DISS. ETH NO. 20314

Scaling Out Column Stores: Data, Queries, and Transactions

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

STEFAN HILDENBRAND

Master of Science in Computer Science, ETH Zurich

born March 8, 1982

citizen of Starrkirch-Wil, SO

accepted on the recommendation of

Prof. Dr. Donald A. Kossmann, examiner
Prof. Dr. Carsten Binnig, co-examiner
Prof. Dr. Thomas Neumann, co-examiner
Prof. Dr. Kenneth A. Ross, co-examiner

2012

Scaling Out Column Stores: Data, Queries, and Transactions

Stefan Hildenbrand

Abstract

The amount of data available today is huge and keeps increasing steadily. Databases help to cope with huge amounts of data. Yet, traditional databases are not fast enough to answer the complex analytical queries that decision makers in big enterprises ask over large datasets. This is where column stores have their field of application. Tailored to this type of on-line analytical processing (OLAP), column stores enable informed decisions based on queries over huge amounts of data.

The idea of physically organizing relational data in columns instead of rows and applying powerful compression techniques helped a long way with OLAP workloads. Yet, the requirements become increasingly demanding based on the following three issues: a) the amount of data is increasing faster than ever before, b) responses to analytical queries based on a slightly out-of-date snapshot are not sufficient anymore but these queries have to be answered based on live, up-to-date data (so called operational business intelligence), and c) most workloads are not pure OLAP workloads but rather a mix of OLAP and OLTP (on-line transactional processing) workloads. These issues have two consequences for the development of column stores: 1) Column stores must be designed to *scale-out*, i.e., if more machines are available, the system should utilize these additional resources to achieve superior performance. Furthermore, the performance requirements can only be met if all data can be kept in main memory. However, with an increasing amount of data, the main memory of a single machine may not be sufficient for some workloads. In this case, only the aggregated amount of main memory of a number of machines will suffice, hence the need for distribution of the data. 2) The limited support for transaction handling in column stores does no longer suffice for mixed OLAP/OLTP workloads. In column stores, transaction handling is usually implemented as an exception rather than the normal (optimized) case, and only simple, centralized solutions are implemented for distributed transaction processing. As these solutions are insufficient for the increasing demands, the design of column stores needs to focus more on the performance of transactional processing, especially in the distributed scenario.

Column stores apply powerful compression techniques. One compression technique in column stores is dictionary compression: instead of storing the actual data, only a reference to an entry in the dictionary is stored in the table. Most dictionary-compression techniques encode the data in an order-preserving way, i.e., if the code of a value x is smaller than the code of another value y, i.e. $code(x) < code(y)$, it implies that $x < y$ and vice versa. In a distributed scenario, data has to be partitioned. In this case, traditional column stores apply compression after partitioning, i.e., the compression is per column

and partition. However, many query operations process data stored in multiple columns or partitions. Since data from different columns or partitions is compressed using a different dictionary, the data has to be decompressed before processing. To improve the performance of such queries, this thesis presents the use of a global dictionary. The global dictionary encodes data from many columns and partitions in an order-preserving way. The requirements for such a global dictionary are different from traditional dictionaries since it has to store more data and receives more updates. This thesis makes two contributions towards a global dictionary: a) data structures to store the data of an order-preserving dictionary, and b) an order-preserving encoding scheme that supports lazy updates.

When data is partitioned, this can also be leveraged in other areas. If partitioning works well, many operations access data from one single partition. Today's data stores do not take advantage of this fact when it comes to transaction handling. If the system does support distributed transactions, the transactional properties are enforced on all transactions. This is implemented using (conceptually) centralized approaches. One important transactional property is *isolation*: a transaction should see the database as if it were alone in the system, i.e., it should not bother about concurrent transactions. Many data stores support *snapshot isolation* to achieve this property. With snapshot isolation, every transaction gets its own (virtual) copy of the database to work with, and conflicts are sorted out by the system on commit.

The third contribution of this thesis is to improve snapshot isolation in the context of distributed databases, here, transactions that only access data from a single partition should not have to access the central coordinator but proceed only with local interaction. This thesis gives a general definition of distributed snapshot isolation and provides protocol variants to implement it. The main contribution is a technique called *incremental*, which can be implemented transparently to the application. This technique improves performance significantly in certain distributed scenarios.

Kurzfassung

Die Datenmenge, die uns heute zur Verfügung steht, ist riesig und wächst ständig weiter. Datenbanken helfen dabei, mit riesigen Mengen an Daten umzugehen. Aber auch traditionelle Datenbanken können die komplexen analytischen Anfragen, welche Manager in grossen Unternehmen über grosse Datenmengen stellen, nicht mehr schnell genug beantworten. Solche Anfragen sind die Stärke von Column Stores. Zugeschnitten auf dieses sogenannte on-line Analytical Processing (OLAP), ermöglichen Column Stores fundierte Entscheidungen basierend auf Anfragen über grosse Mengen an Daten.

Die Idee, relationale Daten in Spalten statt in Zeilen abzuspeichern, und die umfassende Anwendung von Kompressionstechniken haben die Leistung für OLAP Workloads verbessert. Trotzdem steigen die Leistungsanforderungen ständig weiter. Drei Herausforderungen zeichnen sich ab: a) die Menge an Daten steigt heute schneller als je zuvor; b) es reicht nicht mehr, Anfragen basierend auf minim veralteten Daten zu beantworten, die Antworten auf analytische Anfragen müssen auf aktuellen Daten basieren (sogenannte operational business intelligence); und c) die meisten Anwendungen erfordern nicht ausschliesslich OLAP sondern stellen eine Mischung von OLAP und OLTP (online Transactional Processing) Anforderungen an das System. Diese Herausforderungen haben zwei Konsequenzen für die Weiterentwicklung von Column Stores: 1) Column Stores müssen ihre Leistung steigern können, wenn dem System mehr Maschinen zur Verfügung gestellt werden. Das Schlüsselwort lautet *scale out*. Der Grund dafür ist, dass die Anforderungen nur erfüllt werden können, wenn die Daten im Hauptspeicher gehalten werden. Dies ist aber infolge der zunehmenden Datenmengen nicht mehr immer möglich mit einer einzelnen Maschine. In manchen Fällen ist nur der Hauptspeicher von mehreren Maschinen zusammengezählt ausreichend, um die Anforderungen erfüllen zu können. 2) Die begrenzte Unterstützung von Transaktionen in Column Stores ist nicht mehr ausreichend für die Verarbeitung von gemischen OLAP und OLTP Anfragen. In aktuellen Column Stores ist die Unterstützung von Transaktionen meist eher als Ausnahmefall denn als optimierter Regelfall implementiert. Zudem existieren meist nur einfache, zentralisierte Mechanismen für das verteilte Szenario. Diese Mechanismen sind nicht mehr ausreichend um mit den erhöhten Anforderungen mithalten zu können, daher müssen Column Stores die Verarbeitungsgeschwindigkeit von Transaktionen – besonders im verteilten Szenario – verbessern.

Column Stores benutzen mächtige Kompressionstechniken. Eine verteilte Datenbank erfordert, dass die Daten partitioniert werden. Eine eingesetzte Kompressions-Technik ist die sogenannte Dictionary Compression oder Wörterbuch-Kompression: an Stelle

der eigentlichen Daten wird nur ein Verweis auf den entsprechenden Eintrag in einem Wörterbuch gespeichert. Darüber hinaus kodieren die meisten Dictionary Compression-Verfahren die Daten in einer ordnungs-erhaltenden Weise, d.h., wenn der Code eines Wertes x kleiner als der Code eines anderen Wertes y ist ($code(x) < code(y)$), folgt daraus, dass $x < y$. Im verteilten Szenario wenden die meisten Column Stores Kompression nach der Partitionierung an. Das bedeutet, die Kompression findet pro Spalte und Partition statt. Allerdings verarbeiten viele Abfrage-Operationen Daten aus mehreren Spalten oder Partitionen. Da die Daten aus verschiedenen Spalten oder Partitionen mit unterschiedlichen Wörterbüchern komprimiert wurden, müssen die Daten vor der Verarbeitung dekomprimiert werden. Um die Geschwindigkeit von solchen Anfragen zu verbessern, präsentiert diese Arbeit die Verwendung eines globalen Wörterbuches. Das globale Wörterbuch kodiert die Daten von vielen Spalten und Partitionen in einer ordnungs-erhaltenden Weise. Die Anforderungen an ein solches globales Wörterbuch unterscheiden sich von traditionellen Wörterbüchern, da es mehr Daten speichern und mehr Updates verarbeiten muss. Diese Arbeit leistet zwei Beiträge für ein globales Wörterbuch: a) Datenstrukturen um die Daten eines ordnungs-erhaltenden Wörterbuch zu speichern und b) ein ordnungs-erhaltendes Codierverfahren, das es erlaubt, Updates erst zu verarbeiten, wenn die Zeit günstig ist.

Wenn die Daten partitioniert sind, kann das auch in anderen Bereichen Vorteile bringen. Wenn die Partitionierung gut funktioniert, müssen viele Operationen nur auf Daten aus einer einzigen Partition zugreifen. Aktuelle Datenbanken nutzen diese Tatsache bei der Abwicklung von Transaktionen allerdings nicht aus. Falls das System überhaupt verteilte Transaktionen unterstützt, werden die Transaktions-Eigenschaften für auf alle Transaktionen gleichermassen erzwungen. Hierzu werden meist (konzeptuell) zentralisierte Ansätzen benutzt. Eine wichtige Eigenschaft von Transaktionen ist *Isolation*: Eine Transaktion sieht die Datenbank als ob sie alleine im System wäre. Parallele Transaktionen stören sich also nicht gegenseitig. Viele Datenbanken setzen *Snapshot Isolation* ein, um dies zu erreichen. Mit Snapshot Isolation erhält jede Transaktion eine eigene (virtuelle) Kopie der Datenbank mit der sie arbeiten kann. Konflikte werden durch das System am Ende jeder Transaktion erkannt.

Als dritter Beitrag dieser Arbeit wird Snapshot Isolation im Zusammenhang mit verteilten Datenbanken verbessert: Transaktionen, die nur auf Daten aus einer einzigen Partition zugreifen, sollen nicht über den zentralen Koordinator laufen, sondern nur mit lokaler Interaktion abgewickelt werden. Diese Arbeit präsentiert erst eine generelle Definition von verteilter Snapshot Isolation und zeigt dann Varianten auf, um diese Definition zu implementieren. Das wichtigste Element dabei ist eine neue Technik, genannt *incremental*, die für die Anwendung transparent implementiert werden kann. Diese Technik verbessert die Leistung erheblich, wenn die Partitionierung gut funktioniert (d.h., wenn viele Transaktionen lokal sind).

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 3
- 1.3 Contributions . . . 5
- 1.4 Structure . . . 5

2 Column Stores **7**
- 2.1 Overview and Use Cases . . . 8
- 2.2 Delta Indexing . . . 10
- 2.3 Dictionary Compression . . . 11
- 2.4 Multi-version Concurrency Control and Snapshot Isolation . . . 12

3 Data structures for a Global Dictionary **15**
- 3.1 Motivation . . . 15
- 3.2 Related Work . . . 20
- 3.3 Designing Data structures for a Global Dictionary . . . 20
  - 3.3.1 Dictionary Operations . . . 20
  - 3.3.2 Shared-leaves Indexing . . . 21
  - 3.3.3 Requirements and Design Decisions . . . 24
- 3.4 Leaf Structure . . . 26
  - 3.4.1 Memory Layout . . . 26
  - 3.4.2 Leaf Operations . . . 28
- 3.5 Cache-conscious Indexes . . . 31
  - 3.5.1 CS-Array-Trie . . . 32
  - 3.5.2 CS-Prefix-Tree . . . 35
- 3.6 Evaluation . . . 39
  - 3.6.1 Efficiency of Leaf Structure . . . 40
  - 3.6.2 Efficiency of Encoding Indexes . . . 42
  - 3.6.3 Scalability of the Dictionary . . . 43
- 3.7 Conclusion . . . 44

4 Lazy Updates for an Order-preserving Dictionary **47**
- 4.1 Motivation . . . 48
- 4.2 Related Work . . . 51

4.3 Problem Statement . . . 51
4.3.1 Use Case: Global Dictionary . . . 51
4.3.2 Requirements . . . 52
4.3.3 Other Use Cases . . . 53
4.4 Multi-Version Encoding . . . 53
4.4.1 Basic Concepts . . . 53
4.4.2 Fixed-size Code Representation . . . 55
4.4.3 Analysis of Update Patterns . . . 56
4.5 Global Dictionary Compression . . . 56
4.5.1 Overview . . . 56
4.5.2 Encoding . . . 58
4.5.3 Decoding . . . 62
4.5.4 Query Processing . . . 66
4.6 Optimizations . . . 67
4.7 Evaluation . . . 70
4.7.1 Query Processing . . . 72
4.7.2 Version Mapping . . . 73
4.7.3 Encoding and Decoding . . . 75
4.8 Conclusion . . . 78

5 Incremental Snapshots for Distributed Snapshot Isolation 81
5.1 Motivation . . . 81
5.2 Related Work . . . 84
5.2.1 Concepts of Snapshot Isolation . . . 84
5.2.2 Distributed SI in Restricted Setups . . . 84
5.2.3 Snapshot Isolation in Replicated DBMS . . . 85
5.3 Background . . . 86
5.3.1 Architecture . . . 86
5.3.2 Expected Workload . . . 87
5.3.3 Conceptual Database Model . . . 88
5.3.4 Local Snapshot Isolation . . . 89
5.4 Distributed Snapshot Isolation . . . 91
5.4.1 Definition . . . 91
5.4.2 Anomalies in the Distributed Setting . . . 93
5.4.3 Correctness Criteria . . . 95
5.4.4 Proof Based on SSGs . . . 98
5.5 Centralized Coordination . . . 100
5.6 Pessimistic Coordination . . . 101
5.7 Optimistic Coordination . . . 102
5.8 Incremental Snapshots . . . 105
5.9 Evaluation . . . 109
5.9.1 Benchmark Details . . . 109
5.9.2 Horizontal Scalability . . . 112
5.9.3 Comparison Between the Approaches . . . 113

5.9.4 Scaling Number of Nodes in a Transaction 115
5.9.5 Impact of Insufficient Knowledge 116
5.10 Conclusion 118

6 Conclusions **119**
6.1 Summary 119
6.2 Future Work 120

Bibliography **123**

1 Introduction

1.1 Motivation

Nowadays, column stores are both a hot research topic and commercially successful products. The main idea is simple: data is physically organized in columns instead of rows, i.e., the data from the same attribute of a relational table is aligned continuously in memory. Thus, in order to reconstruct a tuple from this relational table, the data has to be combined from the different columns.

Currently, the main application area for column stores are so called OLAP workloads: on-line analytical processing. These are the kinds of questions decision makers of any company tend to ask: aggregations and other analytic queries. Column stores can answer most kinds of such queries in reasonable time even on a huge database. This is for two reasons: 1) Column-oriented storage is better suited for those kinds of queries: Aggregation and analytic queries usually only access a few attributes of many rows from a table (e.g., to calculate the total revenue per month only the month and sales attributes have to be accessed). In column stores, the data of one attribute is stored sequentially in memory (or on disk). Therefore column stores can process aggregations and other queries from OLAP workloads fast since no unnecessary data is brought to the CPU cache. 2) In OLAP scenarios, updates traditionally are processed in bulks (e.g., the data is replicated into the column store over night). This alleviates the disadvantage of rather expensive updates that column stores suffer compared to row stores. An update that requires a single access to memory (or disk) in a row store potentially results in many accesses in different places in a column store.

An additional reason why column stores achieve good performance (in general) is that the design of column stores allows to fully leverage improvements in hardware like more main memory, deeper cache hierarchy, SIMD operations, and multi-core systems. Furthermore, organizing the data in columns leads to an improved compression ratio

since data in the same column tends to be more similar than data in the same row. Better compression implies smaller memory footprint. Therefore column stores are well suited for in-memory databases and can hold more data in-memory as traditional row-organized storage engines. At the moment, in-memory computing is a general industry trend.

However, current workloads rarely are pure OLAP workloads. Most applications provide a mix of OLAP and OLTP (on-line transactional processing) workloads.[1] Given the success of column stores in the area of OLAP, current development aims to improve the performance of column stores in the area of OLTP. One example is the so called operational BI (business intelligence) use case [26]: the goal is to provide the capabilities of column stores for complex data analysis on fresh, live, up-to-date data from the ongoing transactional processing without replication.

