Databases 2

Elements of Data Science and Artificial Intelligence

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

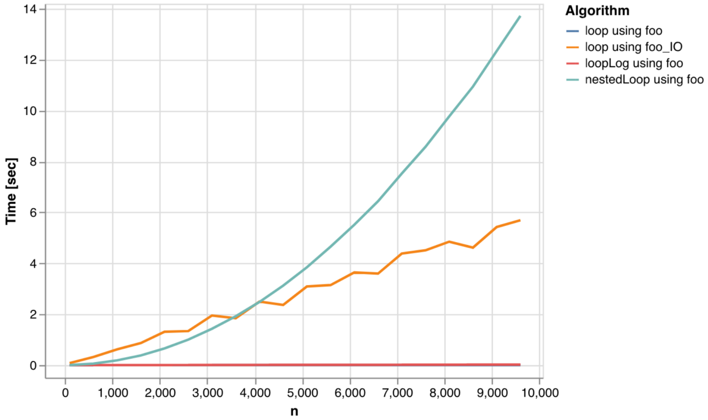
January 20, 2020

The Key Questions with Google Maps (1/2)

Key questions:	**for this concrete application (Google Maps):**
1. How to store, access, and query data?	where and **how** to store and cache the data?

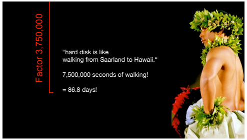
The how-part of the question has a lot to do with how to *transfer* data in-between different layers of the storage hierarchy.

CPU vs I/O, a Performance Primer



the graph shows the runtime of different algorithms,
see the Jupyter Notebook "CPU vs IO, a Performance Primer.ipynb" for details

The Two Benefits of Data Compression



Recall our walking to Hawaii-example:

Access time ∼ how long does it take us to walk to Hawaii and back, i.e. how long does it take for the first Byte to reach Saarbrücken

Bandwidth ∼ how much can we carry on our trip, i.e. how many bytes of data can be transferred in a given time interval

How can we make better use of the available bandwidth?

Compress to Save Bandwidth

We compress data in order to save bandwidth while transferring data in-between storage layers.

Do not confuse this with:

Compress to Save Storage Space

We compress data in order to save space on a particular storage layer.

Examples

- uncompressed (bmp, tga, raw) vs compressed image (RLE, png, jpg, etc.)
- uncompressed (wav) vs compressed music (mp3)
- uncompressed (raw) vs compressed video (mpeg-4)
- uncompressed (data in a database) vs compressed data (compressed database or query results)

run-length-encoding (RLE): blackboard

The images provided by Open Street Map (the open variant of Google Maps) use png.

Transfer Time without Compression

Transfer Time without compression

The overall transfer time to send n Bytes of data from storage layer x to storage layer y without compression is:

$$T_{truc}(n) = T_{tr}(n) = n/BW$$

Here BW is the bandwidth between storage layers x and y in [GB/sec]

Note

In this and the following definitions we assume that the time to read data on storage layer x and then write that data on storage layer y is not the bottleneck.
In other words: the following arguments make sense if **the transfer is the bottleneck**.

Example:

The transfer time for 1 TB of data without compression and $BW = 10GB/sec$ is $T_{tr} = 1000/10\ [\frac{GB}{GB/sec}] = 100\ [\frac{1}{1/sec}] = 100\ sec$.

Transfer Time with Compression

Transfer Time with compression

The overall transfer time to send n Bytes of data from storage layer x to storage layer y with compression is:

$$T_{trc}(n) = T_{compress}(n) + T_{tr}(n_c) + T_{decompress}(n_c)$$

here $T_{compress}(n)$ is the time required to compress the n Bytes into n_c Bytes,
here $T_{decompress}(n_c)$ is the time required to decompress the n_c Bytes back into n Bytes,

Compression Benefit Sweet-spot

Compressing data to save bandwidth only makes sense if $T_{trc}(n) < T_{truc}(n)$. Whether this equation holds depends on:

1. the data compression ratio n/n_c, **and**
2. the runtime of the compression algorithm, i.e. $T_{compress}(n)$, **and**
3. the runtime of the decompression algorithm, i.e. $T_{decompress}(n)$.

