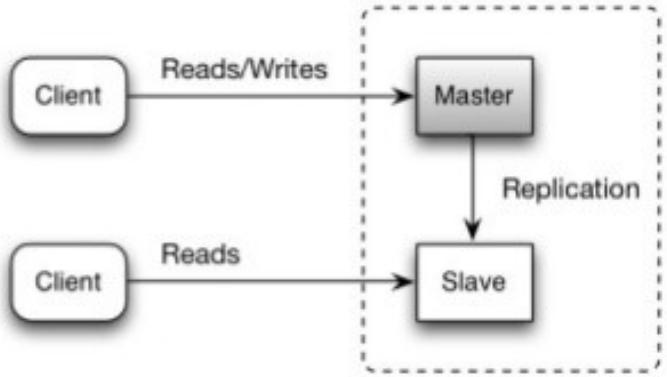
Distributed Data - Replication

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Replication

- Definition - keeping a **copy of the same data on multiple machines** that are connected via a network
  - Assumption: the dataset is so small that each machine can hold a copy of the entire dataset
- Why?
  - **Reduce Latency**: keep data geographically close to users
  - **Increase Availability**: allow the system to continue working even if some parts fail
  - **Scalability**: to scale out the number of machines that can serve read queries (and thus increase read throughput)
- Challenge - handling changes in the data
  - easy task if replicating data does not change over time

Single-Leader Replication



Leader and Followers

- **Replica** - each node that stores a copy of the database
  - Every write to the database needs to be processed by every replica
- Leader-based replication
  - one of the replicas is designated the **leader** (also known as master or primary)
  - other replicas are the **followers** (read replicas, slaves, or hot standbys)
- Other designations:
  - active/passive or master-slave replication

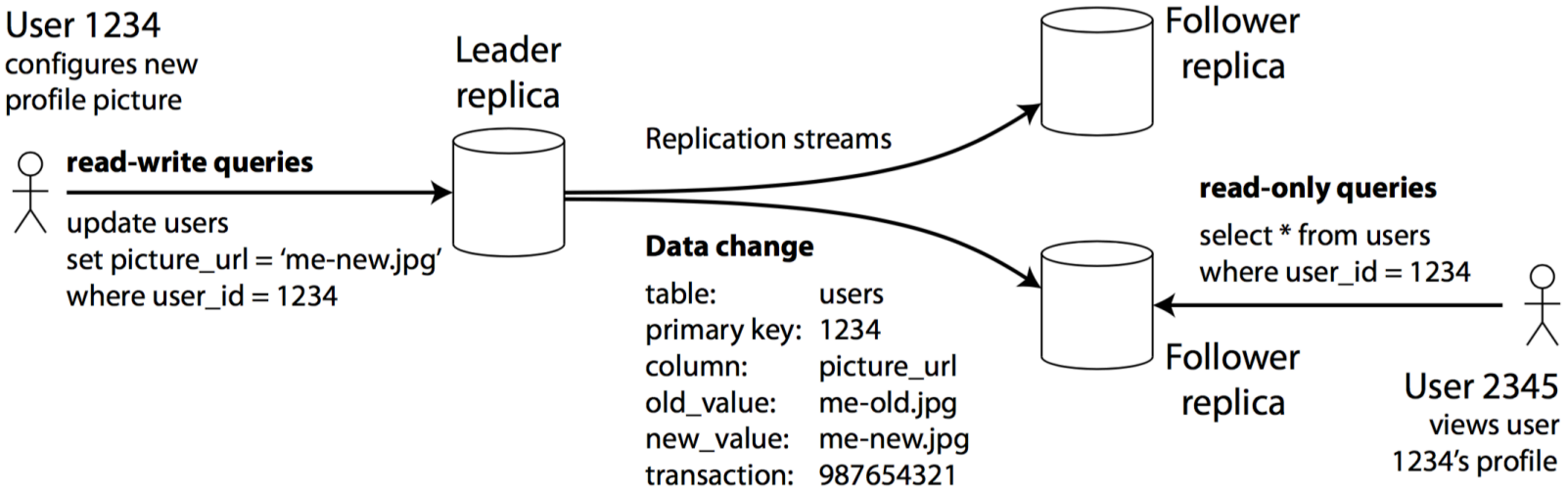
Leader and Followers

- Single-Leader Replication – **most common solution**

- This mode of replication is a built-in feature of many relational databases
  - e.g. PostgreSQL, MySQL, Oracle Data Guard, and SQL Server, …
- Also used in some non-relational databases
  - e.g. MongoDB, RethinkDB and Espresso

Leader and Followers – works like

1. **Clients writes** to the database must be send to the leader
2. **Leader** writes the new data to its local storage
3. Leader sends the data change to the followers
   - using replication log or change stream
4. Each **follower** takes the log and updates its local copy
   - all writes are processed in the same order as processed on the leader
5. **Clients reads** from the database can be done by query either the leader or any of the followers



Synchronous vs. asynchronous (2)

- Replication is usually quite fast
  - most database systems apply changes to followers in less than a second
- But… no guarantee for how long it might take
- There are circumstances when followers might fall behind the leader by several minutes or more, for example:
  - follower is recovering from a **failure**
  - system is operating near maximum **capacity**
  - **network** problems between the nodes

Synchronous vs. asynchronous (3)

- **Advantage** of synchronous replication - the follower have an up-to-date copy (consistent) of the data
  - if leader fails, the data is available on the follower
- **Disadvantage** - the write cannot be processed if the synchronous follower doesn't respond
  - leader must block all writes until the synchronous replica is available again
- Impractical to have all followers synchronous
  - in practice, if you enable synchronous replication on a database, it usually means that one of the followers is synchronous, and the others are asynchronous

Synchronous vs. asynchronous (4)

- **Semi-synchronous** configuration
  - if the synchronous follower becomes unavailable or goes slow, one of the asynchronous followers is made synchronous
  - guarantees an up-to-date copy of the data on at least two nodes: the leader and one synchronous follower
- **Fully asynchronous** configuration is often used
  - advantage that the leader can continue processing writes, even if all of its followers have fallen behind
  - however, write is not guaranteed to be durable, even if it has been confirmed to the client
    - if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost

Follower failure

- **Catch-up recovery**
  1. followers keep a log of data changes (received from leader)
  2. log used to know the last transaction before the fault occurred
  3. connect to leader and request all data changes that occurred during the time when the follower was disconnected
  4. apply these changes
  5. continue receiving a stream of data changes as before (regular operation state)

Leader failure

- Handling a failure of the leader is trickier:
  - one of the followers is promoted to **new leader**
  - clients need to be **reconfigured** to send **writes** to the new leader
  - other **followers** need to start consuming data changes from the new leader
- This process is called **Failover**
- Failover can happen
  - manually
    - an administrator is notified that the leader has failed, and takes the necessary steps to make a new leader
  - automatically

