Cache Conscious Column Organization in In-Memory Column Stores

David Schwalb, Jens Krüger, Hasso Plattner

Technische Berichte Nr. 67

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 67

David Schwalb | Jens Krüger | Hasso Plattner

Cache Conscious Column Organization in In-Memory Column Stores

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6389/
URN urn:nbn:de:kobv:517-opus-63890
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-63890

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-228-5

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Cost models are an essential part of database systems, as they are the basis of query performance optimization. Based on predictions made by cost models, the fastest query execution plan can be chosen and executed or algorithms can be tuned and optimized.

In-memory databases shift the focus from disk to main memory accesses and CPU costs, compared to disk based systems where input and output costs dominate the overall costs and other processing costs are often neglected. However, modeling memory accesses is fundamentally different and common models do not apply anymore.

This work presents a detailed parameter evaluation for the plan operators scan with equality selection, scan with range selection, positional lookup and insert in in-memory column stores. Based on this evaluation, we develop a cost model based on cache misses for estimating the runtime of the considered plan operators using different data structures. We consider uncompressed columns, bit compressed and dictionary encoded columns with sorted and unsorted dictionaries. Furthermore, we discuss tree indices on the columns and dictionaries.

Finally, we consider partitioned columns consisting of one partition with a sorted and one with an unsorted dictionary. New values are inserted in the unsorted dictionary partition and moved periodically by a merge process to the sorted partition. We propose an efficient merge algorithm, supporting the update performance required to run enterprise applications on read-optimized databases and provide a memory traffic based cost model for the merge process.

Chapter 1

Introduction

In-memory column stores commence to experience a growing attention by the research community. They are traditionally strong in read intensive scenarios with analytical workloads like data warehousing, using techniques like compression, clustering and replication of data. A recent trend introduces column stores for the backbone of business applications as a combined solution for transactional and analytical processing. This approach introduces high performance requirements as well for read performance as also for write performance to the systems.

Typically, optimizing read and write performance of data structures results in trade-offs, as e.g. higher compression rates introduce overhead for writing, but increase the read performance. These trade-offs are usually made during the design of the system, although the actual workload the system is facing during execution varies significantly. The underlying idea of this report is a database system, which supports different physical column organizations with unique performance characteristics, allowing to switch and choose the used structures at runtime depending on the current, historical or expected future workloads. However, this report will not provide a complete description or design of such a system. Instead, we provide the basis for such a system by focusing on selected data structures for in-memory column stores, presenting a detailed parameter evaluation and providing cache-based cost functions for the discussed plan operators.

The decision which column organization scheme is used, has to be made by the system based on knowledge of the performance of individual database operators and data structures. This knowledge is represented by the *cost model*. A cost model is an abstract and simplified version of the actual system, which allows to make predictions about the actual system. A model is always a tradeoff between accuracy and speed with which predictions can be made. The most accurate model is the system itself. Obviously, executing a Query Execution Plan (QEP) in order to determine the time it will take for its execution, does not make sense.

Models based on simulations also produce very accurate predictions. However, evaluating such a model requires to run costly simulations. Due to the performance requirements for

query execution, query optimizers usually build on analytical models. Analytical models can be described in closed mathematical terms and are therefore very fast to evaluate.

Query optimizers do not require extremely accurate predictions of the model, but they do require that the relations produced by the model reflect the real system. In other words, the order of QEPs sorted by their performance should be the same in the model and in the real system. Additionally, the fastest QEPs usually do not differ significantly in execution performance, but predicting which of those plans will be the fastest requires a very detailed model. The time saved by finding the faster execution plan is then outweighed by the time needed for the additional accuracy of the prediction. Therefore, we are usually not interested in finding the best possible solution, but rather in finding a sufficiently fast solution in a short time frame.

Our contributions are i) a detailed parameter discussion and analysis for the operations scan with equality selection, scan with range selection, lookup and inset on different physical column organizations in in-memory column stores, ii) a cache based cost model for each operation and column organization plus dictionary and column indices, iii) an efficient merge algorithm for dictionary encoded in-memory column stores, enabling them to support the update performance required to run enterprise application workloads on read-optimized databases.

1.1 Problem Statement

In this work, we focus on in-memory databases [52, 7, 27, 69, 58]. Especially columnar in-memory databases have received recent interest in the research community [25, 44, 59]. A logical column can be represented in main memory in different ways, called physical column organization or schemes. Essentially, the physical schema describes how a column is encoded and stored in main memory.

We focus on estimating query costs for the considered physical column organizations. In order to answer this problem, we assume a closed system in terms of possible queries, physical column organizations and given operator implementations. We want to find an accurate estimation, which is still computable in reasonable time. We also take column and dictionary indices into account and want to provide a respective cost model for the use of these indices.

1.2 Assumptions and Simplifications

We will make certain assumptions throughout this report and focus on estimating query costs based on cache misses for the given operators and physical column organizations in a column oriented in-memory storage engine. In particular, we assume the following:

Enterprise Environment We base this work on the assumptions of using a columnar in-memory database for enterprise applications for combined transactional and analytical systems. Recent work analyzed the workload of enterprise customer systems, finding in total more than 80 % of all queries are read access – for OLAP systems even over 90 % [46]. While this is the expected result for analytical systems, the high amount of read queries on transactional systems is surprising as this is not the case in traditional workload definitions. Consequently, the query distribution leads to the concept of using a read-optimized database for both transactional and analytical systems. Even though most of the workload is read-oriented, 17 % (OLTP) and 7 % (OLAP) of all queries are updates. A read-optimized database supporting both workloads has to be able to support this amount of update operations. Additional analyses on the data have shown an update rate varying from 3,000 to 18,000 updates/second.

Insert Only Column Store In order to achieve high update rates, we chose to model table modifications following the insert-only approach as in [44, 59]. Therefore, updates are always modeled as new inserts, and deletes only invalidate rows. We keep the insertion order of tuples and only the lastly inserted version is valid. We chose this concept because only a small fraction of the expected enterprise workload are modifications [44] and the insert-only approach allows queries to also work on the history of data.

Workload We assume a simplified workload, consisting of the query types scan with equality selection, scan with range selection, positional lookup and inserts. A given workload consists of a given distribution of these queries and their parameters. We chose these operators as we identified them as the most basic operators needed by an insert only database system. Additionally, more complex operators can be assembled by combining these basic operators, as e.g. a join consisting of multiple scans.

Isolated Column Consideration As columnar databases organize their relations in isolated columns, query execution typically is a combination of multiple operators working on single columns and combining their results. For simplicity, we assume queries working on a single column and neglect effects introduced by operators on multiple columns.

Parallelism With high end database servers developing from multi-core to many-core systems, modern databases are highly parallelized using data level parallelism, instruction level parallelism and thread level parallelism. However, for our cost model we assume the algorithms to execute on a single core without thread level or data level parallelism. For future work, an extension for parallel query processing could be introduced based on the algebra of the generic cost model introduced in [52].

Materialization Strategies In the process of query execution, multiple columns have to be stitched together, representing tuples of data as output. This process is basically the opposite operation of a projection in a row store. The optimal point in the query plan to do this is not obvious [1]. Additionally to the decision when to add columns to the query plan, a column store can work on lists of positions or materialize the actual values, which can be needed by some operators. Abadi et al. present four different materialization strategies - early

materialization pipelined, early materialization parallel, late materialization pipelined and late materialization parallel. We refer to the process of adding multiple columns to the query plan as tuple reconstruction. The process of switching the output from lists of positions to lists of position and value pairs is called materialization. Throughout the report, we assume scan operators to create lists of positions as their result. The process of tuple reconstruction can be seen as a collection of positional lookup operations on a column.

Column Indices Index structures for databases have been intensely studied in literature and various indexing techniques exists. We assume a column index to be a mapping from values to positions, building a classical inverted index known from document retrieval systems. We assume the index to be implemented by a B^+-tree structure as described in Section 6.

Bandwidth Boundness In general, the execution time of an algorithm accessing only main memory as the lowest level in the memory hierarchy and no other I/O devices can be assembled by two parts – a) the time spent doing computations on the data and b) the time the processor is waiting for data from the memory hierarchy. Bandwidth bound algorithms spend a significant amount of time fetching data from memory so that the actual processing times are comparable small. We assume our considered algorithms to be bandwidth bound and do not consider T_{CPU} in our cost model and focus on predicting the memory access costs through the number of cache misses. However, database systems are not always bandwidth bound and for more accurate predictions T_{CPU} can be calibrated and considered in the cost model [52].

1.3 Definition of Key Terms

In order to avoid ambiguity, we define the following terms to be used in the rest of the report.

1. *Table:* A relational table or relation with attributes and containing records.
2. *Attribute:* A field of a table with an associated type.
3. *Record:* Elementary units contained in a table, containing one value for each attribute in the table.
4. *Column:* Tables are physically stored in main memory as a collection of columns. A column is the physical representation of an attribute.
5. *Physical Column Organization:* Describes the organization scheme and how a column is physically stored in memory, e.g. if it is stored uncompressed or if compression techniques are applied.
6. *Update:* Any modification operation on the table resulting in a new entry.

7. *Partitioned Column:* A partitioned column consists of two partitions – one uncompressed and write optimized delta partition and one compressed and read optimized main partition. All writes go into the delta partition and are merged periodically into the main partition.

8. *Merge:* The merge process periodically combines the read and write optimized partitions of a partitioned column into one read optimized partition.

9. *Column Index:* A column index is a separate data structure accelerating specific operations on the column by the costs of index maintenance when inserting new values.

10. *Dictionary Index:* A dictionary index is a separate tree index on top of an unsorted dictionary, allowing to leverage binary search algorithms.

11. *Dictionary Encoding:* Dictionary encoding is a well known light weight compression technique, reducing the redundancy by substituting occurrences of long values with shorter references to these values.

12. *Dictionary and Attribute Vector:* In a dictionary encoded column, the actual column contains two containers: the attribute vector and the value dictionary. The attribute vector is a standard vector of integer values storing only the reference to the actual value, which is the index of the value in the value dictionary and is also called value-id.

13. *Bit-Packing:* A bit-packed vector of integer values uses only as many bits for each value as are required to represent the largest value in the vector.

1.4 Structure of this Report

The remainder of the report is structured as follows: Chapter 2 discusses related work regarding in-memory column stores, cost models for columnar in-memory databases, caches and their effects on application performance and indices. Chapter 3 gives an overview of the discussed system and introduces the considered physical column organizations and plan operators. Chapter 4 presents an evaluation of parameter influences on the plan operator performance with varying number of rows, number of distinct values, value disorder, value length and value skewness. Our cache miss based cost model is presented in Chapter 5, followed by Chapter 6 introducing column and dictionary indices and their respective costs. Then, Chapter 7 describes partitioned columns and the merge process including a memory traffic based cost model. Finally, the report closes with concluding remarks and future works in Chapter 8.

Chapter 2

Background and Related Work

This section gives an overview of related work regarding in-memory column stores, followed by work concerning cost models for main memory databases, index structures and cache effects.

Rmakrishnan and Gehrke define a database managements system as "a software designed to assist in maintaining and utilizing large collections of data" [61]. Today, a wide range of database systems exists, varying in their intended use cases and workloads, the used data models, their logical and physical data organization or the primary data storage locations.

Typical use cases and types of databases are text retrieval systems, stream based databases [14], real-time databases, transaction processing systems, data warehouses or data mining systems. Text retrieval systems like [73] usually store collections of free text called documents and are optimized to execute queries matching a set of stored documents. Stream based databases are designed to handle constantly changing workloads, as e.g. stock market data or data generated through RFID tracking [56]. Real-time databases are specifically designed to guarantee specified time constraints as required by e.g. telephone switching, radar tracking or arbitrage trading applications [39].

Transaction processing systems are usually based on the relational model [12] and are designed to store data generated by events called transactions, supporting atomicity, consistency, isolation and durability. OLTP workloads are characterized by a mix of reads and writes to a few rows at a time, typically leveraging B^+-Trees or other index structures. Usually, OLTP systems store their data in row organized tables, which allows fast retrieval of single rows and are optimized for fast write performance. Typical use cases for transactional databases are in the field of operational data for enterprise resource planning systems or as the backbone of other applications.

Conversely, analytical systems usually work on a copy of the operational data extracted through extract-transform-load (ETL) processes and stored optimized for analytical queries. OLAP applications are characterized by bulk updates and large sequential scans, spanning few columns but many rows of the database, for example to compute aggregate values. The

analytical systems are often organized in star schemas and work with pre-aggregated data.

Besides the relational data model, various other data models exist like the network data model, the hierarchical data model or the object data model. However, this report focusses on relational databases in the context of enterprise applications. Relational databases differ in their intended data schemata (like normalized or star schemata), their physical layouts of data storage (e.g. row or column oriented, compression techniques) or their primary data storage locations (disk or main memory).

Classical disk based transactional databases are IBMs DB2, Oracle Database or SAPs MaxDB. H-Store [38] is a row-based distributed database system storing data in main memory. Besides transactional systems, the need for specialized decision support systems evolved [20, 67] resulting in analytical systems optimized for OLAP processing. SybaseIQ is a column oriented disk-based system explicitly designed optimizing the analytical query performance [50]. C-Store and its commercial version Vertica are also disk based column stores designed for analytics, allowing columns to be stored in multiple sort orders. Writes are accumulated in a writeable store and moved by a tuple-mover into the read optimized store [66]. MonetDB and Vectorwise [9, 75, 74, 10] are column oriented databases targeted to support query-intensive applications like data mining and ad-hoc decision support.

A recent research trend started to reunite the worlds of transactional and analytical processing by introducing and proposing systems designed for mixed workloads like SancoussiDB [58, 59], Hyrise [26] or HyPer [41, 40].

2.1 In-Memory Column Stores

Businesses use their transactional data as a basis to evaluate their business performance, gain insights, for planning and predictions of future events with the help of separate analytical systems. In the recent past, a desire for more actual analyses developed, requiring to work directly on the transactional data. Plattner describes the separation of transactional and analytical systems and that database structures were designed to support complex business transactions focusing on the transactional processing [58]. Analytical and financial planning systems were moved into separate systems, which were highly optimized for the read intensive workloads, promising more performance and flexibility.

The main reason for separating transactional and analytical systems were their different performance requirements. Actually, the requirements are even contradicting. Transactional workloads were believed to be very write intensive, selecting only individual rows of tables and manipulating them. Analytical workloads usually have a very high selectivity, scanning entire attributes and joining and grouping them. The underlying problem is that data structures can only be optimized up to a certain point for all requirements. After that point, each optimization in one direction becomes a cost factor for another operation. For example, higher compression of a data structure can speed up complex read operations due to less memory that has to be

read, but increases the expenses for adding new items to the data structure. In extreme cases, the whole compression scheme even has to be rebuilt.

The flexibility and speed which is gained by separating OLTP and OLAP is bought by introducing high costs for transferring the data (ETL processes) and managing the emerging redundancy. Additionally, data is only periodically transferred between transactional and analytical systems, introducing a delay in reporting. This becomes especially important as through analytical reporting on their transactional data, companies are much more able to understand and react to events influencing their business. Consequently, there is an increasing demand for "real-time analytics" – that is, up-to-the moment reporting on business processes that have traditionally been handled by data warehouse systems. Although warehouse vendors are doing as much as possible to improve response times (e.g., by reducing load times), the explicit separation between transaction processing and analytical systems introduces a fundamental bottleneck in analytics scenarios. While the predefinition of data to be extracted and transformed to the analytical system results in business decisions being made on a subset of the potential information, the separation of systems prevents transactional applications from using analytics functionality throughout the transaction processing due to the latency that is inherent in the data transfer.

Recent research started questioning this separation of transactional and analytical systems and introduces efforts of uniting both systems again [58, 59, 46, 40, 25, 41]. Although, the goal is not to advocate a complete unification of OLAP and OLTP systems, because the requirements of data cleansing, system consolidation and very high selectivity queries cannot yet be met with the proposed systems and still require additional systems. However, using a read optimized database system for OLTP allows to move most of the analytical operations into the transactional systems, so that they profit by working directly on the transactional data. Applications like real-time stock level calculation, price calculation and online customer segmentation will benefit from this up-to-date data. A combined database for transactional as well as analytical workloads eliminates the costly ETL process and reduces the level of indirection between different systems in enterprise environments, enabling analytical processing directly on the transactional data.

The authors of [58, 46] pursue the idea of designing a persistence layer which is better optimized for the observed workload from enterprise applications, as today's database systems are designed for a more update intensive workload as they are actually facing. Consequently, the authors start with a read optimized database system and optimize its update performance to support transactional enterprise applications. The back-bone of such a system's architecture could be a compressed in-memory column-store, as proposed in [58, 25]. Column oriented databases have proven to be advantageous for read intensive scenarios [50, 75], especially in combination with an in-memory architecture.

Such a system has to handle contradicting requirements for many performance aspects. Decisions may be made with the future application in mind or by estimating the expected workload. Nevertheless, the question arises which column oriented data structures are used in

combination with light-weight compression techniques, enabling the system to find a balanced trade-off between the contradicting requirements. This report aims at studying these trade-offs and at analyzing possible data structures.

2.2 Index Structures

As in disk based systems, indexing in in-memory databases remains important although sequential access speeds allow for fast complete column scans. Disk based systems often assume that index structures are in memory and accessing them is cheap compared to the main cost factor which is accessing the relations stored on secondary storage. However, in-memory databases still profit from index structures that allow to answer queries selecting single values or ranges of values. Additionally, translating values into value-ids requires searching on the value domain which can be accelerated by the use of indices. For in-memory databases the index access performance is even more essential and must be faster than sequentially scanning the complete column.

Lehman and Carey [48] describe in their work from 1986 how in-memory databases shift the focus for data structures and algorithms from minimizing disk accesses to using CPU cycles and main memory efficiently. They present T-Trees as in-memory index structures, based on AVL and B-Trees. T-Trees are binary search trees, containing multiple elements in one node to leverage cache effects.

A recent trend redesigns data structures to be more cache friendly and significantly improve performance by effectively using caches. Lee et al. introduce a Cache Sensitive T-Tree [47] whereas Rao and Ross optimize B^+-Trees for cache usage introducing Cache Sensitive Search Trees [62, 63]. The work of Kim et al. presents a performance study evaluating cache sensitive data structures on multi-core processors [42].

Besides creating indices manually, the field of automatic index tuning promises self-tuning systems reacting dynamically to their workloads. Some techniques analyze the workload and periodically optimize the system while other techniques constantly adapt the system with every query. Graefe and Kuno propose adaptive merging as an adaptive indexing scheme, exploiting partitioned B-trees [22] by focusing the merge steps only on relevant key ranges requested by the queries [24, 23]. Idreos et al. propose database cracking as another adaptive indexing technique, partitioning the keys into disjoint ranges following a logic similar to a quick-sort algorithm [33, 34]. Finally, Idreos et al. propose a hybrid adaptive indexing technique as a combination of database cracking and adaptive merging [35]. Besides tree or hash based indices, bitmap indices are also widely used in in-memory database systems, e.g. Wu et al. propose the Word-Aligned Hybrid code [72] as compression scheme for bitmap indices.

2.3 Cost Models

Relatively little work has been done on researching main memory cost models. This probably is due to the fact, that modeling the performance of queries in main memory is fundamentally different to disk based systems, where IO access is clearly the most expensive part. In in-memory databases, query costs consist of memory and cache access costs on the one hand and CPU costs on the other hand.

In general, cost models differ widely in the costs they estimate. Complexity based models provide a correlation between input sizes and execution time, but normally abstract constant factors as in the big O-Notation. Statistical models can be used to estimate the result sizes of operations based on known parameters and statistics of the processed data. Other cost models are based on simulations and completely go through a simplified process mapping the estimated operations. Usually simulation based models provide very accurate results but are also expensive and complex. Analytical cost models provide closed mathematical formulas, estimating the costs based on defined parameters. Additionally, system architecture aware models take system specific characteristics into account, e.g. cache conscious cost models which rely on parameters of the memory hierarchy of the system.

Based on IBM's research prototype "Office by Example" (OBE) Whang and Krishnamurthy [70] present query optimization techniques based on modeling CPU costs. As modeling CPU costs is complicated and depends on parameters like the hardware architecture and even different programming styles, Whang and Krishnamurthy experimentally determine *bottlenecks* in the system and their *relative weights* and *unit costs*, building analytical cost formulas.

Listgarten and Neimat [49] compare three different approaches of cost models for in-memory database systems. Their first approach is based on hardware costs, counting CPU cycles. The second approach models application costs, similarly to the method presented by Whang and Krishnamurthy [70]. The third approach is based on execution engine costs, which is a compromise between complex hardware-based models and general application-based models. Listgarten and Neimat argue that only the engine-based approach provides accurate and portable cost models.

Manegold and Kersten [51] describe a generic cost model for in-memory database systems, to estimate the execution costs of database queries based on their cache misses. The main idea is to describe and model reoccurring basic patterns of main memory access. More complex patterns are modeled by combining the basic access patterns with a presented algebra. In contrast to the cache-aware cost model from Manegold, which focusses on join operators, we compare scan and lookup operators on different physical column layouts.

Zukowski, Boncz, Nes and Heman [75, 8] describe X100 as an execution engine for monetDB, optimized for fast execution of in-memory queries. Based on in-cache vectorized processing and the tuple-at-a-time Volcano [21] pipelining model, fast in cache query execution is enabled. The X100 cost model is based on the generic cost model of Manegold and Kersten, modeling query

costs by estimating cache misses.

Sleiman, Lipsky and Konwar [65] present an analytical model, predicting the access times for memory requests in hierarchical multi-level memory environments, based on a linear-algebraic queuing theory approach.

Pirk [57] argues that general purpose database systems usually fall short in leveraging the principle of data locality, because data access patterns can not be determined during designing the system. He proposes storage strategies, which automatically layout the data in main memory based on the workload of the database. His cost analyses are based on the generic cost model of Manegold and Kersten, but are extended for modern hardware architectures including hardware prefetching.

2.4 Caches

The influences of the cache hierarchy on application performance have been extensively studied in literature. Various techniques have been proposed to measure costs of cache misses and pipeline stalling. Most approaches are based on handcrafted micro benchmarks exposing the respective parts of the memory hierarchy. The difficulty is to find accurate measurement techniques, mostly relying on hardware monitors to count CPU cycles and cache misses. Other approaches are based on simulations, allowing detailed statistics of the induced costs.

Drepper [18] describes the concepts of caching, their realization in hardware and summarizes the implications for software design.

Barr, Cox and Rixner [4] study the penalties occurring when missing the translation look-aside buffer (TLB) in systems with radix page tables like the x86-64 system and compare different page table organizations. Due to the page table access, the process of translating a virtual to a physical address can induce additional TLB cache misses, depending on the organization of the page table.

Saavedra and Smith [64] develop a model predicting the runtime of programs on analyzed machines. They present an uniprocessor machine-independent model based on various abstract operations. By combining the measurement of the performance of such abstract operations on a concrete processor and the frequency of these operations in a workload, the authors are able to estimate the resulting execution time. Additionally, penalties of cache misses are taken into account. However, it is assumed that an underlying miss ratio is known for the respective workload and no mechanisms for predicting the number of cache misses are provided.

Hirstea and Lenoski [31] identify the increasing performance gap between processors and memory systems as an essential bottleneck and introduce a memory benchmarking methodology based on micro benchmarks measuring restart latency, back-to-back latency and pipelined bandwidth. The authors show that the bus utilization in uniform-memory-access (UMA) systems has a significant impact on memory latencies.

Babka and Tuma [3] present a collection of experiments investigating detailed parameters and provide a framework measuring performance relevant aspects of the memory architecture of x86-64 systems. The experiments vary from determining the presence and size of caches to measuring the cache line sizes or cache miss penalties.

