Develop your own Database

Week 8

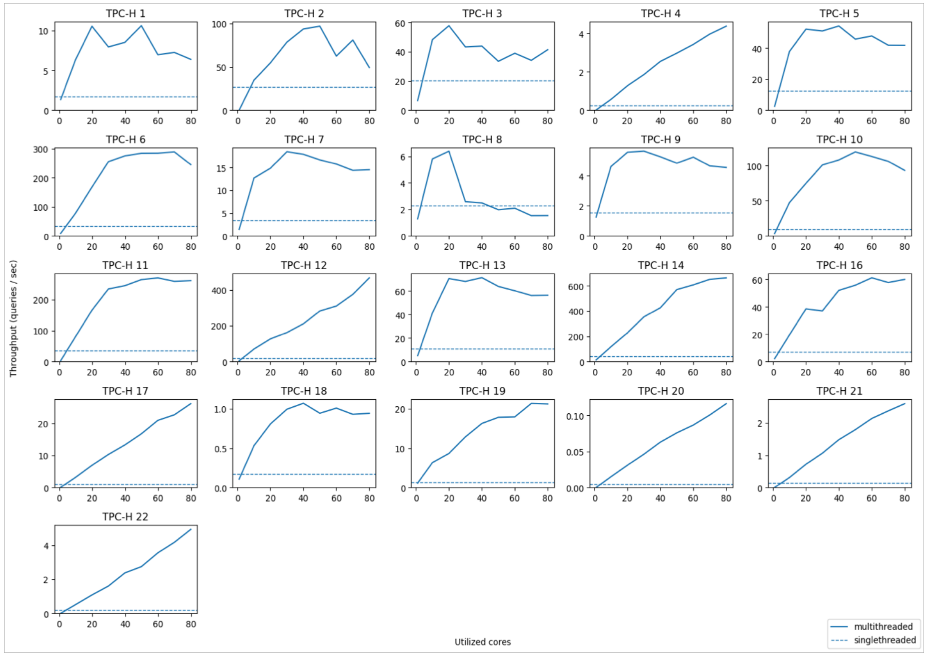
Review Sprint 3

- Master’s project
- Review Sprint 3
- Performance Challenge
- Group Meetings

#1: Multi-Threading and NUMA

- Mostly, we have looked at single-threaded performance

- Some queries can be run in parallel quite well, others perform worse the more cores we use



- We want to improve the scheduler and remove dependencies between tasks

- Wherever it is a bottleneck, we also want to improve the NUMA placement

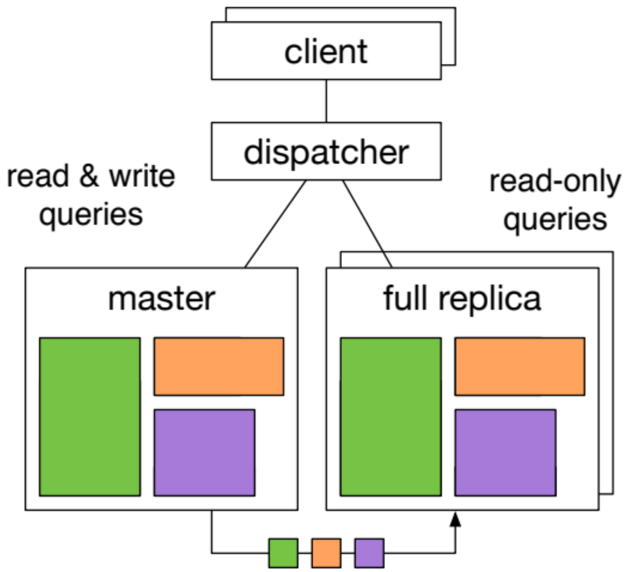
- The measurement infrastructure is there - let‘s run Hyrise on 480 cores

#2: Replication and Logging

- Analytical workloads can saturate single machine database servers

- Replica nodes can process read-only queries on snapshots of the master node without violating transactional consistency

