Build you own Database
The Opossum Blueprint

WS 16/17 :: Sprint 3

Enterprise Platform and
Integration Concepts
Fachgebiet | Hasso-Plattner-Institut
Universität Potsdam

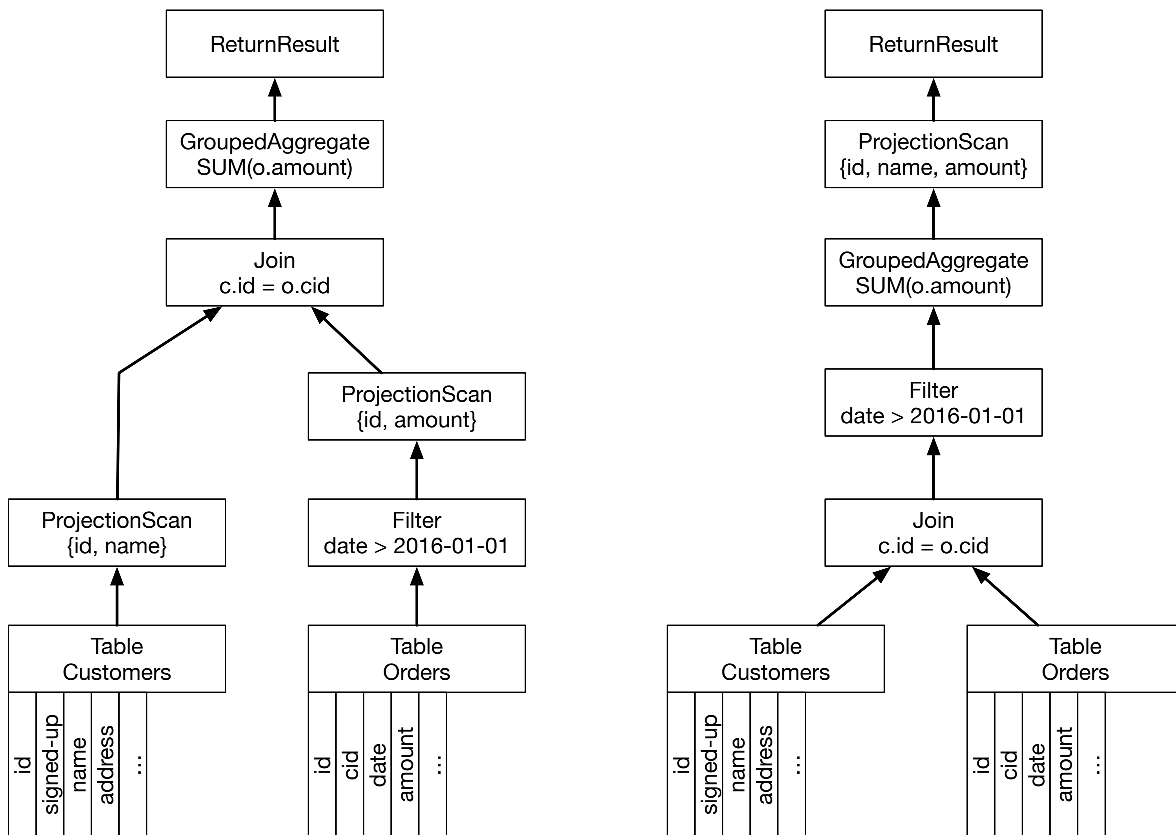
Operator Concept

In the third sprint, you will implement the TableScan operator – one of the most fundamental operators. Of course, the TableScan is not the only operator that we will have in Opossum. Thus, it makes sense to first talk about the operator concept in general.

For executing a query, databases traditionally use something called a query plan or operator tree. Let us look at the operator tree for an example query:

```
SELECT c.id, c.name, SUM(o.amount) FROM customers c, orders o WHERE c.id = o.cid
AND o.date > '2016-01-01' GROUP BY c.id, c.name;
```

This query should give us the id, name, and total amount of orders in this year[1] for every customer. Note how it does not say anything about how the database gets to that result. The two following query plans both have the same result:



One of them, however, is likely to be significantly faster. Selecting a fast query plan out of many potential query plans is the job of the query optimizer. Because we do not yet have such an optimizer, we will build our query plans by hand.

