Termin Endpräsentation

- 16 / 19 können um 11:00 – 12:00, also machen wir es so
Running Order

• Networking
• Small String Optimization
• Partitioning
• Optimizer Rules
• Pruning Filters
• Subqueries
• Self-Driving
• NUMA-Optimized Join
Hyrise as a Server

10.01.2018
Lawrence, Stephan, Robert

Supervisor: Stefan Klauck
Task

> ./hyriseServer 5432
Server running...

Start Hyrise as a Server application

> psql -h 123.456.1.1 -p 5432
=> SELECT * FROM foo;
a
---
123
(1 row)

Use any existing Postgres client and execute queries
Work Done so Far

- SELECT * FROM foo;
- Async request handling
- Basic tests
Async Request Handling

- Handle network connections concurrently from one thread
- `boost::asio::io_service` dispatches methods
Frame 18913: 86 bytes on wire (704 bits), 86 bytes captured (704 bits) on interface 0

Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1


PostgreSQL
Type: Row description
Length: 31

Field count: 1
- Table OID: 0
- Column index: 0
- Type OID: 25
- Column length: -1
- Type modifier: -1
- Format: Text (0)
Work to come

- Full query support
  - Currently only SELECT works
- Benchmarking
  - Send timing information back to client
  - Analyze performance (e.g. with pgbench)
- Optimizing #packets sent
- Testing (unit / end2end)
- Bug fixing
- Submit PR
Small String Optimization
Develop your own database

Benjamin Feldmann, Marcel Jankrift, Toni Stachewicz
Advisor: Jan Kossmann
Hasso Plattner Institute
Problem

- (Enterprise) data usually contains many short strings
- std::string uses small-string-optimization
  - msvc: 0 - 15 byte strings
  - gcc >= 5: 0 - 15 byte strings
  - clang: 0 - 22 byte strings
- Longer strings
  - Compiler allocates memory on heap
  - Additional indirection
  - msvc and clang allocate more memory than the string size
Example

Extract of SAP ACDOCA table

<table>
<thead>
<tr>
<th>MATNR</th>
<th>SEGMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-100</td>
<td>MANF</td>
</tr>
<tr>
<td>P-1</td>
<td>MANF</td>
</tr>
<tr>
<td>P-1234</td>
<td>MANF</td>
</tr>
</tbody>
</table>

Layout of vector:

MATNR: P - 1 0 0 ... P - 1 ...

SEGMENT: M A N F ... M A N ...
Our Solution

- Define maximum string length

- Store all strings in one vector of chars
  - Each string has maximum length
Our Solution: Example

- MATNR string length: 6
- SEGMENT string length: 4

Layout of vector:

```
| MATNR: | P | - | 1 | 0 | 0 | \0 | P | - | 1 | \0 | \0 | \0 |
| SEGMENT: | M | A | N | F | M | A | N | F | M | A | N | F |
```

Chart 5
### Drawbacks

- Only one (or a few) long strings
  - More memory consumption for smaller strings

<table>
<thead>
<tr>
<th>ExampleColumn</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
</tr>
<tr>
<td>cd</td>
</tr>
<tr>
<td>this_is_a_very_very_very_very_very_very_very_very_very_very_very_very_very_very_long_string</td>
</tr>
<tr>
<td>123</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>m</td>
</tr>
</tbody>
</table>
ValueVector Implementation

- New class for:
  - DictionaryColumn :: _dictionary
  - ValueColumn :: _values

- ValueVector can store values in
  std::vector<std::string|int|float|...>
- or strings with a fixed length in std::vector<char>
ValueVector Implementation

- Has std::vector functions
- Additional constructor for fixed string length
Partitioning

Felix Musmann, Jonas Chromik, Niklas Hoffmann
Winter Term 2017 / 2018
DYOD
Partitioning

= Dividing a table in multiple parts

✓ Performance gain for queries
partitions not matching the criteria can be pruned

