Query Optimisation (Part 1)

Big Data Engineering
(formerly Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

May 25, 2023

CC BY-NC-SA

The Past Weeks: Simple Data Analysis

1. Concrete application: IMDb, NSA or similar applications

What have we learned?

Fundamental concepts of query processing:

- entity-relationship model
- relational model
- relational algebra
- SQL

Question

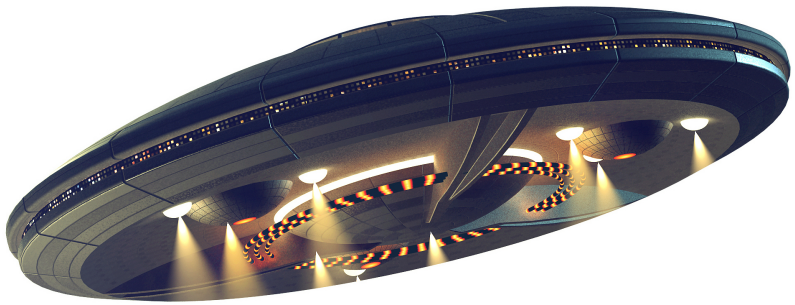
... and what if the data gets bigger? How do we actually get from SQL to an efficient program?

Overview of SQL Processing Systems (SQL Engines)

Sqlite

MySQL

PostgreSQL



Modern DBMS

 CC BY-SA 2.0 Johann-Nikolaus Andreae, (c) istock.com Laspi/phive2015/3000ad

What Do I Do if the Queries Are Too Slow?

Now:

Question 1

How do we actually get from SQL to an executable program in principle?

Automatic Query Optimisation

1. SQL
   ↓ canonical translation
2. annotated relational algebra/logical plan
   ↓ heuristic optimisation
3. transformed logical plan
   ↓ cost-based optimisation
4. physical plan
   ↓ code generation
5. executable code

All these optimisations run internally: the user of a SQL engine **does not** have to worry about them.

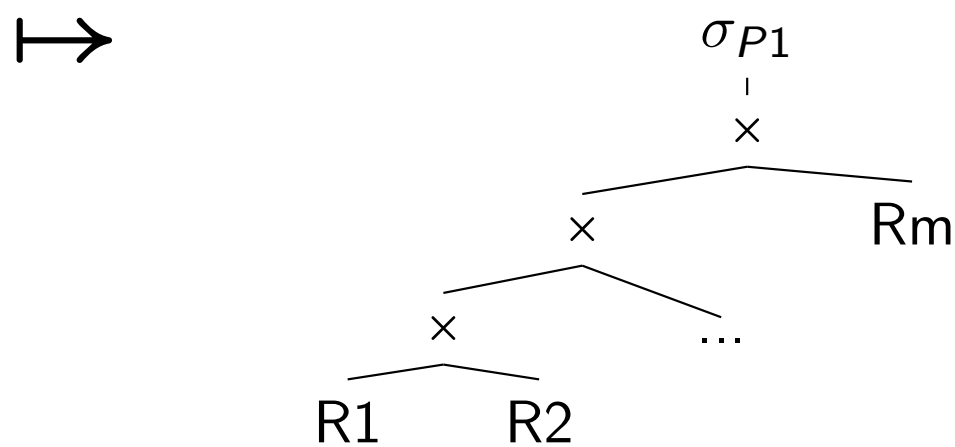
Depending on the query optimiser, some of these translation steps can be very simple, e.g. with very simple heuristics or even without cost-based optimisation.

Canonical Translation

1. SQL
   ↓ canonical translation
2. annotated relational algebra/logical plan

In canonical translation, SQL is translated into relational algebra. The basic pattern is the following:

```
SELECT     A2,..,An, bar()
FROM       R1,...,Rm
WHERE      P1
GROUP BY   A1,...,An
HAVING     P2
```



Canonical Translation

Further steps in canonical translation:

- Handling of views (CREATE VIEW; see SQL notebook, Section "Views"), i.e. view definitions are inserted 1:1 into the SQL query.
- If necessary, replace partial expressions with pre-calculated, cached or pre-materialised (partial) results (and materialised views).

Example:
Exactly the same query has already been calculated AND the database has not changed: return the cached result without re-executing the query.

WHAT vs HOW confusion

Attention: WHAT vs HOW confusion

We can read relational algebra both declaratively and procedurally!

1. **declarative:** WHAT is the result

   The order of the operators does not matter for certain situations, e.g., the order of two selections is irrelevant for correctness. We also do not care about how those operators are implemented.

   That is how we have read it so far.

2. **procedural:** HOW do we calculate the result

   We can specify at least two different things: The operators of relational algebra are executed in the **order** they appear in the expression. Also, we can decide **how** each operator is **implemented**.

   This procedural way of reading will apply in the following.

