Build you own Database
The Opossum Blueprint

WS 17/18 :: Sprint 2

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

Overview

Before you start: Remember to work on a new branch for this so that the pull requests that you submitted for the first sprint do not get changed.

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 columns get compressed.

Dictionary-Encoded Columns

A dictionary-encoded column 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 column 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 columns in that chunk so that they are dictionary columns. As a result, it is not possible for chunks to contain both value and dictionary-encoded columns. You should create a new empty chunk before starting the compression, add the new dictionary-encoded columns to the chunk and in the end put the new columns into place by exchanging the complete chunk.

Additionally, the dictionary column 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 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_column_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 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;

// inserts 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 DictionaryColumn accordingly.

As explained during the lecture, we will not implement Bitpacking 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 14 Nov 2017 11:59 PM CET**.