➙ Manageability
➙ Availability
➙ Load-balancing
Vertical Partitioning

<table>
<thead>
<tr>
<th>Name</th>
<th>Birthday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo</td>
<td>1970-01-01</td>
</tr>
<tr>
<td>Morpheus</td>
<td>1955-12-11</td>
</tr>
<tr>
<td>Trinity</td>
<td>1976-06-07</td>
</tr>
<tr>
<td>Smith</td>
<td>1970-01-01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oslo</td>
<td>Norway</td>
</tr>
<tr>
<td>Murmansk</td>
<td>Russia</td>
</tr>
<tr>
<td>Darwin</td>
<td>Australia</td>
</tr>
<tr>
<td>Yokohama</td>
<td>Japan</td>
</tr>
</tbody>
</table>

similar to database normalization, but on physical level
Horizontal Partitioning

<table>
<thead>
<tr>
<th>Name</th>
<th>Birthday</th>
<th>City</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo</td>
<td>1970-01-01</td>
<td>Oslo</td>
<td>Norway</td>
</tr>
<tr>
<td>Morpheus</td>
<td>1955-12-11</td>
<td>Murmansk</td>
<td>Russia</td>
</tr>
<tr>
<td>Trinity</td>
<td>1976-06-07</td>
<td>Darwin</td>
<td>Australia</td>
</tr>
<tr>
<td>Smith</td>
<td>1970-01-01</td>
<td>Yokohama</td>
<td>Japan</td>
</tr>
</tbody>
</table>

take partitioning key and assign data to partition based on some criteria
Horizontal Partitioning: Partitioning Criteria

Problem: How to determine which tuple resides in which partition?

Approaches:

- Range-based
- Hash-based
- Round Robin
- Tuple matching some predicate
- Time-based
- List-based
How to build this?
How to handle partition management?

Alternative #1

Table

<table>
<thead>
<tr>
<th>Partitions</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 1</td>
<td>Partition 2</td>
<td>Partition 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunks</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk 1</td>
<td>Chunk 2</td>
<td>Chunk 3</td>
<td>Chunk 4</td>
</tr>
</tbody>
</table>

Logic for handling Partitions
How to handle partition management?

Alternative #2

Partition Manager

- Partitions
  - Partition 1
  - Partition 2
  - Partition 3

Logic for handling Partitions

Table

- Chunks

Table

- Chunks

Table

- Chunks
How to handle partition management?

Alternative #3

- Partition Manager
  - Partitions
    - Partition Logic
    - Table
      - Chunks
How we built this.
Partitioning as a Strategy

Table

PartitionSchema

Partition 1
Partition 2
Partition 3

NullPartitionSchema
HashPartitionSchema
RoundRobinPartitionSchema
RangePartitionSchema
Our Implementation: Chunks in Partitions
Our implementation:
Interface

```cpp
void create_hash_partitioning(const ColumnID column_id, const HashFunction hash_function,
                               const size_t number_of_partitions);

void create_range_partitioning(const ColumnID column_id,
                                const std::vector<AllTypeVariant> bounds);

void create_round_robin_partitioning(const size_t number_of_partitions);

bool is_partitioned() const;

void remove_partitioning();

std::vector<ChunkID> get_partition(PartitionID partition_id);

std::vector<PartitionID> get_partition_ids();
```
Problems to solve

// creates a new chunk and appends it
void create_new_chunk();

// returns the chunk with the given id
Chunk& get_chunk(ChunkID chunk_id);
const Chunk& get_chunk(ChunkID chunk_id) const;
ProxyChunk get_chunk_with_access_counting(ChunkID chunk_id);
const ProxyChunk get_chunk_with_access_counting(ChunkID chunk_id) const;

// Adds a chunk to the table. If the first chunk is empty, it is replaced.
void emplace_chunk(Chunk chunk);

(excerpt from table.hpp)
We had to deal with the following questions:

- Single criterion vs. multiple criteria
- What if a tuple does not belong in any partition?
Multiple Criteria: Problem Statement

Partition 1

\[
\text{gender} = 'M' \ \text{AND} \ \text{year} < 1970
\]

Partition 2

[everything else]
Multiple Criteria:
Problem Statement

Partition 1

\( \text{gender} = \text{'M'} \) AND \( \text{year} < 1970 \)

Partition 2

\( \text{gender} = \text{'M'} \) AND \( \text{NOT year} < 1970 \)

Partition 3

\( \text{NOT gender} = \text{'M'} \) AND \( \text{year} < 1970 \)

Partition 4

\( \text{NOT gender} = \text{'M'} \) AND \( \text{NOT year} < 1970 \)
Each table has only one partition schema. The partition schema has to ensure consistency.

