Overview

In the second sprint, you will implement dictionary-encoded chunks for OpossumDB. In contrast to Hyrise v1, HANA, and SanssouciDB, we are not going to have a large single (always dictionary-encoded) partition. Each chunk in OpossumDB starts uncompressed and will eventually be dictionary-compressed at a later point in time. When the chunk is full and thus immutable (we'll discuss ways to invalidate tuples later), its segments get compressed.

Dictionary-Encoded Segments

A dictionary-encoded segment in OpossumDB consists of two main data structures:

- **The attribute vector:** an `std::vector<uint*_t>` storing references into the dictionary. Its length must always be the same as the chunk's length; each entry is an index into the dictionary where the actual value is stored. As a first step, you can implement it for 64-bit integers. Later in this sprint, you should support multiple lengths.
- **The dictionary:** an `std::vector<T>` storing the actual distinct values of the segment in sorted order.

For now, the actual compression is initiated by the `Table::compress_chunk` method. That method takes a chunk id and compresses all value segments in that chunk so that they are dictionary segments. As a result, it is not possible for chunks to contain both value and dictionary-encoded segments. You should create a new empty chunk before starting the compression, add the new dictionary-encoded segments to the chunk and in the end put the new segments into place by exchanging the complete chunk. Keep in mind that database systems are usually accessed by multiple users simultaneously. Others might access a chunk while you are compressing it. Therefore, exchanging uncompressed and compressed chunks should consider concurrent accesses.

Additionally, the dictionary segment has a number of methods that we will use in the next sprint (e.g., lower bound). These behave similar to the methods that the C++ standard library implements (if you don't know about lower bound, consult the reference[1]). If the search value is not found in the dictionary, they return the special value id `INVALID_VALUE_ID`.

Once you have implemented this, you can enable the tests in dictionary_segment_test.cpp. Note that these do not cover all methods and should be extended.

Variable-Width Attribute Vector

If everything works with fixed-size (i.e., 64-bit) entries in the attribute vector, the next step is to introduce different types of attribute vectors. The attribute vector should have a varying width depending on the number of distinct values in the dictionary. If the

[1] https://en.cppreference.com

dictionary only holds three values, using 64 bit for every value id would be a huge waste and 8 bit is more than enough.

This is what is meant with `uint*_t` above. Since `uint*_t` is not an actual class, we will need to implement a wrapper for this vector. For this, implement the new class `FittedAttributeVector<uintX_t>`. This class inherits from BaseAttributeVector and implements the following interface:

```
BaseAttributeVector() = default;
virtual ~BaseAttributeVector() = default;

// returns the value at a given positon
virtual ValueID get(const ChunkOffset i) const = 0;

// sets the value_id at a given position
virtual void set(const ChunkOffset i, const ValueID
     value_id) = 0;

// returns the number of values
virtual size_t size() const = 0;

// returns the width of the values in bytes
virtual AttributeVectorWidth width() const = 0;
```

During the creation of the dictionary, you should check what width you need and initialize `_attribute_vector` in the DictionarySegment accordingly.

We will not implement Bitpacking (Zero suppression)[2] during this sprint. Instead, we will rely on the native integer types: `uint8_t`, `uint16_t`, or `uint32_t` (see http://en.cppreference.com/w/c/types/integer).

In the template, there are no tests that check if the correct width is selected.

Submission instructions

For your final submission, please file a pull request from your forked repository to our repository. Also, please email us (Markus.Dreseler and Jan.Kossmann) the commit ID (i.e., the SHA-1 hash) so that we know which version you consider final **until 13 Nov 2018 11:59 PM CET**. Do not open a pull request!

[2] https://en.wikipedia.org/wiki/Zero_suppression