Big Data for Engineers (Spring 2020)

Summary

Author:
Yannick Merkli
9 Pages

Contents

1 Introduction

1.1 History

Databases have always existed in some way as a way to preserve information. It started with speaking and singing, went on with stone engraving and printing. Even before computers, tables were the primary format to represent data. Things changed drastically with the introduction of computers. Database management systems (DBMS) started with file systems (1960s), then we entered the relational era (1970s) and finally progressed into the NoSQL era (2000s) with the upcoming of big data.

1.2 Big Data

Big Data is a buzzword that goes across many disciplines (distributed systems, high-performance computing, data management, algorithms, statistics, machine learning, etc.) and that involves a lot of proprietary technology (AWS, Google Cloud, Microsoft Azure, etc.) which is simply a result of the need of companies to have efficient data systems.
The big in big data: **three Vs**

- Volume: Nowadays we have lots of sources of data (web, sensors, proprietary, scientific). Storage has become so cheap that we often just store data because we can. Further, data carries value; data is worth more than the sum of its parts (data totality: one must have complete data).
- Variety: We have different **data shapes**: tables, trees, graphs, cubes, text.
- Velocity: Data is generated automatically, Data is a realtime byproduct of human activity

Prefixes (International System of Units)

kilo (k)	1,000 (3 zeros)	kibi (ki)	1,024 (2^{10})
Mega (M)	1,000,000 (6 zeros)	Mebi (Mi)	1,048,576 (2^{20})
Giga (G)	1,000,000,000 (9 zeros)	Gibi (Gi)	1,073,741,824 (2^{30})
Tera (T)	1,000,000,000,000 (12 zeros)	Tebi (Ti)	1,099,511,627,776 (2^{40})
Peta (P)	1,000,000,000,000,000 (15 zeros)	Pebi (Pi)	1,125,899,906,842,624 (2^{50})
Exa (E)	1,000,000,000,000,000,000 (18 zeros)	Exbi (Ei)	1,152,921,504,606,846,976 (2^{60})
Zetta (Z)	1,000,000,000,000,000,000,000 (21 zeros)	Zebi (Zi)	1,180,591,620,717,411,303,424 (2^{70})
Yotta (Y)	1,000,000,000,000,000,000,000,000 (24 zeros)	Yobi (Yi)	1,208,925,819,614,629,174,706,176 (2^{80})

There are three paramount factors to big data:

- Capacity: "How much data can we store?"
- Throughput: "How fast can we transmit data?"
- Latency: "When do I start receiving data?"

Capacity has improved incredibly much over the past 60 years (1956: huge HDD had 5MB storage, 2020: there are palm sized 20TB HDDs). Capacity has increased by a factor $200 * 10^9$ (per unit of volume). However, throughput and latency have only improved by a factor $10'000$ and 200 respectively. This discrepancy creates problems: the throughput no longer scales to the amount of data and the latency no longer scales to the throughput. Solution:

- Capacity-throughput discrepancy: parallelization
- Throughput-latency discrepancy: batch processing

What is big data?
Big Data is a portfolio of technologies that were designed to store, manage and analyze data that is too large to fit on a single machine while accommodating for the issue of growing discrepancy between capacity, throughput and latency.

2 Lessons learnt from the past

Data Independence:
An underlying principle that has been valid for a long time is the principle of **data independence** (developed by Edgar Codd): Data Independence is defined as a property of DBMS that helps you to change the Database schema at one level of a database system without requiring to change the schema at the next higher level. Data independence helps you to keep data separated from all programs that make use of it.
This means we could e.g. change the physical storage (e.g. iPad instead of HDD) *without* changing the logical data model.

