HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

WOCHE 6

DYOD

AGENDA

- Q&A Sprint 3
- Review Sprint 2
- Query Processing
- Group Projects
- Benchmarking?

SPRINT 3

Questions?

REVIEW SPRINT 2

```
_attribute_vector =
    std::dynamic_pointer_cast<BaseAttributeVector>(
        std::make_shared<FittedAttributeVector<uint8_t>>(
            column.size()));
```

```
const std::shared_ptr<ValueColumn<T>>& p_column =
  std::dynamic_pointer_cast<ValueColumn<T>>(base_column);
```

```
const auto value_column =
    dynamic_cast<ValueColumn<T>*>(base_column.get());
```

REVIEW SPRINT 2

```
ValueID lower_bound(const AllTypeVariant& value) const {
  const T val = dynamic_cast<T>(value);

  if (!val) {
    return INVALID_VALUE_ID;
  }

  return lower_bound(val);
}
```

```
auto segment = std::dynamic_pointer_cast<ValueSegment<T>>(base_segment);
auto segment_values = segment->values();
```

REVIEW SPRINT 2 – CASTS

- Do not explicitly upcast pointers
- Do not use `static/dynamic_cast` on smart pointers
  - Check the return value of dynamic_pointer_casts (DebugAssert)
- If the type is already in the same line, do not repeat it - instead use `auto`
- Use `type_cast` for `AllTypeVariant`
- Do not use plain C-style casts

REVIEW SPRINT 2

```
_dictionary = std::make_shared<std::vector<T>>(segment_values);
std::sort(_dictionary->begin(), _dictionary->end());
_dictionary->erase(std::unique(_dictionary->begin(), _dictionary->end()),
_dictionary->end());


...

// fill attribute vector with valueIDs
for (ValueID position(0); position < segment->size(); position++) {
  auto value_id = ValueID(std::distance(_dictionary->begin(), std::find(_dictionary-
>begin(), _dictionary->end(),
                    segment_values.at(position))));
  _attribute_vector->set(position, value_id);
}
```

REVIEW SPRINT 2

```
const auto entropy = ((int) std::log2(_dictionary->size())) + 1;

if(entropy <= 8){
  _attribute_vector =
std::make_shared<FixedSizeAttributeVector<uint8_t>>(FixedSizeAttributeVector<uint8_t>());
} else if(entropy <= 16){
  _attribute_vector =
std::make_shared<FixedSizeAttributeVector<uint16_t>>(FixedSizeAttributeVector<uint16_t>());
} else if(entropy <= 32) {
  _attribute_vector =
std::make_shared<FixedSizeAttributeVector<uint32_t>>(FixedSizeAttributeVector<uint32_t>());
} else{
  throw std::runtime_error(std::string("Not enough memory"));
}
```

REVIEW SPRINT 2

```
void append(const AllTypeVariant&) override {}
```

```
std::mutex compression_mutex;
```

```
18  - add_compile_options(-std=c++1z -pthread -Wall -Wextra -pedantic -Werror -Wno-unused-parameter)
```

“Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any `-std` option used.”

REVIEW SPRINT 2

```
TEST_F(StorageDictionarySegmentTest, SortedValues){
  // Setup
  vc_int->append(87);
  vc_int->append(90);
  vc_int->append(3);
  auto col = opossum::make_shared_by_data_type<opossum::BaseSegment,
opossum::DictionarySegment>("int", vc_int);
  auto dict_col = std::dynamic_pointer_cast<opossum::DictionarySegment<int>>(col);
  //std::cout<<(dict_col->get((opossum::ValueID)0))<<std::endl;
  EXPECT_EQ(dict_col->value_by_value_id((opossum::ValueID)0), 3);
}
```

REVIEW SPRINT 2

```
TEST_F(StorageTableTest, CompressTable) {
  t.compress_chunk((ChunkID) 1);
  EXPECT_TRUE(true);
}
```

```
TEST_F(StorageTableTest, CompressChunkReplacesWithDictionarySegment) {
  t.append({4, "Hello,"});
  t.append({6, "world"});

  t.compress_chunk(ChunkID{0});
  auto& chunk = t.get_chunk(ChunkID{0});
  auto segment_ptr = chunk.get_segment(ColumnID{0});
  auto dictionary_segment_ptr = std::dynamic_pointer_cast<DictionarySegment<int>>(segment_ptr);
  EXPECT_TRUE(dictionary_segment_ptr);
}
```

REVIEW SPRINT 2

2 .gitignore

```
@@ -38,3 +38,5 @@ cmake-build-*
38 + # llvm coverage information
39   default.profdata
40   default.profraw
```

```
38   # llvm coverage information
39   default.profdata
40   default.profraw
41 +
42 + /src/bin/playground.cpp
```