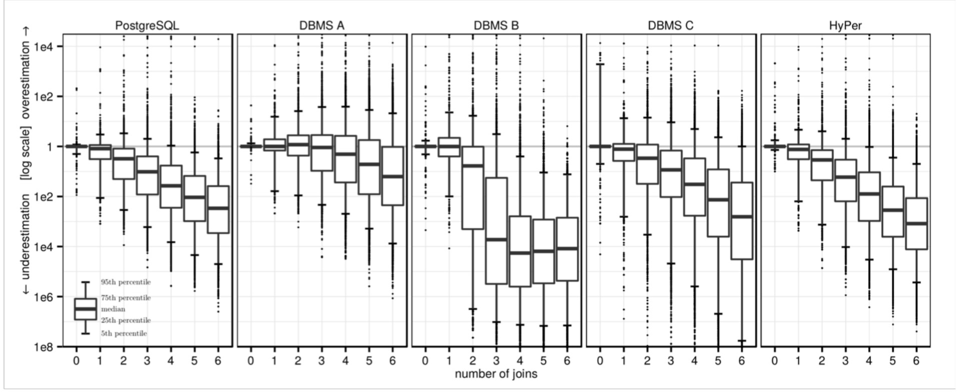
- To update the replica nodes, logging information can be used



HPI Hasso Plattner Institut

#3: Cardinality and Statistics

- Cardinality estimations are crucial for finding good query plans



- Currently, we have only histograms in Hyrise

- There are further statistics with different tradeoffs, i.e., Cost vs Estimation quality

- How do we select the appropriate one?

#4: TPC-DS

- So far, we have worked with the TPC-H benchmark
- The TPC-DS (Decision Support) benchmark uses more complex SQL queries (sometimes 100s of lines)
- By improving the optimizer and the operators, we hope to (again) improve the performance of Hyrise by orders of magnitude
- Also, we will get to implement still missing SQL features

Benchmark	prev. iter/s	new iter/s	change
TPC-H 1	0.827	6.21	+651%
TPC-H 2	0.028	133.346	+471876%
TPC-H 3	12.287	62.216	+406%
TPC-H 4	0.007	23.581	+353477%
TPC-H 5	5.903	50.192	+750%
TPC-H 6	34.571	192.292	+456%
TPC-H 7	0.883	21.315	+2313%
TPC-H 8	0.007	86.341	+1249787%
TPC-H 9	2.07	26.306	+1171%
TPC-H 10	10.964	33.873	+209%
TPC-H 11	27.335	220.087	+705%
TPC-H 12	5.229	63.781	+1120%
TPC-H 13	13.372	28.587	+114%
TPC-H 14	19.918	190.899	+858%
TPC-H 15	0.082	90.281	+110051%
TPC-H 16	3.1	69.887	+2155%
TPC-H 17	0.002	7.15	+386292%
TPC-H 18	0.012	19.353	+160190%
TPC-H 19	0.646	59.471	+9111%
TPC-H 20	0.007	4.112	+58781%
TPC-H 21	0.033	2.082	+6201%
TPC-H 22	0.465	124.38	+26636%
average			+129241%