For example:

- Range partitioning uses split points. 
  
  \([20, 50]\) leads to 3 partitions:
  
  - one with values \(\leq 20\)
  - one with \(20 < \text{values} \leq 50\)
  - one with values \(> 50\)

- Hash partitioning has user-defined number of partitions.
  E.g. modulo of hash value.
Further Work: Liberal Partitioning

Liberal Partitioning:

Partitioning criteria are not disjoint
→ One tuple can be in multiple partitions

If a tuple matches multiple partitions, we have to decide

1. Put it in all matching partitions → Deduplication problem
2. Put it in one matching partition → More partitions to be searched
3. Put it in a remainder partition → Can lead to uneven distribution
Optimizer Rules

Midterm Presentation
Develop your own database
Falco Dürsch, Maxi Fischer, Tim Friedrich
2018-01-10
Optimizer Component

**Idea:** Transform a query plan to a more performant one (memory, CPU)

- Consists of a set of transformation rules
- Loops through them without any sense of ordering (no dependency management between rules)
Predicate Pushdown Explained

Projection

Predicate

Join

Stored Table

1000 rows

1000 rows

1000 rows

1000 rows

Projection

Join

Predicate

Projection

Join

Predicate

Stored Table

1000 rows

1000 rows

300 rows

1000 rows

300 rows

300 rows

1000 rows

1000 rows

300 rows

Optimizer
Dürsch
Fischer
Friedrich
Slide 3
### Predicate Pushdown - Node Types

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stored Table</td>
<td>Tables need to be loaded first</td>
</tr>
<tr>
<td>Aggregate, Limit</td>
<td>Predicate could change row count or refer to aggregated value</td>
</tr>
<tr>
<td>Union, Select Distinct</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Validate, Predicate</td>
<td>Subject to Predicate Reordering Rule</td>
</tr>
</tbody>
</table>
## Predicate Pushdown - Node Types

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stored Table</td>
<td>Tables need to be loaded first</td>
</tr>
<tr>
<td>Aggregate, Limit</td>
<td>Predicate could change row count or refer to aggregated value</td>
</tr>
<tr>
<td>Union, Select Distinct</td>
<td>Not implemented</td>
</tr>
<tr>
<td>Validate, Predicate</td>
<td>Subject to Predicate Reordering Rule</td>
</tr>
<tr>
<td>Projection</td>
<td>Possible if predicate column remains unchanged (arithmetic)</td>
</tr>
<tr>
<td>Sort</td>
<td>Possible</td>
</tr>
<tr>
<td>Join</td>
<td>Possible, dependent on join type Predicate must not involve reference both join partners</td>
</tr>
</tbody>
</table>
Predicate Pushdown Example

- **Projection**
  - WHERE $a = 3$
  - WHERE $a = b$
  - WHERE $a = d$

- **Predicate**
  - WHERE $d = \text{NULL}$

- **Join**
  - 1000 rows
  - 1000 rows
  - 1000 rows

- **Stored Table**
  - R(a, b, c)
  - S(d, e)

**Note:**
- The predicate WHERE $d = \text{NULL}$ should be pushed down to the join operation for optimization.
Conclusion and Next steps

- Costs of joins can be reduced
- Precise definition required to keep LQP idempotence
- Another optimizer rule (Logical optimization, Sort Positioning Rule)
- Support more node types (Projection, Sort)
- Support TPC-H-13 query
Backup
Predicates Pushdown Example
Pruning Filters

Speed up filter operations
What is Pruning?

Reducing the amount of data (chunks in our case) to process when executing queries.

Improves cardinality estimation of filtering operations (and thus helps the optimizer determine a better order of execution)
What do we do?

