Databases 4

Elements of Data Science and Artificial Intelligence

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

January 27, 2020

Performance Dimensions of an Index

The performance of a logical and/or physical index can be analysed along many dimensions:

1. runtime (wall-clock time)
2. clock cycles spent
3. I/O-operations performed (for some or multiple storage layers; counting cache misses is a special case of this)
4. robustness to different workloads (workload := datasets and queries)
5. adaptability to different workloads
6. read vs write-performance
7. scalability, i.e. increasing the workload
8. single vs multi-threaded performance
9. memory consumption
10. etc.

How to Analyse the Performance of an Index?

In general, in computer science there are only three methods to analyse the performance of any algorithm or system:

1. Analytical model (a mathematical model: e.g. complexity or cost model)[1]
2. Simulation (a runnable model focusing on the major effects of a problem)
3. Experiment (a real software running real queries and datasets)

Avoid organisational blindness (German: Betriebsblindheit)

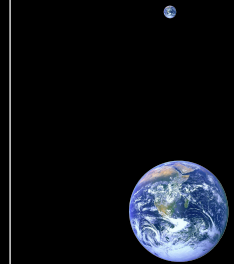
None of these methods is per se better than any other. All of these methods have their pros and cons. For the same problem: Try always to use at least two of those methods.

Excellent book:
Raj Jain. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley Professional Computing. 1991.[2]

[1] This is what we did last lecture in the Jupyter notebook.

[2] If I had to recommend the ten must-read books in CS, this book would be on that list.

method	effort/cost	reality	generalizability
Analytical Modelling			
Simulation			
Experiment			



One-dimensional Range Query

One-dimensional Range Query

Given a relational schema $[R]$ with an attribute A_i, a corresponding non-categorical one-dimensional domain D_i, and two constants $l, h \in D_i$. Then, $\sigma_{l \leq A_i \leq h}(R)$ is called a *range query* on R. This query selects all tuples $t = (a_1, \ldots, a_n) \in R$ where the one-dimensional point a_i is contained in interval $[l; h]$.

Examples:

- $\sigma_{2 \leq \text{zoomLevel} \leq 4}(\mathit{Tiles})$: selects all tiles where the zoom-level is contained in interval $[2; 4]$.
- $\sigma_{0 \leq \text{xpos} \leq 3}(\mathit{Tiles})$: selects all tiles where xpos is contained in interval $[0; 3]$

Note

For $l = h$ a one-dimensional range query is equivalent to a point query.

Closed vs open intervals

Intervals may be defined as open (l and h not included), half-open (l or h included) or closed (l and h included).

Indexing support for One-dimensional Range-Queries

In order to support one-dimensional range queries we basically only have three options:

Option 1: Translate to a Point-Query

We convert the range condition $l \leq A_i \leq h$ into multiple point queries:

$$\sigma_{[l;h]}(R) = \bigcup_{c \in \{[l;h] \cap D_i\}} \sigma_{A_i=c}$$

Examples:

- $\sigma_{2 \leq \text{zoomLevel} \leq 4}(Tiles) = \{\sigma_{\text{zoomLevel}=2}(Tiles) \cup \sigma_{\text{zoomLevel}=3}(Tiles) \cup \sigma_{\text{zoomLevel}=4}(Tiles)\}$
- $\sigma_{0 \leq \text{xpos} \leq 3}(Tiles) = \{\sigma_{\text{xpos}=0}(Tiles) \cup \sigma_{\text{xpos}=1}(Tiles) \cup \sigma_{\text{xpos}=2}(Tiles) \cup \sigma_{\text{xpos}=3}(Tiles)\}$

Problem

This is only efficient (or even feasible) if $\left|[l; h] \cap D_i\right|$ is small! In other words: the number of distinct values and hence distinct point queries to generate for this range is low.

Note: $\left|R\right|$ is the cardinality operator. It returns the number of elements in R.

Examples

How do we translate?

