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.

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.

UNION, EXCEPT and INTERSECT perform set operations, with the keyword ALL to keep duplicates.

4.2 Views and CTEs

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.

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 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.

4.6.1 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.

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.

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).

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.

6 Recursive CTEs

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.

7 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.

7.1 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.

7.2 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.

8 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.

8.1 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.

8.2 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.

8.3 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.

9 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.

10 SPARQL

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.

11 Hashing

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.

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.