Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Build your own Database

Week 6

Agenda

- Q&A Sprint 3
- Review Sprint 2
- Benchmarking
- Group Projects
- NULL values in SQL
- Query Pipeline

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);
}
```

Review Sprint 2 - Casts

- Do not explicitly upcast pointers
- Do not use `static/dynamic_cast` on smart pointers
- If you already have the type in the same line, do not repeat it – instead, use auto
- Use `type_cast` for `AllTypeVariant`

Review Sprint 2

```
ValueID lower_bound(T value) const {
  for (auto it = _dictionary->begin(); it < _dictionary->end(); ++it) {
    if (*it >= value) {
      return static_cast<ValueID>(it - _dictionary->cbegin());
    }
  }
  return INVALID_VALUE_ID;
}
```

Review Sprint 2

```
explicit DictionaryColumn(const
std::shared_ptr<BaseColumn>& base_column) {
  _dictionary = std::make_shared<std::vector<T>>();
  _build_dictionary(base_column);
  _assign_attribute_vector(base_column->size());
  _build_attribute_vector(base_column);
}
```

```
const T DictionaryColumn<T>::get(const size_t i) const {
  return _dictionary->at(_attribute_vector->get(i));
}
```

Review Sprint 2

```
template <typename T>
ValueID FittedAttributeVector<T>::get(const size_t i) const {
  if (i >= _entries.size()) {
    throw std::runtime_error("Index out of range");
  }
  return ValueID(_entries.at(i));
}
```

Review Sprint 2

```
DebugAssert(set_pos < num_unique_elements,
            "The value " + type_cast<std::string>(value) +
            " is not in the dictionary :(");
```

```
16  +   ValueID get(const size_t offset) const override {
17  +     DebugAssert(offset < _values.size(), "invalid offset");
18  +     return ValueID{_values[offset]};
```

17 hours ago • edited

you could use _values.at(...) for runtime bounds checking if required.

Reply...

Review Sprint 2

```
std::for_each(value_segment->values().cbegin(), value_segment->values().cend(),
              [&](const auto& value) {
                 const auto search_iter = std::find(_dictionary_vector->cbegin(),
                                                    _dictionary_vector->cend(), value);
                 _attribute_vector->append(ValueID(std::distance(_dictionary_vector->
                                                                 cbegin(), search_iter)));
});
```

Shorter:

```
for (const auto& value : value_segment->values())
  const auto search_iter = std::find(_dictionary_vector->cbegin(),
                                     _dictionary_vector->cend(), value);
  _attribute_vector->append(ValueID(std::distance(_dictionary_vector->
                                     cbegin(), search_iter)));
});
```

Review Sprint 2

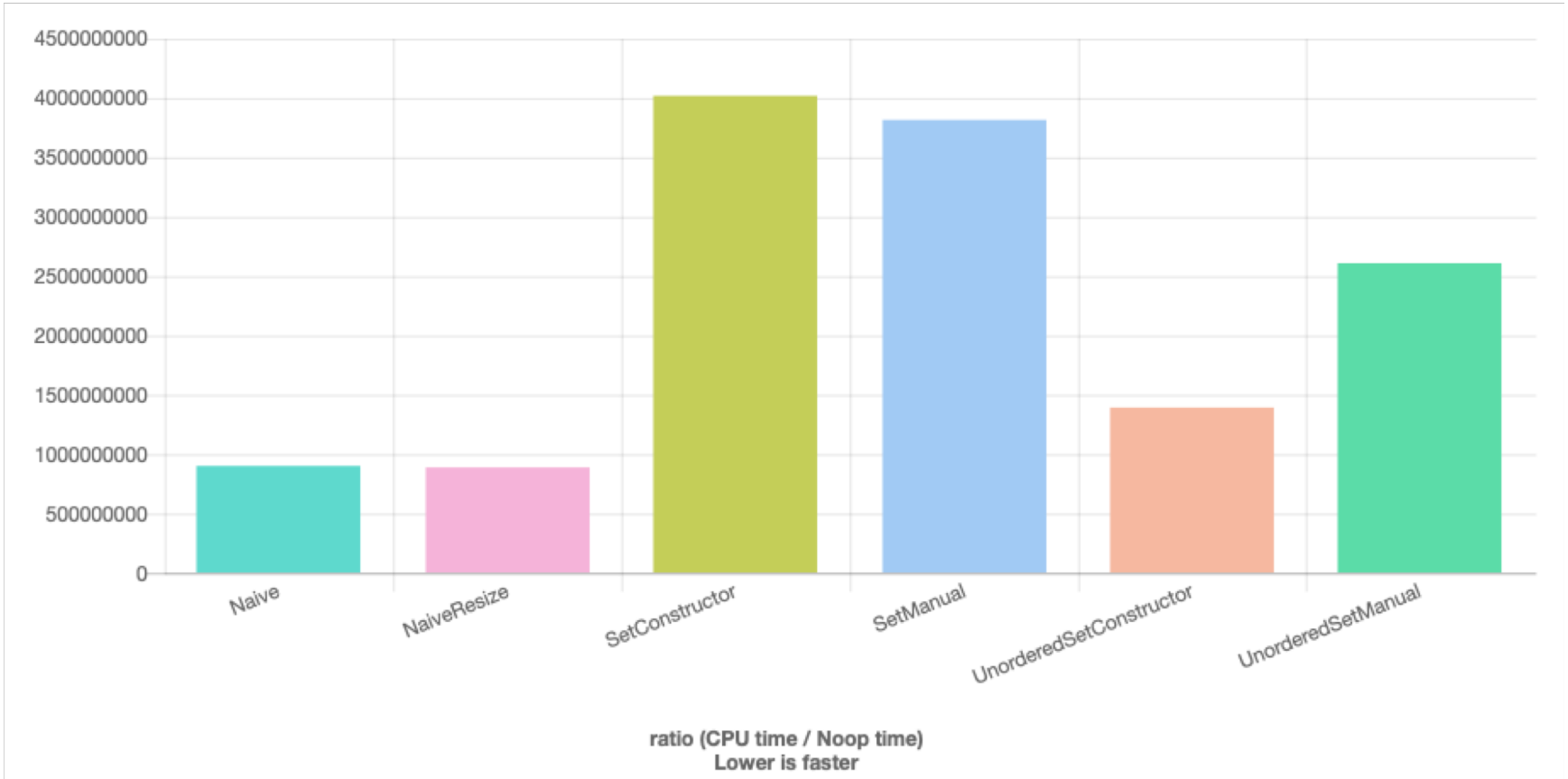
```
template <typename uintX_t>
void FittedAttributeVector<uintX_t>::set(const size_t element, const
                                         ValueID value_id) {
  DebugAssert(element < std::pow(2, (8 * width())), "Index out of
              bounds exception");
  _value_references.insert(_value_references.begin() + element,
                           value_id);
}
```

Sorting and Enforcing Uniqueness

- How can we derive a sorted and unique vector from a non-sorted one that possibly contains duplicates?
  - `std::sort, std::unique, std::erase`
  - `std::sort, std::unique, std::resize`
  - `std::set`
  - `std::unordered_set`
  - `std::map` as intermediary structure
- Benchmark operation on vector of 500.000 `std::strings`

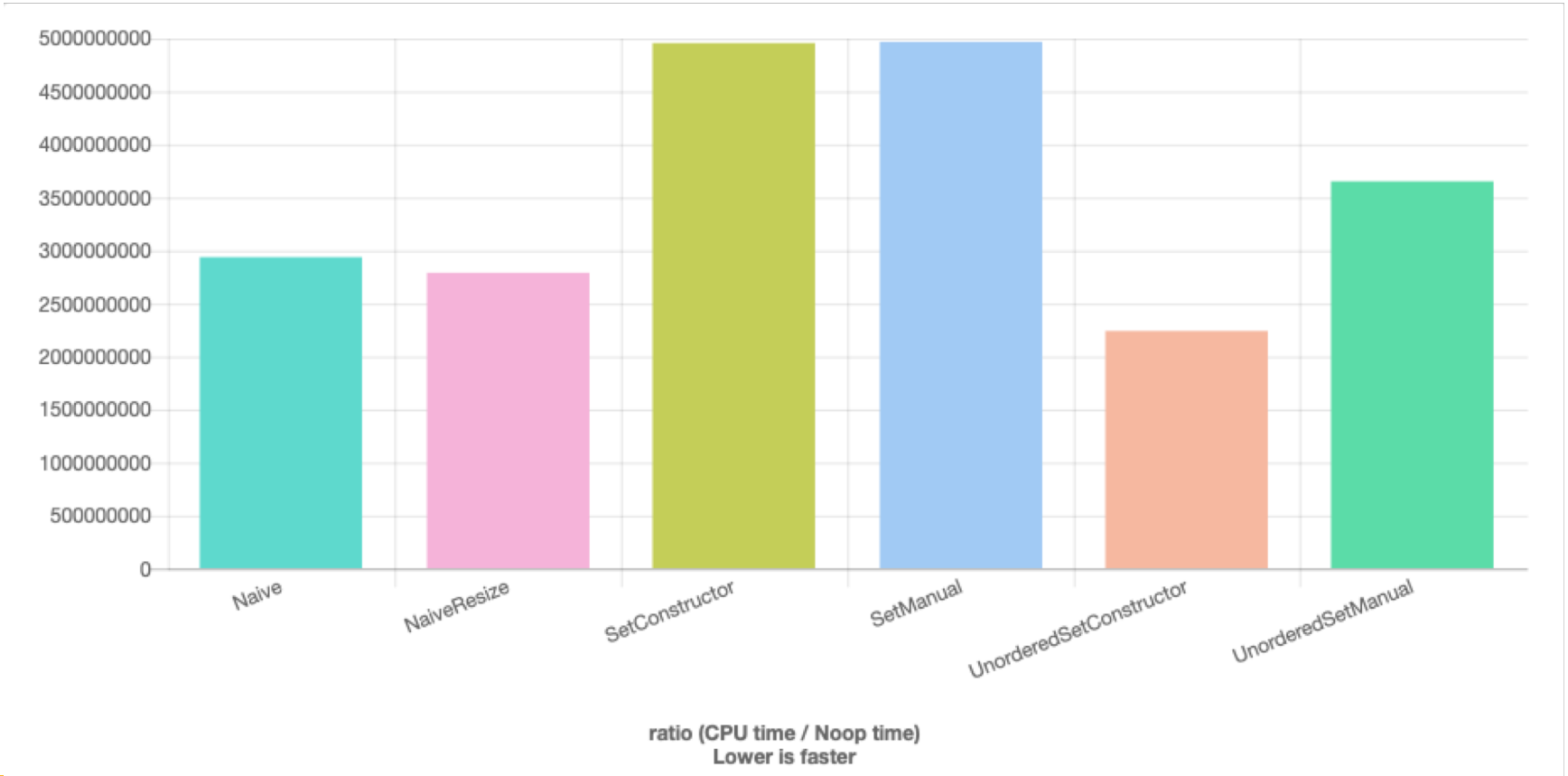
Sorting and Enforcing Uniqueness

String length 10 characters



Sorting and Enforcing Uniqueness