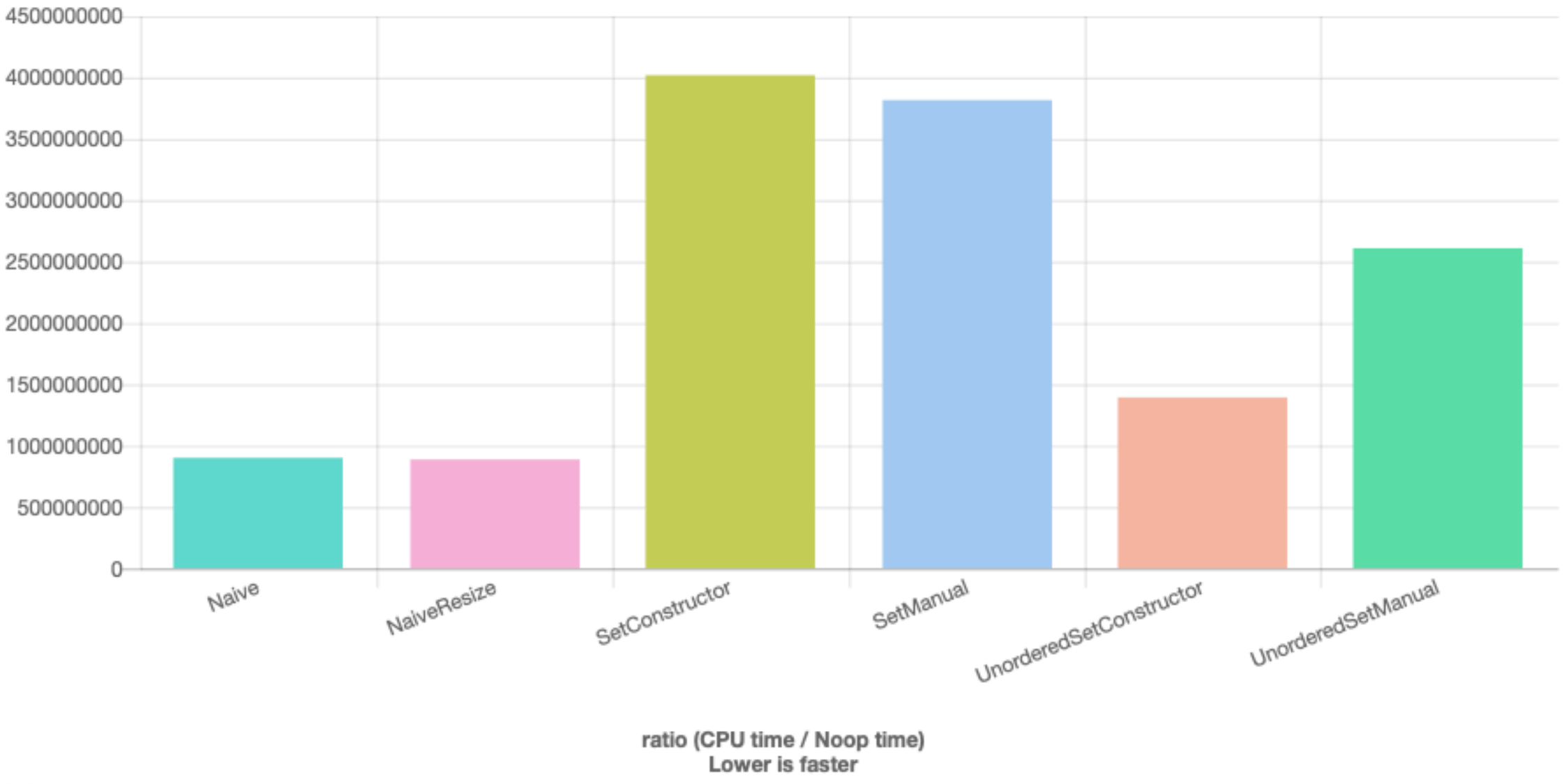
```cpp
size_t attribute_vector_size = 0;
std::set<T> dictionary_helper;

for (size_t segment_iterator = 0; segment_iterator < base_segment->size(); ++segment_iterator) {
  dictionary_helper.insert(type_cast<T>((*base_segment)[segment_iterator]));
  attribute_vector_size++;
}

_dictionary->reserve(dictionary_helper.size());
for (auto it = dictionary_helper.begin(); it != dictionary_helper.end();) {
  _dictionary->emplace_back(std::move(dictionary_helper.extract(it++).value()));
}
```

CREATING A SORTED, UNIQUE DICTIONARY

- How can we derive a sorted and unique `std::vector` from a non-sorted ValueSegment that might contain duplicates?
  - `std::sort, std::unique, std::erase`
  - `std::sort, std::unique, std::resize`
  - `std::set`
  - `std::unordered_set`
  - `std::map` as intermediary structure
- Benchmark above on vector of 500,000 `std::strings`

CREATING A SORTED, UNIQUE DICTIONARY

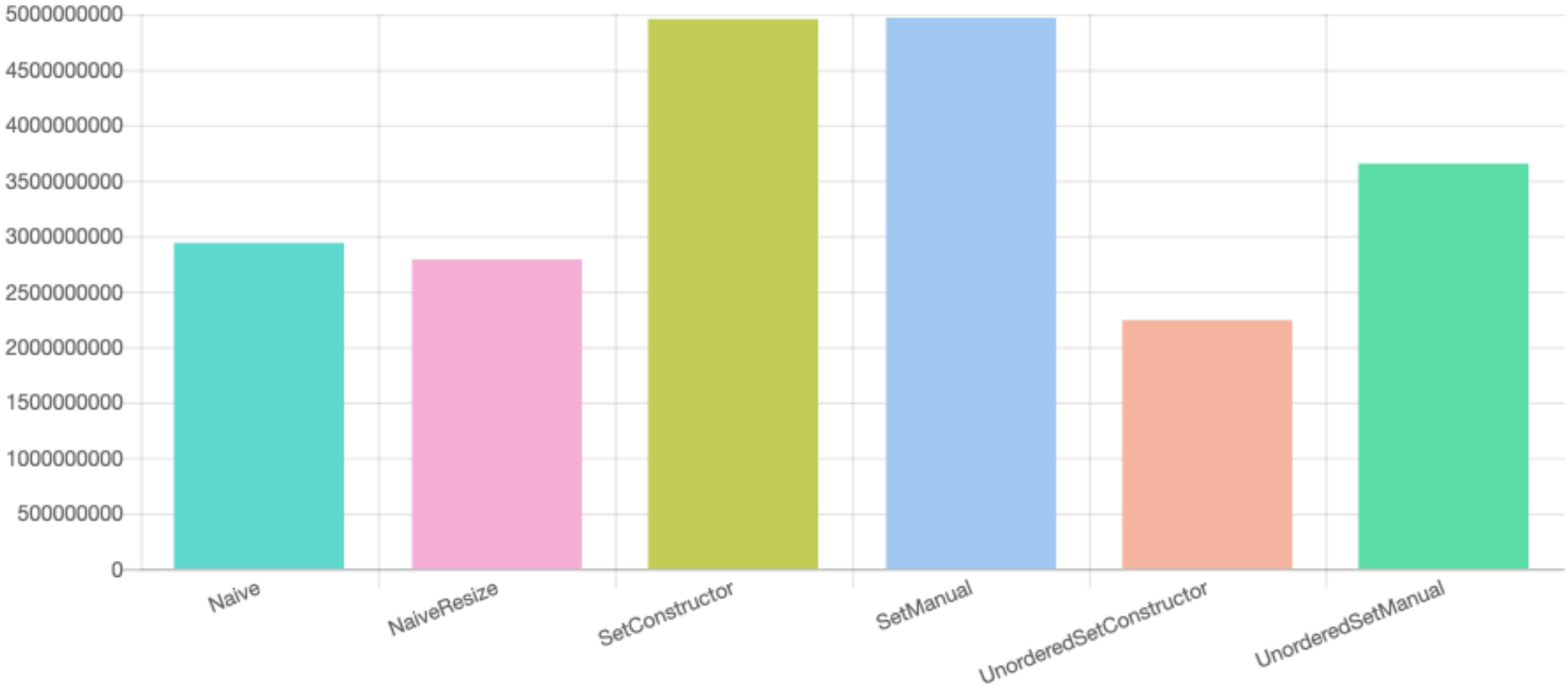
String length 10 characters



ratio (CPU time / Noop time)
Lower is faster

CREATING A SORTED, UNIQUE DICTIONARY

String length 30 characters



ratio (CPU time / Noop time)
Lower is faster

ESTIMATE MEMORY USAGE

Query Processing

Motivation

How does a database actually process incoming SQL queries?