Puzak et al. [60] propose *Pipeline Spectroscopy* to measure and analyze the costs of cache misses. It is based on a visual representation of cache misses, their costs and their overlapping. The spectrograms are created by simulating the program, allowing to closely track cache misses and the induced stalling. Due to the simulation approach, the method is cost intensive and not suited for cost models requiring fast decisions.

Chapter 3

System Definition

This chapter gives an overview of the considered system. First, Section 3.1 gives a formal definition of the system and considered parameters. Then, Section 3.2 introduces the considered physical column organization schemes uncompressed column, dictionary encoded columns and bit-packing. Followed by Section 3.3 discussing operations on columns and their complexities depending on the internal organization. Finally, Section 3.4 introduces the considered plan operators and discusses their theoretical complexities.

3.1 Parameters

We consider a database consisting of a set of tables $\mathbb{T}$. A table $t \in \mathbb{T}$ consists of a set of attributes $\mathbb{A}$. The number of attributes of a table t will be denoted as $|\mathbb{A}|$. We assume the value domain $\mathbb{V}$ of each attribute $a \in \mathbb{A}$ to be finite and require the existence of a total order ρ over $\mathbb{V}$. In particular, we define $\boldsymbol{c.e}$ as the value length of attribute a and assume $\mathbb{V}$ to be the set of alphanumeric strings with the length $\boldsymbol{c.e}$. An attribute a is a sequence of $\boldsymbol{c.r}$ values $v \in \mathbb{D}$ with $\mathbb{D} \subseteq \mathbb{V}$, where $\boldsymbol{c.r}$ is also called number of rows of a and $\mathbb{D}$ also called the dictionary of a.

$\mathbb{D}$ is a set of values $\mathbb{D} = \{v_1, ..., v_n\}$. We define $\boldsymbol{c.d} := |\mathbb{D}|$ as the number of distinct values of an attribute. In case the dictionary is sorted, we require $\forall_{v_i \in \mathbb{D}} : v_i < v_{i+1}$. In case the dictionary is unsorted, $v_1, ..., v_n$ are in insertion order of the values in attribute a. The position of a value v_i in the dictionary defines its value-id $id(v_i) := i$. For bit encoding, the number of values in $\mathbb{D}$ is limited to 2^b, with b being the number of bits used to encode values in the value vector. We define $\boldsymbol{c.e}_c := b$ as the compressed value length of a, requiring $\boldsymbol{c.e}_c \geq \lceil log_2(\boldsymbol{c.d}) \rceil$ bits.

The degree of sortedness in a is described by the measure of disorder denoted by $\boldsymbol{c.u}$, based on Knuth's measure of disorder, describing the minimum amount of elements that need to be

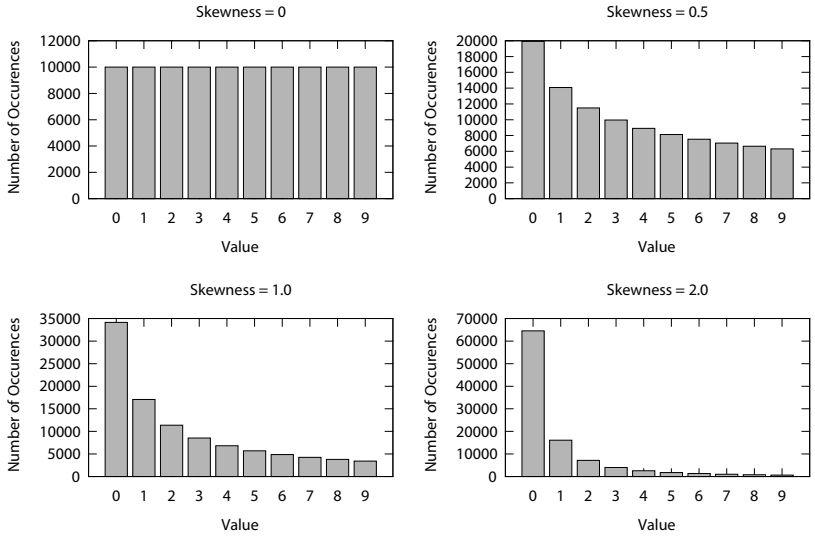


Figure 3.1: Influence of the parameter skewness on the value distribution. The histograms reflect the number of occurrences of each distinct value (for 10 distinct values and 100.000 generated values, skewness varying from 0 to 2).

removed from a sequence so that the sequence would be sorted [43].

Finally, we define the value skewness $\boldsymbol{c.k}$, describing the distribution of values of an attribute, as the exponent characterizing a Zipfian distribution. We chose to model the different distributions by a Zipfian distribution, as the authors in [32] state that the majority of columns analyzed from financial, sales and distribution modules of an enterprise resource planning (ERP) system were following a power-law distribution – a small set of values occurs very often, while the majority of values is rare.

The frequency of the x^{th} value in a set of $\boldsymbol{c.d}$ distinct values can be calculated with the skewness parameter $\boldsymbol{c.k}$ as:

$$f(x, \boldsymbol{c.k}, \boldsymbol{c.d}) = \frac{\frac{1}{x^{\boldsymbol{c.k}}}}{\sum_{n=1}^{\boldsymbol{c.d}} \frac{1}{n^{\boldsymbol{c.k}}}} \tag{3.1}$$

Intuitively, $\boldsymbol{c.k}$ describes how heavily the distribution is drawn to one value. Figure 3.1 shows the influence of $\boldsymbol{c.k}$ on a value distribution with $\boldsymbol{c.d} = 10$ and $\boldsymbol{c.d} = 100.000$ values, displaying the distribution as a histogram for $\boldsymbol{c.k} = 0$, $\boldsymbol{c.k} = 0.5$, $\boldsymbol{c.k} = 1.0$ and $\boldsymbol{c.k} = 2.0$. In the case of $\boldsymbol{c.k} = 0$, the distribution equals a uniform distribution and every value occurs equally often. As $\boldsymbol{c.k}$ increases, the figure shows how the distribution is skewed more and more.

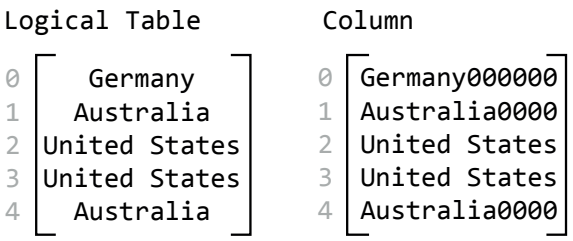


Figure 3.2: Organization of an uncompressed column with value length 13.

3.2 Physical Column Organization

The logical view of a column is a simple collection of values that allows appending new values, retrieving the value from a position and scanning the complete column with a predicate. How the data is actually stored in memory is not specified.

In general, data can be organized in memory in a variety of different ways, e.g. in standard vectors in insertion order, ordered collections or collections with tree indices [59]. In addition to the type of organization of data structures, the used compression techniques are also essential for the resulting performance characteristics. Regarding compression, we will focus on the light weight compression techniques dictionary encoding and bit compression. As concrete combinations, we examine uncompressed columns and dictionary encoded columns with bit compressed attribute vectors whereas the dictionary can be sorted and unsorted.

Uncompressed columns store the values as they are inserted consecutively in memory, as e.g. used in [46]. This design decision has two important properties that affect the performance of the data structure. First, the memory consumption increases and second the scan performance decreases due to the lower number of values per cache line and the lack of a sorted dictionary for fast queries. The update performance can be very high, as new values have to be written to memory and no overhead is necessary to maintain the internal organization or for applying compression techniques.

Figure 3.2 pictures an example of an uncompressed column, showing the logical view of the table on the left and the layout of the column as it is represented in memory on the right side. As no compression schemes are applied, the logical layout equals the layout in memory, storing the values consecutively as fixed length strings in memory.

3.2.1 Dictionary Encoding

Dictionary encoding is a well known, light-weight compression technique [66, 5, 68], which reduces the redundancy by substituting occurrences of long values with shorter references to these values. In a dictionary encoded column, the actual column contains two containers:

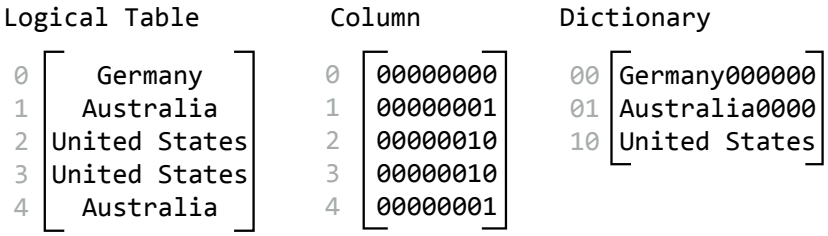


Figure 3.3: Organization of a dictionary encoded column, where the dictionary is unsorted.

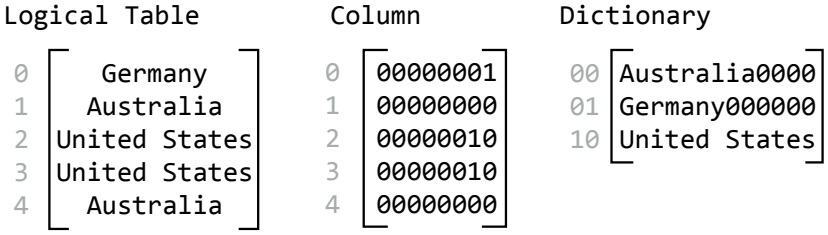


Figure 3.4: Organization of a dictionary encoded column, where the dictionary is kept sorted.

the attribute vector and the value dictionary. The attribute vector is a vector storing only references to the actual values of $\boldsymbol{c.e}_c$ bits, which represent the index of the value in the value dictionary and are also called value-ids. For the remainder, we assume $\boldsymbol{c.e}_c = 32$ bits.

The value dictionary may be an unsorted or ordered collection. Ordered collections keep their tuples in a sorted order, allowing fast iterations over the tuples in sorted order. Additionally, the search operation can be implemented as binary search that has logarithmic complexity. This comes with the price of maintaining the sort order, and inserting in an ordered collection is expensive in the general case.

Figure 3.3 gives an example of a dictionary encoded column. The column only stores references to the value dictionary, represented as 1 byte values. The value-ids are stored sequentially in memory in the same order as the values are inserted into the column. The value dictionary is unsorted and stores the values sequentially in memory in the order as they are inserted. Figure 3.4 shows the same example with a sorted dictionary. Here, the value dictionary is kept sorted, potentially requiring a re-encoding of the complete column when a new value is inserted into the dictionary.

3.2.2 Bit-Packing

A dictionary encoded column can either store the value-ids in an array of a native type like integers or compressed, e.g. in a bit-packed form. When using a native type, the number of representable distinct values in the column is restricted to the size of that type, e.g. with characters of 8 bit to $2^8 = 256$ or with integers on a 64 bit architecture to 2^{64}. The size of the used type is a tradeoff between the amount of representable distinct values and the needed memory space.

When using bit-packing, the value-ids are represented with the minimum number of bits needed to encode the current number of distinct values in the value dictionary. Be d the number of distinct values, then the number of needed bits is $b = \lceil \log_2(d) \rceil$. The value-ids are stored sequentially in a pre-allocated area in memory. The space savings are traded for more extraction overhead when accessing the value-ids. Since normally bytes are the smallest addressable unit, each access to a value-id has to fetch the block where the id is stored and correctly shift and mask the block, so that the value-id can be retrieved.

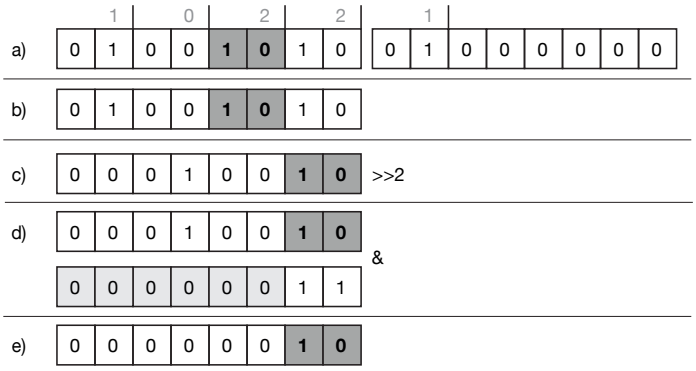


Figure 3.5: Extraction of a bit-packed value-id.

Figure 3.5 outlines an example of how a value-id can be extracted when using bit-packing. Part a) shows the column with the compressed value-ids from the unsorted dictionary example shown in Figure 3.3. As there are only 5 values, the second byte is filled with zeros. The example shows the extraction of the value-id from row 3. In a first step – shown by b), the byte containing the searched value-id has to be identified, here the first of two. In c), a logical right shift of 2 is applied in order to move the searched value-id to the last two bits of the byte. d) extracts the searched value-id by masking the shifted byte with a bit-mask so that the searched value-id is obtained as shown in e). In order to increase the decompression speed techniques like Streaming SIMD Extensions (SSE) can be applied to the extraction process [71]. Another way is to block-wise extract the compressed values, therefore amortizing the decompression costs over multiple values.

3.3 Operations on Data Structures

A column is required to support the two operations *getValueAt* and *getSize* in order to allow accessing and iterating over its values. Additionally, the operation *appendValue* supports inserting new values at the end of the column. A dictionary supports the operations *getValueForValueId* and *getSize* in order to allow accessing and iterating over the stored values. Additionally, *addValue* adds a new value to the dictionary, potentially requiring a rewrite of the complete column. With *getValueIdForValue*, the dictionary can be searched for a specific value. If the dictionary is sorted, a binary search can be used, otherwise the dictionary has to be scanned linearly.

Dictionary Operations As both sorted and unsorted dictionaries are implemented as a sequential array of fixed length values, the operation *getValueForValueId* can be implemented by directly calculating the respective address in memory, reading and returning the requested value. Therefore, the costs only depend on the size of the stored values $\boldsymbol{c.e}$ resulting in a complexity of $\mathcal{O}(\boldsymbol{c.e})$.

In case a new value is added to a sorted dictionary, the value has to be inserted at the correct position into the collection. In the worst case, when the new value is the smallest, every other value has to be moved in order to free the first position. Therefore, the complexity for *addValue* in the sorted case is $\mathcal{O}(\boldsymbol{c.d} \cdot \boldsymbol{c.e})$. In case of an unsorted dictionary, the new value can be directly written to the end of the array, resulting in a complexity of $\mathcal{O}(\boldsymbol{c.e})$.

getValueForValueId searches in the dictionary for a given value, returning its value-id or a not found value in case the value is not in the dictionary. If the dictionary is sorted, the search can be performed with a binary search algorithm, resulting in logarithmic complexity of $\mathcal{O}(\log \boldsymbol{c.d} \cdot \boldsymbol{c.e})$. If the dictionary is unsorted, all values have to be scanned linearly, resulting in complexity of $\mathcal{O}(\boldsymbol{c.d} \cdot \boldsymbol{c.e})$.

Column Operations In the case of an uncompressed column, *getValueAt* can directly return the requested value. If the column is dictionary encoded, the value-id has to be retrieved first and then the requested value is returned. The complexity for the uncompressed case therefore results in $\mathcal{O}(\boldsymbol{c.e})$ and in a complexity for the dictionary encoded cases of $\mathcal{O}(\boldsymbol{c.e} \cdot \boldsymbol{c.e}_c)$.

appendValue on an uncompressed column is trivial, assuming the array has enough free space left, as the value is appended to the array, resulting in a complexity of $\mathcal{O}(\boldsymbol{c.e})$. In case of an unsorted dictionary, we first check whether the appended value is already in the dictionary. This check is performed with the dictionary operation *getValueForValueId* and linearly depends on $\boldsymbol{c.e}$ and $\boldsymbol{c.d}$. If the value is not in the dictionary, it is added with *addValue* and the value-id is appended to the column. The total complexity for the unsorted case is therefore in $\mathcal{O}(\boldsymbol{c.d} + \boldsymbol{c.e}_c + \boldsymbol{c.e})$. In case of a sorted dictionary, *getValueForValueId* has logarithmic costs and *addValue* depends linearly on $\boldsymbol{c.d}$. Additionally, the insertion of a new value-id may invalidate all old value-ids in the column, requiring a rewrite of the complete column and resulting in a complexity of $\mathcal{O}(\boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + \boldsymbol{c.e})$.

	Uncompressed	Dictionary Sorted	Dictionary Unsorted
ScanEqual	$\mathcal{O}(\boldsymbol{c.r} \cdot \boldsymbol{c.e} + q.s)$	$\mathcal{O}(\log \boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + q.s)$	$\mathcal{O}(\boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + q.s)$
ScanRange	$\mathcal{O}(\boldsymbol{c.r} \cdot \boldsymbol{c.e} + q.s)$	$\mathcal{O}(\log \boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + q.s)$	$\mathcal{O}(\boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + \boldsymbol{c.r} \cdot \boldsymbol{c.e} + q.s)$
Lookup	$\mathcal{O}(\boldsymbol{c.e})$	$\mathcal{O}(\boldsymbol{c.e}_c + \boldsymbol{c.e})$	$\mathcal{O}(\boldsymbol{c.e}_c + \boldsymbol{c.e})$
Insert	$\mathcal{O}(\boldsymbol{c.e})$	$\mathcal{O}(\boldsymbol{c.d} \cdot \boldsymbol{c.e} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c)$	$\mathcal{O}(\boldsymbol{c.d} \cdot \boldsymbol{c.e} + \boldsymbol{c.e}_c)$

Table 3.1: Complexity of plan operations by data structures.

3.4 Plan Operators

We now introduce the plan operators scan with equality selection, scan with range selection, positional lookup and insert and discuss their theoretical complexity. These operators were chosen, as we identified them as the most basic operators needed by a database system, assuming an insert only system as proposed in [58, 59, 46, 25]. Additionally, more complex operators can be assembled by combining these basic operators, as e.g. a nested loop join consisting of multiple scans. We differentiate between equality and range selections as they have different performance characteristics due to differences when performing value comparisons introduced by the dictionary encoding. Table 3.1 gives an overview of the asymptotic complexity of the four discussed operators on the different data structures.

3.4.1 Scan with Equality Selection

A scan with equality selection sequentially iterates through all values of a column and returns a list of positions where the value in the column equals the searched value. Algorithm 3.4.1 shows the implementation as pseudocode for the case of an uncompressed column. As the column is uncompressed, no decompression has to be performed and the values can be compared directly with the searched value. The costs for an equal scan on an uncompressed column are characterized by comparing all $\boldsymbol{c.r}$ values and by building the result set, resulting in $\mathcal{O}(\boldsymbol{c.r} \cdot \boldsymbol{c.e} + q.s)$.

Algorithm 3.4.1: ScanEqualUncompressed($X, column$)

$result \leftarrow array[], pos \leftarrow 0$
for each $value \in column$
do $\begin{cases} \textbf{if } value == X \\ \quad \textbf{then } result.append(pos) \\ pos \leftarrow pos + 1 \end{cases}$
return $(result)$

Algorithm 3.4.2: ScanEqualDictSorted($X, column, dictionary$)

$result \leftarrow array[], pos \leftarrow 0$
$valueId_X = dictionary.binarySearch(value)$
for each $valueId \in column$
do $\begin{cases} \textbf{if } valueId == valueId_X \\ \quad \textbf{then } result.append(pos) \\ pos \leftarrow pos + 1 \end{cases}$
return $(result)$

Algorithm 3.4.3: ScanEqualDictUnsorted($X, column, dictionary$)

$result \leftarrow array[], pos \leftarrow 0$
$valueID = dictionary.scanFor(value)$
for each $valueId \in column$
do $\begin{cases} \textbf{if } valueId == valueId_X \\ \quad \textbf{then } result.append(pos) \\ pos \leftarrow pos + 1 \end{cases}$
return $(result)$

We do not discuss a scan for inequality, as the implementation and performance characteristics are assumed to be the same as for a scan operator with equality selection.

Algorithm 3.4.2 shows how a scan with equality selection is performed on a column with a sorted dictionary. First, the value-id in the value dictionary of the column for the searched value x is retrieved by performing a binary search for x in the dictionary. Then, the value-ids of the column are scanned sequentially and each matching value-id is added to the set of results.

The costs for an equal scan on a column with a sorted dictionary consist of the binary search cost in the dictionary and comparing each value-id, resulting in $\mathcal{O}(\log \boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + q.s)$.

Algorithm 3.4.3 outlines a scan with equality selection on a column with an unsorted dictionary. As we are scanning the column searching for all occurrences of exactly one value, we can perform the comparison similarly as in the case of a sorted dictionary based only on

Algorithm 3.4.4: ScanRangeUncompressed($low, high, column$)

$result \leftarrow array[], pos \leftarrow 0$
for each $value \in column$
do **if** $(value > low)$ **and** $(value < high)$
then $result.append(pos)$
$pos \leftarrow pos + 1$

Algorithm 3.4.5: ScanRangeDictSorted($low, high, column, dictionary$)

$result \leftarrow array[], pos \leftarrow 0$
$valueId_{low} \leftarrow dictionary.binarySearch(low)$
$valueId_{high} \leftarrow dictionary.binarySearch(high)$
for each $valueId \in column$
do **if** $(valueId > valueId_{low})$ **and** $(valueId < valueId_{high})$
then $result.append(pos)$
$pos \leftarrow pos + 1$
return $(result)$

Algorithm 3.4.6: ScanRangeDictUnsorted($low, high, column, dictionary$)

$result \leftarrow array[], pos \leftarrow 0$
for each $valueId \in column$
do $value \leftarrow dictionary[valueId]$
if $(value > low)$ **and** $(value < high)$
then $result.append(pos)$
$pos \leftarrow pos + 1$
return $(result)$

the value-ids. The difference to the sorted dictionary case is that a linear search has to be performed instead of a binary search to retrieve the value-id of the searched value x.

Similar to the costs for a scan with equality selection on a column with a sorted dictionary, the costs on a column with an unsorted dictionary consist of the search costs for the scanned value in the dictionary and comparing each value-id. In contrast to the sorted dictionary case, the search costs are linear, resulting in a complexity of $\mathcal{O}(\boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + q.s)$.

3.4.2 Scan with Range Selection

A scan operator with range selection sequentially iterates through all values of a column and returns a list of positions, where the value in the column is between *low* and *high*. In contrast to a scan operator with an equality selection, it is searched for a range of values instead of one

Algorithm 3.4.7: LookupUncompressed($position, column$)

$value \leftarrow column[position]$
return ($value$)

Algorithm 3.4.8: LookupDictionary($position, column, dictionary$)

$valueId \leftarrow column[position]$
$value \leftarrow dictionary[valueId]$
return ($value$)

single value.

Algorithm 3.4.4 outlines the implementation of a range scan in pseudocode for an uncompressed column. Similarly to a scan operator with equality selection, we can perform the comparisons directly on the values while iterating sequentially through the column. Therefore, the costs are determined by the value length $\boldsymbol{c.e}$, the number of rows $\boldsymbol{c.r}$ and the selectivity $q.s$ of the scan, resulting in $\mathcal{O}(\boldsymbol{c.r} \cdot \boldsymbol{c.e} + q.s)$.

Algorithm 3.4.5 shows the implementation for the range scan on a dictionary encoded column with a sorted dictionary. First, the value-ids of *low* and *high* are retrieved with a binary search in the dictionary. As the dictionary is sorted, we know that $id_{low} > id_{high} \Rightarrow value(id_{low}) > value(id_{high})$. Therefore, we can scan the value-ids of the column and decide only by comparing with the value-ids of *low* and *high* if the current value-id has to be a part of the result set. The costs are similar to the costs for a scan with equality selection, determined by the binary search costs, the scanning of the column and building the result set, resulting in $\mathcal{O}(\log \boldsymbol{c.d} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + q.s)$.

