Foundations in Data Engineering

Ilaria Battiston *

Winter Semester 2019-2020

*All notes are collected with the aid of material provided by T. Neumann. All images have been retrieved by slides present on the TUM Course Webpage.

Contents

1 Tools and techniques for large-scale data processing

Data processing, especially in the age of big data, can be done with many different approaches and techniques: it is handled with software layers, which have different principles and use cases. All digital data created reached 4 zettabytes in 2013, needing 3 billion drives for storage.

Scientific paradigms concerning collection and analysis have developed as well, including a preprocessing part consisting in observations, modeling and simulations.

Big data is a relative term: each field has its own definition, but it's usually hard to understand. A model might be as complicated as its information, and tools need to be reinvented in a different context as the world evolves.

1.1 Volume and privacy

Volume is one of the most important challenges when dealing with big data: it is not often possible to store it on a single machine, therefore it needs to be clustered and scaled.

Supercomputers (and AWS) are extremely expensive, and have a short life span, making them an unfeasible option. In fact, their usage is oriented towards scientific computing and assumes high quality hardware and maintenance. Programs are written in low-level languages, taking months to develop, and rely extensively on GPUs.

Cluster computing, on the other hand, uses a network of many computers to create a tool oriented towards cheap servers and business applications, although more unreliable. Since it is mostly used to solve large tasks, it relies on parallel database systems, NoSQL and MapReduce to speed up operations.

Cloud computing is another different instance which uses machines operated by a third party in a data center, renting them by the hour. This often raises controversies about the costs, since machines rarely operate at more than 30% capacity.

Resources in cloud computing can be scaled as requests dictate, providing elastic provisioning at every level (SaaS, IaaS, PaaS). Higher demand implies more instances, and vice versa.

The downside are the proprietary APIs with lock-in that makes customers vulnerable to price increases, and local laws might prohibit externalizing data processing. Providers might control data in unexpected ways, and have to grant access to the government - privilege which might be overused.

Privacy needs to be guaranteed, stated and kept-to: numerous web accounts get regularly hacked, and there are uncountable examples of private data being leaked.

Performance is strictly linked to low latency, which works in function of memory, CPU, disk and network. Google, for instance, rewards pages that load quickly.

1.2 Velocity

Velocity is the problematic speed of data: it is generated by an endless stream of events, with no time for heavy indexing leading to developments of new data stream technologies.

Storage is relatively cheap, but accessing and visualizing is next to impossible. The cause of this are repeated observations, such as locations of mobile phones, motivating stream algorithms.

1.3 Variety

Variety represent big data being inconclusive and incomplete: to fix this problem, simple queries are ineffective and require a machine learning approach (data mining, data cleaning, text analysis). This generates technical complications, while complicating cluster data processing due to the difficulty to partition equally.

Big data typically obeys a power law: modeling the head is easy, but may not be representative of the full population. Most items take a small amount of time to process, but a few require a lot of time; understanding the nature of data is the key.

Distributed computation is a natural way to tackle Big Data. MapReduce operates over a cluster of machines encouraging sequential, disk-based localized processing of data. The power laws applies, causing uneven allocation of data to nodes (the head would go on one or two workers, making them extremely slowly) and turning parallel algorithms to sequential.

2 File manipulation tools

Analyzing a dataset is useful to get a first impression about it, decide which tools are the most suitable and understand the required hardware.

Files can have a huge variety of formats, when in doubt the `file` command can be useful.

CSV are plain text files containing rows of data separated by a comma, in a simple format with customizable separator. It requires quoting of separators within strings.

XML is a text format encoding of semi structured data, better standardized than CSV but not human writable. It is suitable for nested objects and allows advanced features, yet it is very verbose and its use is declining.

JSON is similar to XML although much simpler and less verbose, easy to write. Its popularity is growing.

2.1 Command line tools

Text formats are overall well-known and can be manipulated with command line tools, a very powerful instrument to perform preliminary analysis:

- `cat` shows the content of one or multiple files (works with piped input as well);
- `zcat` is `cat` for compressed files;
- `less` allows paging (chopping long lines);
- `grep` is useful to search text (regex goes between quotes) with options such as file formats, lines not matching and case insensitiveness;
- `sort` to (merge) sort even large output;
- `uniq` handles duplicates;
- `tail` and `less` display suffixes and prefixes;
- `sed` to edit text, match and replace characters (in-place update using the same file);

- `join` to combine sorted files according to a common field;
- `awk` (followed by a `BEGIN-END` block) executes a program for every line of input, such as average or sum;
- ...

Tools can be combined through pipes, which redirect the input/output stream. Some examples are:

- `|` to concatenate two commands (& also pipes standard error);
- `>` to redirect to a file;
- `<` to redirect from a file;
- `&&` executes a command only if the previous one has succeeded;
- `||` executes a command only if the previous one has failed;
- `;` separates multiple commands regardless of their output.

Process substitution `<()` allows the input or the output of a command to appear as a file, to avoid verbose commands and extra processing.
Example: `diff <(sort file1) <(sort file2)`

2.1.1 `cat`

Syntax: `cat [OPTION] [FILE]`

`cat` concatenates file to standard output. When the file is unspecified, it prints to terminal. It can also be used with multiple files, and different formats (binary, compressed). It is mainly used as input for other commands, since displaying large data can be computationally expensive.

2.1.2 `less`

Syntax: `less [OPTIONS]`

`less` is opposite to the analog command `more`, extending it with other features. It does not have to read the entire input file before starting, so with large files it starts up faster than text editors or `cat`.

Commands may be preceded by decimal numbers indicating the number of rows to display/scroll, and basic pattern matching can be performed as well.

2.1.3 grep

Syntax: `grep [OPTION] PATTERNS [FILE]`

It can search (Boyer-Moore algorithm) for any type of string, or any file, or even lists of files and output of commands. It uses basic regular expressions and normal strings.

There also are the variations `egrep` (for extended regular expressions), `fgrep` (fixed, to only search strings) and `rgrep` (recursive, for folders).

Regular expressions syntax:

- `[]` enclose lists of characters (and);

- `-` represents characters range;
- `^` negates an expression;
- `*` denotes zero or more occurrences of previous character;
- `()` start and end of alternation expression;
- `n` range specifier (number of repetitions) → `a{3}`;
- `\<` and `\>` match empty strings at the beginning and the end of a word.

Extended regular expressions treat meta-characters without needing to escape them, and are more powerful thanks to the additional commands:

- `?` matches at most one repetition;
- `+` denotes one or more occurrences of previous character;
- `|` matches either of the expressions → `(a|b)`.

`grep` returns all the lines with a match of the pattern. To only return the match, `-o` option needs to be specified.

2.1.4 `sort`

Syntax: `sort [OPTION] [FILE]`

`sort` sorts lines of text files, writing them to standard input. When no file is specified, it reads standard input. Some ordering options are `-n` (numeric), `-r` (reverse), `-k` (specific key), `-u` (removes duplicates).

It can handle files larger than main memory, and it is very useful to prepare input for other algorithms.

2.1.5 `head\tail`

Syntax: HEAD [OPTION] [FILE]

Those commands print the first (last) lines of files. The default number of lines is 10, and multiple files can be read.

An integer can be specified as option to determine how many lines must be read (and skipped). Those commands are relevant to extract minimum and maximum of computations. Numbers may have a multiplier suffix, indicating bytes or related measurement units.

2.1.6 `uniq`

Syntax: `uniq [OPTION] [INPUT [OUTPUT]]`

`uniq` handles (adjancent) duplicates (similarly to SQL DISTINCT) in sorted input, reporting or omitting them. It also has option `-c` to count duplicates or `-u` for unique lines. With no options, matching lines are merged to the first occurrence.

Its main applications are grouping and counting input.

2.1.7 `cut`

Syntax: `cut [OPTION] [FILE]`

`cut` removes section from each line of files. This can be seen as selecting columns from a CSV, where relevant parts of delimited input get redirected to output.

Example: `cat file | cut -f 1, 3`
Returns specific fields of file, delimiter can be specified with `-d`.

2.1.8 `wc`

Syntax: `wc [OPTION] [FILE]`

`wc` prints newline, word and byte counts for each file (and a total line if more than one file is specified). A word is a non-zero-length sequence of characters delimited by white space.