How does a database process queries?

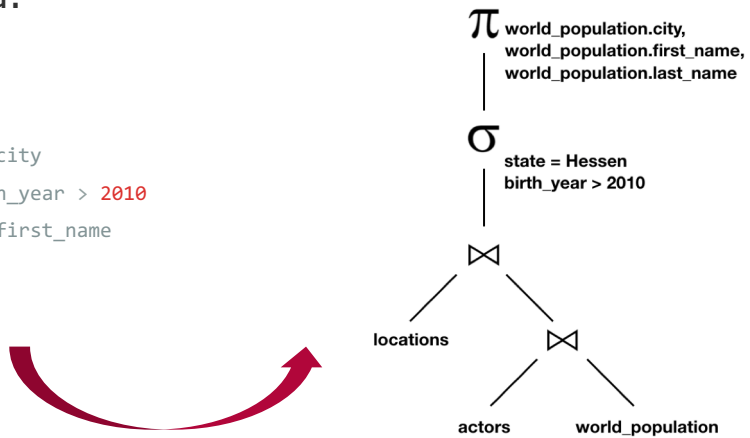
1. The database receives the SQL queries on the network interface and passes it to the SQL parser.

```
1  SELECT wp.city , wp.first_name, wp.last_name
2  FROM world_population AS wp
3  INNER JOIN locations ON wp.city = locations.city
4  WHERE locations.state = 'Hessen' AND wp.birth_year > 2010
5  INNER JOIN actors ON actors.first_name = wp.first_name
6  AND actors.last_name = wp.last_name
```

How does a database process queries?

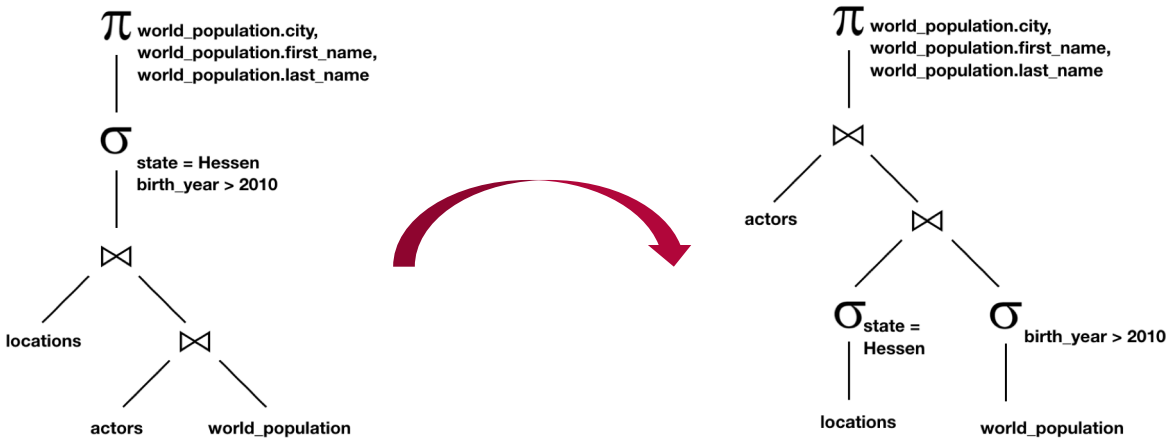
2. The SQL parser generates a logical query plan. This plan contains the **relational operators** required to execute the query and the order in which they have to be called.

```
1  SELECT wp.city , wp.first_name, wp.last_name
2  FROM world_population AS wp
3  INNER JOIN locations ON wp.city = locations.city
4  WHERE locations.state = 'Hessen' AND wp.birth_year > 2010
5  INNER JOIN actors ON actors.first_name = wp.first_name
6  AND actors.last_name = wp.last_name
```



How does a database process queries?

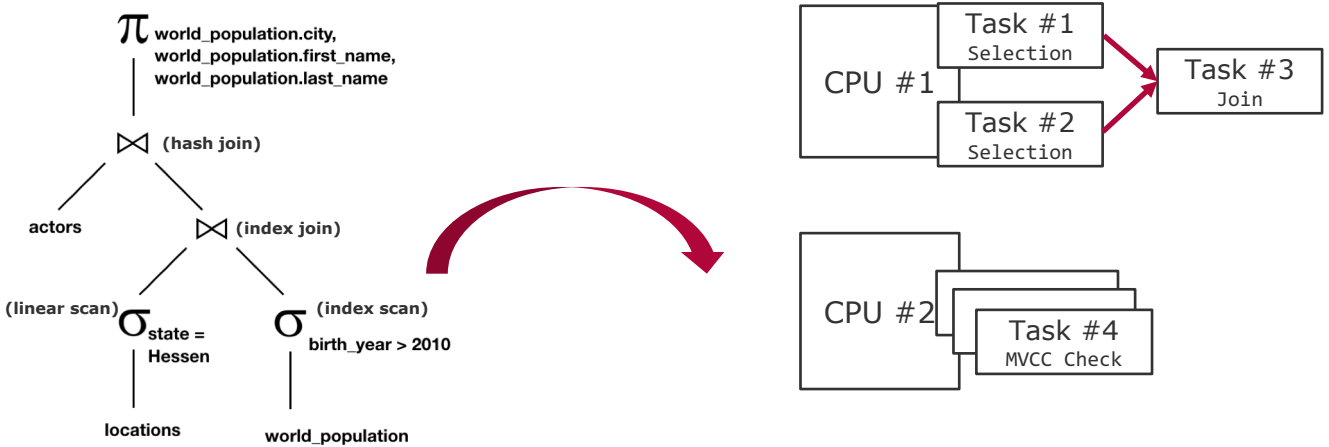
3. Depending on the order of operations in the query plan, runtimes can differ by orders of magnitude. Thus, the database employs the **query optimizer** to determine efficient query plans.