String length 30 characters



DATABASE BENCHMARKING

- **Benchmark**
  - *Executing standardized tests against a system to assess its (relative) performance.*
- **DBMS Benchmark** define
  - Data schemas and their content
  - Query templates and their instances
  - Additional rules
- **Motivation**
  - Comparison of hardware, software, and their interplay
- Who standardizes benchmarks?
  - **T**ransaction **P**rocessing Performance **C**ouncil

http://www.tpc.org/tpch/

TPC-H BENCHMARK

The TPC Benchmark™H (TPC-H) is a decision support benchmark. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

http://www.tpc.org/tpch/

TPC-H BENCHMARK

The TPC Benchmark™H (TPC-H) is a decision support benchmark. The queries and the data populating the database have been chosen to have broad industry-wide relevance. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions.

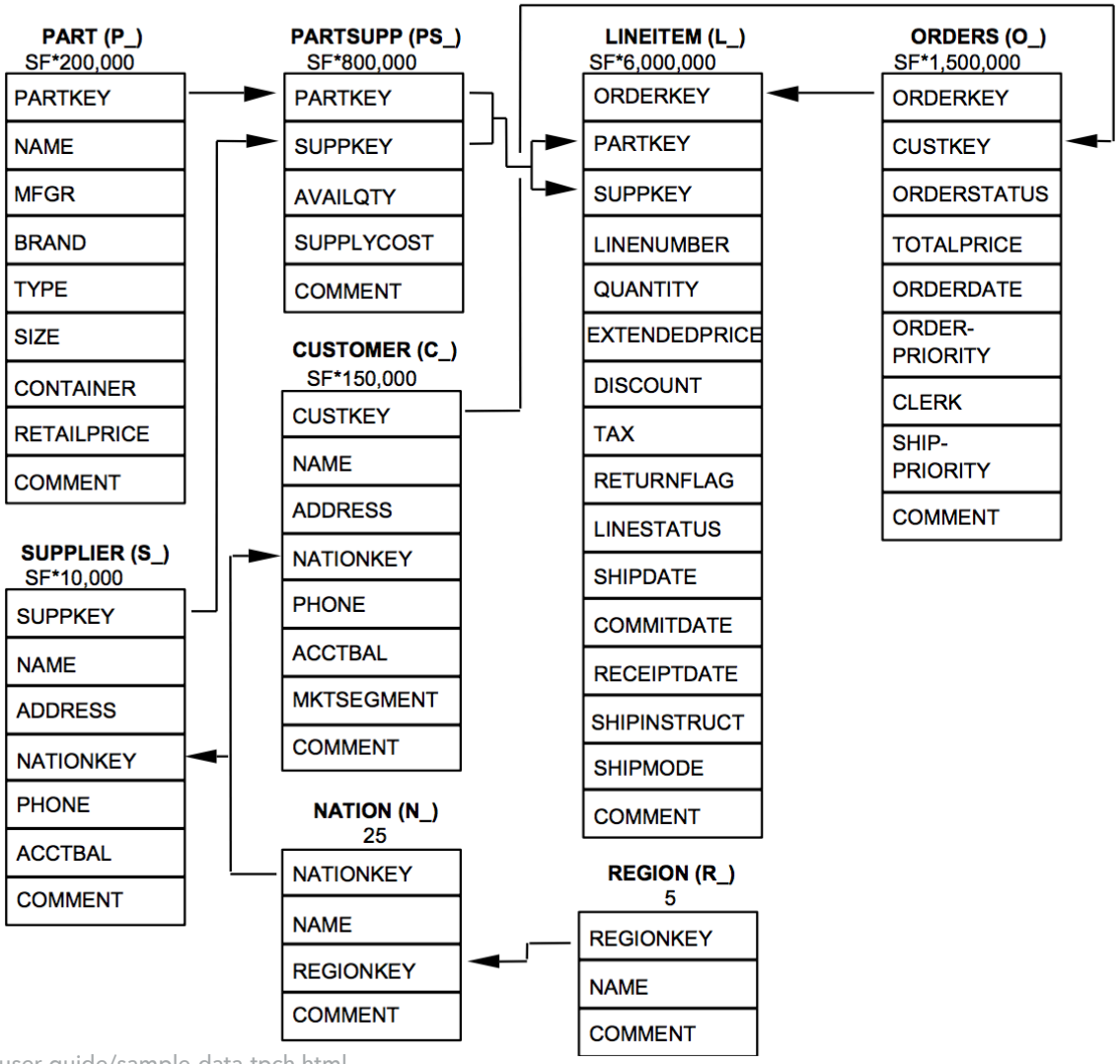
8 Tables containing generated business data

22 Analytical ad-hoc queries

X

http://www.tpc.org/tpch/

TPC-H BENCHMARK - DATA SCHEMA



https://docs.snowflake.net/manuals/user-guide/sample-data-tpch.html

TPC-H BENCHMARK - EXAMPLE QUERY

Business Question

The Order Priority Checking Query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

SQL Query

