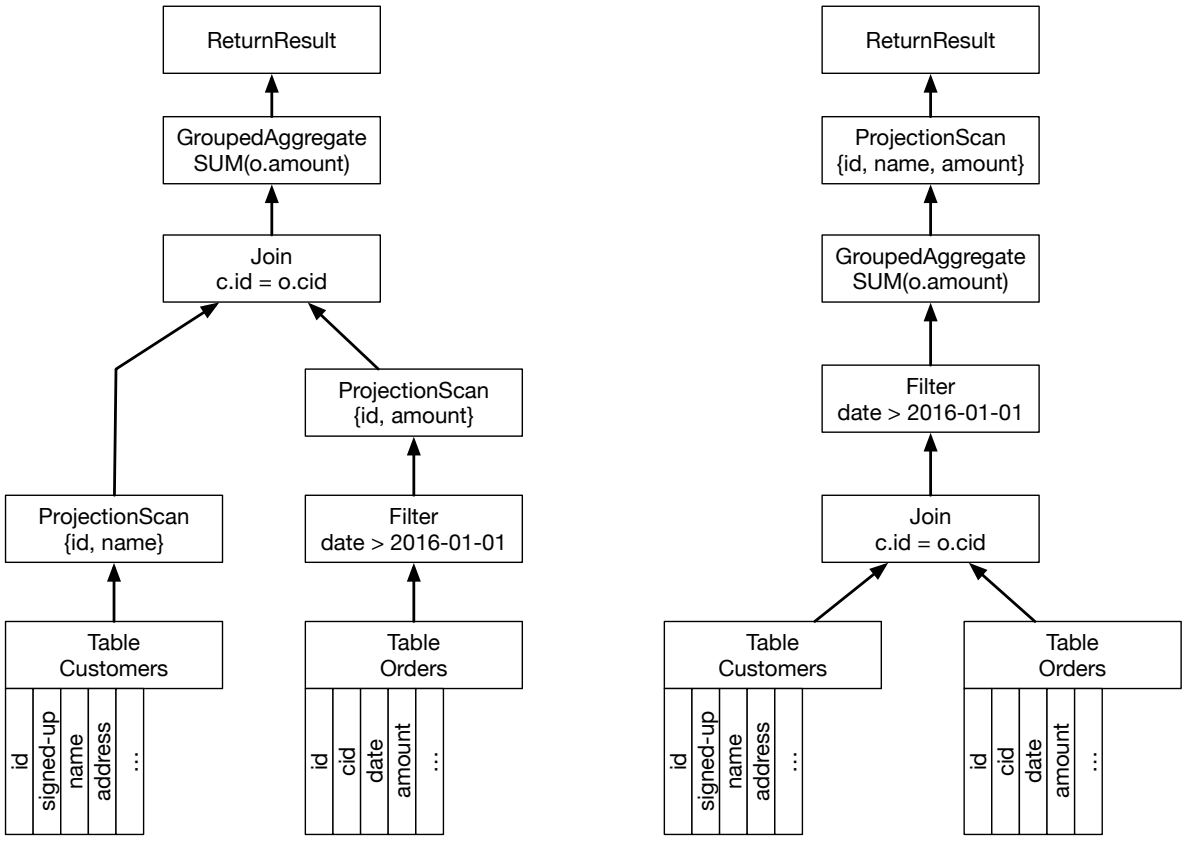
Operator Concept

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 have in a DBMS. 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 gives us the id, name, and total amount of orders since 2016[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 an optimizer, we will build our query plans by hand. Later this term, we will talk

[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?

about how the Hyrise optimizer deals with this. Helping the optimizer with generating more efficient query plans will also be part of some group projects.