Finally, algorithm 3.4.6 shows the implementation of a scan operator with range selection on a column with an unsorted dictionary. As the dictionary is unsorted, we can not draw any conclusions of the relations between two values based on their value-ids in the dictionary. We iterate sequentially through the value-ids of the column. For each value-id, we perform a lookup retrieving the actual value stored in the dictionary. The comparison can then be performed on that value with *low* and *high*.

In contrast to the sorted dictionary case, the costs in the unsorted case are determined by scanning the value-ids, performing the lookup in the dictionary and building the result set, resulting in a complexity of $\mathcal{O}(\boldsymbol{c.r} \cdot \boldsymbol{c.e}_c + \boldsymbol{c.r} \cdot \boldsymbol{c.e} + q.s)$.

Algorithm 3.4.9: InsertDictSorted($value, column, dictionary$)

$valueId \leftarrow dictionary.binarySearch(value)$
if $valueId = NotInDictionary$
  then $\begin{cases} valueId \leftarrow dictionary.insert(value) \\ \textbf{if } valueId < dictionary.size() \\ \quad \textbf{then } column.reencode() \\ column.append(valueId) \end{cases}$
  else $\{column.append(valueId)$

Algorithm 3.4.10: InsertDictUnsorted($position, column, dictionary$)

$valueId \leftarrow dictionary.scanFor(value)$
if $valueId = NotInDictionary$
  then $valueId \leftarrow dictionary.insert(value)$
$column.append(valueId)$

3.4.3 Lookup

A positional lookup retrieves the value of a given position from the column. The output is the actual value, as the position is already known. Algorithm 3.4.7 shows the lookup in case of an uncompressed column, where the value can be returned directly. The costs only depend on the value length, resulting in a complexity of $\mathcal{O}(\boldsymbol{c.e})$.

In the case of a dictionary encoded column, the algorithms for a positional lookup do not differ between a sorted and an unsorted dictionary. Therefore, algorithm 3.4.8 shows the lookup for a dictionary encoded column, where the value-id is retrieved from the requested position and a dictionary lookup is performed in order to retrieve the searched value. The costs depend on the compressed and the uncompressed value length, resulting in a complexity of $\mathcal{O}(\boldsymbol{c.e_c} + \boldsymbol{c.e})$.

3.4.4 Insert

An insert operation appends a new value to the column. As we keep the values always in insertion order, this can be implemented as a trivial append operation, assuming enough free and allocated space to store the inserted value. In the case of a dictionary encoded column, we have to check if the value is already in the dictionary.

Algorithm 3.4.9 outlines the insert operation for a column with a sorted dictionary. First, a

binary search is performed on the dictionary for value v. If v is not found in the dictionary, it is inserted so that the sort order of the dictionary is preserved. In case that v is not inserted at the end of the dictionary, a re-encode of the complete column has to be performed in order to reflect the updated value-ids of the dictionary. After the re-encode or if v was already found in the dictionary, the value-id is appended to the column. The complexity is in $\mathcal{O}(\boldsymbol{c.d} \cdot \boldsymbol{c.e} + \boldsymbol{c.r} \cdot \boldsymbol{c.e}_c)$.

Algorithm 3.4.10 shows the insertion of a new value for a column with an unsorted dictionary. Similarly to the sorted case, we first search for the inserted value in the dictionary by performing a linear search. As the dictionary is not kept in a particular order, the values are always appended to the end of the dictionary. Therefore, no re-encode of the column is necessary. The resulting complexity is $\mathcal{O}(\boldsymbol{c.d} \cdot \boldsymbol{c.e} + \boldsymbol{c.e}_c)$.

Chapter 4

Parameter Evaluation

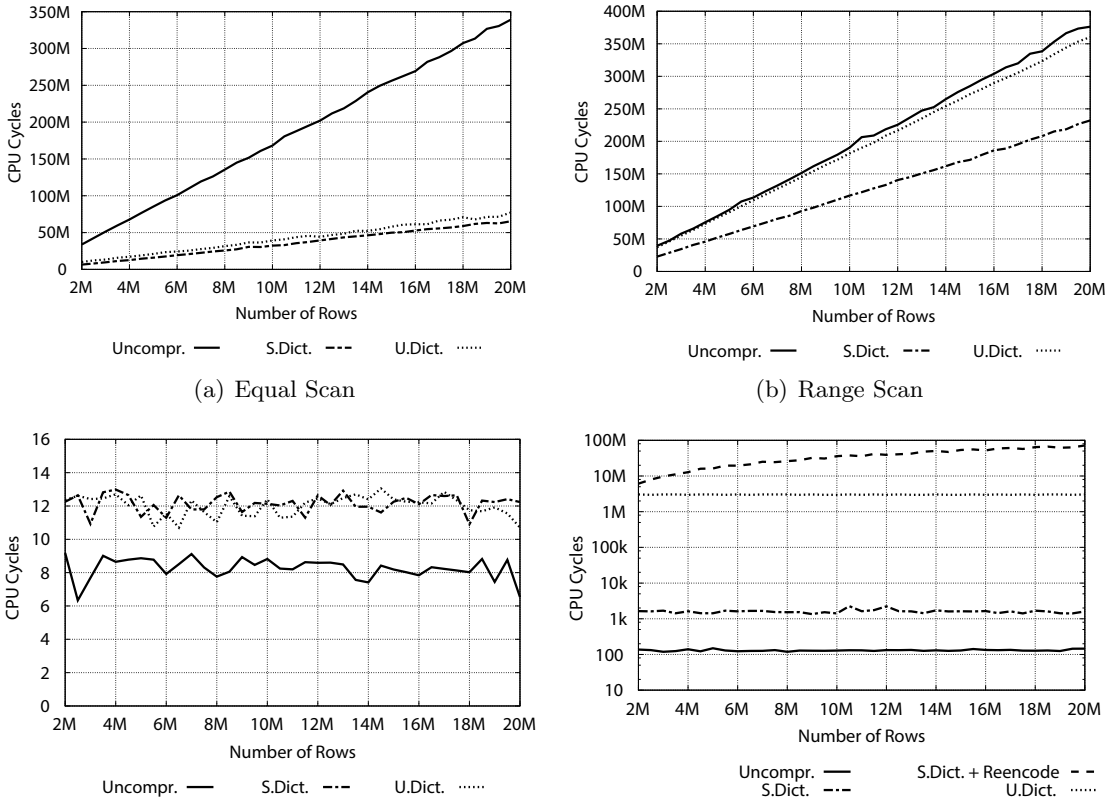
In the previous chapter, we defined plan operators and discussed their implementations and complexity depending on the parameters defined in Section 3.1. This chapter thrives to experimentally verify the theoretical discussion of the parameters and their influence on plan operations.

We implemented all operators in a self designed experimental system written in C++. All experiments were conducted on an Intel Xeon X5650, with 2x6 cores, hyper-threading, 2.67 GHz and 48 GB main memory. The system has 32 KB L1 data cache, 256 KB unified L2 cache, 12 MB unified L3 cache and a two level TLB cache with 64 entries in the L1 data TLB and 512 entries in the unified L2 TLB. The TLB caches are 4-way associative, the L1 and L2 cache are 8-way associative and the L3 cache is 16-way associative. The data was generated in such a way that all parameters were fixed except the varied parameter. We compiled our programs using Clang++ version 3.0 with full optimizations. In order to avoid scheduling effects, we pinned the process to a fixed CPU and increased the process priority to its maximum. We used the PAPI framework to measure cache misses and CPU cycles [17].

4.1 Number of Rows

We start by discussing the influence of the number of rows $\boldsymbol{c.r}$ on the plan operator performance.

Scan with Equality and Range Selection Figure 4.1(a) shows the time needed to perform a scan operator with equality selection on a column with the number of rows $\boldsymbol{c.r}$ varied from 2 million to 20 million. The time for the scan operation increases linearly with the number of rows, whereas the time per row stays constant. Similar to the scan with equality selection, Figure 4.1(b) shows the linear influence of the number of rows on the runtime for a scan with range selection.



(a) Equal Scan

(b) Range Scan

(c) Positional Lookup

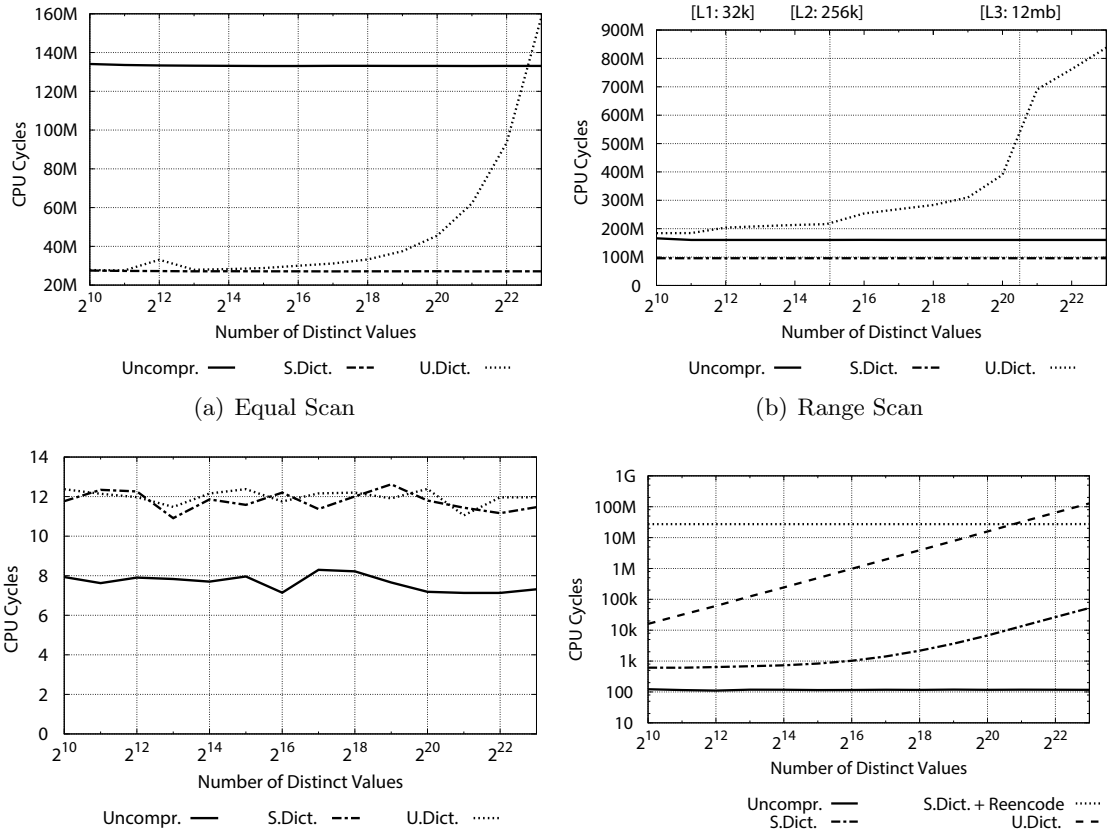
(d) Insert

Figure 4.1: CPU cycles for (a) equal scan, (b) range scan, (c) positional lookup and (d) insert on one column with number of rows $\boldsymbol{c.r}$ varied from 2 million to 20 million, $\boldsymbol{c.d}$ = 200,000, $\boldsymbol{c.u} = 0$, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.k} = 0$ and a query selectivity of $q.s$ = 2,000.

Positional Lookup For a positional lookup on a column, we expect the number of rows to have no influence on the performance on the lookup operation, as confirmed by Figure 4.1(c). The costs for performing a lookup on a dictionary encoded column is greater compared to an uncompressed column, as the value-id has to be retrieved first.

Insert For inserting new values into a column, we do expect the number of rows $\boldsymbol{c.r}$ to have no influence on the time an actual insert operation takes, regardless if the column is uncompressed or dictionary encoded. In case the column uses dictionary encoding with a sorted dictionary, we expect a linear influence on the re-encode operation by the number of rows in the column.

Figure 4.1(d) shows the number of CPU cycles for an insert operation with a varying number of rows on an uncompressed column, a dictionary encoded column with a sorted dictionary, a dictionary encoded column with a sorted dictionary and performing a re-encode



(a) Equal Scan

(b) Range Scan

(c) Positional Lookup

(d) Insert

Figure 4.2: CPU cycles for (a) equal scan, (b) range scan, (c) positional lookup and (d) insert on one column with number of distinct values $\boldsymbol{c.d}$ varied from 2^{10} to 2^{23}, $\boldsymbol{c.r} = 2^{23}$, $\boldsymbol{c.u} = 2^{23}$, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.k} = 0$ and a query selectivity of $q.s = 2{,}000$.

and a dictionary encoded column with an unsorted dictionary.

Inserting into an uncompressed column is very fast and not influenced by the number of rows in the column. In the case of a dictionary encoded column, the insert takes longer as it takes to search the dictionary for the inserted value. The large difference between the sorted and the unsorted dictionary is due to the effect of the binary search, which can be leveraged in the sorted dictionary case. Finally, we can see the linear impact of the number of rows on the insert with the re-encode operation – note the logarithmic scale.

4.2 Number of Distinct Values

We now focus on the number of distinct values $c.d$ of a column and their influence on the operators scan, insert and lookup.

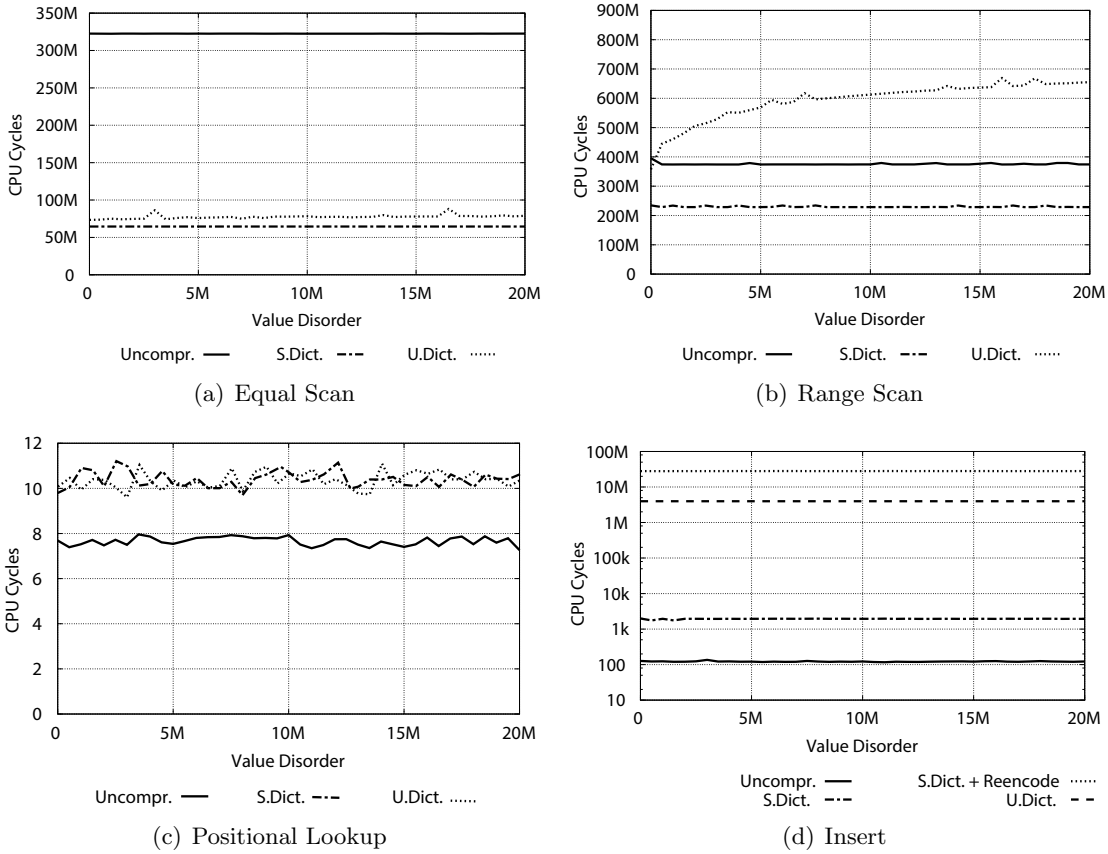
Scan with Equality Selection When scanning a column with an equality selection, we expect the number of distinct values to influence the dictionary encoded columns, but not the uncompressed column. Figure 4.2(a) shows the results of an experiment performing an equal scan on a column with 2^{23} rows and $c.d$ varied from 2^{10} to 2^{23}. We chose a selectivity of 2,000 rows, in order to keep the effect of writing the result set minimal. As expected, the runtime for the scan on the uncompressed column is not affected and we clearly see the linear impact on the unsorted dictionary column. However, the logarithmically increasing runtime for the column using a sorted dictionary is hard to recognize due to the large scale.

Scan with Range Selection In contrast to a scan operator with equality selection, the implementation of a range scan only differs in the case for an unsorted dictionary. The cases for an uncompressed column and a column with a sorted dictionary are the same as for a scan operator with equality selection, as Figure 4.2(b) shows.

For an unsorted dictionary encoded column, Figure 4.2(b) shows a strong impact of the varied number of distinct values on the runtime. Based on our earlier discussion of a scan operator with range selection on an unsorted dictionary, we would not expect this characteristic. The increase in CPU cycles with increasing distinct values is due to a cache effect. As $c.u = 2^{23}$, we access the dictionary in a random fashion while iterating over the column. As long as the dictionary is small and fits into the cache, these accesses are relatively cheap. With a growing number of distinct values the dictionary gets too large for the individual cache levels and the number of cache misses per dictionary access increases. Considering a value length of 8 bytes, we can identify jumps slightly before each cache level size of 32KB, 256KB and 12MB – e.g. at $c.e \cdot 2^{15} = 256$ KB, which are discussed in more detail in Section 5.2.

Lookup Figure 4.2(c) shows the influence of the number of distinct values when performing a positional lookup on one column. As we can see, the time for one single lookup is not influenced by the number of distinct values.

Insert Figure 4.2(d) shows the influence for the number of distinct values when inserting new values into a column. For an uncompressed column, the insert takes the same, independent of the number of distinct values. When using dictionary compression and an unsorted dictionary, the time for inserting a new value increases linearly with the numbers of distinct values as the dictionary is searched linearly for the new value. In case of a sorted dictionary without re-encoding the column, we notice a logarithmic increase with increasing values respective to the binary search in the dictionary. If the newly inserted value did change existing value-ids, the column has to be re-encoded. As we can see, the increase for binary searching the dictionary is negligible, as it is shadowed by the high amount of work for re-encoding the column.



(a) Equal Scan

(b) Range Scan

(c) Positional Lookup

(d) Insert

Figure 4.3: CPU cycles for (a) equal scan, (b) range scan, (c) positional lookup and (d) insert on one column with value disorder $\boldsymbol{c.u}$ varied from 0 to 20 million, $\boldsymbol{c.r} = 20$ million, $\boldsymbol{c.d} = 2$ million, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.k} = 0$ and a query selectivity of $q.s = 2{,}000$.

4.3 Value Disorder

This section evaluates the influence of the parameter value disorder $\boldsymbol{c.u}$ in a column on the discussed operators.

Scan with Equality Selection When performing a scan with equality selection, the comparisons can be done directly on the value-ids in case of a dictionary encoded column or are done directly on the values in case of an uncompressed column. Therefore, we do not expect the value disorder to influence the performance of an equal scan. Figure 4.3(a) confirms this assumption.

Scan with Range Selection Figure 4.3(a) shows the performance of a scan operator with range selection with varied disorder of values in the column. As expected, we see no influence

in case of an uncompressed column or a column with a sorted dictionary. In case of a dictionary encoded column with an unsorted dictionary, we see an increase in CPU cycles. In contrast to a scan with equality selection, a scan operator with range selection on an unsorted dictionary has to lookup the actual values in the dictionary in order to compare them. When the value disorder is low, temporal and spatial locality for the dictionary access is high, which results in good cache usage with a high number of cache hits. The greater the disorder, the more random accesses to the dictionary are performed, resulting in more CPU cycles for the scan operation.

Lookup and Insert We do expect the value disorder to have no influence on single positional lookups and inserts. Figure 4.3(c) and 4.3(d) confirm our assumptions.

4.4 Value Length

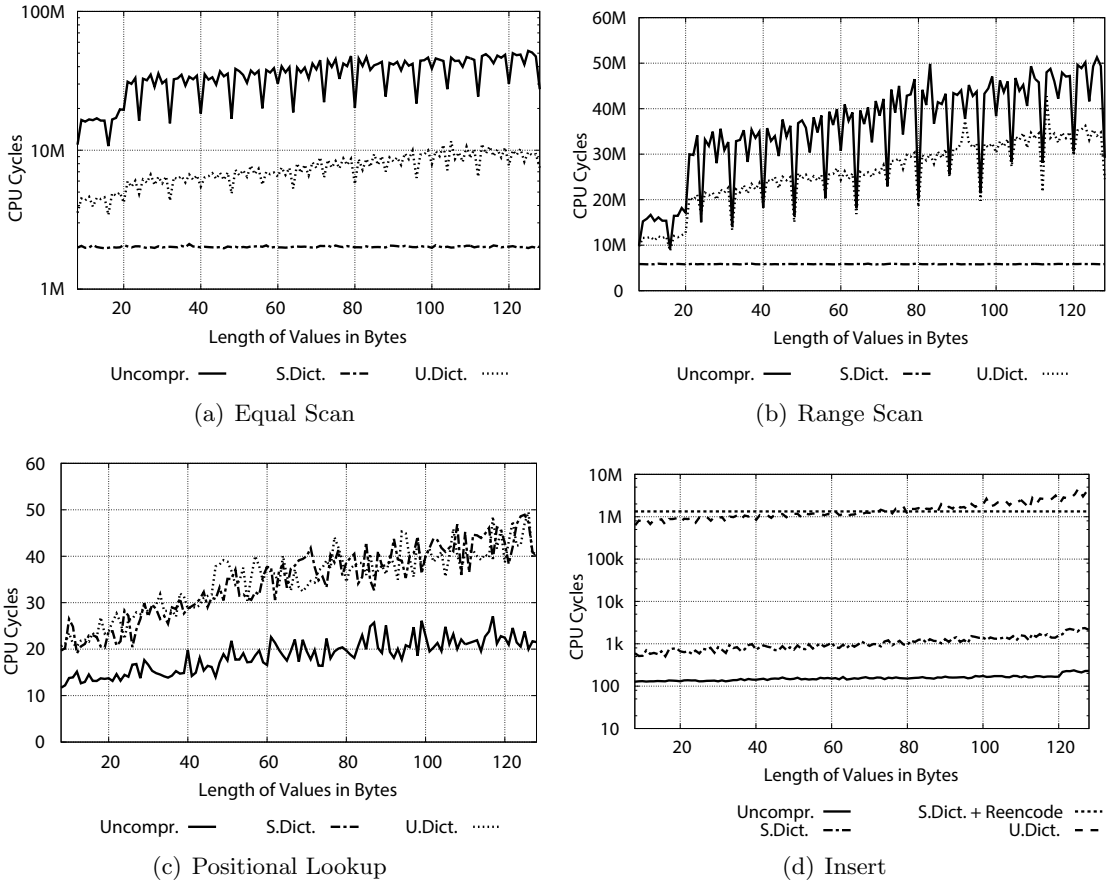
We now focus on the influence of the value length $c.e$ of the values in a column on the discussed operators.

Scan with Equality Selection Figure 4.4(a) shows the influence of an increased value length on a scan operator with equality selection. For uncompressed columns we see an increase in cycles for scan operators with longer values, as expected. However, we see a drop for every 16 bytes, due to alignment effects, showing how crucial it is to align memory correctly. In case of a dictionary compressed column, we see no significant influence in case of a sorted dictionary based on the value length. Although, the costs for the search do increase slightly, the increase is shadowed by the costs for scanning the value-ids. When using an unsorted dictionary, the costs for scanning the dictionary are significantly higher and we see a significant impact of the value length on the total scan costs.

