HPI Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

# Build your own Database

Week 6

# Agenda

- Q&A Sprint 3
- Review Sprint 2
- NULL values in SQL
- Joins
- Virtual Methods

# 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`

Hasso Plattner Institut

# 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;
}
```

Hasso Plattner Institut

# 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);
}
```

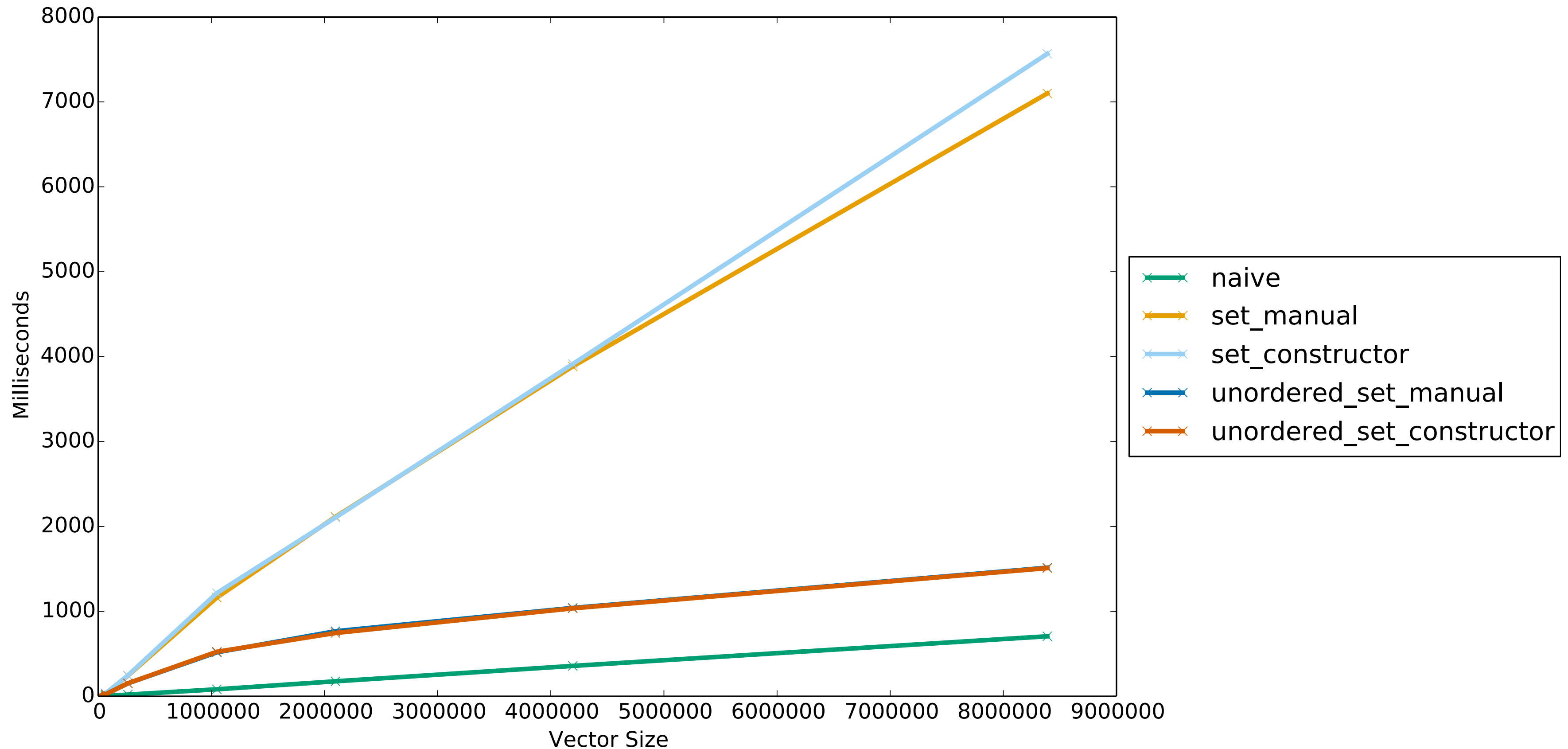
```
const T DictionaryColumn<T>::get(const size_t i) const {
  return _dictionary->at(_attribute_vector->get(i));
}
```

Hasso Plattner Institut

# 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));
}
```

# Sorting and Enforcing Uniqueness



# NULL values in SQL

- `NULL` is used to represent absent values
- It is a state/marker not a value
- DBMS dependent behavior in many cases
  - Arithmetic operations involving `NULL`, usually result in `NULL`: `NULL * 17` → `NULL`, but what about `NULL / 0`?
  - String concatenations involving `NULL`, result in `NULL`

# NULL values in SQL

- `SELECT 17 = NULL` → ?
- SQL offers three logical results: `True`, `False`, `Unknown` → Three-valued Logic (3VL)

| a | b | a AND b | a OR b | a = b |
|---|---|---|---|---|
| True | True | True | True | True |
| True | False | False | True | False |
| True | Unknown | Unknown | True | Unknown |
| False | True | False | True | False |
| False | False | False | False | True |
| False | Unknown | False | Unknown | Unknown |
| Unknown | True | Unknown | True | Unknown |
| Unknown | False | False | Unknown | Unknown |
| Unknown | Unknown | Unknown | Unknown | Unknown |

# NULL values in SQL

**Customer**

| c_id | firstname | lastname |
|---|---|---|
| 0 | NULL | Zayer |
| 1 | Alex | Geier |
| 2 | Frank | Meier |

- `SELECT * FROM customer WHERE firstname = NULL;` → ?
- `SELECT * FROM customer WHERE firstname <> Alex;` → ?
- Rows for which predicates evaluate to `Unknown` are treated like rows that evaluate to `False`
- All standard comparison operators return `Unknown` when comparing `NULL`
- Thus, SQL offers `IS NULL` and `IS NOT NULL`

# NULL values in Opossum

We need a representation for NULL for Value, Reference and Dictionary columns:

```
//Value Columns
static const auto NULL_VALUE = AllTypeVariant{};
// Used to represent NULL in...
constexpr ChunkOffset INVALID_CHUNK_OFFSET{std::numeric_limits<ChunkOffset>::max()};
// Reference Columns
const RowID NULL_ROW_ID = RowID{ChunkID{0u}, INVALID_CHUNK_OFFSET};
// NULL ValueIDs for Dictionary Columns
constexpr ValueID NULL_VALUE_ID{std::numeric_limits<ValueID::base_type>::max()};
```

# VIRTUAL METHODS - INTRODUCTION

- Definition
  - Overridable method, resolved by dynamic dispatch
- Motivation
  - Determine exact implementation at runtime (polymorphism)
- Implementation in C++
  - Standard does not give a definition
  - Major compilers use virtual method tables

# VIRTUAL METHODS - IMPLEMENTATION IN C++

- Compiler adds hidden member variable *vptr at offset 0*
- *vptr* points to a virtual method table
- Virtual method table is an array of function pointers
- Correct function is dispatched at runtime

# VIRTUAL METHODS – EXAMPLE

```
1  class Vehicle {
2        std::string _model;
3    public:
4        virtual int16_t wheels() = 0;
5        virtual std::string get_model() { return _model; }
6  };
7
8  class Car : public Vehicle {
9    public:
10       virtual int16_t wheels() { return 4; }
11 };
12
13 class Motorcycle : public Vehicle {
14   public:
15       virtual int16_t wheels() { return 2; }
16 };
```

# VIRTUAL METHODS – EXAMPLE (II)

```
1 Vtable for Vehicle
2 Vehicle::_ZTV7Vehicle: 4u entries
3 0      (int (*)(...))0
4 8      (int (*)(...))(& _ZTI7Vehicle)
5 16     (int (*)(...))__cxa_pure_virtual
6 24     (int (*)(...))Vehicle::get_model
```

```
1 Vtable for Car
2 Car::_ZTV3Car: 4u entries
3 0      (int (*)(...))0
4 8      (int (*)(...))(& _ZTI3Car)
5 16     (int (*)(...))Car::wheels
6 24     (int (*)(...))Vehicle::get_model
```

```
1 Vtable for Motorcycle
2 Motorcycle::_ZTV10Motorcycle: 4u entries
3 0      (int (*)(...))0
4 8      (int (*)(...))(& _ZTI10Motorcycle)
5 16     (int (*)(...))Motorcycle::wheels
6 24     (int (*)(...))Vehicle::get_model
```

g++ -fdump-class-hierarchy foo.cpp

# VIRTUAL METHODS - OVERHEAD

- Overhead
  - Increased memory consumption
  - Cache pollution
  - Compiler optimization is hard (impossible) because of polymorphism
  - Additional indirection

# VIRTUAL METHODS - OVERHEAD IN OPOSSUM

```
 1 constexpr int64_t rows = 10'000'000;
 2 auto t = Table(0);
 3 t.add_column("col_1", "long");
 4 for (int64_t i = 0; i < rows; ++i) {
 5     t.append({i * 14});
 6 }
 7
 8 auto& chunk = t.get_chunk(0);
 9 std::shared_ptr<BaseColumn> base_column = chunk.get_column(0);
10 auto value_column =
std::dynamic_pointer_cast<ValueColumn<int64_t>>(base_column);
11
12 int64_t sum1 = 0;
13 for (size_t offset = 0; offset < rows; ++offset) {
14   sum1 += different_access_methods;
15 }
```

} Benchmark

# VIRTUAL METHODS - OVERHEAD IN OPOSSUM (II)

- Example: Operator accesses a ValueColumn
  - AllTypeVariant operator[](const size_t i)
  - const std::vector<T>& values()
  - T operator[](const size_t i)
    - On Base- and ValueColumn

# VIRTUAL METHODS – OVERHEAD IN OPOSSUM (III)

# VIRTUAL METHODS - OVERHEAD IN OPOSSUM (IV)

```
650 +0xcfc     nopl      (%rax)
651 +0xd00       movq      (%r15), %rax
652 +0xd03       movq      8(%rax), %rax
653 +0xd07       movq      %r15, %rdi
654 +0xd0a       movq      %rbx, %rsi
655 +0xd0d       callq     *%rax
656 +0xd0f       addq      %rax, %r13
657 +0xd12       addq      $1, %rbx
658 +0xd16       cmpq      $10000000, %rbx
659 +0xd1d       jb        "main+0xd00"
```

T operator[](const size_t i)

```
521 +0xa64     nopw      %cs:(%rax,%rax)
522 +0xa70       vpaddq    -608(%rax,%rcx,8), %ymm0, %ymm0
523 +0xa79       vpaddq    -576(%rax,%rcx,8), %ymm1, %ymm1
524 +0xa82       vpaddq    -544(%rax,%rcx,8), %ymm2, %ymm2
525 +0xa8b       vpaddq    -512(%rax,%rcx,8), %ymm3, %ymm3
526 +0xa94       vpaddq    -480(%rax,%rcx,8), %ymm0, %ymm0
527 +0xa9d       vpaddq    -448(%rax,%rcx,8), %ymm1, %ymm1
528 +0xaa6       vpaddq    -416(%rax,%rcx,8), %ymm2, %ymm2
529 +0xaaf       vpaddq    -384(%rax,%rcx,8), %ymm3, %ymm3
530 +0xab8       vpaddq    -352(%rax,%rcx,8), %ymm0, %ymm0
531 +0xac1       vpaddq    -320(%rax,%rcx,8), %ymm1, %ymm1
532 +0xaca       vpaddq    -288(%rax,%rcx,8), %ymm2, %ymm2
533 +0xad3       vpaddq    -256(%rax,%rcx,8), %ymm3, %ymm3
534 +0xadc       vpaddq    -224(%rax,%rcx,8), %ymm0, %ymm0
535 +0xae5       vpaddq    -192(%rax,%rcx,8), %ymm1, %ymm1
536 +0xaee       vpaddq    -160(%rax,%rcx,8), %ymm2, %ymm2
537 +0xaf7       vpaddq    -128(%rax,%rcx,8), %ymm3, %ymm3
538 +0xafd       vpaddq    -96(%rax,%rcx,8), %ymm0, %ymm0
539 +0xb03       vpaddq    -64(%rax,%rcx,8), %ymm1, %ymm1
540 +0xb09       vpaddq    -32(%rax,%rcx,8), %ymm2, %ymm2
541 +0xb0f       vpaddq    (%rax,%rcx,8), %ymm3, %ymm3
542 +0xb14       addq      $80, %rcx
543 +0xb18       cmpq      $10000076, %rcx
544 +0xb1f       jne       "main+0xa70"
```

const std::vector<T>& values()

# VIRTUAL METHODS - FINAL AND OVERRIDE

- Used in function declarations or definitions
- Ensures function to be virtual
- *Final* functions may not be overridden by derived classes
- Functions declared *override* need to override a virtual function