Overall architecture
The overall architecture of a DBMS consists of:

- Language (e.g. SQL)
- Model (e.g. Tables (old); graphs, trees, cubes (new)
- Compute (e.g. single CPU (old); hadoop cluster (new))
- Storage (e.g. HDD (old); distributed storage (new)

A data model essentially describes *what data looks like* and *what you can do with the data.*

2.1 Basic concepts

- Table (Collection): A set of rows (= business object, item, entity, document, record) and each row has attributes (= columns)
- Attribute (column, field, property): A certain attribute of a row
- Primary key (row ID, name): a unique identifier of a row

2.2 Relational Algebra

We can look at tables as relations or as partial functions, mapping property to value
($f \in \mathbb{S} \nrightarrow \mathbb{V}$), e.g. $city \mapsto Zurich$.

2.2.1 Relations (the math, for database scientists)

A relation R is made of:

- A set of attributes: $Attributes_R \subseteq \mathbb{S}$
- An extension (set of tuples):

$$Extension_R \subseteq \mathbb{S} \nrightarrow \mathbb{V} \quad s.t. \quad \forall t \in Extension_R, support(t) = Attributes_R$$

Tabular integrity: Holds if all rows have the same attributes and have a valid value for the attributes.
Atomic integrity: No tables in tables.
Domain integrity: All attribute values are of the specified type (e.g. an attribute *Name* of type string can't be an integer).

In SQL, tabular integrity, atomic integrity and domain integrity all hold. In NoSQL however, none of these three properties hold.

2.3 Relational queries

The following table shows the operators used in Relational Algebra.

Selection	σ	**Set Minus**	$-$	**right Semi-Join**	$\rtimes$
Projection	π	**Relational Division**	$\div$	**left Outer Join**	$\rdsh$
Cartesian Product	$\times$	**Union**	$\cup$	**right outer Join**	$\ldsh$
Join	$\bowtie$	**Intersection**	$\cap$		
Rename	ρ	**left Semi-Join**	$\ltimes$		

Projection: The projection operator is used to produce from a relation R a new relation that has only some of R's columns. The value of expression $\prod_{A_1,...,A_n}(R)$ is a relation that only consists of columns for the attributes $A_1, ..., A_n$.
Selection: The selection operator applied to R produces a new relation with a subset of R's tuples, namely those who meet some condition C. This operation is denoted by $\sigma_C(R)$.
Cartesian product: The Cartesian product of relations R, S, denoted $R \times S$, simply concatenates every possible combination of tuples $r \in R, s \in S$. If R, S have attributes in common: rename them. In practice rarely used without join operators.
Natural Join: The natural join of relations L, R, denoted $L \bowtie R$, pairs only those tuples from L and R that agree in whatever attributes they share commonly. Natural Join is associative! Other variants:

- Left outer join: natural join & unmatched tuples from L
- Right outer join: natural join & unmatched tuples from R
- Full outer join: natural join & unmatched tuples from both L and R
- Left semi join: tuples from L that match with some tuple in R
- Right semi join: tuples from R that match with some tuple in L

Theta-Join: A theta join $\bowtie_\theta$ allows to join tuples from two relations R, S based on an arbitrary condition θ rather than solely based on attribute agreement. We get this new relation by:

1. Take the Cartesian product $R \times S$
2. select those tuples satisfying condition θ

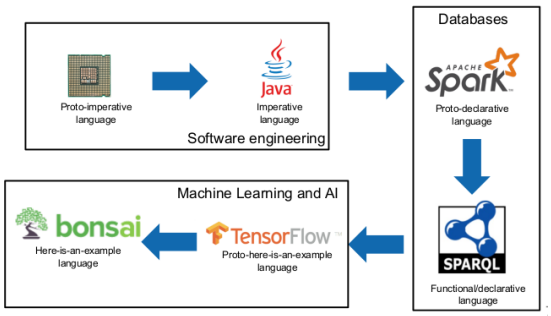
Union, Intersection, Set Minus: requires: both relations have the same schema $\rightarrow$ then consider set of tuples, do corresponding set operations. Note that $R \cap S = R - (R - S)$

Rename:

- to change the name of the relation R to S, we write $\rho_S(R)$.
- to rename attributes of R, we use the operator $\rho_{(A_1,..,A_n)}(R)$ where the attributes in the result relation S are called $A_1, ..., A_n$, respectively.