Furthermore, the amount of digital data is growing faster than ever before [37]. Thus, in order to cope with the increased load, data stores must *scale out*, i.e., scale with the number of available nodes. Scaling out is important for the following reasons: a) It usually is less expensive to have a number of decent sized commodity machines instead of one huge specialized server machine. b) In some scenarios, the amount of main memory that can be put into one node is simply not sufficient for the task at hand. In these scenarios, only the aggregated memory of a number of nodes is sufficient. c) In some cases, the geographical (or political) setup of a company simply forces the database to be hosted on several nodes. This can help to improve locality or to enable local operations to continue even in case of a network downtime. d) Distribution is still a simple way of achieving fault tolerance by using redundant systems.

In addition to scaling out, there is another dimension: *scaling up*, i.e., scaling with the number of cores in a node. Scaling up is important because the cores in new systems do not get much faster (in terms of processing speed or clock frequency) anymore. To overcome this limitation, hardware vendors put more cores in the same amount of space. Thus, new hardware still does improve performance—if the software can put the additional cores to use.

Distributing a data store (be it inside a machine over many cores or over a number of machines) requires partitioning the data. Distribution in the area of column stores is not yet well analyzed. Column stores are optimized for read-mostly, query-intensive workloads (i.e., OLAP workloads). Therefore column stores put a lot of effort into compression and other optimizations that improve query performance. Many of these techniques tend to become difficult in the context of distribution. But distribution is required to scale with the number of cores and machines.

The presented trends above are general industry trends. Users of database software want to do reporting on fresh transactional data without replicating it to a data warehouse. Therefore updates become more common for column stores. Furthermore, OLAP scenarios can be processed orders of magnitude faster if a) the data is organized in

[1] In contrast to OLAP workloads, OLTP workloads mainly access many attributes of a few rows at a time (e.g., to insert a new sales order).

columns, b) the data can be compressed well, and c) the data fits into main memory. If the data does not fit into the main memory of one node, it is partitioned and distributed over multiple nodes. Consequently, this thesis introduces ideas related to these trends, namely: a) a global dictionary to improve compression and alleviate the impact of distribution, b) an order-preserving encoding schema for the dictionary to cope with the increased update rate, and c) improvements to distributed snapshot isolation to address the growing need of fresh data for analytics in distributed databases.

Since this thesis addresses general industry trends, we evaluate the ideas with a commercial column store, the in-memory column engine of SAP HANA DB.[2] The vendor of this column store, SAP, provided the real-world background for the mentioned issues. Close collaboration and reality checking of the ideas provided helpful insight and generated new directions in our research. At the same time some restrictions were imposed. Often, these restrictions turned out to be useful guidelines that limited a potentially unbounded search space.

1.2 Problem Statement

Motivated by the considerations in the previous section, this thesis tackles the three following issues to improve the scalability of column stores.

Data structures for a global dictionary. In column stores, the data is usually compressed using dictionary compression (see Section 2.3). The traditional solution in a distributed (i.e., partitioned) database is to use one dictionary per column and partition. However, this has a negative impact on performance: some query processing operations (e.g., set operations or joins) cannot be restricted to individual partitions (or columns). Such operations tend to access data from multiple partitions (or columns). Since data from different partitions (or columns) is encoded using a different dictionary, the data has to be decompressed before the query operator can be evaluated, which decreases performance. Other approaches use a single dictionary for many partitions or columns but sacrifice the property that the dictionary is order-preserving. However, that property is important to improve performance of certain queries (e.g., range queries or sorting).

Thus, the goal is to have a global order-preserving dictionary that encodes data from all partitions and columns that potentially are processed together. However, traditional order-preserving dictionary approaches are not sufficient because they cannot cope with the update patterns that occur when data from many partitions or columns is encoded using the same dictionary. Furthermore, a global dictionary tends to contain much more data than a traditional dictionary for a single partition (or column).

(1) Consequently, the first issue tackled in this thesis is to design and implement data structures for an order-preserving dictionary that supports the update patterns of a

[2]The column store engine of SAP HANA DB evolved from the SAP business intelligence accelerator (BIA).

global dictionary, i.e., when data from many partitions or columns is encoded using the same dictionary. The goal is to improve query performance without loosing flexibility or performance for updates.

Encoding scheme for a global dictionary. If an order-preserving global dictionary is used, not only the data structures have to cope with the increased load and more difficult update patterns. A change in the global dictionary has huge impact on the entire database. Thus, an encoding scheme is needed that requires few such changes. The obvious answer, namely using a variable-length encoding, does not work well in column stores. This is because column stores perform best if they exactly know where the next value starts, and the distance between two consecutive values is small. Therefore, fixed-length codes are required. It follows that changes to the codes are inevitable when updates occur. The goal is to make these changes as cheap as possible.

(2) Consequently, the second issue tackled in this thesis is to come up with an order-preserving encoding scheme that uses fixed-length codes while still supporting updates. Furthermore, changes to existing codes should be as rare as possible. Since such changes cannot be avoided completely, the encoding scheme should enable lazy re-encoding of existing values.

Distributed Snapshot Isolation. Providing transactional guarantees like the ACID properties is an important aspect of data processing. This aspect gets more and more important in column stores. This is because the advantages of column stores are very attractive such that the trend is towards column stores that also perform well with OLTP workloads.

Multi-version concurrency control (MVCC) and snapshot isolation are a typical way to support transactions in column stores (see Section 2.4). However, current implementations are built on ample central coordination. This used to be fine as long as transactions with increased isolation requirements were the exception. With the trend towards more transactional processing, the central coordination becomes an inherent bottleneck. This problem is even more evident if the database is scaled out, i.e., if more and more nodes are added to the system. Existing alternative solutions require additional information from the application about the intentions of a transaction before the transaction actually started. Furthermore, current definitions of snapshot isolation are not sufficient for certain distributed scenarios.

(3) Consequently, the third issue tackled in this thesis is to define distributed snapshot isolation in a general way and to design a protocol to implement distributed snapshot isolation with low central coordination. At the same time, this protocol has to be implemented transparently, i.e., it must not require additional information from the application.

1.3 Contributions

In summary, this thesis makes three main contributions:

(1) *shared-leaves*, a set of new data structures for an order-preserving dictionary that maps strings to integers and vice versa. These data structures are one building block for a global dictionary. The data structures are optimized for bulk-updates since that is the common update pattern in OLAP style workloads. One part is a new leaf structure to store the data that is optimized to provide access in both directions. On top of that leaf structure, we build two new cache-conscious string indexes for the mapping from strings to integers. For the mapping in the other direction (integers to strings), we re-use the CSS-Tree [66], a cache-aware tree structure.

Compared to other data structures, *shared-leaves* has a smaller memory footprint while still providing direct access to the data. Existing approaches either require more memory or use additional indirection.

(2) *multi-version encoding*, an order-preserving enumeration scheme that supports lazy updates of codes. In contrast to existing work, this enumeration scheme uses fixed-length codes by outsourcing the variable part to special data structures. Furthermore, it works without any assumptions on the workload.

(3) *incremental*, a new technique for snapshot isolation in distributed databases. Moreover, we extend two approaches from federated databases to the distributed scenario. These techniques are investigated in the area of distributed column stores but can be applied to any database that implements snapshot isolation. The new technique, *incremental*, outperforms existing approaches in the general case and matches the performance of existing approaches in scenarios in which all restrictions of existing approaches are fulfilled. Incremental does not require additional knowledge about the transactions and therefore can be implemented transparently in an existing data store to improve performance.

While all three contributions are motivated by use cases in the area of column stores, they are applicable in a more general context. We will discuss this as we go along.

1.4 Structure

The rest of this thesis is structured as follows:

- Chapter 2 introduces column stores on a high level and presents the main building blocks required later in the thesis. It focuses on the column store engine of SAP HANA DB that this thesis mainly worked with.
- Chapter 3 presents data structures for an order-preserving dictionary to compress data from multiple columns of a column store. The goal is to improve performance of query operations across multiple machines.

- Chapter 4 then continues in the same scenario and presents an order-preserving encoding schema that supports lazy updates and does not make any assumptions about the workload.
- Chapter 5 presents *incremental*, a new technique to efficiently implement snapshot isolation in a distributed data store. In addition to that, two existing ideas, *pessimistic* and *optimistic* are extended to support the generalized definition of distributed snapshot isolation.
- Chapter 6 summarizes the thesis and presents future work.

2 Column Stores

The goal of this chapter is to present the basics for the following chapters. This is not complete view of on-going research in the area but we present what has proven useful in the commercial column store we are working with in this thesis.[1]

It is not the goal of this chapter to give a full overview of the wide range of work that was and is still going on in the area of column stores. As an entrance to the world of column stores, a tutorial [2] and the Ten Year Best Paper Award [58] at VLDB 2009 are a good start. Furthermore, the PhD thesis of Peter Boncz [22] presents the design of MonetDB. The PhD thesis of Daniel Abadi [1] presents the blueprints of C-Store, a column store built from scratch. In this thesis, we build upon the following fact which has been established by both empirical research and commercial success: column stores have their advantages, especially on today's hardware.

This chapter is organized as follows:

- Section 2.1 gives a high-level overview of column stores in general and presents some use cases.
- Section 2.2 presents the concept of delta-indexing, an approach where updates are collected in a separate data structure and added to the main data structure only periodically. Although this thesis makes no contributions in that direction, delta-indexing is an important building block and heavily influences the thinking in all other parts of the column store.
- Section 2.3 discusses the area of compression in column stores, mainly dictionary compression.

[1] This has two consequences: a) we cannot go into too much detail as the success of a database depends on an advantage over the competitors, which can very well be performance caused by superior implementation, and b) while we mention alternative approaches, the focus will always be on what can be implemented within the existing system.

- Section 2.4 gives a high-level overview on how multi-version concurrency and snapshot isolation is implemented in the column store of SAP HANA DB. The ideas on snapshot isolation presented in Chapter 5 originated from that implementation but are generally applicable.

2.1 Overview and Use Cases

Column stores physically organize relational tables in columns instead of rows. Therefore the data from the same attribute (i.e., column) is stored sequentially in memory or on disk. This implies that only the required data needs to be read by the CPU, i.e., brought to the cache. Aggregation or filter queries, for example, can then be processed very fast. This is because only the relevant data is read, which reduces the required memory bandwidth [44]. To reconstruct a tuple consisting of multiple attributes, the data has to be fetched from the different columns.

The idea of organizing data in columns instead of rows is column stores is not new [12, 30]. But it took quite some time until the advantages of that data organization were exploited, mainly because the idea only pays off with modern hardware, e.g., more main memory, more CPU cores, and a deeper cache hierarchy. Furthermore, the main sweet spot of column stores in terms of applications are query-intensive workloads, so called OLAP workloads: on-line analytical processing. This kind of workload became increasingly popular in the last years [2, 58]. Today, a number of commercial, open source and research column stores are available, e.g., MonetDB, vertica, or SAP HANA DB to just name a few.

The main market of column stores is data analytics and other OLAP-style workloads [1]. This is because the design of column stores enables them to efficiently scan, filter and aggregate data [2]. This is also the area in which our column store is mainly operating. Chapters 3 and 4 work with this use case.

However, the trend is towards a mix of OLAP and OLTP (on-line transactional processing) workloads. OLTP workloads usually consist of updates to single rows. This kind of update is traditionally a weak point of column stores. To alleviate the impact of this issue, a few improvements (e.g., delta-indexing as presented in Section 2.2) were introduced. However, this is not sufficient anymore due to the growing demands. There are at least two approaches to cope with these mixed workloads: a) The first approach is to use hybrid or adaptive data stores, i.e., a clever combination of row stores for the update-intensive parts and column stores for the query-intensive parts. Again, this is not a new idea: [5, 43, 65]. Actually, the column store we work with in this thesis is part of a larger software stack that includes both row and column storage engines under a common interface. The goal is to achieve good performance for both update- and query-intensive workloads. b) The second approach is to improve performance of updates and transactional processing in column stores. In some sense, this thesis works in both directions: the techniques presented in Chapter 5 aims to improve performance of data stores in the transactional context. We investigate the techniques in the context of column stores but they are applicable in any data store and thus could also be used

in a hybrid storage engine.

Three main trends in todays hardware heavily influence the design and implementation of data stores. Consequently, they also have impact on this thesis.

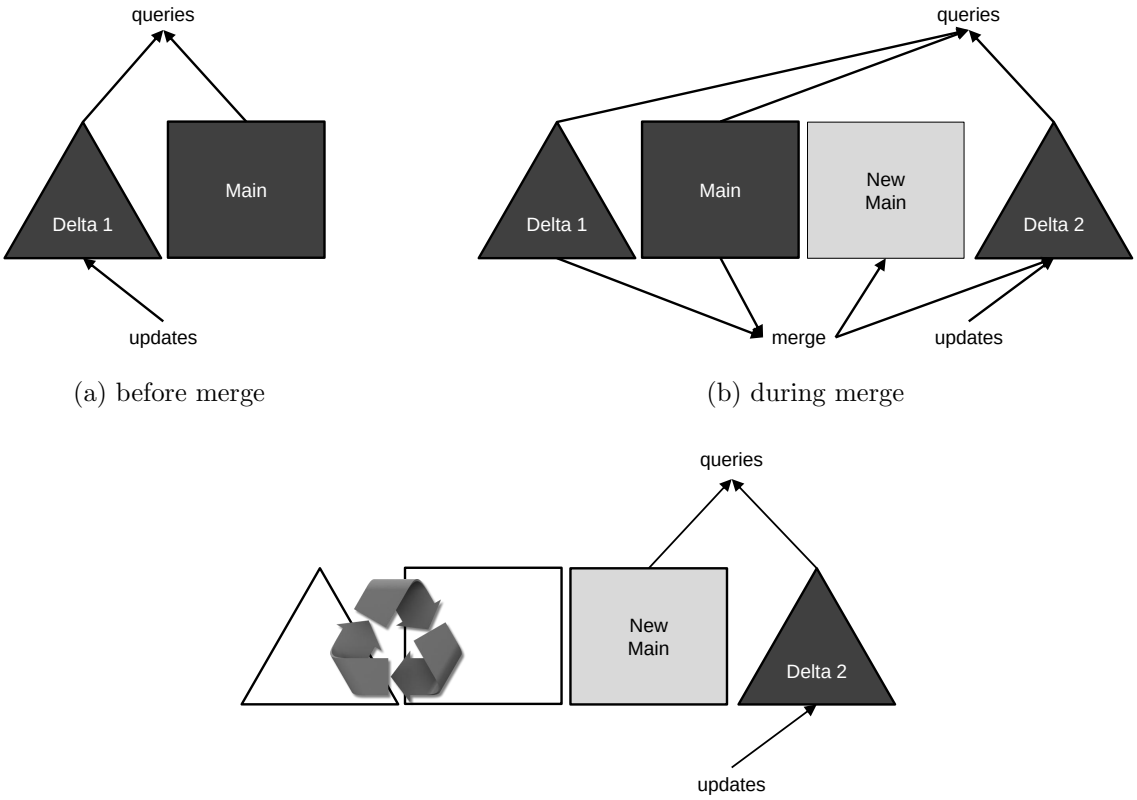
Main Memory One of todays buzz-words is "in-memory computing". The term does not make much sense, since the actual computation always happens in memory as it needs input data and produces an output. But the key observation here is that in many cases nowadays *all* data used for processing fits into the main memory of decent commodity hardware or maybe moderately specialized server systems. The rules for data access in main memory are inherently different than what programmers were used to in the disk drive era. Locality is not dead, it is just different or more precisely, it happens on a different level: cache-awareness is important. Consequently, the data structures presented in Chapter 3 are designed to be cache-aware.

The amount of main memory used by a system is an important cost factor (in terms of money). Furthermore, memory bandwidth can be a limiting factor for performance. Therefore compression and especially lightweight compression is still a technique that can bring benefits. But these benefits have to be re-evaluated in the context of todays hardware. Consequently, Chapters 3 and 4 discuss dictionary compression and pay close attention to keeping the memory footprint small.

Distribution While the trend in today's hardware goes towards more computing power on less space, there are still many scenarios where distribution is indispensable. The amount of data continues to increase. There are business warehouses that do not fit into the main memory of a single machine, or a single machine with that much main memory is too expensive. Thus, in order to keep the data in main memory (for fast processing), the data has to be distributed. Other reasons for distribution are redundancy, e.g., by having a hot backup of the database for recovery from failures; and geographic distribution, e.g., the database system for a multi-national company with inter-connected data centers local to the branches of the company.