[1] No, you should not have an aggregated order amount stored in your database but calculate in on the fly. Bear with me just for the sake of the example, will you?

AbstractOperator

As you can see, an operator has up to two inputs, can have an output, and usually also parameters. We model this using the `AbstractOperator` interface:

```
class AbstractOperator {
 public:
  AbstractOperator(const std::shared_ptr<const AbstractOperator> left
                   = nullptr,
                   const std::shared_ptr<const AbstractOperator> right
                   = nullptr);

  // copying a operator is not allowed
  AbstractOperator(AbstractOperator const&) = delete;
  AbstractOperator &operator=(const AbstractOperator&) = delete;

  AbstractOperator(AbstractOperator&&) = default;
  AbstractOperator &operator=(AbstractOperator &&) = default;

  // abstract method to actually execute the operator
  // execute and get_output are split into two methods to allow for
  // easier asynchronous execution
  virtual void execute() = 0;

  // returns the result of the operator
  virtual std::shared_ptr<const Table> get_output() const = 0;

  virtual const std::string name() const = 0;

  // returns the number of input tables, range of values is [0, 2]
  virtual uint8_t num_in_tables() const = 0;

  // returns the number of output tables, range of values is [0, 1]
  virtual uint8_t num_out_tables() const = 0;
};
```

This way, we can easily chain multiple operators by passing the first operator as a parameter to the second one:

```
auto scan = std::make_shared<Scan>(...);
auto sort = std::make_shared<Sort>(scan, "col_x");
```

The first operator usually is the GetTable operator, which takes no input operators and outputs a table stored in the StorageManager.

As an example, we have provided you with the Print operator. It takes one table as input, forwards it as an output and prints the table to an output stream (by default, std::cout).

ReferenceColumn

Now that we know how to pass input to our operators, the question is what the output looks like. For performance reasons, we do not want to materialize (i.e., copy) each column, especially if it is unaffected. Instead, we want to use indirection. By using references to an existing table, we can get around copying the actual values.

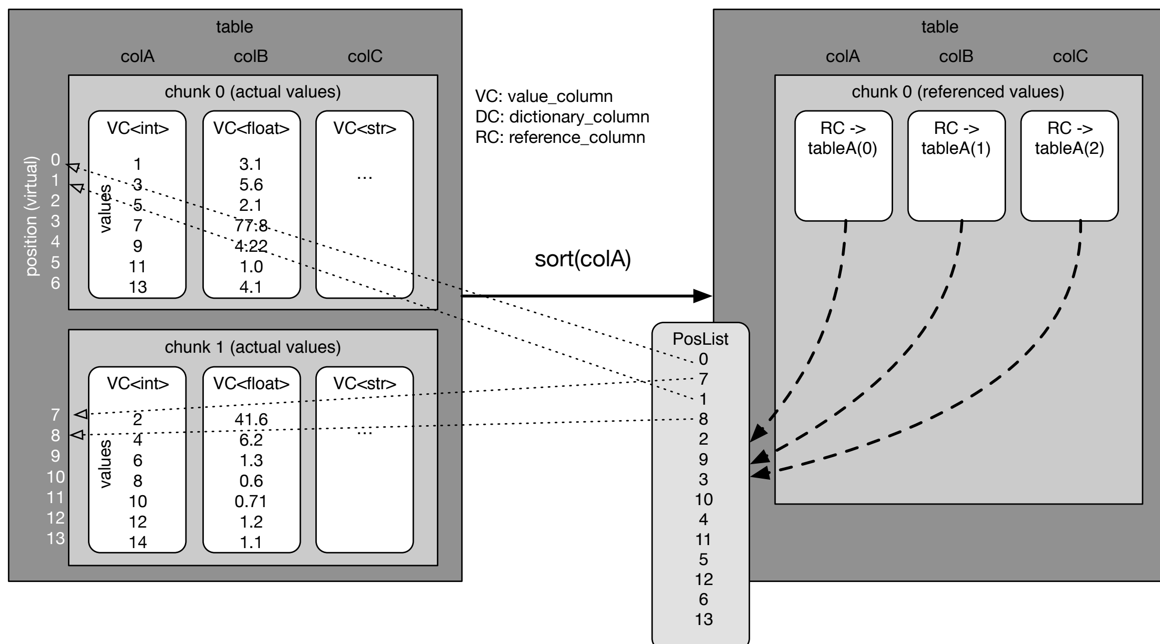
We do this by adding a third column type, which we call ReferenceColumn. A ReferenceColumn allows us to reference **certain positions** in a **different column** within a **different table**.

```
// ReferenceColumn is a specific column type that stores all its values
// as position list of a referenced column
class ReferenceColumn : public BaseColumn {
 public:
  // creates a reference column
  // the parameters specify the positions and the referenced column
  ReferenceColumn(const std::shared_ptr<Table> referenced_table, const
                  size_t referenced_column_id,
                  const std::shared_ptr<PosList> pos);

  virtual const AllTypeVariant operator[](const size_t i) const;

  virtual void append(const AllTypeVariant &) { throw std::logic_error(
                        "ReferenceColumn is immutable"); }

  virtual size_t size() const;

  // returns the list of referenced positions
  const std::shared_ptr<const PosList> pos_list() const { return _pos_list; }

  // returns the table that is referenced
  const std::shared_ptr<const Table> referenced_table() const { return
                                                   _referenced_table; }

  // returns the column id that is referenced
  size_t referenced_column_id() const;
};
```