2.4 Terminology

Data: Data Manipulation Language (DML) (Query, insert, remove rows) Schema: Data Definition Language (DDL) (Create or table/schema, drop it)



(a) Language landscape

Lots of rows	Object Storage
Lots of rows	Distributed File Systems
Lots of nesting	Syntax
Lots of rows/columns	Column storage
Lots of nesting	Data Models
Lots of rows	Massive Parallel Processing
Lots of nesting	Document Stores
Lots of nesting	Querying

(b) Data Scale-Up

2.5 Transactions

The good old times of databases: ACID (Atomicity, Consistency, Isolation, Durability).

- Atomicity: Either the entire transaction is applied, or none of it (rollback).
- Consistency: After a transaction, the database is in a consistent state again.
- Isolation: A transaction feels like nobody else is writing to the database.
- Durability: Updates made do not disappear again.

2.6 Performance

Optimize for read vs. write intensive:

- OnLine Transaction Processing (OLTP): Write-intensive
- OnLine Analytical Processing (OLAP): Read-intensive

There is no such thing as "one size fits all" - data shape matters.

2.7 Data scale-up

Data can have lots of rows, lots of columns and lots of nesting. For the rest of this lecture, we are concerned with exactly this: scaling up!

3 SQL

SQL is a family of standards, namely it includes a data definition language for schemas, a data manipulation language for updates and a query language for reads. Note that SQL is case-insensitive.

3.1 DDL: Data Definition Language

The following code snippet shows the most important data definitions in SQL:

```
character (n), char (n) -- String of length n
varchar(n) -- String of variable length but at most n characters
numeric (p,s) -- decimal value with precision p and scale s
integer -- numeric values
blob, raw -- large binaries
date -- date value
clob -- large string tables
string1 || string2 -- concatenation operator for strings

CREATE TABLE Professor -- create a new table
    (PersNr      integer not null,
     Name        varchar (30) not null,
     Status      character (2) default "AP");
DROP TABLE Professor -- delete table
ALTER TABLE Professor add column(age integer); -- adds new age column to table

CREATE INDEX myIndex on Professor(name, age); -- Index for performance tuning
DROP INDEX myIndex -- delete index
```

3.2 DML: Data Manipulation Language

A real database cannot be manually populated in a tuple-by-tuple manner $\implies$ **ETL: Extract, Transform, Load** is used to extract data from some file, which is then transformed to the data types supported by the database and then loaded into to database as a *bulk* operation. Manual updates work as follows:

```
insert into Student (Legi, Name)          -- standard insertion syntax
    values (16-940-165, 'Jens Eirik Saethre');

-- can also use a nested query to determine the tuples to be inserted.
insert into attends (Legi, 'Databases')
    select Legi
    from Student
    where semester > 2;

delete Student -- standard deletion syntax
where Semester > 13;

update Student -- update all tuples in Student relation table
    set Semester = Semester + 1;

-- there are sequence types for automatic increment (e.g. useful for unique IDs)
create sequence PersNr_seq increment by 1 start with 1;
insert into Professor(PersNr, Name)
    values (PersNr_seq.nextval, "Roscoe");
```

3.3 Simple Queries in SQL

Projection & Selection

The simplest query form in SQL selects tuples from a single relation with some selection criteria and selects only the intersting columns. This query thus combines both *Selection* and *Projection*:

```
select   PersNr, Name              -- use * to select all columns
from        Professor
where    Status = 'FP';
```

One can leave out the `where` clause to get the entire table, one can sort the table by appending the line `order by Status desc, Name asc` or select only distinct entries with the keyword `select distinct`.