Multi Core While the clock frequency of CPUs did not increase much in the last few years, the increase of computing power (and therefore the effect of Moore's Law) was achieved by increasing the number of cores per node. This imposes new challenges on software developers. Programs do not simply run faster because the user bought a more powerful machine. If the software is not prepared to run in parallel, a new machine is not going to bring much benefit. In Chapter 3, we address this issue explicitly and discuss how the presented data structure can be accessed in parallel.

One may ask the question whether column stores have already hit their peak. Trends and new concepts like NoSQL databases, key-value stores and the map-reduce-paradigm



(a) before merge

(b) during merge

(c) after merge

Figure 2.1: Delta indexing approach

challenge the dominance of relational databases and therefore question the very core of column stores. The ideas presented in this thesis originated from issues in the column store world. But it turns out that all of them are independent of column stores. Dictionary compression (Chapters 3 and 4) can be useful in other data-intensive applications. Consistency is an important property of any data-oriented application and the the improvements presented in Chapter 5 are not limited to column stores or relational stores.

2.2 Delta Indexing

The issue that a data structure is optimized for reading and not for update in-place is old. For example, differential files [71] separate the area where new or changed data is written from the area where the last stable state of the data resides. Therefore updates happen in a restricted area and can be processed faster. But the tradeoff is also clear: for queries that require the latest state of the data, both the stable data and the differential file have to be read.

Column stores suffer from a similar issue: the data structure is usually optimized

for queries (i.e., the analytical workload) and updates expensive. Consequently, many column stores, e.g., C-Store [74] and SAP HANA DB, use a similar idea like the differential file. We call this approach *delta indexing*. Figure 2.1 illustrates the idea. New or changed data is written to a write-optimized data structure (called *delta index*). In order to read the current state of the database, queries have to consider both main and delta index. From time to time (or upon request) this delta index is merged to the read-optimized (i.e., sorted and compressed) main data structure. During the merge process (see Figure 2.1(b), the main data structure is completely rebuilt in an optimized way. In that phase, queries go to the old main and both old and new delta index. Updates go to a new delta index.[2] Once the construction of the new main is complete, the new main replaces the old main and the delta index can be reused for the next round.

Delta indexing works quite well as long as there is time, memory, and computing power available for the merge process. If the update rate is too high, i.e., if the merge phase takes too long, the system does not perform well. So, this technique can only reduce the impact of updates on performance, but it cannot bring column stores to the same update performance as a well optimized row store.

This thesis makes no contributions in the area of delta indexing. But the fact that SAP HANA DB uses delta indexing extensively, influences many design decisions and tradeoffs in the system. For example, compression (see next section) can be used with less worries about updates. Furthermore, finding the current version of an object in the database becomes more difficult. Section 2.4 shows how this issue can be solved.

2.3 Dictionary Compression

Data compression is a widely applied technique to reduce the cost (in terms of space or time depending on the scenario) of data processing. In column stores lightweight compression schemes, namely dictionary compression, work particularly well [3]. Since the data is organized in columns rather than rows, the data is grouped by domain. In most cases this implies that adjacent data is similar in some sense (at least more similar than values from different attributes of a row). In such cases, compression works well because the entropy of the data is lower.

In SAP HANA DB, dictionary compression is applied extensively. With very few exceptions, all data (including numerical data) is compressed with this technique. The idea is illustrated in Figure 2.2 and works as follows: For every column, there is a separate dictionary that contains a sorted list of the distinct values that exist in the corresponding column. The data is then stored using the index of the value in the list. On top of that, bit compression is applied. E.g., if the dictionary only contains 16 distinct values, only 4 bits are used to represent the data in the column. Depending on the workload, further compression can be applied [53].

[2] The arrow from the merge process to Delta 2 in Figure 2.1(b) indicates that uncommitted data from Delta 1 is copied to Delta 2 and not written to the main data structure. Therefore the main data structure only contains committed data.

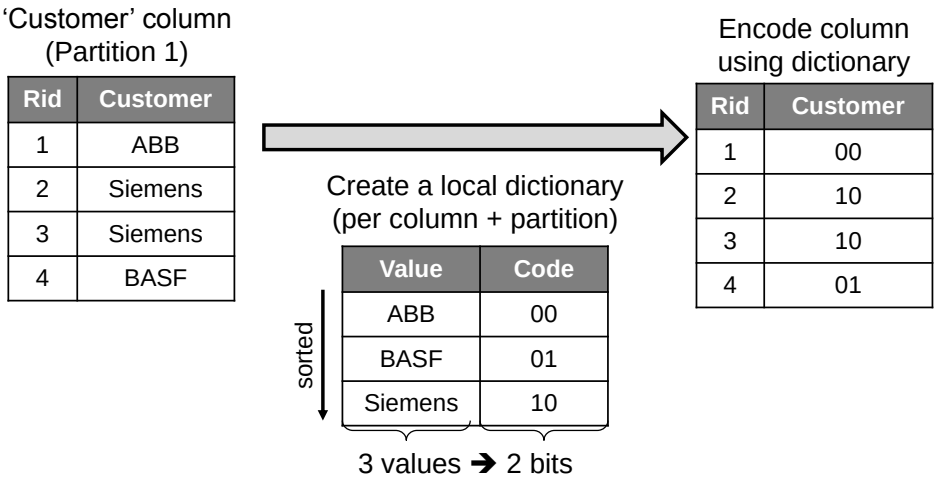


Figure 2.2: Compressing a column using a dictionary

The fact that the dictionary is order-preserving has advantages for query processing, e.g., sorting can happen on the encoded data. The disadvantage is that the dictionary is likely to change with every update and therefore has to be rebuild frequently. This disadvantage is alleviated by the delta index concept (see previous section). Since the column store is optimized for read-mostly workloads, the advantage outweighs the disadvantage [52]. In Chapters 3 and 4 we will argue that this is not the case in some new (in the sense of not traditional for column stores) scenarios and therefore present new ideas for dictionary compression, especially for the distributed scenario.

2.4 Multi-version Concurrency Control and Snapshot Isolation

Even though the traditional workload for column stores are OLAP-style workloads, consistency in the case of concurrent updates is an issue. The goal is to provide every transaction with a consistent view of the database.

Transactions are the units of work within a database. A transaction consists of a begin, one or several read / write operations and either a commit or abort at the end. In databases, several properties of transactions can be guaranteed: atomicity, consistency, isolation and durability (ACID properties). In this thesis, we focus on the *isolation* property. For a transaction x it looks like it runs alone (i.e., isolated) in the database. Effects of other transactions that work concurrently are not visible to transaction x.

To implement isolation and concurrency, SAP HANA DB uses multi-version concurrency control (MVCC) [17, 68] and snapshot isolation [15], one of many possible isolation levels, to guarantee this isolation between transactions. With snapshot isolation every transaction gets its own (virtual) copy of the database to work with. Because of that,

'Order' table (conceptual)

Rid	Customer	Total	Date
1	ABB	12000	2.2.2011
2	Siemens	40000	4.5.2011
3	Siemens	30000	6.8.2011
4	BASF	12000	8.9.2011
3	Siemens	33333	6.8.2011

Bit vectors

base	valid	invalid
1	0	0
1	0	0
1	0	1
0	0	0
0	1	0

Resulting Filter

filter
1
1
0
0
1

Figure 2.3: Filtering the visible tuples in MVCC

every statement within a transaction always reads the state of the database as it was at the beginning of the transaction; including changes from statements of the same transaction, but no changes from concurrent transactions. MVCC is well suited to support this isolation mechanism as it keeps older versions of an object available. Furthermore, a system that enforces snapshot isolation only allows transactions to commit if they did not write the same object as any other concurrent transaction. This is usually implemented as the *first committer wins* rule, i.e., the checks are done on commit. The transaction x that first commits can write its changes. All other transactions that are concurrent to transaction x and write to at least one of the objects transaction x wrote, are aborted. One of the advantages of this scheme is that it can be implemented without locking.

With MVCC "older" data is still available in the system to provide transactions the proper view. This is leveraged for snapshot isolation. In Chapter 5, we use that "old" data in a similar way to provide distributed transactions with the proper view on each node. Moreover, the simplicity of how the view can be selected inspired the techniques in that chapter. In SAP HANA DB, every transaction gets a so called *transaction token (TT)*. That token contains all information about the view of that transaction in a compact format. This information is then turned into filters that are applied to the data in the data store, i.e., data that is not visible for the transaction is filtered out. Figure 2.3 illustrates the idea. The main table is an append-only data structure; in the example, the tuple with Rid 3 is updated which results in an append to the table and an invalidation of the old tuple.

To efficiently select the visible tuples for a transaction, three bit vectors are used: 1) a base vector that contains the visibility information for the part of the data where every transaction has the same view. This base vector is the same for all transactions, 2) a valid vector that contains the information about which tuples are visible for a given transaction (according on the isolation protocol), and 3) an invalid vector that filters out tuples that are not visible anymore for a given transaction. The valid and invalid vectors are different for each transaction.

These bit vectors are then combined to a single filter.[3] From time to time (usually when a delta merge happens, but conceptually independent from that) the vectors are combined and compressed to form a new base vector. If snapshot isolation is used, the base vector can contain all information up to the begin of the oldest still running transaction.

[3]Technically there is one filter per index, i.e., separate filters for the main and the delta indexes.

3

Data structures for a Global Dictionary

In this chapter[1], we argue that order-preserving dictionary compression does not only pay off for attributes with a small fixed domain size as it is traditionally done in column stores. Dictionary compression also works for long string attributes with a large domain size which might change over time. This is the case if the dictionary encodes data from multiple columns or partitions. Such a global dictionary enables more query operations to be executed on the encoded data and therefore improves query performance.

Consequently, this chapter introduces new data structures that efficiently support an order-preserving dictionary compression for (variable-length) string attributes with a large domain size that is likely to change over time. The main idea is that the dictionary is modeled as a table that specifies a mapping from string-values to arbitrary integer codes (and vice versa). A novel indexing approach provides efficient access paths to such a dictionary while compressing the index data. The experiments show that these data structures are as fast as (or in some cases even faster than) other state-of-the-art data structures for dictionaries while being less memory intensive.

3.1 Motivation

Column-oriented database systems (such as Monet-DB [83] and C-Store [74]) perform better than traditional row-oriented database systems on analytical workloads such as those found in decision support and business intelligence applications. Recent work [3, 84] has shown that lightweight compression schemes for column stores enable query processing on top of compressed data and thus lead to significant improvements of the

[1]The contents of this chapter are published in [20].

query processing performance. Dictionary encoding is such a lightweight compression scheme that replaces long (variable-length) values of a certain domain with shorter (fixed-length) integer codes [3]. Chapter 2 explains in more details how column stores and dictionary compression work.

Bit packing is often used on top of dictionaries to further compress the data [44]. This compression scheme calculates the minimal number of bits that are necessary to represent the maximal index into the dictionary. Bit packing makes sense if the size of the domain is stable (or known a-priori). However, in many practical data warehousing scenarios the domain size is not stable. As an example, think of a cube inside a data warehouse of a big supermarket chain which holds the sales of all products per category (e.g., whole milk, low fat milk, fat free milk). While the total number of categories is not too large, it is likely that the categories will change over time (i.e., new products are added to the selection of the supermarket).

In order to deal with situations where the domain size is not known a-priori, existing column stores usually analyze the first bulk of data that is loaded in order to find out the current domain size of a certain attribute (e.g., the total number of product categories) and then derive the minimal number of bits (for bit packing). However, if subsequent bulks of data contain new values that were not loaded previously, existing column stores usually have to decode all the previously loaded data (e.g., the data stored inside the sales cube) and then encode that data again together with the new bulk using more bits to represent the new domain size. This situation becomes even worse if different attributes (that are not known a-priori) share the same global dictionary to enable join processing or union operations directly on top of the encoded data.

In addition, column stores often use order-preserving compression schemes to further improve performance of expensive query operations such as sorting and searching because these operations can then be executed directly on the encoded data. However, order-preserving compression schemes either generate variable-length codes (e.g., [7]) that are known to be more expensive for query processing in column stores than fixed-length codes [44], or they generate fixed-length codes (e.g., by using indexes in a sorted array) that are more difficult to extend when new values should be encoded in an order-preserving way.

In contrast to the existing work, in this chapter we argue that order-preserving dictionary compression does not only pay off for attributes with small domain sizes but also for long string attributes with a large domain size. For example, the sales cube mentioned before could contain product names of type `VARCHAR(100)`. If we encode one million different product names using a dictionary that generates fixed-length integer codes (e.g., 32 bit), we would get a very good average compression rate for that column.

However, using a sorted array and indexes into that array as fixed-length integer codes is too expensive for large dictionaries where the domain size is not known a-priori. There are different reasons for this: First, using a sorted array and binary search as the only access path for encoding data is not efficient for large string dictionaries. Second, if the index into the sorted array is used as integer code, each time a new bulk of string data is loaded it is likely that the complete dictionary has to be rebuilt to generate order-preserving codes and all attributes that use that dictionary have to be re-encoded. For

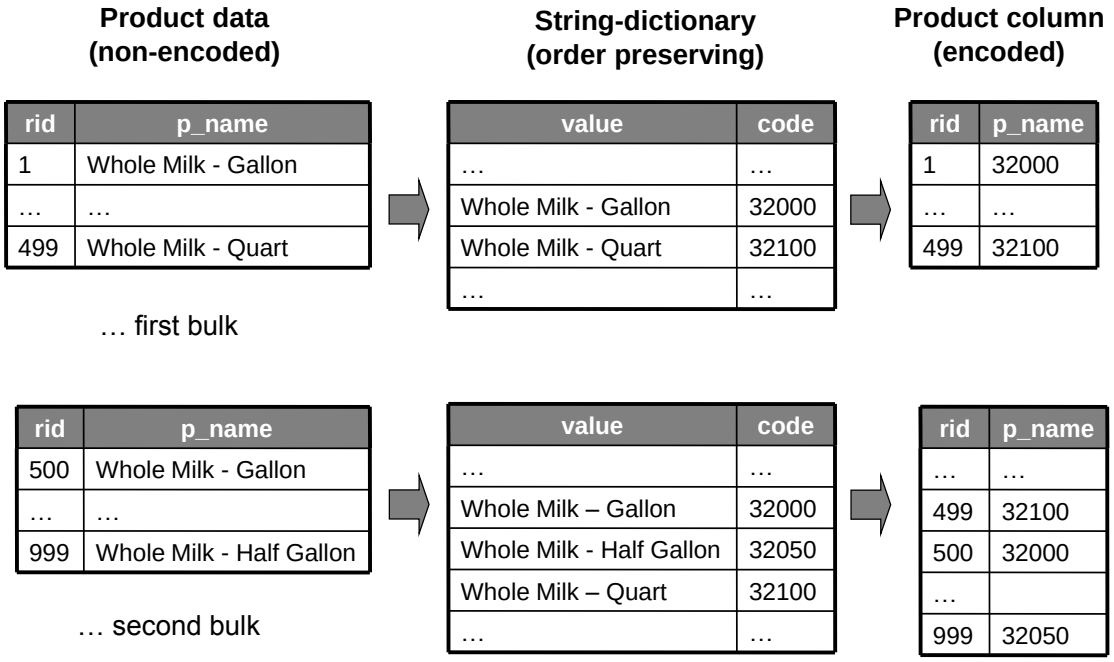


Figure 3.1: Data loading with dictionary-based order-preserving string compression

Query (original):

```
Select SUM(o_total), p_name
From Sales, Products
Where p_name Like 'Whole Milk*'

Group by p_name
```

Query (rewritten):

```
Select SUM(o_total), p_name
From Sales, Products
Where p_name ≥ 32000
  And p_name ≤ 32100
Group by p_name
```

Figure 3.2: Query rewrite with order-preserving string compression

the same reasons, strict bit packing on top of an order-preserving dictionary compression scheme does not make sense either.