Options may be used to select which counts are printed: `-c` for bytes, `-m` for chars, `-l` for lines and so on.

2.1.9 `shuf`

Syntax: `shuf [OPTION] [FILE]`

This command generates random permutations of the input lines to standard output. It is not so common, although very useful when extracting random samples and running performance tests.

2.1.10 `sed`

Syntax: `sed [OPTION] script [FILE]`

`sed` is a stream editor to perform basic text transformations on an input stream (or file). It works by making only one pass over the input, therefore it is more efficient than scripts. It can also filter text in a pipeline.

This command is mostly used to replace text in a file. The operations are separated by a slash, and the letters (flags) represent options.

Example: `cat file | sed 's/oldText/newText/'`
Replaces oldText with newText.

Example: `cat file | sed 's/file[0-9]*.png/"&"/I'`
Case insensitive matching and replacing.

2.1.11 `join`

Syntax: `join [OPTION] FILE1 FILE2`

`join` joins the lines of two files on a common field. For each pair of input lines with identical join fields, writes a line to standard output. The default join field is the first, delimited by blanks. One file can be omitted, reading standard output.

Unmatched lines are removed, or preserved with `-a`.

Field number can be specified with `-j` or `-1 FIELD -2 FIELD`. Values must be sorted before joining, and it is not powerful compared to other tools.

2.1.12 `awk`

Syntax: `gawk [POSIX or GNU style options] -f program-file [--] file`

Gawk is the GNU project's implementation of the AWK programming language, according to the POSIX standards. The command line takes as input the AWK program text.

Gawk also has an integrated debugger and profiling statistics for the execution of programs.

AWK's general structure is:
`awk 'BEGIN init-code per-line-code END done-code' file`

Example: `awk 'BEGIN x=0; y=0 x=x+$2; y=y+1 END print x/y'`
Computes the average of column 2 (identified with the dollar).

3 Performance spectrum

Analyzing the performance is relevant to understand whether an operation is taking too long, and comparing it according to time and input size. It can improve speed and energy consumption.

The performance spectrum involves many variables: theoretical limits, programming tools (interpreted vs. compiled languages) and methods, or hardware (clustering, scaling). Real input is complex, yet just analyzing simple text queries using benchmarks can be highly explicative.

For instance, a program written in AWK or Python will be slower than C++ since those two are interpreted languages and require extra operations.

Example bandwidth for different input processing:

- 1GB ethernet: 100 MB/s;
- Rotating disk (hard disk): 200 MB/s;
- SATA SSD: 500 MB/s;
- PCI SSD (uses multiple lanes to connect to the motherboard): 2 GB/s;
- DRAM (dynamic RAM, has the fastest path to the CPU thanks to semiconductors): 20 GB/s.

Performance when manipulating text completely ignores CPU costs, which are not important for disks but very relevant for reading data from DRAM: when using the largest bandwidth, they have the biggest impact on computational time.

The first time a program is executed, data is stored in main memory: after that, it gets copied to cache, accessed in a quicker way, making the following executions faster (and CPU bound).

Naively optimizing can have downsides, making the situation worse sacrificing portability or creating a messy code: before touching a program there are tools which can be used to analyze time and performance (`perf`).

An efficient way to optimize code is memory mapping: files are saved in the address space of the program and are treated as arrays being accessed in the same way as dynamic memory (better than caching).

A memory mapped file is in fact a segment of virtual memory which has been assigned a byte-for-byte correlation with some resource, increasing I/O performance with faster access. Memory maps are always aligned to page size (4 KiB at most).

Memory mapped files can also be used to share memory between multiple processes without causing page faults or segmentation violations. Executions in C++ with cold cache are slightly slower, but warm cache are faster since data is directly accessed from there without copying.

Another useful way consists in using block operations. Search can be performed for instance through the encoding of special characters, since there is no way to split the data just by directly scanning the memory.

Loop nest optimization applies a set of loop transformations (increasing the speed of loops and improving cache performance) partitioning iteration spaces into smaller chunks, ensuring data stays in cache until it is reused.

Files can be stored in binary format to speed up implementation: using text files implies storing characters in more than 4 bytes, and parsing with delimiters is slower than just copying the bit stream to memory.

SSE instructions are an extension of x86 which define faster floating point operations opposite to the standard architecture, having wider application in digital signal and graphics processing. This method, although fast, is not portable.

Optimization of mathematical operations is done using several strategies:

- Choosing fastest registry operations (64-bit is faster than 32, performed automatically by the compiler);
- `const` instead of `constexpr`;
- Using assembly language to convert floating point to integer;
- Dividing and multiplying by powers of two (shifting bits), for instance $a/c = a*(2^{64}/c) >> 64$;
- Speeding up division and modulo by constant using unsigned integers.

Different data structures also have different performance: arrays have worst-case linear time search, while maps and sets are implemented using trees for better performance.

Multithreading (processing within multiple CPU cores) consistently increases performance - but network speed and remote CPUs can cause connection problems. Warm caches parallelizes well, yet cold cache gets slower.

Different input representation is not always an option, yet when possible a columnar binary format can reduce touched data up to a factor of 50. It requires an additional conversion step, but it pays off for repeated queries.

Using all those performance improvements can make a simple program improve bandwidth up to 5 GB/s.

3.1 Clustering

Using multiple machines to perform operations can seem a feasible option, but it depends on which part of the program has to be sent in the network.

Shipping data with a bandwidth of 100 MB/s having a local bandwidth of up to 5 GB/s is inefficient and would slow down computation.

On the other hand, if only computation can be shipped, the performance is determined by the latency and the number of nodes:

$$\approx 20ms + \frac{|W|}{5GB/s \cdot n}$$

n is the number of nodes, s is the latency in seconds and W is the size of the problem. W has to be larger than 113 MB for 10 nodes for this system to be efficient.

In the end, since a single machine can handle many gigabytes efficiently and network speed can be a bottleneck, scaling up is often preferred over scaling out: the latter, because of its significant overhead, only pays off in the long run.

4 SQL

SQL is a declarative language (orders to execute a query without specifying how) to interact with databases and retrieve data. Optimization is not only on the query level, but also on storage and data types: for instance, strings can be managed adding a pointer to their size (if they are long).

Various SQL dialects have slight differences between semantics, but there are general standards starting from older versions. Different databases might also have different type management, operations and speed.

Another common problem with databases is handling null values while evaluating boolean expressions: NULL and FALSE results in FALSE, therefore this may give an inaccurate representation of information. On the other hand, NOT NULL columns are beneficial for optimization (NULL columns have a reserved memory space for this possibility).

Some PostgreSQL data types:

- Numeric, slow arbitrary precision numbers with unbounded size;
- Float, mobile floating point (4 bytes) or double precision (8 bytes);
- Strings of variable length (pre-allocated or dynamic);
- Other common types such as `bytea`, `timestamp` and `interval`.

4.1 SQL commands

Standard SQL commands are for instance SELECT, FROM, WHERE, GROUP BY, ORDER BY. Those can be found in normal queries or subqueries, which allow to temporarily subset data without saving it to memory.

The `\copy` statement allows to import data from a text file, but can be slow with large datasets since it requires a full scan.

Pattern matching can be done either by comparison (=) or regular expression (LIKE or ∼).

Random samples can be extracted with various methods, of which the most accurate in randomness (Bernoulli) is also the slowest.

Set operations are performed thanks to the following instructions:

1. UNION, which vertically merges two tables with the same attributes;
2. EXCEPT, which performs the set difference `A - B`;
3. INTERSECT, to only keep rows in common between two tables.

The default option implies removing duplicates; to keep them, specifying ALL is necessary.

4.2 Views and CTEs

A view is an abstract database object of a stored query, accessed as a virtual table but physically not containing data. It simplifies complexity of interrogations because it wraps commonly used information and makes it reusable.

Views are faster, but can be viewed globally, causing potential issues with naming.

A better way to optimize through a temporary table is `WITH`, a statement which defines common table expressions (CTE).

CTEs are temporary result sets that can be referenced with another SQL instruction to simplify and speed up queries. Results get materialized and then scanned. The main difference from views is that CTEs have a scope only within the query, and get destroyed as soon as the execution is finished.

CTEs overcome SELECT statement limitations such as referencing themselves, which is the basis of recursion.