How does a database process queries?

SQL Parsing → Plan Building → Optimization → Translation → Execution

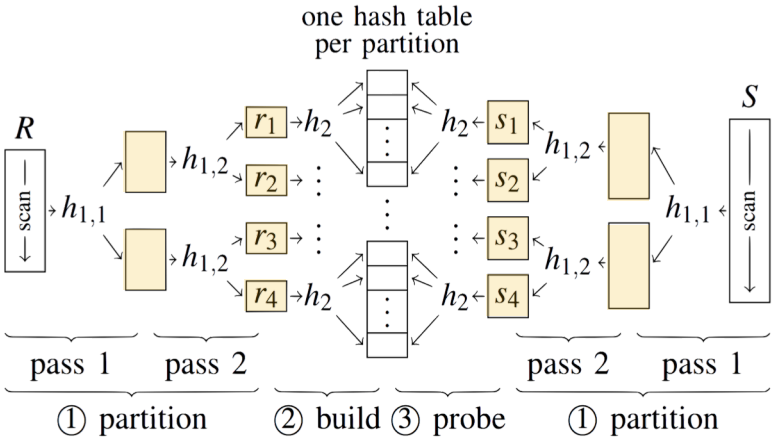
4. After a logical query plan is decided upon, the relational operators are translated to their actual implementations. Further, the **database scheduler** can determine where & when to run the query and how much resources to allocate.



How does a database process queries?

SQL Parsing → Plan Building → Optimization → Scheduling → Execution

5. Finally, the database executes all scheduled tasks and returns the result set to the user.



Query Optimization

Query Optimization
Motivation

Often, the impact of the query optimizer is much larger than the impact of the runtime system [..] Changes to an already tuned runtime system might bring another 10% improvement, but changes to the query optimizer can often bring a factor 10.

T. Neumann. Engineering high-performance database engines. PVLDB, 2014

Query Optimization
Motivation

- For a given query (remember: SQL is declarative), there is a large array of alternative (logically equivalent) query plans
- The query optimizer is a module that enumerates possible query plans and estimates the costs of each plan.
  - Usually selects the plan with the lowest estimated costs.

Costs to consider

- **Algorithmic:** e.g., runtime complexity of different SORT operators
- **Logical:** estimated output size of the operator (e.g., decreasing for filter operations, de- or increasing for joins)
- **Physical:** hardware-dependent cost calculations such as IO bandwidth, cache misses, etc.

Query Optimization
Creating Query Plans

- Operator costs are often interacting with each other, making accurate cost estimations computationally expensive
- As a consequence, most optimizers concentrate on logical costs and thrive to reduce operator results as early as possible
- Reducing logical costs further leads to less memory traffic, which indirectly improves NUMA performance, cache hit rates, and more

How can we reduce the intermediate result size of a query plan (i.e., logical costs) as early as possible?

Execute operators first that exclude large fractions of data (e.g., equi-filters on attributes with many distinct values, joins on foreign keys, etc.)

Query Optimization Introduction

Query optimization can be seen as a two-step process

1. **Semantic query transformations** and **simple heuristics** to reformulate queries
2. **Cost model-driven** approaches that **estimate costs** in order to reorder operators

Query Optimization
Semantic Transformations & Heuristics

Query reformulation: exploit semantic query transformations and simple heuristics to reformulate a query plan to a (logically equivalent) plan with lower expected costs.

```
SELECT * FROM T
WHERE A < 10 AND A > 12
```

» return empty result

```
SELECT * FROM T
WHERE A < 10 AND A < 20
AND A IS NOT NULL
```

» SELECT * FROM T WHERE A < 10

Query Optimization
Semantic Transformations & Heuristics

```
SELECT * FROM T1,
(SELECT * FROM T) AS T2
WHERE T2.B > 17
```

»

```
SELECT * FROM T1,
(SELECT * FROM T WHERE B > 17) AS T2
```

```
SELECT (A + 2) + 4 FROM T
» SELECT A + 2 + 4  FROM T
» SELECT A + 6      FROM T
```

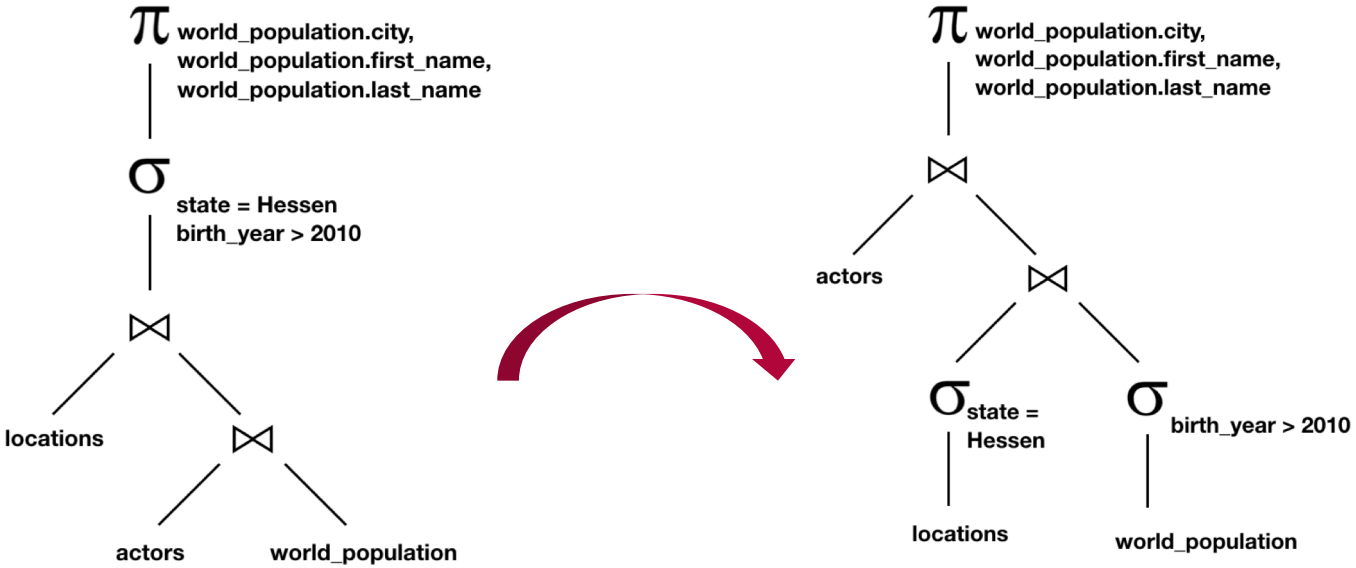
Query Optimization
Semantic Transformations & Heuristics

- Optimization heuristics:
  - Execute most restrictive filters first
  - Execute filters before joins
  - Predicate/limit push downs
  - Join reordering based on estimated cardinalities

- Such optimizations are heuristics as they are usually good estimates of operator costs.
- Nonetheless, possible that joining before filtering can lead to a better query runtime for certain constellations.