Motivated by these considerations, this chapter introduces data structures that efficiently support an order-preserving dictionary-compression of (variable-length) string attributes where the domain size is not known a-priori (e.g., when the dictionary is shared by different attributes). Furthermore, the integer codes that are generated have a fixed length to leverage efficient query processing techniques in column stores. No bit-packing is used on top of these integer codes in order to be able to support updates efficiently. Consequently, the dictionary is modeled as a table that specifies a mapping from string-values to arbitrary integer codes and vice versa. The goal is also to provide efficient access paths (i.e., index structures) to such a dictionary. More precisely, index structures for a string dictionary have to efficiently support the following tasks:

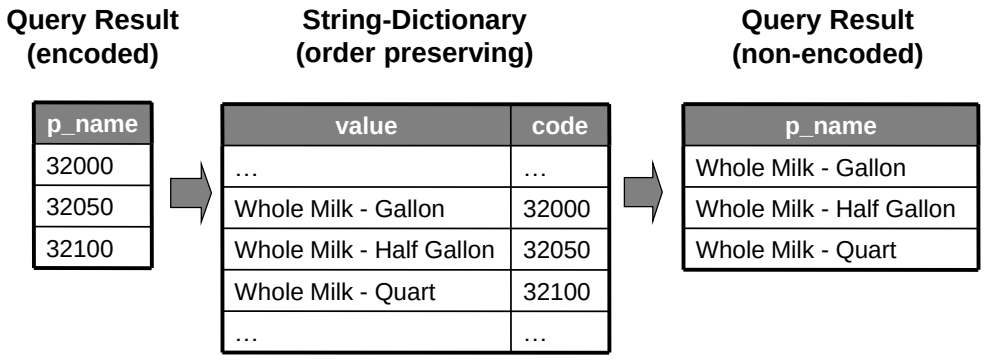


Figure 3.3: Query execution with dictionary-based order-preserving string compression

Data loading As discussed before, data is usually loaded bulk-wise into a data warehouse. Therefore the dictionary must efficiently support the encoding of bulks of string values using integer codes. The encoding logically consists of two operations: the first operation is a bulk lookup of the integer codes for the string values that are already a part of the dictionary and the second operation is the bulk insertion of the new string values as well as the generation of order-preserving integer codes for those new values. As an example, Figure 3.1 shows how two bulks of product data (i.e., the column *p_name*) are loaded into the sales cube.

Query compilation In order to execute analytical queries directly on top of encoded data, it is necessary to rewrite the query predicates. If an order-preserving encoding scheme is used, this step is trivial: The string constants of equality- and range-predicates only have to be replaced by the corresponding integer codes. Moreover, prefix-predicates (e.g., `p_name like 'Whole Milk%'`) can be mapped to range predicates. Consequently, a string-dictionary should enable efficient lookups to rewrite string constants as well as string prefixes. As an example, Figure 3.2 shows how the predicate of a query is rewritten.

Query execution During query execution, the final query result (and sometimes intermediate query results) must be decoded using the dictionary (which can be seen as a semi-join of the encoded result with the dictionary). As most column stores use vectorized query operations (or sometimes even materialize intermediate query results) [44, 84], a string-dictionary should also support the efficient decoding of the query results for bulks (i.e., bulk-lookups of string values for a given list of integer codes). As an example, Figure 3.3 shows how the dictionary is used to decode the column *p_name* of the encoded query result.

While all the tasks above are time-critical in today's data warehouses, query processing

is the most time-critical one. Consequently, the data structures presented in this chapter should be fast for encoding but the main optimization goal is the performance of decoding integer codes during query execution. To achieve this, efficient (cache-conscious) indexes that support the encoding and decoding of string-values using a dictionary are introduced. While there has already been a lot of work to optimize index structures for data warehouses on modern hardware platforms (i.e., multi-core-systems with different cache levels), much of this work concentrated on cache-conscious indexes for numerical data (e.g., the CSS-Tree [66] and the CSB$^+$-Tree [67]). However, there has been almost no work on indexes that enable (cache-)efficient bulk lookups and insertions of string values. Consequently, the focus of this chapter is on indexes for encoding string data.

In addition, the amount of main memory as well as the size of the CPU caches of today's hardware are constantly growing. Consequently, data warehouses start to hold all the critical data in main memory (e.g., the data that is used for on-line analytical reporting). Thus, this chapter addresses the question of how to compress the dictionary in order to keep as much data as possible in the different levels of the memory hierarchy. Again, while there has been a lot of work on compressing indexes for numerical data [39], almost no work exists for string data [13].

The rest of this chapter is structured as follows:

- Section 3.2 presents related work.

- Section 3.3 introduces a new approach for indexing a dictionary of string values (called *shared-leaves*) that leverages an order-preserving encoding scheme efficiently. In the shared leaves approach, indexes on different attributes (that can be clustered the same way) can share the same leaves in order to reduce the memory consumption while still providing efficient access paths.

- Section 3.4 introduces a concrete *leaf structure* for the shared-leaves approach that can be used by the indexes of a dictionary for efficiently encoding and decoding string values while the leaf structure itself is compressed. Furthermore, the most important operations on this leaf structure (i.e., lookup and update) are discussed and their costs are analyzed.

- Section 3.5 presents two new *cache-conscious string indexes* that can be used on top of the leaf structure to efficiently support the encoding of string data in the dictionary. For the decoding of integer codes the CSS-Tree [66] is used.

- Section 3.6 presents experiments that evaluate the new leaf structure and the new cache-conscious indexes under different types of workloads and shows a detailed analysis of their performance and their memory behavior. As one result, the experiments show that in terms of performance the leaf structure is as efficient as other read-optimized indexes while using less memory (due to compression).

- Section 3.7 concludes the chapter.

3.2 Related Work

There has been recent work on dictionary compression in column stores [3, 44, 84]. As mentioned before, that work focuses on small dictionaries for attributes with a stable domain size. Other work on dictionary compression of strings [7, 27] generates variable-length integer keys to support the order-preserving encoding of attributes with a variable domain size. However, none of that work addresses the issue of efficiently encoding a huge set of variable-length string values using fixed-length integer keys and how to efficiently support updates on such a dictionary without having to re-encode all existing data. Furthermore, to the best of our knowledge, there exists no work that exploits the idea of different indexes sharing the same leaves.

Moreover, there exists a lot of work on cache-conscious indexes and index compression [39, 42, 56, 66, 67]. However, not much work is focused on cache-conscious indexing of string values. Compared to the indexes presented in this thesis, [10] presents a cache-conscious trie that holds the string values in a hash table as its leaf structure and thus does not hold the strings in sort order. [13] presents a string index that holds the keys in sort order but is not designed to be cache-conscious.

Furthermore, the tradeoff between compression and query performance has been investigated, e.g., in [40]. [41] presents a compression technique for B-Trees designed to achieve a similar scan performance as a column store. A similar idea to the leaf structure presented in this chapter was implemented in IBM DB2 (LUW) [19].

An area that is also generally related to the work in this chapter is the topic of indexing in main memory databases [21, 38]. Finally, the ideas of [82] are applied to the indexes presented in this chapter for buffering index lookups to increase cache locality. Buffering can also help for bulk loading these indexes.

3.3 Designing Data structures for a Global Dictionary

In this section, we first discuss the operations that an order-preserving string-dictionary must support. Afterwards, we present a new idea for indexing such a dictionary (called *shared-leaves*) which is not bound to particular index structures and we show how the required operations can be implemented using this approach. Finally, we discuss requirements and design decisions for index structures that can be efficiently used for a dictionary together with the shared-leaves approach.

3.3.1 Dictionary Operations

As mentioned in Section 3.1, in this chapter a string dictionary is modeled as a table T with two attributes: $T = (value, code)$. Thus, table T defines a mapping of variable-length string values (defined by the attribute *value*) to fixed-length integer codes (defined by the attribute *code*) and vice versa. In order to support the data loading as well as the query processing task inside a column store, the interface of the dictionary should support the following two bulk operations for encoding and decoding string values:

encode: values → codes This bulk operation is used during data loading in order to encode data of a string column (i.e., the *values*) with corresponding integer codes (i.e., the *codes*). This operation involves 1) the lookup of codes for those strings that are already in the dictionary, and 2) the insertion of new string values as well as the generation of order-preserving codes for those new values. The generation of the codes will be discussed in Chapter 4. For ease of presentation, we will postpone the discussion of problems with the encoding to Chapter 4, i.e., for now we assume that the encoding scheme can handle any update.

decode: codes → values This bulk operation is used during query processing in order to decode (intermediate) query results (i.e., a bulk of integer *codes*) using the corresponding string values (i.e., the *values*).

Moreover, the interface of the dictionary should also support the following two operations in order to enable the rewrite of the query predicates (for query processing):

lookup: (value, type) → code This operation is used during query compilation in order to rewrite a string constant (i.e., the *value*) in an equality-predicate like `p_name = 'Whole Milk - Gallon'` or in a range-predicate like `p_name` $\geq$ `'Whole Milk - Gallon'` with the corresponding integer code (i.e., *code*). The parameter *type* specifies whether the dictionary should execute an exact-match lookup (as it is necessary for string constants in equality-predicates) or return the integer code for the next smaller (or larger) string value (as it is necessary for string constants in range-predicates). An example will be given in the following subsection.

lookup: prefix → (mincode, maxcode) This operation is used during query compilation to rewrite the $prefix$ of a prefix-predicate (e.g., `p_name like 'Whole Milk%'`) with the corresponding integer ranges (i.e., the *mincode* and the *maxcode*). Again, an example will be given in the following subsection.

In order to use table T to efficiently support all these operations, we propose to build indexes for both attributes of table T (*value* and *code*) because encoding and decoding usually access only a subset of the dictionary. Moreover, indexing the dictionary is especially important if the dictionary is shared between different attributes, because then it is more likely that only a subset of the dictionary is touched (if the domains of the individual attributes are not completely overlapping). Consequently, we believe that in many cases a sequential scan of the complete dictionary does not pay off. The choice of whether to use a sequential scan or an index to access the dictionary, however, has to be done as a part of the cost-based query optimization (because it strongly depends on the particular workload). However, this discussion is out of the scope of this thesis.

3.3.2 Shared-leaves Indexing

Traditional approaches for indexing can be classified into two general categories: *direct* and *indirect* indexes. Using these approaches for indexing the two attributes (*value* and *code*) of table T would result in the following situations (see Figure 3.4):

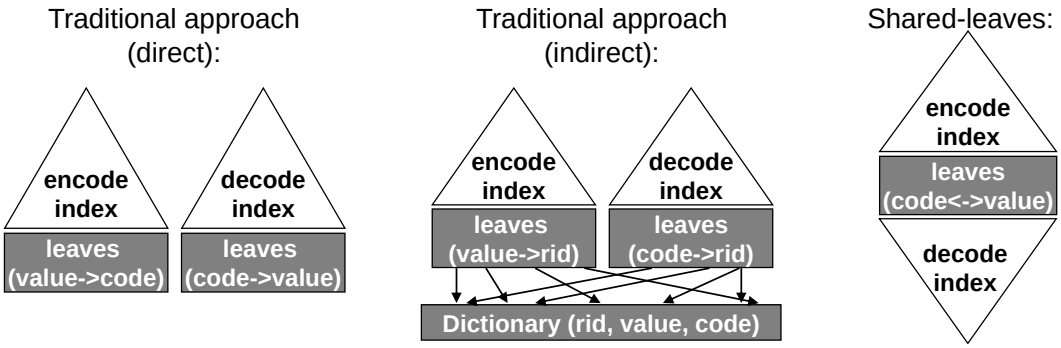


Figure 3.4: Traditional approaches vs. shared-leaves

(1) In the direct indexing approach, two indexes for encoding and decoding are created that hold the data of table T directly in their leaves. In this case, the table T itself does not need to be explicitly kept in main memory since the data of table T is stored in the indexes.

(2) In the indirect indexing approach, two indexes for encoding and decoding are created that hold only references to the data inside table T (i.e., a row identifier *rid*). In this case, the table T itself needs to be explicitly kept in main memory.

While direct indexing (1) has the disadvantage of holding the data of table T redundantly in the two indexes which is not optimal if the indexes should be main memory resident, indirect indexing (2) (which is standard in main-memory databases [21, 38]) requires one level of indirection more than the direct indexing approach (i.e., pointers into the table T). Thus, (2) results in higher cache miss rates on modern CPUs. Another alternative to index the dictionary data is to extend a standard index (e.g., a B^+-Tree) in order to support two key attributes instead of one (i.e., in our case for *value* and *code*). However, in that case both access paths of the index need to read the two key attributes during lookup which increases the cache miss rates (especially when decoding the integer *codes*).

The new idea of this thesis is that the two indexes for encoding and decoding share the same leaves (see *shared-leaves* approach in Figure 3.4) where both indexes directly hold the data of table T in their leaves but avoid the redundancy of the direct indexing approach. Thus, the shared-leaves also avoid the additional indirection level of the indirect indexing approach.

As the string dictionary uses an order-preserving encoding scheme, the string values and the integer codes in table T follow the same sort order (i.e., we can have clustered indexes on both columns and thus can share the leaves between two direct indexes). Consequently, as the attributes *value* and *code* of table T can both be kept in sort order inside the leaves, the leaves can provide efficient access paths for both lookup directions (i.e., for the encoding and decoding) using a standard search method for sorted data (e.g., binary search or interpolation search). Moreover, using the shared-leaves for indexing the dictionary implies that table T does not have to be kept explicitly in main memory because the leaves hold all the data of table T (as for direct indexes).

In the following, we discuss how the shared-leaves approach can support the bulk operations mentioned at the beginning of this section to enable data loading and query processing inside a column store:

Figure 3.5 shows an example of how the shared-leaves approach can be used to efficiently support the bulk operations for encoding and decoding string values. In order to encode a list of string values (e.g., the list shown at the top of Figure 3.5), the encode index is used to propagate these values to the corresponding leaves. Once the leaves are reached, a standard search algorithm can be used inside a leaf to lookup the integer code for each single string value. The decoding operation of a list of integer codes works similar. The only difference is that the decode index is used to propagate the integer codes down to the corresponding leaves (e.g., the list of integer codes shown on the bottom of Figure 3.5).

If some integer codes for string values are not found by the lookup operation on the encode index (e.g., the string values 'aac' and 'aad' in our example), these string values must be inserted into the dictionary (i.e., the shared-leaves) and new integer codes must be generated for those values (see the right side of Figure 3.5). The new codes for these string values have to be added to the result (i.e., the list of *codes*) that are returned by the encoding operation. Moreover, the encoding and decoding indexes must be updated if necessary.

In this chapter, the codes are generated by simply partitioning the code range where new string values are inserted in into equi-distant intervals (e.g., in our example the two strings 'aac' and 'aab' are inserted into the code range between 10 and 20). The limits of these intervals represent the new codes (e.g., 13 for 'aac' and 17 for 'aad' in our example). In case that the range is smaller than the number of new string values that have to be inserted in the dictionary, re-encoding of some string values as well as updating the data (i.e., the columns of a table) that use these string values becomes necessary. Chapter 4 presents a more sophisticated order-preserving encoding scheme that is optimal (i.e., that requires minimal re-encoding) under certain workloads.

In the following, we discuss how the shared-leaves approach can support the two lookup operations mentioned at the beginning of this section that support the predicate rewrite inside a column store:

The lookup operation that is necessary to rewrite the equality- and range-predicates is similar to the bulk lookup explained before: the encoding index propagates the string constant to the corresponding leaf and then a standard search algorithm can be used on the leaf to return the corresponding integer code. For example, in order to rewrite the predicate `value` $\geq$ `'zzc'` using the encode index in Figure 3.5 (left side), the encode index propagates the string value 'zzc' to the rightmost leaf and this leaf is used to lookup the next integer code for that string value that is equal or greater than the given value (i.e., the integer code 970 for the string value 'zzm'). The rewritten predicate thus would be `code` $\geq$ `970`.