4.3 Regular expression matching

There are different algorithms which establish whether a given regex matches a string:

1. Deterministic finite automata, built from a non-deterministic one, which allows to match in linear time yet is constructed in exponential state (to the size);
2. NFA with caching to simulate a DFA, avoiding the construction cost but raising the computational to almost quadratic;
3. Pattern matching by backtracking, forcing an exponential number of sub-cases.

Variations of the Boyer-Moore algorithm (grep) are commonly used, optimized with automata.

Postgres uses Harry Spencer's `regex` library for matching, an hybrid DFA/NFA implementation (thank you Tom Lane for making clear documentation) which initially generates a NFA representation, then optimizes it and executes a DFA materializing states only when needed and keeping them in cache.

The parser also includes an NFA mode to implement features such as capturing parentheses and back-references. It constructs a tree of sub-expressions which is recursively scanned similarly to tradition engines, therefore quite slow.

To make this faster, each node has an associated DFA representing what it could potentially match as a regular expression. Before searching for sub-matches, the string goes through the DFA so that if it does not match the following stages can be avoided.

4.4 String comparison

String comparison is a task which can be computationally expensive in case of very large output.

There are several methods to implement this, such as suffix trees and edit distance, which also allow to search for errors in text through string similarity.

The metric represents the number of operations that need to be applied to one of the inputs to make it equal to the other one, through dynamic programming (construction of a matrix).

4.5 Indexing

Postgres provides several index types, of which the most common is the B-tree, a self-balancing data structure allowing logarithmic-time operations (the most important is searching).

It is useful because nodes can have more than two children: the number of sub-trees can in fact be a multiple of the file system block size. The structure is optimized to have few accesses to disk avoiding loading data within reads.

Further improvements can be made creating auxiliary indexes containing the first record in each disk block, narrowing the search in the main database.

4.6 Counting

Counting, unless an index is present, is not optimizable: to have an exact result, the DBMS must perform a linear scan of the table, to avoid eventual transaction conflicts.

Counting distinct values or grouping before joining, on the other hand, is faster since it involves an approximation.

Number of total rows can be approximated as well, thanks to sketches: probabilistic data structures that contain frequency tables of events in a stream of data.

They use hash tables to map events to frequencies, and at any time the sketch can be queried for the frequency of a particular element.

To reduce variance and have a more accurate estimation, many independent copies can be stored in parallel, and their outputs is combined.

4.7 Count-distinct problem

The count-distinct problem finds how many unique elements are there in a dataset with duplicates.

The naive implementation consists in initializing an efficient data structure (hash table or tree) in which insertion and membership can be performed quickly, but this solution does not scale well.

Streaming algorithms use randomization to produce a close approximation of the number, hashing every element and store either the maximum (likelihood estimator) or the m minimal values.

4.7.1 HyperLogLog

HyperLogLog is one of the most widely used algorithms to count distinct elements avoiding calculating the cardinality of a set.

It is a probabilistic algorithm, able to estimate cardinalities of 10^9 with a typical standard error around 2% using very little memory.

The process derives from Flajolet-Martin algorithm: given a hash function which uniformly distributes output, it is taken the least-significant set bit position from every binary representation.

Since data is uniformly distributed, the probability of having an output ending with 2^k (one followed by k zeros) is $2^{-(k+1)}$. The cardinality is then estimated using the maximum number of leading zeros. With n zeros, the number of distinct elements is circa 2^n.

HyperLogLog also allows to add new elements, count and merge to obtain the union of two sets.

4.8 Sampling

Sampling is usually performed through the hypergeometrical distribution, a formula composed by 3 binomial coefficients. Computing factorials, approximated with exponentiation, can be a heavy operation.

Factorials can be estimated using Stirling approximation: $\log(n!) \approx n \log n - n + \frac{1}{2}(2\pi n)$, which is cheaper and works for sufficiently big n, but still not efficient for very large numbers.

5 Query decorrelation

Often queries are easier to formulate using subqueries, yet this can take a toll on performance: since SQL is a declarative language, the best evaluation strategy is up to the DBMS.

A subquery is correlated if it refers to tuples from the outer query. The execution of a correlated query is poor, since the inner one gets computed for every tuple of the outer one, causing a quadratic running time.

Correlation can be recognized observing fields that do not have a bound with the attributes of the outer query. In SQL, they usually are in WHERE or EXISTS statements.

Example (from TCP-H dataset):

```
select avg(l_extendedprice)
from lineitem l1
where l_extendedprice =
       (select min(l_extendedprice)
       from lineitem l2
       where l1.l_orderkey = l2.l_orderkey;)
```

This query can be rewritten to avoid correlation:

```
select avg(l_extendedprice)
from lineitem l1,
       (select min(l_extendedprice) m, l_orderkey
       from lineitem
       group by l_orderkey) l2
where l1.l_orderkey = l2.l_orderkey
and l_extendedprice = m;
```

The new query is much more efficient, yet not as intuitive. Compilers sometimes manage to automatically decorrelate, but correlations can be complex (inequalities, disjunctions).

Instead of running a function every time, the subset is pre-computated trying to evaluate all the possible choices to then just use a lookup.

A join dependency allows to recreate a table by joining other tables which have subsets of the attributes (relationships are independent of each other). In other words, a table subject to join dependency can be decomposed in other tables and created again joining them.

Dependent joins rely on nested loop evaluations, which are very expensive: the goal of unnesting is to eliminate them all.

The easiest practice consists in pulling predicates up and creating temporary tables inside FROM instead of WHERE, evaluating the subquery for all possible bindings simultaneously.

The subquery in some cases is a set without duplicates which allows equivalence between dependent join and bindings of free variables, therefore dependency can be removed.

Sometimes attributes of the set can be inferred, computing a superset of the values and eliminating them with the final join. This avoids computing the subquery, but causes a potential loss of pruning the evaluation tree.

The approach is overall always working with relational algebra, yet not necessarily on the SQL level: it can add memory overhead which cannot always be removed, despite the ideal linear computational time.

To decorrelate subqueries, a general method can be used which is effective even with more complicated interrogations containing inequalities and several conditions.

The procedure can be executed several times, always starting from the highest level of nesting:

1. Moving the correlated subquery to FROM;
2. Since there might be multiple keys with the same value, putting DISTINCT helps reducing cardinality;
3. Joining is performed in the outer query, and if there is more than one value, an OR condition is applied so that either can be possible;
4. Values are grouped to maintain the same domain, according to the elements in DISTINCT and JOIN;
5. Other conditions taken from outside the subquery are applied after pre-aggregation.

The inner table should always have an unique keys and some groups, which then can be selected in the outer one.

Example:

```
select o_orderpriority ,count (*) as order_count
from orders
where o_orderdate >= date '1993-07-01'
and o_orderdate < date '1993-10-01'
and exists (
       select *
```

```
       from lineitem
       where l_orderkey = o_orderkey
       and l_commitdate < l_receiptdate)
group by o_orderpriority
order by o_orderpriority;
```

Decorrelated query:

```
select o_orderpriority, count(*) as order_count
from orders, (
       select o2.o_orderkey
       from lineitem, (
              select distinct o_orderkey
              from orders) o2
       where l_orderkey = o2.o_orderkey
       and l_commitdate < l_receiptdate
       group by o2.o_orderkey) preagg
where o_orderdate >= date '1993␣-07␣-01'
and o_orderdate < date '1993␣-10␣-01'
and orders.o_orderkey = preagg.o_orderkey
group by o_orderpriority
order by o_orderpriority;
```

6 Recursive CTEs

Recursive CTEs iteratively traverse hierarchical data of arbitrary length, and similarly to algorithms contain a base case (stop condition) and its recursive step. The number of times to iterate does not have to be constant.

```
with recursive r (i) as (
       select 1
       -- non-recursive term
       union all
       -- recursive term
       select i + 1
       from r
       where i < 5)
select * from r;
```

Recursive expressions can be used to manipulate data in a hierarchical structure. The non-recursive part defines the base case with eventual deterministic checks, while the recursive table traverses the tree upwards applying conditions.

Every recursive query can be defined by a simple evaluation algorithm:

```
workingTable = evaluateNonRecursive()
output workingTable
while workingTable is not empty
```

```
        workingTable = evaluateRecursive(workingTable)
        output workingTable
```

It is important to notice that the working table gets fully replaced at every iteration of the second part.

