Build your own Database

Week 9

Assorted Findings

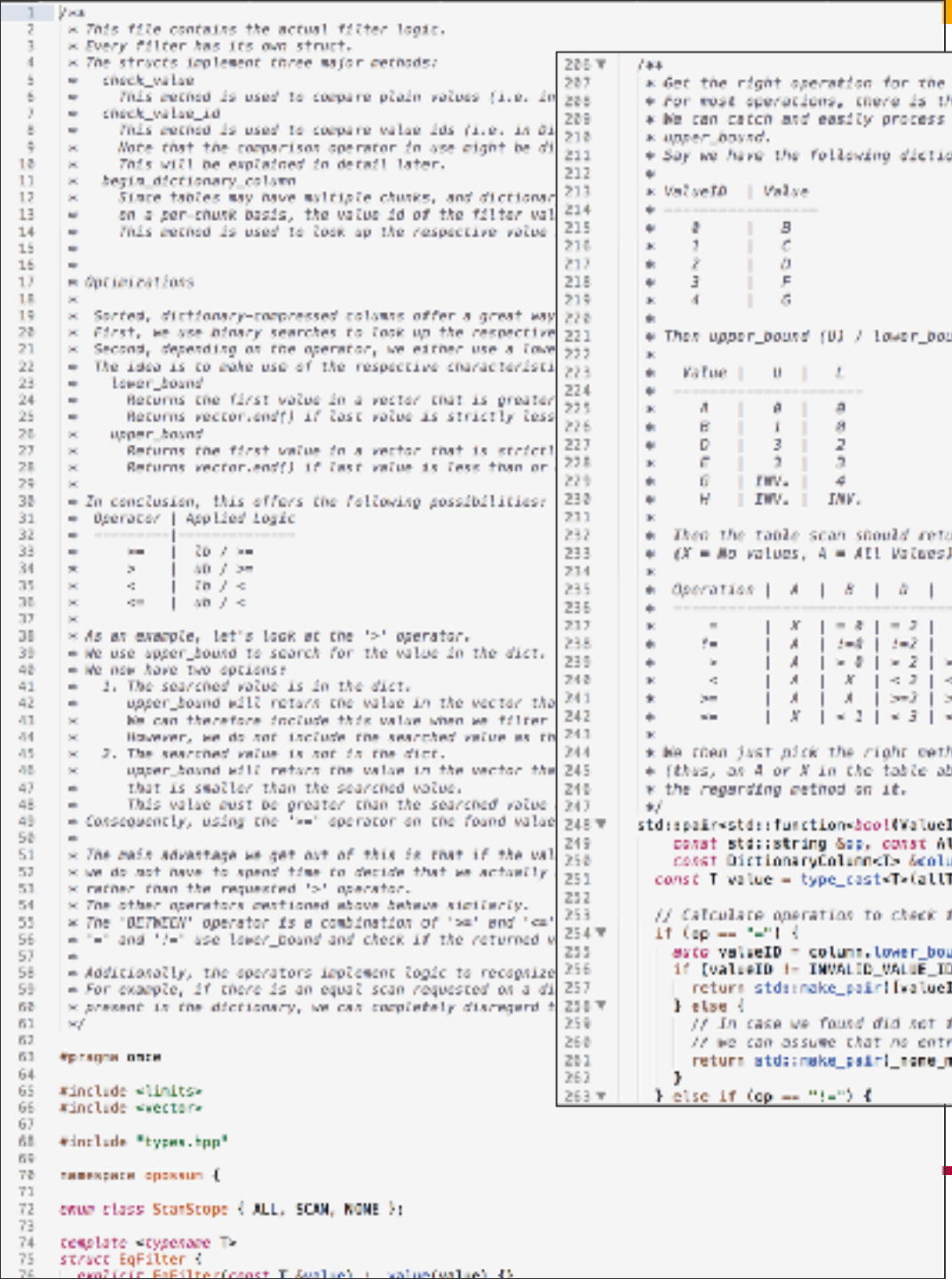
```
auto op = TableScanOp<T>(_operator, _search_value, _search_value_2);
// [...]
if (op(boost::get<T>((*column)[offset]))) {
    _pos_list->push_back(pos);
}
```

```
void handle_reference_column(ReferenceColumn& column,
std::shared_ptr<
                              ColumnVisitableContext> context) {
  // [...]
  for (size_t virtual_id = 0; virtual_id < r_column.size();
       ++virtual_id) {
    if (_visit_operator(type_cast<T>(r_column[virtual_id]), _value,
        _value2)) {
      casted_context->appendPosition((*pos_list)[virtual_id]);
    }
  }
}
```