One can also rename attributes by writing `select PersNr as ProfNr` and use expressions in the `SELECT` (note that SQL is *case-insensitive*) statement to modify the concerned column. The following example of a query in a movie database combines all of the aforementioned special cases:

```
SELECT DISTINCT release AS releaseDate, length*0.16667 AS runtime
FROM            Movies
WHERE           genre = 'Thriller' AND (rating >= 9.0 OR rating <= 2.0);
```

It returns the release date and the runtime in hours of the best and worst-rated movies in the Thriller genre stored in the database. Note that if there are multiple movies that satisfy the condition with the same (ReleaseDate, runtime) pair, only one tuple will appear in the returned relational table.
$\Longrightarrow$ like this, one can also add constant columns to the output by appending to the `SELECT` clause the following: `'hrs.' AS inHours`. Now the table will have three columns, the last one being the string 'hrs.' for all tuples.

Note: `<>` is SQL syntax for "not equal to" and `=` for "equal to".

SQL relates to relational algebra in the sense that a query `SELECT L FROM R WHERE C` is equivalent to the relational algebra expression $\prod_L (\sigma_C(R))$

String Comparison

- comparisons of `varchar` and `char` only compares the actual string and not the padding.
- comparison with an operator like $>$ compare the string's lexicographical order.

Pattern Matching

We can use the pattern matching operator `s LIKE p` to compare a string s to a pattern p:

- ordinary characters in p match only themselves in s
- a _ symbol in p matches one arbitrary character in s
- a % symbol in p matches an arbitrary sequence of characters in s, also with length 0.

Dates and Times

- **Dates** are represented in the format `DATE '1996-12-06'`
- **Times** are represented in the format `TIME '15:23:05.4'`.
- One can combine the two to get a new type `TIMESTAMP '1996-12-06 15:23:05.4'`

`NULL` value in SQL

The *null value* in SQL can either mean that the value is *unknown*, *inapplicable* or *withheld*.

- arithmetic operations that include `NULL` evaluate to `NULL`
- comparisons that include `NULL` evaluate to the truth value `UNKNOWN`
- the `group by` operator returns a group for `NULL`

Note: `NULL` is **not** a constant, we cannot use `NULL` explicitly as an operand.

`UNKNOWN` value in SQL

Assume `TRUE = 1, FALSE = 0, UNKNOWN = 1/2`. Then we have the following rules for logical operators:

- **AND:** minimum of the two values
- **OR:** maximum of the two values
- **NOT:** $1 - v$ where v was the previous value.

When selecting tuples, only those whose truth value to the query is `TRUE` are picked for the resulting relation. $\Longrightarrow$ the following query does not necessarily return all movies:

```
SELECT *
FROM Movies
WHERE length <= 120 OR length > 120
```

If a movie's length is in fact UNKNOWN, than that tuple will not be returned.

Sorting of Tuples

Can use the ORDER BY <list of attributes> clause to order the returned tuples. Ordering is applied before the SELECT part, so it is possible to order on attributes that do not appear in the final output. Impose an order by using ASC, DESC, ascending being default.

3.4 Queries on multiple Relations

Set Operations

We use the keywords UNION, INTERSECT, and EXCEPT for $\cup, \cap, -$, respectively. The syntax is generally as follows: (Query1) INTERSECT (Query2);.

Products

To get the Cartesian product of two relations, simply provide both as arguments to the FROM clause, e.g. SELECT * FROM Movies, Stars.

Joins

Use the product learned above and define in the WHERE clause which attributes should match, e.g. the following query returns the names of the producer of 'Star Wars', when the movie name and the producer name are stored in different relations:

```
SELECT  name
FROM    Movies, MovieProducers
WHERE   title = 'Star Wars' AND producerNr = certificateNr;
```

If attributes of different relations have the same name, we reach non-ambiguity by using the *dot-notation* on the attributes: RelationName.AttributeName. This is generally good practice, even when there is no ambiguity (yet).