```
AllTypeVariant operator[](const size_t i) const final
```

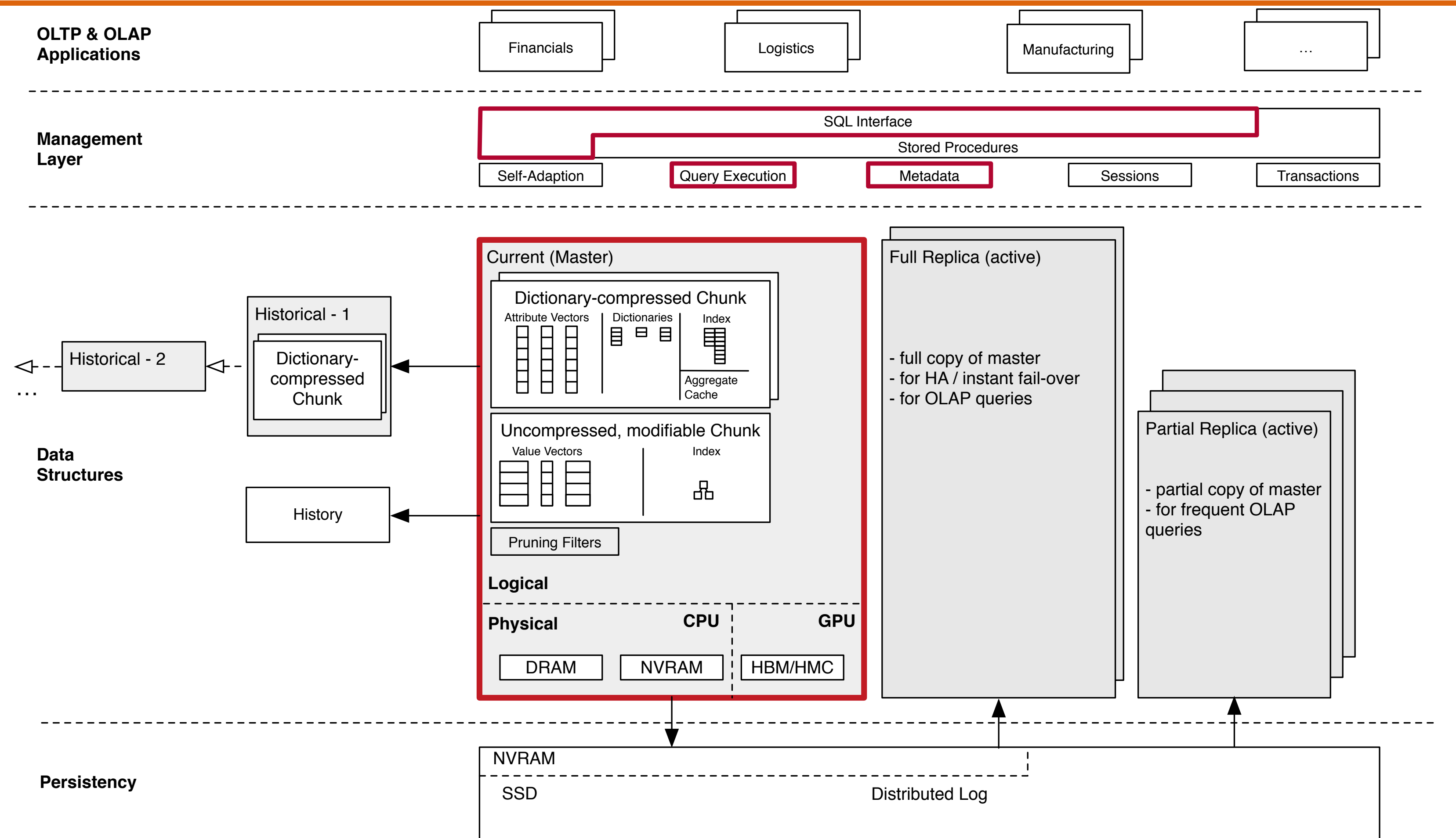
# VIRTUAL METHODS – OVERHEAD IN OPOSSUM (V)

HPI Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

# Query Processing

# Query Processing



# 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.

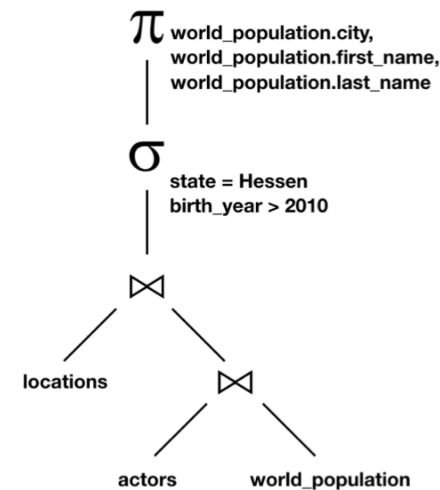
```
1  SELECT wp.city , wp.first_name, wp.last_name
2  FROM world_population AS wp
3  INNER JOIN locations ON wp.city = locations.city
4  WHERE locations.state = ’Hessen’ AND wp.birth_year > 2010
5  INNER JOIN actors ON actors.first_name = wp.first_name
6  AND actors.last_name = wp.last_name
```

# How does a database process queries?

SQL Parsing → Plan Building → Optimization → Scheduling → Execution

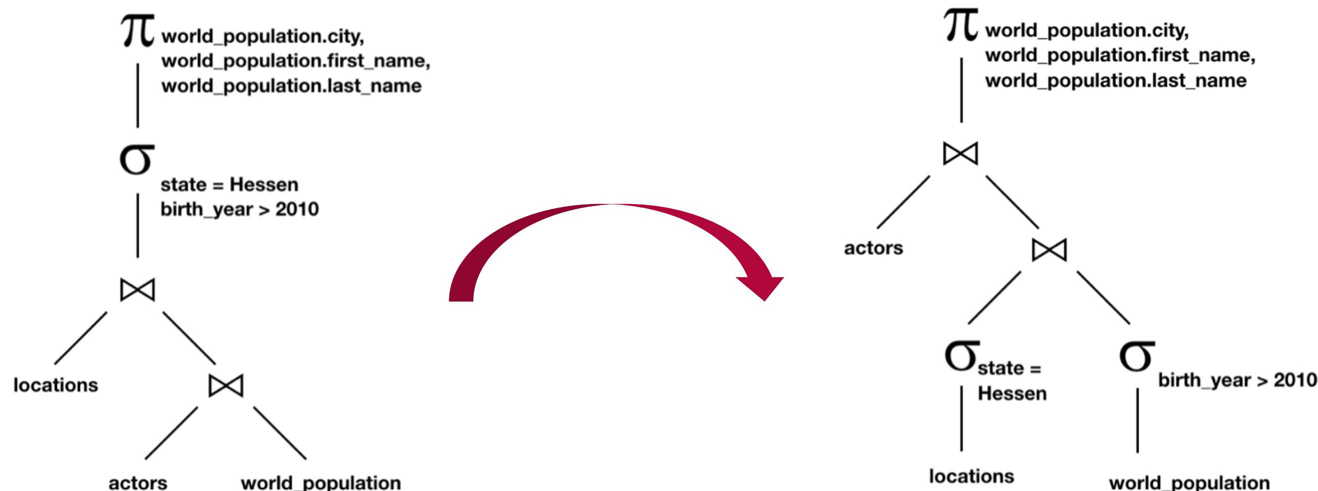
2. The SQL parser generates a logical query plan. This plan contains the **relational operators** required to execute the query and the order in which they have to be called.

```
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?

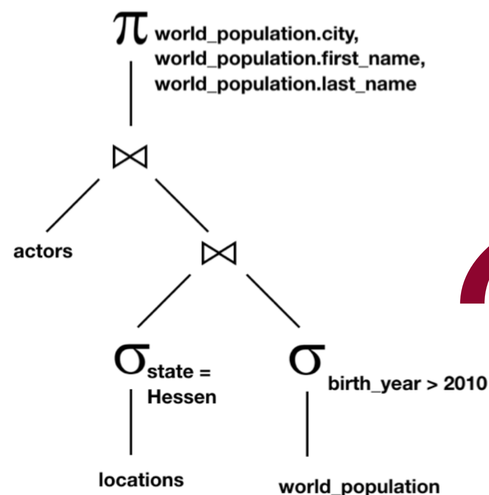
3. Depending on the order of operations in the query plan, runtimes can differ by orders of magnitude. Thus, the database employs the **query optimizer** to determine efficient query plans.

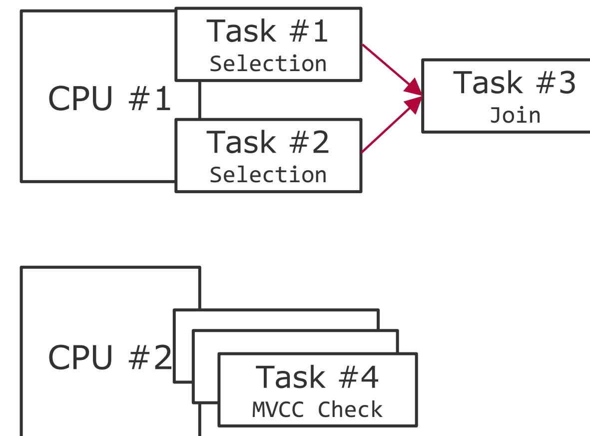


# How does a database process queries?

Hasso Plattner Institut

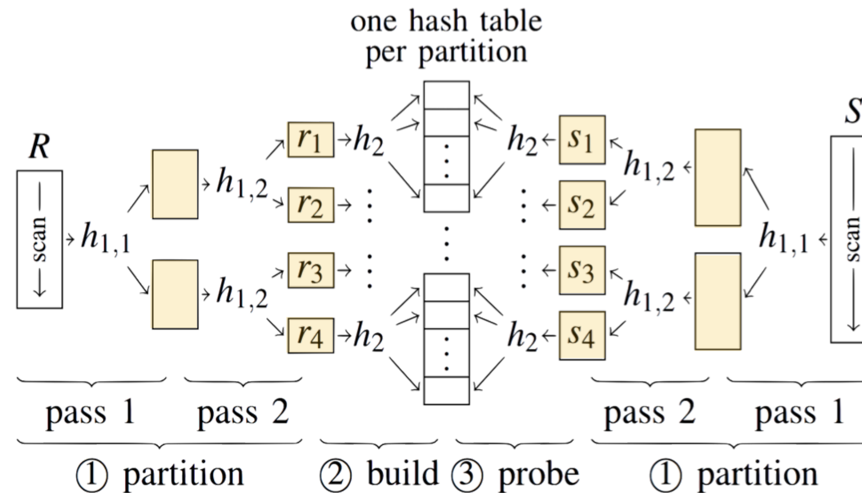
4. After a query plan is decided upon, the **database scheduler** determines where & when to run the query and how much resources to allocate.