We will implement pruning for immutable chunks by extending compressed chunks with statistics (e.g. min, max).

These will be used to reduce the amount of chunks that are considered in the GetTable operator.
How do we do it?

- add statistics calculation to chunk compression
- make chunk statistics available to the optimizer
- add ChunkPruningRule to the query optimizer
- create exclusion list of chunks for the StoredTable LQP Nodes
- use exclusion lists to perform less scanning
- remove prunable chunks in GetTable Operator
How does the optimizer rule work?

```
Predicates:
- Predicate #1: b > 7
- Predicate #2: a < 10
- Predicate #3: c = 8

Table:
- StoredTable

Exclusion list: nullptr

All Chunks: {1, ..., 10}
```
How does the optimizer rule work?

- Predicate #1: $b > 7$
- Predicate #2: $a < 10$
- Predicate #3: $c = 8$

All Chunks: $\{1, \ldots, 10\}$
How does the optimizer rule work?

initialize

calculate and union exclusion lists

Predicate #1
b > 7

Predicate #2
a < 10

Predicate #3
c = 8

StoredTable

exclusion list: {1,2,5,7}

All Chunks: {1, ..., 10}
How does the optimizer rule work?

All Chunks: \{1, \ldots, 10\}
How does the optimizer rule work?

All Chunks: {1, ..., 10}

Predicate #1
\[ b > 7 \]

Predicate #2
\[ a < 10 \]

Predicate #3
\[ c = 8 \]

StoredTable

exclusion list: \{1,2,5,7\}
\{7\}

\{3,7,9\} calculate

intersect
Things that will get modified

- Chunk: store ChunkStatistics with std::optional
- New Class: ChunkStatistics
- compress_* methods: calculate and create ChunkStatistics
- New Optimizer Rule (as shown in previous slide)
- GetTable Operator: create temporary table without the excluded chunks
- StoredTableNode (LQP): store the exclusion list
Timeline

min/max calculation
10.1. now

pruning pipeline
17.1.

benchmarks
24.1.

optional features (e.g. more filters)
31.1.

final presentation
7.2.
Subqueries

Philipp Otto, Juliane Waack, David Hahn
Develop your own Database - Winter Term 2017/18
SELECT a FROM t1
WHERE a < (SELECT MAX(b) FROM t2)
Uncorrelated:

SELECT a FROM t1 WHERE a < (SELECT MAX(b) FROM t2)

Correlated:

SELECT a, (SELECT b FROM t2 WHERE b = a + 4) FROM t1;
(debug)> SELECT a FROM t1

--- Columns
<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
</tr>
</tbody>
</table>

--- Chunk 0 ---
|  1.1 |
|  2.2 |
|  3.3 |
|  4.4 |

4 rows total (PARSE: 13 μs, COMPIL: 16 μs, EXECUTE: 350 μs (wall time))

(debug)> SELECT MIN(b) from t2

--- Columns
<table>
<thead>
<tr>
<th>MIN(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
</tr>
</tbody>
</table>

--- Chunk 0 ---
|  1.1 |

1 rows total (PARSE: 14 μs, COMPIL: 46 μs, EXECUTE: 2573 μs (wall time))
(debug)> SELECT a FROM t1 WHERE a > (SELECT MIN(b) from t2)

--- Columns
<table>
<thead>
<tr>
<th></th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>float</td>
</tr>
</tbody>
</table>

--- Chunk 0 ---
|   | 2.2|
|   | 3.3|
|   | 4.4|

---
3 rows total (PARSE: 14 µs, COMPILE: 16 µs, EXECUTE: 742 µs (wall time))

(debug)>
SELECT a FROM t1
WHERE a < (SELECT MAX(b) FROM t2)
SELECT a FROM t1
WHERE a < (SELECT MAX(b) FROM t2)
SELECT a FROM t1
WHERE a < (SELECT MAX(b) FROM t2)
SELECT a FROM t1
WHERE a < (SELECT MAX(b) FROM t2)
SELECT a FROM t1
WHERE a < (SELECT MAX(b) FROM t2)
Step 1
execute subquery separately for every row