Examples

The transfer time for 1 TB of data and $BW = 10 GB/sec$

Without Compression:

$T_{truc} = 1000/10\ [\frac{GB}{GB/sec}] = 100\ [\frac{1}{1/sec}] = 100\ sec.$ (as above)

With Expensive Compression:

$T_{compress}(n) = 1\ GB/sec$, $T_{decompress}(n) = 5\ GB/sec$, compression ratio $n/n_c = 5$

$$\begin{aligned} T_{trc}(n) &= T_{compress}(1000\ GB) + T_{tr}(200\ GB) + T_{decompress}(200\ GB) \\ &= 1000\ sec + 20\ sec + 40\ sec = \mathbf{1060\ sec} > T_{truc} = 100 sec \end{aligned}$$

With Inexpensive Compression:

$T_{compress}(n) = 50\ GB/sec$, $T_{decompress}(n) = 100\ GB/sec$, compression ratio $n/n_c = 3$

$$\begin{aligned} T_{trc}(n) &= T_{compress}(1000\ GB) + T_{tr}(333\ GB) + T_{decompress}(333\ GB) \\ &= 20\ sec + 33.3\ sec + 3.3\ sec = \mathbf{56.6\ sec} < T_{truc} = 100 sec \end{aligned}$$

Survey

How can we improve T_{trc} further, even for a subset of the non-beneficial scenarios where $T_{trc} > T_{truc}$?

(A): Compress the data to be transferred *before* the request to transfer that data is received.

(B): When receiving the compressed data, do not decompress it. Then do whatever you want to do with the data on the compressed data.

(C): Let compression and transfer overlap.

(D): Let decompression and transfer overlap.

Solution (A–D)

all correct!

Notice our hidden assumption for T_{trc} that the three steps (compress, transfer, and decompress) are executed *one after another* (in computer science-lingo: *in serial*). This is rarely required in practice!

see exercise

The Key Questions with Google Maps (1/2)

Key questions:

2. How to make query processing efficient and scalable?

3. How to make this happen for just any kind of data

for this concrete application (Google Maps):

which queries?:

(a) 2-dimensional range queries,

(b) text search on geonames.

How does a database process such a query?

what data?:

(a) satellite images (raster data),

(b) roads, borders, etc. (vector data),

(c) geographic names (text)

Vektordaten: Polygone &Texte

Rasterdaten: Luftaufnahmen von Satelliten und Flugzeugen

Domains

Domain (German: Domäne, Wertebereich)

A domain D describes all possible values of a variable.

Example:

- integer, float, String, etc.
- all kind of enumerations: $\{$female, male, diverse$\}$
- any restriction/composition of a domain: all integers smaller 42
- any kind of structured type[1] like JSON, a graph, any byte sequence (BLOB: binary large object)

[1] If you see textbooks claiming that domains have to be atomic, just ignore it: it is a historical artefact and outdated.

The Relational Model[2]: Tuples, Attributes

Relation

A relation is a subset of the crossproduct of n domains. In other words, a relation R is defined as $R \subseteq D_1 \times \ldots \times D_n$.

Tuple

Every element $t = (a_1, \ldots, a_n) \in R, a_{1 \leq i \leq n} \in D_i$ is called a *tuple*. The a_i are called **attributes**.

Order of Tuples

The order of the tuples in a relation does not matter.

[2] We follow the notation used in the textbook "Datenbanksysteme" by Kemper&Eickler.

Relational Schema

Relational schema

A relational schema specifies both the domains **and** the attribute names of a relation. This means, in contrast to the domain-based definition of a relation (shown above) we additionally specify attribute names. A relation schema is denoted as a **sequence** $[R] : \{[A_1 : D_1, \ldots, A_n : D_n]\}$.
The attribute names must be duplicate-free[a].
Any instance of a relational schema is (also) called a relation.

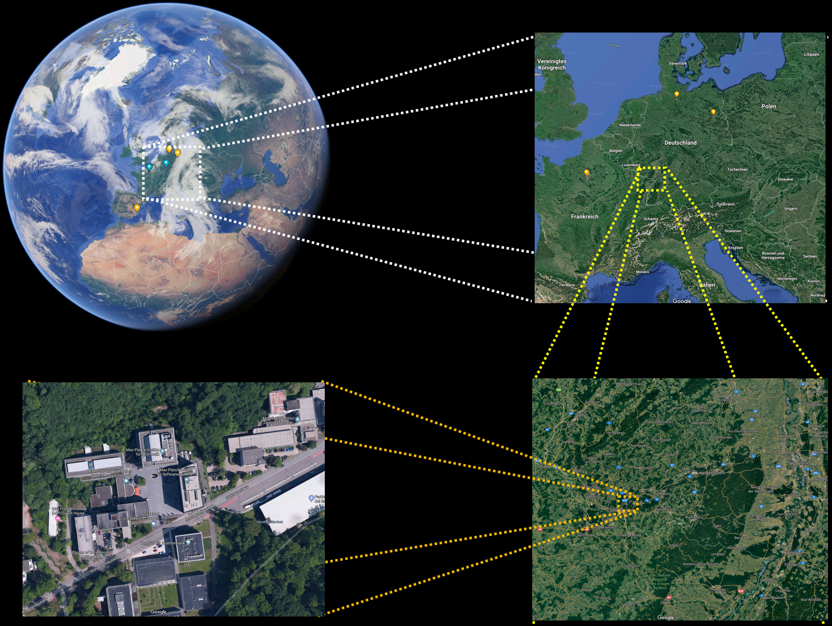
[a] This rule is no strict requirement and could be dropped, but it makes life much easier.

