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

```cpp
_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());
```
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
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;
}
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

```cpp
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)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
<th>a OR b</th>
<th>a = b</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>Unknown</td>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>Unknown</td>
<td>False</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
<td>True</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>False</td>
<td>False</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>
NULL values in SQL

Customer

<table>
<thead>
<tr>
<th>c_id</th>
<th>firstname</th>
<th>lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>Zayer</td>
</tr>
<tr>
<td>1</td>
<td>Alex</td>
<td>Geier</td>
</tr>
<tr>
<td>2</td>
<td>Frank</td>
<td>Meier</td>
</tr>
</tbody>
</table>

- 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 \textit{vptr at offset 0}
- \textit{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

```cpp
class Vehicle {
public:
    std::string _model;
    virtual int16_t wheels() = 0;
    virtual std::string get_model() { return _model; }
};

class Car : public Vehicle {
public:
    virtual int16_t wheels() { return 4; }
};

class Motorcycle : public Vehicle {
public:
    virtual int16_t wheels() { return 2; }
};
```
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
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) {
    t.append({i * 14});
}

auto& chunk = t.get_chunk(0);
std::shared_ptr<BaseColumn> base_column = chunk.get_column(0);
auto value_column = std::dynamic_pointer_cast<ValueColumn<int64_t>>(base_column);

int64_t sum1 = 0;
for (size_t offset = 0; offset < rows; ++offset) {
    sum1 += different_access_methods;
}
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)

Median time for 10M accesses

μseconds

AllTypeVariant []
T [] - BC
T [] - VC
values()
VIRTUAL METHODS – OVERHEAD IN OPOSSUM (IV)

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0xcfc</td>
<td>nop1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0xd00</td>
<td>movq</td>
<td>(%r15), %rax</td>
<td></td>
</tr>
<tr>
<td>+0xd03</td>
<td>movq</td>
<td>8(%rax), %rax</td>
<td></td>
</tr>
<tr>
<td>+0xd07</td>
<td>movq</td>
<td>%r15, %rdi</td>
<td></td>
</tr>
<tr>
<td>+0xd0a</td>
<td>movq</td>
<td>%rbx, %rsi</td>
<td></td>
</tr>
<tr>
<td>+0xd0d</td>
<td>callq</td>
<td>*%rax</td>
<td></td>
</tr>
<tr>
<td>+0xd0f</td>
<td>addq</td>
<td>%rax, %r13</td>
<td></td>
</tr>
<tr>
<td>+0xd12</td>
<td>addq</td>
<td>$1, %rbx</td>
<td></td>
</tr>
<tr>
<td>+0xd16</td>
<td>cmpq</td>
<td>$10000000, %rbx</td>
<td></td>
</tr>
<tr>
<td>+0xd1d</td>
<td>jb</td>
<td>&quot;main+0xd00&quot;</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{const std::vector<T>& values()}
\]

\[
\text{T operator[](const size_t i)}
\]
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)

Median time for 10M accesses

μseconds

- AllTypeVariant []
- T [] - BC
- T [] - VC
- values()
- T [] - VC final

Apparent only with clang
Query Processing
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.

```sql
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?

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 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 Optimization
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.

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

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

⇒ return empty result

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

⇒

```sql
SELECT * FROM T
WHERE A < 10
```

**Query Processing**

Slide 16
Query Optimization
Semantic Transformations & Heuristics

\[
\begin{align*}
\text{SELECT} & \quad \text{FROM} \quad T_1, \\
(\text{SELECT} & \quad \text{FROM} \quad T) \quad \text{AS} \quad T_2 \\
\text{WHERE} & \quad T_2.B \quad > \quad 17
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad (A + 2) + 4 \quad \text{FROM} \quad T \\
\text{»} \quad \text{SELECT} & \quad A + 2 + 4 \quad \text{FROM} \quad T \\
\text{»} \quad \text{SELECT} & \quad A + 6 \quad \text{FROM} \quad T
\end{align*}
\]

Query Processing

Slide 17
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 extent that ensures correct results).

### Query Processing

```
π \text{world\_population.city, world\_population.first\_name, world\_population.last\_name}
\quad \bigwedge \quad \text{state} = \text{Hessen} \quad \text{birth\_year} > 2010
\text{Joining 100 000 tuples} \quad \star \quad \text{1 million tuples}
\quad \bigwedge \quad \text{Scanning} \sim 100 \text{ tuples}
\quad \bigwedge \quad \text{Scanning} \sim 100 000 \text{ tuples}
```

```
π \text{world\_population.city, world\_population.first\_name, world\_population.last\_name}
\quad \bigwedge \quad \text{state} = \text{Hessen}
\quad \bigwedge \quad \text{birth\_year} > 2010
\text{Joining 100 000 tuples} \quad \star \quad \text{8 billion tuples}
\text{Joining 100 000 tuples} \quad \star \quad \text{8 billion tuples}
```
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.)
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
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 Hyrise²) might introduce misleading estimations due to outdated statistics