## How does a database process queries?

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




Query Processing

# Query Optimization

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

# 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

HPI Hasso Plattner Institut

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

Hasso Plattner Institut

```
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
```

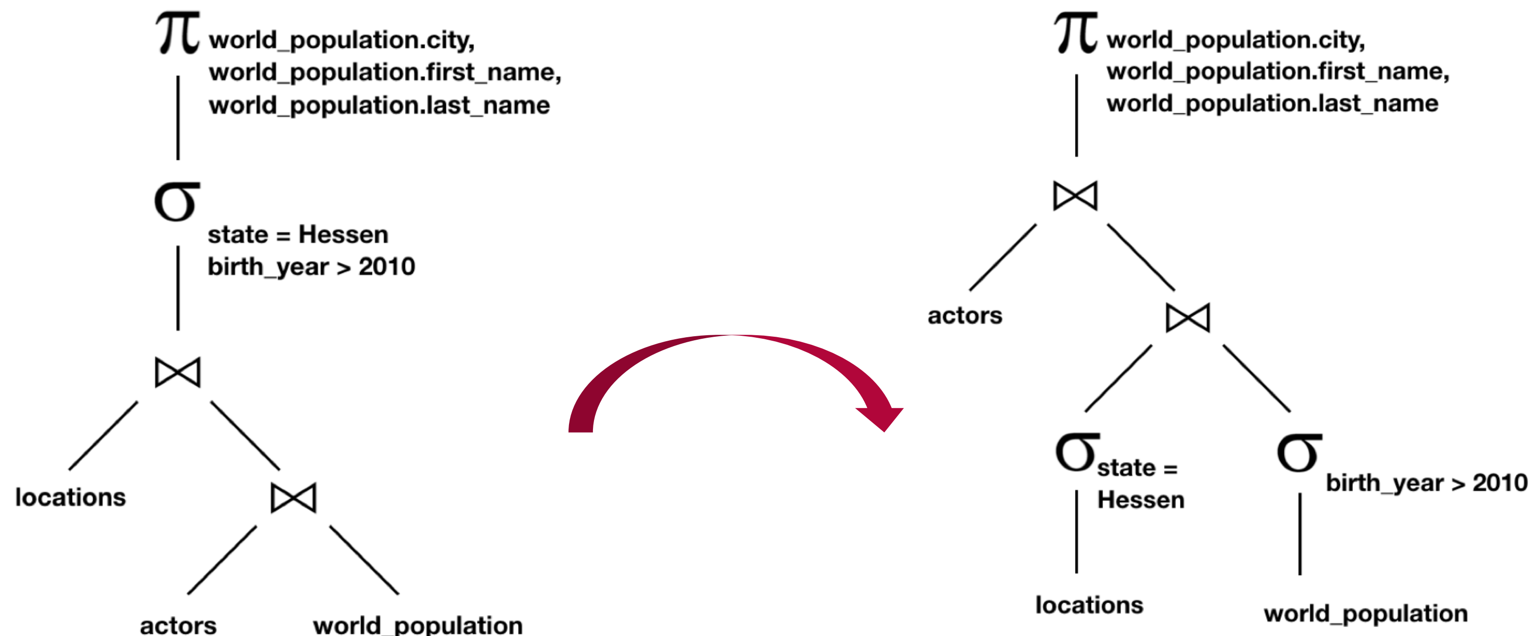
# Query Optimization
# Semantic Transformations & Heuristics

- 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.

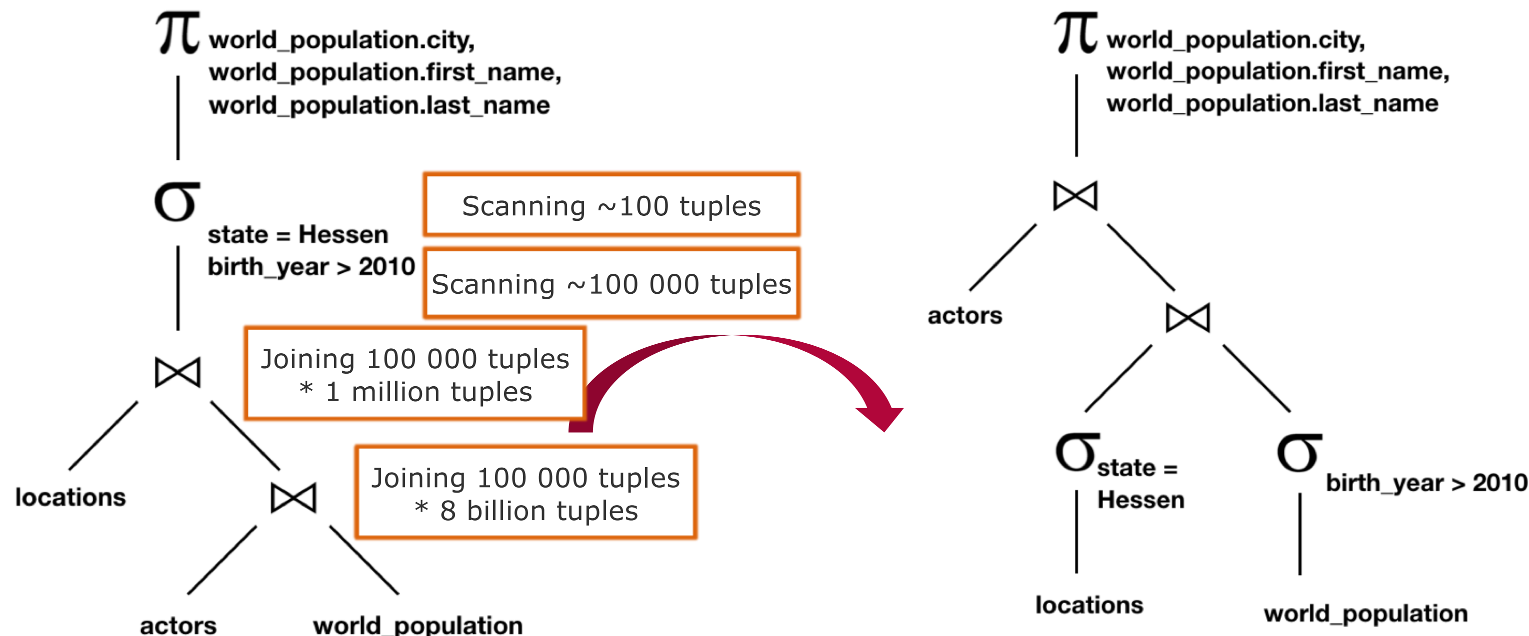
# Query Optimization
# Query Plan Reformulation

- 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)



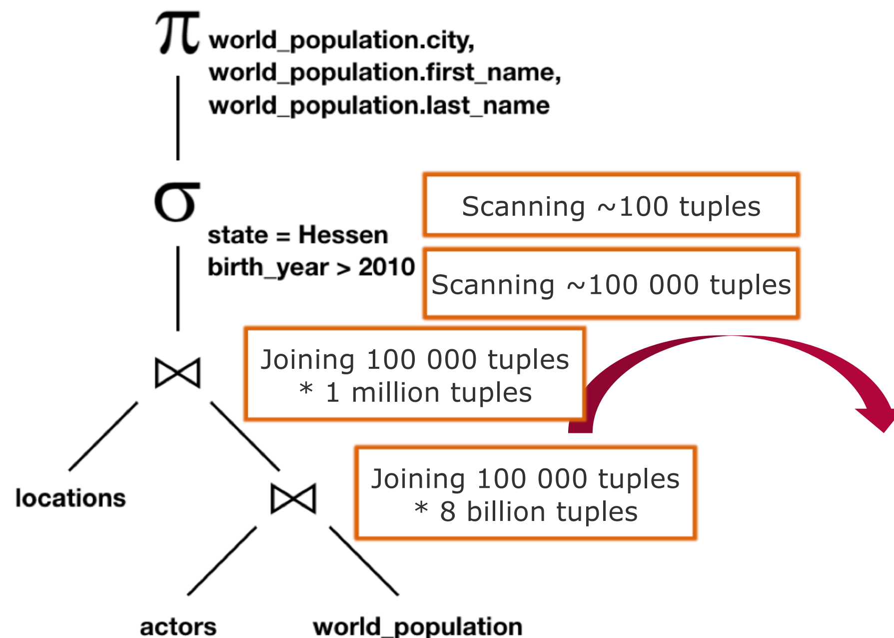
# Query Optimization
# Query Plan Reformulation

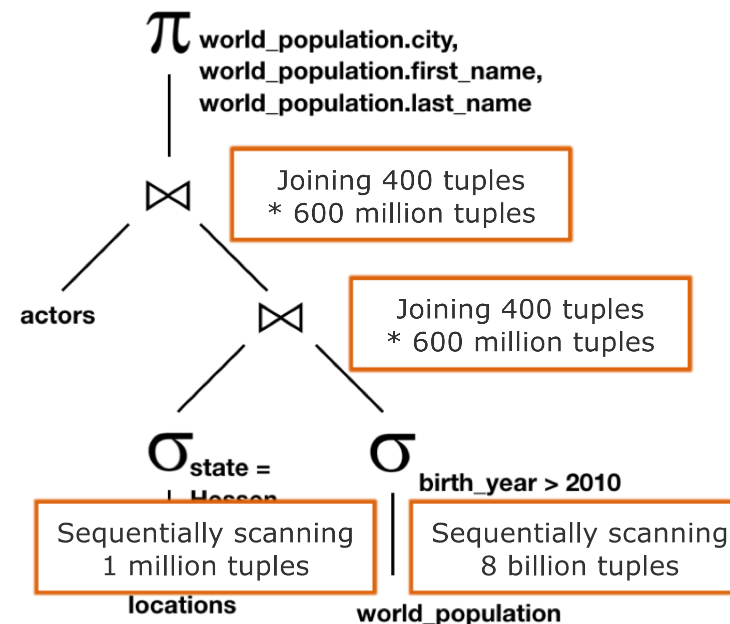
- Logical Query Plan can be seen as a tree of relational algebra operators
- Enumeration phase generates logically equivalent expressions using equivalence rules (i.e., operators can only be reordered to an extend that ensures correct results)



# Query Optimization
# Query Plan Reformulation

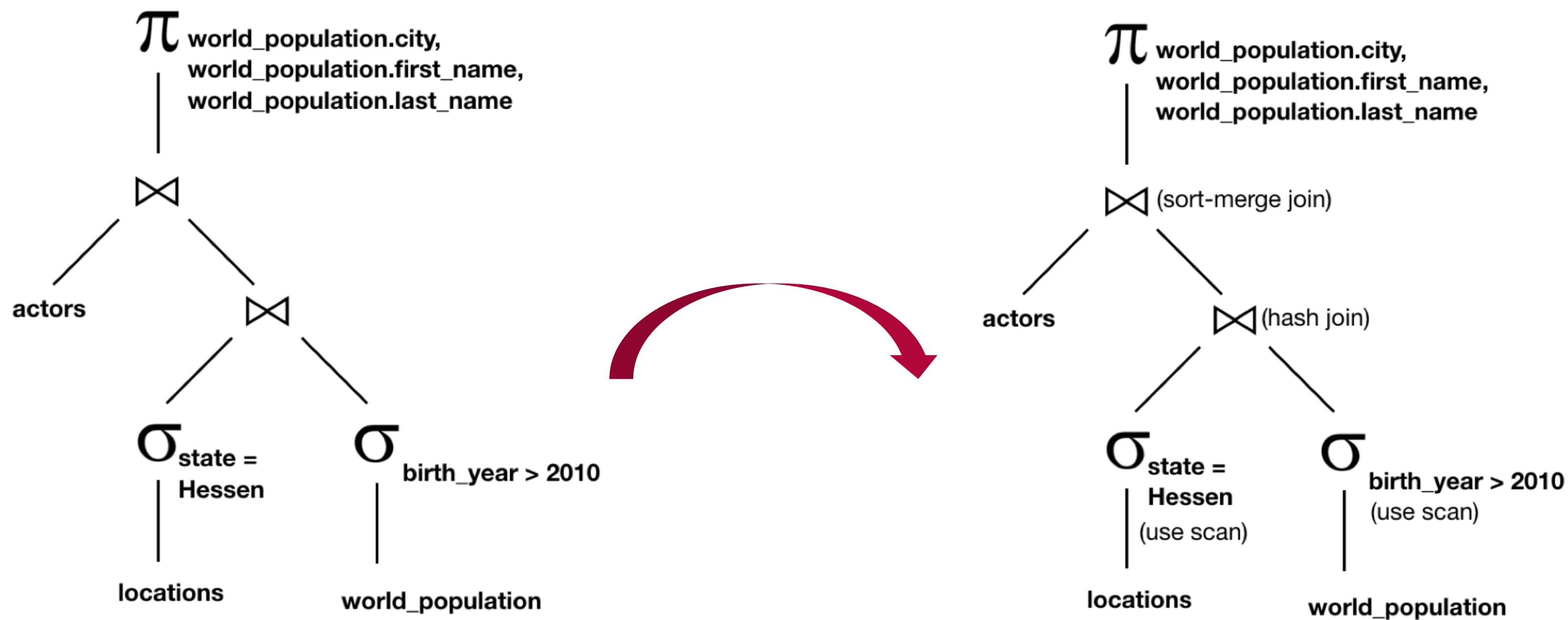
- Logical Query Plan can be seen as a tree of relational algebra operators
- Enumeration phase generates logically equivalent expressions using equivalence rules (i.e., operators can only be reordered to an extend that ensures correct results)