Internally, the ReferenceColumn has three pieces of information:

1. A shared pointer to the table that is referenced. Using a shared pointer here makes sure that the table that holds the actual values does not disappear while it is still being referenced.
2. The column_id of the column that is referenced.
3. A shared pointer to a PosList (which, internally, is just an std::vector<RowID>). We will talk about why it has to be a shared pointer in a second.

To reiterate, a ReferenceColumn does not refer to a single chunk within that table but to the entire table. The necessity of this should become more clear when we think about the result of a sort operation:

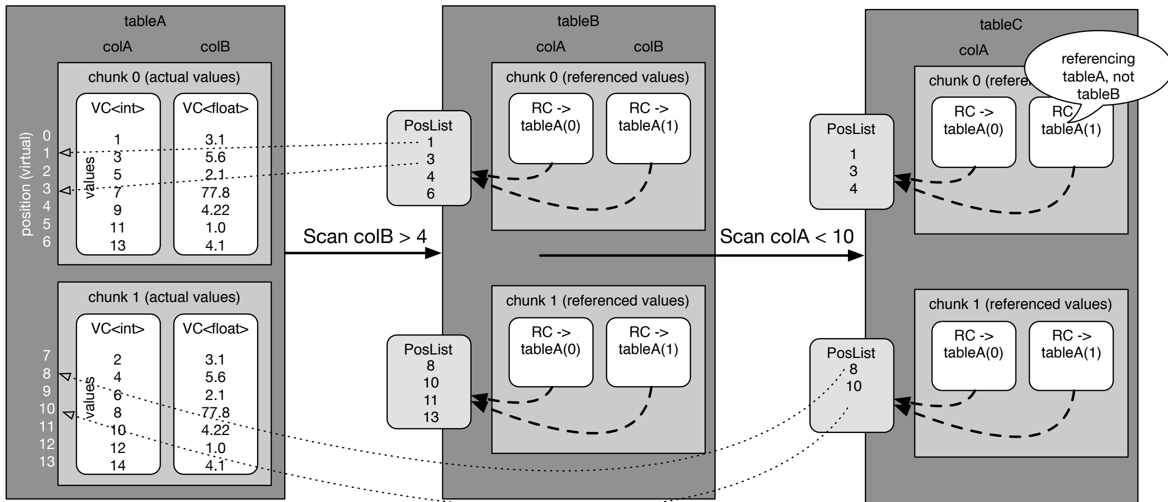


Referencing the positions in alternate order would not be possible here if a ReferenceColumn could only point to a single chunk. There is one shortcoming of this approach: Because a ReferenceColumn cannot point to values in different tables (e.g., because two tables were concatenated using a union), we could not use the same approach for the sort operator if chunk1 was part of a different table. In this case, we will resort to materializing the inputs. Since this problem only happens with the sort operator, which is usually the last operator in a query plan, this materialization is not harmful for performance, as it would have to occur anyway.

Another thing that you can see in the visualization above is that the three ReferenceColumns created by the sort operator point to the same PosList. It allows us to save both the time to generate multiple PosLists and reduce the memory footprint. This is possible because the row ids that the three columns point to are the same. There are other cases where sharing a PosList is not possible – for example if the columns point to different tables.

