



## Overview

In the second sprint, you will implement dictionary-encoded chunks for OpossumDB. In contrast to Hyrise, 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. The reason for this approach is manifold. We will shortly discuss two main access patterns in databases and how they operate on data in order to motivate the chunk design.

Database workloads are often separated in transactional and analytical workloads. Transactional workloads are dominated by single tuple accesses (e.g., request tuple X from table) and modifications (e.g., inserts of new tuples and updates). In contrast, analytical workloads are dominated by queries that process a large number of tuples (e.g., 'what is the sum of column X between timestamp Y and Z?'). Usually, older data is less frequently accessed for updates and thus a chunk's workload becomes more and more dominated by analytical queries over time. For that reason, chunks start being uncompressed and are later dictionary-encoded.

Uncompressed chunks allow fast appending, updates (see 'insert-only' approach), and materializations. Especially appending and updates can be prohibitively expensive on dictionary-encoded columns as these operations might require a complete rewrite of the chunk if the inserted attributes cannot be appended to the dictionaries. Furthermore, materializations are usually highly correlated with a date's age [1]. From a performance point of view it can thus make sense to keep data uncompressed as long as it is regularly accessed for materializations during transactional workloads. While dictionary-encoded columns are well suited for sequential tasks such as scanning, they introduce additional cache misses for point accesses, as the dictionary has to be accessed for every single materialized attribute. Thus, as long as materializations and inserts dominate the workload of a chunk, an uncompressed format is usually the better fit.

After a while, when the chunk is full and thus immutable (we'll discuss ways to invalidate tuples later) in OpossumDB, the chunk's workload usually becomes dominated by analytical queries. As soon as the dominating workload of a chunk shifts to be dominated by sequential tasks, OpossumDB compresses the chunk using dictionary encoding in order to improve performance and at the same time reduce the main memory footprint. Dictionary-encoded columns allow fast range queries and - more importantly - allow for further optimizations such as SIMD (single instruction, multiple data; see OpenHPI course for more information). Typically, most columns can be stored with only one or two bytes per entry, which improves the performance of sequential tasks significantly as more data can be read with each access to the DRAM (i.e., the main bottleneck of in-memory databases).

It is important to understand that there is no single format perfectly suitable for all workloads. But with our chunk model, we can iteratively adapt the storage system to fit the current workload. How and when we compress certain chunks will be a task for later. In this sprint, we are simply going to implement the required data structures and methods without minding the scheduling.



## Dictionary-Encoded Chunks

A dictionary-encoded chunk 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 chunks length; its order is determined by the order data in the original chunk (thus, insert order)
- **The dictionary:** an `std::vector<T>` storing the actual distinct values of the column in sorted order.

The attribute vector has a varying width depending on the number of distinct values in the dictionary. The type should be `uint8_t`, `uint16_t`, or `uint32_t` (see <http://en.cppreference.com/w/c/types/integer>). We recommend starting with a fixed size implementation.

For now, it is not possible for chunks to contain both, plain value and dictionary-encoded columns. A chunk holds either only value or only dictionary-encoded columns. As discussed in today's lecture, there are two ways of exchanging plain value with dictionary-encoded chunks. The first option would be to exchange them one after another. Each time a column would have been compressed, it would take the place of its value column counterpart. We believe the second option will simplify the chunk handling in the future. It also offers the opportunity to get familiar with interesting C++ concepts. Hence, we would like you to go down a different path: 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. This solution should cause some compiler trouble [2] with one of our classes' current interfaces. You are allowed to modify the interface, but please make it a "minimally invasive procedure."

## Performance Challenge

Part of this sprint is a little (not graded) performance challenge. As part of a 'performance test' you should create a table with 10 M tuples in a single chunk. The table contains 10 columns of type `long` filled with random integers and a distinct count of  $2^{(\$columnID\$+1)*2}$  (thus up to 1 M distinct values). After creating this uncompressed chunk, you should call the `compress` method and measure your runtime. You are not allowed to use any additional libraries.

We will later measure the runtime of your `compress` method on a 2,7 GHz Intel Core i7 with 4 cores and 16 GB of DRAM and compare with the implementation of other groups. Feel free to use any of these hardware resources. We are going to present and discuss the fastest solution together.



## 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 **until 16/11/2016 11:59 PM CET**.

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

[1] Boissier et al.: Analyzing Data Relevance and Access Patterns of Live Production Database Systems. CIKM 2016.

[2] <http://i.stack.imgur.com/b2VBV.png>