Overview

In this first sprint, you will implement Opossum's basic storage classes, i.e., segments, chunks, and tables. We provide some code that will help you with this and test cases that you can use to verify your implementation.

Preliminary Information

This first project serves two purposes: First, it allows you to get a better idea of what this seminar will be about. Second, it should give you an idea of the level of C++ programming that we will be expecting in this class. The discussed concepts will be challenging for some students who have not worked with C++ for a while. If you manage to get through this sprint, you should be able to work your into more advanced C++. Once we have built the foundation for our database, we will focus more and more on database architectures and concepts.

We would like you to work on the projects in groups. Remember that this project is a part of the *Leistungserfassungsprozess*. Discussing abstract concepts with other students is ok, sharing (parts of) an implementation is not. Please use a github repository for your development.

In the first three sprints, we will work on a code base where we have provided some boilerplate code. Once we are in the group phase, we will work on the publicly available code base. This way, we can make use of the work that has been done in the seminar and the master's projects, such as the SQL interface and a good selection of operators. For the first three steps, however, please refrain from referring to the Hyrise implementation. While it might make your life easier now, you are cheating yourself out of an opportunity to learn the concepts needed to succeed in the group phase.

Coding Guidelines

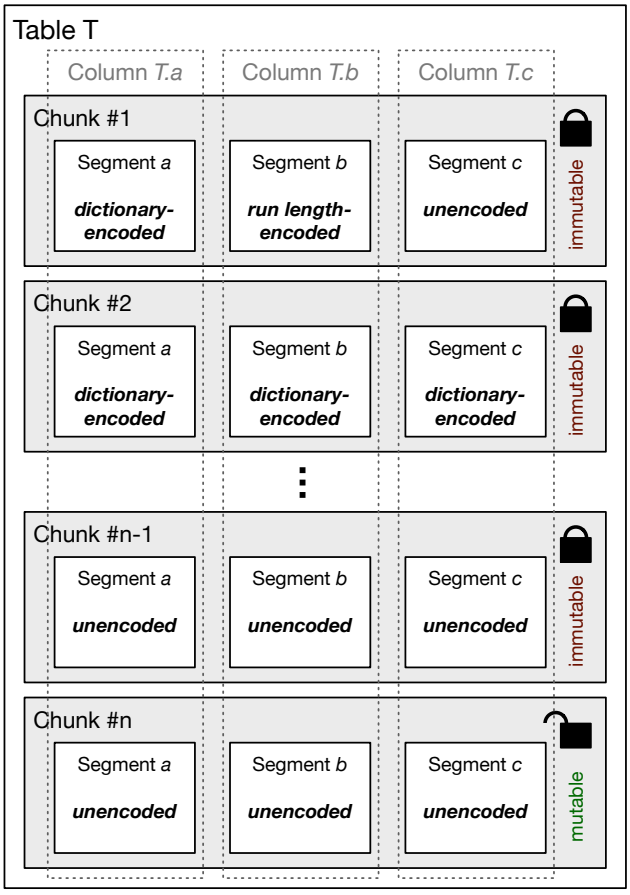
We wrote down some of the principles we follow with Opossum in CONTRIBUTING.md. Please read that file and try to follow the guidelines. This is especially important with regards to the new C++11-style memory management. We do not use `new / malloc` anymore, because these are prone to create leaks. More about this later.

In cases where we have provided a full interface to a class, it should not be necessary to add any public methods or change signatures unless this is explicitly stated in the task. Of course, you may add private methods at will if this helps keeping your code concise. In some places, however, we might have missed specifications such as removing the copy constructor or using `const`. If you believe that this is the case, please let us know.

Remember to comment your code in places where you consider it helpful for an outside reader. This does not mean that every line has to have its comment. Additionally, make sure that you reach acceptable test coverage. While we provide some tests, these do not yet cover all edge cases.

The Opossum Table Model

In Opossum, every table is horizontally partitioned into a number of *chunks.* This partitioning will become helpful later this term when we look into dictionary compression and other techniques.