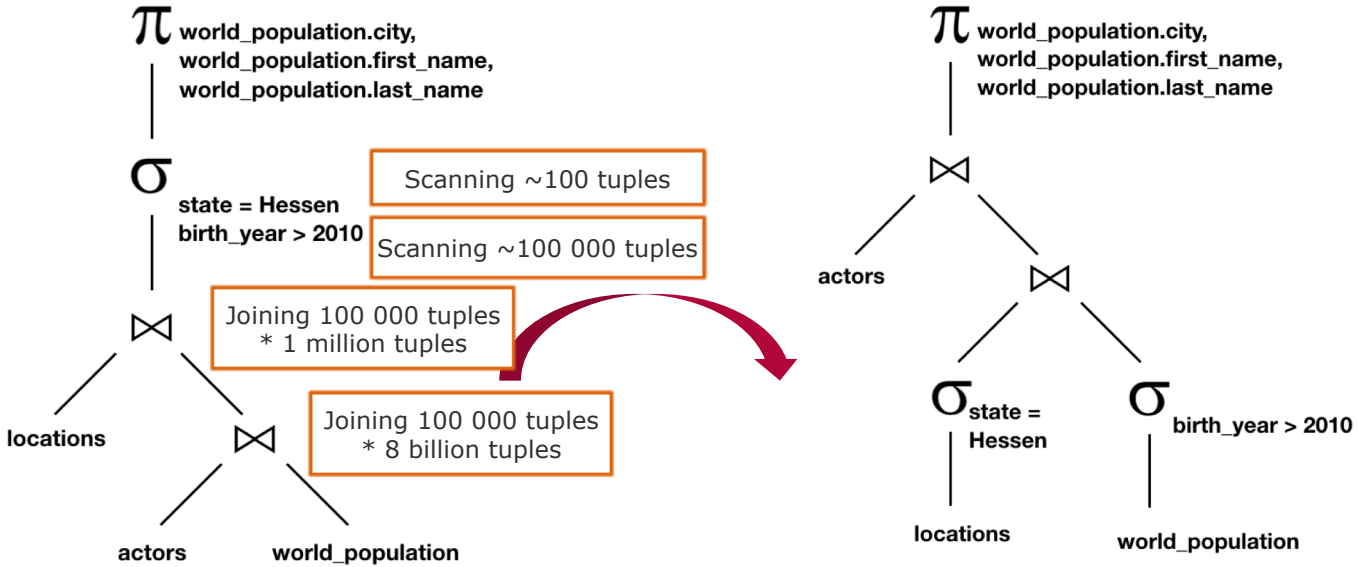
Query Optimization
Query Plan Reformulation

- Logical Query Plan can be seen as a tree of relational algebra operators
- Enumeration phase generates logically equivalent expressions using equivalence rules (i.e., operators can only be reordered to an extend that ensures correct results)



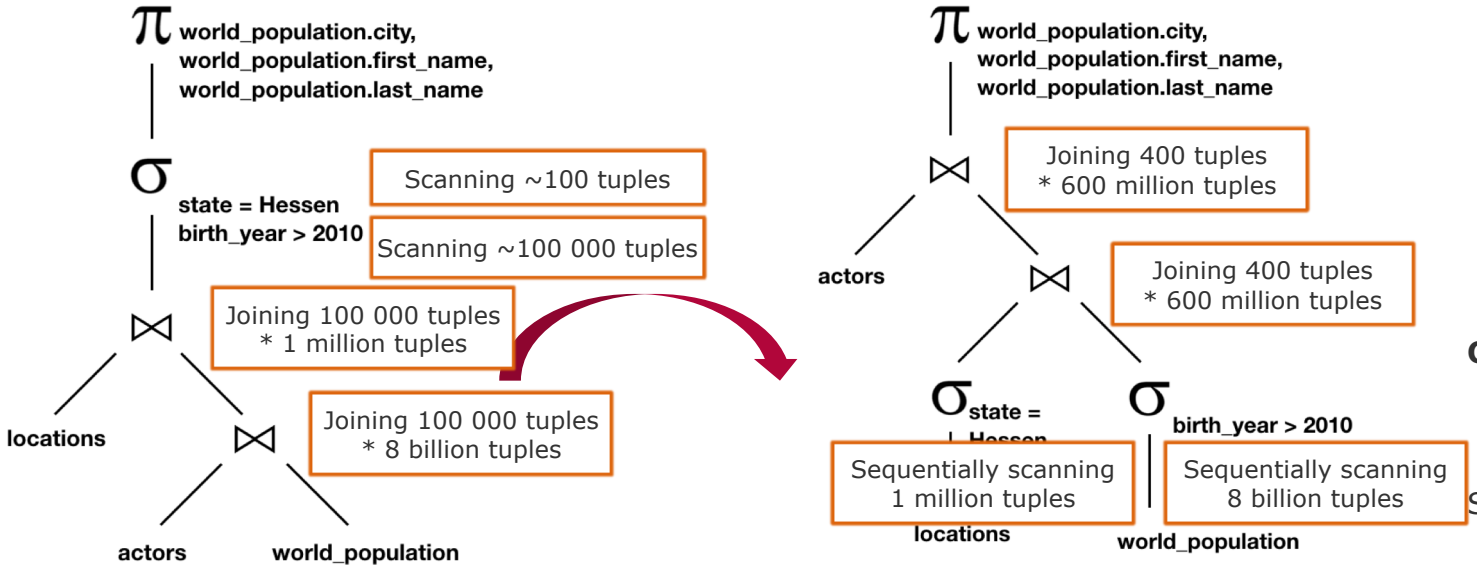
Query Optimization
Query Plan Reformulation