# 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

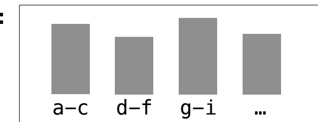
# Query Optimization
# Statistics

- 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

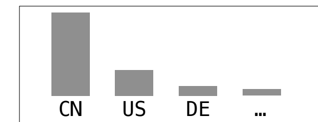
Table: `world_population`

**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

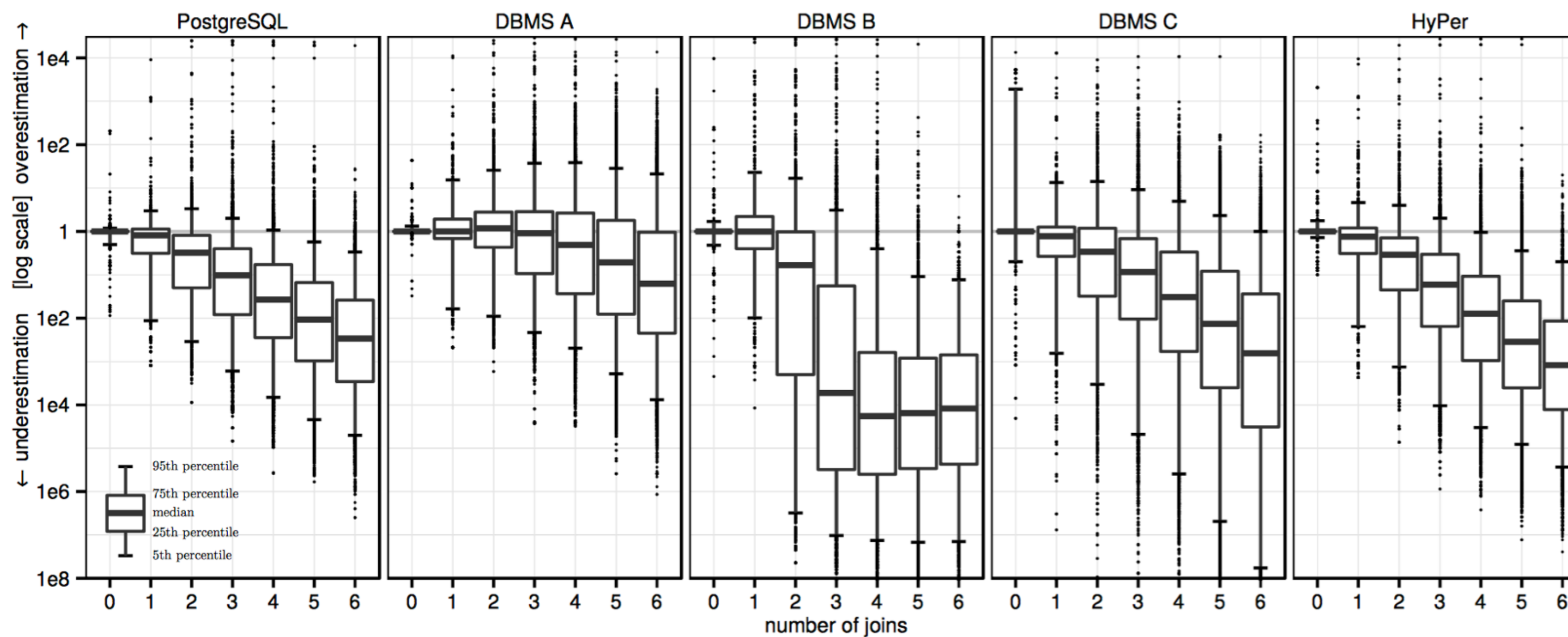
If $r2 \bowtie r3$ is quite large and $r1 \bowtie r2$ is small, we choose

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

so that we compute and store a smaller temporary relation.

# Query Optimization
# Join Ordering

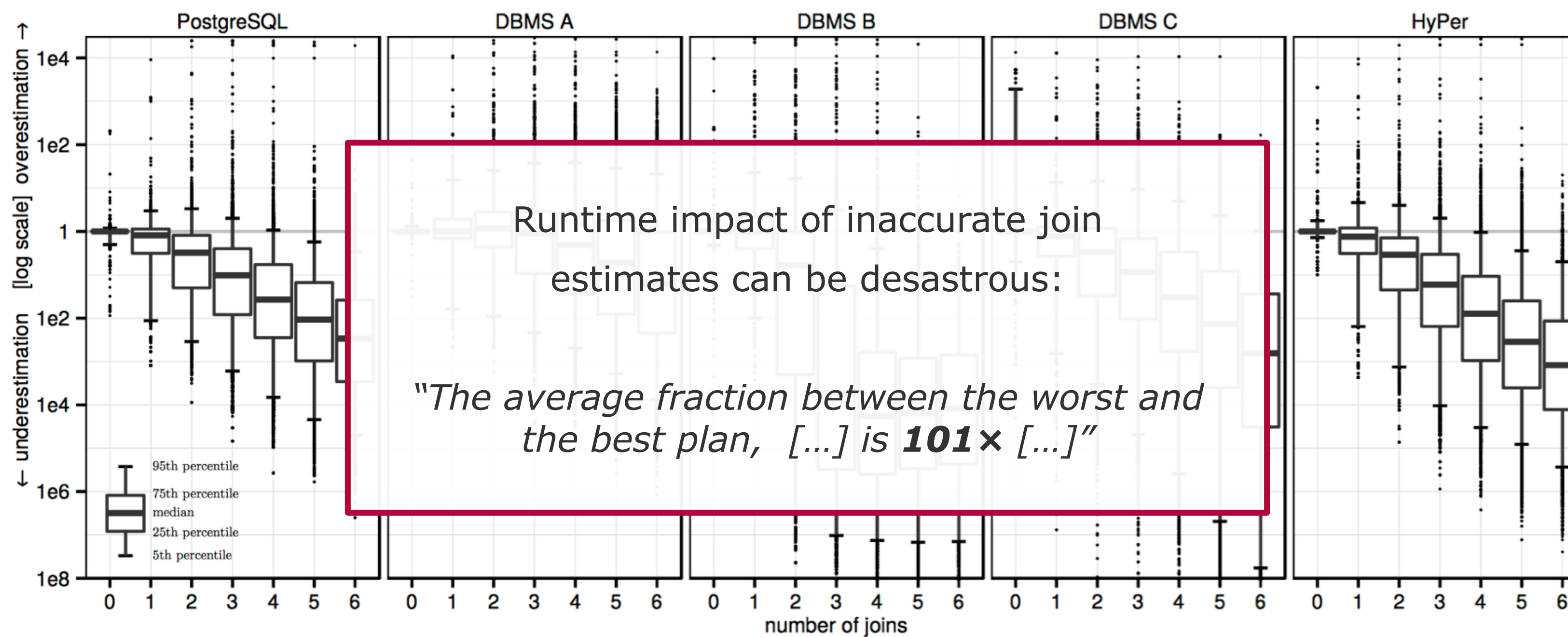
Estimating join cardinalities is one of the challenging tasks of query optimization, but also indispensable to performance.




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.

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

# 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)

HPI Hasso Plattner Institut

# 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.

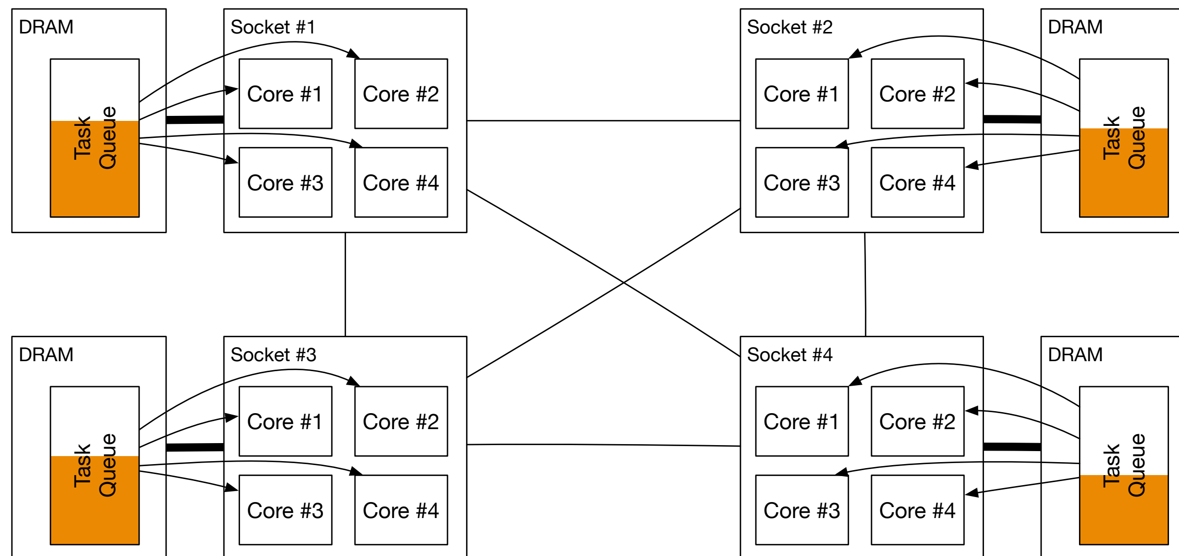
# Query Scheduling
# Task Placement for NUMA Systems

Hasso Plattner Institut

- 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

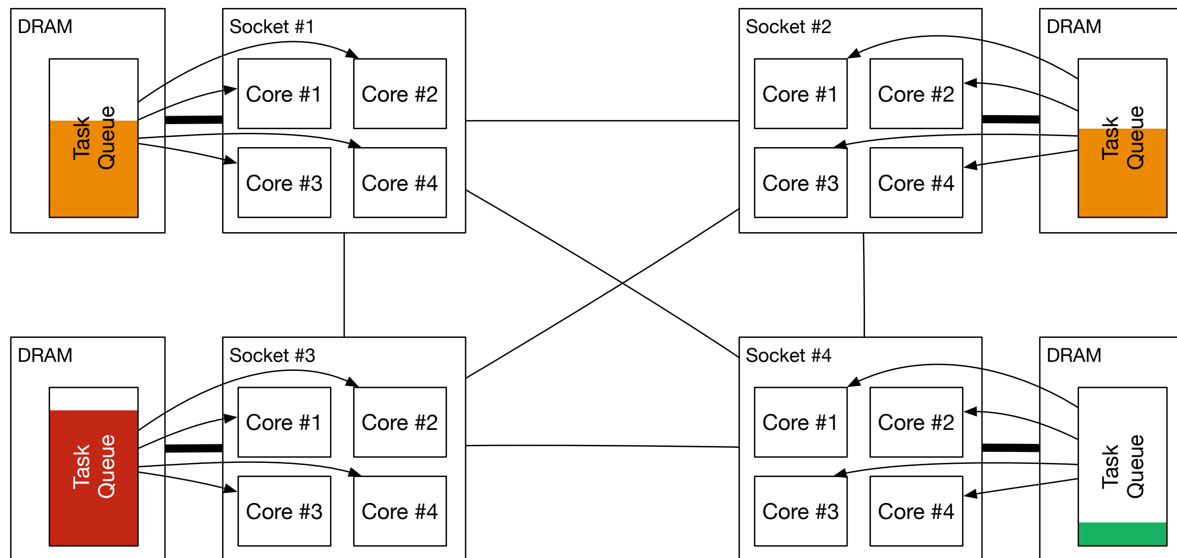
- Every node has its local task queue holding tasks that primarily work on socket-local data.