In order to support the lookup operation that is necessary to rewrite a prefix-predicate, the encoding index needs to propagate the string prefix to those leaves which contain

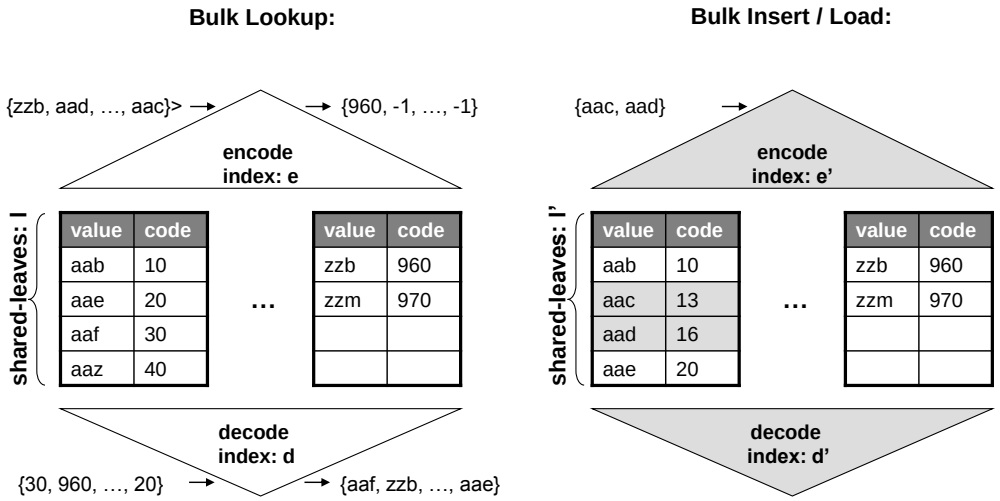


Figure 3.5: Operations on indexes with shared-leaves

the minimum and the maximum string value that matches this prefix. For example, in order to rewrite the predicate `value like 'aa%'` using the encode index in Figure 3.5 (left side), the encode index has to propagate the prefix to the first leaf which contains the minimum and the maximum string value that matches this prefix. Afterwards, those leaves are used to map the strings that represent the boundaries for the given prefix to the corresponding codes (e.g., in our example we retrieve the codes for 'aab' and 'aaz' and rewrite the predicate as $10 \leq$ `code` ≤ 40).

3.3.3 Requirements and Design Decisions

A dictionary should be fast for encoding (i.e., the bulk lookup / insert of integer codes for a list of string values) but the main optimization goal is the performance for decoding (i.e., the bulk lookup of string values for a given list of integer codes). Thus, the data structures of the dictionary (i.e., leaves and indexes) should also be optimized for encoding / decoding bulks instead of single values. Moreover, the data structures should be optimized for modern CPUs (i.e., they should be cache-conscious and the operations should be easy to parallelize). In the following we discuss further requirements and design decision for the leaf structure and the indexes of the dictionary.

Leaf Structure

The most important requirement for the leaf structure is that it must be able to hold the string values as well as the integer codes in sort order to enable efficient lookup operations (e.g., binary search) for both encoding and decoding (while the leaf structure should be optimized for decoding).

As the dictionary should be memory resident, the memory footprint of the dictionary must be small (i.e., it might make sense to apply a lightweight compression scheme

such as incremental encoding to the leaf data). Moreover, the leaf should support the encoding of variable-length string values. While the lookup operations on a leaf are trivial for fixed-length string-values that are not compressed, the lookup operations get more complex if the leaves should also support variable-length string values and compression.

When encoding a bulk of string values, new string values might be inserted into the dictionary which involves updating the shared-leaves. Consequently, the leaf should also enable efficient bulk loads and bulk updates.

Finally, note that the leaf structure can be totally different from the data structure that is used for the index nodes on top. A concrete leaf structure that satisfies these requirements is discussed in detail in the next section.

Encode / Decode index structure

Same as the leaf structure, the indexes for encoding and decoding should keep their keys in sort order to enable efficient lookup operations over the sorted leaves. Another requirement is that the encode index must also support the propagation not only of string constants but also of string-prefixes to the corresponding leaves in order to support the predicate-rewrite task. Moreover, the indexes should also be memory resident and thus must have a small memory footprint.

When bulk encoding a list of string values using the encoding index, in addition to the lookup of the integer codes for string values that are already a part of the dictionary, it might be necessary to insert new string values into the dictionary (i.e., update the leaves as well as the both indexes for encoding and decoding) and generate new order-preserving codes for those values. We propose to combine these two bulk operations (lookup and insert) into one operation. In order to support this, we see different strategies:

All-Bulked First, propagate the string values that need to be encoded to the corresponding leaves using the encode index and lookup the codes for those strings that are already in the leaves. Afterwards, insert the new values that were not found by the lookup into the leaves and if appropriate reorganize the updated leaf level (e.g., create a leaf level where all leaves are filled up to the maximal leaf size). Afterwards generate integer codes for the new string values and bulk load both a new encode and a new decode index from the updated leaf level (in a bottom-up way).

Hybrid First, propagate the string values that need to be encoded to the corresponding leaves using the encoding index and update the encoding index directly (i.e., do updates in-place during propagation). Then, lookup the codes for those strings that are already in the leaves. Afterwards, insert the new values that were not found by the lookup into the leaves and generate integer codes for all new string values. Finally, bulk load a new decode index from the updated leaf level (bottom-up).

All-In-Place First, propagate the string values that need to be encoded to the corresponding leaves using the encoding index and update the encoding index directly (i.e., do updates in-place during propagation). Then, lookup the codes for those strings that are already in the leaves. Afterwards, insert the new values that were not found by the lookup into the leaves and generate integer codes for all new string values. Propagate each update on the leaf level that causes an update of the decode index (e.g., a split of a leaf) directly to the decode index and apply the update.

In the first two strategies above, the decode index is bulk loaded from the updated leaf level. Therefore it should provide a better search performance for decoding than an index that is constantly updated and potentially degenerates. Decoding performance is our main optimization goal. Consequently, in this thesis we focus on the first two strategies.

In order to guarantee consistency of the data dictionary, for simplicity we decide to lock the complete indexes as well as the leaves during data loading (i.e., the encoding of string values). This is sufficient because data loading usually happens only at predefined points in time in data warehousing (i.e., once a day). Thus, no concurrent updates and reads are possible during data loading. However, during query processing (i.e., for decoding query results), we allow concurrency because these are read-only operations.

For persisting the dictionary, currently we only write the updates leaves sequentially to disk as a part of data loading. More sophisticated persistence strategies can be applied but are outside of the scope of this thesis.

3.4 Leaf Structure

In this section, we present a leaf structure that can be used in the shared-leaves approach (see Section 3.3) for efficiently encoding and decoding variable-length string values on a particular platform. The general idea of this leaf structure is to keep as much string values as well as the corresponding fixed-length integer codes sorted and compressed together in one chunk of memory in order to increase the cache locality during data loading and lookup operations.[2]

3.4.1 Memory Layout

Figure 3.6 shows an example of the memory layout of one concrete instance of a leaf structure that represents the dictionary shown in Figure 3.3. The leaf structure compresses the string values using incremental-encoding [78] while each n-th string (e.g., each 16-th string value) is not compressed to enable an efficient lookup of strings without having to decompress the complete leaf data: In the example, *value 16* 'Whole Milk - Gallon' is not compressed and *value 17* 'Whole Milk - Half Gallon' is compressed using

[2]In this chapter, we assume that string values are encoded in ASCII using one byte per character.

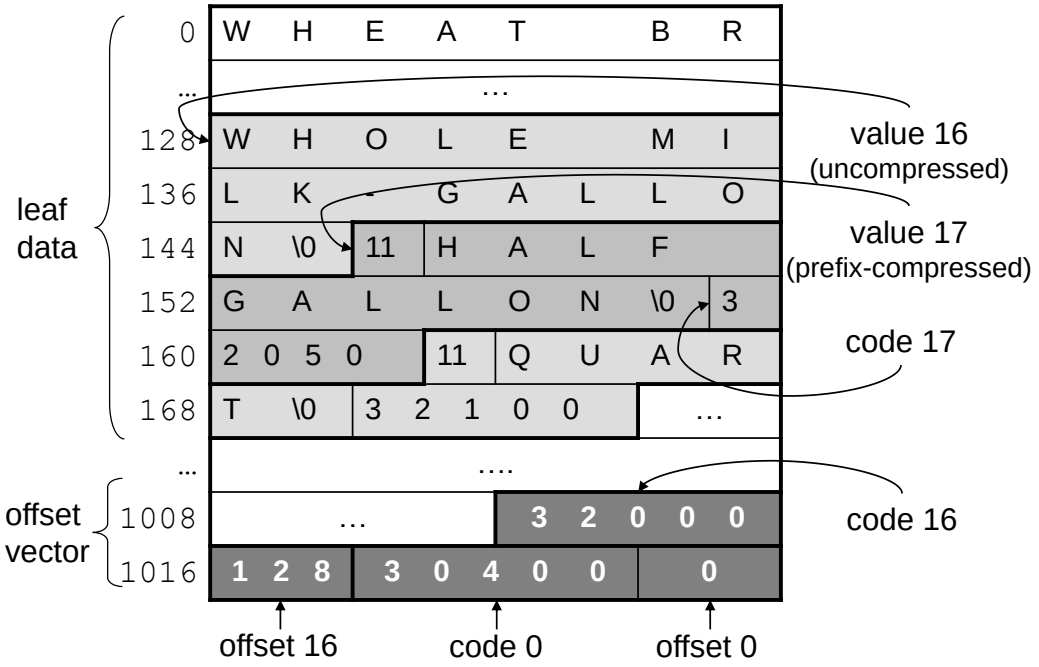


Figure 3.6: Leaf for variable-length string values

incremental-encoding; i.e., the length of the common prefix compared to the previous value (e.g., 11) is stored together with the suffix that is different (e.g., 'Half Gallon')

In order to enable an efficient lookup using this leaf structure, an offset vector is stored at the end of the leaf. This offset vector holds references (i.e., offsets) and the integer codes of all uncompressed strings of a leaf also in a sorted way. For example, the offset 128 and the code 32000 are stored in the offset vector for *value 16* (see Figure 3.6). The integer codes of the compressed string-values are stored together with the compressed string values in the data section and not in the offset vector (e.g., the code 32050 for *value 17*). The offset vector is stored in a reverse way to enable efficient data loading while having no negative effect on the lookup operations.

In order to adapt the leaf structure to a certain platform (e.g., to the cache sizes of a CPU) different parameters are available:

Leaf-size l defines the memory size in bytes that is initially allocated for a leaf. In our example, we used $l = 1024$.

Offset-size o defines the number of bytes that are used as offset inside the leaf. This parameter limits the maximal leaf size to 2^{8o}. If we use $o = 2$ bytes, as in the example in Figure 3.6, then the maximal leaf size is 2^{16} bytes (64 kB).

Code-size c defines the number of bytes that are used to represent an integer codeword. This limits the maximal number of different codewords to 2^{8c}. If we use for example $c = 4$, we can generate 2^{32} different codewords.

Prefix-size p defines the number of bytes that are used to encode the length of the common prefix for incremental encoding. This parameter is determined by the maximal string length as $p = (\log_2 max_str_len)/8$. For example, if we use strings with a maximal length of 255, then we can use $p = 1$ because using one byte allows us to encode a common prefix length from 0 to 255.

Decode-interval d defines the interval that is used to store uncompressed strings (16 in our example). This parameter has influence on the size of the offset vector (i.e., the smaller this interval is, the more space the offset vector will use).

We do not allow the definition of these parameters on the granularity of bits because most CPUs have a better performance on data that is byte-aligned (or even word- or double-word aligned on modern CPUs). We decide to use byte-aligned data (and not word- or double-word aligned data) because the other variants might lead to a dramatic increase in the memory consumption of the leaves.

As an example, assume that we want to tune the leaf structure for a platform using one CPU (with one core) having an L2 cache of 3 MB and an L1 cache of 32 kB. Moreover, assume that the leaf should only store strings with an average length of 50 characters and a maximum length of 255 characters (which implies that $p = 1$ can be used). In that case it would make sense to use a leaf size not bigger than the L2 cache size, say at most 2 MB. Consequently, an offset-size $o = 3$ is sufficient to address all values inside such a leaf. The code-size depends only on the overall number of strings that should be stored in a dictionary. We assume that $c = 4$ is an appropriate choice for this example.

Defining the value for the decode interval d is more difficult: in general, we want to make sure that the offset vector once loaded remains in the L1 cache (e.g., having a maximum size of 32 kB) such that an efficient binary search is possible for a bulk of strings. With the given settings of the example above, we assume that a leaf will be able to store approximately 42000 strings (which results from the maximum leaf size of 2 MB and the average string length of 50). Moreover, each uncompressed string utilizes 7 bytes of the offset vector (for the offset and the code). Consequently, the offset vector can store max. $32k/7 \approx 4681$ entries (i.e., offsets and codes) for uncompressed strings which implies that we could store each $d \approx 42000/4681 \approx 9$th string uncompressed in the example.

3.4.2 Leaf Operations

The most important operations on the leaf structure are the lookup operations for encoding string values with their integer codes and decoding the integer codes with their string values. Moreover, the leaf also supports updates (i.e., inserting new strings). In the following paragraphs, we examine these operations in detail.

Bulk Lookup

In this paragraph, we first explain how the lookup works for a single value and then discuss some optimizations for bulks. The leaf supports the following lookup operations:

one to lookup the code for a given string value (i.e., value $v \rightarrow$ code c) and another one that supports the lookup vice versa (i.e., code $c \rightarrow$ value v). In order to search the code c for a given value v, the procedure is as follows:

(1) Use the offset vector to execute a binary search over the uncompressed strings of a leaf in order to find an uncompressed string value v' that satisfies $v' \leq v$ and no other uncompressed value $\bar{v}$ exists with $v' < \bar{v} < v$.

(2) If $v' = v$ return the corresponding code c for v that is stored in the offset vector.

(3) Otherwise, sequentially search the leaf for value v starting from value v' (using the offset from the offset vector to directly get to that starting position) until v is found or the next uncompressed value appears. In the first case return the code c, in the second case indicate that the value v was not found.

Note that for sequentially searching over the incrementally encoded string values no decompression of the leaf data is necessary. Algorithm 3.1 shows a search function that enables the sequential search over the compressed leaf data. The parameters of this function are: the leaf data (i.e., `leaf`), the offset where to start and end the sequential search (i.e., `start` and `end`), as well as the value that we search for (i.e., `v`). The return value is the corresponding code `c` if the value is found, otherwise the algorithm returns `-1`. The basic idea of the algorithm is that it keeps track of the common prefix length (i.e., `prefix_len`) of the current string (at the offset `start`) and the search string `v`. If this common prefix length is the same as the length of the search string then the correct value is found and the code can be returned. The variables `p` and `c` are constants that represent the prefix-size and the code-size to increment the offset value `start`.

The lookup operation to find a string value `v` for a given code `c` works similar as the lookup operation mentioned before using the offset vector. The differences are that the first step (i.e., the search over the offset vector) can be executed without jumping into the leaf data section because the codes of the uncompressed strings are stored together with the offset vector. In contrast to the other lookup operation, for the search over the offset vector we theoretically expect to get only one L1 cache miss using a simplified cache model (if the offset vector fits into the L1 cache). Another difference of this lookup operation is that during the sequential search the string values have to be incrementally decompressed.

When executing both lookup operations, we expect to theoretically get one L2 cache miss in a simplified cache model if we assume that loading the complete leaf causes one miss and the leaf fits into the L2 cache.[3] The average costs for these two lookup operations are as follows (where n is the number of strings / codes stored in a leaf and d is the decode interval):

$$O(log(n/d)) + O(d)$$

[3] A more fine grained cache model would respect cache lines and cache associativity. However, as we focus on bulk lookups our simplified model is sufficient to estimate the costs of cache misses.

Listing 3.1: Sequential search of string v on compressed leaf

```
1  int SequentialSearch(leaf, start, end, v) {
     v_prime = leaf[start];
     // read string v_prime at offset start
     start = start + size(v_prime);
     // increment offset by string size
6    prefix_len =  prefix_len(v,v_prime);
     // calculate common prefix len

     while ( start <= end && prefix_len < size(v)) {
       curr_prefix_len = leaf[start];
11     // get current prefix len
       start = start + p;
       // increment offset by prefix-size (p=1)
       v_prime = leaf[start];
       start = start + size(v_prime)
16     if (curr_prefix_len != prefix_len) {
         continue;
         // prefix of current value v_prime is too short/long
       } else if (compare(v_prime, v) > 0) {
         return -1;
21       // current value v_prime comes after search value v
       }

       prefix_len = prefix_len + prefix_len(v,v_prime);
       start = start + c
26     // increment offset by code-size (c=4)
     }

     if (prefix_len = size(v)) {
       return leaf[start-c];
31     // string v found: return code
     } else {
       return -1;
       // string v not found
     }
36 }
```

In order to optimize the lookup operation for bulks (i.e., a list of string values or a list of integer codes), we can sort the lookup probe. By doing this, we can avoid some search overhead by minimizing the search space after each search of a single lookup probe (i.e., we do not have to look at that part of the offset vector / leaf data that we already analyzed).