Automatic failover process

- **Determining that the leader has failed**
  - there is no foolproof way of detecting
  - most systems simply use a timeout
    - a node is assumed to be dead if it doesn't respond for some period of time
- **Choosing a new leader**
  - through an election process, where the leader is chosen by a majority of online replicas, or it can be appointed by a previously-elected controller node
    - the best candidate for leadership is usually the most up-to-date replica
- **Reconfiguring the system** to use the new leader
  - clients now need to send their write requests to the new leader (using a router/dns/service discovery-kind service)
  - problem: if the old leader comes back, it might still believe that it is leader
  - the system needs to ensure that the old leader becomes a follower and recognizes the new leader

Implementation of replication logs

- A replication log includes information about write operations in the database
  - e.g., about Inserted, Deleted and Updated rows

- Four main methods
  - Statement-based replication
  - Write-Ahead Log (WAL)
  - Logical log replication
  - Trigger-based replication

1. Statement-based replication

- Leader logs executed write statement and sends that statement log to the followers
  - Each follower executes the statement in its node
- Problems:
  - calls to non-deterministic function, for example NOW() or RAND(), will generate a different value on each replica.
  - statements using an auto-incrementing column or depend on the existing data in the database
  - statements that have side-effects (e.g. triggers, sp, udf) may result in different results on each replica
  - ensure execution order of statements in every node
- Other replication methods are generally preferred

2. Write-ahead log (WAL)

- Whenever a query comes to a system, even before executing that query, it is written in an **append-only log** file also known as Write-ahead log file.
- Besides writing the log to disk, the leader also sends it across the network to its followers
- Disadvantage:
  - log describes the data on a very low level
    - makes the replication closely coupled to the storage engine
    - difficult to run different versions of the database software on the leader and the followers
- This method of replication is used in PostgreSQL and Oracle, among others

3. Logical log replication

- An alternative is to use **different log formats** for replication and for the storage engine.
- More easily be kept backwards-compatible
  - leader and follower can run different versions of the database software, or even different storage engines
- A logical log format is also easier for external applications to parse
  - e.g., data warehouse for offline analysis, or for building custom indexes and caches

4. Trigger-based replication

- **Replication at application layer** - more flexibility
- Usage examples:
  - replicate a subset of the data
  - replicate from one kind of database to another
- Approaches (at database system layer):
  - by reading the database log
  - use database features (e.g., triggers and stored procedures)
    - trigger can log database writes into a separate table where an external process can read it
    - trigger-based replication typically has greater overheads than other replication methods

Replication Lag Problem

- Read-scaling architecture
  - add followers to increases the capacity for serving read-only requests
  - removes load from the leader
- But this may cause a **Replication Lag**
  - reads from asynchronous followers may see outdated information
  - E.g., if we run the same query on the leader and a follower at the same time, we may get different results
- **Eventual consistency**
  - This inconsistency is just a temporary state
  - the followers will eventually catch up and become consistent with the leader.

Replication Lag solutions

- Approaches to deal with Replication Lag problems:
  - **Read-after-write consistency**: a user should always see data that they submitted themselves
  - **Monotonic reads**: after a user has seen the data at one point in time, they shouldn't later see the data from some earlier point in time
  - **Consistent prefix reads**: users should see the data in a state that makes causal sense, for example seeing a question and its reply in the correct order
    - This is a problem in partitioned (sharded) databases, which we will discuss later

Read-after-write consistency

- When new data is submitted, it must be sent to the leader, but when the user views the data, it can be read from a follower
  - this is especially appropriate if data is frequently viewed, but only occasionally written
- A problem with asynchronous replication
  - the new data may have not yet reached the replica
- **Solution**: read-after-write consistency
  - also known as read-your-writes consistency

Read-after-write consistency

- **Implementation**
  - read from the leader something that the user may have modified, otherwise read it from a follower
  - requires some way of knowing what data have been modified
  - potential problem: if most things are potentially editable by the user, that approach won't be effective (negating the benefit of read scaling)
- **Other criteria** to read from the leader
  - tracking the time of the last update - for *X* minute(s) after the last update, all reads are made from the leader
  - client record the timestamp of its most recent writes
    - system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp

Monotonic reads

- Problem
  - when you read data, you may see an old value
  - can happen if makes several reads from different replicas

- This scenario is quite likely if the user refreshes a web page, and each request is routed to a random server
- **Solution**: Monotonic reads
  - ensures that this kind of anomaly does not happen by making each user reading always from the same replica
  - different users can read from different replicas

Consistent prefix reads

- Replication lag anomalies concerns violation of causality
  - if some partitions are replicated slower than others, an observer may see the answer before they see the question
- Consistent prefix reads prevents this kind of anomaly:
  - guarantees that, if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order
- This is a problem in partitioned (sharded) databases
  - which we will discuss later

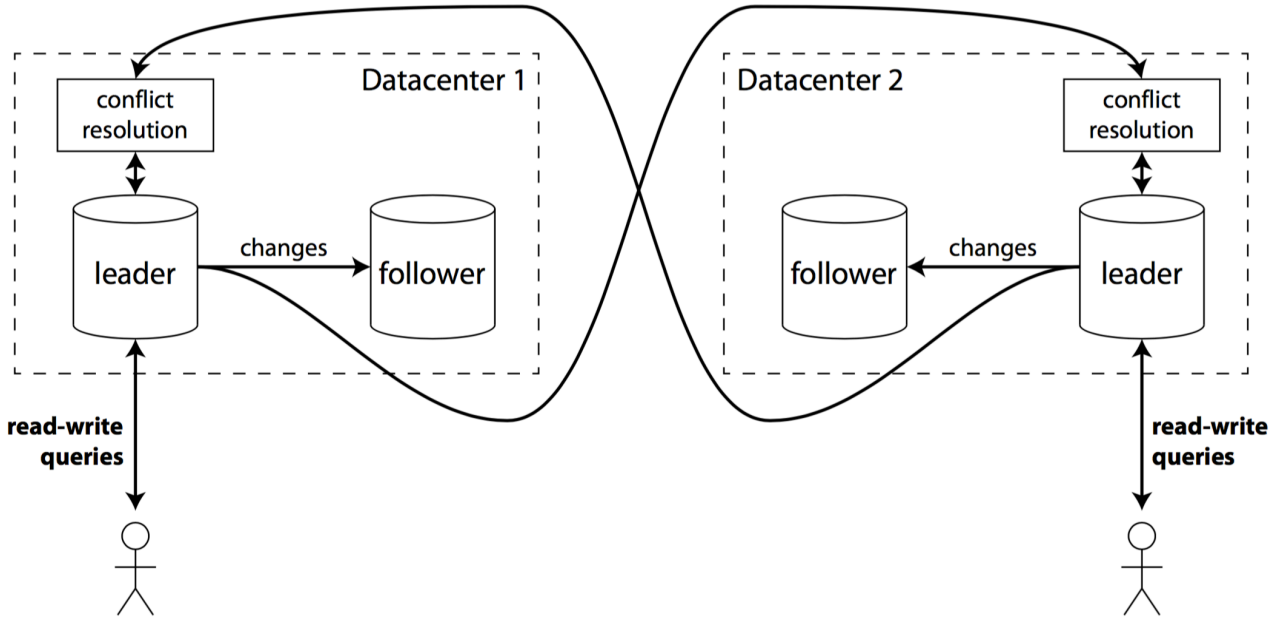
Multi-Leader Replication

Multi-leader replication

- **Leader-based** replication problems:
  - all writes must go through the single leader
  - inaccessible leader => can't write to the database
- **Multi-leader** configuration
  - **more than one node can accept writes**
  - known as master-master replication or active/active
- An extension of the leader-based replication
  - replication happens in the same way: each node that processes a write must forward that data change to all the other nodes
- Each leader simultaneously acts as a follower to the other leader