It is also possible to increment counters within RCTEs, and create trees. To avoid infinite loops in cyclic structures, using `UNION` instead of `UNION ALL` allows to remove duplicates and checking before writing a new value.

To avoid loops within graphs, a NOT EXISTS condition can be added to check whether a node has already been discovered.

Best practices to deal with graph-like data involve knowing its cardinality, to be aware of the maximum distance, yet most problems are Turing complete.

Postgres only allows one mention of the recursive relation in the recursive subquery.

7 Window functions

Window functions are versatile tools to implement time series analysis, percentiles and cumulative sums.

A window function can be seen as a special case of aggregate, evaluated after every clause but ORDER BY, but opposite to aggregation they do not change the result: only additional columns are computed.

The term window describes the set of rows on which the window operates, returning values from it.

Example (from Postgres documentation):

```
select product_name,
       price,
       group_name,
       avg(price) over (
                       partition by group_name
                       )
from products
       inner join product_groups
       using (group_id);
```

The previous query returns the product name, the price, product group name, along with the average prices of each product group.

AVG works as window function, operating on the subset of rows specified by OVER.

PARTITION BY distributes the rows of the result set into groups, each one of them subject to AVG, returning the average price.

Calculations on window functions are always performed on the result set after the other clauses but ORDER BY.

General window functions syntax:

```
window_function(arg1, arg2,..) OVER (
       [PARTITION BY partition_expression]
       [ORDER BY sort_expression [ASC | DESC] [NULLS {FIRST | LAST }]
       [frame_clause])
```

To summarize:

- The window function is an aggregate function, such as SUM or AVG;
- The (optional) partitioning clause divides the rows into multiple groups;
- The ordering clause specifies the order of internal rows;
- The framing clause defines a subset of rows in the current partition to which the window function is applied.

Multiple window functions can also be applied in the same query. Some other examples which ignore framing are:

- Ranking:
  - `rank()` and `dense_rank()`, returning the rank of the current row with or without gaps;
  - `row_number()`;
  - `ntile(n)`, distributing data over buckets;
- Distribution:
  - `percent_rank()` and `cume_dist()`, returning the relative rank of the current row or peer group (rows with equal partitioning);
- Navigation:
  - `lag(expr, offset, default)` and `lead(expr, offset, default)` evaluate the expression respectively on preceding and following row.

Framing can follow only a few specifications, not too complex:

- `current row`;
- `unbounded preceding` or `unbounded following`, first and last row in the partition;
- `range between unbounded preceding and current row`, default frame with order specified;
- `range between unbounded preceding and unbounded following`, default frame with order unspecified;
- `aggregates()`, computing aggregates over all tuples in current frame;
- `first_value()`, `last_value()`, `nth_value()`, evaluating an expression on determined rows of the frame.

7.1 Segment trees

Segment trees are an useful data structure for efficient evaluation, used for storing information related to intervals (segments).

Querying a static segment tree allows knowing which of the stored segments contains a given element, using logarithmic time operations.

The set of existing values gets divided into possible interval endpoints, sorted from left to right. All leaves in the tree correspond to the elementary intervals induced by endpoints, while internal nodes are the union of two intervals.

Any interval is stored in the canonical set for at most two nodes at the same depth, therefore the required storage is $O(n \log n)$ with n intervals.

Space for leaves is linear, and aggregated functions can be calculated on intervals very quickly.

Segment trees are useful for window functions, since intervals can be seen as windows, either static or sliding. For each call of a window function, a new segment tree gets built: the time for construction is just linear, since ordering is already performed by the ORDER BY clause.

7.2 Other aggregates

Newer version of Postgres allow calculation of more complex statistical aggregates, such as standard deviation, correlation and linear regression.

Mode and percentiles are also supported, yet they require materialization and sorting therefore they need a special syntax and the clause `within group`.

7.2.1 Grouping sets

A grouping set is a set of columns by which data is grouped, for instance using aggregates. They can be unified using UNION ALL.

Since uniting requires all sets to have the same number of columns, eventual inconsistencies must be filled with NULL: this is both lengthy and nonperforming (linear scans).

GROUPING SETS, a GROUP BY option, allows to define multiple grouping sets in the same query. It is possible to group by one or multiple columns, or even nothing. Missing values are automatically filled with NULL.

ROLLUP calculates all the grouping possibilities (2^n) and applies them.

8 Database clusters

Storing a large quantity of data on a single machine can have limitations: performance is restricted to hardware, and in case of failure transactions get lost.

Using multiple machines is therefore a valid solution to handle databases: clusters are transparent and present themselves as a single system, still maintaining consistency between nodes.

8.1 ACID vs BASE

ACID and BASE are two standards to describe the properties of databases. One could be defined as the opposite of the other, because their usage is distinct between relational models and NoSQL.

- Atomic: all operations in a transaction must either succeed or be reverted;

- Consistent: each transaction is always propagated to all nodes;
- Isolated: transactions do not contend with one another;
- Durable: results are permanent, even with failures.
- Basically Available: ensures an operating cluster yet not all data might be immediately accessible after a failure;
- Soft state: data needs to be periodically refreshed, it does not have a permanent state;
- Eventual consistency: consistency is not achieved all the time, but will be reached at some point after a number of updates.

Both models have advantages and disadvantages, but generally BASE properties are much looser and less strict than ACID.

BASE does not provide a consistent view of the data, which can lead to confusion and cannot be used for critical systems, therefore it should only be used when performance is more important than ACID guarantees.

8.2 CAP theorem

The CAP theorem states that is impossible for a distributed database to simultaneously achieve more than two of the following guarantees:

1. Consistency (every read receives either the most recent write or an error);
2. Availability (every request receives a response);
3. Partition tolerance (system operating even with dropped messages or delays).

Partition tolerance is not really an option: some systems cannot be partitioned (servers with no replication), and for a distributed system not guaranteeing partition tolerance would imply to never drop messages and never have a dying node.

The probability of any node to fail increases exponentially as the number of total nodes increase.

When a network partition failure happens, therefore, the choice has to be between canceling the operation decreasing availability or proceed propagating it risking consistency.

If a system chooses to provide consistency, it will preserve atomic reads but refuse to respond to some requests.

If, on the other hand, availability is chosen, the system will respond to all requests, potentially returning outdated information and accepting conflicting writes.

8.3 Transaction handling

In distributed systems, transactions should be atomic. This means there are two options for ending one:

- Commit, the transaction was successful and it can become persistent and visible to all nodes;
- Abort, the transaction failed or violates a constraint, therefore the system needs to be reverted to its previous state.

Since node crashes in a distributed system are independent, the protocol needs to be extended to ensure atomicity of transactions: a valid method is two-phase locking (2PL).

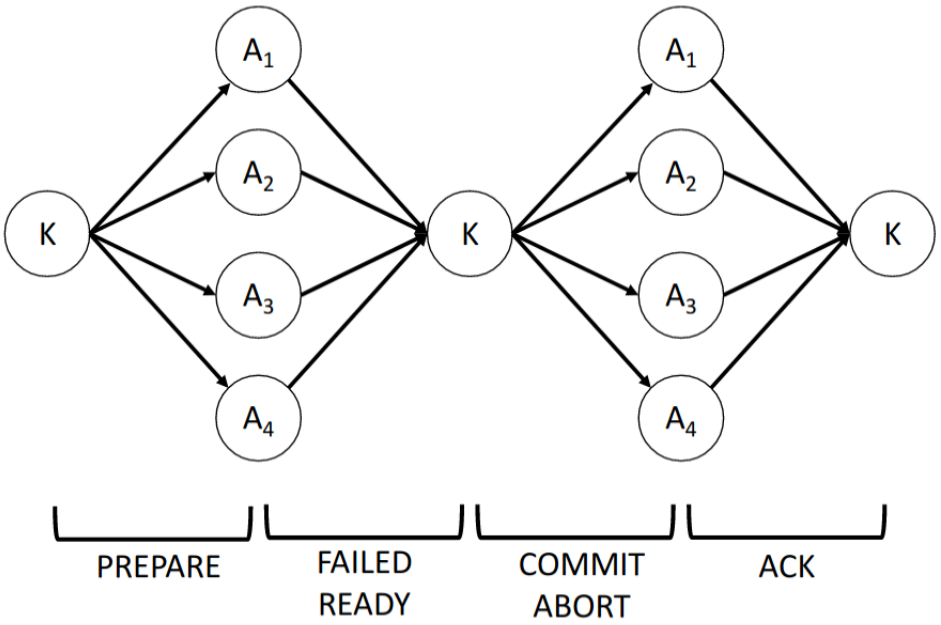
It is characterized by the presence of a coordinator node which allows n agents of a system $A_1, A_2, \ldots, A_n$ to either all persist the changes of T or all discard them.