Review Sprint 3

- Good Things First ☺
  - Most implementations work out of the box
    - 6 / 7 compile on macOS / Linux without changes
  - The number of tests range from 46 to 71 (Group 5)
    - One group’s tests were failing
    - One test case took more than 80 seconds to complete
  - Some shortcut opportunities for dictionaries are missed
  - Sometimes code could be made more understandable by adding a comment
  - Not all groups did format / lint their code before handing it in

```cpp
/**
 * This file contains the actual filter logic.
 * Every filter has its own struct.
 * The structs implement three major methods:
 *   check_value
 *     This method is used to compare plain values (i.e. in
 *   check_value_id
 *     This method is used to compare value ids (i.e. in Di
 *     Note that the comparison operator in use might be di
 *     This will be explained in detail later.
 *   begin_dictionary_column
 *     Since tables may have multiple chunks, and dictionar
 *     on a per-chunk basis, the value id of the filter val
 *     This method is used to look up the respective value
 *
 *
 * Optimizations
 *
 *  Sorted, dictionary-compressed columns offer a great way
 *  First, we use binary searches to look up the respective
 *  Second, depending on the operator, we either use a lowe
 *  The idea is to make use of the respective characteristi
 *    lower_bound
 *      Returns the first value in a vector that is greater
 *      Returns vector.end() if last value is strictly less
 *    upper_bound
 *      Returns the first value in a vector that is strictl
 *      Returns vector.end() if last value is less than or
 *
 * In conclusion, this offers the following possibilities:
 *  Operator | Applied Logic
 *  ---------|--------------
 *      >=   |  lb / >=
 *      >    |  ub / >=
 *      <    |  lb / <
 *      <=   |  ub / <
 *
 * As an example, let's look at the '>' operator.
 * We use upper_bound to search for the value in the dict.
 * We now have two options:
 *   1. The searched value is in the dict.
 *      upper_bound will return the value in the vector tha
 *      We can therefore include this value when we filter
 *      However, we do not include the searched value as th
 *   2. The searched value is not in the dict.
 *      upper_bound will return the value in the vector tha
 *      that is smaller than the searched value.
 *      This value must be greater than the searched value
 * Consequently, using the '>=' operator on the found value
 *
 * The main advantage we get out of this is that if the val
 * we do not have to spend time to decide that we actually
 * rather than the requested '>' operator.
 * The other operators mentioned above behave similarly.
 * The 'BETWEEN' operator is a combination of '>=' and '<='
 * '=' and '!=' use lower_bound and check if the returned v
 *
 * Additionally, the operators implement logic to recognize
 * For example, if there is an equal scan requested on a di
 * present in the dictionary, we can completely disregard t
 */

#pragma once

#include <limits>
#include <vector>

#include "types.hpp"

namespace opossum {

enum class ScanScope { ALL, SCAN, NONE };

template <typename T>
struct EqFilter {
  explicit EqFilter(const T &value) : _value(value) {}
```

```cpp
/**
 * Get the right operation for the given operator.
 * For most operations, there is the possibility that either all values or no values match.
 * We can catch and easily process these cases by looking at the value that is returned by lower_bound or
 * upper_bound.
 * Say we have the following dictionary vector:
 *
 * ValueID  | Value
 * -----------------
 *    0     |   B
 *    1     |   C
 *    2     |   D
 *    3     |   F
 *    4     |   G
 *
 * Then upper_bound (U) / lower_bound (L) return the following values:
 *
 *   Value |   U  |   L
 *  ---------------------
 *     A   |   0  |   0
 *     B   |   1  |   0
 *     D   |   3  |   2
 *     E   |   3  |   3
 *     G   | INV. |   4
 *     H   | INV. |  INV.
 *
 *  Then the table scan should return all values that match the following:
 *  (X = No values, A = All Values)
 *
 *  Operation |  A  |  B  |  D  |  E  |  G  |  H  |
 *  -----------------------------------------------
 *      =     |  X  | = 0 | = 2 |  X  | = 4 |  X  |
 *     !=     |  A  | !=0 | !=2 |  A  | !=4 |  A  |
 *      >     |  A  | > 0 | > 2 | > 2 |  X  |  X  |
 *      <     |  A  |  X  | < 2 | < 3 | < 4 |  X  |
 *     >=     |  A  |  A  | >=2 | >=3 | >=4 |  X  |
 *     <=     |  X  | < 1 | < 3 | < 3 |  A  |  X  |
 *
 * We then just pick the right method, according to the upper tables, and check for edge cases
 * (thus, an A or X in the table above). Afterwards, we iterate over the attribute vector and execute
 * the regarding method on it.
 */
std::pair<std::function<bool(ValueID)>, Match> get_dictionary_comparator(
    const std::string &op, const AllTypeVariant &allTypeVariant, const optional<AllTypeVariant> &allTypeVariant2,
    const DictionaryColumn<T> &column) const {
  const T value = type_cast<T>(allTypeVariant);

  // Calculate operation to check for valid entries.
  if (op == "=") {
    auto valueID = column.lower_bound(value);
    if (valueID != INVALID_VALUE_ID && column.value_by_value_id(valueID) == value) {
      return std::make_pair([valueID](ValueID entry) { return entry == valueID; }, Match::some);
    } else {
      // In case we found did not find a value id that matches the given value,
      // we can assume that no entries with this value exist -> return an empty position list.
      return std::make_pair(_none_match, Match::none);
    }
  } else if (op == "!=") {
```