- $[0.34357783; 0.4578784323]$
- $[adsv; bsdgsd]$
- $[1; 1000]$
- $[1; 10]$

Indexing support for One-dimensional Range-Queries

Option 2: Range-aware Index Search Algorithm

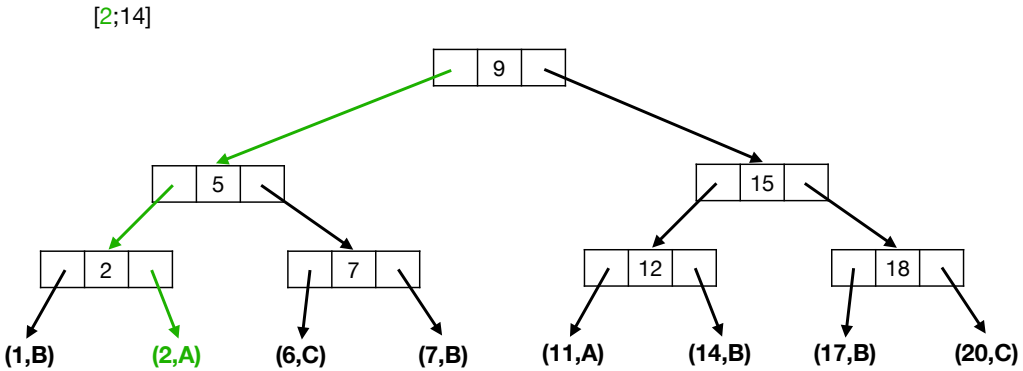
Change the index search algorithm to support range queries.

Option 3: Range-aware Index Structure

Change the index structure to support range queries.

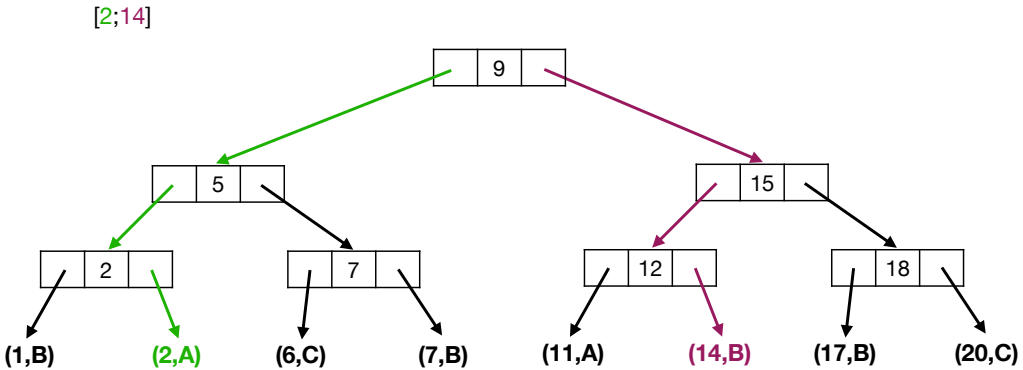
Let's look at Option 2 first:

Example: range query $[2; 14]$



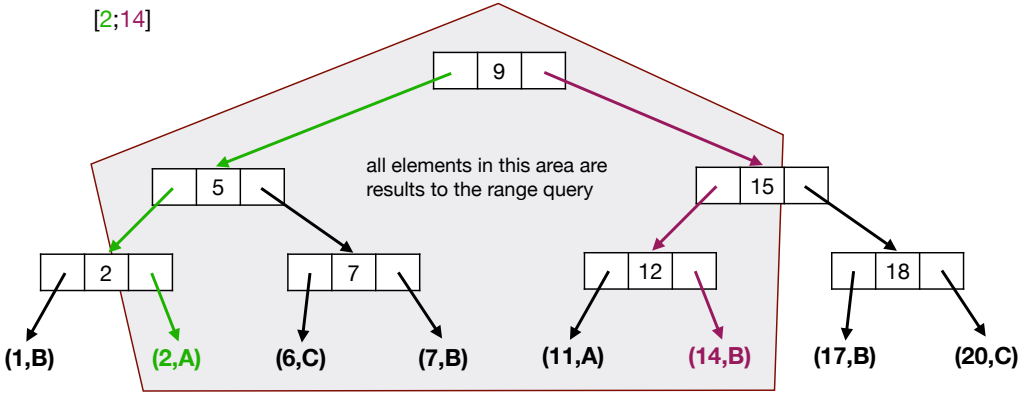
First: run a point query with the left boundary of the range: $c = 2$.