This is a commitment protocol that coordinates all processes participating in a distributed atomic transaction, achieving its purpose using logging of the states to aid recovery procedures.

The two phases of the protocol in a normal execution are:

- Commit-request phase, in which a coordinator process attempts to prepare all the processes participating to the transaction to take the necessary steps and vote whether the operation can be committed;
- Commit phase, based on the votes, whose result gets communicated to the participants and the needed actions to commit or abort are taken.

Message flow in 2PL with 4 agents:



Issues of this method include the crash of nodes (both coordinator and agent) or potential lost messages.

8.4 Replication

Replication is a technique consisting in keeping a copy of the dataset in each machine. Redundant resources improve query throughput, can accommodate node failures and solve locality issues.

Access to a replicated entity is typically uniform with access to a single non-replicated entity, and copying should be transparent to external users.

Systems can also use a master-slave scheme, predominant in high availability clusters, where a single node is designated to process all requests.

8.4.1 Horizontal partitioning

Horizontal sharding involves every machine having only a chunk of the dataset. Query runtimes are improved, yet communication between shards should be minimized, due to network throughput.

Performance, in fact, relies heavily on the used interconnect, but allows nodes to have much less occupied memory so that dataset eveb bigger than the capacity of the machine can be accommodated.

Optimization of queries imply processing clauses separately, locally on each node, and aggregating results later so that the least amount of data is sent.

8.4.2 Shard allocation

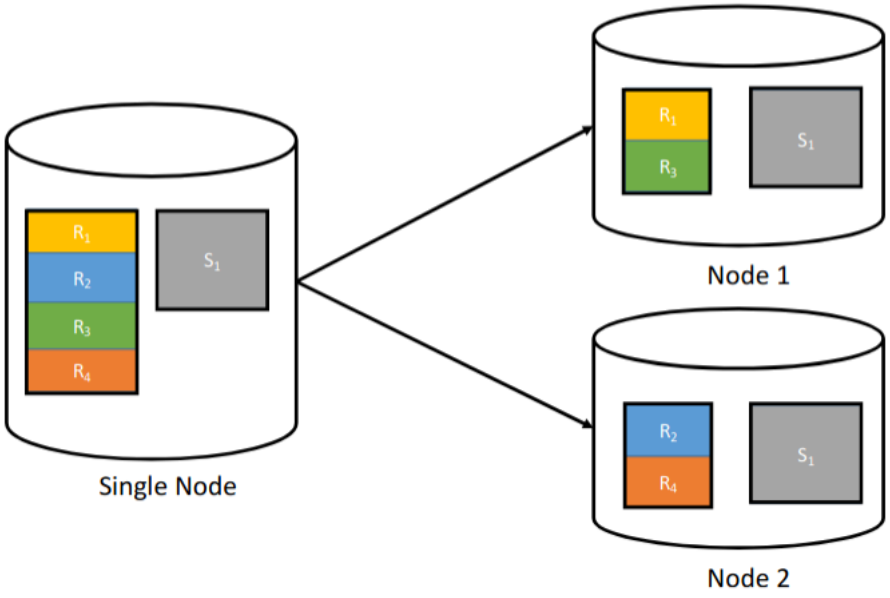
Shards can be allocated to the cluster machines in different ways:

- Every machine receives one shard;
- Every machine receives multiple shards.

Having more shards than machines results in better skew handling: they can take advantage of the resources available on nodes, increasing performance.

Also, there is no guarantee that the chosen hashing is uniform, and when values in between different shards are queried, it is likely that they are not all on the same machine.

Replication and sharding can be applied together, combining the benefits of both, but this leads to an increased resource consumption.



8.5 Joins

Joins in a distributed system are not different than local ones in presence of a replicated environment, yet can become complicated when data is sharded across multiple machines.

8.5.1 Co-located joins

A co-located join is the most efficient way to join two large distributed tables: every table has a common distributed column and is sharded across machines in the same way.

All rows with the same column, therefore, are always on the same machine, even across different tables, so that relational operations can be performed within the groups.

The joins can be executed in parallel locally on the workers, returning results to the coordinator node.

8.5.2 Distributed joins

A distributed join involves data which is sharded across several nodes.

Data is usually filtered as much as possible before being sent, but data movement is still necessary for joining: all nodes send their part of the table to a single one for it to run operations.

This join is only performing when the number of rows is small, otherwise the node doing the join will be overwhelmed by the dataset size and unable to parallelize.

8.5.3 Broadcast joins

A broadcast join is applicable when only one side of the join requires a small set of rows (small table or very selective filter).

The side of the join with the small set is sent to every node in the cluster, then joined locally with the large table on the other side of the join.

8.5.4 Considerations

All the previous methods still needs to consider size of the tables and bandwidth: some operations might be faster reading the data from remote machines rather than on local disks, since cache size is limited.

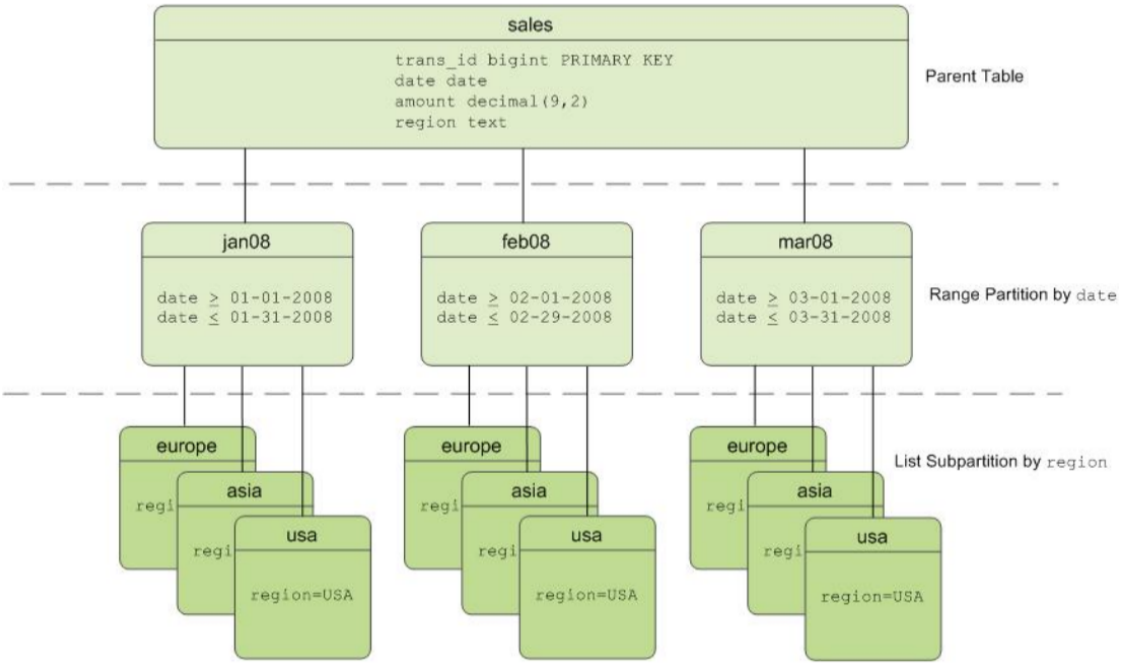
In this case, data can be shared to multiple nodes, each of them storing a partition of the table and computing local results which are then aggregated.

Postgres supports read replicas, kept in sync for consistent reads: each application can send data either to the master or the slaves, and the master itself can also trigger replication.

However, it is not natively a distributed database: its alternative Greenplum (still based on Postgres) allows data distribution and partitioning, both of data and schema.

Distributed and partitioning schemes can be specified creating a table, adding DISTRIBUTED BY to define how data is spread, or PARTITION BY for partitioning within a segment.

Distributions may be round-robin (as evenly as possible, non-deterministic), or hashed (rows dispersed according to a hash function).