Logical vs Physical Operators

When introducing the relational model and operators, we said:
“

Attention

Relational algebra only abstractly describes **WHAT** has to be calculated, but not **HOW** this calculation is implemented algorithmically!

”

In the implementation in the notebook “Relational Algebra” we already distinguished between logical and physical operators.
In that notebook, both aspects (order and implementation of the operators) are already taken into account (otherwise it would be impossible to implement these operators...).

Logical vs Physical Operators

Logical	**Physical**
$\bowtie$	nested-loops (cross product) plus post-filtering
	index-based
	... (see core lecture)
$\cap$	nested-loops (cross product) plus post-filtering
	index-based[1]
	...
σ	iteration/scan plus post-filtering
|	index-based?
R	...

[1] cf. implementation of intersection in the notebook "Relational Algebra" through Python set

Plan and Quality of a Plan

Plan

Any valid relational algebra expression consisting of logical and/or physical operators is called a **plan**. An expression consisting only of logical operators is called a **logical plan**. An expression consisting only of physical operators is called a **physical plan**.

Quality of a Plan

To determine the quality of a plan, we need to develop a function that can compute or estimate that quality.

"Quality" may be defined in various different ways: low runtime, low power consumption, little data is loaded from the network, etc.

Before being able to define "quality" meaningfully, we first need to introduce some basic terms.

Basic Terms: Selectivity

Selectivity

For a unary operator op(R)$\rightarrow$ R', its selectivity is the ratio of the size of its output relation R' to its input relation R:

$$sel_{op} = \frac{|\text{R'}|}{|\text{R}|} \leq 1.$$

Example:
$|R| = 1.000.000$, $|\sigma_p(R)| = 1.000$, $|\sigma_q(R)| = 500.000$

High selectivity ("many tuples are filtered out"):
Selectivity of $\sigma_p(R)$: $\frac{|\sigma_p(R)|}{|R|} = \frac{1.000}{1.000.000} = 0.001 = 0.1\%$

Low selectivity ("few tuples are filtered out"):
Selectivity of $\sigma_q(R)$: $\frac{|\sigma_q(R)|}{|R|} = \frac{500.000}{1.000.000} = 0.5 = 50\%$

Selectivity: Example With Two Selections

Example:
$|R| = 1.000.000$, $|\sigma_p(R)| = 1.000$, $|\sigma_q(R)| = 500.000$

Low selectivity $\sigma_p\big(\sigma_q(R)\big)$:
Here σ_q first filters 1,000,000 input tuples to 500,000 output tuples.
Then σ_p filters 500,000 input tuples.

In total 1,500,000 tuples are inspected ($\rightarrow$**quality measure**)

High selectivity $\sigma_q\big(\sigma_p(R)\big)$:
Here σ_p first filters 1,000,000 input tuples to 1,000 output tuples.
Then σ_q filters 1,000 input tuples.

In total, 1,001,000 tuples are inspected ($\rightarrow$**quality measure**)

Basic Terms: Cost Models

Cost model (aka cost function)

A cost model estimates the execution costs for a given (partial) plan using a suitable model. For any plan P we define a function for this: $\text{costs}(P) \mapsto \text{float}$. And we interpret this as the quality of a plan.

Example: In the selectivity example above, we argued that the evaluation order $\sigma_q\big(\sigma_p(R)\big)$ is better because fewer tuples need to be inspected during execution:

Cost model: Intermediate results

$\text{costs}_{\text{Intermediate results}}(P) :=$ Sum over the cardinalities of the output relations of all operators in P.

This cost model estimates the total cost of a plan based on the size of the intermediate results. Costs for the individual operators (what is the runtime of the join?) or the access to certain relations (do I have to read this from the network or from SSD?) are **ignored** in this cost model!

Other Cost Models

Cost model: I/O costs

$\text{Costs}_{\text{I/O}}(P)$:= Sum over all access times/read times from hard disk/SSD/network over all input relations.

Cost model: Total runtime

$\text{Costs}_{\text{Total runtime}}(P)$:= Total runtime of the plan in seconds.

Cost model: Energy consumption

$\text{Costs}_{\text{Energy consumption}}(P)$:= Energy needed to execute the plan.

Cost model: Main memory

$\text{Costs}_{\text{Main memory}}(P)$:= Amount of main memory required to be able to execute the plan.

Heuristic Optimisation

2. annotated relational algebra/logical plan
   ↓ heuristic optimisation
3. transformed logical plan

Basic idea

- Heuristic optimisation iteratively transforms the canonical plan into a (hopefully) better logical plan using rules.
- Such a rule-based transformation never changes the outcome of the plan.
- Attempts to transform the plan are made until none of the heuristic rules are applicable.
- Only rules that **always reduce** (well, let's say in 99.99% of the cases) the costs under a given cost model are applied.