```
SELECT
   o_orderpriority,
   count(*) as order_count
FROM orders
WHERE
   o_orderdate >= date '[DATE]'
   AND o_orderdate < date '[DATE]' + interval '3' month
   AND exists (
       SELECT *
       FROM lineitem
       WHERE
           l_orderkey = o_orderkey
           AND l_commitdate < l_receiptdate
       )
GROUP BY o_orderpriority
ORDER BY o_orderpriority;
```

GROUP TOPICS

- Primary Keys and Related Optimizations (JK)
- Between Optimizations (JK)
- Multi-predicate Joins (JK)
- Heavy-weight Compression (MB)
- Rewriting Subselects to Joins (MB)
- Indexes for Scans and Joins in TPC-H (MB)
- Rewrite the Aggregate Operator (MD)
- MVCC Physical Delete (MD)
- Sorted Segments (SK)
- More Optimization Rules (SK)

PRIMARY KEYS AND RELATED OPTIMIZATIONS

- **Primary Keys:** minimal set of attributes that uniquely identify a tuple
  - DBMSs ensure uniqueness and hold an index on these attributes
- **Motivation**
  - Primary keys are specified in many real-world and benchmark data schemas
  - This information can be used for more efficient query processing
- **Tasks**
  - Implementation of a primary key entity
  - Use the available information in the query optimizer
- **Evaluation**
  - Investigate applicability and impact for the TPC-H Benchmark

BETWEEN OPTIMIZATIONS

- **Motivation**
  - `WHERE l_quantity >= 530 AND l_quantity <= 530 + 10`
    - `l_quantity BETWEEN 530 AND 530 + 10`
  - `WHERE p_name like 'RED%'`
    - `WHERE p_name >= 'RED' AND p_name < 'REE'`
- **Tasks**
  - Identify beneficial cases
  - Implement optimizer rules to achieve presented rewriting
- **Evaluation**
  - Micro- and TPC-H Benchmark

MULTI-PREDICATE JOINS

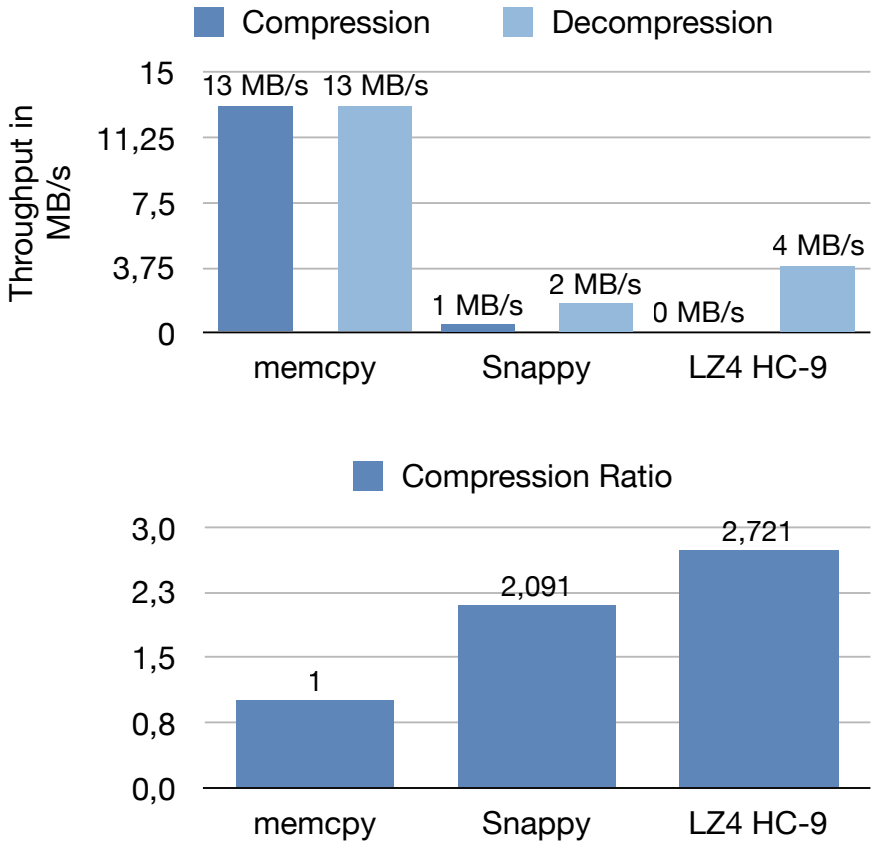
```
SELECT
   nation,
   o_year,
   sum(amount) as sum_profit
FROM (
   SELECT
       n_name as nation,
       extract(year FROM o_orderdate) as o_year,
       l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity
                          as amount
   FROM part, supplier, lineitem, partsupp, orders, nation
   WHERE
       s_suppkey = l_suppkey
       AND ps_suppkey = l_suppkey
       AND ps_partkey = l_partkey
       AND p_partkey = l_partkey
       AND o_orderkey = l_orderkey
       AND s_nationkey = n_nationkey
       AND p_name like '%[COLOR]%'
   ) as profit
GROUP BY nation, o_year
ORDER BY nation, o_year DESC;
```

MULTI-PREDICATE JOINS

- **Motivation**
  - Benchmark and real-world queries contain joins on multiple attributes
  - By avoiding the execution of two separate joins, redundant work can be minimized
- **Tasks**
  - Detect joins on multiple attributes during optimization
  - Implement a specialized join operator for such cases
- **Evaluation**
  - Demonstrate the impact on affected TPC-H Benchmark queries

HEAVY-WEIGHT COMPRESSION

- Currently, Hyrise includes various encodings to compress data
  - Dictionary, run-length, frame-of-refence encoding
- For data rarely accessed, we'd like to study the impact of heavy-weight compression
  - Widely used libraries in databases are *Snappy* or *LZ4*



HEAVY-WEIGHT COMPRESSION

- **Tasks**
  - Integrate heavy-weight compression into Hyrise's encoding framework
  - Evaluate compression levels and further optimizations *
- **Evaluation**
  - Runtime impact on typical DB operators
  - Compression ratios for TPC-H and other data sets

* Similar to GZIP and Co., most libraries provide different compression levels with varying compression time/ratios. Furthermore, Zstd allows to provide a dictionary of often occurring items to further improve compression ratios.

TRANSFORM SUBSELECTS TO JOINS

```
SELECT p_brand, p_type, p_size,
       count(distinct ps_suppkey) as supplier_cnt
FROM partsupp, part
WHERE
   p_partkey = ps_partkey
   AND p_brand <> '[BRAND]'
   AND p_type not like '[TYPE]%'
   AND p_size in ([SIZE1], [SIZE2], [SIZE3], [SIZE4],
                  [SIZE5], [SIZE6], [SIZE7], [SIZE8])
   AND ps_suppkey not in (
       SELECT s_suppkey
       FROM supplier
       WHERE s_comment like '%Customer%Complaints%'
   )
GROUP BY p_brand, p_type, p_size
ORDER BY supplier_cnt DESC, p_brand, p_type, p_size;
```

TRANSFORM SUBSELECTS TO JOINS

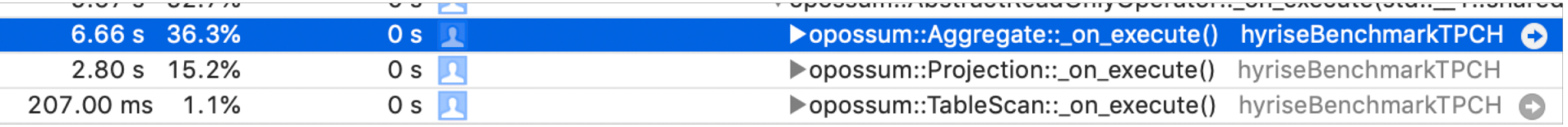
- **Motivation**
  - Many real-world queries can be optimized
  - Especially for analytical queries, many constructs can be reformulated to more efficient join variants
- **Tasks**
  - Implement optimizer rules to recognize potential reformulation candidates (i.e., IN, EXISTS, and more) and adapt the query plan accordingly
  - Develop simple cost models to estimate whether a reformulation will be beneficial
- **Evaluation**
  - Show runtime impact for analytical TPC-H queries as well as selected transactional queries

INDEX JOIN OPTIMIZATIONS

- **Motivation**
  - For transactional workloads, secondary indexes remain indispensable.
  - Besides index scans, indexes can also be exploited in joins (so-called *index joins* or *lookup joins*).
  - However, Hyrise's architecture with the freedom to add indexes only to a subset of chunks makes the decision when to use index joins less straightforward.
- **Tasks**
  - Create simple cost models to estimate the costs of standard joins (e.g., a hash join) and index joins.
  - Extend the optimizer to use index joins whenever beneficial.
  - Improve the current fallback and use appropriate joins for non-indexed chunks.
- **Evaluation**
  - Show runtime impact for analytical TPC-H queries as well as selected transactional queries.

Rewrite the Aggregate Operator