1. Multi-datacenter operation

- A database with replicas in several datacenters
  - To tolerate failure of the datacenter or to be closer to users
- Each datacenter can have a leader
  - leader-based replication is used in each datacenter
  - each leader replicates its changes to other leaders

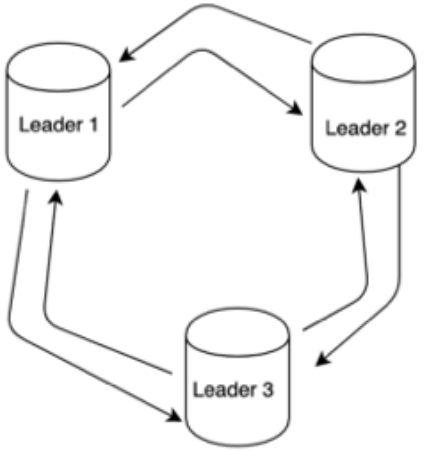


1. Multi-datacenter operation

- **Benefits**
  - Performance: local optimization while the inter-datacenter network delay is hidden from users
    - single-leader adds significant latency to writes between datacenters nodes
  - Tolerance to datacenter problems: multi-leader configuration allows each datacenter can continue operating independently of the other(s)
  - Tolerance to network problems: a temporary network interruption does not prevent writes being processed
- **Drawbacks**
  - Write conflicts: the same data may be concurrently modified
  - Auto-incrementing keys, triggers and integrity constraints can be problematic
  - Multi-leader replication is often considered much complex, and fail prone, so should be avoided if possible!

2. Clients with offline operation

- **Every device with a local database will act as a leader**
- Asynchronous multi-leader replication process
  - changes made in offline need to be synced with a server and other devices when the device comes again online
  - replication lag: hours or even days
- This is like the multi-leader replication between datacenters, taken to the extreme
  - the network connection between them is extremely unreliable
  - each device became a 'datacenter'

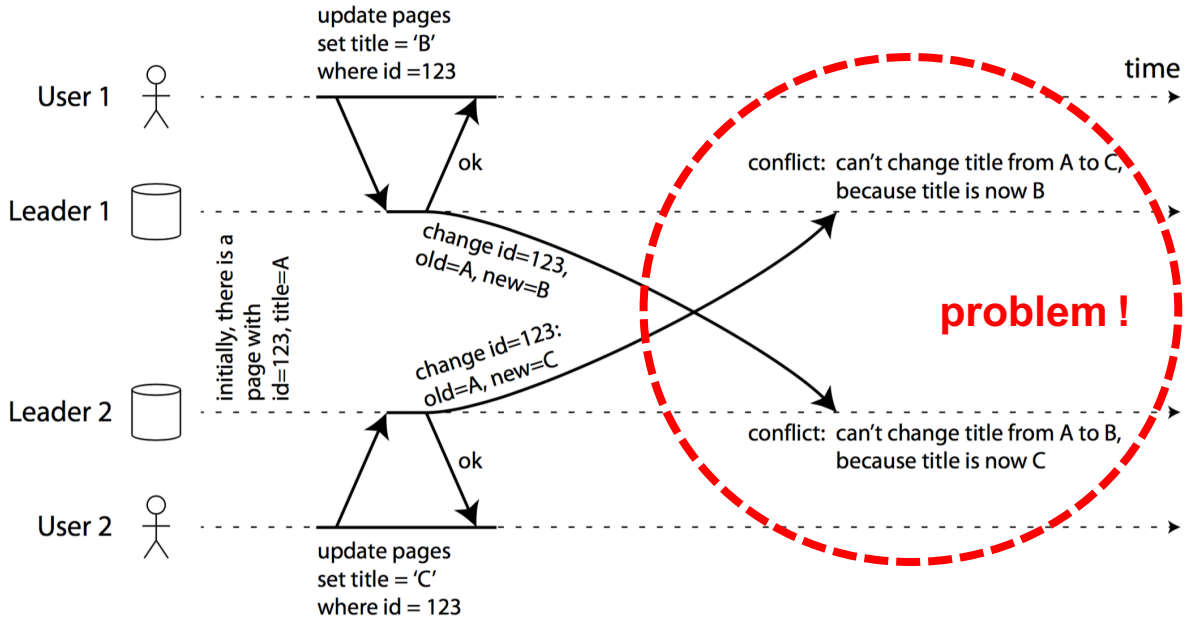


3. Collaborative editing

- Real-time collaborative editing applications allow several people to edit a document simultaneously
  - For example, Google Docs
- When one user edits a document
  - the changes are instantly applied to their local replica
  - and asynchronously replicated to the server and any other users who are editing the same document
- To avoid editing conflicts, the application must obtain a lock on the document before a user can edit it
- Faster collaboration: requests a unit of change very small and avoid locking
  - this allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication, including requiring conflict resolution

Multi-leader Write conflicts

- The biggest problem with multi-leader replication is that write conflicts can occur
  - this problem does not occur in a single-leader
- The conflict is detected when changes are asynchronously replicated

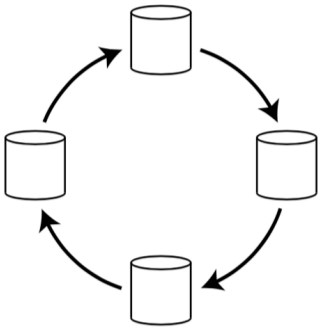


Conflict resolution

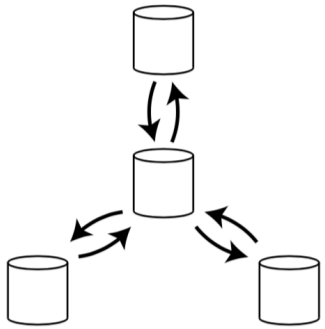
- **Convergent conflict resolution**:
  - Give each write a unique ID – pick the highest as winner
    - E.g., timestamp, a technique known as **last write wins (LWW)**
  - give each data replica a unique ID - writes from the higher-numbered replica will be the winner
  - merge the values together (e.g., data concatenation)
  - record the conflict in an explicit data structure with all information, and write application code which resolves the conflict at some later time (perhaps by prompting the user)
- **Custom conflict resolution logic**
  - It use application code to resolve conflicts on read and on write operations

Multi-leader replication topologies

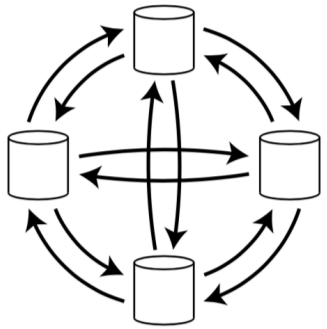
- Replication topology describes the communication paths that propagates writes from node to node
- Different topologies are possible:
  - all-to-all is the most general topology
  - circular topology is a more restricted topology (for example, MySQL support this by default)
  - star is another popular topology that can be generalized to a tree



(a) Circular topology



(b) Star topology



(c) All-to-all topology

Leaderless Replication

Leaderless replication

- **No leader**
- Any replica accepts writes from clients
  - in some implementations, the client directly sends its writes to several replicas
  - in others, a coordinator node does this on behalf of the client
    - unlike a leader database, that coordinator does not enforce a particular ordering of writes.