Examples:

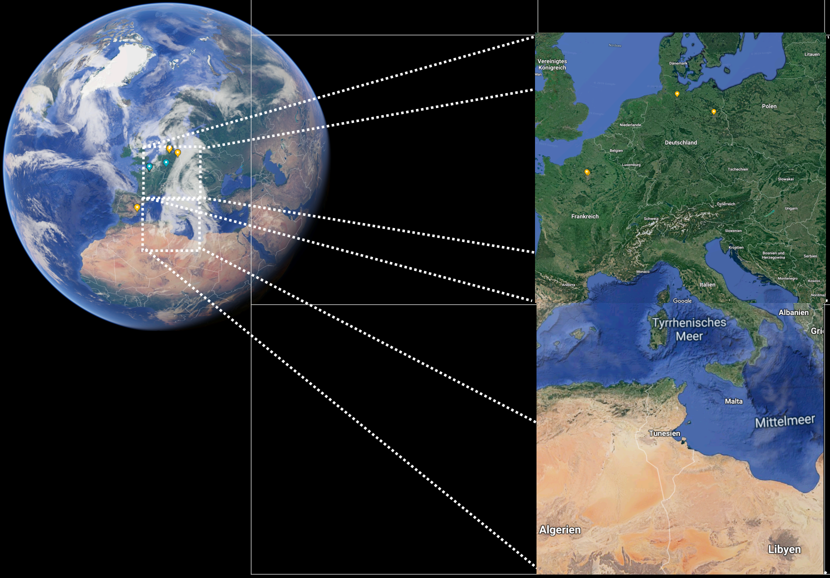
[cities] : {[id:int, name:string, latitude:float , longitude:float, inhabitants: int]}

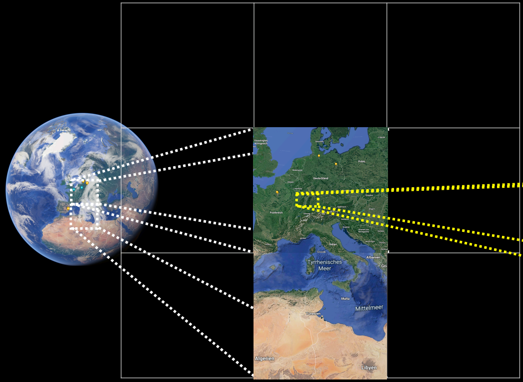
Order of Attributes

The order of the attributes in a relational schema definition does not matter.



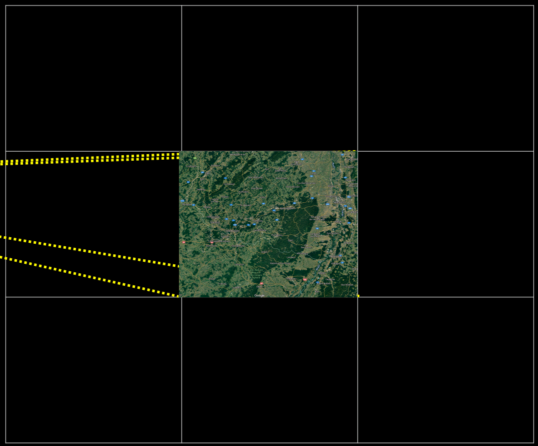
Vereinigtes Königreich
Danemark
Litauen
Polen
Niederlande
Belgien
Deutschland
Luxemburg
Tschechien
Slowakei
Frankreich
Österreich
Ungarn
Schweiz
Slowenien
Kroatien
Bosnien und Herzegowina
Serbien
Monaco
Andorra
Italien
Montenegro
Kosovo
Google
Kaiserslautern
Mannheim
Homburg
Saarbrücken
Max-Planck-Institut für Informatik
Max-Planck-Institut für Softwaresysteme
Saarland Informatics Campus
Max Planck Institute (MPI) Guest House
Zentrum für Bioinformatik
Cluster of Excellence Multimodal
Stuhlsatzenhausweg
Google


Vereinigtes
Königreich
Litauen
Polen
Niederlande
Deutschland
Belgien
Luxemburg
Tschechien
Slowakei
Frankreich
Österreich
Ungarn
Schweiz
Slowenien
Kroatien
Bosnien und
Herzegowina
Serbien
Monaco
Andorra
Italien
Montenegro
Kosovo
Google
Albanien
Tyrrhenisches
Meer
Malta
Mittelmeer
Tunesien
Algerien
Libyen



Raster mit Auflösung 1 Pixel per 10 km Breite

1 Pixel ≙ 100km²



Raster mit Auflösung 1 Pixel per 1 km Breite

1 Pixel ≙ 1km²

Anzahl der Pixel wächst quadratisch in der Auflösung!

