Build you own Database
The Opossum Blueprint

WS 16/17 :: Sprint 1

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

HPI

Overview

In this first sprint, you will implement Opposum's basic storage classes, i.e., columns, 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 description is handed out in advance of the first class meeting. Doing so serves two purposes: First, it allows you to get a better idea of what this class will be about. Second, it should give you an idea of the level of C++ programming that we will be expecting in this class. While the discussed concepts might be challenging for some students who have not worked with C++ for a while, it will *not* get more difficult from a programming perspective later on. Instead, 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. We will discuss group formation during the first class. You can start working on the project alone, but we would like every group to submit only one implementation. 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 *private* git repository (github.com or gitlab.hpi.de) for your development.

Coding Guidelines

We wrote down some of the principles we follow with Opossum in docs/guidelines.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.

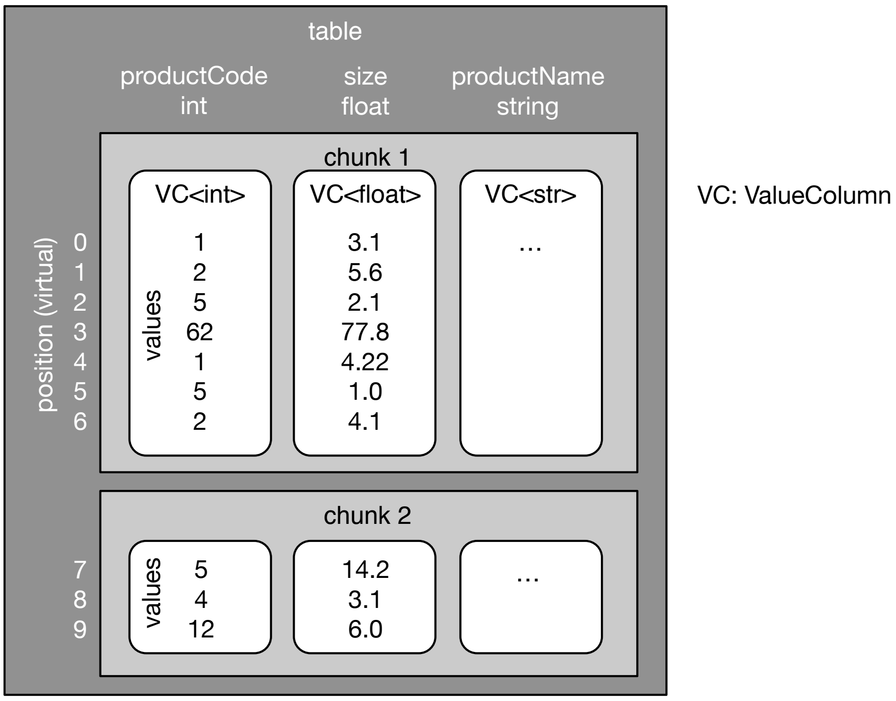
In cases where we have provided a full interface to a class, it should not be necessary to add any public methods or change signatures unless this is explicitly stated in the task. Of course, you may add private methods at will if this helps keeping your code concise. In some places, however, we might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

Within each chunk, the actual values are stored column by column. The column is responsible for the actual representation of the values. Here, we use a `ValueColumn`, which stores its entries directly in an `std::vector`. Later, we will also encounter other column types, such as `ReferenceColumn` and `DictionaryCompressedColumn`.



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

Step 1: Set up your build environment

Prerequisites: We have tested the project on OS X 10.11, 10.12 and Ubuntu 16.10 If possible, please use one of these environments for your work, using a virtual machine where necessary. Others might work, but are not supported.

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

```
git@gitlab.hpi.de:BYOD_WS16/OpossumDB.git
```

and run the `install.sh` script. This will automatically install a tool for generating Makefiles (premake4), a current version of gcc (needed because we use the latest C++17 features, fresh from the oven – clang works as well), and boost::hana (unrelated to the database with a similar name).

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

```
premake4
make -j test
```

This should show a single, passing test. 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 server` builds the Opossum server (which does not exist, because you have not yet written it). `make playground` 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. Finally, `make clean` removes all build files and the Makefile, after which you have to call `premake4` again.

Because of the way the build environment is set up, you have to call premake after adding new files. While this might appear weird, we believe that the overall build environment is easier to work with than regular Makefiles.

Step 2: ValueColumn

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

As mentioned above, the `ValueColumn` simply stores all its values in an `std::vector`[1]. 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 `ValueColumn`, 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 `types.hpp`. Later this term, we might come back to that file – for now, you will not have to deal with it.