Example: range query $[2; 14]$



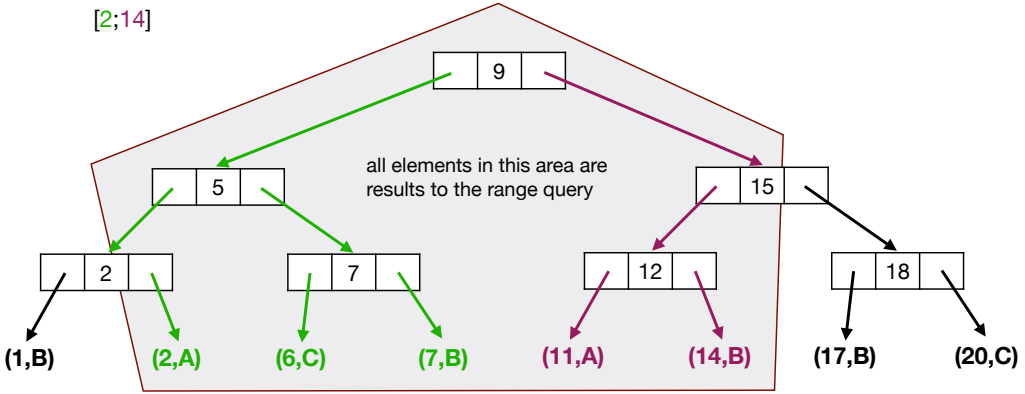
Second: run a point query with the right boundary of the range: $c = 14$.

Identifying the Result Area



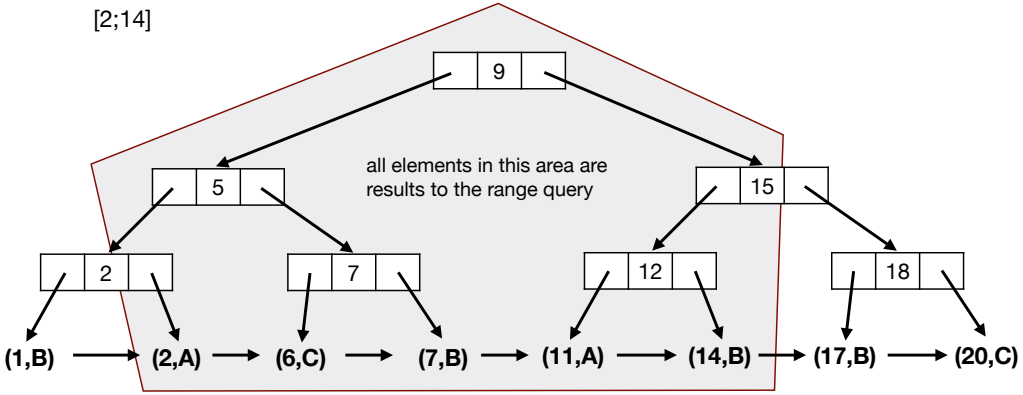
Using the two point queries we have identified the part of the search tree that contains the results to the range query (marked in gray).