Scan with Range Selection The impact of an increasing value length on a scan operator with range selection is similar to a scan with equality selection, as shown by Figure 4.4(b). In case of an uncompressed column, we see an increase in costs with larger value lengths and the same alignment effect, as the values are compared directly and larger values result in larger costs for comparing the values. We see no significant impact of the value length for the case of a sorted dictionary but an increase in case of an unsorted dictionary.

Positional Lookup The costs for a positional lookup do increase linearly with increasing size of values, as the actual values are returned by the lookup operation, resulting in more work for longer values as shown by Figure4.4(c)

Insert The costs for inserting new values always increase with larger values, as the costs for writing the values do increase. Figure 4.4(d) shows that in case of an uncompressed column the increase is quite small, which probably is due to block writing and write caching effects. The costs on a column with a sorted dictionary also increase slightly based on the increased costs for the binary search. However, in case a re-encode is necessary, the increase is shadowed



(a) Equal Scan

(b) Range Scan

(c) Positional Lookup

(d) Insert

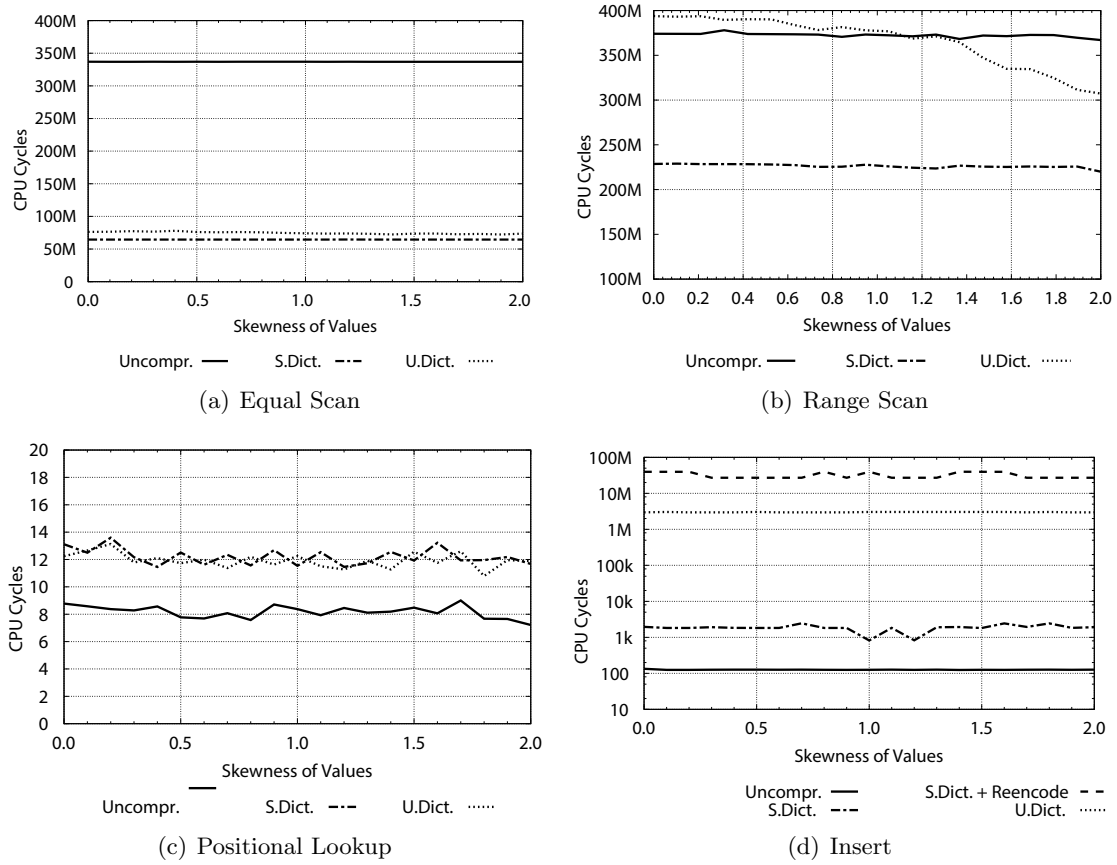
Figure 4.4: CPU cycles for (a) equal scan, (b) range scan, (c) positional lookup and (d) insert on one column with value length $\boldsymbol{c.e}$ varied from 8 to 128 byte, $\boldsymbol{c.r} = 512{,}000$, $\boldsymbol{c.d} = 100{,}000$, $\boldsymbol{c.u} = 512{,}000$, $\boldsymbol{c.k} = 0$ and a query selectivity of $q.s = 1{,}024$.

by the large costs for re-encoding the column. When using an unsorted dictionary, we also see a linear increase in costs with larger values.

4.5 Value Skewness

We now focus on the influence of the skewness $\boldsymbol{c.k}$ of the value distribution of the values in a column on the discussed operators.

Scan with Equality Selection, Lookup and Insert The skewness of values influences the pattern in which the dictionary of a column is accessed when scanning its value-ids and looking them up in the dictionary, the skewer the value distribution, the less cache misses



(a) Equal Scan

(b) Range Scan

(c) Positional Lookup

(d) Insert

Figure 4.5: CPU cycles for (a) equal scan, (b) range scan, (c) positional lookup and (d) insert on one column with value skewness $\boldsymbol{c.k}$ varied from 0 to 2, $\boldsymbol{c.r}$ = 20 million, $\boldsymbol{c.d}$ = 200,000, $\boldsymbol{c.u}$ = 20 million, $\boldsymbol{c.e}$ = 8 and a query selectivity of $q.s$ = 2,000.

occur. In case of a scan operator with equality selection, positional lookup or insert we do not have this pattern of scanning the column and accessing the dictionary. Therefore, we do not expect the skewness of values in a column to influence these operators, which is confirmed by the experimental results shown in Figures 4.5(a),4.5(c) and 4.5(d).

Scan with Range Selection In contrast, Figure 4.5(b) shows the influence of the value skewness on a scan operator with range selection. In case of a dictionary encoded column with an unsorted dictionary, we scan the value-ids of the column sequentially and randomly access the value dictionary (value disorder $\boldsymbol{c.u}$ = 20 million). The skewer the value distribution is, the more likely it gets that a value with a high frequency is accessed and is still in the cache. Therefore, the number of cache misses is reduced for skewed value distributions, resulting in a faster execution of the scan operator.

4.6 Conclusions

In this section, we presented a detailed parameter evaluation for the in Chapter 3 presented algorithms. We presented a set of experimental results, giving an overview of the influences of our defined parameters on the operators scan with equality selection, scan with range selection, positional lookup and insert.

Additionally, we discussed and confirmed the expected influences based on the complexity discussion of the operators presented in Section 3.4 and confirmed our expectations. However, we found some cases like the influence of the number of distinct values or the effect of the value skewness on a scan operator with range selection in case an unsorted dictionary is used, that we did not predict based on the discussion of complexity. These influences are based on caching effects, which will be discussed in detail in the following chapter.

Chapter 5

Estimating Cache Misses

In our experimental validation of the parameter influences on the operators scan with equality selection, scan with range selection, insert and lookup we found significant influences depending on parameters introduced by cache effects. First, this chapter describes the memory hierarchy on modern computer systems and discusses why the usage of caches is so important for the performance of applications. Second, we introduce a cost model to predict the number of cache misses for plan operators and to estimate their execution time.

5.1 Background on Caches

In early computer systems, the frequency of the Central Processing Unit (CPU) was the same as the frequency of the memory bus and register access was only slightly faster than memory access. However, CPU frequencies did heavily increase in the last years following Moores Law [55], but frequencies of memory buses and latencies of memory chips did not grew with the same speed. As a result, memory access gets more expensive, as more CPU cycles are wasted while stalling for memory access.

This development is not due to the fact that fast memory can not be built, it is an economical decision as memory, which is as fast as current CPUs, would be orders of magnitude more expensive and would require extensive physical space on the boards. In general, memory designers have the choice between Static Random Access Memory (SRAM) and Dynamic Random Access Memory (DRAM).

5.1.1 Memory Cells

SRAM cells are usually built out of six transistors (although variants with only 4 do exist but have disadvantages [53, 18]) and can store a stable state as long as the power is supplied. Their

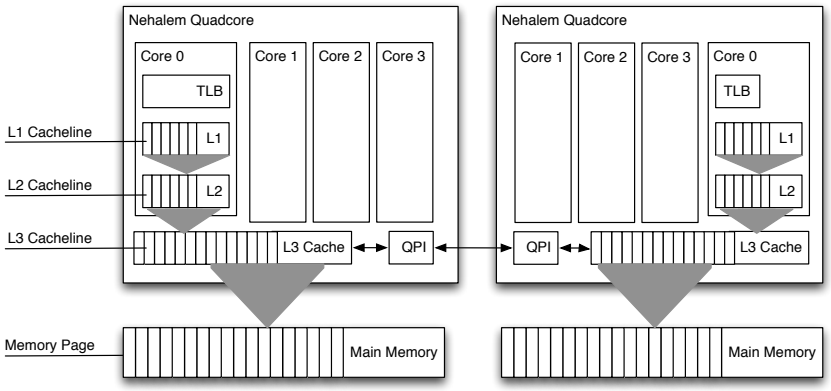


Figure 5.1: Memory Hierarchy on Intel Nehalem architecture.

stored state is immediately available for reading and no synchronization or wait periods have to be considered.

In contrast, DRAM cells can be constructed using a much simpler structure consisting of only one transistor and a capacitor. The state of the memory cell is stored in the capacitor while the transistor is only used to guard the access to the capacitor. This design is more economical compared to SRAM. However, it introduces a couple of complications. First, the capacitor discharges over time while reading the state of the memory cell. Therefore, today's systems refresh DRAM chips every 64 ms and after every read of the cell in order to recharge the capacitor [16]. During the refresh, no access to the state of the cell is possible. The charging and discharging of the capacitor takes time, which means that the current can not be detected immediately after requesting the stored state, therefore limiting the speed of DRAM cells.

In a nutshell, SRAM is fast but expensive as it requires a lot of space. In contrast, DRAM chips are slower but cheaper as they allow larger chips due to their simpler structure. For more details regarding the two types of Random Access Memory (RAM) and their physical realization the interested reader is referred to [18].

5.1.2 Memory Hierarchy

An underlying assumption of the memory hierarchy of modern computer systems is a principle known as *data locality* [29]. Temporal data locality indicates that accessed data is likely to be accessed again soon, whereas spatial data locality indicates that data stored together in memory is likely to be accessed together. These principles are leveraged by using caches, combining the best of both worlds by leveraging the fast access to SRAM chips and the sizes made possible by DRAM chips. Figure 5.1 shows a hierarchy of memory on the example of the Intel Nehalem architecture. Small and fast caches close to the CPUs built out of SRAM cells cache accesses to the slower main memory built out of DRAM cells. Therefore, the hierarchy

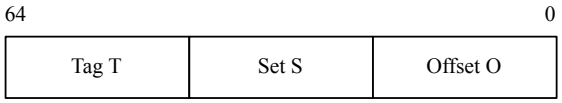


Figure 5.2: Parts of a memory address.

consists of multiple levels with increasing storage sizes but decreasing speed. Each CPU core has its private L1 and L2 cache and one large L3 cache shared by the cores on one socket. Additionally, the cores on one socket have direct access to their local part of main memory through an Integrated Memory Controller (IMC). When accessing other parts than their local memory, the access is performed over a Quick Path Interconnect (QPI) controller coordinating the access to the remote memory.

The first level is the actual registers inside the CPU, used to store inputs and outputs of the processed instructions. Processors usually only have a small amount of integer and floating point registers which can be accessed extremely fast. When working with parts of the main memory, their content has to be first loaded and stored in a register to make it accessible for the CPU. However, instead of accessing the main memory directly the content is first searched in the Level 1 Cache (L1 Cache). If it is not found in L1 Cache it is requested from Level 2 Cache (L2 Cache). Some systems even make use of a Level 3 Cache (L3 Cache).

5.1.3 Cache Internals

Caches are organized in cache lines, which are the smallest addressable unit in the cache. If the requested content can not be found in any cache, it is loaded from main memory and transferred down the hierarchy. The smallest transferable unit between each level is one *cache line*. Caches where every cache line of level i is also present in level $i+1$ are called *inclusive caches*, otherwise they are called *exclusive caches*. All Intel processors implement an inclusive cache model, which is why an inclusive cache model is assumed for the rest of this report.

When requesting a cache line from the cache, the process of determining if the requested line is already in the cache and locating where it is cached is crucial. Theoretically, it is possible to implement fully associative caches, where each cache line can cache any memory location. However, in practice this is only realizable for very small caches as a search over the complete cache is necessary when searching for a cache line. In order to reduce the search space, the concept of a *n-way set associative cache* with associativity A_i divides a cache of size C_i into $C_i/B_i/A_i$ sets and restricts the number of cache lines which can hold a copy of a certain memory address to one set or A_i cache lines. Thus, when determining if a cache line is already present in the cache, only one set with A_i cache lines has to be searched.

For determining if a requested address is already cached, the address is split into three parts as shown by Figure 5.2. The first part is the offset O, which size is determined by the cache line size of the cache. So with a cache line size of 64 byte, the lower 6 bits of the address would

be used as the offset into the cache line. The second part identifies the cache set. The number s of bits used to identify the cache set is determined by the cache size C_i, the cache line size B_i and the associativity A_i of the cache by $s = \lceil \log_2(C_i/B_i/A_i) \rceil$. The remaining $64 - o - s$ bits of the address are used as a tag to identify the cached copy. Therefore, when requesting an address from main memory, the processor can calculate S by masking the address and then search the respective cache set for the tag T. This can be easily done by comparing the tags of the A_i cache lines in the set in parallel.

5.1.4 Address Translation

The operating system provides each process a dedicated continuous address space, containing an address range from 0 to 2^x. This has several advantages as the process can address the memory through virtual addresses and does not have to bother with the physical fragmentation. Additionally, memory protection mechanisms can control the access to memory and restrict programs to access memory which was not allocated by them.

Another advantage of virtual memory is the use of a paging mechanism, which allows a process to use more memory than is physically available by paging pages in and out and saving them on secondary storage. The continuous virtual address space of a process is divided into pages of size p, which is on most operating system 4 KB. Those virtual pages are mapped to physical memory. The mapping itself is saved in a page table, which resides in main memory itself. When the process accesses a virtual memory address, the address is translated by the operating system into a physical address with help of the memory management unit inside the processor.

The address translation is usually done by a multi-level page table, where the virtual address is split into multiple parts, which are used as an index into the page directories resulting in a physical address and a respective offset. As the page table is kept in main memory, each translation of a virtual address into a physical address would require additional main memory accesses or cache accesses in case the page table is cached.

In order to speed up the translation process, the computed values are cached in the Transaction Look-Aside Buffer (TLB), which is a small and fast cache. When accessing a virtual address, the respective tag for the memory page is calculated by masking the virtual address and the TLB is searched for the tag. In case the tag is found, the physical address can be retrieved from the cache. Otherwise, a TLB miss occurs and the physical address has to be calculated, which can be quite costly. Details about the address translation process, TLBs and paging structure caches for Intel 64 and IA-32 architectures can be found in [36].

The costs introduced by the address translation scale linearly with the width of the translated address [29, 15], therefore making it hard or impossible to built large memories with very small latencies.

5.1.5 Prefetching

Modern processors try to guess which data will be accessed soon and initiate loads before the data is accessed in order to reduce the incurring access latencies. Good prefetching can completely hide the latencies so that the data is already in the cache when accessed. However, if data is loaded which is not accessed later it can also evict data inducing additional misses. Processors support software and hardware prefetching. Software prefetching can be seen as a hint to the processor, indicating which addresses are accessed next. Hardware prefetching automatically recognizes access patterns by utilizing different prefetching strategies. The Intel Nehalem architecture contains two second level cache prefetcher – the L2 streamer and data prefetch logic (DPL) [37]. The prefetching mechanisms only work inside the page boundaries of 4KB, in order to avoid triggering expensive TLB misses.

5.2 Cache Effects on Application Performance

The described caching and virtual memory mechanisms are implemented as transparent systems from the viewpoint of an actual application. However, knowing the system and its characteristics can have crucial implications on application performance.

5.2.1 The Stride Experiment

As the name random access memory suggests, the memory can be accessed randomly and one would expect constant access costs. In order to test this assumption we ran a simple benchmark accessing a constant number (4,096) of addresses with an increasing stride between the accessed addresses. We implemented this benchmark by iterating through an array chasing a pointer.

The array is filled with structs so that following the pointer of the elements creates a circle through the complete array. The structs consist of a pointer and an additional data attribute realizing the padding in memory, resulting in a memory access with the desired stride when following the pointer chained list. In case of a sequential array, the pointer of element i points to element $i + 1$ and the pointer of the last element references the first element so that the circle is closed. In case of a random array, the pointer of each element points to a random element of the array while ensuring that every element is referenced exactly once.

If the assumption holds and random memory access costs are constant, then the size of the padding in the array and the array layout (sequential or random) should make no difference when iterating over the array. Figure 5.3 shows the result for iterating through a list with 4,096 elements, while following the pointers inside the elements and increasing the padding between the elements. As we can clearly see, the access costs are not constant and increase with an increasing stride. We also see multiple points of discontinuity in the curves, e.g. the

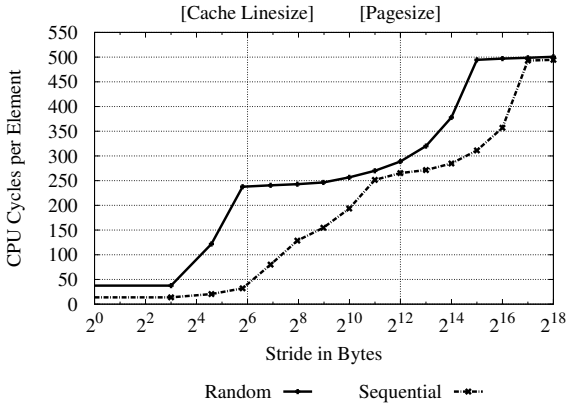
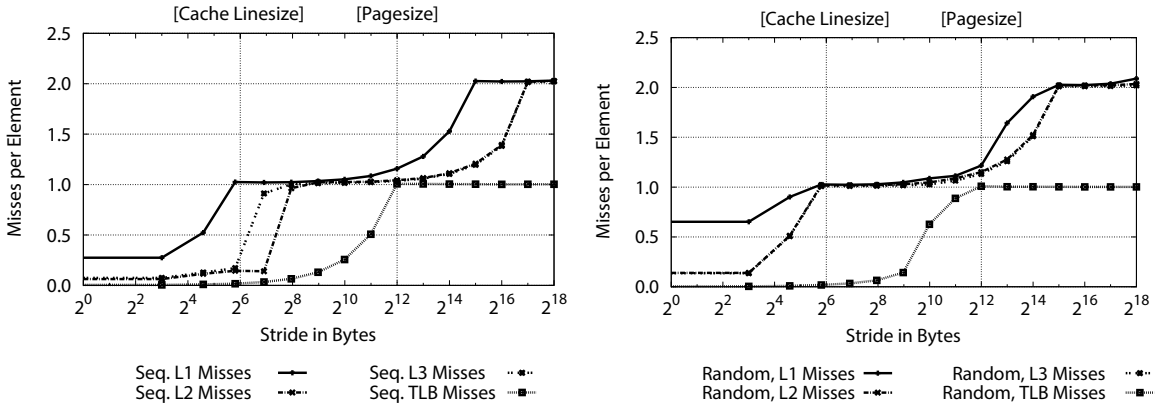


Figure 5.3: Cycles for cache accesses with increasing stride.



(a) Sequential Access

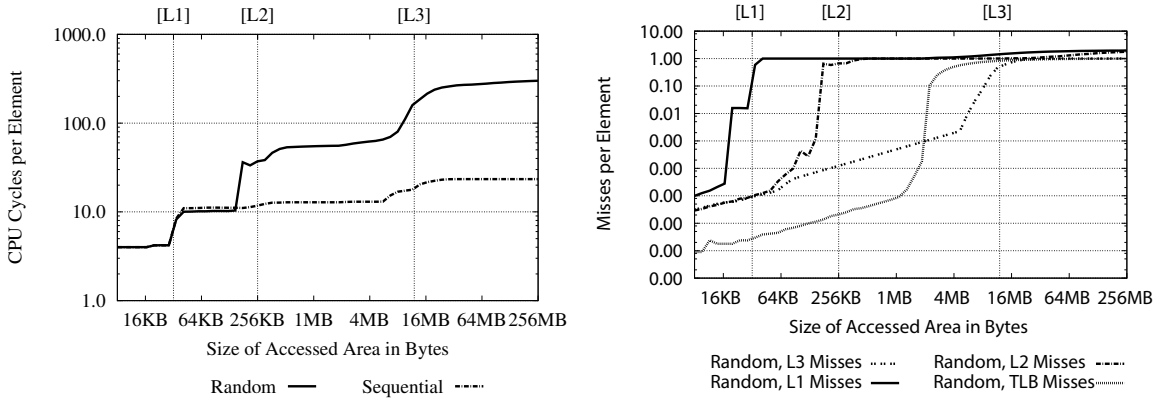
(b) Random Access

Figure 5.4: Cache misses for cache accesses with increasing stride.

random access times increase heavily up to a stride of 64 byte and continue increasing with a smaller slope.

Figure 5.4 confirms the assumption, that an increasing number of cache misses is causing the increase in access times. The first point of discontinuity in Figure 5.3 is roughly the size of the cache lines of the test system. The strong increase is due to the fact, that with a stride smaller than 64 byte, multiple list elements are located on one cache line and the overhead of loading one line is amortized over the multiple elements.

For strides greater than 64 byte, we would expect a cache miss for every single list element and no further increase in access times. However, as the stride gets larger, the array is placed over multiple pages in memory and more TLB misses occur, as the virtual addresses on the new pages have to be translated into physical addresses. The number of TLB cache misses increases up to the page size of 4 KB and stays at its worst case of one miss per element.



(a) Sequential Access (b) Random Access

Figure 5.5: Cycles and cache misses for cache accesses with increasing working sets.

With strides greater as the page size, the TLB misses can induce additional cache misses when translating the virtual to a physical address. These cache misses are due to accesses to the paging structures, which reside in main memory [4, 3, 64].

5.2.2 The Size Experiment

In a second experiment, we access a constant number of addresses in main memory with a constant stride of 64 byte and vary the size of the working set size or accessed area in memory. A run with n memory accesses and a working set size of s byte would iterate $\frac{n}{s \cdot 64}$ times through the array.

Figure 5.5(a) shows that the access costs differ up to a factor of 100 depending on the working set size. The points of discontinuity correlate with the sizes of the caches in the system. As long as the working set size is smaller than the size of the L1 Cache, only the first iteration results in cache misses and all other accesses can be answered out of the cache. As the working set size increases, the accesses in one iteration start to evict the earlier accessed addresses, resulting in cache misses in the next iteration.

Figure 5.5(b) shows the individual cache misses with increasing working set sizes. Up to working sets of 32 KB the misses for the L1 Cache go up to one per element, the L2 cache misses reach their plateau at the L2 cache size of 256 KB and the L3 cache misses at 12 MB.