Review Sprint 3

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

```
const std::shared_ptr<ValueSegment<T>>& p_segment =
  std::dynamic_pointer_cast<ValueSegment<T>>(base_segment);
```

```
const auto value_column =
    dynamic_cast<ValueSegment<T>*>(base_segment.get());
```

Review Sprint 3

```
...
else if (scan_type == ScanType::OpNotEquals &&
         search_value_lower_bound == search_value_upper_bound) {
  for (ChunkOffset chunk_offset{0}; chunk_offset < values.size();
       chunk_offset++) {
    pos_list->push_back(RowID{chunk_id, chunk_offset});
  }
  return;
}
```

Review Sprint 3

```
switch (attribute_vector->width()) {
    case sizeof(uint8_t): {
      const auto fitted_attribute_vector = std::static_pointer_cast<
                                           const FittedAttributeVector<
                                           uint8_t>>(attribute_vector);
      DebugAssert(fitted_attribute_vector != nullptr, "cast failed");
```

```
auto row_id = RowID();
row_id.chunk_offset = i;
row_id.chunk_id = current_chunk_id;
pos_list->emplace_back(std::move(row_id));
```

mrzzzrm 5 hours ago

Although the compiler *might* speed this up, constructing the RowID and only writing its values afterwards is possibly slow. Also, `std::move(row_id)` will have no effect, RowID doesn't have data for which moving is more efficient than copying.

This is both shorter and likely faster: `pos_list->emplace_back(chunk_id, chunk_offset);`

Reply...

Hasso Plattner Institut

Review Sprint 3

```
// Scanning reference columns
for (const auto& pos : *ref_pos_list) {
  const auto& value_seg = std::dynamic_pointer_cast<ValueSegment<T>>(
                           ref_seg);
  const auto& dict_seg = std::dynamic_pointer_cast<DictionarySegment<T>
                          >(ref_seg);


  if (value_seg != nullptr) {
    const auto& value = value_seg->values()[chunk_offset];
    match = _matches_search_value(value);
  } else if (dict_seg != nullptr) {
    const auto& value = dict_seg->get(chunk_offset);
    match = _matches_search_value(value);
  }
  …
}
```

Review Sprint 3

```
+            // Add entry to pos list using the compare function.
+            for (ChunkOffset chunk_offset = 0; chunk_offset < attribute_vector->size(); ++chunk_offset) {
+              if (is_greater_than_whole_chunk || is_unequal_whole_chunk ||
```

mflueggen a day ago

If you pull `is_greater_than_whole_chunk` and `is_unequal_whole_chunk` out of the loop and just add all the chunk_offsets to pos_list you might gain a performance benefit.

mrzzzrm 4 hours ago

If you already know (from `is_greater_than_whole_chunk || is_unequal_whole_chunk`) that the entire Chunk matches, why check this again for each line?

Reply...

```
+                  compare_function(attribute_vector->get(chunk_offset), value_id)) {
+                pos_list->emplace_back(RowID({ChunkID{chunk_index}, ChunkOffset{chunk_offset}}));
+              }
+            }
```