Traversing Everything in the Result Area



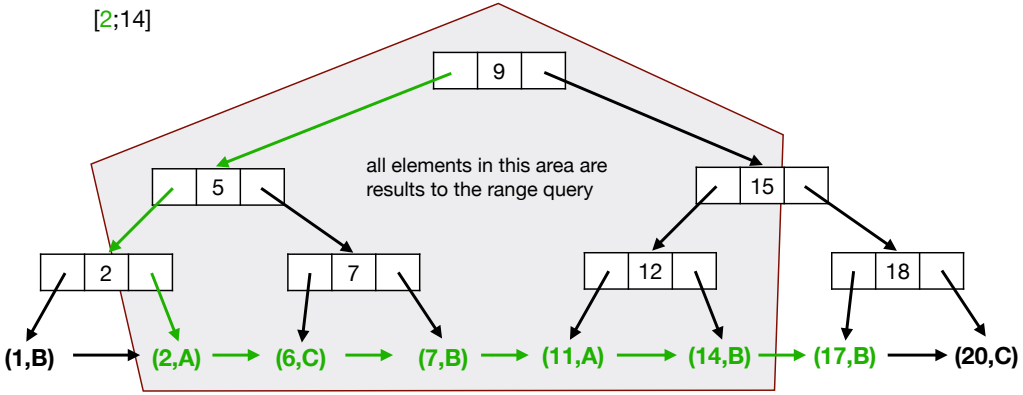
We may now traverse all arrows in the result area to retrieve all results, i.e. all entries in all leaves in the result area.

ISAM: Additional Chain on the Leaf-Level



But actually, it would be better (yet it depends...) if the tree structure provided a means of traversing all leaves in order. This is Option 3.
This particular method is coined ISAM (Index Sequential Access Method).

ISAM-Query: One Point Query plus Chain-Traversal



Like that we can also answer the range query by running a single point query with $c = 2$. Then we traverse the list of leaves in ascending order an d check each entry whether it is smaller equal 14. As soon as that condition is false, we terminate.

Which Option should we use: 1, 2 or 3?

Option 1: Translate to a Point-Query

+ easy to implement
+ works well if cardinality of the range is very low
– bad or even no option if cardinality of the range is high

Option 2: Range-aware Index Search Algorithm

+ a good compromise in practice
+ no structural modification to the tree required, i.e. no extra ISAM
– typically not the most efficient option on the storage hierarchy

Option 3: Range-aware Index Structure

+ typically the most efficient option on the storage hierarchy (if you want to support range queries!)
– extra chain (ISAM) has to be maintained under inserts/deletes

Range Queries in Google Maps?

Recall: in the jupyter notebook we created (and then searched) an index that mapped from the ID of a geographical name, e.g. a city, to its geographical location (longitude, latitude). In other words, this was a key/value-mapping:

Index 1:

ID $\rightarrow$ (longitude, latitude)

Only if we need to query multiple IDs in a range in a single query, e.g. ID-ranges like $[7, 8, 9, \ldots, 42]$, we need to worry about range queries.

Another very useful index in Google Maps is the following:

Index 2:

asciiname $\rightarrow$ (longitude, latitude)

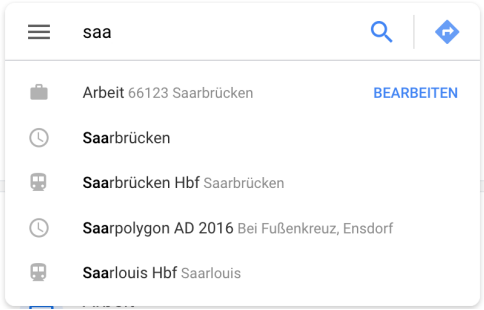
Example:

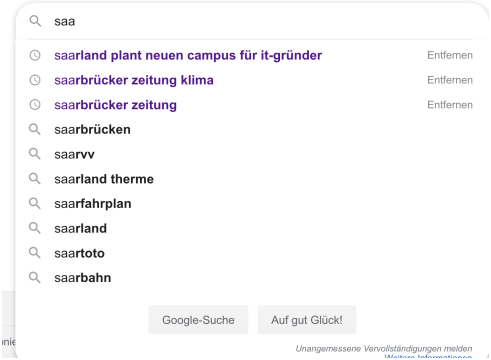
a query asking for "Saarbrücken": $\sigma_{asciiname=\text{Saarbrücken}}(geonames)$

Range Queries in Google Maps?