Bulk Update

In this paragraph, we explain how to insert new strings into the leaf structure. As we assume that data is loaded in bulks, we explain the initial bulk load and the bulk insert of strings into an existing leaf.

In order to initially bulk load a leaf with a list of string values, we first have to sort the string values. Afterwards, the leaf data can be written sequentially from the beginning of the leaf while the offset vector is written reversely from the end. If the string values do not utilize the complete memory allocated for a leaf (because we do not analyze the compression rate before bulk loading) then the offset vector can be moved to the end of the compressed data section and the unused memory can be released.

In order to insert a list of new string values into an existing leaf, we again have to sort these string values first. Afterwards, we can do a sort merge of these new string values and the existing leaf in order to create a new leaf. The sort merge is cheaper if we can reuse as much of the compressed data of the existing leaf as possible and thus do not have to decode and compress the leaf data again. Ideally, the new string values start after the last value of the existing leaf. In that case, we only have to compress the new string values without decoding the leaf. If the list of string values and the existing leaf data do not fit into one leaf anymore, the data has to be split. However, as the split strategy depends on the index structure that we build on top of the leaves, we discuss this in the next section.

3.5 Cache-conscious Indexes

In this section, we present new cache-conscious index structures that can be used on top of the leaf structure presented in the previous section. These indexes support one of the first two update strategies (*All-Bulked* and *Hybrid*) discussed in Section 3.3. For the encoding index, we present a new cache sensitive version of the patricia trie (called *CS-Array-Trie*) that supports the *Hybrid* update strategy and a cache sensitive version of the Prefix-B-Tree [13] (called *CS-Prefix-Tree*) that supports the *All-Bulked* update strategy.

As decoding index, we reuse the CSS-Tree [66] which is known to be optimized for read-only workloads. We create a CSS-Tree over the leaves of the dictionary using the minimal integer codes of each leaf as keys of the index (i.e., the CSS-Tree is only used to propagate the integer values that are to be decoded to the corresponding leaves). As the CSS-Tree can be bulk loaded efficiently in a bottom-up way using the leaves of the dictionary, it satisfies the requirements for both update strategies (*Hybrid* and

All-Bulked).

3.5.1 CS-Array-Trie

As a first cache-conscious index structure that can be used as an encode index to propagate string lookup probes and updates to the corresponding leaf in a shared-leaf approach, we present the *CS-Array-Trie*. Compared to existing trie implementations the CS-Array-Trie uses read-optimized cache-conscious data structures for the index nodes and does not decompose the strings completely.

General Idea

Many existing trie implementations are using nodes that hold an array of pointers to the nodes of the next level with the size of the alphabet (e.g., 128 for ASCII). While such an implementation allows efficient updates and lookups on each node, it is not memory-efficient because the array trie allocates space for a pointers per node, where a is the size of the alphabet (while a pointer uses 8 bytes on a 64 bit system).

Other trie implementations avoid the memory overhead and use a sorted linked list [51] to hold only the characters of the indexed strings together with a pointer to the next level of the trie. While this implementation still offers efficient node updates, the lookup must execute a search over the characters stored in the linked list. However, linked lists are known to be not very cache efficient on modern CPUs because of pointer-chasing [77]. Moreover, most existing trie implementations decompose the indexed strings completely (i.e., each letter of a string results in a node of the trie).

Compared to the implementations mentioned above, a node of the *CS-Array-Trie* uses an array instead of a linked list to store the characters of the indexed string values. Compared to a linked list an array is not efficient when sequentially inserting single values into a trie. However, when bulk inserting new values into a CS-Array-Trie, we need increase the size of the array of a node only once for each bulk. In order to lookup a string value of a CS-Array-Trie, the search over the characters of a node (i.e., the array) is more efficient than the sequential search on a linked list because the array supports binary search and all characters are stored clustered in memory.

The second key idea of the *CS-Array-Trie* is that it does not decompose the string values completely. The *CS-Array-Trie* stores a set of strings that have the same prefix together using the leaf structure that we discussed in the previous section. A leaf of the *CS-Array-Trie* stores the complete strings and not only their suffixes (without the common prefix) in order to enable efficient decoding of integer codes using the same leaves (as described in our shared-leaves approach). Moreover, storing the complete strings in a leaf (i.e., repeating the same prefix) is still space efficient because we use incremental encoding to compress the strings. Finally, for those trie nodes that only hold a pointer to one child node, we use the path compression used in patricia tries [59].

Figure 3.7 (left side) shows an example of a CS-Array-Trie that indexes nine different strings. If we follow the path 'aa' in the trie, we reach a leaf that holds only strings that

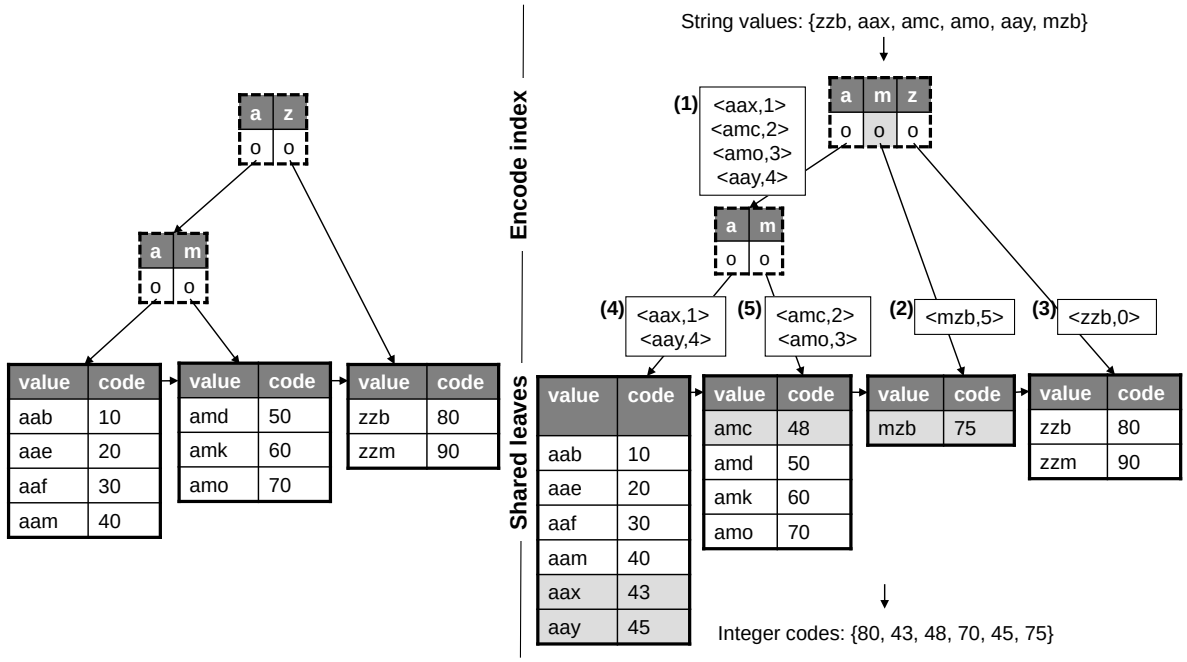


Figure 3.7: Encode data using the CS-Array-Trie

start with the prefix 'aa'. The leaves are shown as uncompressed tables for readability. The physical memory layout is as discussed in Section 3.4.

Index Operations

In order to encode a bulk of string values during data loading using that trie, we implement the *Hybrid* update strategy for the CS-Array-Trie, i.e., new string values are inserted into the trie while the strings that should be encoded are propagated through the encoding index (i.e., the trie). In order to leverage the fact of having bulk inserts, we propagate the string values (in pre-order) to the leaves using variable buffers at each node of the trie to increase the cache locality during lookup as described in [82]. Moreover, when using buffers at each node we can increase the size of the array of characters stored inside a node only once per bulk. The work in [82] showed that this effectively reduces data cache misses for tree-based indexes and results in a better overall lookup performance.

Figure 3.7 (right side) shows an example for encoding a bulk of six strings using the existing encode index (i.e., the trie shown on the left side). First, all strings that need to be encoded are propagated from the root node of the trie to the first level of nodes creating three buffers ((1), (2), and (3)). In order to keep the order of strings in the lookup probe for creating the encoded result (at the bottom of Figure 3.7), a sequence number is added for each value in the buffers (as suggested in [82]). The strings that are propagated from the root to the first level of buffers are analyzed and the missing

character 'm' for the string 'mzb' is added to the root node.[4]

Afterwards, the buffer (1) is propagated to the next level creating two new buffers (4) and (5) and buffer (1) is returned to a buffer pool (to avoid expensive memory allocation for buffer pages). Next, buffers (4) and (5) are processed, i.e., values for the codes for existing string values are looked up and new strings are inserted into the existing leaves with a placeholder as their integer code (e.g., -1): While buffer (4) contains two new strings 'aax' and 'aay' that are inserted to the leftmost leaf, buffer (5) contains only one new string 'amc' that is inserted the next leaf. Note, that the new values in buffers (4) and (5) are not yet deleted and kept to lookup the integer codes for those string values (that are not yet generated).

A question that arises here is, whether the new strings fit into the existing leaf (i.e., the new leaf size is expected to be less than the maximal leaf size) or whether the leaf must be split up into several leaves. In order to estimate the expected leaf size, we add the size (uncompressed) of all new strings in a buffer page as well as the size of their new codes (without eliminating duplicates) to the current leaf size. If the bulk is heavily skewed and contains many duplicates it is likely that we decide to split a leaf even if it is not necessary.

When all string values are propagated to their corresponding leaves (i.e., the new strings are inserted into the leaves), new integer codes are generated for the new string values. This is done by analyzing the number of strings that are inserted in between two existing string values. For example, in order to generate codes for the three new string values that are inserted into the first two leaves between 'aam' and 'amd' in Figure 3.7 (right side), we have to generate three new codes that must fit into the range between 40 and 50. Using our equi-distance approach (discussed in Section 3.3), we generate 43, 45, and 48 as new codes. Therefore, the trie must allow us to sequentially analyze all leaves in sort order (i.e., each leaf has a pointer to the next leaf). Finally, after generating the integer codes for the new string values of the trie, we use the buffer pages that we kept on the leaf level (e.g., the new strings in the buffers (2), (4), and (5) in the example) to lookup the integer codes for the new string values and use the sequence number to put the integer code at the right position in the answer (see bottom of Figure 3.7).

Another task that can efficiently be supported using the CS-Array-Trie, is the predicate rewrite: for equality- and range-predicates the constants are simply propagated through the trie without buffering. For prefix-predicates, the prefix is used to find the minimal and maximal string value that matches this prefix (which is also trivial when using a trie).

Cost Analysis

For propagating a bulk of string values from the root of a CS-Array-Trie to the leaves, we theoretically expect to get one L2 data cache miss for each node in our simplified cache model. In addition, for every leaf that has to be processed during the lookup, we expect to get another L2 data cache miss using our simplified cache model.

[4] All updates on the trie and the leaves are shown in light gray in Figure 3.7 (right side).

Moreover, the input and output buffers of a node should be designed to fit into the L2 cache together with one node (to allow efficient copying from input to output buffers). In that case, we expect to get one cache miss for each buffer page that needs to be loaded. Moreover, generating the new integer codes will cause one L2 data cache miss for each leaf of the trie. Finally, executing the lookup of the new integer codes, will also cause one cache miss for each buffer page that has to be loaded plus the L2 cache misses for executing the lookup operations on the leaves (as discussed in the section before).

The costs for encoding a bulk that contains no new string values (i.e., a pure lookup) are composed of the costs for propagating the strings through the trie plus the costs for the lookup of the codes for all strings using the leaves. If we assume that all strings in the lookup probe have the same length m and are distributed uniformly, the height of the trie is $log_a(s/l)$ in the best case and equal to the length of the string m in the worst case (where s is the total number of strings, l is the number of strings that fit in one leaf, and a the size of the alphabet). The lookup costs on each node are $O(log(a))$. Thus, the average costs for propagation are:

$$O(s * ((log_a(s/l) + m)/2) * log(a))$$

Parallelization

The propagation of string values from the root of the trie to the leaves (including the update of the nodes) can be easily parallelized for different sub-tries because sub-tries share no data.

Moreover, the generation of new integer codes can be done in parallel as well (without locking any data structures). For this we need to find out which leaves hold contiguous new string values (i.e., sometimes a contiguous list of new string values might span more than one leaf as shown in Figure 3.7 on the right side for the first two leaves).

Finally, the lookup operation of the new string values can also be parallelized without locking any data structures. In Figure 3.7 (right side), the new values in the buffer pages (2), (4), and (5) can be processed by individual threads for example. We can see that each thread writes the resulting integer codes to different index positions of the result (shown at the bottom) using the sequence number of the buffer pages. Thus, no locks on the result vector are necessary.

3.5.2 CS-Prefix-Tree

As a second cache-conscious index structure that can be used as an encode index to propagate string lookup probes and updates to the corresponding leaf in a shared-leaf approach, we present the *CS-Prefix-Tree*. This index structure combines ideas from the Prefix-B-Tree [13] and the CSS-Tree [66].

General Idea

Same as the Prefix-B-Tree, a node of a CS-Prefix-Tree contains the shortest prefixes that enable the propagation of string values to the corresponding child nodes. However, instead of storing a pointer to each child, the *CS-Prefix-Tree* uses a contiguous block of memory for all nodes and offsets to navigate through this block (as the CSS-Tree). This effectively reduces memory consumption and avoids negative effects on the performance due to pointer chasing. In order to further reduce the memory footprint of the CS-Prefix-Tree, we only store the offset to the first child node explicitly. Since the nodes have a fixed size s, we can calculate the offset to a child node using offset arithmetics (i.e., the i-th child of a node can be found at offset $o = offset(first\_child) + ((i-1) * s)$).

In order to enable fast search over the variable-length keys of a node (e.g., binary search), we store the offsets to the keys (i.e., the string prefixes) in an offset vector at the beginning of each node. Moreover, the node size has to be fixed in order to use offset arithmetics for computing the index to the child nodes. Thus, the number of children of a node is variable because we store variable-length keys inside a node.

Index Operations

The CS-Prefix-Tree (as the CSS-Tree) can only be bulk loaded in a bottom-up way and is therefore only suitable for the *All-bulked* update strategy discussed in Section 3.3. Using the *All-bulked* update strategy for the encoding a list of string values implies that the new string values (that are not yet a part of the dictionary) must first be inserted into the leaves (in sort order) and then the index can be created.

Thus, if the first bulk of string values should be encoded using the *All-bulked* update strategy, the complete leaf level has to be built using these string values. Therefore, we reuse the idea in [72] and create a trie (more precisely a CS-Array-Trie) to partition the string values into buckets that can be sorted efficiently using multi-key quicksort [14]. The sorted string values can then be used to create leaves that are filled up to the maximum leaf size. From these leaves, a new encode index (i.e., a CS-Prefix-Tree) is bulk loaded in a bottom-up way.

Figure 3.8 shows an example of a CS-Prefix-Tree. In the following, we describe the bulk load procedure of a CS-Prefix-Tree from a given leaf level:

(1) In order to bulk load the CS-Prefix-Tree, we process the leaf level in sort order: We start with the first two leaves and calculate the shortest prefix to distinguish the largest value of the first leaf and the smallest value of the second leaf. In our example it is sufficient to store the prefix ‘am’ in order to distinguish the first two leaves. This prefix is stored in a node of the CS-Prefix-Tree. Note, a node does not store a pointer to each child since we can derive an offset into the leaf level (i.e., the sorted list of leaves). Since we do not know the size of the offset vector in advance, we write the offset vector from left to right and store the keys from right to left. The example assumes a fixed node size of 32 bytes and therefore we store offset 29 in the offset vector and write the prefix at the corresponding position.