```
SELECT c_custkey, COUNT(*) … GROUP BY c_custkey
```



- The Aggregate operator was already optimized, but is still quite slow
- It‘s architecture makes it a bad fit for cases where a single column is in the GROUP BY clause

- For some queries, a sort-based implementation of the Aggregate would keep us from having to manually sort results later

- Selling point: We already have working tests and a performance baseline to compare to

MVCC Physical Delete

- Updating data traditionally requires locking the row for the remainder of the transaction
- Like most modern databases, we use Insert-Only and only invalidate updated rows

  - Each query executes a *Validate* operator, which behaves similar to a table scan and checks if a row is valid
  - Without physically deleting rows, we end up with a table that is difficult to process
  - Goal: Remove definitely unreachable rows without violating the ACID criteria

Sorted Segments

- Motivation
  - The knowledge of sorted segments can be used to speed up database operators,
  - e.g., scan, join, aggregations, ...
  -
- Task
  - Implement (meta information for) sorted segments
  - Use sorting information in: scans, … (joins?)

- Evaluation
  - TPC-H, e.g., Q6
  - Own queries

More Optimizer Rules

- Motivation
  - "Query optimization is absolutly [sic] essential for virtually any database system that
  - has to cope with reasonably complex queries. As such, it always pays off to invest time
  - in the optimizer. Often, the impact of the query optimizer is much larger than
  - the impact of the runtime system!"
    - Thomas Neumann. Engineering High-Performance Database Engines. VLDB 2014.
  -
- Task
  - Limit operator (+ push down)
  - Choose good join algorithm
  - Order for scans/joins/aggregates with sorted segments

- Evaluation
  - TPC-H (where appropriate)
  - Own queries

Next steps

- Please send us a list of **all topics that you are interested in** until Sunday, 25 November, 23:59pm CET.
- All choices have the same priority and you can submit as many choices as you want.
- If you have questions send an email to:
  - Martin Boissier, Markus Dreseler, Stefan Klauck, Jan Kossmann

Query Processing

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Lecture Motivation

Modern database machines are increasingly large NUMA systems and process complex queries on huge data sets.

How does query processing in modern databases work and incorporate hardware developments?

Overview

i. Query Optimization

ii. Query Scheduling

iii. Query Execution

   i. Joining

   ii. Radix-Partitioned Hash Join

Motivation

How does a database actually process incoming SQL queries?

How does a database process queries?

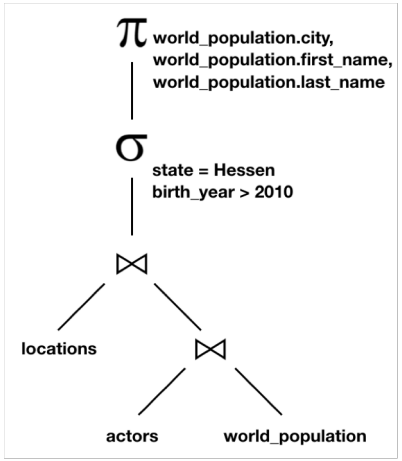
1. The database receives the SQL queries on the network interface and passes it to the SQL parser.

```
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?

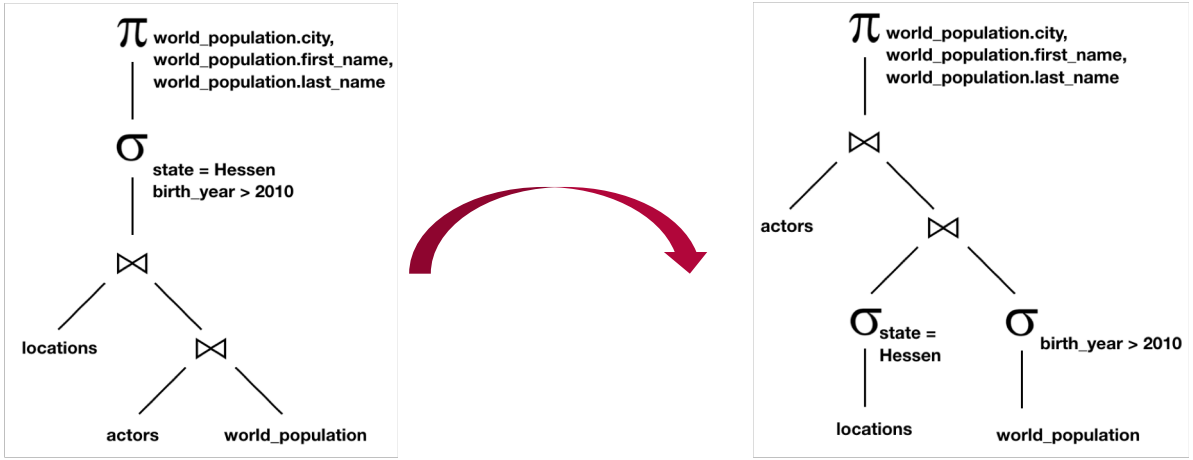
2. The SQL parser generates a logical query plan. This plan contains the **relational operators** required to execute the query and the order in which they have to be called.

```
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
```



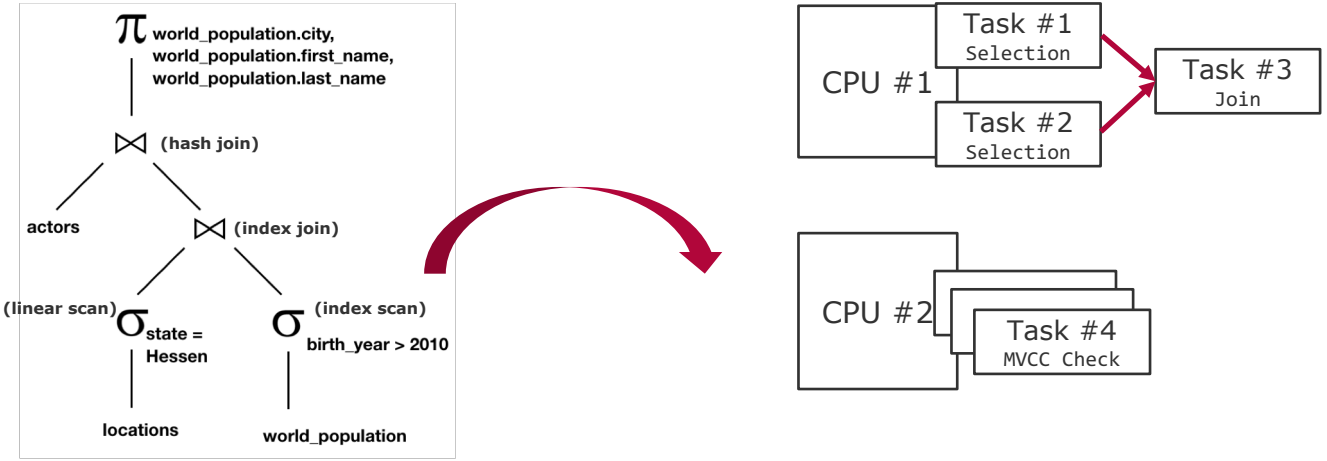
How does a database process queries?

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?

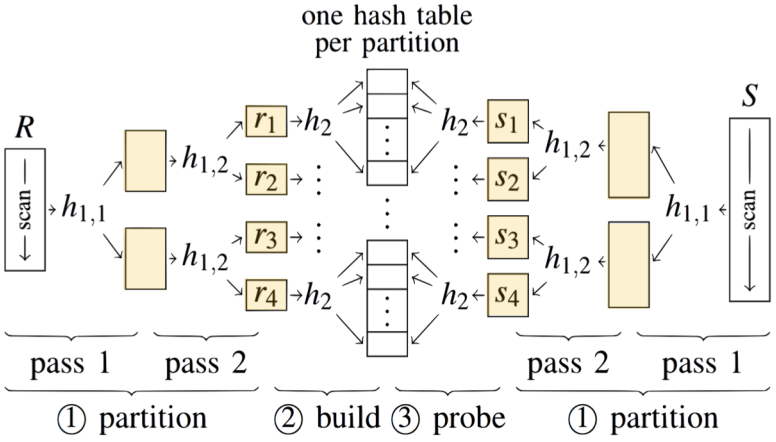
4. After a logical query plan is decided upon, the relational operators are translated to their actual implementations. Further, the **database scheduler** can determine where & when to run the query and how much resources to allocate.



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.



HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

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
```

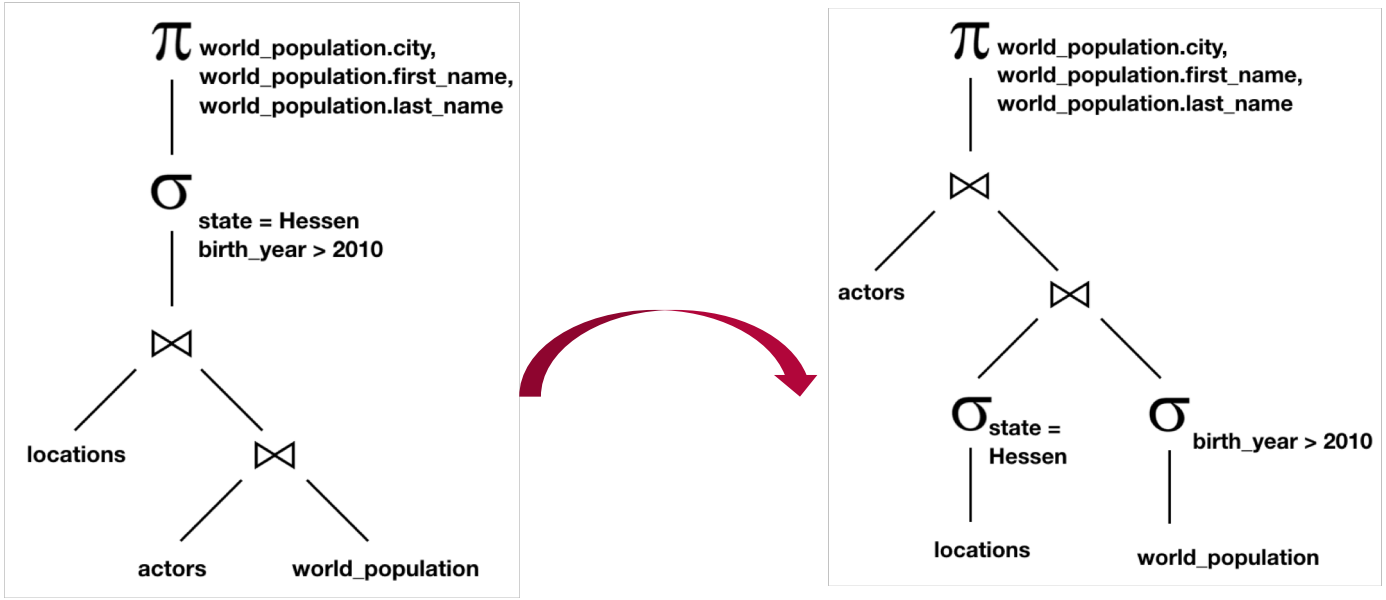
Query Optimization
Semantic Transformations & Heuristics

- 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.

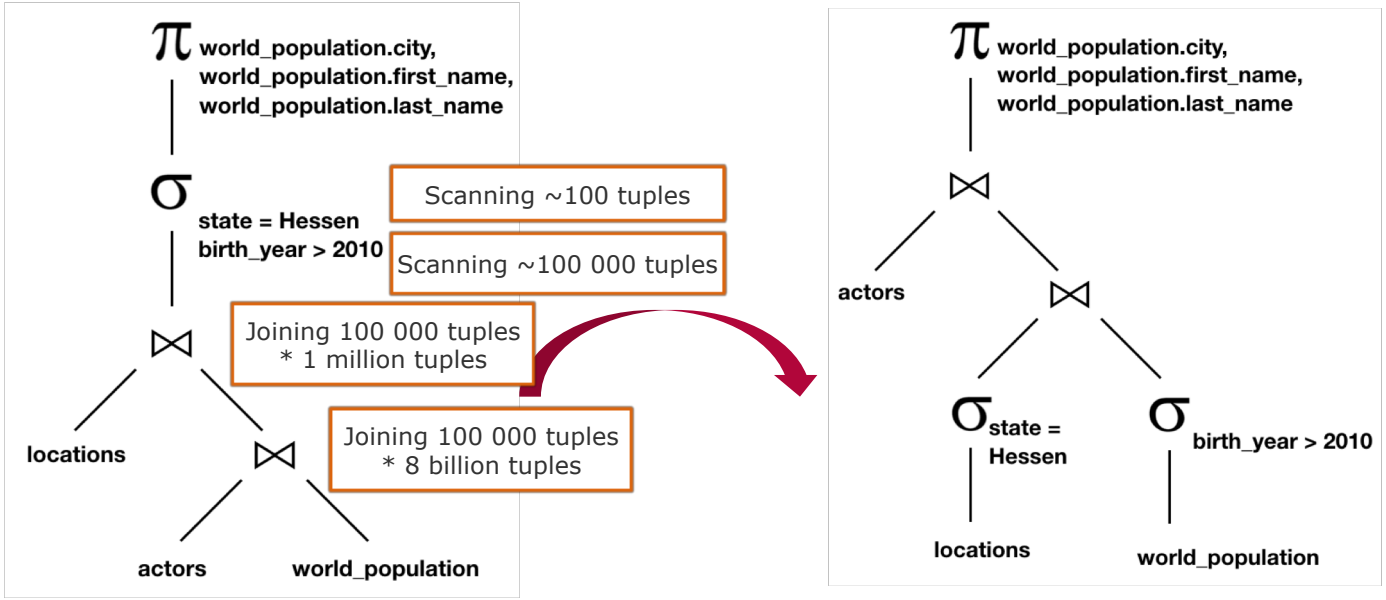
Query Optimization
Query Plan Reformulation

- 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)



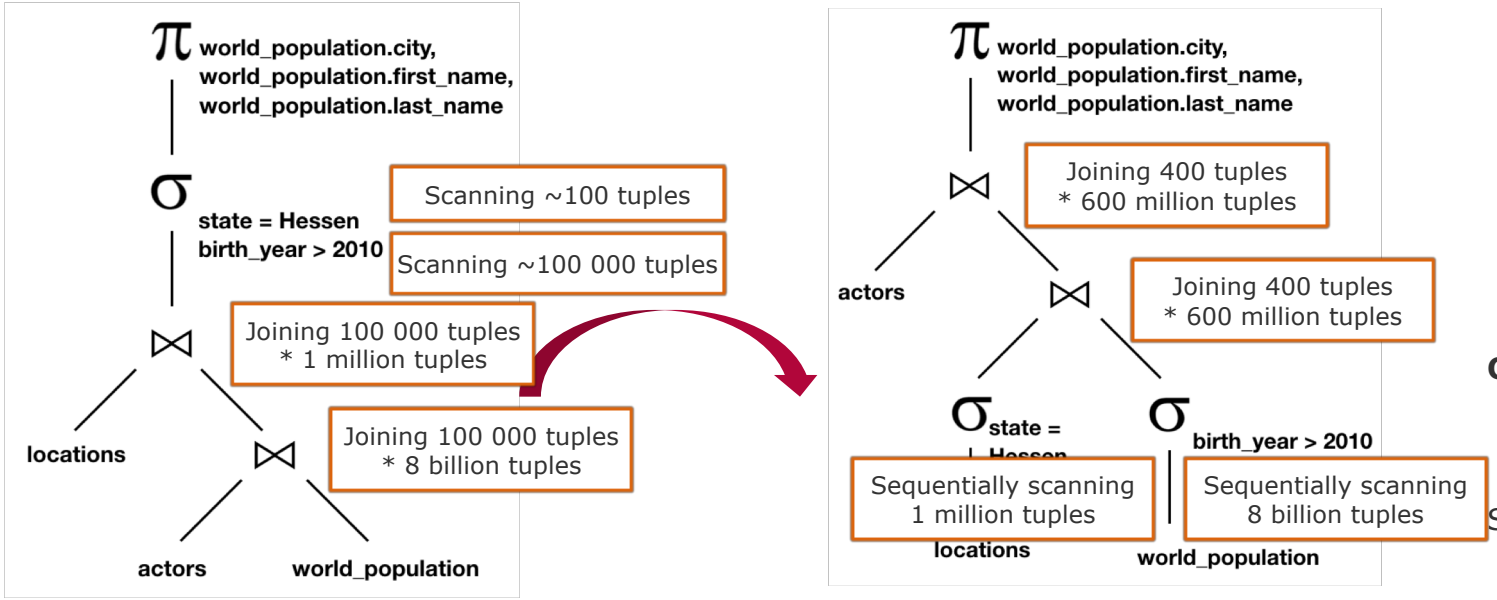
Query Optimization
Query Plan Reformulation

- 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)



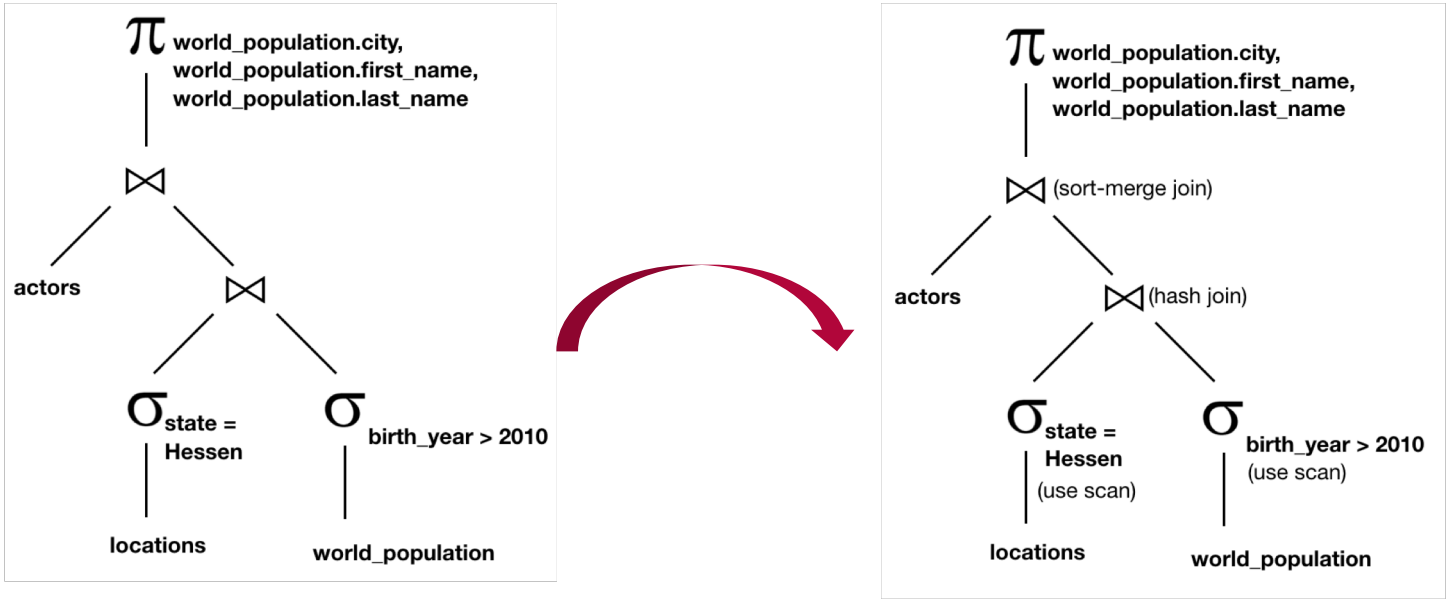
Query Optimization
Query Plan Reformulation

- 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

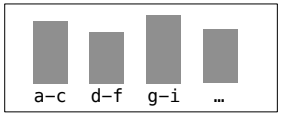
Query Optimization
Statistics

- 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

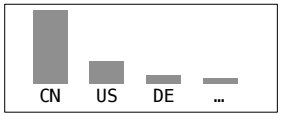
Table: `world_population`

Meta Data