MemSQL is another distributed database system based on SQL, which also allow replication, distributed and co-located joins.

A MemSQL cluster consists of aggregator and leaf nodes: aggregators handle metadata, monitoring and results of queries, while leaves act as storage layer and execute queries.

By default, MemSQL shards tables by their primary key, while manual sharding needs to be specified.

Distributed systems overall have more resources, but their synchronization might be expensive.

9 Autonomous database systems

Autonomous database systems are designed to remove the burden of managing DBMS from humans, focusing on physical database design and configuration/query tuning. The first attempts to program self-adaptive systems were in the 1970s, and have now evolved to learned components.

Indexing has been replacing with a neural network which predicts the location of a key, and transaction are scheduled through the machines by learned scheduling with unsupervised clustering. The most promising innovation is learned planning, deep learning algorithms which help running the query planner, estimating the best execution order of the operations.

A self driving DBMS is therefore a system that configures, manages and optimizes itself for the target database and its workload, using an objective function (throughput, latency) and some constraints. The two components of this are an agent (the one who makes decisions and learns over time) and an environment, the subject of the actions. Environments have a state used as feedback for the agent. This system can help exploring unknown configurations (and their consequences) and generalizing over applications or hardware.

9.1 State modeling

State modeling concerns the representation of the environment state, and has to be performed in an accurate way. The model is stochastic, non stationary and episodic (there is a fixed endpoint), and it can be represented with a Markov decision process model. The entire history of the DBMS is encoded in a state vector: its contents, design, workload and resources.

The table state contains number of tuples, attributes, cardinality and other values which altogether contribute to the choice of specific implementations (for instance indexes), but changes are constrained since the number of dimensions of the feature vectors always has to be the same. The content is approximated through a state vector, therefore there is no precise information regarding how the actual table looks like.

The knob configuration state depends on the machine, and is not really scalable. Not every configuration is applicable to servers, but one potential solution is to store hardware resources according to their percentages in respect of the available amount.

Feature vectors can be reduced (PCA), exacerbating instability, and hierarchical models help isolating components to reduce the propagation of changes.

Acquisition of data is done through targeted experiments while the system is running in production, training the model with end-to-end benchmarks of sample workloads. Instead of running the full system, micro-benchmarks can be run on a subset of components. This method is more effective and requires less iteration to converge.

To avoid slowing down the entire system, training is performed on replicas (master-slave architecture) having an agent recognize the resources and eventually propagating the best changes to the master. The application server primarily communicates with the master through reads and writes, and obtaining a complete view of all the operation is sometimes hard since not everything is sent to the replicas (failed queries, for instance).

An alternative is imitation learning: the model observes a tuned DBMS and tries to replicate those decisions. The state of the database still needs to be captured to extrapolate why changes have been applied, and training data will be sparse or noisy.

9.2 Reward observation

Rewards are classified as short-term and long-term, the latter more problematic since it is difficult to predict the workload trends. Transient hardware problems are hard to detect and could mask the true reward of an action, so current benchmarks have to be compared with historical data to reconsider recent events.

Many systems concern both OLTP and OLAP workloads, and changes in the objective functions may have a negative impact on either. Generally multiple policies define preference of one over th other in mixed environments.

Other common problems regard action selection policies, transparency and human interaction.

10 Hashing

To make Maths people feel better about this exam, with the courtesy of Harald Räcke.

The purpose of hashing is having a data structure that tries to directly compute the memory location from the given key, obtaining constant search time.

Hash functions and their associated hash tables are used in data storage and retrieval applications to access and store information thanks to their efficiency.

Given an universe U of keys and a set $S \leq U$ of used keys to be distributed over an array $T[0, \ldots, n-1]$, an optimal hash function $H : U \to [0, \ldots, n-1]$ should be fast to evaluate, have small storage requirements and a good distribution of elements over the whole table.

Ideally, the hash function would map all keys to different memory locations (direct addressing), yet this is unrealistic since the size of the table would be much larger than available memory.

Knowing previously the set S of actual keys, it could be possible to design a simple function that maps all of them to different memory locations (perfect hashing); but this is not often the case.

The best expectation would be a function which distributes keys evenly across the table, yet this leads to the problem of collision: usually the universe U is much larger than the table size n, hence there may be two elements k_1, k_2 from the same set S so that $h(k_1) = h(k_2)$, and therefore getting mapped to the same memory location.

Typically, collisions do not appear once the size of the set S of actual keys get close to n, but otherwise the probability of having a collision when hashing m elements under uniform hashing is at least:

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}}$$

Practically, hash tables should be larger than the amount of keys to store, for instance the next power of two. There also are scaling coefficients such as 1.5 or 1.5, to avoid resizing too much and wasting space. Precomputing prime numbers is an useful option, but requires a range of about 10% more than the number of keys.

There are methods to deal with collisions: open addressing, hashing with chaining, or just not resolving them at all.

Hashing with chaining arranges elements that map to the same position in a linear list, inserting at the front of the list. The average time required for an unsuccessful search is $A^- = 1 + \frac{m}{n}$, while for a successful search it is $A^+ \leq 1 + \frac{m/n}{2}$.

Open addressing, in particular, stores all objects in the table defining a function that determines the position through a permutation of all possible $n - 1$ values. If an insertion fails, a new position is searched (either through a different function or looking for the first free slot).

A good choice for $h(i, j)$ to have an uniform distribution over the possible values is a modulo with a prime number.

Introducing prime numbers is relevant because the size of hash table should not have factors in common with the linear component of the function, otherwise keys would be mapped to the same position. Increasing has also to be done by odd numbers, to reduce the possibility of remainder zero.

Prime numbers can be generated through algorithms, of which the most basic is the prime sieve: a list of integers up to a desired number is created, and composite numbers are progressively removed until only primes are left.

For large primes used in cryptography, a random chosen range of odd numbers of the desired size is sieved against a number of relatively small primes, and the remaining are subject to primality testing (Miller-Rabin).

To keep uniformity of the distribution even allowing new elements to be added, rehashing might be necessary, or incremental of the prime.

10.1 Fibonacci hashing

Fibonacci hashing is a variation of classic hashing using instead of a modulo operation, a multiplication by the golden ratio:

$$\frac{x}{y} = \frac{x+y}{x} \implies \phi = \frac{1+\sqrt{5}}{2}$$

The golden ratio has a close relationship with Fibonacci numbers: the previous equality, in fact, holds between every Fibonacci number and its successor.

The n^{th} Fibonacci number is obtained calculating $F_n = \frac{1}{\sqrt{5}}(\phi^n - \varphi^n)$, where $\phi = \frac{(1+\sqrt{5})}{2}$ and $\varphi = \frac{(1-\sqrt{5})}{2}$.

In the context of Fibonacci hashing, it is taken the reciprocal of ϕ, and hashing is performed multiplying the value k by the integer relatively prime to k which is closer to $\phi^{-1}k$.

Consecutive keys have an uniform spread distribution, therefore it is an efficient method.

10.2 Neojoin

Hashing is relevant in a distributed systems architecture because it can help determining how to locate relations on nodes.

While performing a distributed join, tuples may join with tuples on remote nodes: relations can be repartitioned and redistributed for local joins, so that tuples will only join with the corresponding partition.

The partitioning scheme defines how to assign partitions to nodes, using for instance a hash function. There are several options to implement a scheme:

1. Minimizing network traffic, assigning a partition to the node which owns its largest part. This way, only the small fragments of a partition are sent over the network, but a schedule with minimal network traffic may have high duration;
2. Minimizing response time, i. e. the time from request to result. This is dominated by network duration, which depends on maximum straggler;
3. Minimizing maximum straggler, formulated as a NP-hard linear programming problem where nodes have the most similar possible amount of workload.

Satisfying those constraints may lead to unused resources, for instance nodes not doing any work.

10.3 Chord hashing

Chord is a protocol for a peer-to-peer distributed hash table. it stores key-values assigning a node the values for which it is responsible, using a chord to assign and discover values.

Nodes and keys are given an hashed identifier of m bits, and using the Chord lookup protocol nodes are put in a circle that has as most 2^m nodes.

Each node has a successor and a predecessor, going in clockwise direction, but normally there are holes in the sequence. In general, the successor of a node k has the first identifier equal or greater than k.