5.3 A Cache-Miss Based Cost Model

In traditional disk based systems, IO operations are counted and used as a measurement for costs. For in-memory database systems IO operations are not of interest and the focus shifts to

description	unit	symbol
cache level	-	i
cache capacity	[byte]	C_i
cache block size	[byte]	B_i
number of cache lines	-	$\#i = C_i/B_i$

Table 5.1: Cache Parameters with $i \in \{1, ..., N\}$

main memory accesses. The initial examples from Section 5.2, measuring memory access costs for varying strides and working set sizes, showed that memory access costs are not constant and an essential factor of the overall costs. In general, assuming all data is in main memory, the total execution time of an algorithm can be separated into computing time and time for accessing data in memory [51].

$$T_{Total} = T_{CPU} + T_{Mem} \tag{5.1}$$

In case the considered algorithms are close to bandwidth bound, T_{Mem} is the dominant factor driving the execution time. Additionally, modeling T_{CPU} requires internal knowledge of the used processor, is very implementation specific and also dependent on the resulting machine code created by the compiler which makes it extremely hard to model. As our considered operators on the various data structures only perform a small amount of computations while accessing large amounts of data, we assume our algorithms to be bandwidth bound and believe T_{Mem} to be a good estimation of overall costs.

As discussed in Section 5.2, the costs for accessing memory can vary significantly due to the underlying memory hierarchy and mechanisms like prefetching and virtual address translation. Therefore, the costs for accessing memory can be quantified by the number of cache misses on each level in the memory hierarchy, assuming constant costs for each cache miss.

In this section, we will provide explicit functions to calculate the estimated number of cache misses for each operator and column organization. Some cost functions are based on the work presented by Manegold and Kersten [51], presenting a generic cost model to estimate the execution times of algorithms based on their cache misses by modeling basic access patterns and an algebra to combine basic patterns to model more complex ones. However, we develop our own cost functions, as they are specifically designed for the operators and column organization schemes.

We develop parameterized functions estimating the number of cache misses for each operation. With the specific parameters for each cache level, the cache misses on that level can be predicted. Furthermore, the total costs can be calculated by multiplying the number of cache misses with the latency of the next level in the hierarchy as proposed in [51]. Measuring the

individual cache level latencies requires accurate calibration and is very system specific. As a simpler and more robust estimation, we use the number of cache misses as a direct indicator for the resulting number of cycles, only roughly weighting the different cache levels. The cache level in the hierarchy is indicated by i, whereas the TLB is treated as an additional level in the hierarchy. The cache line size or block size of a respective level is given by B_i and the size by C_i. The number of cache lines at the level i is denoted by $\#i$. Table 5.1 gives an overview of the required parameters of the memory hierarchy.

The function M_i(o, c) describes the estimated amount of cache misses for an operation o on a column c. The operations are *escan*, *rscan*, *lookup* and *insert*. The respective physical column organization is given by a subscript indicating A) an uncompressed column, B) a dictionary encoded column with an unsorted dictionary and C) a dictionary encoded column with a sorted dictionary.

5.3.1 Scan with Equality Selection

We start by developing functions estimating the number of cache misses for a scan operator with equality selection on an uncompressed column. The scan consists of sequentially iterating over the column, resulting in one random miss and as many sequential misses as the column covers cache lines.

$$\mathbf{M_i}(escan_A, c) = \left\lceil \frac{\boldsymbol{c.r} \cdot \boldsymbol{c.e}}{B_i} \right\rceil$$

In case the column is dictionary encoded with a sorted dictionary, the binary search for the searched value results in log_2 random cache misses, while the sequential scan over the compressed value-ids results in as many cache misses as the compressed column covers cache lines.

$$\mathbf{M_i}(escan_B, c) = \left\lceil \frac{\boldsymbol{c.r} \cdot \boldsymbol{c.e}_c}{B_i} \right\rceil + \lceil \log_2(\boldsymbol{c.d}) \rceil \cdot \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil$$

The number of cache misses for an equal scan on a column with an unsorted dictionary is similar as with a sorted dictionary, but instead of a binary search a linear search is performed, scanning in average half the dictionary.

$$\mathbf{M_i}(escan_C, c) = \left\lceil \frac{\boldsymbol{c.r} \cdot \boldsymbol{c.e}_c}{B_i} \right\rceil + \left\lceil \frac{\boldsymbol{c.d} \cdot \boldsymbol{c.e}}{2 \cdot B_i} \right\rceil$$

Figure 5.6(a) shows the number of L1 cache misses of an experiment performing a scan with equality selection on a column while varying the number of rows from 2 million to 20 million. The experimental results are compared to the predictions calculated based on the presented cost functions. The predicted cache misses follow closely the measured number of misses.

5.3.2 Scan with Range Selection

In case of an uncompressed column, a scan with range selection iterates sequentially over the uncompressed values, comparing the values with the requested range. Similarly, in case of a sorted dictionary, the searched values are retrieved from the dictionary with a binary search and the value-ids are scanned sequentially for the searched value-ids. In both cases, the resulting cache misses are the same as for a scan with equality selection.

In case of an unsorted dictionary, the scan operation sequentially iterates over the column and has to perform a random access into the dictionary due to the range selection. Regarding the random access into the dictionary, we assume that every value in the dictionary is accessed at least once. In the best case, the access to the dictionary is sequentially, utilizing all values in a cache-line. In the worst case, every access to the dictionary may result in a cache miss. The number of cache misses increases with increasing dictionary sizes respective to the cache size and the amount of disorder in the column. Therefore, we model the number of random misses by interpolating between 0 and the number of rows in the column.

In order to smoothly interpolate between two values, we define the following helper functions. I_l is a simple linear interpolation function between y_0 and y_1, whereas t varies from 0 to 1. Furthermore, we define I_d as a decelerating interpolation function.

$$I_l(y_0, y_1, t) = y_0 + t \cdot (y_1 - y_0)$$

$$I_d(y_0, y_1, t) = I_l(y_0, y_1, 1 - (1 - t)^2)$$

Based on I_l and I_d, we construct I_c as a cosinus-based interpolation function to smoothly interpolate between two values, as we found this interpolation type to fit well to the cache characteristics.

$$I_c(y_0, y_1, t) = I_l\Big(y_0, y_1, \frac{1 - \cos(\pi \cdot I_d\langle 0, 1, t\rangle)}{2}\Big)$$

Finally, we introduce I as a helper function modeling a function stepping smoothly from y_0 to y_1 around a location of x_0, whereas ρ indicates the range in which the interpolation and τ the degree of how asymmetric the interpolation is performed. These values might be system specific and can be calibrated as needed.

$$I(x, x_0, y_0, y_1, \rho, \tau) = \begin{cases} y_0 : x < 2^{x_0 - \rho} \\ y_1 : x \geq 2^{x_0 + \rho * \tau} \\ I_c(y_0, y_1, \frac{log_2(x) - x_0 + \rho}{\rho * (\tau + 1)}) : else \end{cases}$$

If the number of covered cache-lines C_i is smaller than the number of available cache-lines $\#_i$, every cache-line is loaded at its first access and remains in the cache. For subsequent accesses, this cache-line is already in the cache and the access does not create an additional cache miss.

If $C_i > \#_i$, then already loaded cache-lines may be evicted from cache by loading other cache-lines. Subsequent accesses then have to load the same cache-line again, producing more cache misses.

The worst case is that every access to a cache-line has to load the line again, because it was already evicted, resulting in $\boldsymbol{c.r}$ cache misses. Assuming randomly distributed values in a column, how often cache-lines are evicted depends on the ratio of the number of cache-lines $\#_i$ and the number of covered cache-lines C_i. With increasing C_i the probability that cache-lines are evicted before they are accessed again increases. This results in the number of random cache misses of:

$$\mathbf{M_i^r}(rscan_C, c) = I(\boldsymbol{c.d}, \log_2(C_i), 0, \boldsymbol{c.r}, \rho, \tau)$$

The number of sequential cache misses depends on the success of the prefetcher. In case no or only a few random cache misses occur, the prefetcher has not enough time to load the requested cache lines, resulting in sequential misses. With increasing numbers of random cache misses, the time window for prefetching increases, resulting in less sequential cache misses.

Assuming a page size of 4KB, we found $\mathbf{M_i^s}$ to be a good estimation, as a micro benchmark turned out that every three random cache misses when accessing the dictionary leave the prefetcher enough time to load subsequent cache lines. We calculate the number of sequential cache misses as follows:

$$\mathbf{M_i^s}(rscan_C, c) = \max\left(\left\lceil \frac{\mathbf{C}(i,c)}{4096} \right\rceil, \mathbf{C}(i,c) - \frac{\mathbf{M_i^r}}{3}\right)$$

Additionally, we also have to consider extra penalties payed for TLB misses. In case an address translation misses the TLB and the requested page table entry is not present in the respective cache level, another cache miss occurs. In the worst case, this can introduce an additional cache miss for every dictionary lookup. Therefore, we calculate the number of TLB misses by:

$$\mathbf{M_i^{tlb}}(rscan_C, c) = I(\boldsymbol{c.d}, \log_2(C_{tlb} \cdot 4^i), 0, \boldsymbol{c.r}, \rho, \tau)$$

Finally, the total number of cache misses for a scan operation with range selection on a column with an uncompressed dictionary is given by adding random, sequential and TLB misses. Figure 5.6(b) shows a comparison of the measured effect of an increasing number of distinct values on a range scan on an uncompressed column with the predictions based on the provided cost functions. The figure shows the number of cache misses for each level and the model correctly predicts the jumps in the number of cache misses.

5.3.3 Lookup

A lookup on an uncompressed column results in as many cache misses as one value covers cache lines on the respective cache level.

$$\mathbf{M_i}(lookup_A, c) = \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil$$

In case the column is dictionary encoded it makes no difference if the lookup is performed on a column with a sorted or an unsorted dictionary, hence we provide one function $\mathbf{M_i}(lookup_{B/C})$ for both cases.

$$\mathbf{M_i}(lookup_{B/C}, c) = \left\lceil \frac{\boldsymbol{c.e_c}}{B_i} \right\rceil + \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil$$

Figure 5.6(c) shows a comparison of the predicted number of L1 cache misses and the experimental results for a lookup operation while varying the number of rows and that the prediction. The predicted number of cache misses closely matches the experimental results.

5.3.4 Insert

The insert operation is the only operation we consider writing to main memory. Although it is not quite accurate, we will treat write access similar as reading from main memory and consider the resulting cache misses.

$$\mathbf{M_i}(insert_A, c) = \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil$$

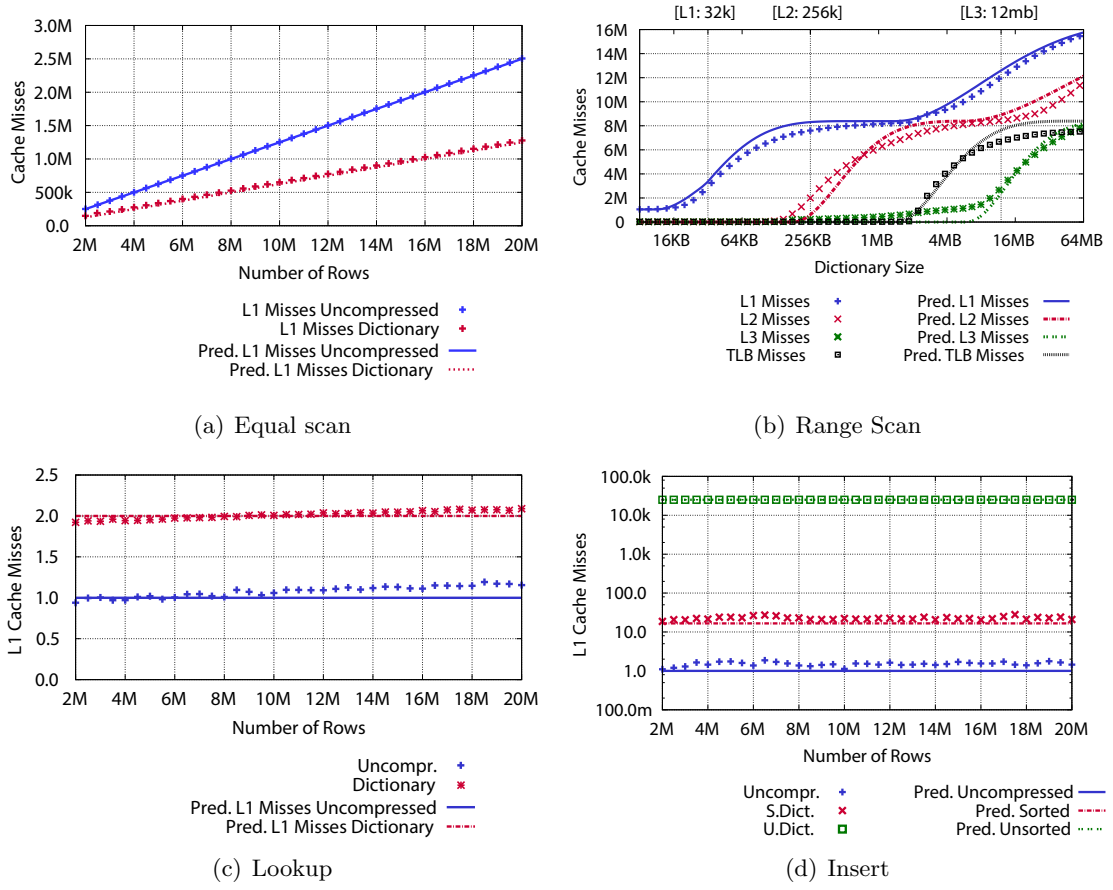
In case we perform an insert into a column with a sorted dictionary, we first perform a binary search determining if the value is already in the dictionary, before writing the value and value-id, assuming the value was not already in the dictionary.

$$\mathbf{M_i}(insert_B, c) = \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil + \left\lceil \frac{\boldsymbol{c.e_c}}{B_i} \right\rceil + \lceil \log_2 (\boldsymbol{c.d}) \rceil \cdot \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil$$

The number of cache misses in the unsorted dictionary case are similar to the sorted dictionary case, although the cache misses for the search depend linearly on the number of distinct values.

$$\mathbf{M_i^s}(insert_C, c) = \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil + \left\lceil \frac{\boldsymbol{c.e_c}}{B_i} \right\rceil + \left\lceil \frac{\boldsymbol{c.d}}{2} \right\rceil \cdot \left\lceil \frac{\boldsymbol{c.e}}{B_i} \right\rceil$$

Figure 5.6(d) shows the predicted number of cache misses for an insert operation and that the prediction closely matches the experimental results.



(a) Equal scan

(b) Range Scan

(c) Lookup

(d) Insert

Figure 5.6: Evaluation of predicted cache misses. Predicted cache misses for (a) equal scan, (b) range scan, (c) lookup and (d) insert. For (a), (c) and (d), the number of rows $\boldsymbol{c.r}$ was varied from 2 million to 20 million, $\boldsymbol{c.d} = 200{,}000$, $\boldsymbol{c.u} = 0$, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.k} = 0$ and a query selectivity of $q.s = 2{,}000$. For (b), the number of distinct values was varied from 1024 to 2^{23}, $\boldsymbol{c.r} = 2^{23}$, $\boldsymbol{c.u} = 2^{23}$, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.k} = 0$.

Chapter 6

Index Structures

This chapter discusses the influence of index structures on top of the evaluated data structures and their influence on the discussed plan operators. First, we extend the unsorted dictionary case by adding a tree structure on top, keeping a sorted order and allowing binary searches. Second, we discuss the influence of inverted indices on columns and extend our model to reflect these changes.

Indices have been studied by many researchers and database literature uses the term index in many ways. Some disk based database systems define an index as an alternative sort order of a table on one or more attributes. Thus, by leveraging the index structure, a query can search a table in logarithmic complexity with binary search. In disk based systems, it is often assumed that the index structure is in memory and accessing it is cheap, as accessing the relation stored on secondary storage is the main cost factor.

In the field of text search and search engines, an inverted index maps words to documents, so for every word a list of matching document-ids is maintained. The index is called *inverted*, as a document traditionally is a list of words and the index enables a mapping from words to documents. Recent literature mentions inverted indices [59] for main memory column stores. However, we think the term *inverted index* is misleading at that point as a classical index also provides the mapping from values to record-ids.

In this report, we assume an index to be a data structure that allows us to efficiently find tuples satisfying a given search condition. As tuples are stored in insertion order, we assume an index to be a separate auxiliary data structure on top of a column, not affecting the placement of values inside the column. Furthermore, we distinguish between column indices and dictionary indices. A column index is built on top of the values of one column, e.g. by creating a tree structure to enable binary search on the physically unsorted values in the column. In contrast, a dictionary index is a B^+-Tree built only on the distinct values of a column, enabling binary searching an unsorted dictionary in order to find the position of a given value in the dictionary.

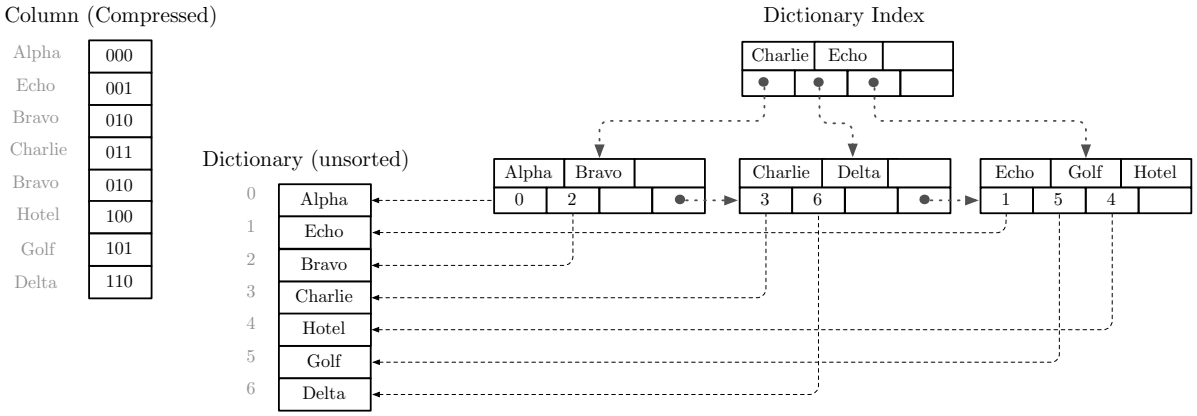


Figure 6.1: Example dictionary index.

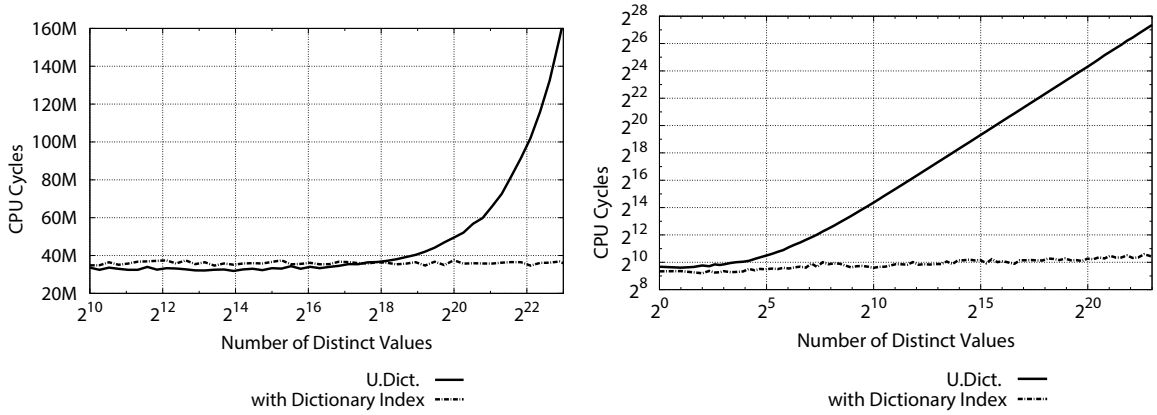
Column and dictionary indices are assumed to be implemented as B$^+$-Tree structures. A B$^+$-Tree is a tree structure supporting insertions, deletions and searches in logarithmic time and are optimized for systems reading blocks of data as it is the case for disk based systems but also when accessing main memory and reading cache lines. In contrast to B-Trees, the internal nodes of B$^+$-Trees store copies of the keys and the information is stored exclusively in the leaves, including a pointer to the next leaf node for sequential access. We denote the fan out of a tree index structure with I_f and the number of nodes needed to store $\boldsymbol{c.d}$ keys with I_n. The fan out constrains the number n of child nodes of all internal nodes to $I_f/2 \leq n \leq I_f$. I_{B_i} denotes the numbers of cache lines covered per node at cache level i. The number of matching keys for a scan with a range selection is denoted by $q.n_k$ and $q.n_v$ denotes the average number of occurrences of a key in the column.

6.1 Dictionary Index

A dictionary index is defined as a B$^+$-Tree structure [13] on top of an unsorted dictionary. Figure 6.1 shows an example of an uncompressed column, which is encoded with an unsorted dictionary. The compressed column and the unsorted dictionary are stored as described in Section 3.2. Additionally, a B$^+$-Tree with a branching factor $I_f = 2$ is maintained as a dictionary index.

We now discuss the influence of a dictionary index on our examined operations and compare a column using an unsorted dictionary with and without a dictionary index.

Scan with Equality Selection The algorithm on a column with a dictionary index is similar to Algorithm 3.4.3, except that for retrieving the value-id we leverage the dictionary index and perform a binary search. The number of cache misses regarding the sequential scanning of the value-ids of the column stays the same as in Equation 5.3.1 and the costs for the binary search



(a) Scan with equality selection

(b) Insert

Figure 6.2: Influence of a dictionary index on CPU cycles for (a) scan with an equality selection and (b) inserting new values into a column with an unsorted dictionary while varying the number of distinct values. $\boldsymbol{c.r} = 10\text{M}$, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.u} = 0$, $\boldsymbol{c.k} = 0.5$, $q.s = 1000$.

logarithmically depend on the number of distinct values of the column.

Figure 6.2(a) shows how the costs for a scan with equality selection develop with an increasing number of distinct values for a column using an unsorted dictionary without a dictionary index compared to a column with a dictionary index. We notice similar costs for the scan operation on columns with few distinct values. However, as the dictionary grows, the costs for linearly scanning the dictionary increase linearly in case of not using a dictionary index and the costs with an index only increase slightly due to the logarithmic cost for the binary search, resulting in better performance when using a dictionary index.

Scan with Range Selection Although the dictionary index maintains a sort order over the dictionary, the value-ids of two values still allow no conclusions about which value is larger or smaller as in the case of a sorted dictionary. Therefore, a scan with a range selection still needs to lookup and compare the actual values as described in Algorithm 3.4.6.

Insert a Record One main cost factor for inserting new values into a column with an unsorted dictionary is the linear search determining if the value is already in the dictionary. This can be accelerated through the dictionary index, although it comes with the costs of maintaining the tree structure as outlined in Algorithm 6.1.1. Assuming the new value is not already in the dictionary, the costs for inserting it are writing the new value in the dictionary, writing the compressed value-id, performing the binary search in the index and adding the new value to the index.

Lookup a Record Looking up a record is not affected by a dictionary index as the index can not be leveraged performing the lookup and does not have to be maintained. Therefore

Algorithm 6.1.1: InsertDictIndex($column, dictionary$)

$valueId \leftarrow dictionary.index.binarySearch(value)$
if $valueId = NotInDictionary$
then $\begin{cases} valueId \leftarrow dictionary.insert(value) \\ dictionary.index.insert(value, valueId) \end{cases}$
$column.append(valueId)$

the Algorithm as outlined in 3.4.8 is also applicable when using a dictionary index.

Conclusions

Considering the discussed operations, a dictionary encoded column always profits by using a dictionary index. Therefore, we do not provide adapted cost functions for a dictionary index as we do not have to calculate in which cases it is advisable to use. Even insert operations do profit from the index as the dictionary can be searched with logarithmic costs which outweighs the additional costs of index maintenance, as shown by Figure 6.2(b). The costs for index maintenance overshadow the saved costs only in cases with very few distinct values, where scanning the dictionary is similarly fast as a binary search. However, the additional costs in these cases are very small and a dictionary index for unsorted dictionaries is still advisable.