- Used in some of the **earliest replicated data systems**
  - fashionable architecture for databases after Amazon used it for their in-house Dynamo system
    - Riak, Cassandra and Voldemort are datastores inspired by Dynamo

Recovering missing writes

- A node comes back online, how does it get the missing writes?
- Two mechanisms are often used:
  - **Read repair**: client reads from several nodes in parallel. If detecting a stale responses, it writes the newer value back to that replica. Works well for values that are frequently read
  - **Anti-entropy process**: in addition, some datastores have a background process that constantly looks for differences in the data between replicas, and solves the problems
- Not all systems implement both mechanisms
  - without an anti-entropy process, values that are rarely read may be missing from some replicas and thus have reduced durability

Quorum for reading and writing

- **Consistency Quorum**
  - architecture with **n replicas**
  - every **write must be confirmed by w nodes** to be considered successful
  - the **client must query at least r nodes** for each read
- No quorum?
  - writes or reads return an error
- **Consistency Quorum condition**: w + r > n
- Normally, reads and writes are always sent to all n replicas in parallel
- Databases with appropriate quorum can tolerate the failure of individual nodes without need for failover

Without quorum consistency

- Case: $w + r \leq n$
- Reads and writes will still be sent to n nodes, but a smaller number of successful responses is required for the operation to succeed
- We are more likely to read stale values
- This configuration allows lower latency and higher availability
  - if there is a network interruption and many replicas become unreachable, there's a higher chance that you can continue processing reads and writes

Sloppy quorum

- When it is **not possible to assemble a quorum**?
  - Large clusters, network interruption, etc., a client may not be able to connect all the nodes
- In that case, database designers face a trade-off:
  - is it better to **return errors** to all requests for which we cannot reach a quorum of w or r nodes?
  - or **should we accept writes** anyway, and write them to some nodes that are reachable but aren't among the n nodes on which the value usually lives?
- The latter is known as a **sloppy quorum**
  - writes and reads still require w and r successful responses,
    - but those may include nodes that are not among the designated n "home" nodes for a value
  - Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes.
    - This is called **hinted handoff**

Multi-datacenter operation

- **Leaderless replication** is also well suited for multi-datacenter
  - tolerates conflicting concurrent writes, network interruptions and latency spikes
    - E.g., Cassandra implement multi-datacenter support within the normal leaderless model – allow specifying how many replicas per datacenter
- **Each client write is sent to all replicas**, regardless of datacenter
  - the client usually only waits for acknowledgement from a quorum of nodes within its local datacenter
    - avoiding delays and interruptions on the cross-datacenter link
- The higher-latency **writes to other datacenters** are often configured to happen **asynchronously**

Handling concurrent writes

1. Last write wins (LWW)

- Alike LWW in multi-leader replication
- Each replica only stores the most 'recent' value
  - and allows 'older' values to be overwritten and discarded
- Example:
  - attach a timestamp to each write and pick the biggest timestamp as the most 'recent'
- There are some situations, such as caching, in which lost writes may be acceptable.
  - If losing data is not acceptable, LWW is a poor choice for conflict resolution.
- LWW is the supported conflict resolution method in Cassandra, and an optional feature in Riak

Handling concurrent writes

2. Keys with version number

- The server maintains a **version number for every key**
  - incremented every time it is written
  - When a client reads a key, the server returns the value and its version number
- Clients must **read a key before writing** it
  - Clients can update an item, but only if the version number on the server side has not changed
  - If there is a version mismatch, it means that someone else (concurrent process) has modified the item before
  - the update attempt fails, because you have a stale version of the item
  - If this happens, you simply try again by retrieving the item and then attempting to update it

Handling concurrent writes

3. Merging concurrently written values

- Ensures that **no data is silently dropped**
  - But it, implies that clients have to merge the concurrently written values
    - Riak calls these concurrent values **siblings**
  - Merging sibling values is essentially the same problem as conflict resolution in multi- leader replication
- Possible approaches:
  - use a simple approach like LWW or version number
  - do something more intelligent like a union to merge values
  - no delete data, leaving a marker with an appropriate version number to indicate that the item has been removed when merging siblings.
    - Such a deletion marker is known as a tombstone.

Handling concurrent writes

4. Version vectors

- Algorithm for multiple replicas, but no leader
- Use a **version number per replica as well as per key**
  - This collection of version numbers is called a version vector
- Each replica increments its own version number when processing a write
  - also keeps track of the version numbers it has seen from all of the other replicas
  - it can then use that information to figure out which values to overwrite and which values to keep as siblings
  - The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica

Distributed Data - Partitioning

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

Partitioning

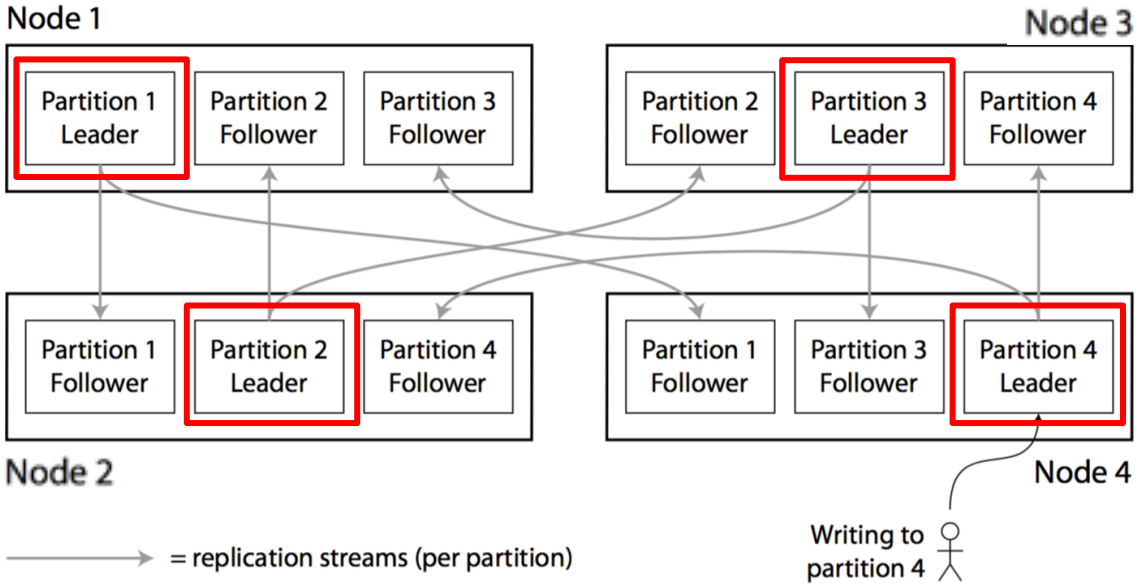
- **What?**
  - splitting a big database into smaller subsets called partitions, so that different partitions can be assigned to different nodes (also known as sharding)
- **Why?**
  - for very large datasets or very high query throughput
- **How?**
  - Usually, partitions are defined in such a way that each piece of data (each record, row or document) belongs to exactly one partition
  - Each partition is a small database of its own
  - Database may support operations over multiple partitions at the same time
- **Terminology**
  - shard in MongoDB, Elasticsearch and SolrCloud, a region in HBase, a tablet in BigTable, a vnode in Cassandra and Riak, and a vBucket in Couchbase