Hasso Plattner Institut

Review Sprint 3

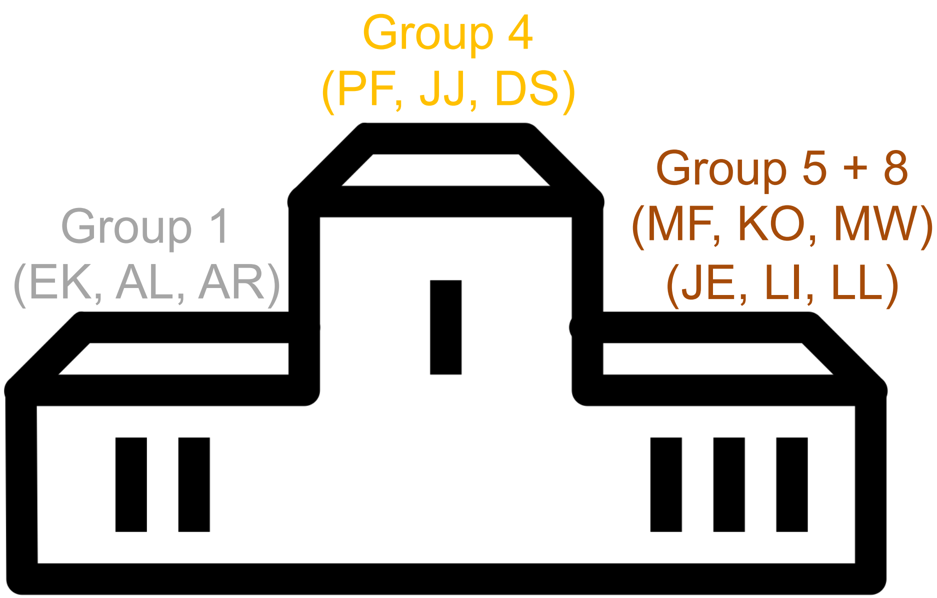
```
template <typename S>
std::function<bool(S, S)> get_comparator(ScanType scanType) {
  std::function<bool(S, S)> result;
    switch (scanType) {
      case ScanType::OpEquals: {
        result = [](S left, S right) { return left == right; };
        break;
      }
      ...
    }
  }
}
```

```
template<typename S, typename Comparator>
void scan(...) {
  for (const auto& value : segment) {
    if (Comparator{}(value, search_value) {
      // ... the value matches
    }
  }
}
scan<T, std::equals<T>>()
```