6.2 Column Index

A column index can be any auxiliary data structure accelerating the search of tuples given a value or range of values for an attribute. The most popular methods are hash-based indexing and tree-based indexing as described in [61].

Hash-based indexing groups all values in a column into *buckets*, based on a hash function determining which value belongs into which bucket. A bucket consists of a linked chain of values and can hold a variable number of values. When searching for a specific value, calculating the value of the hash function and accessing the resulting bucket can be achieved in constant time, assuming a good distribution over all buckets. However, support for selections with range queries is not given using hash-based indexing.

Tree-based indexing allows for fast selections using equality predicates and also supports efficient range queries. Therefore, we assume a column index to be a B^+-Tree structure, similar to the dictionary index described above. However, the index is built on top of the complete column and not only on the distinct values. Therefore, the index does not only store one

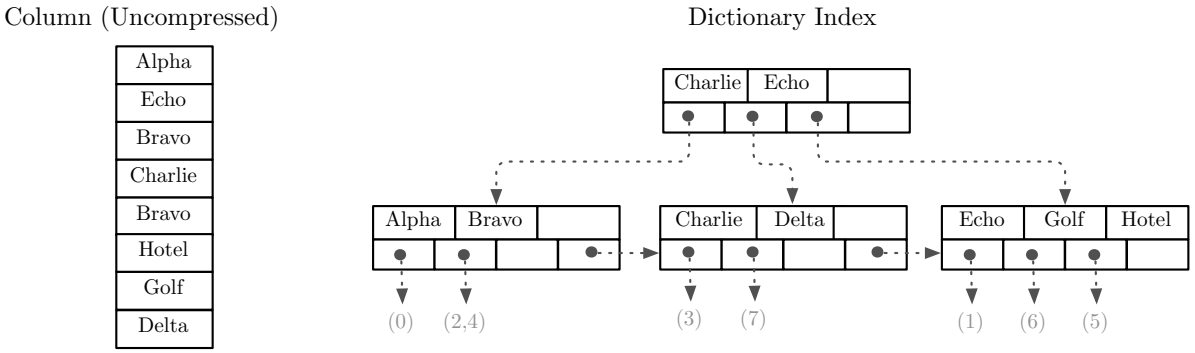


Figure 6.3: Example column index.

position, but has to store a list of positions for every value. A column index can be added to any column, regardless of the physical organization of the column. Figure 6.3 shows an example of a column index on an uncompressed column.

Lookup Performing a positional lookup on a column can not profit from a column index and also does not require any maintenance operations for the index. Therefore the Algorithm as outlined in 3.4.8 is still applicable.

Search with Equality Selection A search with equality selection can be answered entirely by using the column index. Therefore, the costs do not depend on the physical layout of the column and the same algorithm can be used for all column organizations, as outlined by Algorithm 6.2.1. First, the index is searched for value X by binary searching the tree structure resulting in a list of positions. If the value is not found, an empty list is returned. The resulting list of positions then has to be converted into the output format by adding all positions to the result array. Locating the leaf node for the searched key requires reading $\log_{I_f}(I_n) \cdot I_{B_i}$ cache lines for reading every node from the root node to the searched leaf node, assuming each accessed node lies on a separate cache line. Then, iterating through the list of positions and adding every position to the result array requires to read and write $q.n_v/B_i$ cachelines, assuming the positions are placed sequentially in memory.

$$\mathbf{M_i}(escan_I) = \log_{I_f}(I_n) \cdot I_{B_i} + 2 \cdot q.n_k \cdot \left\lceil \frac{q.n_v}{B_i} \right\rceil \tag{6.1}$$

Search with Range Selection Similarly to the search with equality selection, a search with range selection can be answered entirely by using the column index, as outlined by Algorithm 6.2.2. Assuming the range selection matches any values, we locate the node with the first matching value by performing a binary search on the column index. The number of cache misses for the binary search are $\log_{I_f}(I_n) \cdot I_{B_i}$. Then, we sequentially retrieve the next nodes by following the next pointer of each node until we find a node with a key greater or equal to *high*. Assuming completely filled nodes, this requires reading all nodes containing the $q.n_k$

Algorithm 6.2.1: SEARCHEQUALINDEX($X, columnIndex$)

$result \leftarrow array[]; node \leftarrow columnIndex.binarySearch(X)$
if $node$ **not** $NotFound$
then { **for each** $pos \in node.positionList$ **do** $\{result.append(pos)$
$return(result)$

Algorithm 6.2.2: SEARCHRANGEINDEX($X, Y, columnIndex$)

$result \leftarrow array[]$
$node \leftarrow columnIndex.binarySearch(X)$
if $node$ **not** $NotFound$
then { **while** $node.key < Y$ **do** { **for each** $pos \in node.positionList$ **do** $\{result.append(pos)$ $node \leftarrow node.next$
$return(result)$

matching keys, resulting in $q.n_k/I_f$ nodes. For all matching nodes the positions are added to the result array, requiring to read and write $q.n_v/B_i$ cache lines per key.

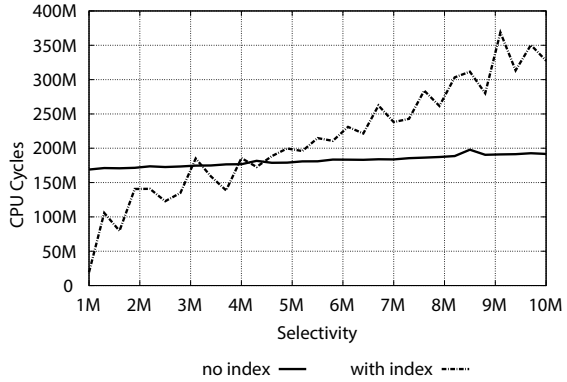
$$\mathbf{M_i}(rscan_I) = \log_{I_f}(I_n) \cdot I_{B_i} + I_{B_i} \cdot \frac{q.n_k}{I_f} + 2q.n_k \cdot \left\lceil \frac{q.n_v}{B_i} \right\rceil \tag{6.2}$$

Insert Inserting new values into the physical organization of a column is not affected by a column index. However, the new value has also to be inserted into the column index. The costs incurring for the index maintenance are independent from the physical organization of the column. This requires searching the tree structure for the inserted value, reading $\log_{I_f}(I_n) \cdot I_{B_i}$ cache lines. If the value already exists, the newly inserted position is added to the list of positions of the respective node, otherwise the value is inserted and the tree has to be potentially rebalanced. The costs for rebalancing are in average $\log_{I_f}(I_n) \cdot I_{B_i}$.

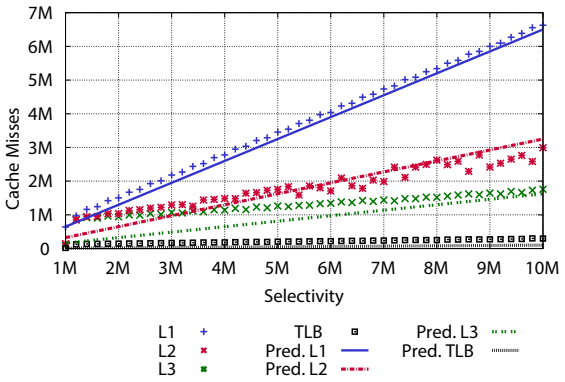
$$\mathbf{M_i}(insert_I) = 2 \cdot \log_{I_f}(I_n) \cdot I_{B_i} \tag{6.3}$$

Evaluation

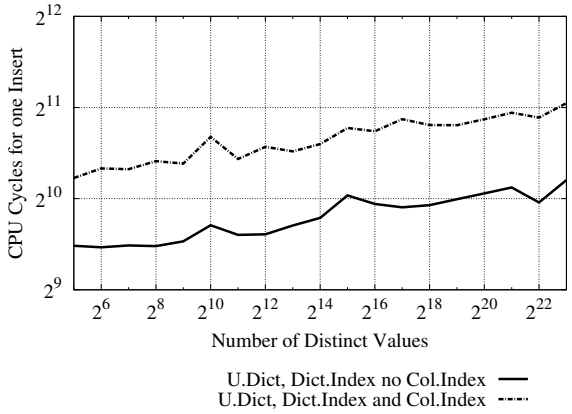
Figure 6.4(a) shows a comparison for a range scan on a column index compared to a column with a sorted dictionary and without an index. The figure shows the resulting CPU cycles for



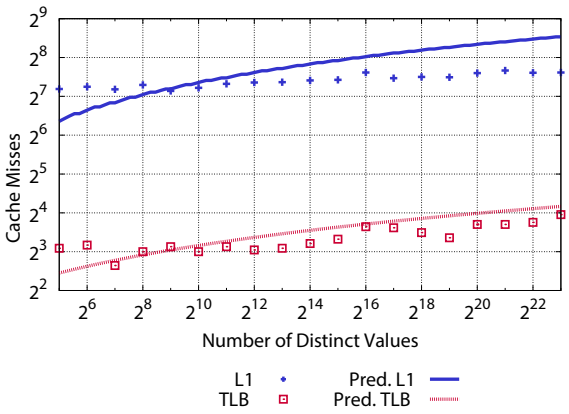
(a) Range scan performance with and without column index



(b) Cache misses for range scan with column index



(c) Insert performance with and without column index



(d) L1 and TLB misses for insert with column index

Figure 6.4: CPU cycles for (a) scan with a range selection and (c) inserting new values into a column with and without a column index. (b) and (d) show the respective cache misses for the case of using a column index. $\boldsymbol{c.r} = 10M$, $\boldsymbol{c.d}$ =1M, $\boldsymbol{c.e} = 8$, $\boldsymbol{c.u} = 0$, $\boldsymbol{c.k} = 0$

the scan operation with increasing result sizes. For small results the index performs better, but around a selectivity of roughly 4 million the complete scan performs better due to its sequential access pattern. Figure 6.4(b) shows the resulting cache misses for the scan operation using the column index and the predictions based on the defined model.

Figure 6.4(c) shows the costs for inserting a new value into a column using dictionary encoding with an unsorted dictionary plus dictionary index and no column index compared to a column with a column index. As expected, the cost for inserting in a column with a column index are approximately twice as high. Figure 6.4(d) shows the resulting L1 and TLB cache misses for inserting into a column with a column index and the respective predictions.

Chapter 7

Partitioned Columns

This chapter introduces a partitioned column organization, consisting of a main and a delta partition and outlines the merge process. We present an efficient merge algorithm, enabling dictionary encoded in-memory column stores to support the update performance required to run enterprise application workloads on read-optimized databases. The chapter is based on the work presented in [46].

Traditional read-optimized databases often use a compressed column oriented approach to store data [66, 50, 75]. Performing single inserts in such a compressed persistence can be as complex as inserting into a sorted list [28]. One approach to handle updates in a compressed storage is a technique called differential updates, maintaining a write-optimized delta partition that accumulates all data changes. Periodically, this delta partition is combined with the read-optimized main partition. We refer to this process as *merge* throughout the report, also referred to as checkpointing by others [28]. This process involves uncompressing the compressed main partition, merging the delta and main partitions and recompressing the resulting main partition. In contrast to existing approaches, the complete process is required to be executed during regular system load without downtime.

The update performance of such a system is limited by two factors – (a) the insert rate for the write-optimized structure and (b) the speed with which the system can merge the accumulated updates back into the read-optimized partition. Inserting into the write-optimized structure can be performed fast if the size of the structure is kept small enough. As an additional benefit, this also ensures that the read performance does not degrade significantly. However, keeping this size small implies merging frequently, which increases the overhead of updates.

7.1 Merge Process

For supporting efficient updates, we want the supported update rate to be as high as possible. Furthermore, the update rate should also be greater than the minimum sustained update rate required by the specific application scenario. When the number of updates against the system durably exceed the supported update rate, the system will be rendered incapable of processing new inserts and other queries, leading to failure of the database system.

An important performance parameter for a partitioned column is the frequency at which the merging of the partitions must be executed. The frequency of executing the merging of partitions affects the size (number of tuples) of the delta partition. Computing the appropriate size of the delta partition before executing the merge operation is dictated by the following two conflicting choices:

1. **Small delta partition** A small delta partition implies a relatively low overhead to the read query, implying a small reduction in the read performance. Furthermore, the insertion into the delta partition will also be fast. This means however that the merging step needs to be executed more frequently, thereby increasing the impact on the system.
2. **Large delta partition** A large delta partition implies that the merging is executed less frequently and therefore adds only a little overhead to the system. However, increasing the delta partition size implies a slower read performance due to the fact that the delta partition stores uncompressed values, which consume more compute resources and memory bandwidth, thereby appreciably slowing down read queries (scan, index lookup, etc.) Also, while comparing values in the main partition with those in delta partition, we need to look up the dictionary for the main partition to obtain the uncompressed value (forced materialization), thereby adding overhead to the read performance.

In our system, we trigger the merging of partitions when the number of tuples $\mathbf{N}_D$ in the delta partition is greater than a certain pre-defined fraction of tuples in the main partition $\mathbf{N}_M$.

Figure 7.1 shows an example of a column with its main and delta partitions. Note that the other columns of the table would be stored in a similar fashion. The main partition has a dictionary consisting of its sorted unique values (6 in total). Hence, the encoded values are stored using $3 = \lceil \log_2 6 \rceil$ bits. The uncompressed values (in gray) are not actually stored, but shown for illustration purpose. The compressed value for a given value is its position in the dictionary which is also called value-id, stored using the appropriate number of bits (in this case 3 bits).

The delta partition stores the uncompressed values themselves. In this example, there are five tuples with the shown uncompressed values. In addition, a Cache Sensitive B^+-Tree (CSB-Tree) [63] containing all the unique uncompressed values is maintained. Each value in the tree also stores a pointer to the list of tuple ids where the value was inserted. For example, the

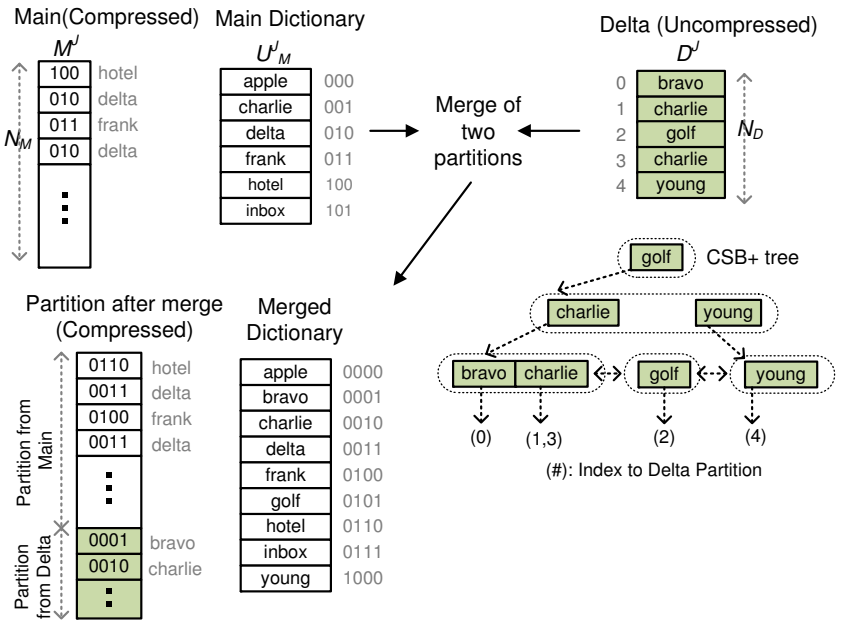


Figure 7.1: Example showing the data structures maintained for each column. The main partition is stored compressed together with the dictionary. The delta partition is stored uncompressed, along with the CSB+ tree. After the merging of the partitions, we obtain the concatenated compressed main column and the updated dictionary.

value "charlie" is inserted at positions 1 and 3. Upon insertion, the value is appended to the delta partition and the CSB+ tree is updated accordingly.

After the merging of the partitions has been performed, the main and delta partitions are concatenated and a new dictionary for the concatenated partition is created. Furthermore, the compressed values for each tuple are also updated. For example, the encoded value for"hotel" was 4 before merging and is 6 after merging. Furthermore, it is possible that the number of bits that are required to store the compressed value will increase after merging. Since the number of unique values in this example is increased to 9 after merging, each compressed value is now stored using $\lceil \log 9 \rceil = 4$ bits.

7.2 Merging Algorithm

We now describe the merge algorithm in detail and enhance the naïve merge implementation by applying optimizations known from join processing. Furthermore, we will parallelize our implementation and make it architecture-aware to achieve the best possible throughput. For the remainder of the report, we refer to the symbols explained in Table 7.1.

We use a compression scheme wherein the unique values for each column are stored in a

Description	Unit	Symbol
Number of columns in the table	-	$\mathbf{N}_C$
Number of tuples in the main table	-	$\mathbf{N}_M$
Number of tuples in the delta table	-	$\mathbf{N}_D$
Number of tuples in the updated table	-	$\mathbf{N}_M{}'$
For a given column $j; j \in [1 \dots \mathbf{N}_C]$:		
Main partition of the j^{th} column	-	$\mathbf{M}^j$
Merged column	-	$\mathbf{M}'^j$
Sorted dictionary of the main partition	-	$\mathbf{U}^j_{\mathbf{M}}$
Sorted dictionary of the delta partition	-	$\mathbf{U}^j_{\mathbf{D}}$
Updated main dictionary	-	$\mathbf{U}'^j_{\mathbf{M}}$
Delta partition of the j^{th} column.	-	$\mathbf{D}^j$
Uncompressed Value-Length	bytes	$\mathbf{E}^j$
Compressed Value-Length	bits	$\mathbf{E}^j_{\mathbf{C}}$
Compressed Value-Length after merge	bits	$\mathbf{E}'^j_{\mathbf{C}}$
Fraction of unique values in delta	-	$\lambda^j_{\mathbf{D}}$
Fraction of unique values in main	-	$\lambda^j_{\mathbf{M}}$
Merge auxiliary structure for the main	-	$\mathbf{X}^j_{\mathbf{M}}$
Merge auxiliary structure for the delta	-	$\mathbf{X}^j_{\mathbf{D}}$
Memory Traffic	bytes	MT
Number of available parallel threads	-	$\mathbf{N}_T$

Table 7.1: Symbol Definition. Entities annotated with $'$ represent the merged (updated) entry.

separate dictionary structure consisting of the uncompressed values stored in a sorted order. Hence, $|\mathbf{U}^j_{\mathbf{M}}| = \lambda^j_{\mathbf{M}} \cdot \mathbf{N}_M$ with $|X|$ denoting the number of elements in the set X. By definition, $\lambda^j_{\mathbf{M}}, \lambda^j_{\mathbf{D}} \in [0 \dots 1]$. Furthermore, the compressed value stored for a given value is its index in the (sorted) dictionary structure, thereby requiring $\lceil \log |\mathbf{U}^j_{\mathbf{M}}| \rceil$ bits [1] to store it. Hence, $\mathbf{E}^j_C = \lceil \log |\mathbf{U}^j_{\mathbf{M}}| \rceil$.

Input(s) and Output(s) of the Algorithm:

For each column of the table, the merging algorithm combines the main and delta partitions of the column into a single (modified) main partition and creates a new empty delta partition. In addition, the dictionary $\mathbf{U}'^j$ maintained for each column of the main table is also updated to reflect the modified merged column. This also includes modifying the compressed values stored for the various tuples in the merged column.

For the j^{th} column, the input for the merging algorithm consists of $\mathbf{M}^j$, $\mathbf{D}^j$ and $\mathbf{U}^j_{\mathbf{M}}$, while

[1] Unless otherwise stated, log refers to logarithm with base 2 ($\log_2$).

the output consists of $\mathbf{M}'^j$ and $\mathbf{U}'^j_\mathbf{M}$. Furthermore, we define the cardinality $\mathbf{N}_M{}'$ of the output and the size of the merged dictionary $|\mathbf{U}'^j_\mathbf{M}|$ as shown in Equation 7.1 and 7.2.

$$\mathbf{N}_M{}' = \mathbf{N}_M + \mathbf{N}_D \tag{7.1}$$

$$|\mathbf{U}'^j_\mathbf{M}| = |\mathbf{D}^j \cup \mathbf{U}^j_\mathbf{M}| \tag{7.2}$$

We perform the merge using the following two steps:

1. **Merging Dictionaries:** This step consists of the following two sub-steps: a) Extracting the unique values from the delta partition $\mathbf{D}^j$ to form the corresponding sorted dictionary denoted as $\mathbf{U}^j_\mathbf{D}$. b) Merging the two sorted dictionaries $\mathbf{U}^j_\mathbf{M}$ and $\mathbf{U}^j_\mathbf{D}$, creating the sorted dictionary $\mathbf{U}'^j_\mathbf{M}$ without duplicate values.
2. **Updating Compressed Values:** This step consists of appending the delta partition $\mathbf{D}^j$ to the main partition $\mathbf{M}^j$ and updating the compressed values for the tuples, based on the new dictionary $\mathbf{U}'^j_\mathbf{M}$. Since $\mathbf{D}^j$ may have introduced new values, this step requires: a) Computing the new compressed value-length. b) Updating the compressed values for all tuples with the new compressed value, using the index of the corresponding uncompressed value in the new dictionary $\mathbf{U}'^j_\mathbf{M}$.

We now describe the above two steps in detail and also compute the order of complexity for each of the steps. As mentioned earlier, the merging algorithm is executed separately for each column of the table.

7.2.1 Merging Dictionaries

The basic outline of step one is similar to the algorithm of a sort-merge-join [54]. However, instead of producing pairs as an output of the equality comparison, the merge will only generate a list of unique values. The merging of the dictionaries is performed in two steps (a) and (b).

Step 1 (a) This step involves building the dictionary for the delta partition $\mathbf{D}^j$. Since we maintain a CSB+ tree to support efficient insertions into $\mathbf{D}^j$, extracting the unique values in a sorted order involves a linear traversal of the leaves of the underlying tree structure [63]. The output of Step 1(a) is a sorted dictionary for the delta partition $\mathbf{U}^j_\mathbf{D}$, with a run-time complexity of $\mathcal{O}(|\mathbf{U}^j_\mathbf{D}|)$

Step 1(b) This step involves a linear traversal of the two sorted dictionaries $\mathbf{U}^j_\mathbf{M}$ and $\mathbf{U}^j_\mathbf{D}$ to produce a sorted dictionary $\mathbf{U}'^j_\mathbf{M}$. In line with a usual merge operation, we maintain two pointers called $iterator_\mathcal{M}$ and $iterator_\mathcal{D}$, to point to the values being compared in the two dictionaries $\mathbf{U}^j_\mathbf{D}$ and $\mathbf{U}^j_\mathbf{M}$, respectively. Both are initialized to the start of their respective dictionaries. At each step, we compare the current values being pointed to and append the

smaller value to the output. Furthermore, the pointer with the smaller value is also incremented. This process is carried out until the end of one of the dictionaries is reached, after which the remaining dictionary values from the other dictionary are appended to the output dictionary. In case both values are identical the value is appended to the dictionary once and the pointers for the dictionaries are incremented. The output of Step 1(b) is a sorted dictionary $\mathbf{U}_\mathbf{M}^{\prime j}$ for the merged column, with $|\mathbf{U}_\mathbf{M}^{\prime j}|$ denoting its cardinality. The run-time complexity of this step is $\mathcal{O}(|\mathbf{U}_\mathbf{M}^{j}| + |\mathbf{U}_\mathbf{D}^{j}|)$.

7.2.2 Updating Compressed Values

The compressed values are updated in two steps – (a) computing the new compressed value length and (b) writing the new main partition and updating the compressed values.