Partitioning

- The main reason for partitioning data is **scalability**
  - A large dataset can be spitted across many machines
  - Distinct partitions can be placed on different nodes in a shared-nothing cluster
- **Query load** can be also distributed
  - Small/regular queries operate on a single partition – each node can independently execute the queries for its own partition
  - Large/complex queries can potentially be parallelized across many nodes
- **Query throughput** can be scaled by adding more nodes

Leader-follower replication model

- Each partition's leader is assigned to one node and its followers are assigned to other nodes
- Each node may be the leader for some partitions, and a follower for other partitions
- Example:
  - 4 partitions
  - 3 replicas



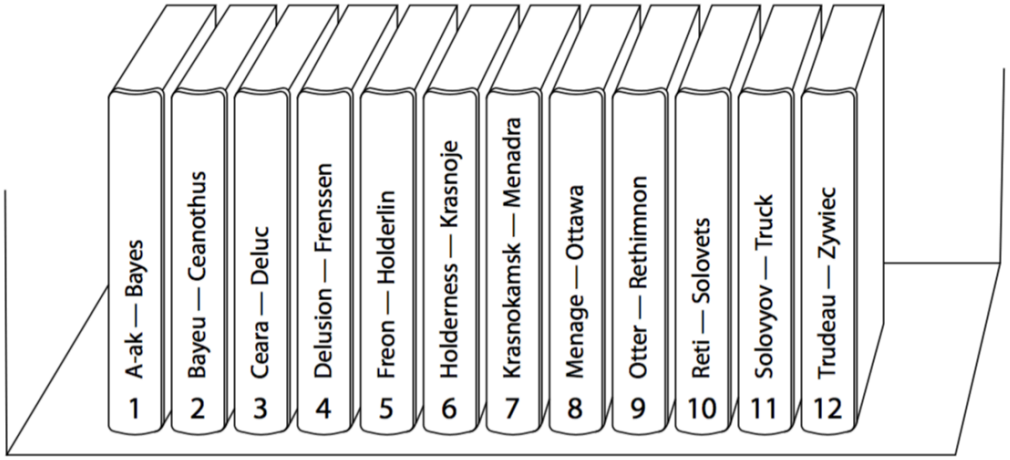
Note: the choice of partitioning scheme is mostly independent of the choice of replication scheme, so we will ignore replication in this module

Data partitioning

- **Challenge**: Partition of large amount of data
- **Goal**: spread the data and the query load across nodes … in a uniform way
- **Problems**:
  - skewed: some partitions have more data or serve a greater number of queries than others
  - hot spot: when one partition has disproportionately high load (skewed extreme case)
- **Solution**:
  - Algorithms to split records across nodes

1. Partitioning by key range

- Assign a continuous range of keys to each partition
  - know the boundaries between the ranges
  - boundaries chosen automatically or manually
  - easily determine which partition contains a given key
  - can make requests directly to the appropriate node
- Example: encyclopedia volumes



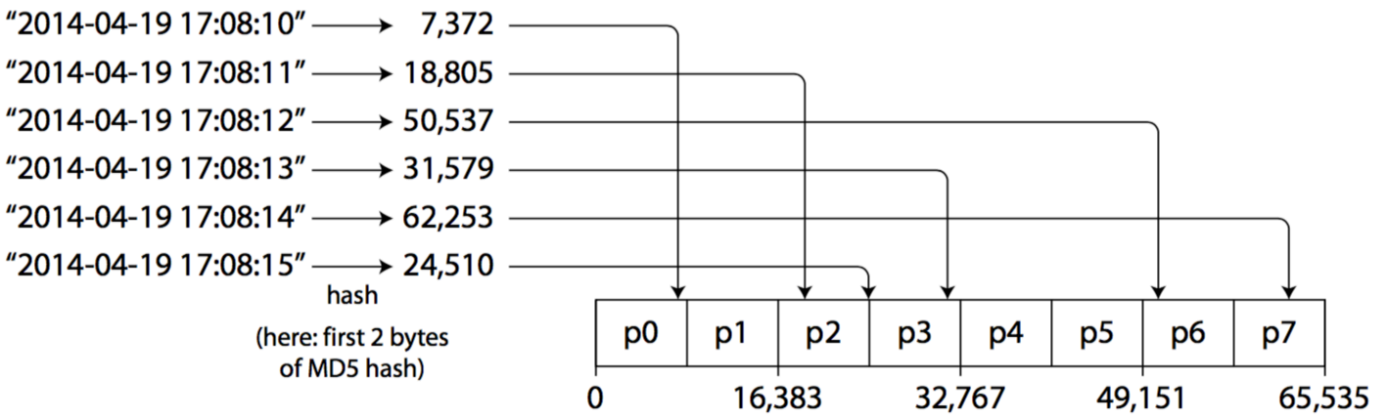
1. Partitioning by key range (cont.)

- Ranges of keys are not necessarily evenly spaced
  - data may not be evenly distributed
    - encyclopedia volume 1 contains words starting with A and B, but volume 12 contains words starting with T, U, V, X, Y and Z
- Within each partition, keys can be sorted
  - Allowing range queries
- Key can be a concatenated index in order to fetch several related records in one query
  - e.g., a timestamp 'year-month-day-hour-minute-second'

- Key range **downside**:
  - certain access patterns can lead to hot spots

2. Partitioning by hash of key

- Hash function to determine the partition for a key
  - reduce the risk of skew and hot spots
  - used by many distributed datastores, e.g., Cassandra and MongoDB
- A good hash function takes skewed data and makes it uniformly distributed
- Example: a hash function takes a string and returns a 32-bit integer



2. Partitioning by hash of key (cont.)

- Problem – no order of keys
  - lost of efficient range queries
- **MongoDB** with hash-based sharding enabled
  - range query are sent to all partitions
- Riak, Couchbase and Voldemort
  - range queries on the primary key are not supported
- **Cassandra** - compromise between two strategies
  - a table can be declared with a compound primary key consisting of several columns.
  - Only the first part of that key is hashed (partition key) to determine the partition,
    - the other columns (clustering key) are used as a concatenated index for sorting the data in SSTables
  - can search by a fixed value for the first column and perform an efficient range scan based on the other columns of the key
  - allows an elegant data model for one-to-many relationships