Lookup is performed passing the query through the successors of a node, if the key cannot be found locally. This leads to linear worst-case performance, but can be avoided implementing a finger table in each node.

A finger table contains up to m entries, and entry i of node n contains the successor of key $(n+2^{i-1})$ mod 2^m. At every lookup, the query is passed to closest successor or predecessor of k in the finger table, narrowing worst-case performance to logarithmic.

Each peer is responsible for all keys larger than the predecessor number until its own. Introducing a factor of 2^k, the distance always increases by 2, halving the number of hops each time.

11 Distributed data processing

Distributed databases are a great solution to optimize memory and speed, yet not every operation is supported by them. Machine learning or graph algorithms, for instance, are hard to compute on a sharded relational schema.

Analysis can be implemented in several different ways, but certain tasks need to be handled in every implementation:

- Parallelization/synchronization;
- Distribution of computation and data;
- Communication between nodes;
- Node failures.

Creating a custom solution takes time and is error prone; functional abstractions can be used to specify computation in parallel. The runtime system, this way, manages program execution, data distribution and persistence, node failures and communication.

When a cluster is necessary, it is possible to use pre-configured structures:

1. Infrastructure as a Service, externalizing resources and storage;
2. Platform as a Service, renting databases and web servers;
3. Software as a Service, using developer tools and additional packages.

11.1 MapReduce

MapReduce is a programming model with associated implementation for processing and generating big data sets with a parallel distributed algorithm on a cluster.

It is composed of:

1. A Map procedure, which performs filtering and sorting, applied by each worker to the local data and collecting output to a temporary storage;
2. A Shuffle phase, where workers redistribute data based on the same output keys, so that all data with the same key is located on the same node;
3. A Reduce method, which executes a summary operation (such as counting) on local data in parallel.

The model is inspired by functional programming, but its key contributions are the scalability and fault tolerance achieved by optimizing the execution engine, especially on multi-threaded implementations.

Maps can be performed in parallel, and a set of reducers can perform the reduction phase. Parallelism offers possibility of recovering from partial failure of servers or storage: the work can be rescheduled without recomputing all of it.

Logically, Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$Map(k_1, v_1) \rightarrow list(k_2, v_2)$$

The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

$$Reduce(k_2, list(v_2)) \rightarrow list((k_3, v_3))$$

Each Reduce call typically produces either one key-value par or an empty return.

Example (from Wikipedia):

```
function map(String name, String document):
       // name: document name
       // document: document contents
       for each word w in document:
              emit (w, 1)

function reduce(String word, Iterator partialCounts):
       // word: a word
       // partialCounts: a list of aggregated partial counts
       sum = 0
       for each pc in partialCounts:
              sum += pc
       emit (word, sum)
```

This MapReduce performs a word count on a document. Each document is split into words, and every word is counted by Map; then all pairs with the same key are put together and sent to Reduce, which sums the relative counts.

MapReduce implementaton can be applied to many algorithms, such as joining of two relations or PageRank.

11.1.1 PageRank

PageRank is an algorithm used by Google Search to rank web pages in their search engine results, measuring the relevance of them.

It works assigning a numerical weight to each element of an hyperlinked set of documents, with the purpose of obtaining its relative importance within the set.

It outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. Links from a page to itself are ignored.

The PageRank transferred from a given page to the targets of its outbound links upon the next iteration is divided equally among all outbound links.

The value for a page v is dependent on the values for each page in the set containing all pages linking to v, divided by the number of links from v.

Pseudo-code PageRank:

1. Retrieve total number of nodes n;
2. Write initial values in each node, computed as $\frac{1}{n}$;
3. Find next node as $\frac{1-d}{n}$, where d is the correction factor;
4. For each iteration of the algorithm:
   (a) Compute outgoing page rank fraction;
   (b) Add fraction to all neighbors;
5. Swap next node and current;
6. Reset next reassigning it the value of $\frac{1-d}{n}$.

This algorithm looks at each node in the graph, computes the fraction of rank it has to pass on to neighbors, and writes it.

For better performance, the value can be computed in reverse, collecting all fractions from reverse neighbors.

The sequential reads are optimized for modern computers, but the random access to next node exhibits performance problems. Caches help alleviating this, since they can store most visited nodes.

If the memory requirements exceed the available main memory, the system will allocate too much memory, running into several page faults. Sorting by degree will make often-accessed nodes all in the same page.

The distributed version sends computed values to other nodes, but can cause overhead and network congestion. MapReduce assigns the Shuffle phase the duty of message passing.

Computing PageRank using MapReduce is also performed using the function to repeatedly multiply a vector by the matrix representing how the distribution changes after every single step.

11.2 Hadoop

Hadoop is an open-source implementation of the MapReduce framework, written in Java (opposed to the proprietary of Google in C++). It is now an Apache project used for very large clusters to handle big data sets.

11.2.1 MapReduce

Hadoop MapReduce is an application on the cluster that negotiates resources with YARN: a JobTracker globally manages jobs, and every node has its own TaskTracker.

1. An user submits job configuration to JobTracker;
2. JobTracker submits mapper tasks to TaskTrackers, scheduling computation on all available cluster nodes;
3. Each task works on the data local to its node;
4. The Shuffler moves output produced my Map among the nodes in the cluster;
5. Once shuffling is complete, JobTracker schedules Reduce to all nodes.

Since the datasets are much larger than the main memory and processing happens on very big clusters, intermediate results must be written to disk and shuffled results should be ordered.

MapReduce is however a framework not suitable for all uses: not everything can be expressed through the functions it offers, plus it is not a good fit for workloads with a lot of computation and smaller datasets.

11.2.2 HDFS

Hadoop relies on a distributed file system (HDFS), the primary data storage set used by its applications. It supports the rapid transfer of data between compute nodes, breaking information down into separate blocks and distributing it.

The file system ensures durability and data distribution, storing data read by Map, and pre-sorting and saving its output.

Shuffling moves data between nodes and merges pre-sorted data, and results of Reduce are again written to HDFS.

HDFS is highly fault tolerant: the system replicates data multiple times, placing at least one copy on a different server rack than the others.

Files are split into blocks of 64MB, and by default each block has 3 replicas, so that missing data can be reconstructed.

It follows a master-slave architecture: Name node manages file system and regulates access, while data nodes manage attached storage.

I/O is boosted by striping blocks of data over multiple disks, writing in parallel. HDFS exposes append-only files to users.

JobTracker is responsible to cope with failing nodes: it notices that one TaskTracker is not responsive, and its task gets reassigned to another node.

If its job is in Reduce phase, the correspondent function is restarted on other TaskTrackers.

11.2.3 YARN

YARN stands for Yet Another Resource Negotiator, and is a redesigned resource manager and large-scale distributed operating system for Hadoop.

YARN splits up the functionalities of resource management and job scheduling into separate daemons, to obtain a global resouce manager and a master for every application.

The ResourceManager and the NodeManager form the data computation framework. A NodeManager is the agent in every machine which is responsible for containers, monitoring their resource usage and reporting results to the ResourceManager.

The Scheduler is responsible for allocating resources to the various running applications subject to constraints such as capacity, performing no tracking of application status.

11.3 Optimizations

To optimize MapReduce it is possible to add a combiner, an optional class that operated by accepting input from Map and passing it to Reduce after summarizing records with the same key.

It operates on each Map output key, having the same key-value output as the Reducer, and it helps segregating data into multiple groups.

Folding is a function to save disk space: it analyzes a recursive data structure, and returns recombined results by recursively processing its constituent parts.

Programs working well on large clusters require a deep understanding of warehouse-sized computers: different settings need different techniques. In general, HDFS should be portable.

Overall, Hadoop considers hardware failure to be the norm, with applications needing streaming access on large datasets. Its model, therefore, uses a model aiming to minimize writes and keeps computation local with the data.

11.4 Scale up vs. out

A data center is usually made of computers piled in racks, and racks are connected via a switch to form a cluster.

Accessing data on remote machines comes with a price: the faster is a local DRAM (20GB/S), but cluster DRAMs and disks are relatively slow.

In case no single machine is large enough to store all the data, it is necessary to choose between scaling up and out: a cluster of large expensive machines or a large cluster of cheap commodity machines.