# Query Scheduling
# Task Placement for NUMA Systems

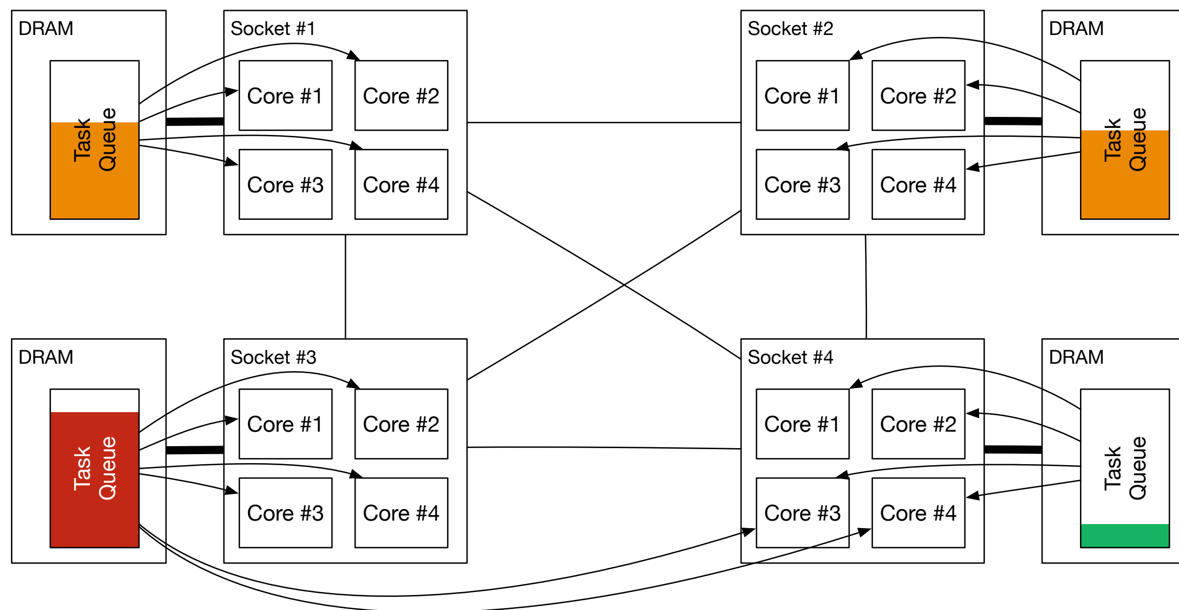


- In real-world applications, workloads are often highly skewed…



# 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.




Hasso Plattner Institut

# Query Scheduling
# Data Placement for NUMA Systems

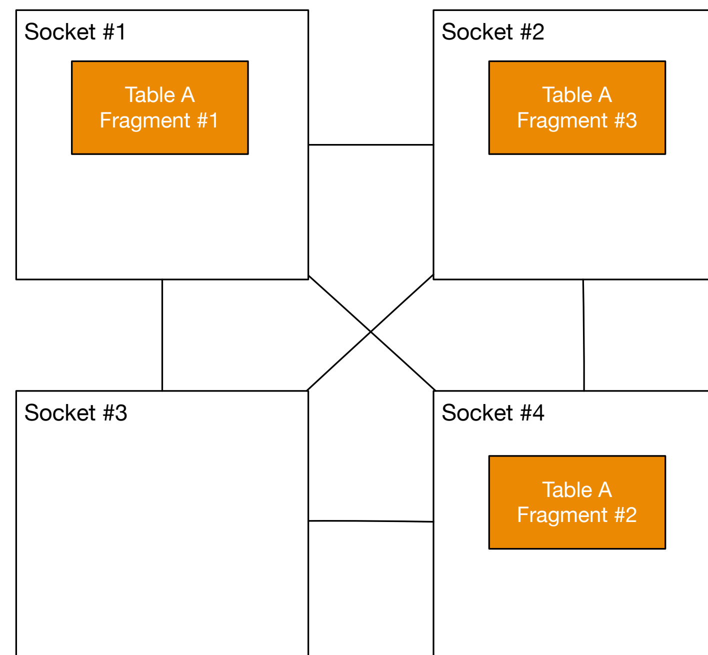


- For any NUMA-aware system, workers should primarily access data that is local to itself (NUMA-aware data placement)
- Thus, the database engine cannot rely on the OS's data placement scheme (e.g., first-touch or interleaved) but has to distribute data across the NUMA nodes on her own and place tasks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

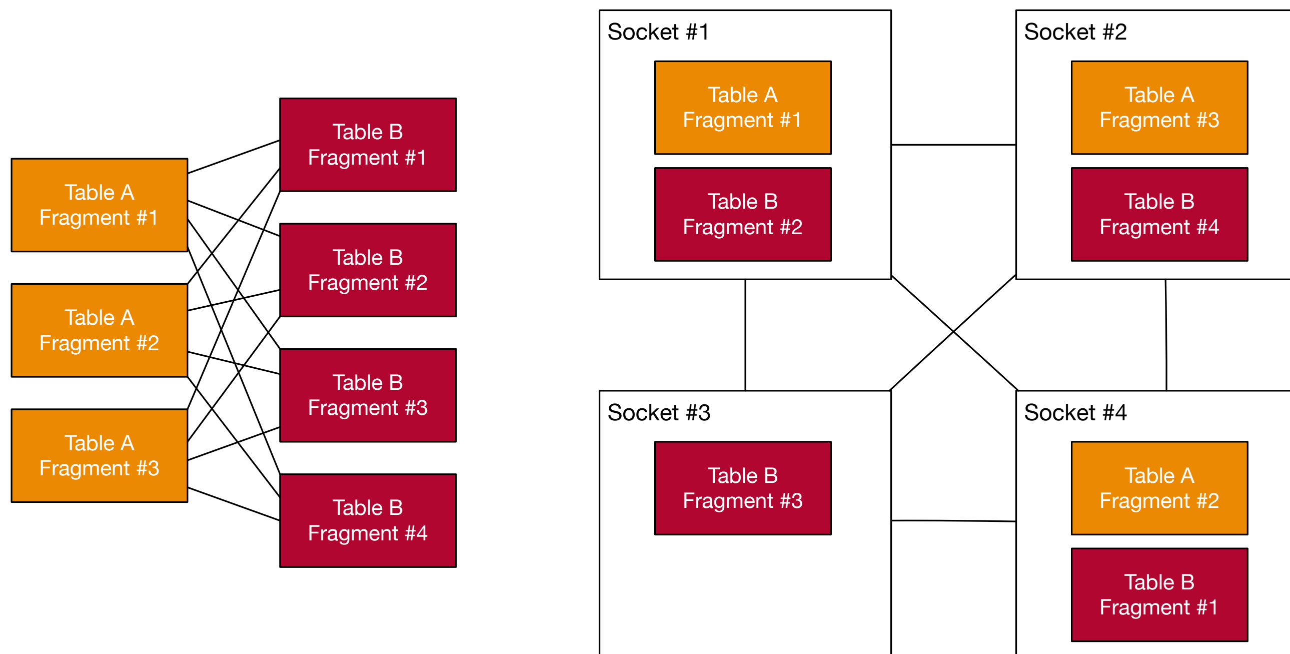
# Query Scheduling
# Data Placement for NUMA Systems

- 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:

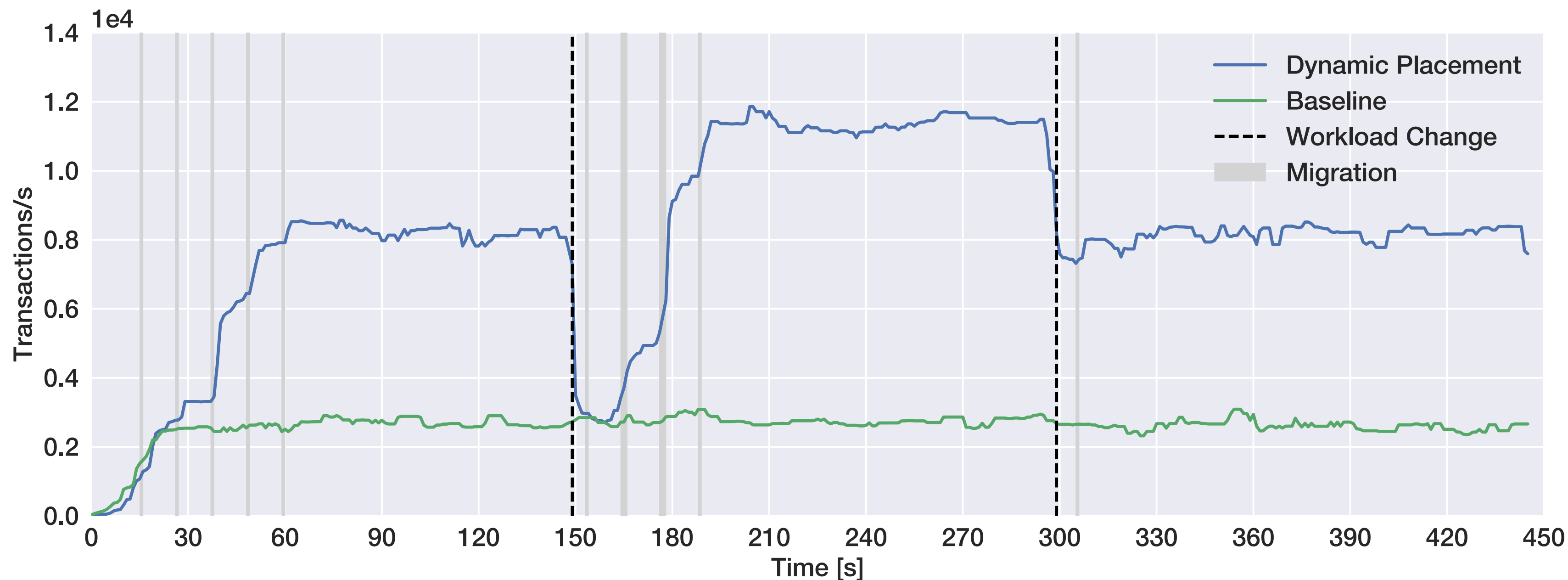



Figure from Norman Rzepka's Master Thesis "Adaptive NUMA-Aware Load-Balancing for In-Memory Databases", 2017

# 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

# Query Execution: Joins

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

# Query Processing – Joins
# What is a Join?



Combination of tuples from different tables

**General categories**

- *Inner Join* – Combine the tuples of two tables by combining each tuple of the first input table with each tuple of the second table to apply a join-predicate, which joins two tuples only if they match
- *Outer Join* – If the join predicate matches, tuples from both join relations are used, if not, the non-matching tuple is filled with NULL values

**Further Specializations**

- *Equi-Join* – most often used join-type, e.g., `tbl_a.attr_1=tbl_b.attr_1`
- *Inequality-Join* – often used in time-series analyses, e.g., joining over range predicates

# Query Processing – Joins
# Join Example

**Who are the actors in the state of Hessen born after 2010?**

```
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
```

Simplifying assumption for this example: city names are unique, last_name identifies unique people

Hasso Plattner Institut

**The three basic Join Algorithms**

1. Nested-Loop Join
2. Sort-Merge Join
   - Principle of merging sorted lists
3. Hash-Based Join
   - Build hash map for smaller join relation
   - Sequential scan over larger join relation and probe into hash map

# Query Processing – Joins
# Nested-Loop Join

Despite its runtime complexity of *O(m*n)* one of the most used joins

**Reasons**

- Low memory consumption
- Runtime complexity often neglectable for small inputs (e.g., small input sizes due to restriction upfront filtering)
- Often the default join operator when an join attributes is indexed (e.g., joining foreign key partners)