Step 2
optimize uncorrelated subqueries by only executing once

Step 3
flatten correlated subqueries to JOINs if possible
Step 3

SELECT a, (SELECT b FROM t2 WHERE b = a + 4) FROM t1;

SELECT a, b FROM t1 LEFT JOIN t2 ON b = a + 4;
Subqueries

Philipp Otto, Juliane Waack, David Hahn

Develop your own Database - Winter Term 2017/18
Self Driving Database

Adrian Holfter, Arthur Silber, Lukas Wenzel
Instructor: Jan Kossmann
Manual DB tuning is difficult

Which indexes should be created?
How should the data be partitioned?
How many threads should be used?

Large Problem Space, Inter-dependencies, Workload-specific decisions
The database knows best how to tune itself

Use Heuristics to automatically create indices
## Example: Bank Accounts

<table>
<thead>
<tr>
<th>NAME</th>
<th>BALANCE</th>
<th>INTEREST</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Danni Cohdwell</td>
<td>144'811</td>
<td>0.157</td>
<td>3</td>
</tr>
<tr>
<td>Rosemary Picardi</td>
<td>236</td>
<td>0.226</td>
<td>3</td>
</tr>
<tr>
<td>Xenia Ziegler</td>
<td>424'675</td>
<td>0.239</td>
<td>2</td>
</tr>
<tr>
<td>Lilly Goodwin</td>
<td>3'645</td>
<td>0.538</td>
<td>5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
# Example: Bank Accounts

```sql
select BALANCE from CUSTOMER where NAME = 'Danni Cohdwell'
select NAME from CUSTOMER where LEVEL = 5
select INTEREST from CUSTOMER where NAME  = 'Rosemary Picardi'
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>BALANCE</th>
<th>INTEREST</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Danni Cohdwell</td>
<td>144'811</td>
<td>0.157</td>
<td>3</td>
</tr>
<tr>
<td>Rosemary Picardi</td>
<td>236</td>
<td>0.226</td>
<td>3</td>
</tr>
</tbody>
</table>
Example: Bank Accounts

select BALANCE from CUSTOMER where NAME = 'Danni Cohdwell'
select NAME from CUSTOMER where LEVEL = 5
select INTEREST from CUSTOMER where NAME  = 'Rosemary Picardi'

⇒ Which indices should be created?
Example: Bank Accounts

select BALANCE from CUSTOMER where NAME = 'Danni Cohdwell'
select NAME from CUSTOMER where LEVEL = 5
select INTEREST from CUSTOMER where NAME  = 'Rosemary Picardi'

scanned 2x
unique values

<table>
<thead>
<tr>
<th>NAME</th>
<th>BALANCE</th>
<th>INTEREST</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Danni Cohdwell</td>
<td>144'811</td>
<td>0.157</td>
<td>3</td>
</tr>
<tr>
<td>Rosemary Picardi</td>
<td>236</td>
<td>0.226</td>
<td>3</td>
</tr>
</tbody>
</table>
Example: Bank Accounts

select BALANCE from CUSTOMER where NAME = 'Danni Cohdwell'
select NAME from CUSTOMER where LEVEL = 5
select INTEREST from CUSTOMER where NAME = 'Rosemary Picardi'

<table>
<thead>
<tr>
<th>NAME</th>
<th>BALANCE</th>
<th>INTEREST</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Danni Cohdwell</td>
<td>144'811</td>
<td>0.157</td>
<td>3</td>
</tr>
<tr>
<td>Rosemary Picardi</td>
<td>236</td>
<td>0.226</td>
<td>3</td>
</tr>
</tbody>
</table>

- **scanned 2x**
  - unique values
- **scanned 1x**
  - repeating values
Example: Bank Accounts

<table>
<thead>
<tr>
<th>NAME</th>
<th>BALANCE</th>
<th>INTEREST</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Danni Cohdwell</td>
<td>144,811</td>
<td>0.157</td>
<td>3</td>
</tr>
<tr>
<td>Rosemary Picardi</td>
<td>236</td>
<td>0.226</td>
<td>3</td>
</tr>
<tr>
<td>Xenia Ziegler</td>
<td>424,675</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Lilly Goodwin</td>