Step 2(a) The new compressed value-length is computed as shown in Equation 7.3. Note that the length for storing the compressed values in the column may has increased from the one used for storing the compressed values before the merging algorithm. Since we use the same length for all the compressed values, this step executes in $\mathcal{O}(1)$ time.

$$\mathbf{E}_C^{\prime j} = \lceil \log(|\mathbf{U}_\mathbf{M}^{\prime j}|) \rceil \text{ bits} \tag{7.3}$$

Step 2(b) We need to append the delta partition to the main partition and update the compressed values. As far as the main partition $\mathbf{M}^j$ is concerned, we use the following methodology. We iterate over the compressed values and for a given compressed value K_C^i, we compute the corresponding uncompressed value K^i by performing a lookup in the dictionary $\mathbf{U}_\mathbf{M}^j$. We then search for K^i in the updated dictionary $\mathbf{U}_\mathbf{M}^{\prime j}$ and store the resultant index as the encoded value using the appropriate number of $\mathbf{E}_C^j$ bits in the output. For the delta partition $\mathbf{D}^j$, we already store the uncompressed value, hence it requires a search in the updated dictionary to compute the index, which is then stored.

Since the dictionary is sorted on the values, we use a binary search algorithm to search for a given uncompressed value. The resultant run-time of the algorithm is

$$\mathcal{O}(\mathbf{N}_M + (\mathbf{N}_M + \mathbf{N}_D) \cdot \log(|\mathbf{U}_\mathbf{M}^{\prime j}|)). \tag{7.4}$$

To summarize the above, the total run-time for the merging algorithm is dominated by Step 2(b) and depends heavily on the search run-time. As shown in Section 7.4, this makes the merging algorithm prohibitively slow and infeasible for current configurations of tables. We now present an efficient variant of Step 2(b), which performs the search in linear time at the expense of using an auxiliary data structure. Since merging is performed on every column separately, we expect the overhead of storing the auxiliary structure to be very small, as compared to the total storage, and independent of the number of columns in a table and the number of tables residing in the main memory.

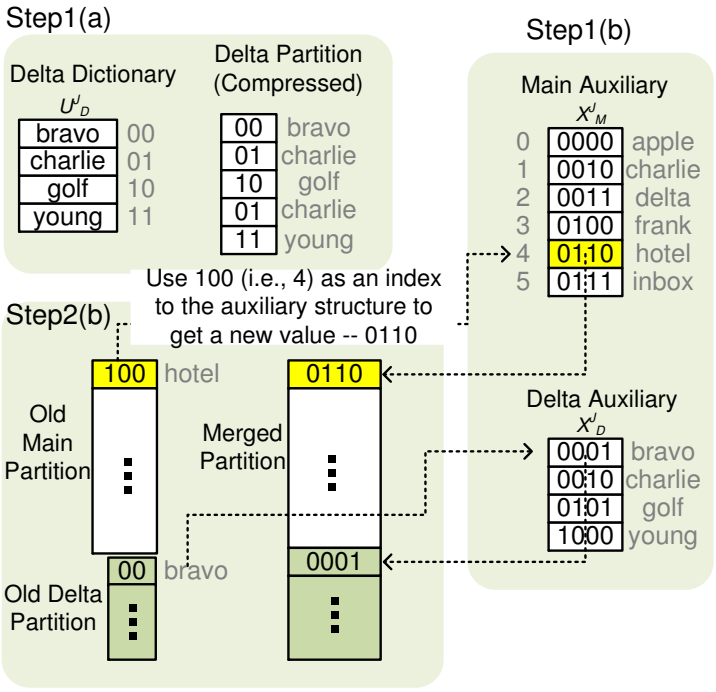


Figure 7.2: Example showing the various steps executed by our linear-time merging algorithm. The values in the column are similar to those used in Figure 7.1.

7.2.3 Initial Performance Improvements

Based on the previously described naïve algorithm, we significantly increase the merge performance by adding an additional auxiliary data structure per main and per delta partition, denoted as $\mathbf{X}^j_\mathbf{M}$ and $\mathbf{X}^j_\mathbf{D}$ respectively. The reasoning for the auxiliary data structure is to provide a translation table with constant access cost during the expensive Step 2(b). We now describe the modified Steps 1(a), 1(b) and 2(b) to linearize the update algorithm of compressed values and improve the overall performance.

Modified Step 1(a) In addition to computing the sorted dictionary for the delta partition, we also replace the uncompressed values in the delta partition with their respective indices in the dictionary. By this approach lookup indices for Step 2 are changed to fixed width and allow better utilization of cache lines and CPU architecture aware optimizations like SSE.

Since our CSB+ tree structure for the delta partition also maintains a list of tuple ids with each value, we access these values while performing the traversal of the tree leaves and replace them by their newly computed index into the sorted dictionary. Although this involves non-contiguous access of the delta partition, each tuple is only accessed once, hence the runtime is $\mathcal{O}(\mathbf{N}_D)$. For example, consider Figure 7.2, borrowing the main/delta partition values depicted in Figure 7.1. As shown in Step 1(a), we create the dictionary for the delta partition

(with 4 distinct values) and compute the compressed delta partition using 2 bits to store each compressed value.

Modified Step 1(b) In addition to appending the smaller value (of the two input dictionaries) to $\mathbf{U'^j_M}$, we also maintain the index to which the value is written. This index is used to incrementally map each value from $\mathbf{U^j_D}$ and $\mathbf{U^j_M}$ to $\mathbf{U'^j_M}$ in the selected mapping table $\mathbf{X^j_M}$ or $\mathbf{X^j_D}$. If both compared values are equal, the same index will be added to the two mapping tables.

At the end of Step 1(b), each entry in $\mathbf{X^j_M}$ corresponds to the location of the corresponding uncompressed value of $\mathbf{U^j_M}$ in the updated $\mathbf{U'^j_M}$. Similar observations hold true for $\mathbf{X^j_D}$ (w.r.t. $\mathbf{U^j_D}$). Since this modification is performed while building the new dictionary and both $\mathbf{X^j_M}$ and $\mathbf{X^j_D}$ are accessed in a sequential fashion while populating them, the overall run-time of Step 1(b) remains as noted in Equation 7.2 – linear in sum of number of entries in the two dictionaries. Step 1(b) in Figure 7.2 depicts the corresponding auxiliary structure for the example in Figure 7.1.

Modified Step 2(b) In contrast to the original implementation described earlier, computing the new compressed value for the main (or delta) table reduces to reading the old compressed value $\mathbf{K}^i_C$ and retrieving the value stored at $\mathbf{K}^i_C th$ index of the corresponding auxiliary structure $\mathbf{X^j_M}$ or $\mathbf{X^j_D}$. For example in Figure 7.2, the first compressed value in the main partition has a compressed value of 4 (100_2 in binary).

In order to compute the new compressed value, we look up the value stored at index 4 in the auxiliary structure that corresponds to 6 (1100_2 in binary). So value 6 is stored as the new compressed value, using 4 bits since the merged dictionary has 9 unique values, as shown in Figure 7.1. Therefore, a lookup and binary search in the original algorithm description is replaced by a lookup in the new algorithm, reducing the run-time to $\mathcal{O}(\mathbf{N}_M + \mathbf{N}_D)$.

To summarize, the modifications described above result in a merging algorithm with overall run-time of

$$\mathcal{O}(\mathbf{N}_M + \mathbf{N}_D + |\mathbf{U^j_M}| + |\mathbf{U^j_D}|) \tag{7.5}$$

which is linear in terms of the total number of tuples and a significant improvement compared to Equation 7.4.

7.3 Merge Implementation

In this section, we describe our optimized merge algorithm on modern CPUs in detail and provide an analytical model highlighting the corresponding compute and memory traffic requirements. We first describe the scalar single-threaded algorithm and later extend it to exploit the multiple cores present on current CPUs.

The model serves the following purposes: (1) Computing the efficiency of our implementation. (2) Analyzing the performance and projecting performance for varying input parameters and underlying architectural features like varying core and memory bandwidth.

7.3.1 Scalar Implementation

Based on the modified Step 1(a) (Section 7.2.3), extracting the unique values from $\mathbf{D}^j$ involves an in-order traversal of the underlying tree structure. We perform an efficient CSB+ tree traversal using the cache-friendly algorithm described by Rao et al. [63]. The number of elements in each node (of cache-line size) depends on the size of the uncompressed values $\boldsymbol{c.e}^j$ in the j^{th} column of the delta partition. For example, with $\boldsymbol{c.e}^j = 16$ bytes, each node consists of a maximum of 3 values. In addition to append the value to the dictionary during the in-order traversal, we also traverse the list of tuple-ids associated with that value and replace the tuples with the newly computed index into the sorted dictionary. Since the delta partition is not guaranteed to be cache-resident at the start of this step (irrespective of the tree sizes), the run-time of Step 1(a) depends on the available external memory bandwidth.

As far as the total amount of data read from the external memory is concerned, the total amount of memory required to store the tree is around 2X the total amount of memory consumed by the values themselves [63]. In addition to traversing the tree, writing the dictionary $\mathbf{U}^j_\mathbf{D}$ involves fetching the data for write. Therefore, the total amount of bandwidth required for the dictionary computation is around $4 \cdot \boldsymbol{c.e}$ bytes per value ($3 \cdot \boldsymbol{c.e}$ bytes read and $1 \cdot \boldsymbol{c.e}$ bytes written) for the column. Updating the tuples involves reading their tuple id and a random access into $\mathbf{D}^j$ to update the tuple. Since each access would read a cache-line (B_i bytes wide) the total amount of bandwidth required would be $(2 \cdot B_i + 4)$ bytes per tuple (including the read for the write component). This results in the total required memory traffic for this operation as shown by Equation 7.7. Note that at the end of Step 1(a), $\mathbf{D}^j$ also consists of compressed values (based on its own dictionary $\mathbf{U}^j_\mathbf{D}$).

Applying the modified Step 1(b) (as described in Section 7.2.3), the algorithm iterates over the two dictionaries and produces the output dictionary with the auxiliary structures. As far as the number of operations is concerned, each element appended to the output dictionary involves around 12 ops[2] [11]. As far as the required memory traffic is concerned, both $\mathbf{U}^j_\mathbf{M}$ and $\mathbf{U}^j_\mathbf{D}$ are read sequentially, while $\mathbf{U}'^j_\mathbf{M}$, $\mathbf{X}^j_\mathbf{M}$ and $\mathbf{X}^j_\mathbf{D}$ are written in a sequential order. Note that the compressed value-length used for each entry in the auxiliary structures is

$$\boldsymbol{e}'_c = \lceil \log(|\mathbf{U}'^j_\mathbf{M}|) \rceil. \tag{7.6}$$

Hence, the total amount of required read memory traffic can be calculated as shown in Equation 7.8. The required write memory traffic for building the new dictionary and generate the translation table is calculated as shown in Equation 7.9.

[2] 1 op implies 1 operation or 1 executed instruction.

$$MT = 4 \cdot \boldsymbol{c.e} \cdot |\mathbf{U}^j_{\mathbf{D}}| + (2 \cdot B_i + 4) \cdot \mathbf{N}_D \tag{7.7}$$

$$MT = \boldsymbol{c.e} \cdot (|\mathbf{U}^j_{\mathbf{M}}| + |\mathbf{U}^j_{\mathbf{D}}| + |\mathbf{U}'^j_{\mathbf{M}}|) + \frac{\boldsymbol{e}'_c \cdot (|\mathbf{X}^j_{\mathbf{M}}| + |\mathbf{X}^j_{\mathbf{D}}|)}{8} \tag{7.8}$$

$$MT = \boldsymbol{c.e} \cdot |\mathbf{U}'^j_{\mathbf{M}}| + \frac{\boldsymbol{e}'_c \cdot (|\mathbf{X}^j_{\mathbf{M}}| + |\mathbf{X}^j_{\mathbf{D}}|)}{8} \tag{7.9}$$

As shown for the modified Step 2(b) in Section 7.2.3, the algorithm iterates over the compressed values in $\mathbf{M}^j$ and $\mathbf{D}^j$ to produce the compressed values in the output table $\mathbf{M}'^j$. For each compressed input value, the new compressed value is computed using a lookup into the auxiliary structure $\mathbf{X}^j_{\mathbf{M}}$ for the main partition or $\mathbf{X}^j_{\mathbf{D}}$ for the delta partition, with an offset equal to the stored compressed value itself.

The function shown in Equation 7.10 is executed for each element of the main partition and similar for the delta partition.

$$\mathbf{M}'^j[i] \leftarrow \mathbf{X}^j_{\mathbf{M}}[\mathbf{M}[i]] \tag{7.10}$$

As far as the memory access pattern is concerned, updating each successive element in the main or delta partition may access a random location in the auxiliary data structure (depending on the stored compressed value). Since there may not exist any coherence in the values stored in consecutive locations, each access can potentially access a different cache line (size B_i bytes). For scenarios where the complete auxiliary structure cannot fit in the on-die caches, the amount of read memory traffic to access the auxiliary data structure is approximated by Equation 7.11.

$$MT = B_i \cdot (\mathbf{N}_M + \mathbf{N}_D) \tag{7.11}$$

In addition, reading the main/delta partition requires a read memory traffic as shown in Equation 7.12, while writing out the concatenated output column requires a total memory traffic that can be calculated as in Equation 7.13.

$$MT = \boldsymbol{c.e}_C \cdot (\mathbf{N}_M + \mathbf{N}_D)/8 \tag{7.12}$$

$$MT = 2\boldsymbol{e}'_c (\mathbf{N}_M + \mathbf{N}_D)/8 \tag{7.13}$$

In case $\mathbf{X}^j_{\mathbf{M}}$ (or $\mathbf{X}^j_{\mathbf{D}}$) fits in the on-die caches, their access will be bound by the computation rate of the processor, and only the main/delta partitions will be streamed in and the concatenated table is written (streamed) out. As far as the relative time spent in each of these steps is concerned, Step 1 takes about 33 % of the total merge time (with $\boldsymbol{c.e} = 8$ bytes and 50 % unique values) and Step 2 takes the remainder. [3] In terms of evidence of compute and

[3] Section 7.4 gives more details.

bandwidth bound, our analytical model defines upper bounds on the performance, if the implementation was indeed bandwidth bound (and a different bound if compute bound). Our experimental evaluations show that our performance closely matches the lower of these upper bounds for compute and bandwidth resources – which proves that our performance is bound by the resources as predicted by the model.

7.3.2 Exploiting Thread-level Parallelism

We now present the algorithms for exploiting the multiple cores/threads available on modern CPUs. $\mathbf{N}_T$ denotes the number of available processing threads.

Parallelization of Step 1

Recalling Step 1(a), we perform an in-order traversal of the CSB+ tree and simultaneously update the tuples in the delta partition with the newly assigned compressed values.

There exist two different strategies for parallelization:

1. Dividing the columns within a table amongst the available threads: Since the time spent on each column varies based on the number of unique values, dividing the columns evenly amongst threads may lead to load imbalance between threads. Therefore, we use a task queue based parallelization scheme [2] and enqueue each column as a separate task. If the number of tasks is much larger than the number of threads (as in our case with only a few tens to hundred columns and a few threads), the task queue mechanism of migrating tasks between threads works well in practice to achieve a good load balance.

2. Parallelizing the execution of Step 1(a) on each column amongst the available threads: Since a small portion of the run-time is spent in computing the dictionary, we execute it on a single-thread, and keep a cumulative count of the number of tuples that needs to be updated as we create the dictionary. We parallelize the next phase where these tuples are evenly divided amongst the threads and each thread scatters the compressed values to the delta partition.

For a table with very few columns, scheme (ii) performs better than scheme (i). We implemented both (i) and (ii) and since our input table consisted of few tens to hundreds of columns, we achieved similar scaling for both these schemes on current CPUs. In Section 7.4.2, we report the results for (i) – the results for (ii) would be similar. Step 1(b) involves merging the two sorted dictionaries $\mathbf{U}^j_\mathbf{M}$ and $\mathbf{U}^j_\mathbf{D}$ with duplicate removal and simultaneously populating the auxiliary structures $\mathbf{X}^j_\mathbf{M}$ and $\mathbf{X}^j_\mathbf{D}$. This is an inherent sequential dependency in this merge process and also requires the merge process to remove duplicates. Also note that for tables with large fractions of unique values or large value-lengths ($\geq$ 8 bytes) a significant portion

of the total run-time is spent in Step 1(b). Therefore, it is imperative to parallelize well, in order to achieve speedups in the overall run-time of the merging algorithm. We now describe our parallelization scheme in detail that in practice achieves a good load-balance. Let us first consider the problem of parallelizing the merging of $\mathbf{U}^j_\mathbf{M}$ and $\mathbf{U}^j_\mathbf{D}$ without duplicate removal. In order to evenly distribute the work among the $\mathbf{N}_T$ threads it is required to partition both dictionaries into $\mathbf{N}_T$-quantiles. Since both dictionaries are sorted this can be achieved in $\mathbf{N}_T \log(|\mathbf{U}^j_\mathbf{M}| + |\mathbf{U}^j_\mathbf{D}|)$ steps [19]. Furthermore, we can also compute the indices in the two dictionaries for the i^{th} thread $\forall i \in \mathbf{N}_T$ following the same algorithm as presented in [11]. Thus, each thread can compute its start and end indices in the two dictionaries and proceed with the merge operation. In order to handle duplicate removal while merging, we use the following technique consisting of three phases. We additionally maintain an array *counter* of size $(\mathbf{N}_T + 1)$ elements.

Phase 1 Each thread computes its start and end indices in the two dictionaries and writes the merged output, while locally removing duplicates. Since the two dictionaries consisted of unique elements to start with, the only case where this can create duplicate elements is when the last element produced by the previous thread matches the first element produced by the current thread. This case is checked for by comparing the start elements in the two dictionaries with the previous elements in the respectively other dictionary. In case there is a match, the corresponding pointer is incremented before starting the merge process. Once a thread (say the i^{th} thread) completes the merge execution, it stores the number of unique elements produced by that thread to the corresponding location in the counter array (i.e. $counter[i]$). There is an explicit global barrier at the end of phase 1.

Phase 2 In the second phase, we compute the prefix sum of the counter array, so that $counter[i]$ corresponds to the total number of unique values that would be produced by the previous i threads. Additionally, $counter[\mathbf{N}_T]$ is the total number of unique values that the merge operation would produce. We parallelize the prefix sum computation using the algorithm by Hillis et al. [30].

Phase 3 The counter array produced at the end of phase 2 also provides the starting index at which a thread should start writing the locally computed merged dictionary. Similar to phase 1, we recompute the start and end indices in the two dictionaries. Now consider the main partition. The range of indices computed by the thread for $\mathbf{U}^j_\mathbf{M}$ also corresponds to the range of indices for which the thread can populate $\mathbf{X}^j_\mathbf{M}$ with the new indices for those values. Similar observations hold for the delta partition. Each thread performs the merge operation within the computed range of indices to produce the final merged dictionary and auxiliary data structures.

Summary In comparison to the single-threaded implementation, the parallel implementation reads the dictionaries twice and also writes the output dictionary one additional time, thereby increasing the total memory traffic by

$$\boldsymbol{c.e} \cdot \left(|\mathbf{U}^j_\mathbf{M}| + |\mathbf{U}^j_\mathbf{D}|\right) + 2\boldsymbol{c.e} \cdot |\mathbf{U}'^j_\mathbf{M}| \tag{7.14}$$

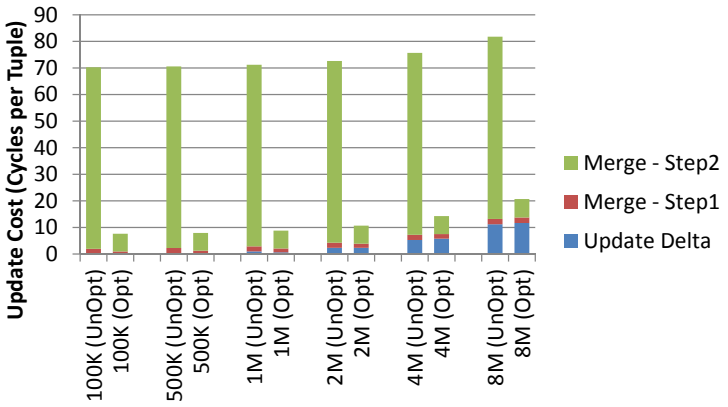


Figure 7.3: Update Costs for Various Delta Partition Sizes with a main partition size of 100 million tuples with 10 % unique values using 8-byte values. Both optimized (Opt) and unoptimized (UnOpt) merge implementations were parallelized.

bytes. The overhead of the start and end index computation is also very small as compared to the total computation performed by Step 1(b). The resultant parallel algorithm evenly distributes the total amount of data read/written to each thread, thereby completely exploiting the available memory bandwidth.

Parallelization of Step 2

To parallelize the updating of compressed values, we evenly divide the total number of all tuples $\mathbf{N}'_M$ amongst the available threads. Specifically, each thread is assigned $\mathbf{N}'_M/\mathbf{N}_T$ tuples to operate upon. Since each thread reads/writes from/to independent chunks of tables, this parallelization approach works well in practice. Note that in case any of $\mathbf{X^j_M}$ or $\mathbf{X^j_D}$ can completely fit in the on-die caches, this parallelization scheme still exploits to read the new index for each tuple from the caches and that the run-time is proportional to the amount of bandwidth required to stream the input and output tables.

7.4 Performance Evaluation

We now evaluate the performance of our algorithm on a dual-socket six-core Intel Xeon X5680 CPU with 2-way SMT per core, and each core operating at a frequency of 3.3 GHz. Each socket has 32 GB of DDR (for a total of 64 GB of main memory). The peak external memory bandwidth on each socket is 30 GB/sec. We used SUSE SLES 11 as operating system, the pthread library and the Intel ICC 11.1 as compiler. As far as the input data is concerned, the number of columns in the partition $\mathbf{N}_C$ varies from 20 to 300. The value-length of the uncompressed value $\boldsymbol{c.e}$ for a column is fixed and chosen from 4 bytes to 16 bytes. The number

of tuples in the main partition $\mathbf{N}_M$ varies from 1 million to 1 billion, while the number of tuples in the delta partition $\mathbf{N}_D$ varies from 500,000 to 50 million, with a maximum of around 5 % of $\mathbf{N}_M$. Since the focus of the report is on in-memory databases, the number of columns is chosen so that the overall data completely fits in the available main memory of the CPU. The fraction of unique values $\lambda^j_{\mathbf{M}}$ and $\lambda^j_{\mathbf{D}}$ varies from 0.1 % to 100 % to cover the spectrum of scenarios in real applications. For all experiments, the values are generated uniformly at random. We chose uniform value distributions, as this represents the worst possible cache utilization for the values and auxiliary structures. Different value distributions can only improve cache utilization, leading to better merge times. However, differences in merge times due to different value distributions are expected to be very small and are therefore negligible. We first show the impact of varying $\mathbf{N}_D$ on the merge performance. We then vary $\boldsymbol{c.e}$ from 4–16 bytes to analyze the effect of varying value-lengths on merge operations. We finally vary the percentage of unique values ($\lambda^j_{\mathbf{M}}$, $\lambda^j_{\mathbf{D}}$) and the size of the main partition $\mathbf{N}_M$. In order to normalize performance w.r.t. varying input parameters, we introduce the term – *update cost*. Update Cost is defined as the amortized time taken per tuple per column (in cycles/tuple), where the total time is the sum of times taken to update the delta partition $\mathbf{T}_U$ and the time to perform the merging of main and delta partitions $\mathbf{T}_M$, while the total number of tuples is $\mathbf{N}_M + \mathbf{N}_D$.