Heuristic Optimisation: Basic Algorithm

Basic idea:

RuleOpt (Plan P = canonical plan, set of rules R)

1. As long as any rule r from R on P matches:
2. $\quad P = r.\text{modify}(P)$

Extensions:

- Termination conditions of the iteration:
  - cap the maximum number of iterations
  - cap maximum effort for optimisation
  - prevent oscillation (actually mostly an artifact of rule-based optimisation)
- include priority of rules (e.g. first rule 42, then rule 4)

Basic Idea of a Rule (see rule.py)

Rule

match (OperatorTree op) $\mapsto$ bool

Search method that checks whether the input tree op matches a certain substructure, a certain pattern.

modify (OperatorTree op) $\mapsto$ OperatorTree

Transformation method that changes the input tree op and returns the modified input tree op'.

Examples:

match: is there a selection with a conjunction in the predicate?
modify: break up that predicate and replace the selection with two separate selections

match: is there a selection with join predicate directly above a a cross product?
modify: combine that selection and cross product into a join

The Mother of All Transformation Rules: Predicate Pushdown

Rule 1: Predicate Pushdown (Joins and Cross Products)

Let R_1 and R_2 be relations, p a predicate and $attr(p)$ the attributes on which p imposes conditions, then we can formulate the following optimisation rule:

$$\frac{\sigma_p(R_1 \oplus R_2) \qquad attr(p) \subseteq [R_2] \qquad \oplus \in \{\bowtie, \times\}}{R_1 \oplus \left(\sigma_p(R_2)\right)}$$

"Selections should be pushed down as close as possible to the data sources, i.e. down."

Examples of Predicate Pushdown

- satellite images: only send images to earth that meet the filter predicate
- sensor data: evaluate filter conditions directly when measuring the data
- mail (IMAP): filter on server vs filter mails on client
- smart disks: discard non-qualifying tuples directly while reading from the storage medium; this applies to all storage media: hard disks, SSDs, NAS, DRAM, etc.
- distributed systems: evaluate filter on remote system, then send only filtered results instead of all data

Projection Pushdown

Rule 2a: Projection Pushdown (Selection)

Let R be a relation, p a predicate and $attr(p)$ the attributes on which p imposes conditions, then we can formulate the following optimisation rules:

$$\frac{\pi_{[A]}\big(\sigma_p(R)\big) \qquad attr(p) \subseteq [A]}{\sigma_p\big(\pi_{[A]}(R)\big)}$$

$$\frac{\pi_{[A]}\big(\sigma_p(R)\big) \qquad attr(p) \nsubseteq [A] \subseteq [R] \qquad [A] \cup attr(p) = [B] \subseteq [R]}{\pi_{[A]}\Big(\sigma_p\big(\pi_{[B]}(R)\big)\Big)}$$

"Projections should be pushed as close as possible to the data sources, i.e. down."

Example for Projection Pushdown

Please do not confuse Projection Pushdown with Predicate Pushdown.

[R] : {[id: int, a:int, b:int]},
[S] : {[pid: int, r_id: int, c:int, d:int]}.

The original expression
$\pi_{a,c}\left(R \bowtie_{id=r\_id} S\right)$

is rewritten to:

$\pi_{a,c}\left(\pi_{id,a}(R) \bowtie_{id=r\_id} \pi_{r\_id,c}(S)\right)$

In this example, **additional** projections are introduced that project attributes away as early as possible. Care must be taken that attributes that are necessary for query processing are not projected away too early.

In this example: *id* and *r_id*.

Projection Pushdown: Joins

Rule 2b: Projection Pushdown (Join)

Let R_1 and R_2 be relations, p a predicate and $attr(p)$ the attributes on which p imposes conditions, then we can formulate the following optimisation rule:

$$\frac{\pi_{[A]}\big(R_1 \bowtie_p R_2\big) \qquad [A] \subset \big([R_1] \circ [R_2]\big) \qquad \big([A] \cup attr(p)\big) \cap [R_2] = [B] \subset [R_2]}{\pi_{[A]}\Big(R_1 \bowtie_p \big(\pi_{[B]}(R_2)\big)\Big)}$$

"Projections can be moved past a join as long as no attributes of the join predicate are lost."

Important Transformation Rules

Rule 3: Conjunctions can be broken up

Let p, q, s be predicates and R be a relation. It holds:

$$\frac{\sigma_p(R) \qquad p = q \wedge s}{\sigma_q\big(\sigma_s(R)\big)}$$

This then allows pushdown of individual selections (Rule 1).

Important Transformation Rules

Rule 4: Summary of Selection with Cross Products&Joins

Let R_1 and R_2 be relations, p and q predicates and $attr(p)$ and $attr(q)$ the attributes on which p and q impose conditions, then we can formulate the following optimisation rules:

$$\frac{\sigma_p(R_1 \times R_2) \qquad attr(p) \cap [R_1] \neq \emptyset \qquad attr(p) \cap [R_2] \neq \emptyset}{R_1 \bowtie_p R_2}$$

$$\frac{\sigma_q(R_1 \bowtie_p R_2) \qquad attr(q) \cap [R_1] \neq \emptyset \qquad attr(q) \cap [R_2] \neq \emptyset}{R_1 \bowtie_{p \wedge q} R_2}$$

This is basically a variant of Rule 1 (Predicate Pushdown): instead of first generating the entire cross product and post-filtering, the filter is taken into account directly when generating the tuples.