Assorted Findings

```
void TableScan::execute() {
  const size_t column_id = _input_left->column_id_by_name(_filter_column_name);

  const std::string type_string =  input_left->column_type(column_id);
  auto column_instance =
opossum::ValueColumn>(ty

  const auto& instance =
  const auto& type_id =
  const auto& type_hash

  if (type_hash == typei
    _out = TableScanImpl
_value2).execute();
  } else if (type_hash =
    _out = TableScanImpl
_value2).execute();
  } else if {
    // [...]
  } else {
    throw std::runtime_e
  }
}
```

```
void TableScan::execute() {
  _input_left = _in->get_output();
  _filter_column_id = _input_left->column_id_by_name(
                        _filter_column_name);
  auto column_type = _input_left->column_type(_filter_column_id);

  if (column_type == "int") {
    typed_execute<int32_t>();
  } else if (column_type == "long") {
    typed_execute<int64_t>();
  } else if (column_type == "float") {
    typed_execute<float>();
  } else if (column_type == "double") {
    typed_execute<double>();
  } else if (column_type == "string") {
    typed_execute<std::string>();
  } else {
    Fail("Unimplemented column_type");
  }
}
```



```
/**
 * 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 re
 *   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 ScanType { ALL, SCAN, NONE };

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

```
  /**
   * 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
   *     F    |   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 | F | G | H |
   *  ----------------------------------
   *     =      | X | = B | = D | X | = G | X |
   *     !=     | A | !=B | !=D | A | !=G | A |
   *     >      | A | > B | > D | X | X | X |
   *     <      | A | X | < D | < D | < G | X |
   *     >=     | A | A | >=D | >=D | >=G | X |
   *     <=     | X | < D | < D | A | X | X |
   *
   * We then just pick the right method, according to the upper tables, and check for edge cases
   * (thus, as 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 == "!=") {
```

Assorted Findings

```
template <typename T>
class ValueColumn : public BaseColumn {
  // [...]
  template <typename Operator>
  PosList typed_table_scan(const T& filter_value, const ChunkID
                           chunk_id, const PosList* in_pos_list =
                           nullptr) const;

  PosList table_scan_between(const T& filter_value, const T&
                             filter_value_2, const ChunkID chunk_id,
                             const PosList* in_pos_list) const;
};
```

Assorted Findings

```
void Projection::execute() {
  // [...]
  for (size_t chunk = 0; chunk < _table_in->chunk_count(); ++chunk) {
    // [...]
    for (size_t col = 0; col < _table_in->col_count(); ++col) {
      if (std::count(_columns.begin(), _columns.end(), _table_in->
          column_name(col)) > 0) {
        // [...]
        _table_out->get_chunk(chunk).add_column(_table_in->get_chunk(
                                  chunk).get_column(col));
      }
    }
  }
}
```

Zeitplan

Woche	Phase	Mittwoch	Donnerstag
1	Sprint 1	19.10.2016	20.10.2016
2		26.10.2016	27.10.2016
3	Sprint 2	2.11.2016	3.11.2016
4		9.11.2016	10.11.2016
5	Sprint 3	16.11.2016	17.11.2016
6		23.11.2016	24.11.2016
7	Gruppenphase	30.11.2016	1.12.2016
8		7.12.2016	8.12.2016
9		**14.12.2016**	15.12.2016
10		4.1.2017	5.1.2017
11		11.1.2017	12.1.2017
12		18.1.2017	19.1.2017
13		25.1.2017	26.1.2017
14		1.2.2017	2.2.2017
15		8.2.2017	9.2.2017