Hasso Plattner Institut

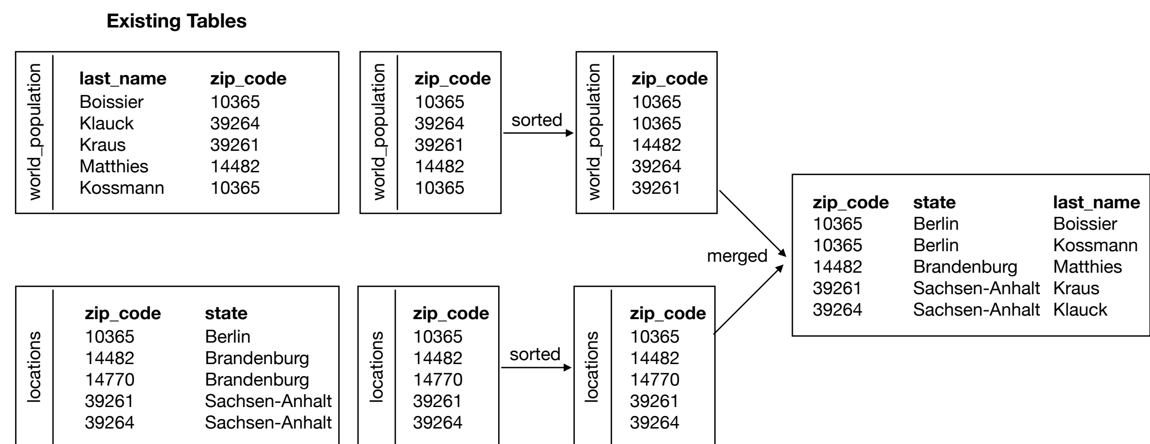
# Query Processing – Joins
# Sort-Merge Join

Hasso Plattner Institut

## Algorithm

- Sort both join keys
- Merge both sorted lists to join

- Fully sequential operation
- With modern CPU-optimized sorting techniques (e.g., bitonic sort) sort-merge joins can outperform hash joins
- Fast for already sorted tables



# Query Processing – Joins
## Hash-Based Join

- Hashing introduces additional complexity because we cannot guarantee the absence of collisions and cannot handle inequality joins
- Hash Join algorithm
  1. **Hash phase**: join attribute of first join table is scanned and a hash map is produced where each value maps to the position of this value in the first table
  2. **Probe phase**: second join table is scanned on the other join attribute and for each tuple the hash map is probed to determine if the value is contained or not; if the value is contained the result is written

HPI Hasso Plattner Institut

# Query Processing – Joins
# Which Join Algorithms to choose?

## When to choose which algorithm?

- *Nested-Loop Join* – very small data set; both other algorithms would require too much time for additional data structures and sorting; indexed join column
- *Sort-Merge Join* – join column(s) already sorted, inequality joins, default fallback
- *Hash Join* – one relation significantly smaller; equi-join

## Complexity comparison

- *Nested-Loop Join – O(N*M)*
- *Sort-Merge Join – O(N*log(N) + M*log(M))*
- *Hash Join – O(N+M)*

## But

- *Sort-merge Join* on already sorted input has a complexity of *O(N+M)* since the data does not need to be sorted
- *Hash Join* performs best if the hash map is expected to be very small (thus probably cache-resident)

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

# Join Execution on Hyrise²
# Radix-Partitioned Hash Join

# Query Processing – Radix-Partitioned Hash Join Joining Large Data Sets



Hyrise[2] is optimized for mixed workloads including complex analytical joins over large data sets. Joining is the *most expensive operation* in mixed workloads.

Text book implementations do not scale well on large multi-core systems:

1. *Hash Join:* random accesses to large non-cache-resident hash maps are not prefetchable and exhibit bad caching utilization.
2. *Sort-based Join:* overly expensive sorting costs in the sort-phase for very large data sets.

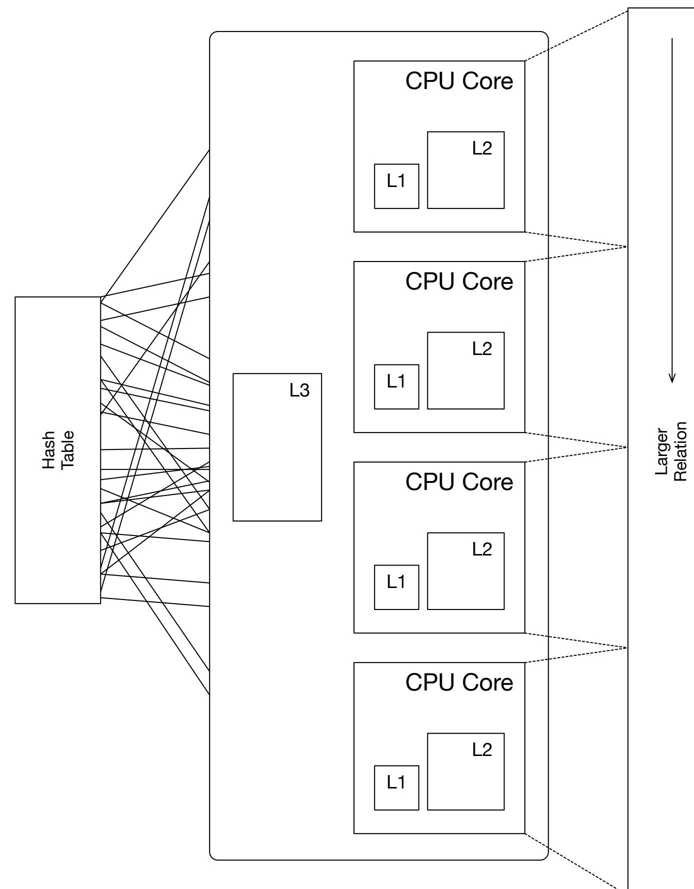
# Query Processing – Radix-Partitioned Hash Join



- Larger relation is sequentially scanned in parallel

**But …**

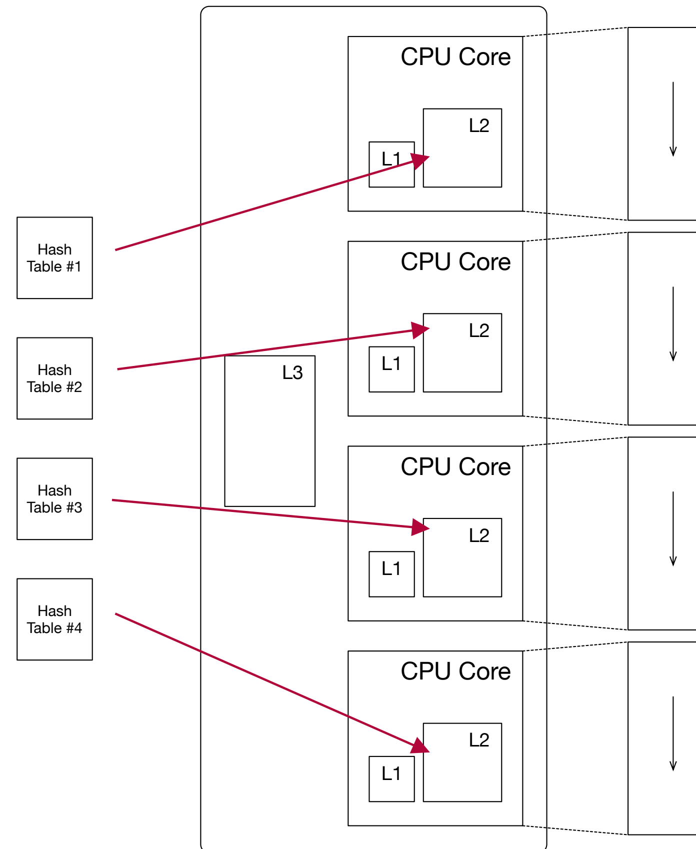
- Hash table of smaller relation is larger than cache sizes
- CPU cores are randomly accessing the hash table which potentially leads to an L3 cache miss for every access



# Query Processing – Radix-Partitioned Hash Join



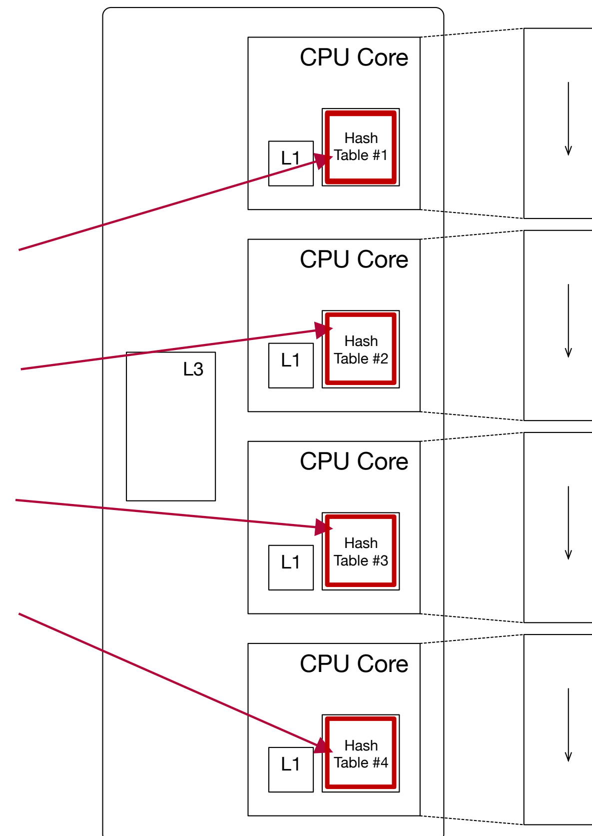
- Key idea: partition both relation in smaller partitions
- We aim to chose a partition size such that the hash tables can be cache-resident



# Query Processing – Radix-Partitioned Hash Join

HPI Hasso Plattner Institut

- CPU cores still process larger relation sequentially
- Random accesses during probing phase are now fully cache-resident in L2 or L3 cache
- Usually, the added work to partition the data is offset by faster joining phase



# Query Processing – Radix-Partitioned Hash Join
# Joining Large Data Sets

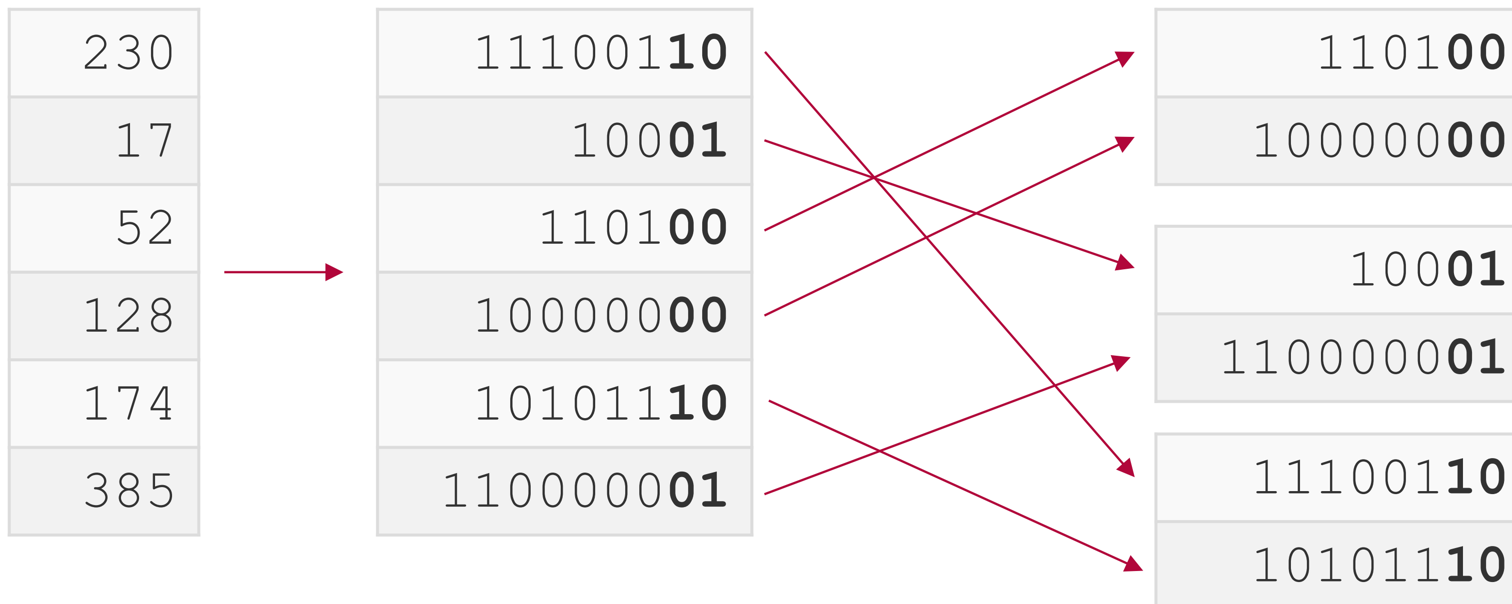


- Parallelization should be used when load permits it
- *Idea:* add additional work to partition input relations into smaller partitions that can be joined locally