Table: world_population

<table>
<thead>
<tr>
<th>Meta Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes: {'first_name': char(50), 'last_name' [...]}</td>
</tr>
<tr>
<td>Indexed Columns: {'first_name', 'last_name', [...]}</td>
</tr>
<tr>
<td>Statistics:</td>
</tr>
<tr>
<td>min/max: {'birth_year': ['1900', '2017'], [...]}</td>
</tr>
<tr>
<td>distinct_counts: {'birth_year': 118, [...]}</td>
</tr>
<tr>
<td>histograms:</td>
</tr>
<tr>
<td>first_name: a-c  d-f  g-i  [...]</td>
</tr>
<tr>
<td>country: CN  US  DE  [...]</td>
</tr>
</tbody>
</table>

Data

Slide 24
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
For all relations $r_1$, $r_2$, and $r_3$,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

→ Join Associativity

If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

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.

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

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

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
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).
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.
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.
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.
In real-world applications, workloads are often highly skewed...
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.
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 its 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 be executed in parallel with optimal data locality.
- An aggregation on table A (e.g., \( \text{min}() \)) can be executed in parallel with optimal data locality, but final result merging accesses remote data.
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

In case of changing workloads, data placement has to be adapted:
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
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.
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
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
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)
Query Processing – Joins
Sort-Merge Join

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
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.
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** – $O(N\times M)$
- **Sort-Merge Join** – $O(N\times \log(N) + M\times \log(M))$
- **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)
Join Execution on Hyrise²
Radix-Partitioned Hash Join
Hyrise\textsuperscript{2} is optimized for mixed workloads including complex analytical joins over large data sets. Joining is the \textit{most expensive operation} in mixed workloads.

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

1. \textit{Hash Join}: random accesses to large non-cache-resident hash maps are not prefetchable and exhibit bad caching utilization.

2. \textit{Sort-based Join}: overly expensive sorting costs in the sort-phase for very large data sets.
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
Key idea: partition both relation in smaller partitions

We aim to choose a partition size such that the hash tables can be cache-resident
- 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
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

Partition with n=2 radix bits

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>230</td>
<td>11100110</td>
</tr>
<tr>
<td>17</td>
<td>10001</td>
</tr>
<tr>
<td>52</td>
<td>110100</td>
</tr>
<tr>
<td>128</td>
<td>10000000</td>
</tr>
<tr>
<td>174</td>
<td>10101110</td>
</tr>
<tr>
<td>385</td>
<td>11000001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101</td>
<td>00</td>
</tr>
<tr>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>11000001</td>
<td></td>
</tr>
<tr>
<td>11100110</td>
<td></td>
</tr>
<tr>
<td>10101110</td>
<td></td>
</tr>
</tbody>
</table>
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

Partition with n=2 radix bits

<table>
<thead>
<tr>
<th>230</th>
<th>11100110</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>10001</td>
</tr>
<tr>
<td>52</td>
<td>110100</td>
</tr>
<tr>
<td>128</td>
<td>10000000</td>
</tr>
<tr>
<td>174</td>
<td>10101110</td>
</tr>
<tr>
<td>385</td>
<td>110000001</td>
</tr>
</tbody>
</table>

Prefix sums

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>2</td>
</tr>
<tr>
<td>01</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Parallel sequential write to partitions.

Query Processing

Slide 59
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.
Radix-partitioned hash join for Hyrise².

<table>
<thead>
<tr>
<th>Materialization</th>
<th>Build</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Partitioning)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Query Processing – Radix-Partitioned Hash Join

Hash-Join

\[ R \downarrow \text{scan} \rightarrow h : : : h \leftarrow S \downarrow \text{scan} \]

① build
② probe
“A Partitioning Phase to the hash joins is introduced to reduce cache misses”
Query Processing – Radix-Partitioned Hash Join Build Phase

Left Table

<table>
<thead>
<tr>
<th>P0</th>
<th>100</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

Hash table

<table>
<thead>
<tr>
<th>#1</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>01</td>
</tr>
<tr>
<td>#2</td>
<td>101</td>
</tr>
<tr>
<td>#3</td>
<td>1101</td>
</tr>
</tbody>
</table>

Hash function

Slide 65
Query Processing – Radix-Partitioned Hash Join

Probe Phase

Right Table

Partition 0

101
101
1101

Hash table 0

Join Matches

Partition 1

111
11
1011

Hash table 1

Join Matches

Join Matches

Slide 66
Query Processing – Radix-Partitioned Hash Join Parallelization

- Partition-wise
- Chunk-wise

Diagram showing the process of query processing with Radix-Partitioned Hash Join and parallelization:

1. **Pass 1**
   - **Partition**
   - **Pass 2**
   - **Build**
   - **Probe**

2. **Pass 2**
   - **Partition**

The diagram illustrates the steps involved in query processing and the parallelization process.
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