AbstractOperator

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

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

  // we need to explicitly set the move constructor to default when
  // we overwrite the copy constructor
  AbstractOperator(AbstractOperator&&) = default;
  AbstractOperator& operator=(AbstractOperator&&) = default;

  void execute();

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

  // Get the input operators.
  std::shared_ptr<const AbstractOperator> input_left() const;
  std::shared_ptr<const AbstractOperator> input_right() const;

protected:
  // abstract method to actually execute the operator
  // execute and get_output are split into two methods to allow for easier
  // asynchronous execution
  virtual std::shared_ptr<const Table> _on_execute() = 0;

  std::shared_ptr<const Table> _input_table_left() const;
  std::shared_ptr<const Table> _input_table_right() const;

  // Shared pointers to input operators, can be nullptr.
  std::shared_ptr<const AbstractOperator> _input_left;
  std::shared_ptr<const AbstractOperator> _input_right;

  // Is nullptr until the operator is executed
  std::shared_ptr<const Table> _output;
};
```

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, ColumnID{3});
```

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

As an example, we have provided you with the Print operator. It takes one table as input, prints the table to an output stream (by default, std::cout), and forwards it as an output. This way, it can be placed anywhere in the query plan.

The columns that the operators should work on are given as ColumnIDs, which denote the position of the referenced column in the input table. We prefer this over using column names because it makes name resolution in case of aliases much easier.

An operator only has an output table if it has already been executed. This is so that you can first create the operators and then execute them (or have the scheduler execute them).

Furthermore, we decided that operators must not return empty chunks (except, of course, if there is no result at all). This is because these empty chunks will not be relevant for future operators. As a consequence, we do not assume that a table scan returns the same number of output chunks as it has as input chunks.

ReferenceSegment

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 segment, 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.

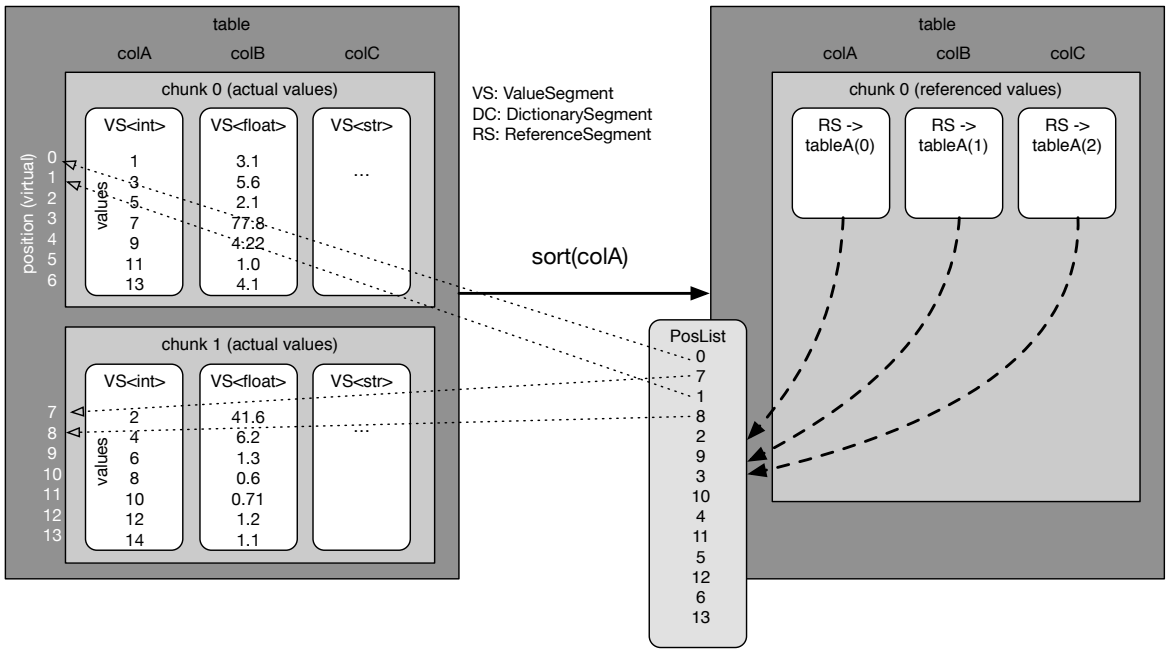
We do this by adding a third segment type, which we call ReferenceSegment. A ReferenceSegment allows us to reference **certain positions** in a **different segment** within a **different table**.