Tuple Variables

If a query involves two ore more tuples from the same relation, we need an **alias** to refer to them individually. This is achieved by the following syntax:

```
SELECT      Star1.name, Star2.name
FROM        MovieStar Star1, MovieStar Star2
WHERE       Star1.address = Star2.address AND Star1.name < Star2.name;
```

Note: 1.) even though Star1 appears in the statement before it is defined, this is okay since the FROM part is evaluated earlier. 2.) the string comparison is necessary to avoid pairing people with themselves. Further, comparison by <> would produce each pair twice.

A problem with the semantics may arise when we think that we can use simple SQL queries for set operations like $R \cap (S \cup T)$. Assume R, S, T are all unary relations with attribute A. One would think that the following query works perfectly fine:

```
SELECT R.A  FROM R, S, T      WHERE R.A = S.A OR R.A = T.A;
```

However, if T were to be empty, one would expect the result to be $R \cap S$, but the query returns nothing. This is due to the semantics of multi-relational queries and how they are implemented.

3.5 Full-Relation Operations

Some operations act on relations as a whole an not just on single tuples. We have already looked at the DISTINCT keyword to eliminate duplicates in query results. Further, set operations eliminate duplicates by default. If this is not desired $\Longrightarrow$ use the ALL keyword, e.g. for relations R, S:

```
R INTERSECT ALL S
```

returns the intersection of "bags"

Grouping and Aggregation

We often want to partition the tuples into groups on which we can then apply an aggregation operator, like one of the following:

- `avg, max, min, count, sum`

We can then do grouping by the clause `GROUP BY` which follows the `WHERE` clause. We can e.g. sum all the lengths of movies given studios have produced by the query:

```
SELECT studioName, SUM(length)
FROM Movies
GROUP BY studioName;
```

Note that `NULL` values are ignored in any aggregation (e.g. in `avg` it would make a difference, in `count(*)` it only counts non-`NULL` values). However, `NULL` is treated as an ordinary group.

`HAVING` clauses

This clause is used when we want to choose our groups based on some aggregate property of the group itself $\implies$ the `HAVING` clause then follows the `GROUP BY`. Example:

```
SELECT  name, SUM(length)
FROM    Movies, MovieProducers
WHERE   producerNr = certificateNr
GROUP BY name
HAVING MIN(year) < 1930;
```

This query prints the names of producers with their total movie lengths which have at least produced one movie prior to 1930.

Note: one can aggregate any attribute appearing in the relations declared in the `FROM` clause, but one can only use attributes that are in the `GROUP BY` list in an unaggregated way.

3.6 Subqueries

A query can be part of another query itself $\implies$ already saw example in set operations.

- Subqueries can return a single constant $\implies$ use in `WHERE` clauses
- Subqueries can return relations used in `WHERE` clauses
- Subqueries can appear in `FROM` clauses followed by a tuple variable

Conditions involving Relations

We first assume unary = one-column relations for simplicity:

- `EXISTS R` is true $\iff$ R is not empty
- `s IN R` is true $\iff$ s is equal to at least 1 value in R. Can also use the `NOT IN` operator
- `s > ALL R` is true $\iff$ s greater than every value in R.
- `s > ANY R` is true $\iff$ s not the smallest value in R.

Note: if R returns zero rows, then any comparison with `ALL` clause returns true.

Conditions involving Tuples

A tuple in SQL is a parenthesized list of scalar values e.g. `(name, address, networth)` $\implies$ if tuple t has same number of components as relation R, we can compare it by using e.g. `t IN R` or `t <> ANY R`, the latter asking if there exists any tuple in R that is other than t.

To find all producers of movies where Leonardo DiCaprio starred in, one can use the following query:

```sql
SELECT  name
FROM    MovieProducers
WHERE   certificateNR IN
    (SELECT producerNR
     FROM   Movies
     WHERE  (title, year) IN
        (SELECT movieTitle, movieYear
         FROM   StarsIn
         WHERE  starName = 'Leonardo DiCaprio'));
```

Queries like these should be analysed from inside out.