- Logical Query Plan can be seen as a tree of relational algebra operators
- Enumeration phase generates logically equivalent expressions using equivalence rules (i.e., operators can only be reordered to an extend that ensures correct results)



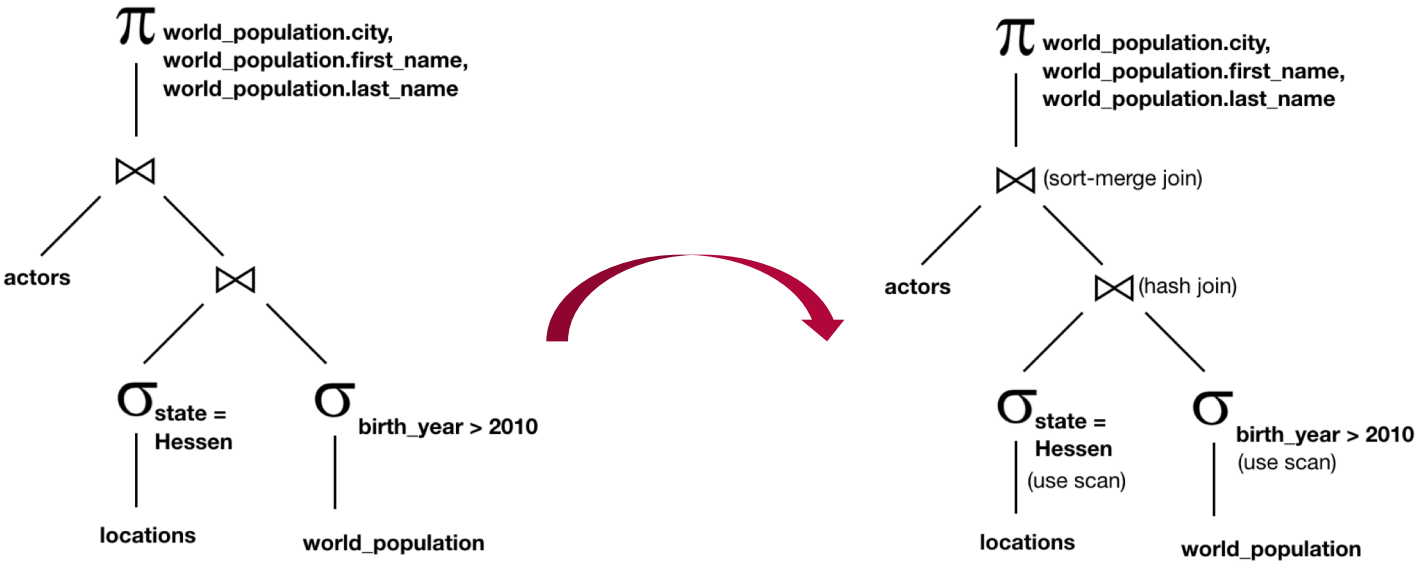
Query Optimization
Query Plan Reformulation

- Logical Query Plan can be seen as a tree of relational algebra operators
- Enumeration phase generates logically equivalent expressions using equivalence rules (i.e., operators can only be reordered to an extend that ensures correct results)



Query Optimization
Physical Query Plan

The Physical Query Plan/Evaluation Plan defines which algorithm is used for each operation, and how the execution of operations is coordinated.



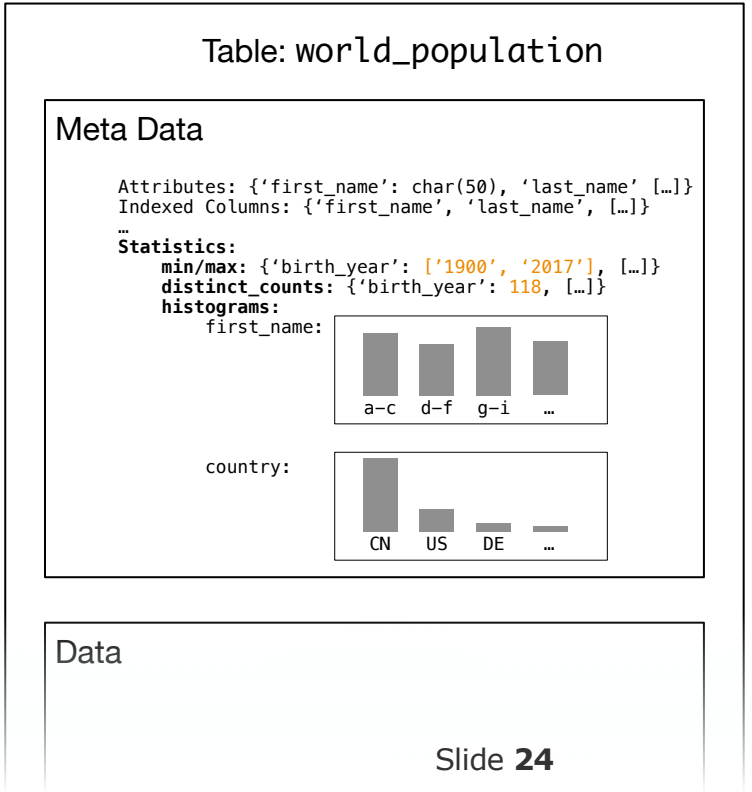
Query Optimization
Statistics

- Statistics are, e.g., used to estimate intermediate result size for logical cost estimations to compute overall cost of complex expressions.
- Especially for cost model-driven approaches, accurate statistics are indispensable.
- Such statistics include:
  - Number of distinct values for a table
  - Presence or absence of indices
  - Value distribution of attributes (e.g., histograms)
  - Top-n values with occurrence count
  - Min/Max values

http://www.cbcb.umd.edu/confcour/Spring2014/CMSC424/query_optimization.pdf

Query Optimization
Statistics

- Accuracy of estimation depends on quality and currency of statistical information DBMS holds
- Keeping statistics up to date can be problematic
  - Updating them on the fly increases load on latency-critical execution paths
  - Updating them periodically (e.g., during chunk compression in Hyrise[2]) might introduce misleading estimations due to outdated statistics



Query Optimization
Join Ordering

The task of join ordering is to find a join order that is estimated to have the lowest costs (ordered by input and output cardinality).