Within each chunk, the actual values are stored in so called segments. For each column in the table, the chunk has a segment. In turn, the segments across all chunks form a column. The segment is responsible for the actual representation of the values. Here, we use a `ValueSegment`, which stores its entries directly in an `std::vector`. Later, we will also encounter other column types, such as `ReferenceSegment` and `DictionarySegment`.

The `StorageManager` maintains a mapping from table names to table objects.

Step 0: Remember to sign up to Piazza

Please sign up to our Piazza class at:
https://piazza.com/hpi.uni-potsdam.de/fall2019/dyod
We will use Piazza to make important announcement and as a discussion platform for questions outside of our regular meetings.

Step 1: Set up your build environment

Prerequisites: We have tested the project on OS X 10.15 and Ubuntu 19.04[1]. If possible, please use one of these environments for your work. Others might work, but are not supported.

[1] We use a docker image with Ubuntu 19.04 for our CI server. You can obtain it with: `docker pull hyrise/opossum-ci`

We already have some code prepared for you. Check out the git repository at

```
git@github.com:hyrise/DYOD_WS1920.git
```

and read the `README.md`. This will automatically install a tool for generating Makefiles (cmake), a current version of gcc/clang (needed because we use the latest C++17 features, fresh from the oven), and boost::hana (unrelated to the database with a similar name).

To make sure that everything is set up correctly, compile Opossum using

```
./install.sh
mkdir cmake-build-debug
cd cmake-build-debug
cmake ..
make -j4 hyriseTest
cd ..
./cmake-build-debug/hyriseTest
```

This should show two passing tests. All other tests are currently disabled, because you have not yet written the code that they require.

We have a number of other make targets. `make hyrisePlayground` builds the `playground.cpp` found in the `bin/` folder. You can use this playground to experiment with new features without having to use the test framework.

After adding new files, add them to `src/(lib|test)/CMakeLists.txt` so that they become part of the build process.

To keep the code base maintainable and ensure code style guidelines, we offer easy ways to lint and format the source code. The folder `scripts` contains files that simplify linting and formatting. In addition, `make hyriseSanitizers` creates a binary that utilizes llvm's AddressSanitizer[2]and UndefinedBehaviorSanitizer[3]. Make sure to use these tools.

Before you commit, it is a good idea to do the following:

```
./scripts/format.sh
./scripts/lint.sh
```

Make sure that there are no linting errors in your code.

[2] https://clang.llvm.org/docs/AddressSanitizer.html
[3] https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

Step 2: ValueSegment

Covered C++ concepts: Templates, deleted copy constructors, const

As mentioned above, the `ValueSegment` simply stores all its values in an `std::vector`[4]. If you lookup the reference for the vector, you will find that it requires you to define the stored data type, for example `std::vector<int>`. Make yourself familiar with this usage of C++ templates. We will need templates for the `ValueSegment`, which will have to hold different Opossum data types.

To simplify handling different data types, we have given you a class `AllTypeVariant` that can store any of Opossum's data types. You can use it like this:

```
AllTypeVariant foo = 4; // now storing an int

AllTypeVariant giveFloat() { return 4.3f; }
AllTypeVariant giveInt() { return 5; }

foo = giveFloat();
std::cout << foo << ", " << giveInt() << std::endl;
float bar = type_cast<float>(giveInt());
```

Its implementation is in `all_type_variant.hpp`. You do not have to understand the definition of the class for now.

A caveat of this is that an `AllTypeVariant` always uses the maximum size of all data types – meaning that a `char` has the same size as a `long`. Obviously, we want to save space in our database. Furthermore, accessing the contained value is more expensive than accessing a trivial data type. As a result, we must not store `AllTypeVariant`s in our vector. Instead, we will use the actual data type as a template parameter for our `ValueSegment` class.