Important Transformation Rules

Rule 5: Join, Union, Intersection and Cross Product are commutative and associative.

Let R_1, R_2, R_3 be relations. Let $OP = \{\bowtie, \cup, \cap, \times\}$. Then it holds $\forall \oplus \in OP$:

1. $\oplus$ is commutative: $R_1 \oplus R_2 = R_2 \oplus R_1$, and
2. $\oplus$ is associative: $R_1 \oplus (R_2 \oplus R_3) = (R_1 \oplus R_2) \oplus R_3$.

Selectivity

Rule 6: Selections are interchangeable

Let p and q be predicates and let R be a relation. Then it holds:

$$\sigma_p\big(\sigma_q(R)\big) = \sigma_q\big(\sigma_p(R)\big).$$

This follows directly from Rule 3 and the commutative law.

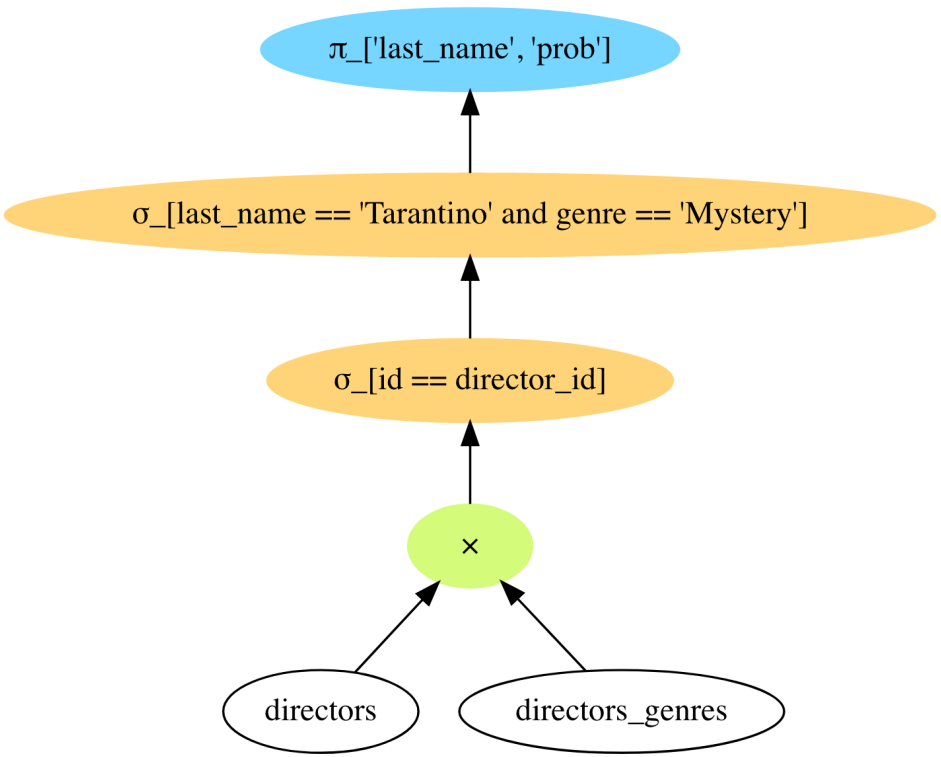
This plays performance-wise a role when one predicate is much more selective than the other.

Heuristic Optimisation in Python

In [4]:
```
# Unoptimized plan
cp = Cartesian_Product(directors, directors_genres)
sel1 = Selection(cp, "id == director_id")
sel2 = Selection(sel1, "last_name == 'Tarantino' and genre == 'Mystery'")
proj = Projection(sel2, ['last_name', 'prob'])

graph = proj.get_graph()
Source(graph)
```

Out[4]:



GitHub: Rule-based Optimization.ipynb

Induced Recursive Horizontal Partitioning

Induced Recursive Horizontal Partitioning

Let $p_0, p_1, \ldots, p_k$ be partitioning functions with $p_i : [R] \rightarrow D_i \ \forall_{0 \leq i \leq k}$.

Start: We apply p_0 to R. This results in a set of horizontal partitions $\{R_{i_0}\}$ for which $\forall_{t \in R}\ t \in R_{p_0(t)}$. (cf. Non-recursive Induced Horizontal Partitioning, 02 IMDb slides)

Recursion step: We apply p_{i+1} iteratively to all horizontal partitions, that were created by the previous partitioning step $p_{i \geq 0}$.

