Summary

Updating compressed column-stores

Our modern society is relying more and more on information technology to manage both data and everyday activities. Over the years, this has resulted in an exponential growth in the amount of information being processed and stored by computer systems. Since the 1970's, *database management systems* (DBMS) have been used to organize, query and manipulate digital information in a structured way. Early database systems focused mainly on *transaction processing*, which deals with retrieving and manipulating data items, like, for example, the balance of a bank account, or address information of a customer. With time, the rapidly growing volume of data being gathered by organizations has sparkled a desire to gain more high-level insight into such data, for the purpose of providing, for example, historical or predictive views of business operation. This novel, *analytic* kind of data processing is often termed *business intelligence*, and is characterized by an "ad-hoc" query style, often involving large volumes of data, combined with a competitive or economic advantage if queries complete rapidly.

The growth in data volume has been accompanied by exponential improvements in certain areas of computer hardware, most notably CPU processing power and storage capacity of both main memory and disk. However, developments in data access *latency* and data transfer *bandwidth* have been lagging behind this exponential trend. The consequence being, that software developers have to deal with a widening gap between processing power on one hand, and data access cost on the other. Furthermore, the block-oriented access mechanisms provided by storage devices increasingly favor sequential over random access patterns, as latency improvements are falling behind advances in transfer bandwidth.

For analytic database workloads, which typically process large amounts of data, these trends have led to the wide adoption of so-called *column-oriented* storage layouts. Most database systems implement the relational model, where data is stored in two-dimensional *tables*, consisting of *rows* and *columns*. Each row, or *tuple* represents some entity, while each column represents an *attribute* of each entity. In a column-oriented, or *decomposed storage model* (DSM), tables are partitioned vertically on disk, meaning that attribute values from a single column are stored contiguously within fixed sized pages – the unit of transfer – on disk. The advantage being, that for queries requiring access to only a subset of table columns, it suffices to scan only the relevant attribute pages from disk, resulting in considerable I/O bandwidth savings compared to a *row-oriented*, or *N-ary storage model* (NSM), where a page always contains a sequence of complete tuples.

But column-oriented storage also provides benefits in terms of data processing efficiency within the CPU. By reading small fragments, or *vectors*, of attribute values, from a page in memory into the CPU cache, we make optimal use of cache-lines – the unit of transfer between memory and CPU – exposing comparable bandwidth savings as obtained by reading column-oriented pages

from disk. Furthermore, database *primitives*, responsible for performing actual computations on vectors of data, can be implemented as simple and efficient loops over arrays that are kept free of function calls and branching logic, enabling compilers to produce highly CPU-efficient code. Research around this *vectorized execution model* has resulted in the column-oriented *MonetDB/X100* DBMS, which was later commercialized under the name *Vectorwise*.

The work in this thesis provides solutions to two problems that may arise in a column-oriented and vectorized database engine like Vectorwise: (*i*) the danger of I/O bandwidth bottlenecks, which may occur even when using DSM, and (*ii*) how to provide efficient transactional update support on a storage layout that is notoriously unfriendly to in-place updates.

Given the growing gap between CPU power and disk transfer bandwidth, it may be impossible to deliver raw data from disk at a rate that matches the computational power of a vectorized engine. We show that data compression can be used to significantly alleviate such I/O bottlenecks, on both analytic and information retrieval workloads. The idea is to read compressed pages from disk, and decompress them at a minimal CPU cost, so that perceived I/O bandwidth can be improved, at most by a factor equal to the compression ratio. Here, we trade CPU processing power, which evolves favorably, for perceived disk bandwidth, which is scarce.

For such techniques to be effective even on high-performance disk systems, like RAID or SSD, it is important for (de)compression methods to be *lightweight*. We contribute three new compression schemes, PFOR, PFOR-DELTA and PDICT, that are specifically designed for the super-scalar capabilities of modern CPUs. These algorithms exploit the fact that attribute values within a column are from the same domain and of equal type, allowing (de)compression to be kept simple but fast (i.e. gigabytes per second). Furthermore, we show that by decompressing compressed pages in memory only at small granularity, in an on-demand fashion, directly *into the CPU cache*, we can eliminate the danger of memory bandwidth becoming a bottleneck.

To add transactional update support to a column-store DBMS, we consider in-place updates to be out of question, due to I/O costs that are proportional to the number of table attributes, and the fact that tables are often stored sorted and compressed. Rather, we focus on differential update techniques, where delta changes against otherwise immutable tables from a scan-optimized "read-store" are maintained in a memory-resident, update-friendly "write-store". During a scan, write-store updates are merged with the read-store table to produce an up-to-date table image. The idea is to sacrifice main-memory storage, which evolves favorably, to reduce I/O accesses, which are expensive, both in terms of latency and bandwidth.

We propose to organize delta updates against a read-store table by update *position*, or tuple offset, in a novel index structure called *positional delta tree* (PDT). Alternatively, one could organize update tuples by their (sort) key attribute *values*, for example in a B^+-tree-like structure. We show that positional update maintenance has several advantages over a value-based approach, most notably in the area of merging efficiency during a table scan. Furthermore, PDTs allow efficient encoding of attribute level modifications (i.e. SQL UPDATE), and can be stacked on top of each other, allowing for a clean implementation of snapshot isolation to support transactions.