But what if we do not specify the entire asciiname but only a prefix, e.g.:

- All cities starting with ‘Saa’
- All cities starting with ‘Saarb’





Prefix Query

These queries are called *prefix queries*.

Prefix Query

Given an index on an attribute A on an arbitrary domain D. Let c be a constant broken into a sequence of *digits* $c = d_0, \ldots, d_k$ where each d_j with $0 \leq j \leq k$ is a digit. A digit may be a character, a numbers, an individual or multiple bits. Let $c_i = d_0, \ldots, d_i$ be the digits from d_0 up to and including $d_{i \leq k}$.
Then c_i is called a *prefix* and any point query with equality predicate $P = A == c_i$ is called a *prefix query*.

Example:
d_0 ='S', d_1 ='a', d_2 ='a', c_2 ='Saa', $P = A$ =='Saa' is a prefix query

Prefix to Range Query Translation

A prefix query can often be translated to a range query.

Prefix to Range Query Translation

Given a prefix query $P = A == c_i$ with a non-categorical domain D, i.e. the elements of D can be ordered. Then the result of the prefix query is equivalent to the range query $P = c_i \leq A < (c_i + 1)$.

Examples:

(1.) c_2 ='Saa', $P = A$ =='Saa' is a prefix query

we can rewrite this to a range query P ='Saa' $\leq A <$ 'Sab'

(2.) $D = int$ c_2 ='134', $P = A$ =='134' is a prefix query

we can rewrite this to a range query $P = 134 \leq A < 135$.

Hence, whether we ask for all integers starting with the prefix '134', e.g. 1347843, 13462, 134, etc. **or** whether we ask for all integers in range $[134; 135]$ makes no difference, right?

Upps...

What is the Problem here?

Answer:

In the definition on slide 19 we assumed a digit-wise comparison! This works for example (1.)

However, in example (2.) we performed an integer comparison comparing *different* digits!
We compared $c_2^1 = 134$ with $c_6^2 = 1347843$ by comparing the digits as c_0^1 with c_0^2, then c_0^1 with c_0^2, rathern than defining globally $c_6^1 = 1340000$ and then performing the query translation.

see notebook: Digit-wise vs Integer-comparison

Be careful

Prefix to range query-translation only works, if we assume a digit-wise comparison!

```
# the problem:
print('string data:')
data = ['Saa', 'Sa', 'S', 'Sab', 'Saaa', 'Sb', 'Saab', 'Sab
data_sorted = sorted(data)
for suffix in data_sorted:
    if suffix >= 'Saa' and suffix < 'Sab':
        print(suffix)

print('integers represented as strings:')
data2 = ['1347843', '13462', '134']
data2_sorted = sorted(data2)
for suffix in data2_sorted:
    if suffix >= '134' and suffix < '135':
        print(suffix)

print('integers represented as integers:')
data3 = [1347843, 13462, 134]
data3_sorted = sorted(data3)
for suffix in data3_sorted:
    if suffix >= 134 and suffix < 135:
        print(suffix)
```

```
string data:
Saa
Saaa
Saab
integers represented as strings:
134
13462
1347843
integers represented as integers:
134
```

Multi-dimensional Range Query

Multi-dimensional Range Query

Given a relational schema $[R]$ with $k \geq 1$ attributes $A_{i_1}, \ldots, A_{i_k}$, corresponding non-categorical one-dimensional domains $D_{i_1}, \ldots, D_{i_k}$, and constants $l_{i_1}, h_{i_1} \in D_{i_1}, \ldots, l_{i_k}, h_{i_k} \in D_{i_k}$.
Then, $\sigma_{l_{i_1} \leq A_{i_1} \leq h_{i_1} \wedge \ldots \wedge l_{i_k} \leq A_{i_k} \leq h_{i_k}}(R)$ is called a *multi-dimensional range condition* or *volume query* on R.
In other words, we select all tuples $t = (a_1, \ldots, a_n) \in R$ where the k-dimensional point $(a_{i_1}, \ldots, a_{i_k})$ is contained in volume $[l_{i_1}; h_{i_1}] \times \ldots \times [l_{i_k}; h_{i_k}]$.