Now, start implementing the `ValueSegment<T>` in `value_segment.hpp` by adding a (non-public) vector and by writing the following (public) methods:

[4] http://en.cppreference.com/w/cpp/container/vector

```
// default constructor
ValueSegment();

// return the value at a certain position
AllTypeVariant operator[](const ChunkOffset chunk_offset)
const final;

// add a value to the end
void append(const AllTypeVariant& val) final;

// return the number of entries
size_t size() const final;

// Return all values
const std::vector<T>& values() const;
```

Once you are done with this, you can enable the tests in `value_column_test.cpp`. Check that all tests pass before you continue.

Step 3: Chunks

Covered C++ concepts: Managed pointers, inheritance

Let's move on to implement the `Chunk` class. The only job of a chunk is to hold pointers to all of its segments. Since C++11, we can use smart pointers (i.e., `std::shared_ptr<int>` and `std::unique_ptr<int>`) instead of raw pointers (`int*`). Lookup the advantages and the usage of these smart pointers if you are unfamiliar with them. We do not use any old-style allocations (malloc or new) in Opossum.

An easy way to store all segments within a chunk would be to have an

```
std::vector<std::shared_ptr<ValueSegment>>
```

Unfortunately, ValueSegment is not a complete type, because we have templated it above. A correct way to use the vector would be

```
std::vector<std::shared_ptr<ValueSegment<int>>>
```

but that would mean that all segments are of the int type.

To avoid this problem, we created a non-templated super class `BaseSegment` from which `ValueSegment` inherits. This way, you can add different types of ValueSegment to a chunk:

```
chunk.add_segment(std::make_shared<ValueSegment<int>>());
chunk.add_segment(std::make_shared<ValueSegment<float>>())
;
```

Next, create the chunk class. In addition to the non-public vector holding the columns, you will need the following public methods:

```
// creates an empty chunk
Chunk();

// adds a segment to the "right" of the chunk
void add_segment(std::shared_ptr<BaseSegment> column);

// returns the number of columns
uint16_t column_count() const;

// returns the size (i.e., the number of rows)
uint32_t size() const;

// adds a new row, given as a list of values, to the chunk
// implemented in step 4
void append(std::vector<AllTypeVariant> values);
```

To make types more strict and achieve better data type semantics, we decided to use Boost's strong typedefs[5].

```
// from types.hpp
STRONG_TYPEDEF(uint32_t, ChunkID);
STRONG_TYPEDEF(uint16_t, ColumnID);

// returns the column at a given position
std::shared_ptr<BaseSegment> get_segment(ColumnID
column_id) const;

// usage example
chunk.get_segment(ColumnID{1});
```

You can now enable the `AddColumnToChunk` test in chunk_test.cpp.

[5] http://www.boost.org/doc/libs/1_63_0/libs/serialization/doc/strong_typedef.html

Step 4: Appending to a chunk

Covered C++ concepts: debug checks and release builds

Now that we have a chunk that can store our data, we need a method to insert it. Because of our `AllTypeVariant`, we could do something like this:

```
void append_to_segment(int column, const AllTypeVariant
value);
```

However, inserting into a long table becomes tedious and error-prone:

```
chunk.append_to_segment(0, 2);
chunk.append_to_segment(1, 5.3f);
chunk.append_to_segment(1, "Hallo Welt");
// d'oh - copy paste error
```

This would be much nicer:

```
chunk.append({2, 5.3f, "Hallo Welt"});
```

For this, we implement the method

```
// adds a new row, given as a list of values, to the chunk
void append(const std::vector <const AllTypeVariant>&
values);
```

Your goal is to implement the method so that the first value is inserted into the first segment, the second value into the second segment, and so on.

To make sure that the method is used correctly, add a check if the number of passed arguments matches the number of columns. For performance reasons, we only want this check executed during development, not when we measure the performance. We defined a macro `DebugAssert(check, msg)` that tests if the constant `IS_DEBUG` is set and only then performs the check, printing a message if it fails. Because the value of `IS_DEBUG` is known at compile time, the debug blocks will be removed by the compiler for the release build. Make sure that the check is not executed if you build with

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

You can now enable the remaining tests in chunk_test.cpp.