Correlated Subqueries

In a subquery, we can refer to e.g. an attribute from the relation of the `FROM` clause from the super-query. We can refer to it by using an *alias*. This subquery then needs to be executed every time for each tuple in the super-query. An example would be:

```sql
SELECT  title
FROM    Movies Old
WHERE   year < ANY
    (SELECT year
     FROM   MOVIES
     WHERE  title = Old.title);
```

This query finds all movie titles that appeared multiple times. The subquery needs to be executed for every tuple in Movies that we are looking at. *Scoping rules* work similar to other mainstream programming languages.

Subqueries in `FROM` clauses

Can use a parenthesized query instead of a stored relation $\implies$ use an *alias* for the result of the query to be able to refer to it.

Problem with Universal Quantification

SQL does not support relational division directly, that's why one has to play tricks, namely using the following:

$$\forall t.P(t) \equiv \neg\exists t.\neg P(t) \qquad \text{and} \qquad R \implies T \equiv \neg R \vee T$$

Syntactic Sugar

```sql
SELECT  *
FROM    STUDENT
WHERE   semester between 1 and 6; -- variant 1
WHERE   semester in (2,4,6);     -- variant 2

-- case statement can be implemented like this. note that only one clause executes,
   meaning no need for break
SELECT  legi, (CASE WHEN grade >= 5.0 THEN 'gut'
                    WHEN grade >= 4.0 THEN 'ausreichend'
                    else 'nicht bestanden' END)
FROM        Tests;
```

Joins in SQL

```sql
-- Cartesian Product
Movies CROSS JOIN StarsIn;

-- Theta Join (still has all the columns from both relations -> Preceed by a SELECT
   statement to reduce redundancy)
Movies JOIN StarsIn ON
    title = movieTitle AND year = movieYear; -- theta condition

```

```
-- Natural Join (joins on all attributes with the same name)
MovieStar NATURAL JOIN MovieProducer

-- OuterJoins (FULL, LEFT or RIGHT are interchangeable in this example)
MovieStar NATURAL FULL OUTER JOIN MovieExec;
```

SnapShot Semantics

Updates on relations are carried out in two steps, first one marks the tuples to be updated (also deleted) and in a second step, one actually does the update. This is to prevent inconsistencies.

3.7 Views

A view is a *logical relation* in a DBMS. It allows for easier access of data that is complicatedly stored and could only be accessed with both deep knowledge about the underlying way the data is stored and highly sophisticated queries.

```
CREATE VIEW <name of view> AS <sql query>;
```

Now one can use `<name of view>` just like a table. This allows for privacy and better usability resulting in simpler queries. Views get evaluated by automatic *query rewriting*. Example:

```
CREATE VIEW StudProf(Sname, Semester, Title, Pname) AS
    SELECT  s.Name, s.Semester, l.Title, p.Name
    FROM    Student s, attends a, Lecture l, Professor p
    WHERE   s.Legi = a.Legi AND a.Nr = l.Nr AND l.PersNr = p.PersNr;

-- now one can use this View in queries:
SELECT DISTINCT Semester
FROM    StudProf
WHERE   Pname = 'Alonso';
```

One can also use Views for "is-a" relationships by providing base tables and creating Views for frequently used concepts.

Updates

Generally, it is not possible to update a view, since the table it represents is not physically stored, and would lead to anomalies in the physical table, e.g. when a view combines only some columns of multiple different relations. What would a change to a view incur? Would all the relations that have some attributes in the view need to change? $\Longrightarrow$ inconsistencies. Same happens with aggregations. This leads to the following theorem about views in SQL:

Theorem 3.1. A SQL View is updateable $\Longleftrightarrow$ the view involves only one base relation, the view involves the key of that base relation and the view does **not** involve aggregates, group-by or duplicate elimination.

References