To do so, we need to estimate the size of the join result (so-called *join cardinality estimation*):

- Knowledge about foreign key relationships can be used
- Values are rarely uniformly distributed, histograms help estimating
- But histograms do not contain correlation information

Query Optimization
Join Ordering

For all relations $r1$, $r2$, and $r3$,

$$(r1 \bowtie r2) \bowtie r3 = r1 \bowtie (r2 \bowtie r3)$$

→ Join Associativity

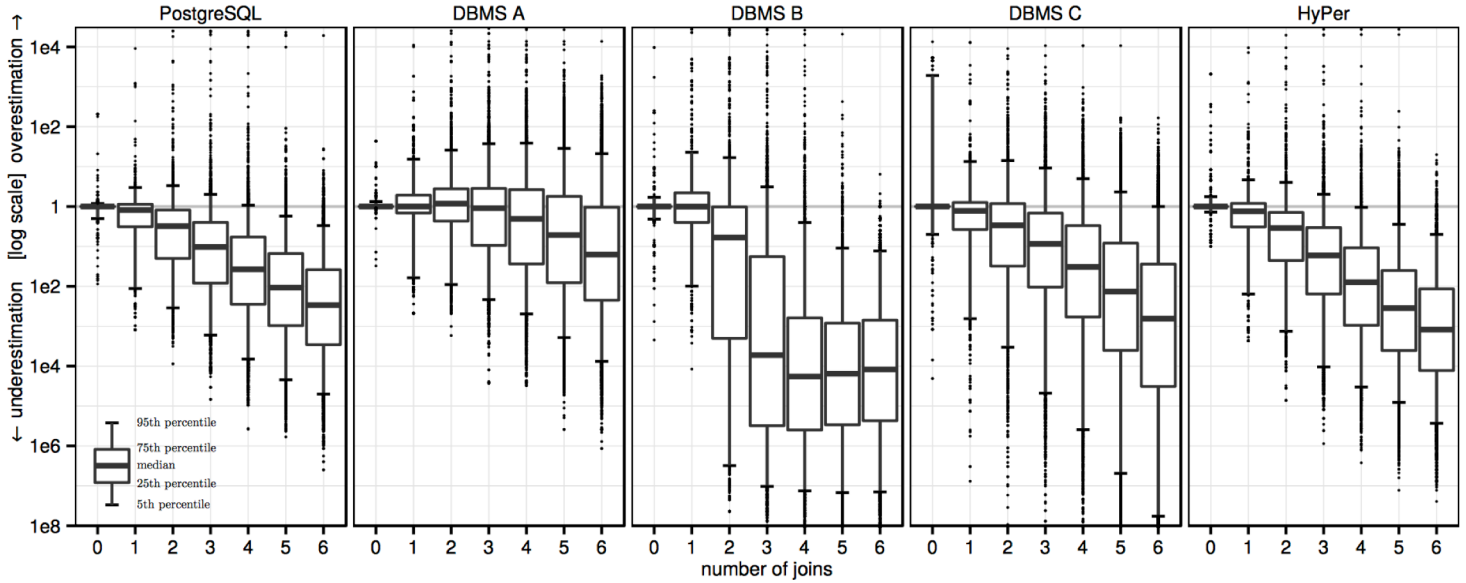
If $r2 \bowtie r3$ is quite large and $r1 \bowtie r2$ is small, we choose

$$(r1 \bowtie r2) \bowtie r3$$

so that we compute and store a smaller temporary relation.

Query Optimization
Join Ordering

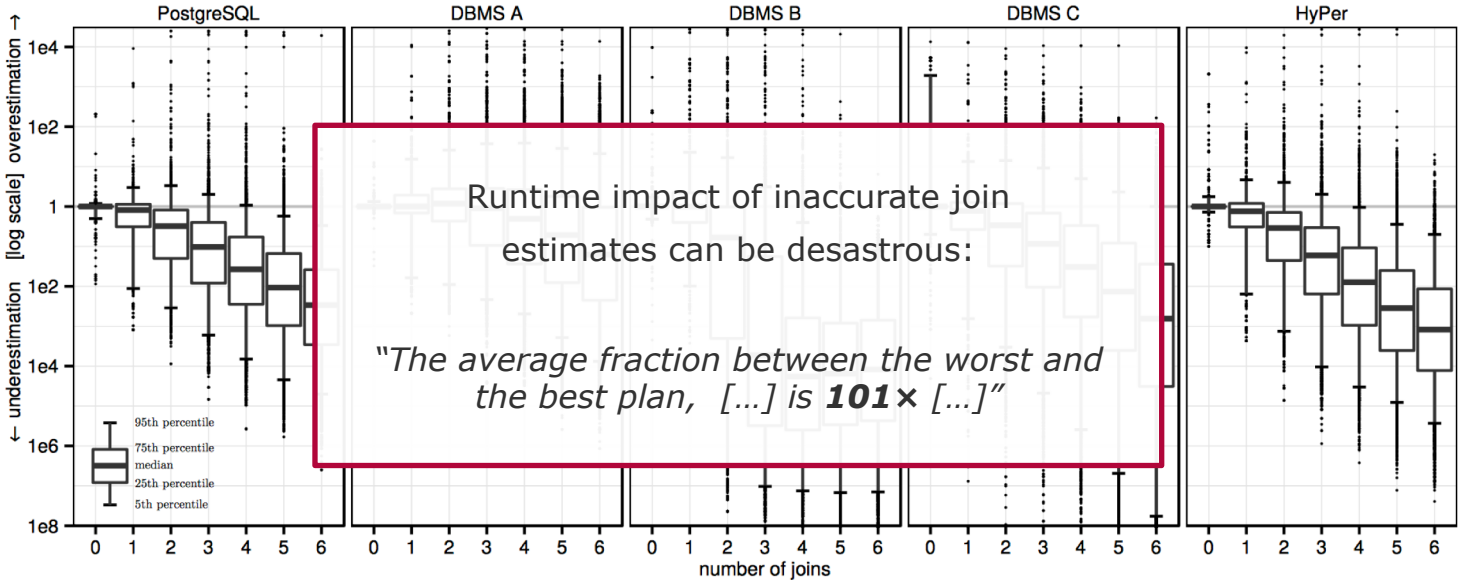
Estimating join cardinalities is one of the challenging tasks of query optimization, but also indispensable to performance.



Leis et al., How Good Are Query Optimizers, Really?, PVLDB, vol 9, no 3, 2015.