Step 5: Table

Covered C++ concepts: Type dispatch

While we now have chunks that hold segments of different types, we do not yet have any notion of column names or a way to group multiple chunks to a table. For this, we now implement the table.

When a table is created, an optional parameter defines the maximum size of a chunk. By default, this is `std::numeric_limits<ChunkOffset>::max() - 1`[6]. The maximum chunk size is stored in the table and cannot be changed. Inserts are always done into the last chunk, checking if this chunk has already reached its maximum size. If this is the case, a new chunk is created. To make things easier, creating a table also creates the first chunk.

In addition to the list of chunks, the table also holds the column names and types, both as strings.

```
explicit Table(const size_t chunk_size =
std::numeric_limits<ChunkOffset>::max() - 1);

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

// returns the number of columns
uint16_t column_count() const;

// returns the number of rows
uint64_t row_count() const;

// returns the number of chunks
ChunkID chunk_count() const;

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

// returns the column name of the nth column
const std::string &column_name(size_t column_id) const;
```

[6] -1 because we reserve the last possible values for NULL values, which we do not cover here.

```
// returns the column type of the nth column
const std::string &column_type(size_t column_id) const;

// returns the column with the given name
ColumnID column_id_by_name(const std::string &column_name)
const;

// return the maximum chunk size
uint32_t max_chunk_size() const;

// adds a column to the end, i.e., right, of the table
void add_column(const std::string &name, const std::string
&type;

// inserts a row at the end of the table
void append(std::vector<AllTypeVariant> values);
```

Adding a column

When adding a new column to a table, the name and the type have to be stored in the appropriate places so that the access methods (e.g., `column_name`) work properly. We also want to add a ValueSegment in which values can be stored.

You will notice that chunk.add_segment expects a pointer to a BaseSegment, for example a ValueSegment<int>. So how can we create a ValueSegment<int> if we only have the desired column type as a string?

The straight forward way would be to use a list of if-statements (remember – C++ does not allow for a switch on a string):

```
std::shared_ptr<BaseSegment> column;
if(type == "int") {
     column = std::make_shared<ValueSegment<int>>();
} else if(type == "float") {
     return std::make_shared<ValueSegment<float>>();
} ...
```

This comes with two issues: First, it requires us to list all possible data types, making it difficult to add new ones. Second, this code will likely be required in other places as well, leading to code duplication.

Instead, we provide you with a method in resolve_type.hpp called `make_shared_by_data_type`. It works as follows:

```
auto segment = make_shared_by_data_type<BaseSegment,
ValueSegment>(type);
```

For now, you may treat the implementation of that method as a black box of dark template magic.

Appending values

The next method, append, should be easy to implement. You will have to pass the list of values to the last chunk in the table. Remember to first create a new chunk if the last chunk has reached its maximum capacity.

Once you are done, you can enable the tests in `table_test.cpp`.

Step 6: StorageManager

Of course, we do not want to hand out pointers to a Table object. Instead, we want to refer to tables by name. Maintaining a mapping from table names to tables is the job of the StorageManager. For now, it does nothing else.

Because the StorageManager is a single point of entry, we want to implement it as a singleton. Look up singleton patterns in C++. For implementing the get method, you will only need two lines and no additional members in the class.

```
public:
  static StorageManager &get();

  void add_table(const std::string& name,
std::shared_ptr<Table> tp);
  void drop_table(const std::string& name);
  std::shared_ptr<Table> get_table(const std::string
&name) const;
  bool has_table(const std::string& name) const;
  std::vector<std::string> table_names() const;
  void print(std::ostream& out = std::cout) const;
  static void reset();
```

After implementing all methods, you can enable the remaining tests.

Development & Submission instructions

Create a branch in your repository named *sprint1*. For your final submission, please file a pull request from *sprint1* to the *master* branch in your repository. Also, please email us (Markus.Dreseler and Jan.Kossmann) the URL of this pull request and the commit ID (i.e., the SHA-1 hash) so that we know which version you consider final. **Deadline: 29 October 23:59h MEZ.**