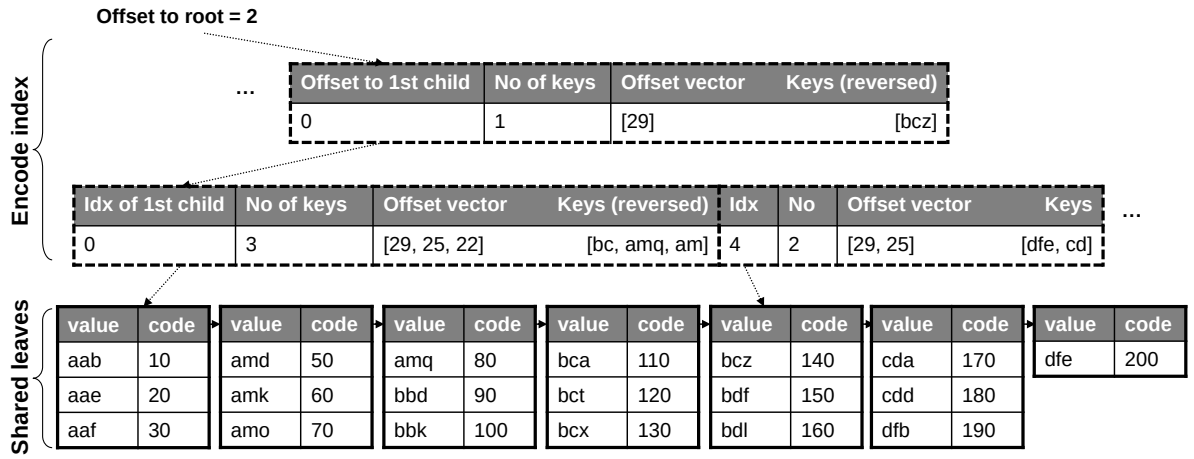


Figure 3.8: Example of a CS-Prefix-Tree

(2) Next, we calculate the shortest prefix to distinguish the largest value of the second leaf and the smallest value of the third leaf and so on until all leaves are processed. If a node is full, we start a new node and store the index to the first leaf that will be a child of this new node as an anchor. In our example the first node covers the first four leaves and therefore the index of the first child in the second node is 4. Note that the nodes are stored continuously in memory.

(3) As long as more than one node is created for a certain level of the CS-Prefix-Tree, we add another level on top with nodes that store prefixes that distinguish their child nodes. In our example we have one node on top of the lowest level of the CS-Prefix-Tree. This node is the root of the tree and we store the offset to this node in the contiguous memory block (2 in our example). Since the tree is built bottom up, the nodes have to be stored in that sequence in memory.

For subsequent bulks, we use the existing CS-Prefix-Tree to propagate the string values that are to be encoded to the corresponding leaves. In our current implementation we buffer the string values only on the leaf level (and not within the CS-Prefix-Tree). Afterwards, we do a sort-merge of the existing leaf with the new string values stored in the buffers. If the new string values in the buffers and the values of the existing leaf do not fit into one leaf, we simply create another leaf. This very simple sort-merge strategy can be improved in different ways: for example, one could try to create equally sized leaves to avoid degenerated leaves that contain only a small number of strings. Moreover, after encoding several bulks it might make sense to reorganize the complete leaf level by filling all leaves to their maximum. Once all updates are processed we bulk load a new CS-Prefix-Tree from the merged leaf level.

For example, in order to propagate a bulk of strings to the leaves using the CS-Prefix-Tree, we proceed as follows: assume we are looking for value 'amk' in our example in Figure 3.8. We start at the root node and see that 'amk' is smaller than 'bcz', thus we

proceed at the first child of this node at offset 0. We find that 'amk' is between 'am' and 'amq' and thus proceed at the second child of this node at index 1 (calculated from the index of the first child). The CS-Prefix-Tree stores the information that nodes below a certain offset point to leaves instead of internal nodes.

To rewrite query predicates using the CS-Prefix-Tree, we do a simple lookup with the string constants if it is an equality-predicate or a range-predicate. If it is a prefix-predicate, the prefix is used to find the minimal string value that matches the prefix. The lookup for the prefix will end up at that leaf which contains this value even if the value itself is not in the dictionary. From that leaf on, we execute a sequential search for the maximum string value that matches the prefix. We could save some effort compared to a sequential search, if we also use the index to find the leaf that holds the maximum value directly (similar to a skip list).

One issue with using a contiguous block of memory and offsets is that the memory has to be allocated in advance. We calculate the maximum amount of memory that all nodes of the CS-Prefix-Tree need by introducing an artificial limit on the maximum length of the keys in the tree. We then can calculate the minimum number of keys that fit into one node and thus can estimate the maximal number of nodes that we need to store the data. One possibility to overcome this issue is to leverage the idea of a CSB-Tree [67] to use a mix of pointers and offset arithmetics (e.g., one pointer per node) to identify the correct child and thus allow multiple blocks of memory instead of one single block.

Cost Analysis

We suggest to set the size of a node of a CS-Prefix-Tree at most to the size of the L2 cache. Thus for propagating a bulk of string values from the root of a CS-Prefix-Tree to the leaves, we theoretically expect to get one L2 data cache miss (in our simplified cache model) for each node that is traversed during the lookup for each string value and another L2 data cache miss when the value is written to the corresponding buffer of the leaf (which should also be designed to fit into the cache). Moreover, the generation of new codes and the encoding itself will each cause one L2 data cache miss for each leaf in the leaf level (if the leaf and the output buffer fit in the L2 cache together).

The costs for encoding a bulk that contains no new string values (i.e., a pure lookup) is composed of the costs for propagating the strings through the tree plus the costs of the lookup in the leaf. If we assume that all strings in the lookup probe have the same length and are distributed uniformly, the height of the tree is $log_k(s/l)$ (where s is the total number of strings, l is the number of strings that fit in one leaf, and k the number of keys that fit into one node). The lookup costs on each node are $O(log(k))$. Thus, the average costs for propagation are:

$$O(s * log_k(s/l) * log(k))$$

Compared to the CS-Array-Trie, it is more expensive to build a CS-Prefix-Tree because the data has to be sorted first and then the CS-Prefix-Tree is loaded bottom-up as opposed to the CS-Array-Trie that is loaded top-down and implicitly sorts the data during that process. We show in our experiments that the CS-Prefix-Tree performs

slightly better than the CS-Array-Trie for a pure lookup workloads (i.e., encoding a bulk of strings that does not contain new values) since on average the tree is expected to be less high than the trie and the leaves are organized more compact.

Parallelization

Our current implementation supports multiple threads at the leaf level. Once the bulk is completely buffered at the leaves, the lookup can be executed in parallel as described for the CS-Array-Trie. Since we want to support variable length keys we cannot parallelize the bottom-up bulk loading of the tree.

Currently, the bulk lookup on the index is single threaded since we are not buffering the requests within the tree and thus have no point to easily distribute the workload to multiple threads. One way to parallelize the lookup would be to partition the bulk before accessing the index and process these partitions using multiple threads. Since the buffers at the leaf level would then be shared by several threads, either locking would be needed or the partitioning has to be done according to the sort order, which is expensive.

3.6 Evaluation

This section shows the results of our performance experiments with the prototype of our string-dictionary (i.e., the leaf structure discussed in Section 3.4 and the cache conscious indexes discussed in Section 3.5). We executed three experiments: the first experiment (Subsection 3.6.1) shows the efficiency of our leaf structure and compares it to other read-optimized indexes[5], the second experiment (Subsection 3.6.2) examines the two new cache conscious indexes using different workloads, and the third experiment (Subsection 3.6.3) shows the overall performance and scalability of our approach.

We implemented the data structures in C++ and optimized them for a 64 bit Suse Linux Server (kernel 2.6.32) with two Intel Xeon 5450 CPUs (each having four cores) and 16 GB of main memory. Each CPU has two L2 caches of 6 MB (where two cores share one L2 cache) and one L1 cache for each core with 32 kB for data as well as 32 kB for instructions.

In order to generate meaningful workloads, we implemented a string data generator that allows us to tune different parameters like the number of strings, string length, alphabet size, the distribution, and others. We did not use the TPC-H data generator *dbgen*, for example, because most string attributes either follow a certain pattern (e.g., the customer name is composed of the prefix 'Customer' and a unique number) or the domain size of such attributes is too low (e.g., the name of a country). Thus the data generated by *dbgen* does not have any interesting properties that could be analyzed. We will show the properties of the workloads that we generated for each experiment individually.

[5]We used PAPI to measure the performance counters: `http://icl.cs.utk.edu/papi/`.

Parameter	Value
Leaf-size l	64kB - 16MB
Offset-size o	4 bytes
Code-size c	8 bytes
Prefix-size p	1 byte
Decode-interval d	16
String-length	(1) 25, (2) 100
String-number	(1) $\sim$ 450000, (2) $\sim$ 150000
Alphabet-size	128
Distribution	Distinct (unsorted)

Table 3.1: Parameters for the leaf and workload generation

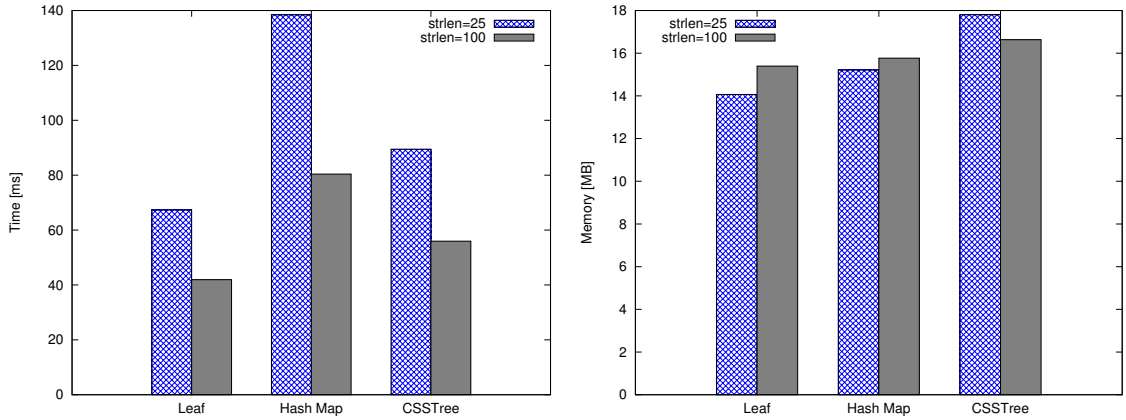
3.6.1 Efficiency of Leaf Structure

In this experiment, we analyze the efficiency of the leaf structure discussed in Section 3.4. The general idea of this experiment is to show the performance and memory consumption of the leaf operations for two different workloads. We used the parameters in Table 3.1 to configure the leaf structure (first part) and to generate our workloads (second part).

The two different workloads ((1) and (2) in Table 3.1) were designed that each of these workloads fits into a leaf with a size of 16 MB while each workload uses a different fixed string-length and thus represents a different number of strings. Both workloads result in roughly the same size in memory if uncompressed. We only used distinct unsorted workloads (i.e., no skew) because these workloads represent the worst case for all lookup operations (i.e., each string value / integer code of the leaf is encoded / decoded once for each workload). Since the leaves are bulkloaded, insert skew is not analyzed in this section. We used each of these workloads to load a set of leaves that hold the workload in a sorted way while for each workload we used different leaf sizes varying from 64 kB to 16 MB (resulting in a different set of leaves for each combination of workload and leaf size). The focus of this section is to evaluate the idea of *shared leaves* which themselves are sophisticated datastructures and can be seen as small indexes (e.g., the offset vector in the leaf) within the index. Thus, the experiments work with rather large leaf sizes in order to evaluate the idea.

The first goal of this experiment is to show the costs of the bulk loading and bulk lookup operations caused by the different workloads without the overhead of an encoding and decoding index on top. The experiment is conducted by simulating pure lookups on the leaves. We measured the time as well as the L2 cache misses that resulted from executing the bulk loading of the leaves and executing the lookup operations for encoding as well as decoding.

In order to load the leaves, we first sorted the workloads and then bulk loaded each leaf up to its maximal size (see Section 3.4). Afterwards, we generated the integer codes for these leaves. As shown in the table before, we use 8 bytes for the integer code



(a) Time (b) Memory Consumption

Figure 3.9: Bulk loading the data structures

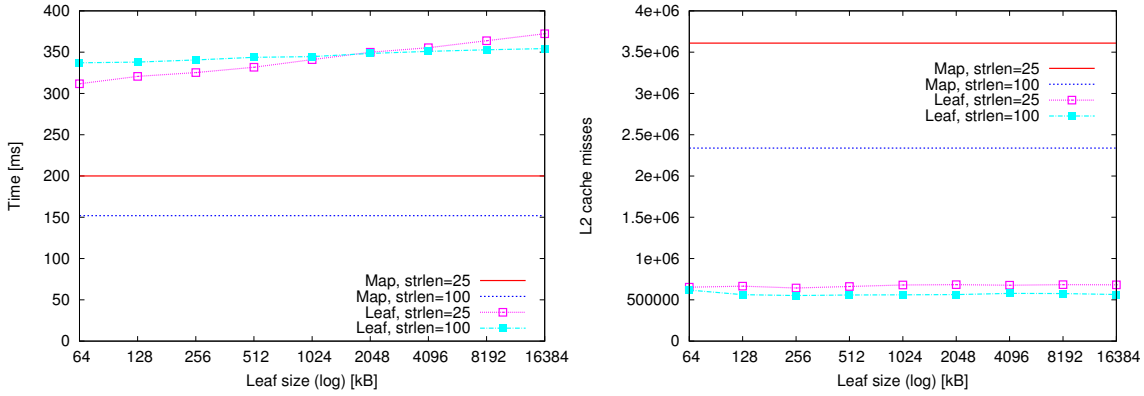
in this experiment to show the memory consumption expected for encoding attributes with a large domain size. In order to measure the pure lookup performance of the leaf structures, we assigned each string value of the workloads mentioned above to the corresponding leaf using a buffer and then we looked up the code for each string in the individual buffers (i.e., we created a corresponding encoded workload for each leaf). Finally, we used the encoded workload to execute the lookup operation (in the same way as described before) on the leaves to decode the integer codes again.

A second goal of this experiment is to show a comparison of the 16 MB leaf structure (which can be used for encoding as well as for decoding) and two cache-conscious read-optimized index structures using the workloads (1) and (2): for encoding the string values we compare the leaf structure to the compact-chain hash table [11] and for decoding integer codes we compare the leaf structure to the CSS-tree [66].

The main results of this experiment are that (1) the leaf structure is optimal when having a medium size ($\sim$ 512 kB) and (2) the performance of our leaf structure is comparable to the read-optimized index structures mentioned above while using less memory:

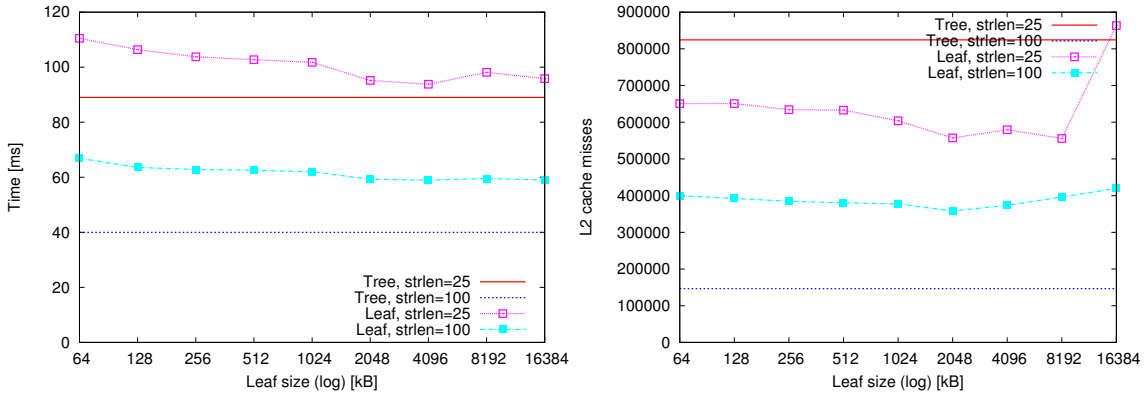
- Figure 3.9 shows the time and memory that is needed to bulk load the leaf structure (of 16 MB size) compared to bulk loading the compact-chain hash table and the CSS-tree. As a result, we can see that bulk loading the leaf structure is faster for both workloads (i.e., string length 25 and 100) and uses less memory compared to the compact-chain hash table and the CSS-tree.

- Figure 3.10 shows that the time and the L2 data cache misses for encoding the two workloads of this experiment (using the bulk loaded leaves) are increasing for large leaf sizes. Moreover, in terms of performance we can see that the 16 MB leaf structure is not as good as the compact-chain hash map (Map) but this overhead is tolerable since the leaf offers sorted access and compresses the data.