```
Attributes: {'first_name': char(50), 'last_name' […]}
Indexed Columns: {'first_name', 'last_name', […]}
…
Statistics:
    min/max: {'birth_year': ['1900', '2017'], […]}
    distinct_counts: {'birth_year': 118, […]}
    histograms:
        first_name:
```



country:



Data

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

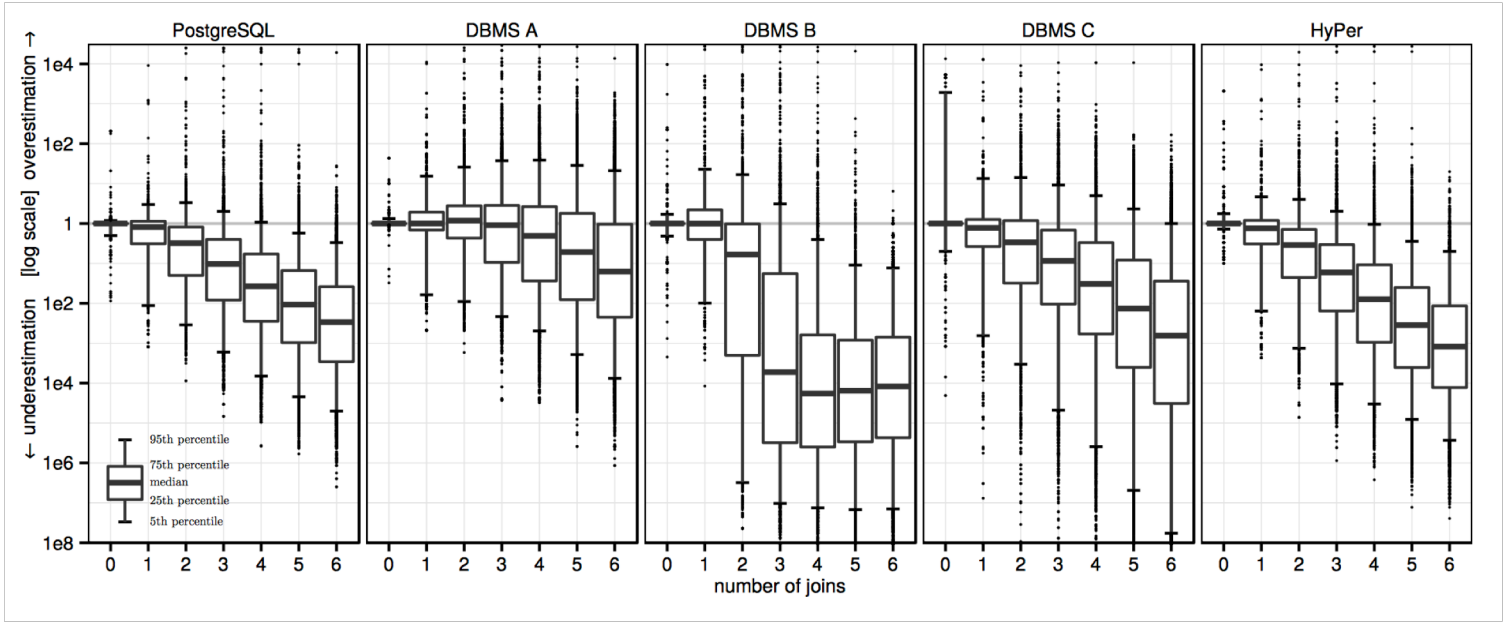
If $r2 \bowtie r3$ is quite large and $r1 \bowtie r2$ is small, we choose

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

so that we compute and store a smaller temporary relation.

Query Optimization
Join Ordering

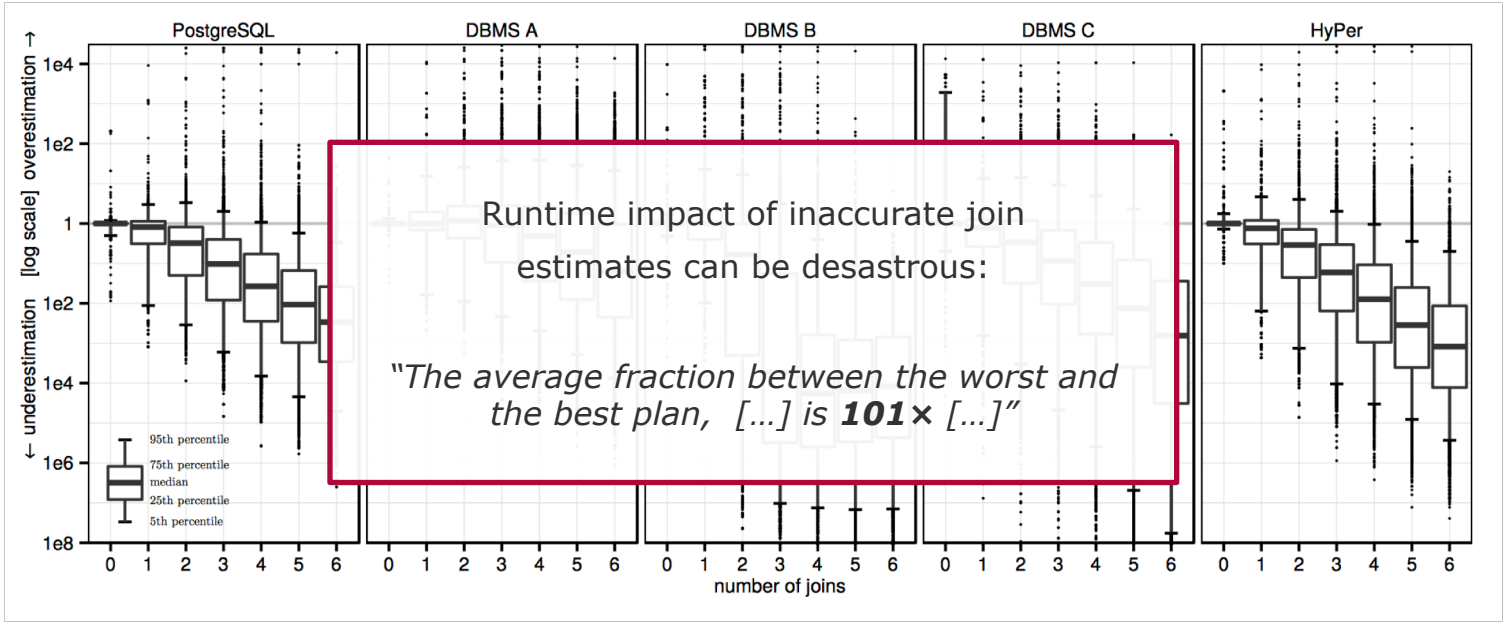
Estimating join cardinalities is one of the challenging tasks of query optimization, but also indispensable to performance.



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.



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.

Query Scheduling

Query Scheduling Overview

- Modern mixed workload systems handle *tens of thousands of queries per second* on servers with dozens of CPU cores
  - But plain concurrent execution can significantly hurt performance
  - The database needs to balance the overall system's *throughput* vs. *latency* of single query execution

- The goal is to spawn the right amount of parallel work given the particular hardware & workload (hence scheduler can be highly hardware dependent)

Query Scheduling
Scheduling Units

- A physical query plan contains **operators**, each execution is an **operator instance**.
- The execution of an operator instance is divided into *1-n* **tasks**.

- **Workers** execute the tasks. Depending on the database's architecture a worker is …
  - a process, or
  - a thread.

Further, workers can be grouped into process/thread pools.

Query Scheduling
Scheduling Units

- The extend of parallelism varies from database to database
  - One task per query, queries are executed concurrently: so-called ***inter-query parallelism***
  - One task per operator, where operators that do not depend on each other are executed concurrently: so-called ***intra-query parallelism***
  - Multiple tasks per operator, where the execution of an operator is split into concurrent tasks: so-called ***intra-operator parallelism***

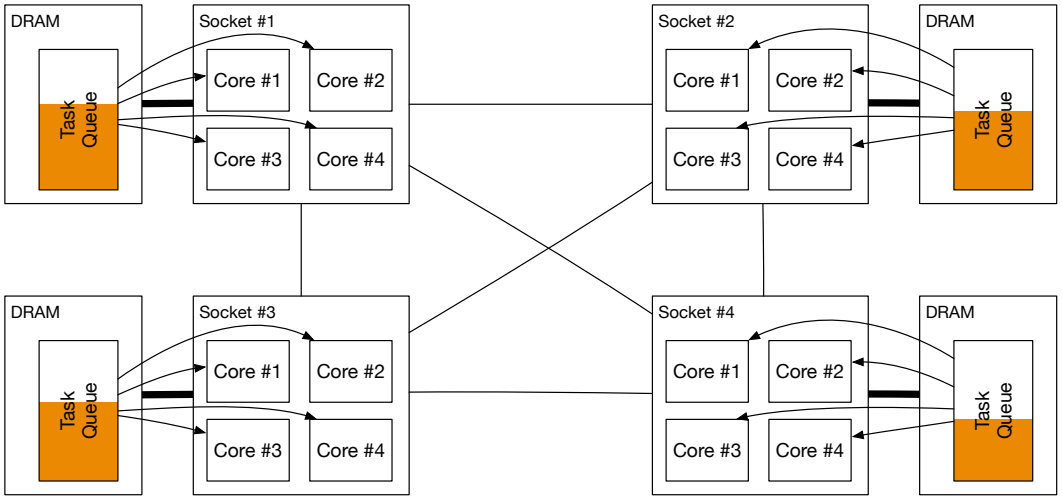
- With the rise of many-core systems and mixed workloads, most systems use both intra- and inter-query parallelism.
- Most database systems create fixed-size *threads pools* to limit threading overhead for highly concurrent workloads.

Query Scheduling
Task Placement for NUMA Systems

- For NUMA systems, workers should primarily execute near the data they operate on.
- Most NUMA-optimized databases spawn a *worker thread pool* per socket.
- To feed the *socket-bound workers*, the database has one or more local task queues.

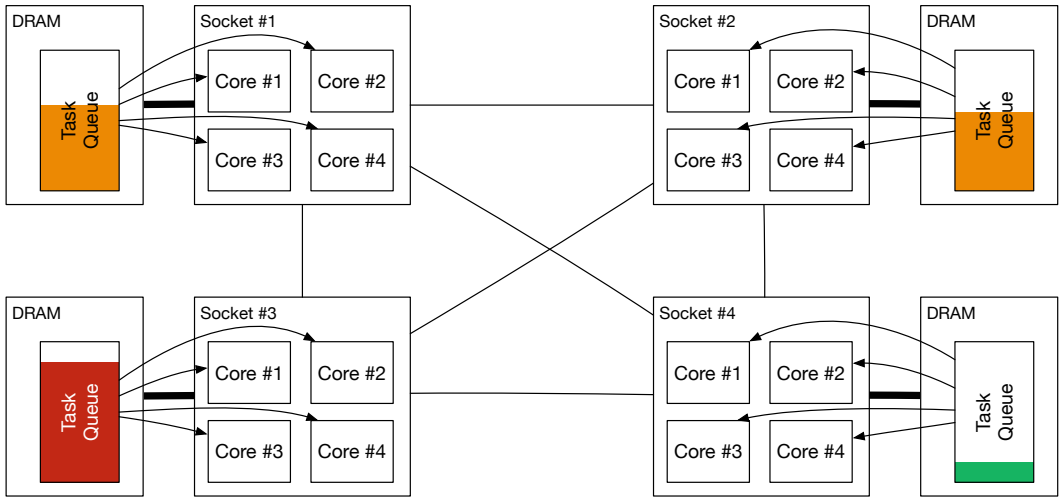
Query Scheduling
Task Placement for NUMA Systems

- Every node has its local task queue holding tasks that primarily work on socket-local data.