Examples:

- $\sigma_{0 \leq \text{xpos} \leq 3 \wedge 5 \leq \text{ypos} \leq 6}(\mathit{tiles})$: selects all tiles with xpos 0, 1, 2 or 3 where additionally ypos is either 5 or 6.
- $\sigma_{49 \leq \text{latitude} \leq 50 \wedge 9 \leq \text{longitude} \leq 10}(\mathit{cities})$: selects all cities in lat/lon-volume $[49; 50] \times [9; 10]$

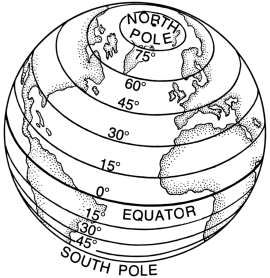
Note

For $k = 1$ this query is equivalent to a one-dimensional range query.

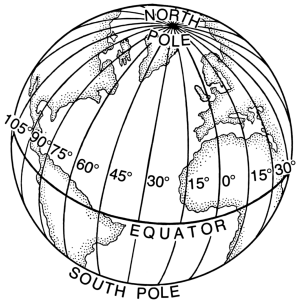
Indexing support for Multi-dimensional Range-Queries

Index 3:
we need to support multi-dimensional range queries along latitude (Breite) and longitude (Länge) and zoomLevel

(latitude, longitude, zoomLevel) $\rightarrow$ tile



Latitude (Breite)



Longitude (Länge)

Relational Schema for this Scenario (Repeated and extended)

[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:

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

Open Street Map Tile Addressing Scheme

https://a.tile.openstreetmap.de/<zoomLevel>/<x>/<y>.png

Examples:

https://a.tile.openstreetmap.de/0/0/0.png
retrieves the world map:

https://a.tile.openstreetmap.de/1/1/0.png
retrieves upper right tile at zoom level 1:

https://a.tile.openstreetmap.de/1/0/0.png
retrieves upper left tile at zoom level 1:

https://a.tile.openstreetmap.de/7/70/44.png
a tile at zoom level 7:

Open Street Map Tile Addressing Scheme

OK, we can now translate longitude/latitude to xpos/ypos.

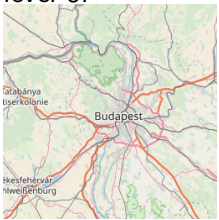
But:

1. this translation has to be done for each zoom-level independently
2. (x,y)-coodinates across zoom-levels are not necessarily spatially related

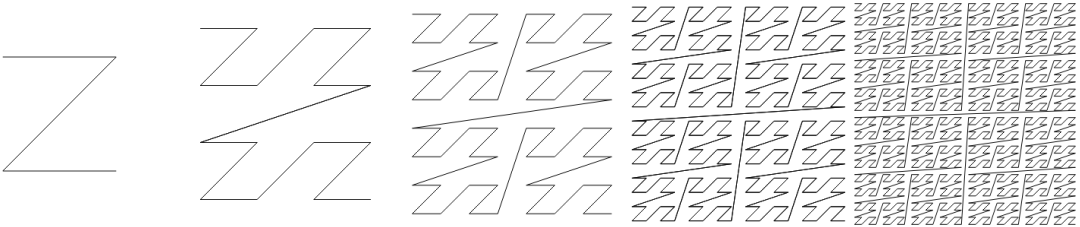
Examples:

https://a.tile.openstreetmap.de/7/70/44.png yields the tile including Budapest at zoom level 7:

https://a.tile.openstreetmap.de/8/141/89.png yields the tile including Budapest at zoom level 8:



z-codes



see Notebook: z-codes