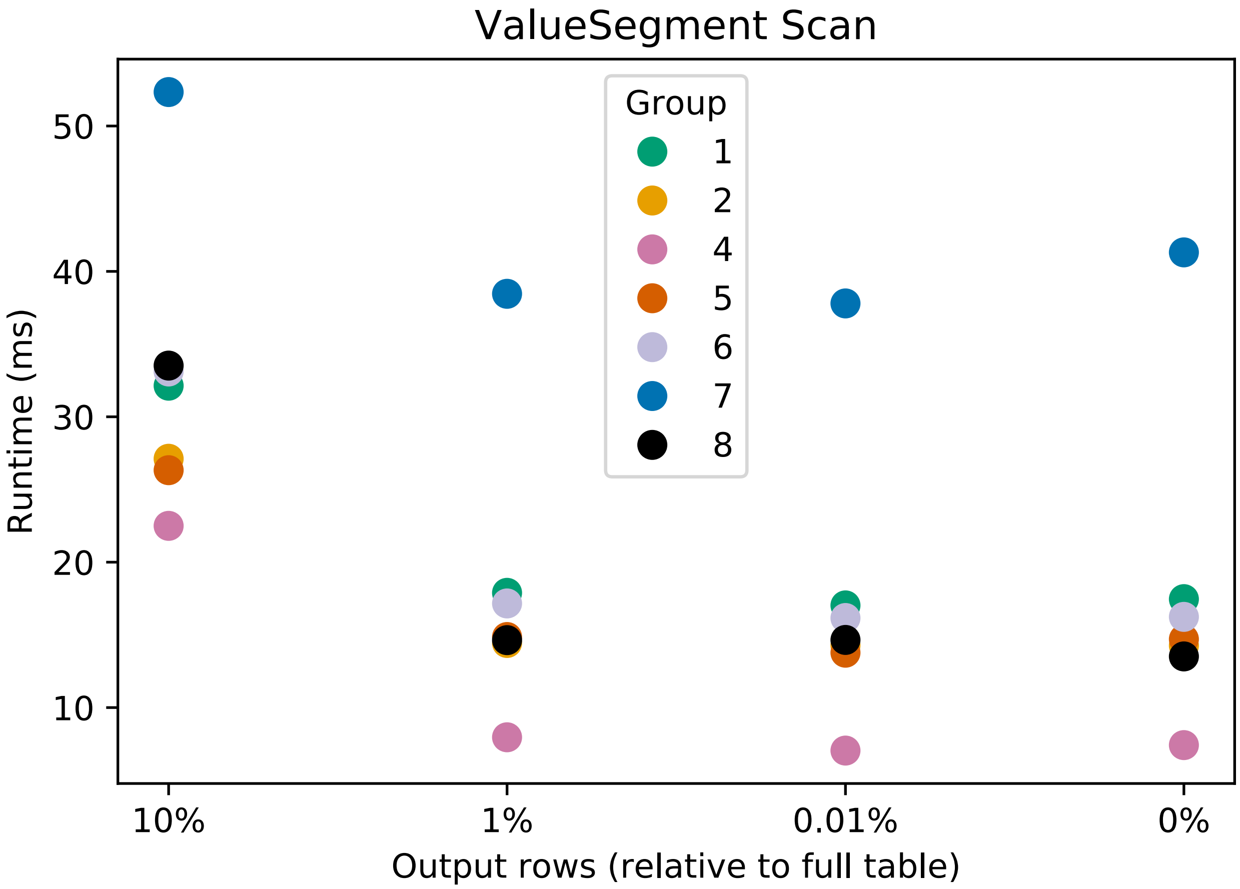
Performance Challenge

- One table of 10.000.000 records
  - Chunk sizes: 100K, 1M, 9.9M
  - 3 `integer` columns
  - Varying uniqueness -> different selectivities
- Several experiments on
  - ValueSegments
  - DictionarySegments
  - Dictionary + ReferenceSegments
- 100 executions per experiment including cache flushing
- All working solutions produced the same results