A simple execution model calculates the total cost summing the expense of computation with all global data access. The data kept locally is inversely proportional to the size of cluster.

$$TotalCost = \frac{1ms}{n} + f \times \left[100ns \times \frac{1}{n} + 100\mu s \times \left(1 - \frac{1}{n}\right)\right]$$

- Light communication: $f = 1$;
- Medium communication: $f = 10$;
- Heavy communication: $f = 100$.

Workloads with light communication benefit from large clusters, while large amounts of random communication impede performance. Random reads must be avoided!

11.5 Spark

Spark is an alternative to Hadoop which provides more expressive fault-tolerant runtime for distributed batched data analytics. It can handle more complex analysis with interactive queries processing real time streams.

The purpose of a new programming model is extending the existing naive algorithms implementation:

- Data can be outputted by sorted key;
- Locality can be increased using different kinds of partitioning;
- Window queries can be answered sorting by group.

The MapReduce model is extended too with complex data pipelines, but recovery mechanisms are lightweight and require less additional computation.

Sparks offers additional API for more functionalities, such as Spark SQL, GraphX and MLIB.

11.5.1 RDDs

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel.

RDD is considered the evolution of the general Spark model, offering datasets that can be efficiently cached and manipulated.

RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset.

All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file).

They are only computed when an action requires a result to be returned to the driver program, and the dataset gets only loaded when operations are performed on it. This design enables Spark to run more efficiently.

Pipelines allow to chain transformations and intermediate steps do not need to be materialized since they can just be recomputed on node failure.

11.5.2 DataFrames

DatFrames are dynamically strongly typed distributed collections of data organized into columns, conceptually equivalent to a table in a relational database.

They can be interacted with using SQL functions, allowing query optimization and avoiding interpretation overhead, although with a limited set of functions.

11.5.3 Datasets

A Dataset is a combination between DataFrame and RDD, optionally weakly typed, to retain code compilation. It is still mapped as a RDD.

Datasets can be created, analyzed (limited, filtered) and be subject of exploration, management or aggregation just like tables.

11.5.4 Optimization

RDDs track lineage information to rebuild lost data, logging everything in a graph to recompute missing or damaged partitions. Spark offers support for DAGs.

Snapshots are automatically created after repartitioning transformations and managed using a LRU cache to speed up recovery.

The optimizer builds a logical plan and applies transformation rules, to then generate the physical plan and select the cheapest cost model.

Spark's runtime focuses on three main points:

- Memory Management and Binary Processing, leveraging semantics to eliminate the overhead of deserialization and garbage collection;
- Cache-aware computation, using columnar storage and compression to exploit memory hierarchy;
- Code generation to optimize usage of modern compilers and CPU.

Operations to take care of reading, managing ETL pipelines, training and query are performed within the same framework.

12 Key-value storage

One of the benefits of this kind of storage is the lack of schema: they are defined with human readable languages (XML, SVG) for structured and semi-structured data. The possible formats vary from free text to relational models, the latter less common; data can also be compressed or uncompressed.

For instance, possible choices for configuration files are:

1. JSON, although kind of hard to write long files;
2. YAML, really complicated;
3. TOML, with minimal syntax therefore easy.

12.1 XML

XML documents have a rooted hierarchical structure named Document Object Model, where elements are declared between tags with eventual attributes. Those documents allow to store reasonably large information, queried in a declarative language (XPath, XQuery) which guarantees results appearing in the same order as they were saved. Validation requires constraints which imply the choice between a simple grammar or a more powerful one. Grammar is only defined for tags and attributes, not contents.

An XPath is the tree containing elements subject of a query: it stores information about preceding nodes, siblings, descendants, parents and ancestors in the DOM. It can be navigated through the syntax `axis::node[predicate]/@attr` also allowing retrieval of a set related to the matching node (for instance `:.` indicates parent).

Grammar definition and DTD are defined in the headers of a document, although parsing can still be difficult due to malfunctioning web pages.

12.2 JSON

JSON is a language similar to XML, but closer to a relational database: its structure consists in objects and arrays recursively nested in an arbitrarily complex way. It can be used for a broad range of data types and evaluated like JavaScript (not the best practice for security issues). Keys are between quotes to avoid referencing variables.

Data can be accessed through indexes and JavaScript syntax.

12.3 Other

Schema is similar to XSD and has the same syntax as JSON, yet it is rarely used.

Transact-SQL is the Microsoft JSON query language.

13 Resource Description Framework

RDF is a W3C standard to define semantics of resources (web pages) with URIs as keys, mostly used for arbitrarily connect semantics and information in a graph.

The smallest structure in RDF is a triple of `subject, predicate, object`, each of them represented as a node in the graph (or a column in a table).

Information is linked using common data or models, often categorized in a hierarchical manner. Ontologies define objects in an official way.

RDF data is stored by serialization in different formats: N3 notation avoids repetition by defining URI prefixes and storing multiple predicate-object pairs without repeating the subject.

Microformats annotate (X)HTML documents with RDF snippets, not very human readable.

Data types are similar to the primitives commonly found in databases and programming languages (integer, string, ...) but can also be defined by users.

Similarly to domain calculus, there is no equal condition (when joining, for instance) since the check is implicit thanks to the usage of the variable two times, requesting the same identifier. Queries can be translated to relational algebra, binding every WHERE clause with a join just above in the tree.

This property makes it easier to iteratively change data and schema, connecting them to other sources, but on the other side it's harder to optimize storage and queries.

14 SPARQL

SPARQL is the RDF query language, working with pattern matching over triples and returning the matching ones. It only contains basic CRUD functionalities with three types of data: URI, literal and blank node. The latter represents a variable with local scope, not appearing in SELECT.

The subject and the predicate must always be an URI, while objects can also be literals. Selection can be over distinct or reduced values, which allows to convert inner join to left outer, ignoring how many matchings are there.

SPARQL permits advanced SELECT queries such as FILTER, containment check or output as a graph. Examples of semantic information that can be queried are found on DBPedia.

14.1 Query processing

On a large graph, queries could be performed simulating a relation database, loading triples in a table: this approach does not work since the database is not aware of the types of stored data.

RDF graphs are usually implemented through clustered B^+-trees, storing triples in lexicographical order allowing good compression (encoding differences with the previous node) and fast lookup on disk.

Sorting requires a definition of the ordering field: since space is optimized due to compression, it is sometimes possible to create different trees according to each possibility ($3! = 6$), leading to efficient merge joins.

Queries can be performed in linear time through the following optimizations:

- Extensive compression of trees;
- Indexing for joins;
- Information skipping large parts of data while traversing;
- Encoded data.

RDF triples are closely related, disallowing independence. Soft functional dependencies are indeed very common, with bind triple patterns making the others unselective and not captured by histograms.

Cardinality estimation is not well suited for RDF data: the independence assumption does not hold (correlation is the norm), and leads to severe underestimation. Multi-dimensional histograms would help, but are really expensive to compute due to exponential growth of dimensions.

RDF is also really unfriendly for sampling, since there is no schema: triples can be diverse and unrepresentative, therefore a sample needs to be huge to be useful.

15 Graph databases

A graph $G = (V, E)$ is a set of edges and vertices which can have different properties. Edges can also have intermediate ramifications between nodes.

Useful information regarding a graph can be:

- Degree;

- Centrality;
- Betweeness;
- todo

Graph databases can be implemented using semantic data in triples, introducing properties of nodes.

There is no defined standard for structure of graphs, in fact they can also be implemented using relational databases, yet some tools are more popular then others.

Queries on graph databases do not strictly return a set of tuples: they allow pattern matching, algorithms, aggregation, or properties of graph and its elements.

15.1 Cypher

Cypher is one of the most supported query languages for graphs. Its syntax allows nodes and edges to have a type along with some attributes.

Neo4j, implementing Cypher, is a graph database offering dynamic visualization and machine learning algorithms, but it kind of sucks.

15.2 Pregel/Giraph

Pregel is Google's iterative graph processing system for distributed data, while Giraph is its open source alternative.

They work using Gremlin, a functional graph traversal language, which with one operation lets a vertex modify its state and propagate information.

The reason why this tool has been developed is because MapReduce systems do not allow operations on a vertex level. In addition, MapReduce requires transferring state to all nodes.

15.3 Turi

todo

15.4 Graph algorithms

BFS is one of the most important graph traversal algorithms to search a node

Dijkstra yeah we know that cmon Neumann do databases