**Radix Partitioning**

- Partitions a data set by their *n* least significant bits
- Introduces (potentially multiple) additional scans of the data to join
- Apparently expensive, but surprisingly efficient on columnar IMDBs

# Query Processing – Radix-Partitioned Hash Join
## Radix Partitioning

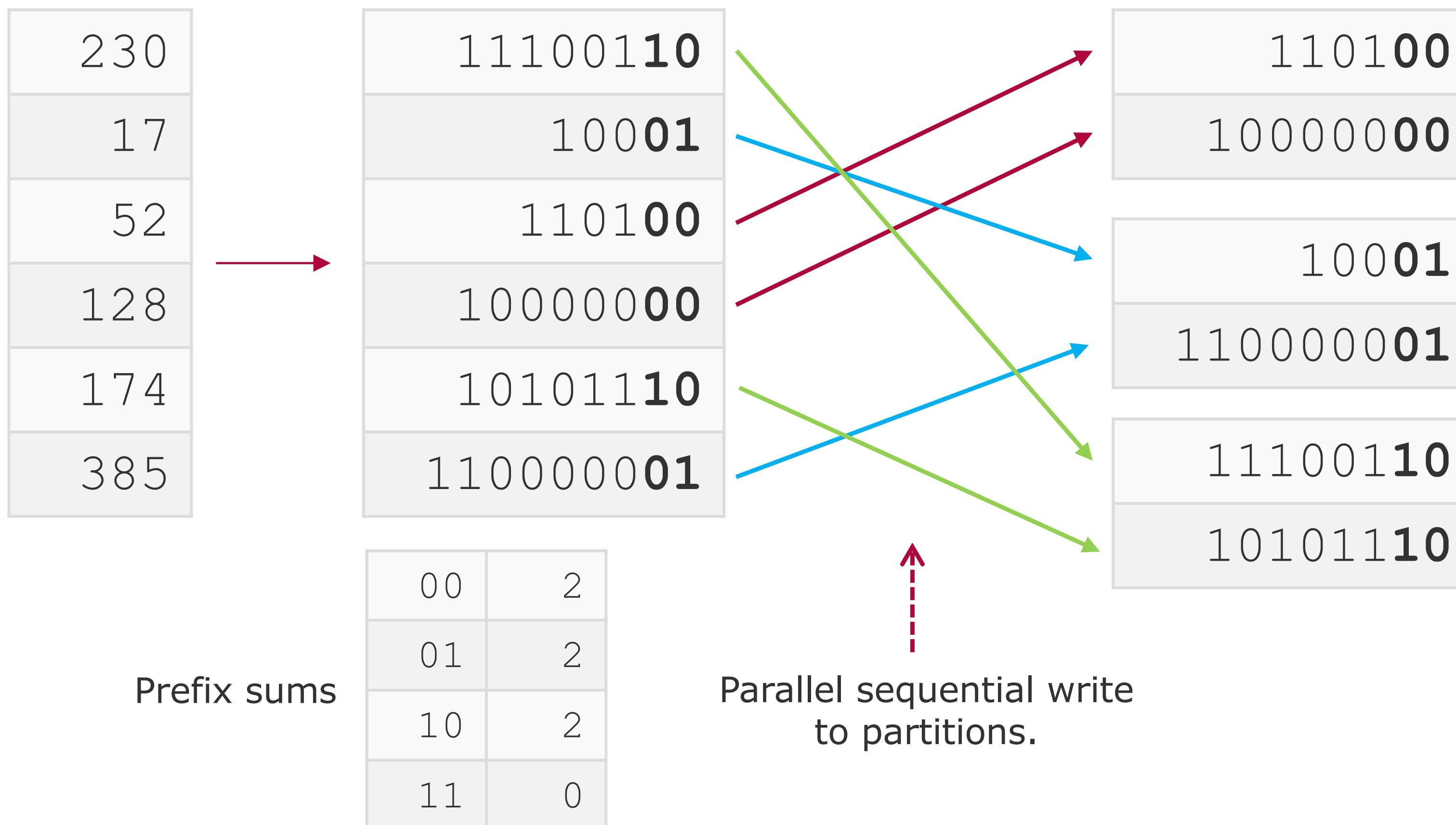


# Query Processing – Radix-Partitioned Hash Join
Radix Partitioning

**Process**

- Determine number of radix bits *n*, prepare $2^n$ (atomic) counters
- First pass: scan entire join columns, determine radix bits for each value and increase the counter accordingly
- Allocate a pre-sized output vector using the calculated prefix sums of the counters
- Second pass: using the prefix sums, directly write values in the corresponding slot of the output vector (sequential writes)
- Usually straightforward to parallelize (in Hyrise[2] parallelized on chunk level)

# Query Processing – Radix-Partitioned Hash Join
## Radix Partitioning




HPI Hasso Plattner Institut

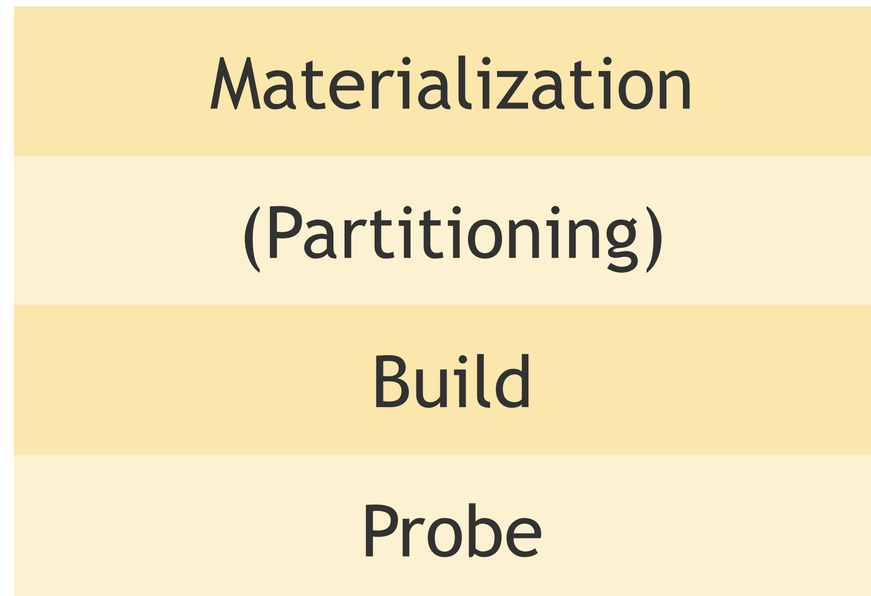
# Query Processing – Radix-Partitioned Hash Join
## Radix Partitioning – Why?

HPI Hasso Plattner Institut

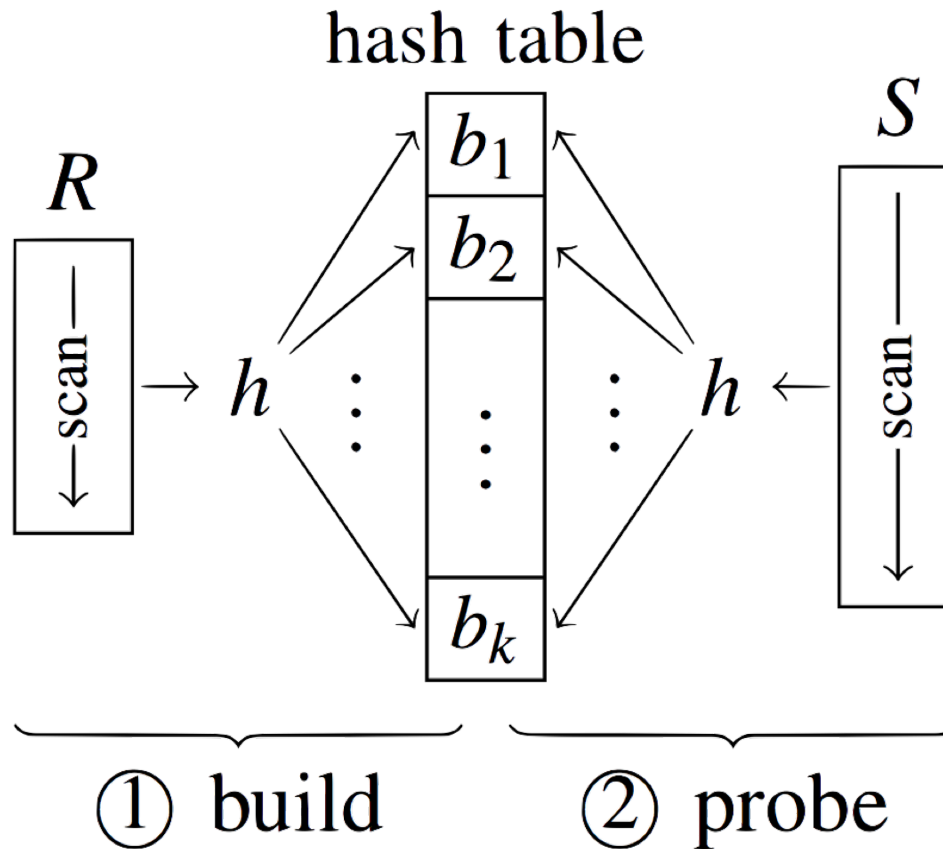
- *Radix partitioning* is the fastest way to *range partition* data without expensive comparisons
- Using least significant bits avoids most data skews
- For equi-join, only radix clusters with the same radix bit pattern have to be joined
  - For a large enough number of radix bits, the hash tables will thus probably be cache-resident
  - Thus, random lookups but fully cached hash table

# Query Processing – Radix-Partitioned Hash Join
## Hash-Join

Radix-partitioned hash join for Hyrise[2].