Because we will not use dictionary compression on a ReferenceColumn, there is no need to split large columns into chunks. As such, the ReferenceColumn does not obey the maximum_chunk_size setting of a table.

While indirection is helpful, we want to avoid excessive indirection. If we had ReferenceColumns pointing to ReferenceColumns pointing to ReferenceColumns, each access would have to go through three levels of indirection (plus virtual method calls). This kills our performance. Thus, we define that ReferenceColumns can only refer to positions in a ValueColumn or a DictionaryColumn:



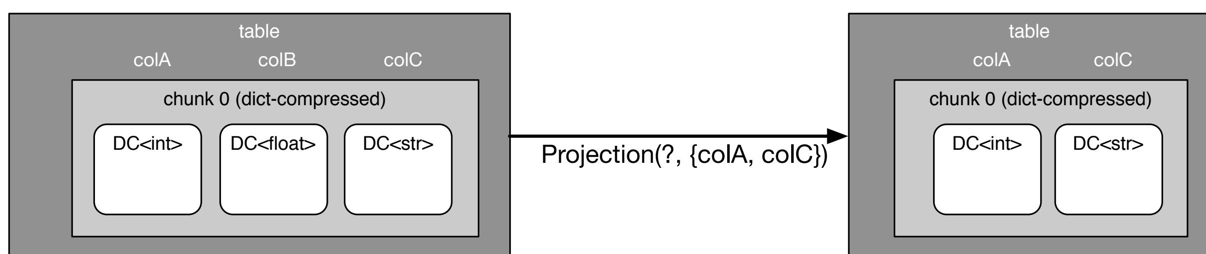
GetTable

Let's start implementing things. We begin with the GetTable operator that takes a table name, looks up the associated table in the StorageManager and returns it.

```
explicit GetTable(const std::string &name);
```

Projection

The next operator that we want to implement is the projection. It takes exactly one table and a list of strings that act as column filters:




```
Projection(const std::shared_ptr<const AbstractOperator> in, const
           std::vector<std::string> &columns);
```

In generating the position list, it will include all positions that were included in the input. Pay attention to what happens if you input a ValueColumn or a ReferenceColumn that was already filtered.

TableScan

Now it becomes interesting. A TableScan takes an input table, and filters a given column by comparing all values to a given operator:

```
TableScan(const std::shared_ptr<AbstractOperator> in, const std::
          string &filter_column_name, const std::string &operator,
          const AllTypeVariant value, const optional<AllTypeVariant>
          value2 = nullopt);
```

Most of the actual implementation is left to you. Below, we want to give you some tips on what we found helpful. Keep in mind that the interface is the only contract that your operator has to fulfill. Internal implementations can vary. Also, we do not define the structure of the output. You could either have a single chunk with ReferenceColumns that point to the found values or multiple chunks.

TableScanImpl

Remember that we do not want to use `operator[]` on columns. It is solely there for debugging and testing purposes, but its performance (due to the virtual method calls involved and the use of AllTypeVariant) prohibits its use in operators. As a result, you will have to get the ValueColumns' value vector or the DictionaryColumns' attribute vector and the corresponding dictionary when you want to scan the values. These are templated to match the type stored in the column. The TableScan operator, however, is not templated.

In our implementation, we solved this problem by writing a class internal to TableScan:

```
class TableScan : public AbstractOperator {
protected:
  template <typename T>
  class TableScanImpl;
};
```

This internal class contains the actual implementation. TableScan is just a wrapper around it that creates the Impl class by using `make_unique_by_column_type` and forwards `execute()` and `get_output()`. In TableScanImpl, we can then get the search value as a T by using the `type_cast<T>` method.

Checking for a column's type

At various points in your implementation, you will need to check the type of a BaseColumn. There are two ways to do it. First, you can rely on RTTI information and use `std::dynamic_pointer_cast<DictionaryColumn>(b)`. If `b` can be casted to a DictionaryColumn, the pointer cast returns such a pointer. If `b` cannot be casted, it returns `nullptr`.

The second option is to use the visitor pattern. In this case, the TableScan calls the visit method on the BaseColumn. It then uses dynamic polymorphism (i.e., a virtual method call) to call the visit method on the appropriate class. We gave you an implementation of the visitor pattern in `column_visitable.hpp`. By deriving from `ColumnVisitable`, a class can call visit on a column. Optionally, it can pass a context that contains information needed in the `handle_x` methods.