510.100.000 km² ≙ 5 M Pixel

510.100.000 km² ≙ 510 M Pixel

Relational Schema for this scenario

[tiles] : {[id:int, zoomlevel:int, xpos:int, ypos: int, filepath:string]}

Explanation:

- zoomlevel: from 0 to *maxZoomlevel*, 0 being the lowest, *maxZoomlevel* the highest resolution
- xpos: the offset of a tile in x-direction
- ypos: the offset of a tile in y-direction
- filepath: the filepath to the tile image on disk (alternatively a BLOB, binary large object)

Constraints:

- xpos $\in [0, \ldots, 2^{zoomlevel} - 1]$
- ypos $\in [0, \ldots, 2^{zoomlevel} - 1]$

Number of tiles/tuples is:

$$4^0 + 4^1 + 4^2 + \ldots = \sum_{zoomlevel=0}^{maxZoomlevel} 4^{zoomlevel} = \frac{1}{3} \cdot (4^{maxZoomLevel+1} - 1)$$

Query and Point Query

Query

Any expression $\sigma_P(R)$ where P is a predicate defined on relational schema $[R]$, i.e., a function $P : R \mapsto \{true, false\}$, is called a *query* on R.
The result of a query $\sigma_P(R) \subseteq R$ contains all tuples of R for which the predicate P holds.

Example: $P :=$ zoomLevel $= 2$: This is a predicate on the relational schema $[tiles]$.

Equality Predicate/Point Query

Given a relational schema $[R]$ with an attribute A_i, a corresponding one-dimensional domain D_i, and a constant $c \in D_i$. Then, $\sigma_{A_i=c}(R)$ is called an *equality predicate* or *point query* on R.
This query selects all tuples $t = (a_1, \ldots, a_n) \in R$ where the one-dimensional point a_i equals c.

Example: $\sigma_{\text{zoomLevel}=2}(tiles)$: This is a query selecting all tiles on zoom-level 2.

Computing the Results to a Query

In order to compute the results to a query of form $\sigma_P(R)$ we basically only have two options:

Brute-Force (aka Scan)

We inspect each and every tuple in R and check whether P holds, and if that is the case add the tuple to the result set.

Index (aka Index Scan)

We organize the contents of R such that we do not have to inspect each and every tuple to determine whether a tuple belongs to the result of a query.

In order to understand 'Indexing' we first have to introduce a couple of concepts[3]...

[3] The following introduction diverts from well-known textbooks and other explanations as those explanations frequently mix up the core idea of an index with its concrete physical realisation.

Horizontal Partitioning

Horizontal Partitioning

Given a relation R any assignment of the tuples of R into relations $R_1, \ldots, R_k$ is called a *horizontal partitioning of* R if $\forall_{t \in R} \exists_{R_i, 1 \leq i \leq k}$ with $t \in R_i$. The R_is are called the *horizontal partitions* of R.

Examples: $R = \{(2, A), (7, B), (1, B), (6, C)\}$

- $R_1 = \{(2, A), (1, B), R_2 = \{(7, B)), (6, C)\}$ is a horizontal partitioning.
- $R_1 = (1, B), R_2 = \{(7, B), (2, A), (6, C)\}$ is a horizontal partitioning.
- $R_1 = \{(2, A), (1, B), R_2 = \{(2, A), (6, C)\}$ is **not** a horizontal partitioning.

Disjoint Horizontal Partitioning

A horizontal partitioning is called *disjoint* if $R_i \cap R_j = \emptyset \ \forall_{i, j \neq i}$.

$R_1 = \{(2, A), (1, B), R_2 = \{(7, B), (2, A), (6, C)\}$ is a horizontal partitioning but **not** disjoint.

Partitioning Function

Partitioning Function

Given a domain D, any function $p : [R] \rightarrow D$ is called a *partitioning function*.

Examples:

$[R] = \{[a : int, b : char]\}, R = \{(2, A), (7, B), (1, B), (6, C)\}$

$p_0 : [R] \rightarrow int, p_0(t) := t.a$ modulo 2

- $p_0((2, A)) = 0$
- $p_0((7, B)) = 1$
- $p_0((1, B)) = 1$
- $p_0((6, C)) = 0$

$p_1 : [R] \rightarrow char, p_1(t) := t.b$

- $p_1((2, A)) = A$
- $p_1((7, B)) = B$
- $p_1((1, B)) = B$
- $p_1((6, C)) = C$