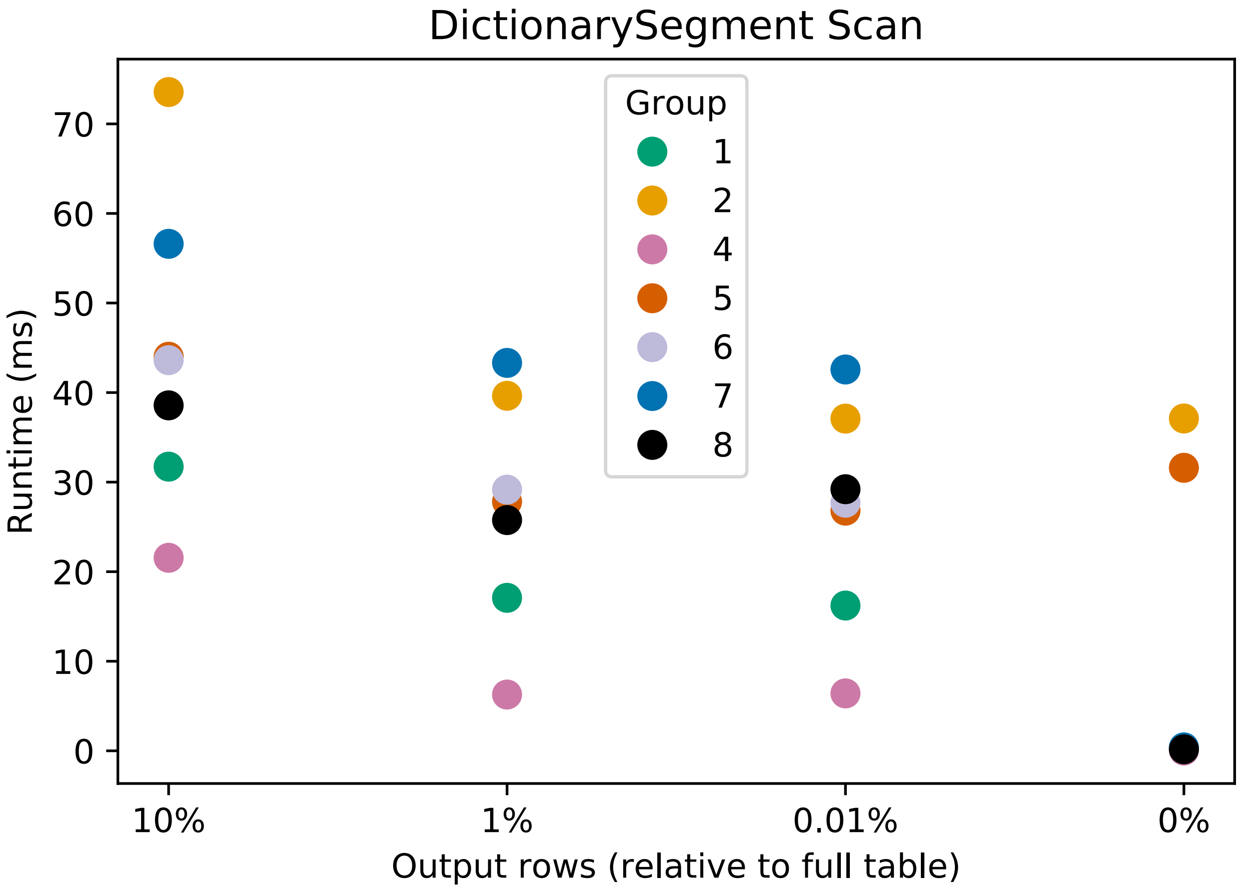
Ranking



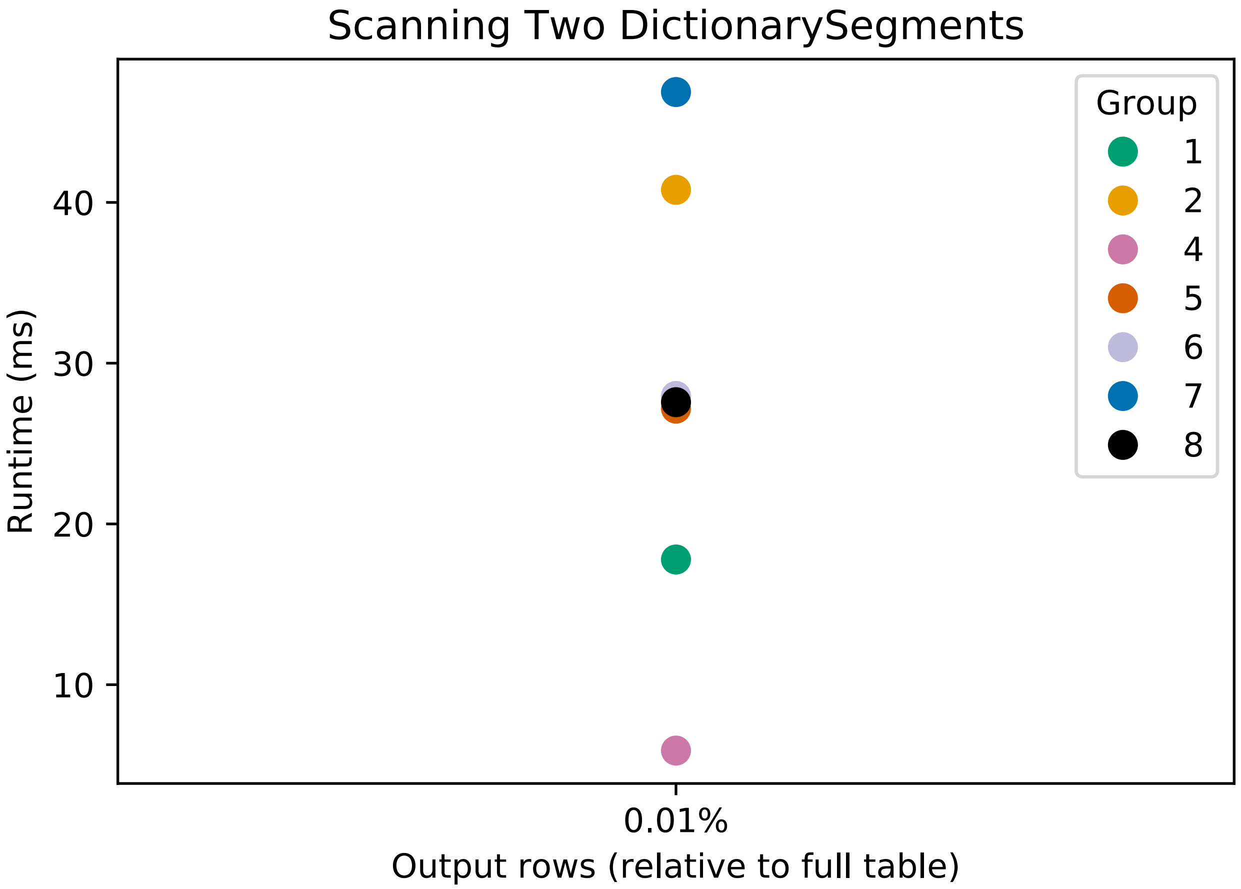
ValueSegment



DictionarySegment



ReferenceSegment



Hasso Plattner Institut