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

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

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

  size_t size() const override;

  const std::shared_ptr<const PosList> pos_list() const;
  const std::shared_ptr<const Table> referenced_table() const;

  ColumnID referenced_column_id() const;
};
```

Internally, the ReferenceSegment 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 ReferenceSegment 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:

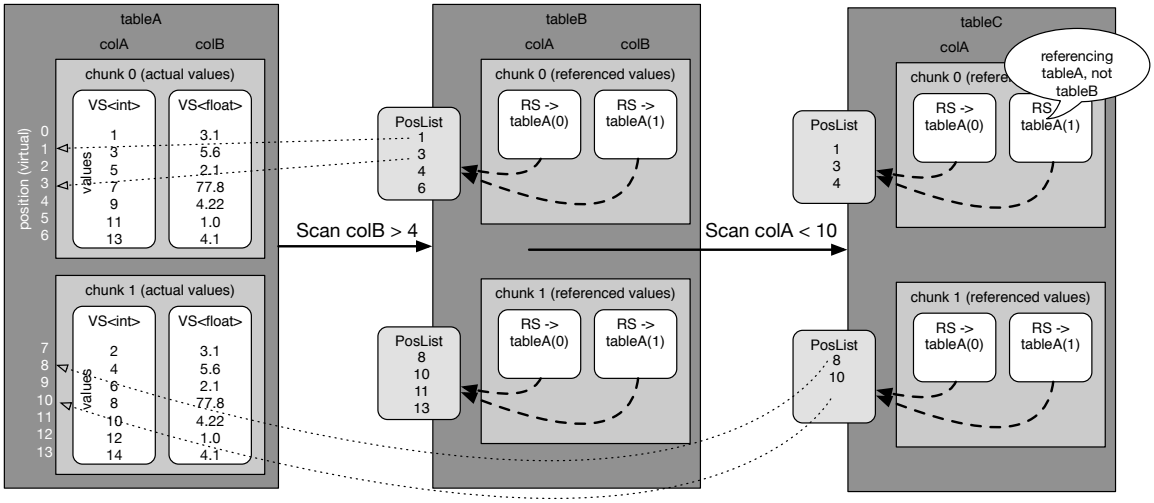


Referencing the positions in alternate order would not be possible here if a ReferenceSegment could only point to a single chunk. There is one shortcoming of this approach: Because a ReferenceSegment cannot point to values in different tables (e.g., because two tables were concatenated using a union), we could not use the same approach for the sort operator if the first chunk 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 ReferenceSegments 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 RowIDs 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.

Remember how we mentioned that you cannot assume that the size of a table can be calculated by the number of chunks times the maximum chunk size? This becomes clear here, as the size of a ReferenceSegment that comes from a scan can be anywhere between zero and the size of the entire table. Because we will not use dictionary compression on a ReferenceSegment, there is no need to split large columns into chunks. As such, the ReferenceSegment does not obey the *maximum_chunk_size* setting of a table.

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



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);
```

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<const AbstractOperator> in, ColumnID column_id,
          const ScanType scan_type, const AllTypeVariant search_value);
```

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 ReferenceSegments that point to the found values or one output chunk per input chunk. Keep in mind that we do not produce empty chunks, unless the result is empty, in which case we produce a single empty chunk.

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 ValueSegments' value vector or the DictionarySegments' 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 operator implementation, you will need to check the type of a BaseSegment. You can rely on **R**un-**T**ime **T**ype **I**nformation and use, e.g., `std::dynamic_pointer_cast<DictionarySegment>(b)`. If `b` can be casted to a DictionarySegment, the pointer cast returns such a pointer. If `b` cannot be casted, it returns `nullptr`.

Doing the actual comparison

In this sprint, we will not deal with varying data types. You do not have to support comparing ints with floats. However, if the types do not match, your implementation should notice this and throw an exception.

Comparing values stored in a ValueSegment 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 DictionarySegment, 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 ValueID.
(Lemma 1) $x \leq y \leftrightarrow x' \leq y'$

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 ValueIDs in the attribute vector with the ValueID from the dictionary. As an example[2] 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);
  }
}
```

Performance Challenge

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 SIMD, 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

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.

Your submission is due Tuesday, November 27, 11:59 PM CET. Please send an email with the SHA hash of your commit (as shown in the diagram in the slides of week two and four) to Markus and Jan.

[2] In this example, we use short variable names so that we can fit everything on one page. For your code, please use speaking names.