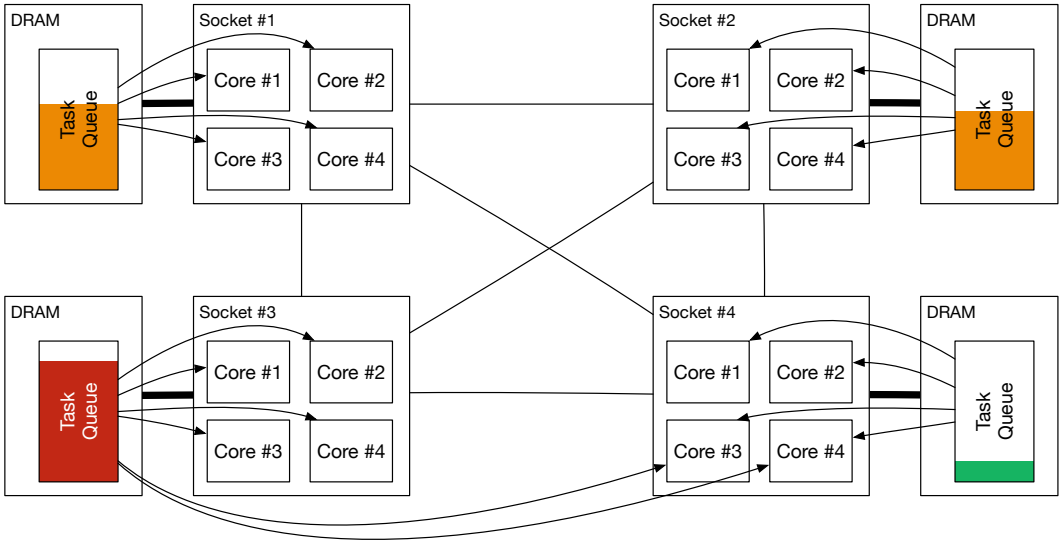
Query Scheduling
Task Placement for NUMA Systems

- In real-world applications, workloads are often highly skewed…



Query Scheduling
Task Placement for NUMA Systems

- If the task queue is empty, workers can overtake work from other worker pools (so-called *task/work stealing*).
- The degree of how much work stealing is allowed depends on node distance, CPU load, QPI saturation, and more.



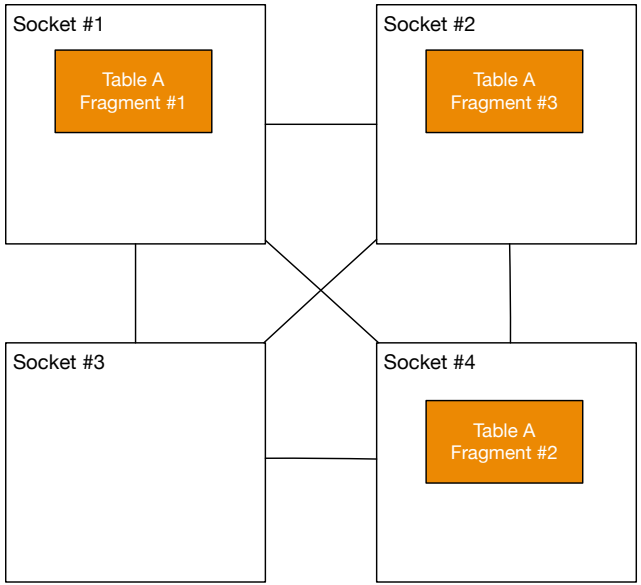
Query Scheduling
Data Placement for NUMA Systems

- For any NUMA-aware system, workers should primarily access data that is local to itself (NUMA-aware data placement)
- Thus, the database engine cannot rely on the OS's data placement scheme (e.g., first-touch or interleaved) but has to distribute data across the NUMA nodes on her own and place tasks accordingly

- Straightforward approach is round-robin chunk placement
  - Advantage: simplicity and automatic handling of workload skew
  - Disadvantage: operations may combine outputs from multiple nodes when correlated tables are scattered (e.g., foreign key relationships)
- Goal is to *distribute data both skew- and workload-aware* in the first place and dynamically *adapt to changing workload patterns*

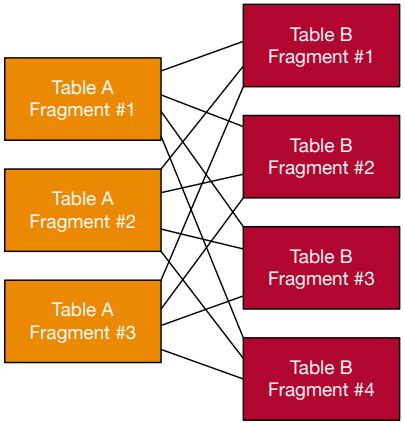
Query Scheduling
Data Placement for NUMA Systems

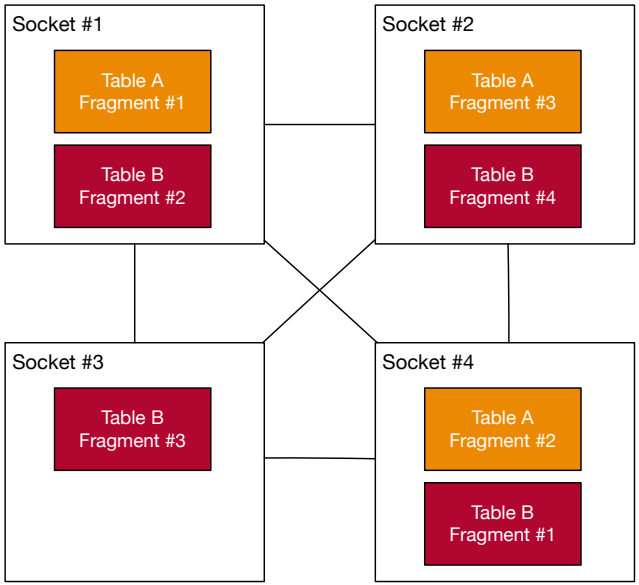
- A scan on `table A` can executed in parallel with optimal data locality.
- An aggregation on `table A` (e.g., `min()`) can first be executed in parallel with optimal data locality, but final result merging accesses remote data.



Query Scheduling
Data Placement for NUMA Systems

- Joining `table A` and `table B` inevitably needs to move data across the QPI. Ideally, regularly together joined tables are co-located.





Query Scheduling
Data Placement in Hyrise[2]

In case of changing workloads, data placement has to be adapted:

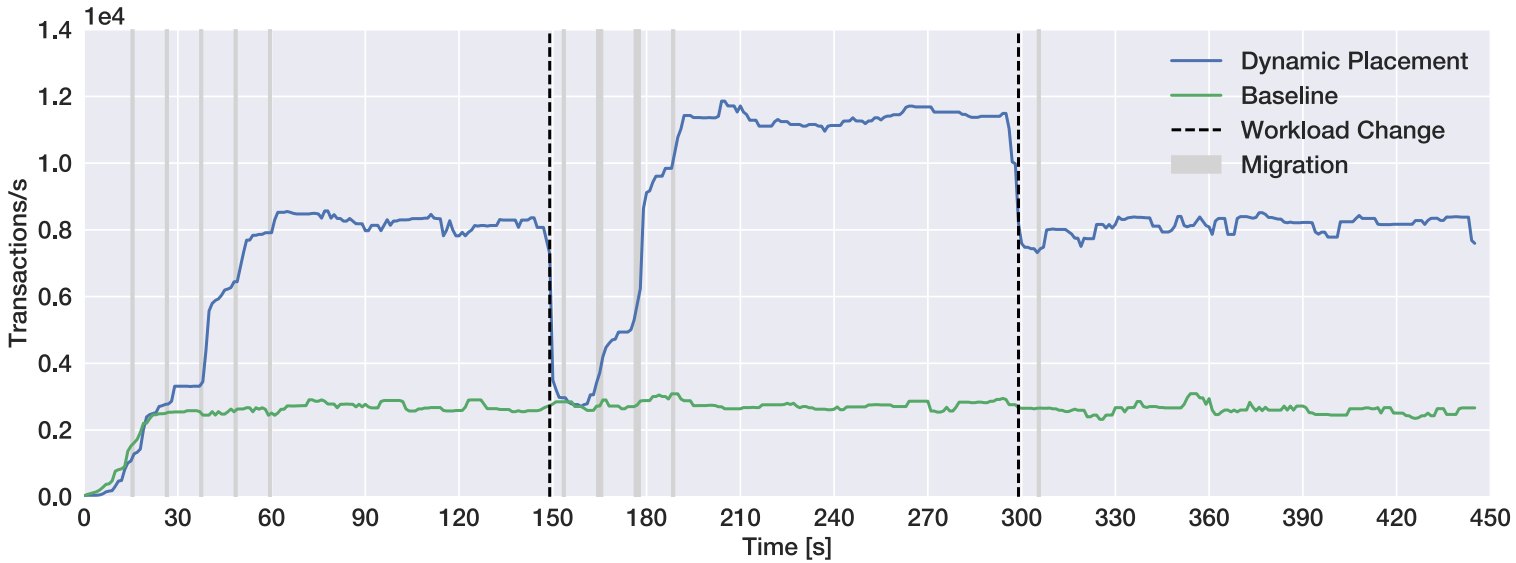


Figure from Norman Rzepka's Master Thesis "Adaptive NUMA-Aware Load-Balancing for In-Memory Databases", 2017

Query Scheduling Summary

We learned that scheduling becomes increasingly important due to …

- balancing between throughput and query latencies
- diverse memory hierarchies (DRAM, NVRAM, NUMA hops)
- mixed workloads with both short queries & long-running complex queries

Overall Summary

Overall Summary

Modern systems execute complex analytical queries

- Optimization remains challenging for complex queries
- Join estimation is an open research problem with huge performance impact

Modern database servers are large NUMA systems

- Expensive and long-running operators need to be parallelized properly
- Effective data placement is crucial
- Scheduling of queries needs to balance throughput and latency

Modern database servers have dozens of CPU cores

- Choice of join is not just dependent on runtime complexity, but also on the join's fit to the database engine, server hardware, and workload