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 problem with this are the proprietary APIs wit 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 can be highly explicative.

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

Naively optimising 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`).

Performance when manipulating text completely ignores CPU costs, which are not important for disks but very relevant for reading data from DRAM: this option is one of the fastest, followed by SSD.

A good 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).

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.

Optimisation of mathematical operations is done using several strategies:

- By the compiler, choosing fastest registry operations (64-bit is faster than 32);
- $a/c = a * (2^{64}/c) >> 64$;
- `const` instead of `constexpr`;
- ...

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.

Multithreading consistently increases performance - but network speed and remote CPUs can cause connection problems.

4 SQL

SQL is a declarative (orders to execute a query without specifying how) language to interact with databases and retrieve data. Optimisation 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

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

Regular expression matching is performed through `LIKE` or $\sim$, but cannot run in linear time since the construction of the NFA is exponential in the size of input.

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

Views create faster reusable tables, but are seen globally and this could cause issues with naming. Another way to optimize through a temporary table is `WITH`, a common table expression.

CTEs are performed only once within each query and destroyed as soon as the query ends on Postgres, yet other database systems may differ in implementation.

Recursive CTEs traverse hierarchical data of arbitrary length, and consist in a base case (stop condition) and its recursive step. To manipulate values before the previous one, it is useful to store them in columns. It is possible to increment counters within RCTEs, and create trees.

To avoid infinite loops, using `UNION` instead of `UNION ALL` allows to remove duplicates and checking before writing a new value.