Set of induced partitions: This results in a set of horizontal partitions $\{R_{i_0,\ldots,i_k}\}, i_j \in D_{l_j}$.

Example:
$R = \{(2, A), (7, B), (1, B), (6, C)\}$
$k = 1, p_0 : [R] \rightarrow int, p_0(t) := t.a \bmod 2,\ p_1 : [R] \rightarrow char, p_1(t) := t.b,$

Induced recursive horizontal partitioning:

$R_{0,A} = \{(2, A)\},\ R_{0,C} = \{(6, C)\},\ R_{1,B} = \{(7, B), (1, B)\}$

Index

Index

Any set of (recursive or non-recursive) horizontal partitioning functions that can be used by a query to reduce the query's search space is called an *index* (or index structure).

More concretely, the goal of an index is to reduce the set of horizontal partitions we have to inspect to compute the result for a query.

Examples:
$[R] = \{[a : int, b : char]\}$
$R_{0,A} = \{(2, A)\}$, $R_{0,C} = \{(6, C)\}$, $R_{1,B} = \{(7, B), (1, B)\}$
$p_0(t) := t.a \bmod 2$, $p_1(t) := t.b$

$\Rightarrow \sigma_{b='C'}(R) = \sigma_{b='C'}(R_{0,C})$, i.e. filter only 1 tuple instead of 4

$\Rightarrow \sigma_{a=7}(R) = \sigma_{a=7}(R_{1,B})$, i.e. filter only 2 tuples instead of 4

$\Rightarrow \sigma_{a=8}(R) = \sigma_{a=8}(R_{0,A} \cup R_{0,C})$, i.e. filter only 2 tuples instead of 4

Query Types and their Relationship to Suitable Indexes

Point Query $\sigma_{A=c}$

Any query $\sigma_{A=c}$ is called a point query.

Range Query $\sigma_{c\in[low;high]}$ or $\sigma_{low\leq c\leq high}$

Any query $\sigma_{c\in[low;high]}$ or $\sigma_{low\leq c\leq high}$ is called a range query.

Typical Index Structures in SQL Engines

1. Hash tables: suitable for point queries (though special variants for range queries exist)
2. B-trees: suitable for both point and range queries
3. Bitmaps: suitable for point queries (though special variants for range queries exist)

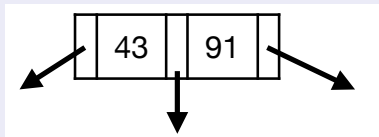
B-Trees

B-Tree: Nodes, Leaves, and the Root

A B-tree is an $(n+1)$-ary $(n \geq 1)$ tree consisting of two types of structures:

1. **Internal Nodes (or Nodes)**: A node contains n keys, where

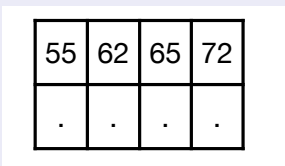
$$n \in \begin{cases} [k; 2k], & \text{if the node is not the root} \\ [1; 2k], & \text{if the node is the root} \end{cases}$$



A node contains $n+1$ pointers to the next tree level's nodes (or leaves).

2. **Leaf Nodes (or Leaves)**: A leaf contains n^* key/value-pairs, where

$$n^* \in \begin{cases} [k^*; 2k^*], & \text{if the leaf is not the root} \\ [1; 2k^*], & \text{if the leaf is the root} \end{cases}$$