Skewed workloads

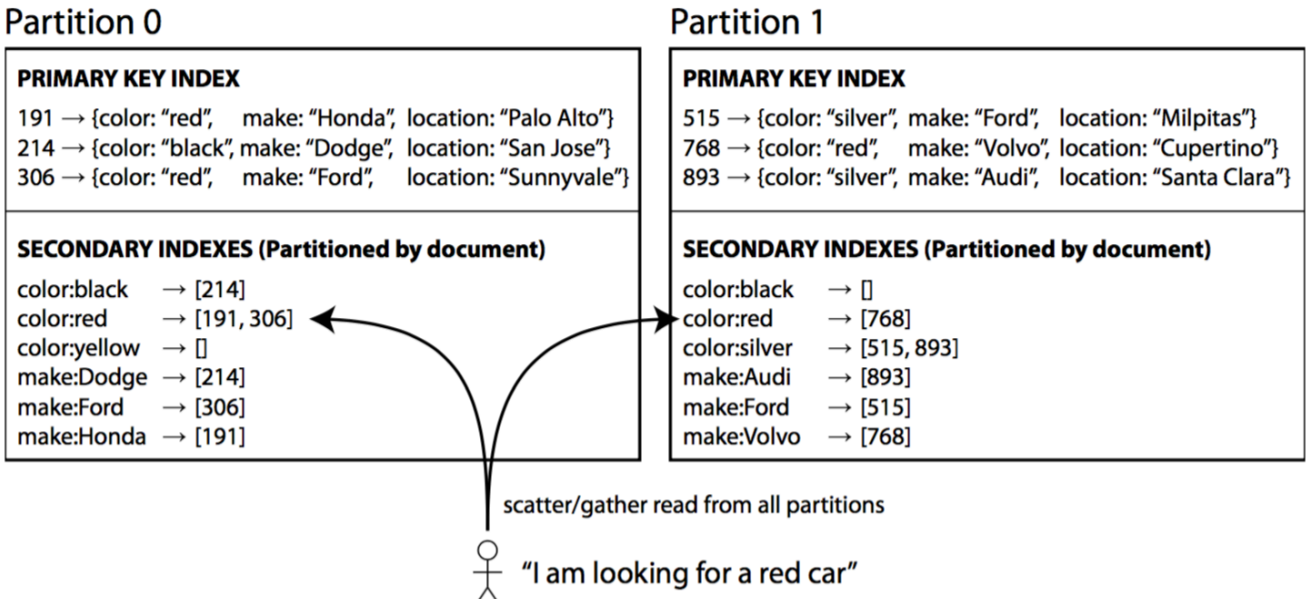
- Hashing a key cannot avoid hot spots entirely
  - in a extreme case, all reads and writes are for the same key. So, all requests will be routed to the same partition
  - this kind of workload is unusual but not impossible
    - for example, a celebrity in a social media site with millions of followers may cause a storm of activity when they do something
- Most data systems are not able to automatically detect and compensate for such a highly skewed workload
- Responsibility of the application to reduce the skew

Partitioning and secondary indexes

- Records are accessed via primary key that allow to determine the right partition to read and write
- Problem… secondary indexes
- DB Scenario
  - well supported by relational databases and common in document databases
  - Many key-value stores avoid secondary indexes because of their added implementation complexity
- Two main approaches to partitioning a database with secondary indexes:
  - **document-based partitioning**
  - **term-based partitioning**

1. Document partitioned index

- Characteristics:
  - index per partition
  - known as a local index
- Example: database for selling used cars
  - cars unique ID (document ID) used for partitioning the DB
  - secondary index: search by color and by make

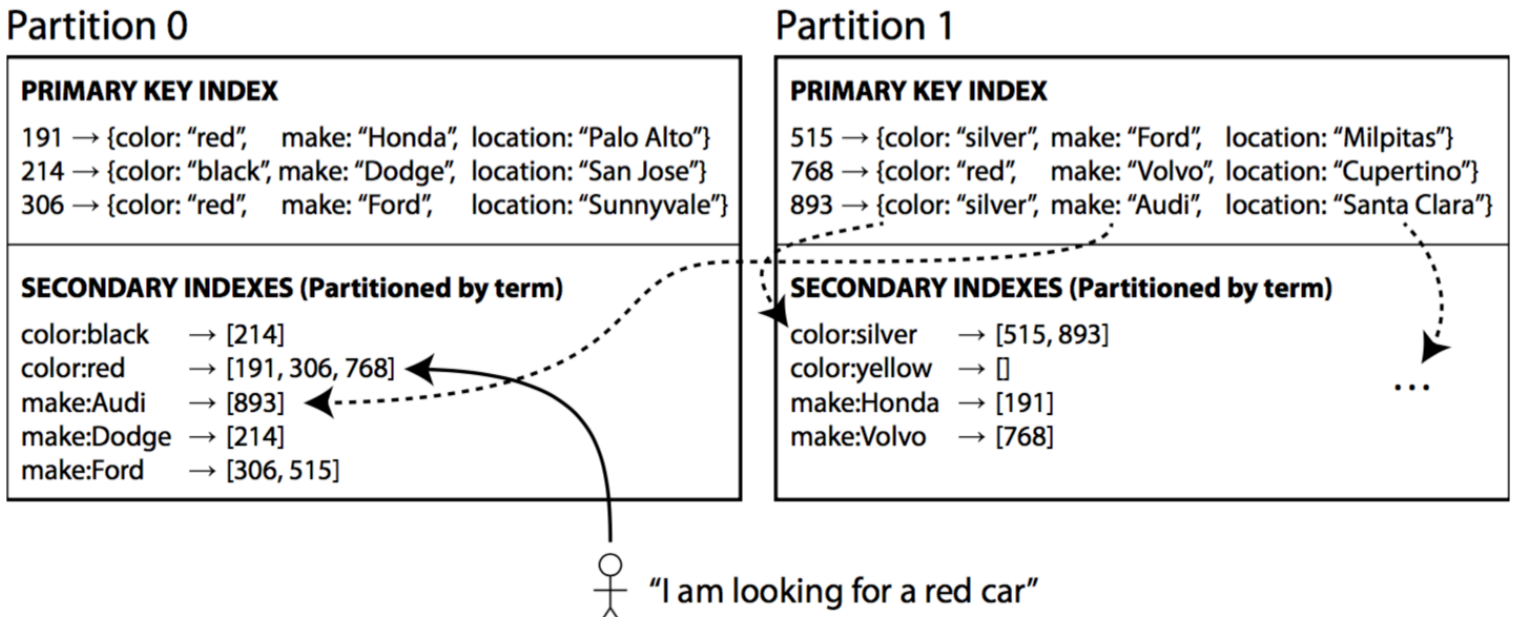


1. Document partitioned index (cont.)

- Each partition operates in a separate way
- **Writing** (add, remove or update) – only need to deal with the partition that contains a document ID
  - each partition maintains its own secondary indexes, covering only the documents in that partition
- **Reading** – requires to send the query to all partitions, and combine all the results obtained
  - this querying approach is known as scatter/gather
  - expensive process even if you query the partitions in parallel, is prone to latency increasing
- Used by MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud,..