7.4.1 Impact of Delta Partition Size

Figure 7.3 shows the update cost for varying tuples of the delta partition. In addition to the delta partition update cost, we also show both run-times for the unoptimized and optimized Steps 1 and 2 in the graph. $\mathbf{N}_M$ is fixed to be 100 million tuples, while $\mathbf{N}_D$ is varied from 500,000 (0.5 %) to 8 million (8 %) tuples. $\lambda^j_{\mathbf{M}}$ and $\lambda^j_{\mathbf{D}}$ are fixed to be around 10 %. The uncompressed value-length $\boldsymbol{c.e}$ is 8 bytes. We fix the number of columns $\mathbf{N}_C$ to 300. Note that the run-times are on a *parallelized code for both implementations* on our dual-socket multi-core system.

As far as the unoptimized merge algorithm is concerned, Step 2 (updating the compressed values) takes up the majority of the run-time and does not change (per tuple) with the varying number of tuples in the delta partition. The optimized Step 2 algorithm drastically reduces the time spent in the merge operation (by 9–10 times) as compared to the unoptimized algorithm. Considering the optimized code, as the delta partition size increases, the percentage of the total time spent on delta updates increases and is 30 % – 55 % of the total time. This signifies that the overhead of merging contributes a relatively small percentage to the run-time, thereby making our scheme of maintaining separate main and delta partitions with the optimized merge an attractive option for performing updates without a significant overhead.

The update rate in tuples/second is computed by dividing the total number of updates with the time taken to perform delta updates and merging the main and delta partitions for the $\mathbf{N}_C$ columns. As an example, for $\mathbf{N}_D = 4$ million and say $\mathbf{N}_C = 300$, an update cost of

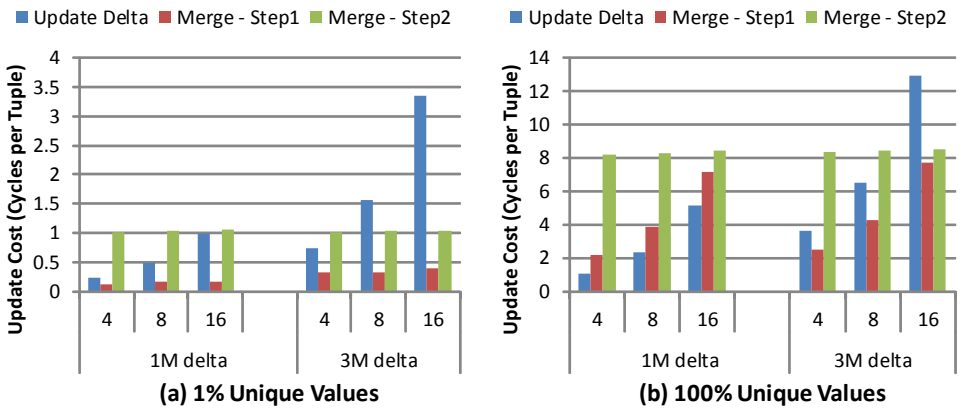


Figure 7.4: Update Costs for Various Value-Lengths for two delta sizes with 100 million tuples in the main partition for 1 % and 100 % unique values.

13.5 cycles per tuple (from Figure 7.3) evaluates to

$$\frac{4,000,000 \cdot 3.3 \cdot 10^9}{13.5 \cdot 104,000,000 \cdot 300} \approx 31,350 \text{ updates/second.} \tag{7.15}$$

7.4.2 Impact of Value-Length and Percentage of Unique values

Figure 7.4 shows the impact of varying uncompressed value-lengths on the update cost. The uncompressed value-lengths are varied between 4, 8 and 16 bytes. We show two graphs with the percentage of unique values fixed at (a) 1 % and (b) 100 % respectively. $\mathbf{N}_M$ is fixed to be 100 million tuples for this experiment and the breakdown of update cost for $\mathbf{N}_D$ equal to 1 million and 3 million tuples is shown. We fix the number of columns $\mathbf{N}_C$ to 300.

As the value-length increases, the time taken per tuple to update the delta partition increases and becomes a major contributor to the overall run-time. This time also increases as the size of the delta partition increases. For example, in Figure 7.4(a), for an uncompressed value-length of 16 bytes, the delta update time increases from about 1.0 cycles per tuple for $\mathbf{N}_D$ = 1 million to about 3.3 cycles for $\mathbf{N}_D$ = 3 million. This time increases as the percentage of unique values increases. The corresponding numbers in Figure 7.4(b) for 100 % unique values are 5.1 cycles for $\mathbf{N}_D$ = 1 million and 12.9 cycles for $\mathbf{N}_D$ = 3 million.

As far as the Step 2 of the merge is concerned, the run-time depends on the percentage of unique values. For 1 % unique values, the auxiliary structures fit in cache. As described in Section 7.3.2, the auxiliary structures being gathered fit in cache and the run-time is bound by the time required to read the input partitions and write the updated partitions. We get a run-time of 1.0 cycles per tuple (around 1.8 cycles per tuple on 1-socket), which is close to the bandwidth bound computed in Section 7.3.2. For 100 % unique values, the auxiliary structures do not fit in cache and must be gathered from memory. The time taken is then

around 8.3 cycles (15.0 cycles on 1-socket), which closely matches (within 10 %) the analytical model developed in Section 7.3. The time for Step 2 mainly depends on whether the auxiliary structures can fit in cache and therefore is constant with small increases in the delta size from 1 million to 3 million.

As far as Step 1 is concerned, for a given delta partition size $\mathbf{N}_D$, the time spent in Step 1 increases sub-linearly with the increase in value-length (Section 7.3.1). For a fixed value-length, the time spent increases marginally with the increase in $\mathbf{N}_D$ – due to the fact that this increase in partition size only changes the unique values by a small amount and hence the compressed value-length also changes slightly, resulting in a small change in the run-time of Step 1. With larger changes in the percentage of unique values from 1 % to 100 %, the run-time increases. For instance, for 8-byte values and 1 million delta partitions, Step 1 time increases from 0.1 cycles per tuple at 1 % unique values to 3.3 cycles per tuple at 100 % unique values.

Finally, the percentage of time spent in updating the tuples as compared to the total time increases both with increasing value-lengths for fixed $\mathbf{N}_D$ and increase in $\mathbf{N}_D$ for fixed value-lengths.

7.5 Merge Strategies

An important performance parameter for a system with a merge process as described in the preceding sections, is the frequency at which the merging of the partitions must be executed. This frequency is given by the size of the delta partition, at which a merge process is initiated. The appropriate size is dictated by two conflicting choices: a) A small delta partition means fast insertions into the delta partition and only a small overhead for read queries, but implies a more frequently executed merge process with higher impact to the system. b) A large delta partition reduces the overhead of the merge process, but implies higher insertion costs in the delta partition and a higher overhead for read queries. In our system, we trigger the merging of partitions when the number of tuples $\mathbf{N}_D$ in the delta partition is greater than a certain pre-defined fraction of tuples in the main partition $\mathbf{N}_M$.

In contrast to the previously introduced partitioned column with one main and one delta partition, we extend the concept of a partitioned column and allow an arbitrary number of partitions. A partition can be either a write optimized store (WOS) or a read optimized store (ROS). A WOS is optimized for write access similar to the delta partition whereas a ROS is optimized for read queries similar to the main partition. In order to further balance this tradeoff between merge costs and query performance, merge strategies can be applied, similar to tradeoffs in the context of index structures between index maintenance costs and query performance [6]. A merge strategy defines which partitions are merged together when a merge process is executed. We now discuss the strategies *immediate merge*, *no merge* and *logarithmic merge*.

Immediate Merge Every write optimized store is merged immediately with the existing read optimized store and hence the strategy is called *Immediate Merge*. Note that this means that there is exactly one WOS and one ROS at all times and that the one ROS is constantly growing and potentially very large. This results in high costs for the merge process, because for every merge all tuples are touched. For the query processing this is an advantage, because only one read and one write optimized storage have to be considered instead of combining the results of multiple storages. The immediate merge strategy can be classified as an extreme strategy, with an unbalanced trade-off between query and merge costs.

No Merge The counterpart strategy of the immediate merge is called *No Merge*. In order to compact one WOS, the strategy always creates one new ROS out of the WOS. Therefore, only tuples from the small WOS have to be considered, which makes the merge very fast. However, the result is a growing collection of equally sized ROS, which all have to be queried in order to answer one single query. Although the queries against the storages can be parallelized, the conflation of the results is an additional overhead. The No Merge Strategy can be considered as an extreme strategy as well, shifting the costs from merge process to the query processing.

Logarithmic Merge The idea behind the logarithmic merge strategy is to find a balance between query and merge costs. This is accomplished by allowing multiple ROS to exists, but also merging several WOS into one ROS from time to time. In a b-way logarithmic merge strategy each storage is assigned a generation g. If a new ROS is created exclusively from one WOS, it is assigned $g = 0$. When b storages with $g = x$ exist, they are merged into one storage with $g = x+1$. The number of ROS is bounded by $O(\log n)$, where n is the total size of the collection.

With an Immediate Merge Strategy the storage maintenance costs grow linearly with every applied merge whereas the query costs stay low. When a No Merge Strategy is applied the contrary image appears where merge costs stay low and query costs grow linearly. The center part shows how query and merge costs grow with a 2-way Logarithmic Merge Strategy. The merge costs are varying depending on how many storages have to be touched. In every 2^x cases all storages have to be merged into one. The query costs depend on the number of existing storages and are therefore lowest when all storages have been merged into one.

In order to choose a merge strategy, the workload has to be analyzed and characterized. In a second step the resulting costs for each merge strategy can be calculated and the most promising strategy should be chosen, based on an analytical cost model predicting the cost differences for each merge strategy. Building on top of the presented cache-miss-based cost model, our goal is to roughly estimate the ratio between the costs for read queries and write queries. In order to do so, we calculate the number of select statements compared to insert statements and weigh them according to their complexity and estimated selectivity. This results in an abstract ratio q between read and write query costs for the workload. This number is used as an input parameter for the calculated cost functions for each merge strategy.

The actual total costs for a workload depend on several parameters. The costs are added

up from the costs for read queries, write queries and the performed merges. The costs for the read queries highly depend on the number of queries, their selectivity, their complexity and the size of the tables they are accessing. The costs for the write queries are given by the number of write queries and the number of distinct values in the write optimized storage. The merge costs depend on the number of touched tuples per write and the characteristics of the tables' distinct values (intersection, tail).

However, we are not interested in modeling the actual total costs for one merge strategy. Instead, we are interested in modeling only the costs that are influenced by the merge strategy in order to determine the merge strategy with the lowest total costs. The total costs for one merge strategy comprise the read, write and merge costs. The cost estimations are very similar to the index maintenance case presented in [6].

Chapter 8

Conclusions and Future Work

This last chapter presents the conclusion of this report. We summarize the content of this work and give an outlook of future work.

In this report, we presented a cost model for estimating cache misses for the plan operators scan with equality selection, scan with range selection, positional lookup and insert in a column-oriented in-memory database. We outlined functions estimating the cache misses for columns using different data structures and covered uncompressed columns, bit-packed and dictionary encoded columns with sorted and unsorted dictionaries. We presented detailed cost functions predicting cache misses and TLB misses.

Chapter 3 gave an overview of the discussed system and introduced the considered physical column organizations and plan operators. As expected, uncompressed columns are well suited if fast insertions are required and also support fast single lookups. However, scan performance is slow compared to dictionary encoded columns. Especially scan operators with equality selection profit from scanning only the compressed attribute vector.

Chapter 4 presented n evaluation of parameter influences on the plan operator performance with varying number of rows, number of distinct values, value disorder, value length and value skewness. For dictionary encoded columns using an unsorted dictionary, we identified – besides the number of rows and the value length – three additional parameters influencing the performance of scan operations. The number of distinct values has a strong impact on range and equal scans, and renders unsorted dictionaries unusable for columns with a large amount of distinct values and dictionaries larger than available cache sizes. However, if the disorder in the column is low, the penalties paid for range scans are manageable. Additionally, the skewness of values in a column can influence the performance of range scan operators, although the impact is small unless the distribution is extremely skewed. Regarding single lookup operations, the physical column organizations do not largely differ, except that the lookup in uncompressed columns is slightly faster.

In a nutshell, uncompressed columns seem to be well suited for classical OLTP workloads

with a high number of inserts and mainly single lookups. As the number of scan operations and especially range scans increases, the additional insert expenses pay off, rendering dictionary encoded column suitable for analytical workloads. Considering mixed workloads as in [46, 45], there is no optimal column organization as the result is depending on the concrete query distribution in the workload. Especially in the cases of mixed workloads, where the optimal column organization is unclear, our analytical cost model presented in Chapter 5 and extended for index structures in Chapter 6, allows to roughly estimate the costs and decide for a suited column organization.

Chapter 7 introduced partitioned columns and proposed an optimized online merge algorithm for dictionary encoded in-memory column stores, enabling them to support the update performance required to run enterprise application workloads on read-optimized databases. Additionally, we introduced a memory traffic based cost model for the merge process and proposed merge strategies to further balance the tradeoff between merge costs and query performance.

Possible directions of future work could be the extension of the model for non uniform memory access systems or to take more sophisticated algorithms for the discussed plan operations or additional operators into account.

For example, the range scan operation on an uncompressed dictionary can be implemented by first scanning the dictionary to build a bitmap index and then iterating over the value-ids performing lookups into the bitmap index instead of accessing the dictionary. Such an algorithm performs well in case the bitmap index fits into the cache. A similar direction of research would be to extend the model for various index structures.

References

[1] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. *IEEE 23rd International Conference on Data Engineering, 2007. ICDE 2007*, pages 466–475, 2007.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA*, pages 1–12, 2000.

[3] V. Babka and P. Tůma. Investigating Cache Parameters of x86 Family Processors. *Computer Performance Evaluation and Benchmarking*, pages 77–96, 2009.

[4] T. Barr, A. Cox, and S. Rixner. Translation Caching: Skip, Don't Walk (the Page Table). *ACM SIGARCH Computer Architecture News*, 38(3):48–59, 2010.

[5] J. Bernet. *Dictionary Compression for a Scan-Based, Main-Memory Database System.* PhD thesis, ETH Zurich, Apr. 2010.

[6] S. Blum. A generic merge-based dynamic indexing framework for imemex. Master's thesis, ETH Zurich, 2008.

[7] P. Boncz, S. Mangold, and M. L. Kersten. Database Architecture Optimized for the new Bottleneck: Memory Access. page 12, Nov. 1999.

[8] P. Boncz and M. Zukowski. MonetDB/X100: Hyper-pipelining query execution. *Proc. CIDR*, 2005.

[9] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51(12):77, Dec. 2008.

[10] P. A. Bonczk. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications.* PhD thesis, CWI Amsterdam, 2002.

[11] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.

[12] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 1970.

[13] D. Comer. Ubiquitous B-Tree. *CSUR*, 1979.

[14] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. *Journal of Instruction-Level Parallelism*, 10:1–33, 2008.

[15] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. *Proceedings of the 26th annual international symposium on ...*, 1999.

[16] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. High-performance DRAMs in workstation environments. *Computers, IEEE Transactions on*, 50(11):1133–1153, 2001.

[17] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems. *Conference on Linux Clusters: The HPC Revolution*, 2001.

[18] U. Drepper. What Every Programmer Should Know About Memory. *http://people. redhat. com/drepper/cpumemory. pdf*, 2007.

[19] R. S. Francis and I. D. Mathieson. A Benchmark Parallel Sort for Shared Memory Multiprocessors. *IEEE Trans. Computers*, 37(12):1619–1626, 1988.

[20] C. D. French. One Size Fits All Database Architectures Do Not Work for DSS. *SIGMOD*, 1995.

[21] G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 1994.

[22] G. Graefe. Sorting and indexing with partitioned B-trees. *Proc. of the 1st Int'l Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA*, 2003.

[23] G. Graefe and H. Kuno. Adaptive indexing for relational keys. *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 69–74, 2010.

[24] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. *Proceedings of the 13th International Conference on Extending Database Technology*, pages 371–381, 2010.

[25] M. Grund et al. HYRISE—A Main Memory Hybrid Storage Engine. *VLDB*, 2010.

[26] M. Grund et al. Optimal Query Operator Materialization Strategy for Hybrid Databases. *DBKDA*, 2010.

[27] B. He, Y. Li, Q. Luo, and D. Yang. EaseDB: a cache-oblivious in-memory query processor. *SIGMOD*, 2007.

[28] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. A. Boncz. Positional update handling in column stores. *SIGMOD*, 2010.

[29] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[30] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[31] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. *ACM/IEEE Supercomputing Conference*, 1997.

[32] F. Hübner, J. Böse, and J. Krüger. A cost-aware strategy for merging differential stores in column-oriented in-memory DBMS. *BIRTE*, 2011.

[33] S. Idreos, M. Kersten, and S. Manegold. Database cracking. *Proceedings of CIDR*, 2007.

[34] S. Idreos, M. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. *SIGMOD Conference*, pages 297–308, 2009.

[35] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *VLDB*, 2011.

[36] Intel Inc. *TLBs, Paging-Structure Caches, and Their Invalidation*, 2008.

[37] Intel Inc. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2011.

[38] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, and Y. Zhang. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment archive*, 1(2):1496–1499, 2008.

[39] B. Kao and H. Garcia-Molina. *An overview of real-time database systems*. Prentice-Hall, Inc, 1995.

[40] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. *ICDE*, 2011.

[41] A. Kemper, T. Neumann, F. Funke, V. Leis, and H. Muehe. HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *Bulletin of the Technical Committee on Data Engineering*, 2012.

[42] K. Kim, J. Shim, and I.-h. Lee. Cache conscious trees: how do they perform on contemporary commodity microprocessors? *ICCSA'07*, 2007.

[43] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional, 1973.

[44] J. Krueger, M. Grund, C. Tinnefeld, H. Plattner, A. Zeier, and F. Faerber. Optimizing Write Performance for Read Optimized Databases. *Database Systems for Advanced Applications*, pages 291–305, 2010.

[45] J. Krueger, C. Tinnefeld, M. Grund, A. Zeier, and H. Plattner. A case for online mixed workload processing. *DBTest*, 2010.

[46] J. Krüger, K. Changkyu, M. Grund, N. Satish, D. Schwalb, and C. Jatin. Fast Updates on Read Optimized Databases Using Multi Core CPUs. *VLDB*, 2011.

[47] I. Lee, S. Lee, and J. Shim. Making T-Trees Cache Conscious on Commodity Microprocessors. *Journal of information science and engineering*, 27:143–161, 2011.

[48] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. *VLDB*, 1968.

[49] S. Listgarten. Modelling Costs for a MM-DBMS. *RTDB*, 1996.

[50] R. MacNicol and B. French. Sybase IQ Multiplex — Designed For Analytics. *VLDB*, 2004.

[51] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. *VLDB*, 2002.

[52] S. Manegold and M. Kersten. Generic database cost models for hierarchical memory systems. *Proceedings of the 28th international conference on Very Large Data Bases*, pages 191–202, 2002.

[53] A. Mazreah, M. Sahebi, M. Manzuri, and S. Hosseini. A Novel Zero-Aware Four-Transistor SRAM Cell for High Density and Low Power Cache Application. In *Advanced Computer Theory and Engineering, 2008. ICACTE '08. International Conference on*, pages 571–575, 2008.

[54] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *CSUR*, 1992.

[55] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38:114 ff., 1965.

[56] J. Müller. *A Real-Time In-Memory Discovery Service*. PhD thesis, Hasso Plattner Institute, 2012.

[57] H. Pirk. Cache Conscious Data Layouting for In-Memory Databases. Master's thesis, Humboldt-Universität zu Berlin, 2010.

[58] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. *ACM Sigmod Records*, 2009.

[59] H. Plattner and A. Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. Springer, 2011.

[60] T. Puzak, A. Hartstein, P. Emma, V. Srinivasan, and A. Nadus. Analyzing the Cost of a Cache Miss Using Pipeline Spectroscopy. *Journal of Instruction-Level Parallelism*, 10:1–33, 2008.

[61] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 3rd edition, 1999.

[62] J. Rao and K. A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. *VLDB*, 1999.

[63] J. Rao and K. A. Ross. Making B^+-Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.

[64] R. Saavedra and A. Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Transactions on Computersn*, 1995.

[65] M. Sleiman, L. Lipsky, and K. Konwar. Performance Modeling of Hierarchical Memories. *CAINE*, 2006.

[66] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. *VLDB*, 2005.

[67] M. Stonebraker and U. Cetintemel. "One size fits all": an idea whose time has come and gone. *ICDE*, 2005.

[68] J. A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, 1988.

[69] C. T. Team. In-memory data management for consumer transactions the timesten approach. *SIGMOD*, 1999.

[70] K. Whang. Query optimization in a memory-resident domain relational calculus database system. *TODS*, 1990.

[71] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proceedings of the VLDB Endowment*, 2(1):385–394, 2009.

[72] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *TODS*, 2006.

[73] N. Zhang, A. Mukherjee, T. Tao, T. Bell, R. VijayaSatya, and D. Adjeroh. A Flexible Compressed Text Retrieval System Using a Modified LZW Algorithm. *Data Compression Conference*, 2005.

[74] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, CWI Amsterdam, 2009.

[75] M. Zukowski, P. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Engineering Bulletin*, 2005.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
66	978-3-86956-227-8	**Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software**	Thomas Vogel, Holger Giese
65	978-3-86956-226-1	**Scalable Compatibility for Embedded Real-Time components via Language Progressive Timed Automata**	Stefan Neumann, Holger Giese
64	978-3-86956-217-9	**Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants**	Basil Becker, Holger Giese
63	978-3-86956-204-9	**Theories and Intricacies of Information Security Problems**	Anne V. D. M. Kayem, Christoph Meinel (Eds.)
62	978-3-86956-212-4	**Covering or Complete? Discovering Conditional Inclusion Dependencies**	Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, Felix Naumann
61	978-3-86956-194-3	**Vierter Deutscher IPv6 Gipfel 2011**	Christoph Meinel, Harald Sack (Hrsg.)
60	978-3-86956-201-8	**Understanding Cryptic Schemata in Large Extract-Transform-Load Systems**	Alexander Albrecht, Felix Naumann
59	978-3-86956-193-6	**The JCop Language Specification**	Malte Appeltauer, Robert Hirschfeld
58	978-3-86956-192-9	**MDE Settings in SAP: A Descriptive Field Study**	Regina Hebig, Holger Giese
57	978-3-86956-191-2	**Industrial Case Study on the Integration of SysML and AUTOSAR with Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Stefan Neumann, Sebastian Wätzoldt
56	978-3-86956-171-4	**Quantitative Modeling and Analysis of Service-Oriented Real-Time Systems using Interval Probabilistic Timed Automata**	Christian Krause, Holger Giese
55	978-3-86956-169-1	**Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium**	Peter Tröger, Andreas Polze (Eds.)
54	978-3-86956-158-5	**An Abstraction for Version Control Systems**	Matthias Kleine, Robert Hirschfeld, Gilad Bracha
53	978-3-86956-160-8	**Web-based Development in the Lively Kernel**	Jens Lincke, Robert Hirschfeld (Eds.)
52	978-3-86956-156-1	**Einführung von IPv6 in Unternehmensnetzen: Ein Leitfaden**	Wilhelm Boeddinghaus, Christoph Meinel, Harald Sack
51	978-3-86956-148-6	**Advancing the Discovery of Unique Column Combinations**	Ziawasch Abedjan, Felix Naumann
50	978-3-86956-144-8	**Data in Business Processes**	Andreas Meyer, Sergey Smirnov, Mathias Weske
49	978-3-86956-143-1	**Adaptive Windows for Duplicate Detection**	Uwe Draisbach, Felix Naumann, Sascha Szott, Oliver Wonneberg

ISBN 978-3-86956-228-5
ISSN 1613-5652