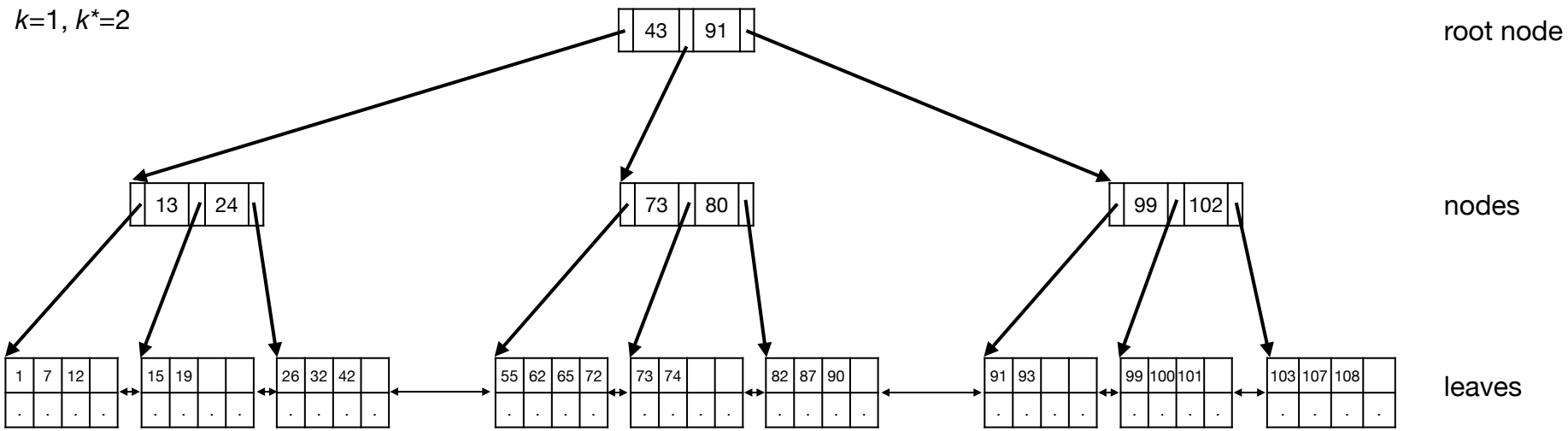


A leaf has one pointer to its left and one pointer to its right sibling.

The top-most node (or leaf) is called the root.

k and k^* can be defined freely, however, in practical implementations they are implicitly given by the maximum size of a node/leaf in bytes.

Example B-Tree



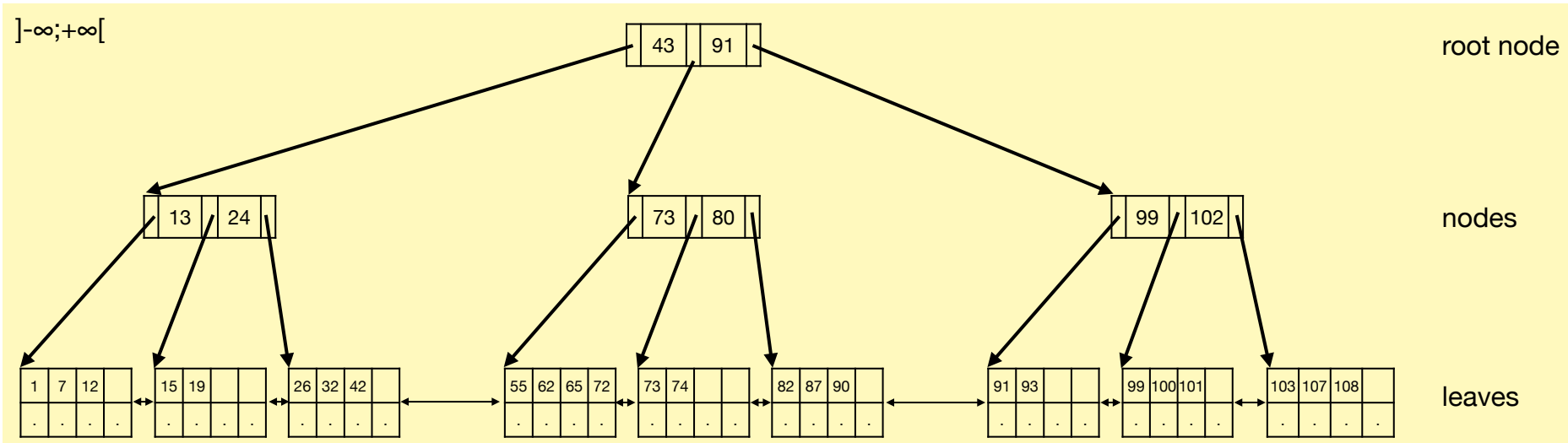
Point Query $\sigma_{A=c}$ Algorithm

Start at root and recursively pick suitable subtree that may contain c until leaf is reached.

Range Query $\sigma_{c\in[low;high]}$ or $\sigma_{low\leq c\leq high}$ Algorithm

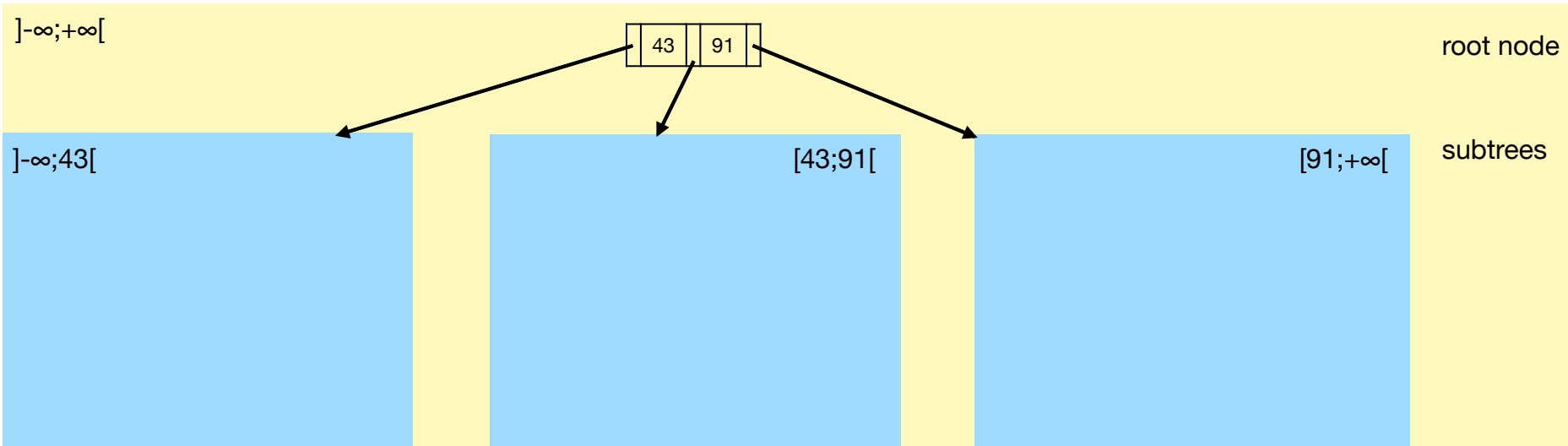
Run a point query $\sigma_{A=low}$. Then proceed along the leaves until an entry $e > high$ is found.