# Hash-Join

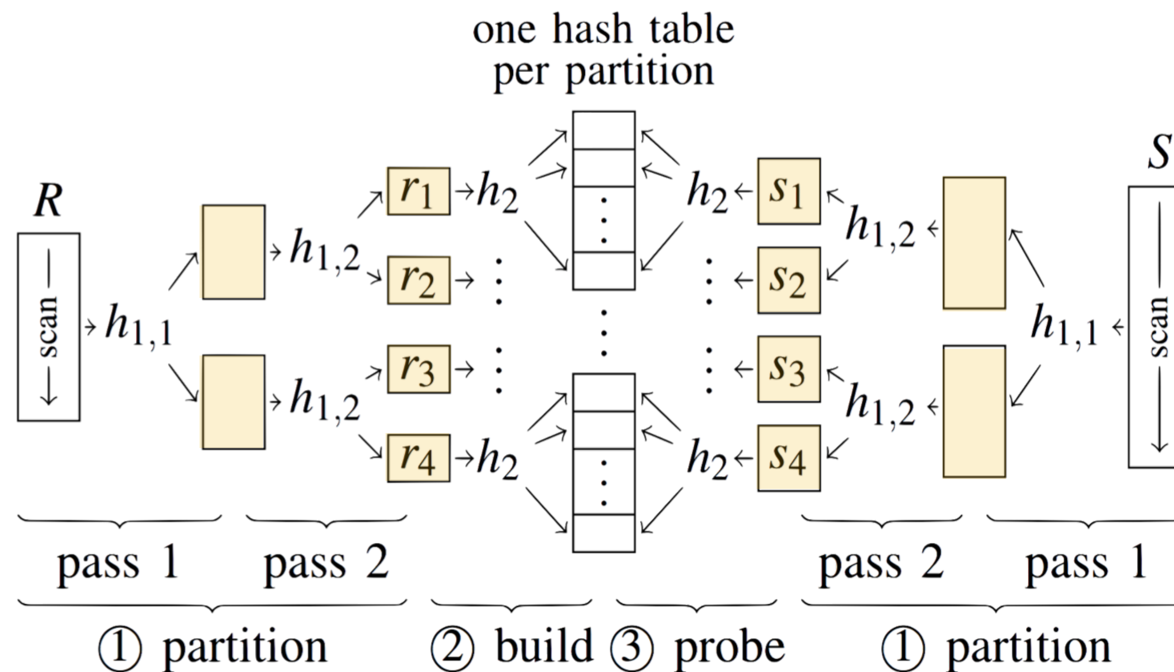



HPI Hasso Plattner Institut

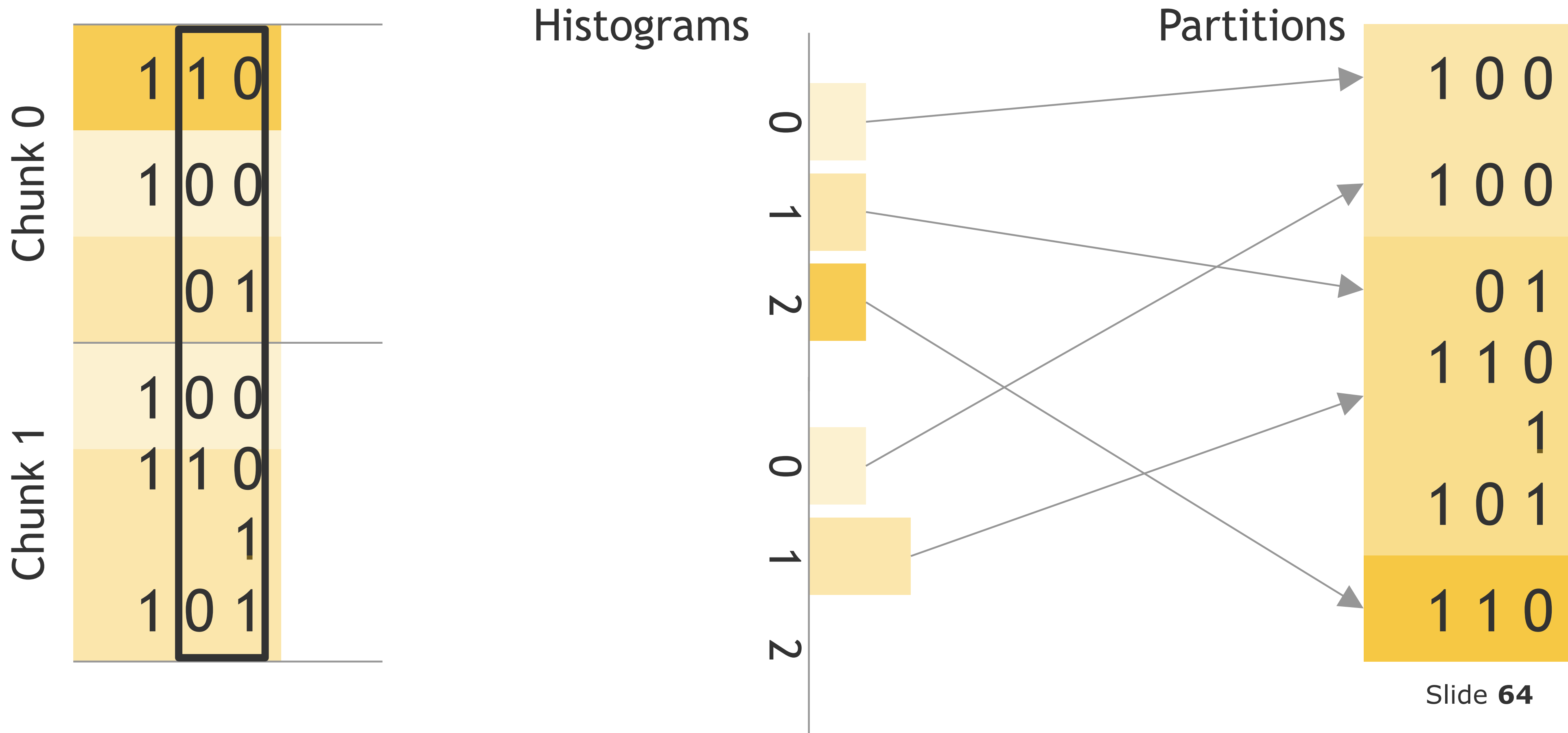
# Query Processing – Radix Partitioned Hash Join
## Parallel Radix Hash-Join



"A Partitioning Phase to the hash joins is introduced to reduce cache misses"



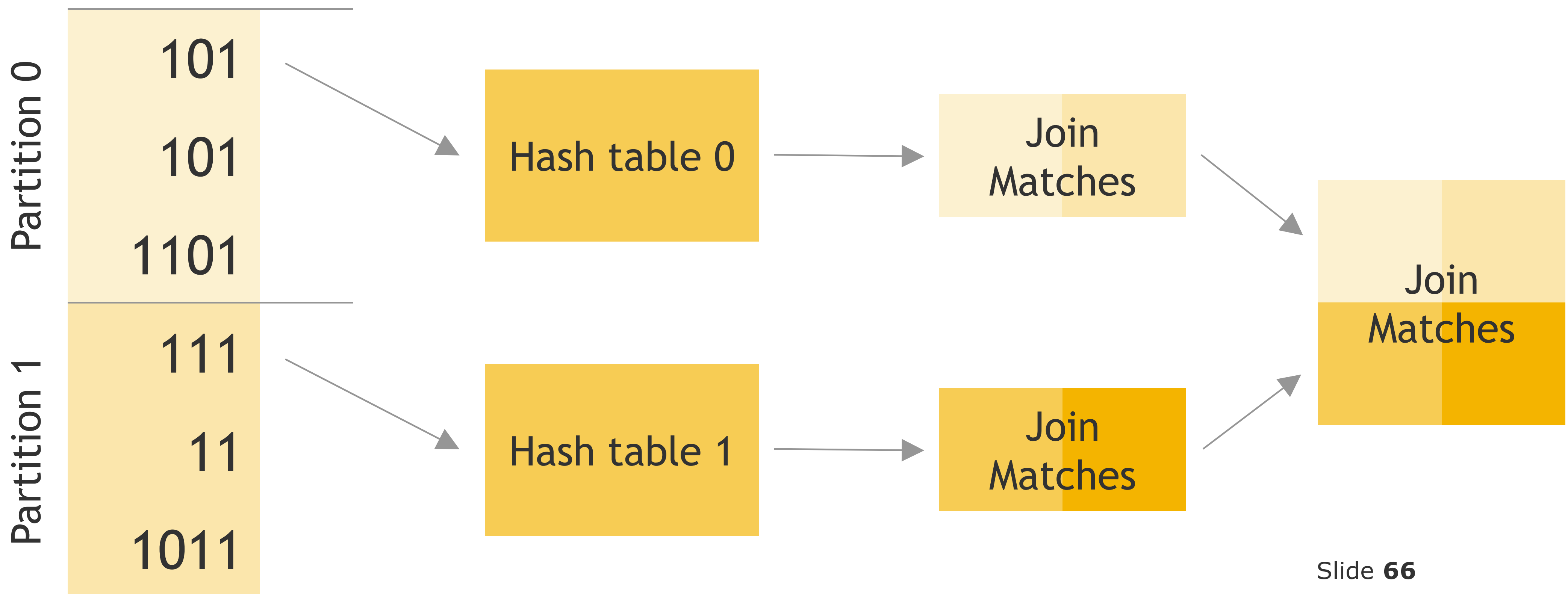
# Query Processing – Radix-Partitioned Hash Join
## Partition Phase – Radix (One Pass)




Hasso Plattner Institut

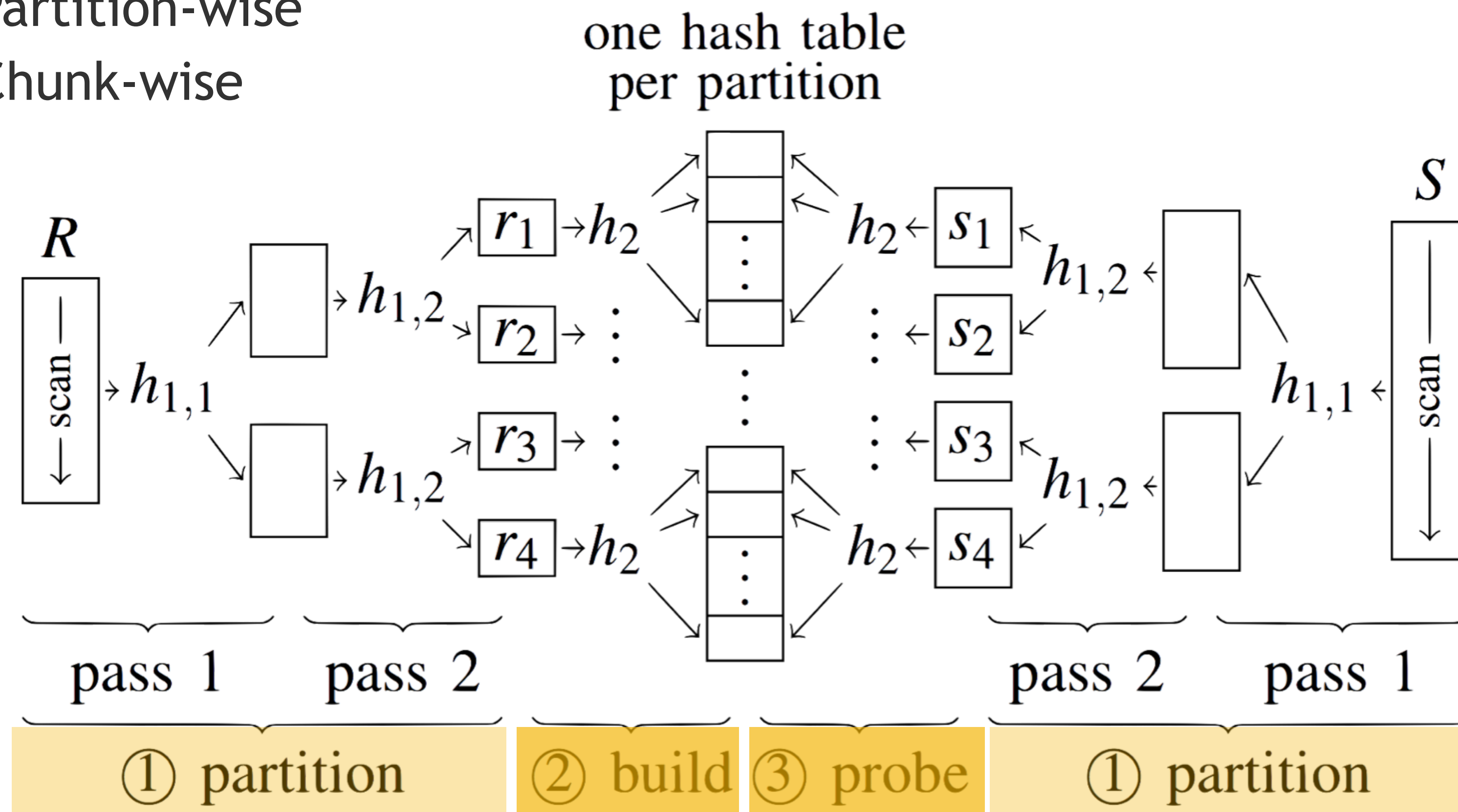
# Query Processing – Radix-Partitioned Hash Join Build Phase

# Query Processing – Radix-Partitioned Hash Join
Probe Phase



# Query Processing – Radix-Partitioned Hash Join Parallelization

- Partition-wise
- Chunk-wise



# Query Processing – Radix-Partitioned Hash Join Summary

Hasso Plattner Institut

We learned that the join is one of the most expensive database operations

- As such, joins are one of the most tuned pieces of code in databases

We discussed the radix-partitioned hash join

- Radix-partitioning allows for improved cache locality of hash maps and
- Allows for improved parallelization

We learned that the optimal join-type choice depends on many aspects

# ORGANISATION

- Deadline Sprint 3
  - 3 December 2017
- Next Week
  - Presentation of group projects
  - Scheduling
  - MVCC