2. Term-based index

- Characteristics:
  - a global index that covers data in all partitions
  - partitioned differently from the primary key index
    - store the index in only one node become a bottleneck and against the purpose of partitioning



2. Term-based index (cont.)

- Partitioning sec. index by term or by its hash is also possible with (dis)advantages discussed before
- Advantage of term-partitioned index over document-partitioned index is that it can make reads more efficient
  - client only needs to make a request to the partition containing the term
- Downside of a global index is that writes are slower and more complicated
  - write a document may now affect multiple partitions
- Updates to global secondary indexes are often asynchronous
  - if we read the index shortly after a write, the change may not yet be reflected in the index

Rebalancing partitions

- **Things that change** in a database over time:
  - query throughput increases => add more CPUs to handle the load
  - dataset size increases => add more disks and RAM to store it
  - a machine fails => other machines assumes its responsibilities
- **Rebalancing**: the process of moving data (partitions) between nodes in the cluster
- Independently of partitioning scheme used, rebalancing has usually some **minimum requirements**:
  - while rebalancing, the database should continue operating (accepting reads and writes)
  - after rebalancing, the load (data storage, read and write requests) should be shared fairly between all nodes
  - don't move more data than necessary between nodes, to avoid overloading the network

Rebalancing

- **Scenario**: partitioning by the hash of a key
  - assign *key* to partition 0 if $0 \leq hash(key) < b0$
  - assign *key* to partition 1 if $b_0 \leq hash(key) < b_1$
  - etc.
- **Strategy**: Round-robin 'hash(key) mod N'
  - N is the number of nodes
    - for example, N=10 returns a number between 0 and 9
- But this strategy has major **drawbacks**
  - when the number of nodes N changes, most of the keys would need to be moved from one node to another
  - makes rebalancing excessively expensive

Rebalancing – fixed n° of partitions

- **Scenario**: partitioning by the hash of a key
- **Approach**: number of partitions is fixed
  - what changes is the assignment of partitions to nodes
  - only entire partitions are moved between nodes
- More partitions than nodes
  - assign several partitions to each node
- Example
  - database running on a cluster of 10 nodes may be split into 1,000 partitions (100 partitions assigned to each node)
- Used by Riak, Cassandra, Elasticsearch, Couchbase, ..

Rebalancing – Dynamic partitioning

- **Scenario**: key range partitioning
  - problem: skew and hot spots
- **Approach**: create partitions dynamically
- Partition split
  - when partition grows and exceed a configured size
  - split into two partitions ~ half of the size
  - after split, a partition can be transferred to another node
- Partition merge
  - when lots of data is deleted, it can be merged with an adjacent partition
- Used in databases such as HBase and RethinkDB

Rebalancing – Dynamic partitioning

- **Advantage**: number of partitions adapts to data volume
- **Limitation**: when the dataset is small, it works with a single partition. So, all writes are processed by a single node while the other nodes sit idle
  - to mitigate this, HBase and MongoDB allow the setup of an initial set of partitions on an empty database (pre-splitting)

- Dynamic partitioning can also be used with hash-partitioned data
  - MongoDB supports both key-range and hash partitioning, but it splits partitions dynamically in either case

Rebalancing – automatic or manual?

- Rebalancing is an expensive operation
  - requires re-routing requests and moving a large amount of data between nodes
  - it can overload the network or the nodes, and cause performance problems for other systems
- Dangerous in combination with automatic failure detection - cascading failure!
- Fully automated rebalancing:
  - convenient: less operational/maintenance work
  - downside: can have unpredictable results
- Recommendation: human should control the process.
  - slower process but can help to prevent operational surprises
  - Couchbase, Riak generate a suggested partition assignment automatically, but require an administrator commit

Routing process

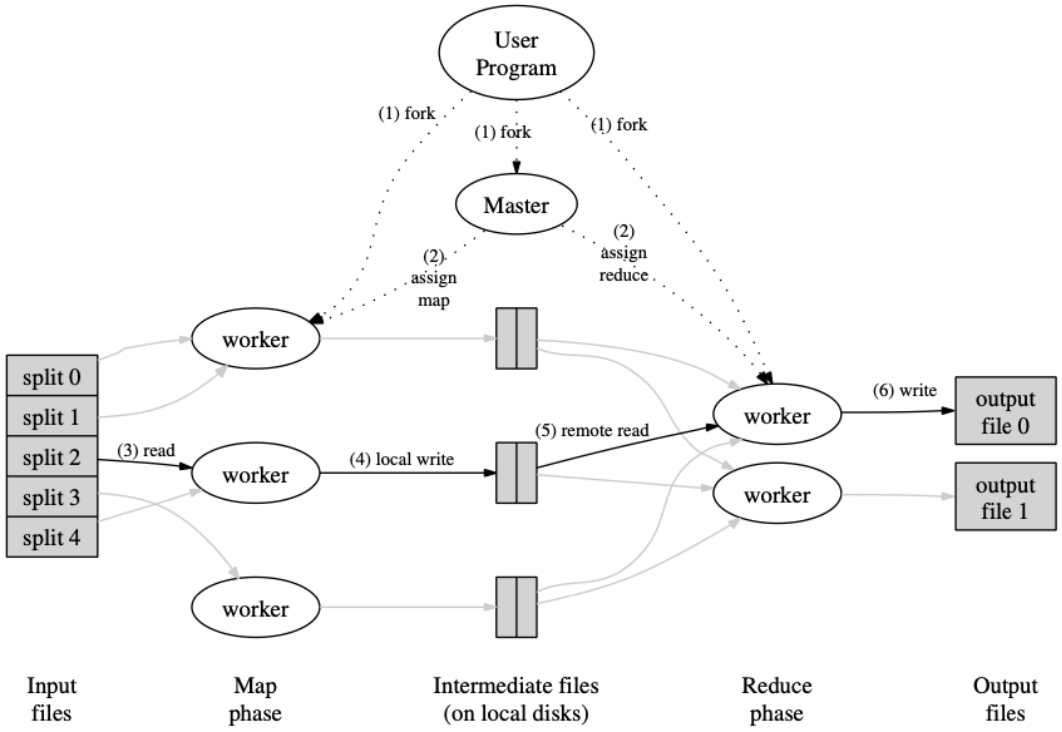
- We have now partitioned our dataset across multiple nodes running on multiple machines.
- But there remains an open question:
  - when a client wants to make a request, how does it know which node to connect to?
- The problem increase as partitions are rebalanced
  - the assignment of partitions to nodes changes.
- **Service discovery**
  - Allow clients to contact any/all node
    - e.g. via a round-robin load balancer
  - Clients are aware of the partitioning and the assignment of partitions to nodes.
    - can connect directly to the appropriate node, without any intermediary.
  - Send all requests from clients to a **routing tier first**
    - which determines the node that should handle the request and forwards it accordingly.

Distributed Data Processing

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

What is MapReduce?