B-Tree Inception



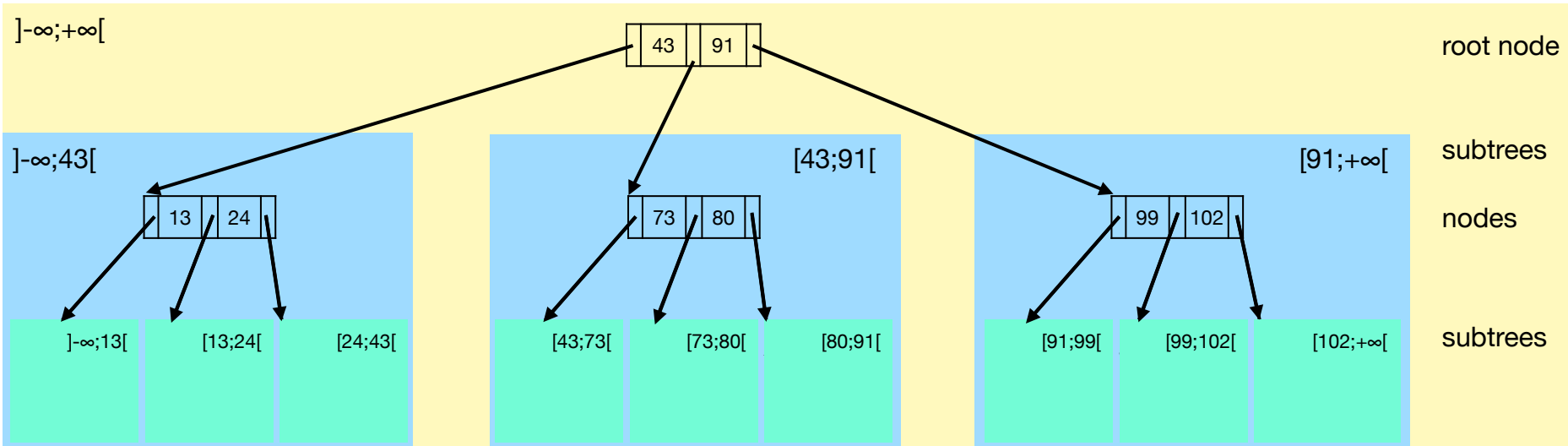
This is one tree, right?

One Root Node and its Subtrees



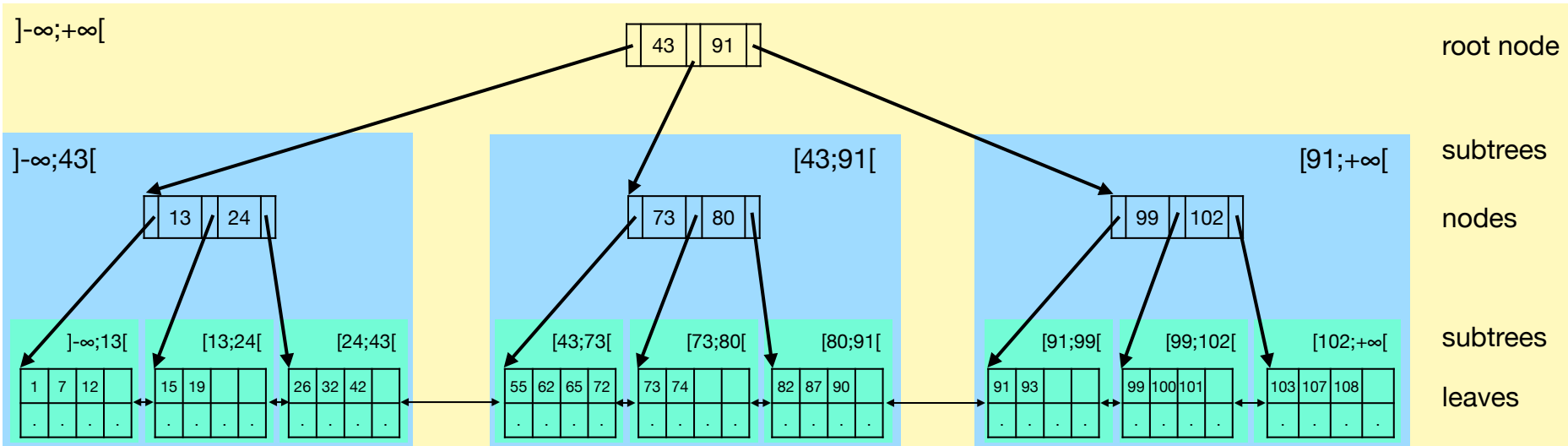
Actually, we can consider this tree a root node pointing to subtrees. We can ignore how those subtrees are organised internally.