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

[1] http://www.cplusplus.com/reference/vector/vector/

space in our database. As a result, we must not store `AllTypeVariants` in our vector. Instead, we will use the actual data type as a template parameter for our `ValueColumn` class.

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

```
// default constructor
ValueColumn();

// return the value at a certain position
const AllTypeVariant operator[](const size_t i) const;

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

// return the number of entries
size_t size() 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 its columns. 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. Remember that we do not use any old-style allocations (malloc or new) in Opossum.

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

```
std::vector<std::shared_ptr<ValueColumn>>
```

Unfortunately, ValueColumn 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<ValueColumn<int>>>
```

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

To avoid this problem, create a non-templated super class `BaseColumn` from which `ValueColumn` inherits. This way, you can add different types of ValueColumn to a chunk:

```
chunk.add_column(std::make_shared<ValueColumn<int>>());
chunk.add_column(std::make_shared<ValueColumn<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();

// copying a chunk is not allowed
Chunk(const Chunk &) = delete;

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

// adds a column to the "right" of the chunk
void add_column(std::shared_ptr<BaseColumn> column);

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

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

// returns the column at a given position
std::shared_ptr<BaseColumn> get_column(size_t column_id)
const;
```

Note how we deleted the copy constructor above. This is because copying a chunk should not be needed anywhere in our database. If we did not delete the copy constructor, this would be possible:

```
Chunk a;
[...]
Chunk b = a; // now we have a copy of the same chunk
```

Especially in places where copying is expensive, this can lead to a high cost of creating the copy. Furthermore, it can lead to unexpected situations when a chunk exists twice. Instead, we want to pass around references or smart pointers to a chunk. To make sure that we do not accidentally copy a chunk, we explicitly delete the copy constructor. Keep this in mind and delete the copy constructor in all classes where you consider it reasonable to do so.

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

Step 4: Appending to a chunk

Covered C++ concepts: initializer lists, 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 appendToColumn(int column, const AllTypeVariant
value);
```

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

```
chunk.appendToColumn(0, 2);
chunk.appendToColumn(1, 5.3f);
chunk.appendToColumn(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::initializer_list<const
AllTypeVariant> values);
```

If you haven't worked with them yet, look up how initializer lists work. Your goal is to implement the method so that the first value is inserted into the first column, the second value into the second column, 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 `IS_DEBUG` that evaluates to true if we are in a debug build and to false in a release build. 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

```
make -j test config=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 columns 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. A chunk size of 0, which is the default value, specifies an unlimited chunk size. 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.

```
// creates a table
// the parameter specifies the maximum chunk size, i.e.,
// partition size
// default (0) is an unlimited size
explicit Table(const size_t chunk_size = 0);

// copying a table is not allowed
Table(Table const &) = delete;

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

// returns the number of columns
size_t col_count() const;

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

// returns the number of chunks
size_t chunk_count() const;
```

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

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

// 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
size_t column_id_by_name(const std::string &column_name)
const;

// return the maximum chunk size
size_t 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, bool create_value_column = true);

// inserts a row at the end of the table
void append(std::initializer_list<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. In the default case, `create_value_column==true`, we also want to add a ValueColumn in which values can be stored. The other case, in which we do not need a ValueColumn, will become important in a later sprint.

You will notice that chunk.add_column expects a pointer to a BaseColumn, for example a ValueColumn<int>. So how can we create a ValueColumn<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<BaseColumn> column;
if(type == "int") {
     column = std::make_shared<ValueColumn<int>>();
} else if(type == "float") {
     return std::make_shared<ValueColumn<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 types.hpp called `make_shared_by_column_type`. It works as follows:

```
auto column = make_shared_by_column_type<BaseColumn,
ValueColumn>(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 maximum chunk size of the table is set and the last chunk already has that size.

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;
  StorageManager(StorageManager const &) = delete;
  StorageManager(StorageManager &&) = delete;

 protected:
  StorageManager() {}
```

After implementing all methods, you can enable the tests in our storage_manager_test.cpp.

Submission instructions

For your final submission, please give read access to the instructors, whose git accounts you can find below. Also, please email us the commit ID (i.e., the SHA-1 hash) so that we know which version you consider final.

GitLab	GitHub
Martin.Boissier	
Markus.Dreseler	mdsl
Stefan.Klauck	klauck
Jan.Kossmann	Bensk1