Query Optimization
Join Ordering

Estimating join cardinalities is one of the challenging tasks of query optimization, but also indispensable to performance.



Runtime impact of inaccurate join estimates can be desastrous:

*"The average fraction between the worst and the best plan, […] is **101×** […]"*

Leis et al., How Good Are Query Optimizers, Really?, PVLDB, vol 9, no 3, 2015.

Query Optimization Summary

We learned that query optimization becomes increasingly important due to …

- ever growing data sets
- increasingly complex queries.

However, finding efficient plans remains a challenging task as …

- the number of possible plans is enormous, and
- costs rely on estimation using potentially outdated statistics.

GROUP TOPICS

- Query Plan Cache Parametrization (JK)
- Smart Positions Lists (JK)
- Cost Model Calibration (MB/JK)
- Utilize sortation during query execution (MB)
- More Statistics (MB)
- Faster Statistics (TB)
- Set Operations (MD)
- Speed-up sorting (MD)
- Transactions and benchmarking over the network (SH)

QUERY PLAN CACHE PARAMETRIZATION

- **Introduction:**
  - Complex transformations create imperative query plans from declarative SQL queries
  - Query plans are cached to avoid expensive repeated transformations/optimizations
- **Motivation**
  - For fast, short-running queries, optimizations cause significant overhead
  - Our current cache can only handle identical queries: `WHERE x = 4` != `WHERE x = 5`
- **Tasks**
  - Enable caching for almost identical queries based on parsing structures
  - Use data statistics to determine when a plan could be reused for similar queries
- **Evaluation**
  - Investigate impact on selected TPC-C and TPC-H benchmark queries

SMART POSITION LISTS

- **Motivation**
  - Currently, position lists are (almost) only wrapping `std::vectors<RowID>`
  - A little bit of additional state/behavior offers potential for performance optimizations
- **Tasks**
  - Introduce a `matches_all` flag to avoid costly translations from Data- to Reference-Tables
  - If all rows reference the same chunk, an `std::vector<ChunkOffset>` is sufficient (IndexScan)
  - Further ideas: nullable, sortation information
- **Evaluation**
  - Investigate impact on TPC-C, -H, -DS, and the Join Order Benchmark

COST MODEL CALIBRATION

- **Cost Models:** predict the execution time of database operators
  - Cost models are often built with the help of statistics or machine learning techniques
- **Motivation**
  - Such learned cost models need to be trained on data that allows to generalize for different workloads
  - This training data must be obtained quickly
- **Tasks**
  - Generate and execute *calibration* queries that enable generalization and export the results
  - Train simple models on the observed measurements
- **Evaluation**
  - Investigate the models' accuracy for the TPC-H benchmark

SORT-BASED QUERY EXECUTION

- **Motivation**
  - Sorted data allows for various optimizations (e.g., binary search)
  - Several operators in Hyrise profit from sorted input, which can be the result of previous operators (e.g., sort, sort-merge joins, …)
- **Tasks**
  - Improve the **passing of *sort information*** throughout logical and physical plans
  - Improve existing operators to **make use of the *sort information***
  - Implement simple and defensive **optimizer rules** when to use sort-based operators
- **Evaluation**
  - Measure the runtime effects for TPC-H

BETTER ESTIMATIONS THROUGH SAMPLING

- **Motivation**
  - To optimize a query, accurate cardinality estimations are mandatory
  - Hyrise uses histograms, which can be very inaccurate for string estimations or outliers
- **Tasks**
  - Implement sampling in Hyrise, focus on efficiency
  - For every sufficiently large table, a small sample is taken which is processed whenever histograms are expected to be inaccurate (or always?)
- **Evaluation**
  - Measure the effects on estimation accuracy for an array of different cases
  - Evaluate the memory overhead as well as the runtime overhead for sample collection and cardinality estimation using samples

FASTER STATISTICS GENERATION

- **Motivation**
  - Cost-based query optimization depends on accurate cost estimates
  - Cost estimates result from a cost model and summary statistics (histograms, samples, sketches)
  - Hyrise employs histograms, which can be costly to generate (for many attributes, very accurate)
    - This hinders experimentation, benchmarks, and practical statistics updates!
- **Tasks**
  - Extend binary data export/import with statistics
  - Parallelize histogram generation at segment level; merge per-segment histograms
  - Meet the scheduler and the profiler
- **Evaluation**
  - Measure the effects of the parallelization on histogram and estimation accuracy
  - Evaluate the runtime gains of the various improvements

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

TRANSACTIONS AND BENCHMARKING OVER THE NETWORK

- **Hyrise Network Interface**
  - Implements the PostgreSQL wire protocol
  - We believe it has decent performance, but it is currently difficult to benchmark, because (1) functionality, e.g., support to load data and transaction support, and (2) tool support are missing
- **Motivation**
  - Network is the primary interface for a database
  - Besides good performance, the network interface must provide functionality to the user
- **Tasks**
  - Add and maintain transaction state information for database connections
  - Integrate and run existing TPC-C and TPC-H benchmarks in Hyrise
- **Evaluation**
  - Demonstrate transaction support via the network interface
  - Compare the TPC-H benchmark performance of the Hyrise library and server for different data set sizes

Faster Sort

- The current sort implementation was one of the first operators in Hyrise and has been practically untouched since then
- Improvements in the query plans and other operators mean that the performance of sort now becomes an issue
- Challenges:
  - Sorting across multiple columns – can we do better than sorting multiple times?
  - Exploiting information from the encoding – can dictionary encoding speed up the sort process?
  - Parallelism – can we sort segments in parallel and merge the results?

SQL Set Operations

SELECT name FROM students **UNION** SELECT name FROM teachers WHERE name LIKE ‘Peter%’

- SQL supports three different (multi-)set operations
  - UNION (ALL), INTERSECT, EXCEPT
- Supporting these enables multiple TPC-DS queries and opens up new optimization challenges
  - In the example above, can we push the predicate below the set?
- This project is a good chance to work on different steps in the pipeline, including SQL parsing and translation, optimization, and execution

NEXT STEPS

- Please send us a list of **all topics that you are interested in** until Sunday, 24 November, 23:59pm CET.
- The current groups stay the same for the project phase
- All choices have the same priority and you can submit as many choices as you want.
- The supervisors are not fix yet.
- For further questions, send an email to:
  - Martin Boissier, Markus Dreseler, Stefan Halfpap, Jan Kossmann, Thomas Bodner