- "MapReduce is a programming model and an associated implementation for processing and generating large data sets"
  - Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." (2004). [Google]



MapReduce frameworks

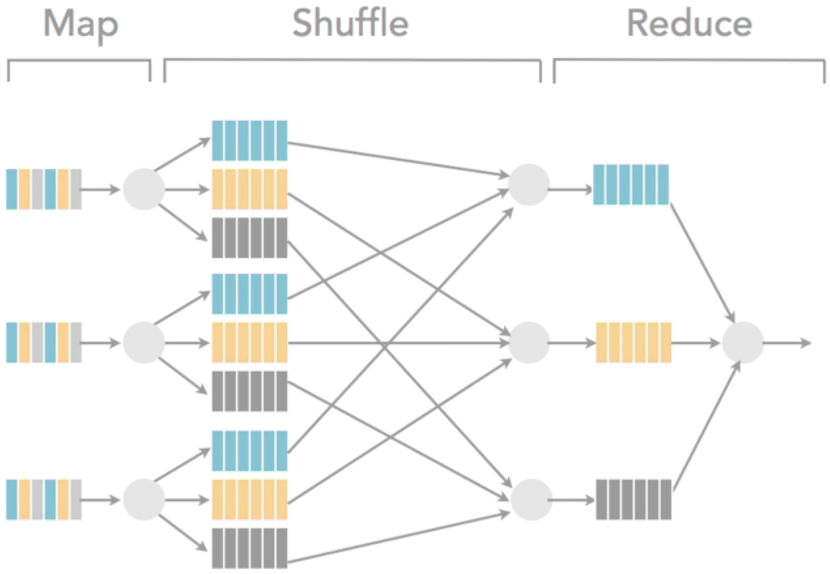
- Library/Tools that allow easily writing applications that process large amounts of data in parallel
  - distributed across several nodes
- Good retry/failure semantics
- Solution Pattern
  - many problems can be modelled in this way
- Implementations
  - **Hadoop**: the mapper and reducer are each a Java class that implements a particular interface.
  - **Spark**: newer and faster, it processes data in RAM using a concept known as an RDD, Resilient Distributed Dataset.

MapReduce frameworks

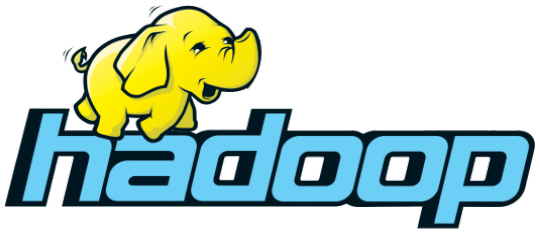
- **Framework**…
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates
- Two main tasks:
  - Mappers & Reducers
- Pipeline
  - Splits the input data-set into independent chunks
  - Mappers process of chunks in a parallel manner
  - Sort the outputs of the maps
  - Reducers process the sorted mappers output

MapReduce dataflow

- The **map** function
  - is called once for every input record to extract (one or more) key-value from the input record
- MapReduce framework
  - collects all the key-value pairs with the same key – **shuffle**
- The **reduce** function
  - iterates over each collection of values with the same key and can produce output records
    - e.g., the number of occurrences of the key

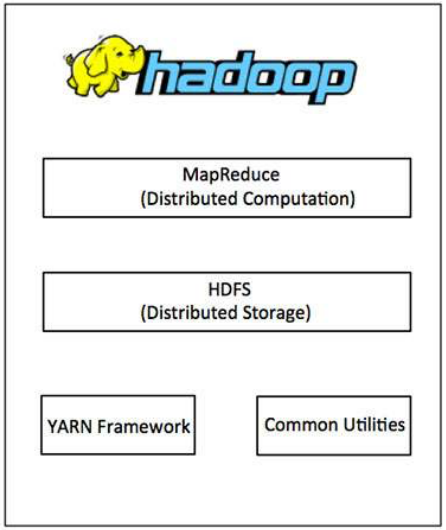


Apache Hadoop



Apache Hadoop Framework

- A framework that allows distributed processing of **large data sets across clusters of computers** using simple programming models.
- Open-source framework
  - Java
  - https://hadoop.apache.org
- **Main omponents**
  - Hadoop Distributed File System (HDFS)
    - Distributed, scalable, and portable file system
  - Hadoop Yet Another Resource Negotiator (YARN)
  - Hadoop Common Utilities
  - Hadoop MapReduce
    - Implementation of the MapReduce programming model



Hadoop Distributed File System (HDFS)

- Distributed file system designed to run on commodity hardware.
- Stores data redundantly on multiple nodes – Fault-tolerant file system
  - 3+ replicas for each block
  - Default Block Size : 128MB
- Master-Slave architecture
  - NameNode: Master
  - DataNode: Slave
- Typical usage pattern
  - Huge files (GB to TB)
  - Data is rarely updated
  - Reads and appends are common
    - Usually, random read/write operations are not performed

HDFS – Assumptions and Goals

- **Hardware Failure**
  - An HDFS instance may consist of +100/+1000 of server machines.
- **Streaming Data Access**
  - HDFS is designed more for batch processing rather than interactive use. The emphasis is on high throughput of data access rather than low latency of data access.
- **Large Data Sets**
  - Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size.
- **Simple Coherency Model**
  - HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed except for appends and truncates.
- **"Moving Computation is Cheaper than Moving Data"**
  - Especially true when the size of the data set is huge. Minimizes network congestion and increases the throughput of the system.
- **Portability Across Heterogeneous Platforms**

HDFS

- Master-Slave architecture
  - Master: NameNode
  - Slave: DataNode
- The Master node is a special node/server that
  - Stores HDFS metadata
    - E.g., the mapping between the name of a file and the location of its chunks
  - Might be replicated
- Client applications: file access through HDFS APIs
  - Talk to the master node to find data/chuck servers associated with the file of interest
  - Connect to the selected chunk servers to access data

Hadoop MapReduce

- MapReduce is the data processing layer of Hadoop.
- MapReduce job comprises a number of map tasks and reduces tasks.
  - Each task works on a part of data. This distributes the load across the cluster.
  - **Map task** load, parse, transform and filter data.
  - Each **reduce task** works on the sub-set of output from the map tasks (e.g., by applying grouping and aggregation).

Java Interface

- **Mapper** class
  - Implementation of the **map function**
  - Template parameters

    `KEYIN, VALUEIN` – types of input key-value pairs

    `KEYOUT, VALUEOUT` – types of intermediate key-value pairs
  - Intermediate pairs are emitted via **`context.write(k, v)`**

```
class MyMapper extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  @Override
  public void map(KEYIN key, VALUEIN value, Context context)
    throws IOException, InterruptedException
  {
    // Implementation
  }
}
```

Java Interface

- **Reducer** class
  - Implementation of the **reduce function**
  - Template parameters

    `KEYIN, VALUEIN` – types of intermediate key-value pairs

    `KEYOUT, VALUEOUT` – types of output key-value pairs
  - Output pairs are emitted via **`context.write(k, v)`**

```
class MyReducer extends Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  @Override
  public void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)
    throws IOException, InterruptedException
  {
    // Implementation
  }
}
```