One Root Node and its Subtrees of Subtrees



Recursively, each of those subtrees can be considered a root node pointing to subtrees ...

One Root Node and its Subtrees of Subtrees



Here, each of those subtrees is implemented by a single root leaf.

B-Tree Invariants and Rules

B-Tree Invariants

- **Inv1** each node implicitly creates a horizontal partitioning of the data
- **Inv2** at all times a node or leaf is at least half full (if not root)
- **Inv3** the path from the root to any leaf always has the same length
- **Inv4** entries inside each node and leaf are sorted to allow for binary search inside a node/leaf
- **Inv5** each leaf has pointers to its left and its right sibling (or subtree)

Rule 1

Any insert, update, and delete-operation on a B-Tree must preserve the B-Tree invariants. There is no Rule 2.

vs

Rule 2

There is no Rule 1.

What I mean by that: some B-Tree implementations 'relax' some of the textbook invariants, e.g. Inv2.

A File-System-based Index

In [10]:
```
# print the partitioning tree created by the recursive partitioning:
# note that the leaf nodes, i.e. the csv-files are not printed
tree('myindex',' ', False)
```

```
+myindex/
  +6/
  | +5/
  |   +5/
  +1/
  | +1/
  |   +9/
  |     +7/
  +2/
    +9/
    | +0/
    | +1/
    | +4/
    | +3/
    | +2/
    | +5/
    +8/
      +9/
      +0/
      +7/
      +6/
      +1/
      +8/
      +4/
      +3/
      +2/
      +5/
```

see Notebook: Indexing by Recursive External Partitioning.ipynb

Model vs Reality

Wrong models

All models are wrong, but some are useful.
[*George Box*]

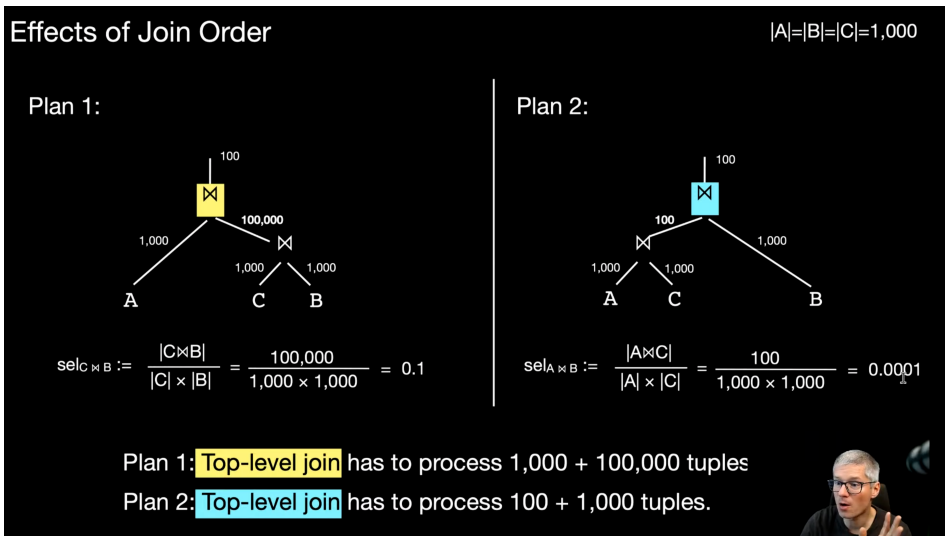
- **Never** confuse model and reality!
- Regularly compare model and reality!
- A model always has a purpose, a certain use.
- Certain things are simplified or omitted in the model in order to bring out others.
- That is the essence of:

Abstraction

"Thinking process of omitting details and transferring them to something more general or simple."

- Wikipedia: All models are wrong-History
- Wikipedia: Abstraction

Further material (in German and English)



Youtube Videos of Prof. Dittrich about Query Optimization (rule- vs cost-based, join graph, interesting orders) in English, also covers material from Part 2

- Chapter 8 in Kemper&Eickler in German
- Chapter 11.4 in Elmasri&Navathe in German or in English