Checking for the operator type

We pass the operator as a string. Supported values are "=", "!=", "<", "<=", ">", ">=", and "BETWEEN". At some point, we need a mapping from the string to something the compiler can work with – for example an enum. It is acceptable to do this with a long if statement:

```
if (op == "=") {
    [...]
} else if (op == "!=") {
    [...]
```

For performance reasons, you do not want to execute this block each time you make a comparison. Instead, do it somewhere outside of the loop(s).

Doing the actual comparison

Comparing values stored in a ValueColumn is trivial once you have a `T value_to_be_compared_to` and the value vector `const std::vector<T>&`. For dictionary columns, this is more complicated – at least if you care about performance. The trivial solution would be to decompress every value id and then compare it to the search value. In fact, you should try implementing this first before you do the more efficient approach outlined below.

When scanning a DictionaryColumn, we can make use of the fact that a total order on the values also reflects a total order on the dictionary. Also, there is an entry for a given value in the dictionary if (if and only if) the value is included in the uncompressed representation of the column. In other words:

Let x be a value in the column and x' its representation as a value id.
(Lemma 1) $a \leq b \leftrightarrow a' \leq b'$

Let V be the list of uncompressed values in the column and D the column's dictionary
(Lemma 2) $\forall_x : x \in V \leftrightarrow x \in D$

For our table scan, this means that we can perform each of the comparison operators listed above by comparing value ids, not values. We do this by first retrieving the value id from the dictionary using lower_ or upper_bound (whatever is appropriate) and then comparing the value ids in the attribute vector with the value id from the dictionary. As an example in pseudo code:

```
scan(AttributeVector av, Dictionary d, Operator op, T value) {
  op = "<";
  ValueId search_vid = d.lower_bound(value);
  for(col_vid : av) {
    if(col_vid < search_vid) emit(row_id);
  }
}
```

Splitting a RowId into ChunkId and ChunkOffset

Most likely, you will need to convert row ids into chunk ids and chunk offsets. For this, we added two methods in the table:

```
// returns the number of the chunk and the position in the chunk for
// a given row
std::pair<ChunkID, ChunkOffset> locate_row(RowID row) const;

// calculates the row id from a given chunk and the chunk offset
RowID calculate_row_id(ChunkID chunk, ChunkOffset offset) const;
```

tableA

RowID	ChunkID	ChunkOffset	colA	colB
			chunk 0 (actual values)	
			VC<int> (values)	VC<float>
0	0	0	1	3.1
1	0	1	3	5.6
2	0	2	5	2.1
3	0	3	7	77.8
4	0	4	9	4.22
5	0	5	11	1.0
6	0	6	13	4.1
			chunk 1 (actual values)	
			VC<int> (values)	VC<float>
7	1	0	2	3.1
8	1	1	4	5.6
9	1	2	6	2.1
10	1	3	8	77.8
11	1	4	10	4.22
12	1	5	12	1.0
13	1	6	14	4.1

Please note that due to the introduction of reference columns, we cannot be sure anymore that all chunks have `maximum_chunk_size` entries. If you relied on that in your `row_count` implementation, you will need to adjust it accordingly.

For now, a trivial implementation of these methods (iterating over the chunks) is fine. At a later point, we will describe a more efficient way of doing this by limiting the possible values of `maximum_chunk_size` to powers of two and using bit shifts.

Performance Challenge

Again, we will measure the performance of your TableScan to find the most efficient solution. This will be executed on various types of input columns with varying length. You can use optimizations such as SSE, but no multithreading. If you believe that you can achieve a better performance by modifying the interface, please let us know and we can discuss this. The focus, however, is on writing readable and maintainable code.

Submission Instructions

By now, you should know the drill. Because you will write more lines of code in this sprint than before and because the actual implementation of your TableScan is mostly left to you, it becomes more important to write good comments. Our implementation has more than 50 lines of comments. When writing comments, think about readers who have not implemented the TableScan themselves.

Also, remember to make sure that you have adequate test coverage. Not all combinations of Value-, Dictionary-, and ReferenceColumn are covered in the tests given out.

Your submission is due Wednesday, November 30, 11:59PM CET.