<td>3,645</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

- select BALANCE from CUSTOMER where NAME = 'Danni Cohdwell'
- select NAME from CUSTOMER where LEVEL = 5
- select INTEREST from CUSTOMER where NAME = 'Rosemary Picardi'

scanned 2x unique values

scanned 1x repeating values
Example: Bank Accounts

select BALANCE from CUSTOMER where NAME = 'Danni Cohdwell'
select NAME from CUSTOMER where LEVEL = 5
select INTEREST from CUSTOMER where NAME = 'Rosemary Picardi'

⇒ Create index on NAME and (maybe) LEVEL

<table>
<thead>
<tr>
<th>NAME</th>
<th>BALANCE</th>
<th>INTEREST</th>
<th>LEVEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Danni Cohdwell</td>
<td>144'811</td>
<td>0.157</td>
<td>3</td>
</tr>
<tr>
<td>Rosemary Picardi</td>
<td>236</td>
<td>0.226</td>
<td>3</td>
</tr>
</tbody>
</table>

scanned 2x
unique values

scanned 1x
repeating values
Our Architecture

1. Evaluate
   - IndexHeuristic

2. Uses
   - System Statistics

3. List of proposals:
   - (table, column) + desirability

4. Create/delete indices

- IndexTuner

- System Statistics
  - Query Cache
  - Table Statistics
  - ...
Loading binary table...
Table loaded (10'000'000 rows in 10 chunks)
Executing queries a first time to fill up the cache...
Execute IndexTuner...

Recommended changes:
  Create index on table CUSTOMER, column NAME (desirability 100%)
  Create index on table CUSTOMER, column LEVEL (desirability 75%)
Demo

Executing queries a second time (with optimized indices)...
Execution times are in microseconds

SELECT BALANCE FROM CUSTOMER WHERE NAME = 'Danni Cohdwell'
  reduced to: 27.829%  (before/after: 2060 / 558)

SELECT NAME FROM CUSTOMER WHERE LEVEL = 5
  reduced to: 7.507%  (before/after: 60370 / 4810)

SELECT INTEREST FROM CUSTOMER WHERE NAME  = 'Rosemary Picardi'
  reduced to: 2.896%  (before/after: 2033 / 57)
Outlook

- Desirability metrics (consider value distributions, query frequencies)
- Index budgeting (creation and maintenance costs, memory footprint) → Cost/Benefit optimization
- Integration into Hyrise:
  - Generalize cache implementation specifics
  - SQL Pipeline does not yet use caching
  - IndexScan not yet used
NUMA-Optimized Join

Develop Your Own Database - WS 17/18
Mid-term Presentation
Jonas Beyer, Julian Niedermeier, Florian Wagner
Existing Joins

1. Hash Join
2. Sort Merge Join
3. Nested Loop Join
Existing Joins

1. Hash Join
2. Sort Merge Join
3. Nested Loop Join

Not NUMA aware
What is NUMA? Reminder

NUMA-Optimized Join - Develop Your Own Database - WS 17/18 - 10.01.18
Massively Parallel Sort-Merge Join (MPSM)

- No hash calculation
- No hashmap probing
- Only sequential data access through histogram based partitioning
- No synchronisation during join
Massively Parallel Sort-Merge Join
Massively Parallel Sort-Merge Join
Massively Parallel Sort-Merge Join - Phase 2

NUMA-Optimized Join - Develop Your Own Database - WS 17/18 - 10.01.18
Massively Parallel Sort-Merge Join - Phase 4

NUMA-Optimized Join - Develop Your Own Database - WS 17/18 - 10.01.18
In Hyrise (Outlook)

- Build Index Join
- Implement Range-Partitioned MPSM Join
- Extend MPSM Join to split smartly (using histograms)
- *(Make Index Join NUMA aware)*
Benchmarks

- **TPCH**
  - Queries 2, 8, 9

- **Join on selected columns**
  - Happens often
  - Can skew range partitions

- **Generated data**
  - Different skew
  - Different relative sizes