(a) Time (b) L2 cache misses

Figure 3.10: Encoding data



(a) Time (b) L2 cache misses

Figure 3.11: Decoding data

- Figure 3.11 shows that our leaf structure is optimized for decoding: While decoding $450k$ strings of length 100 using the smallest leaf size takes about 65 ms, it takes 340 ms to encode them. Moreover, there are significantly more L2 cache misses during encoding than during decoding these strings. Finally, compared to the CSS-tree (Tree) our 16 MB leaf structure is only a little slower when used for decoding.

3.6.2 Efficiency of Encoding Indexes

This experiment shows the efficiency of our new encoding indexes compared to an implementation of the list-trie that decomposes the strings completely (i.e., it does not use our leaf structure). The idea of this experiment is to show the costs for encoding workloads that produce different update patterns on the indexes (which cause different

costs). Again, the focus is on the datastructures, issues with the encoding scheme (e.g., because of clustered inserts) are discussed in Chapter 4.

The experiment works as follows: we first bulk load a dictionary with $10M$ strings and afterwards encode another bulk of $10M$ strings that represents a certain update pattern (details later). All workloads consist of string values with a fixed length of 20 characters (while the other parameters to generate the workload are the same as in the experiment before). For the leaves we fixed the maximum leaf size to be 512 kB (while the other leaf parameters are the same as in the experiment before). As update patterns we used:

No-updates The second bulk contains no new strings.

Interleaved 10% The second bulk contains 10% new string values where each 10th string in sort order is new.

Interleaved 50% The second bulk contains 50% new string values where every other string in sort order is new.

Interleaved 100% The second bulk contains 100% new string values whereas each new string is inserted between two string values of the first bulk.

Append The second bulk contains 100% new string values whereas all string values are inserted after the last string of the first bulk.

The interleaved patterns cause higher costs (independent from the index on top) because all leaves must be decompressed and compressed again to apply the inserts of the new strings. Figure 3.12 shows the time to first bulk load the dictionary with $10M$ strings and afterwards encode the other $10M$ strings for the different update patterns (when we either use the CS-Array-Trie or the CS-Prefix-Tree as the encoding index).

While the CS-Prefix-Tree is more expensive to bulk load (because we need to sort the complete input before bulk loading the index) than the CS-Array-Trie (which supports efficient top-down bulk loading), the workload which represents the *no-update* pattern is faster on the CS-Prefix-Tree because the CS-Prefix-Tree is (1) read-optimized and (2) not as high a the CS-Array-Trie. Furthermore, in general a workload that is more update-intensive (i.e., which has more new string values) has a better performance on the CS-Array-Trie. The performance of our CS-Array-Trie is comparable to the pure list-trie (that also uses buffers to speed-up the lookup of bulks).

3.6.3 Scalability of the Dictionary

This experiment shows the performance of our data structures for unsorted workloads of different sizes with $1M$ up to $64M$ distinct strings (with a fixed string length of 10). The remaining configuration for data generation was the same as in Subsection 3.6.1. For the leaves, we used the same configuration as in the experiment before.

Figure 3.13(a) shows the time for encoding these workloads using different encoding indexes for the dictionary (starting with an empty dictionary). After encoding, we bulk

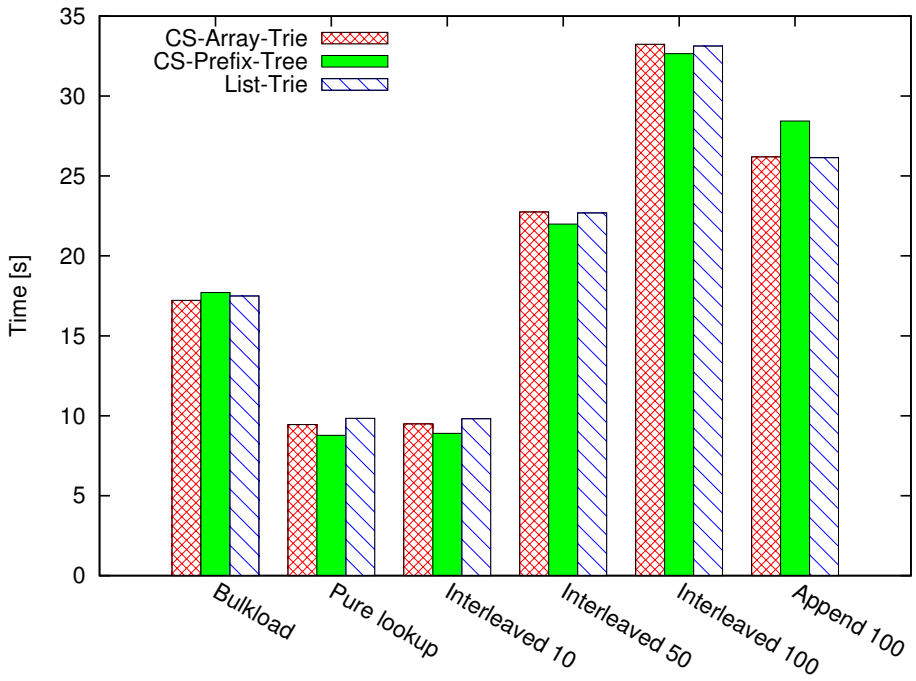


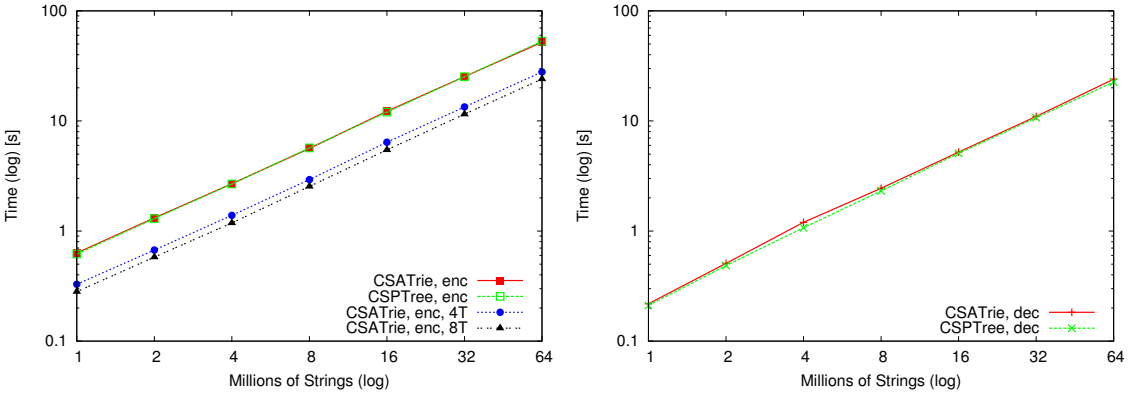
Figure 3.12: Lookup / update costs of encoding indexes

loaded a decode index (i.e., a CSS-Tree) from the leaves of the encoding indexes and used the encoded workloads that we created before for decoding (Figure 3.13(b)). For example, encoding $8M$ strings with the CS-Array-Trie takes 5.6 s and the decoding takes 2.4 s (while using one thread for both operations). We also scaled-up the number of threads (up to 16) to show the effects of parallelization on the CS-Array-Trie. The CS-Prefix-Tree does not support parallelization. For example, using 8 threads reduces time for encoding $8M$ strings from 5.4 s to 2.5 s. Figure 3.13(a) does not show the results for 16 threads because the performance slightly decreased compared to 8 threads due to the overhead for thread synchronization. Moreover, pinning individual threads to a single CPU to avoid thread migration did not improve the performance.

3.7 Conclusion

In this chapter, we showed an approach for efficiently using dictionaries to compress a large set of variable-length string values with fixed-length integer keys in column stores. The dictionary produces order-preserving codes and supports updates (i.e., inserts of new string values). Furthermore, we have presented a new approach for indexing such a dictionary (called shared-leaves) that compresses the dictionary itself while offering efficient access paths for encoding and decoding. We also discussed a concrete leaf structure and two new cache-conscious indexes that can leverage the shared-leaves indexing approach.

The main motivation for such a global dictionary is to enable the query processing



(a) Encoding (b) Decoding

Figure 3.13: Scalability of the dictionary

on compressed data for most of the operations and during most of the operator chain. This is especially important to scale-out databases since then data gets more and more distributed and partitioned. A global dictionary then enables most steps, including the merge-step during the final query result production, to be executed on the compressed data.

An evaluation conducted in the SAP HANA DB team (and outside of the scope of this thesis) showed that an implementation based on a hash table for the global dictionary is sufficient for the current use cases, namely data warehousing scenarios. Thus, order-preservation is not such an important aspect for a global dictionary in column stores. One reason for this is that the sorting of strings usually happens on the final result that is presented to the user and has more "aesthetic" reasons than semantic impact on the query result. The amount of data that needs to be sorted tends to be small. Thus, sorting can be done on the limited amount of uncompressed data with low overhead. Range predicates on strings (e.g. `a like 'aa%'`) can be pushed down to the local dictionaries in many cases. Most other operators (e.g., set operations) work just as well with hash-based codes as with order-preserving codes. Thus the benefits of an order-preserving dictionary are limited in the context of data analytics. Applications that have to sort large (intermediary) results could benefit from an order-preserving dictionary. However, Applications in other areas are outside the scope of this thesis.

4

Lazy Updates for an Order-preserving Dictionary

Order-preserving encoding schemes are often used in database systems to compress data and to improve the performance of certain query processing operations such as sorting and searching. While a variable-size physical representation of the codes is more flexible with regard to updates, a fixed-size physical representation enables a more efficient processing of the codes. Consequently, an ideal order-preserving encoding scheme would use a fixed-size physical representation while still enabling efficient updates. However, in a dynamic setting with updates, a fixed-size physical representation often comes with the need to re-encode existing codes which has a negative impact on the overall performance of a database system if not done carefully. Thus, in order to postpone the need for re-encoding, order-preserving encoding schemes usually leave 'gaps' to prepare for future insertions based on certain assumptions about the workload which are in most cases not perfect.

In this chapter, we present a novel order-preserving encoding scheme called *multi-version encoding* that can use a fixed-size physical representation while still supporting updates efficiently. Compared to existing order-preserving encoding schemes 1) it does not make any assumptions on where to leave gaps for future insertions, and 2) it provides means to re-encode existing codes lazily (e.g., when data is re-written anyway). Moreover, in this chapter we also discuss the integration of multi-version encoding into a dictionary-based compression scheme of a column store as presented in the previous chapter. In the experiments, we show that the generated codes enable efficient query processing while keeping the overall impact of re-encoding low.

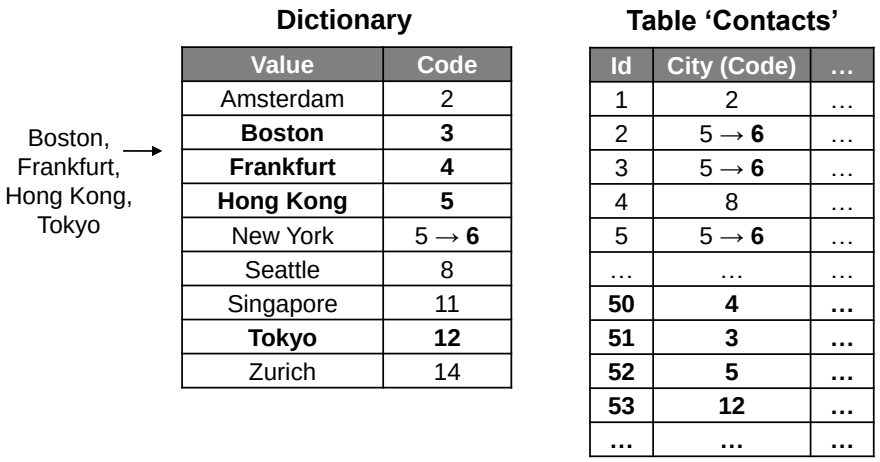


Figure 4.1: Traditional dictionary-based order-preserving encoding schemes

4.1 Motivation

Order-preserving encoding schemes are often used in database systems to compress data and to improve the performance of certain query processing operations such as sorting and searching [3, 6, 7]. The design of a concrete order-preserving encoding scheme mainly depends on whether it should be used in a static setting (i.e., without updates) or in a dynamic setting and whether it can use a variable-size or a fixed-size physical representation. While a variable-size physical representation is more flexible with regard to updates, a fixed-size physical representation enables a more efficient processing of the codes.

Consequently, an ideal order-preserving encoding scheme would use a fixed-size physical representation while still enabling efficient updates. However, in a dynamic setting with updates, a fixed-size physical representation of the codes comes with the need to update existing codes. This happens when the code space is exceeded in certain regions where new values are inserted more often than in other regions. In order to postpone the need for re-encoding existing codes, order-preserving encoding schemes often leave 'gaps' to be able to squeeze in codes for new values. However, predicting where to leave gaps and which size the gaps should have is often based on certain assumptions (e.g., statistics of previously inserted values) which are in most cases not perfect. As a consequence, leaving gaps in the generated codes does not avoid the need for re-encoding and even worse the gaps can often not be leveraged because they are at the wrong place [28].

Figure 4.1 shows an example of compressing strings using an order-preserving dictionary based encoding scheme which leaves gaps of size 2 and uses a fixed-size code representation of 4 bits (i.e., possible codes range from 0 to 15). In the Figure, a situation is shown where the dictionary contains five string values (non-bold) and a bulk of four new string values (bold) is inserted into the same dictionary. In order to encode that bulk, however, existing codes must be re-encoded (e.g., we need to update the code for the value 'New York' in that case from 5 to 6 as shown in Figure 4.1).

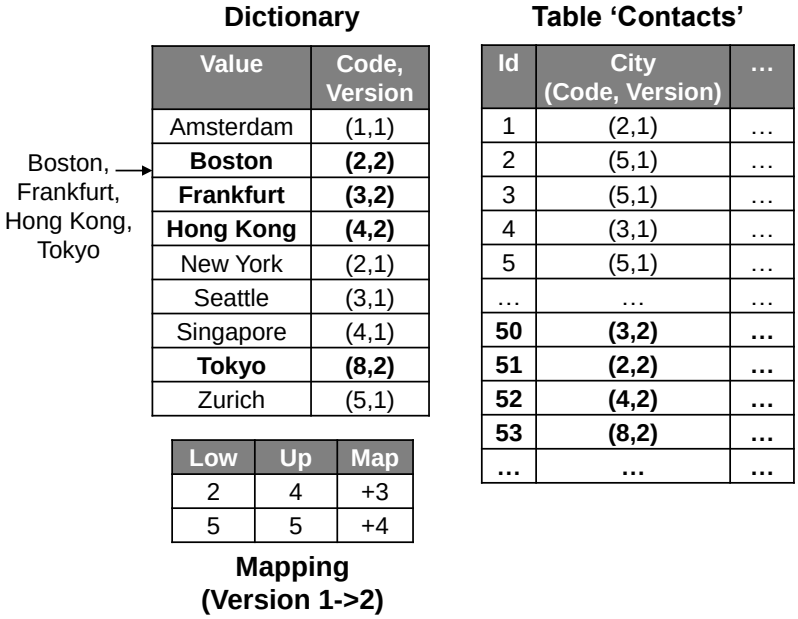


Figure 4.2: Multi-version encoding scheme

Moreover, re-encoding existing codes implies that we need to update all references to these codes. In the example above, the dictionary codes are used in a database table called 'Contacts' to compress information about cities (see Figure 4.1 on the right side). When re-encoding the dictionary (i.e., by inserting new values), we also need to update the affected codes of that table. However, in database systems updating (i.e., re-writing) huge parts of a table is not desirable because it has a significant negative impact on the overall performance. This is an issue especially if the dictionary is used for more than one column or partition like in the case of a global dictionary.

In this chapter, we present a novel order-preserving encoding scheme called *multi-version encoding* that can use a fixed-size physical representation while still supporting updates efficiently. Compared to other order-preserving encoding schemes 1) it does not make any assumptions on where to leave gaps for future insertions, and 2) it provides means to re-encode existing codes lazily which gives us the flexibility to decide when to apply re-encoding (e.g., when data is re-written anyway, for example during a merge of the write-optimized delta-index with the read-optimized main index [48]).

Figure 4.2 shows a similar situation as the one in Figure 4.1 with the difference that the dictionary uses a multi-version encoding scheme to compress the string values. The dictionary contains five string values (non-bold) that already have been encoded using the codes 1 to 5 in version 1 (denoted by $(1,1)$ to $(5,1)$). Moreover, a bulk of four new string values (bold) is inserted to the dictionary. However, in code version 1 there are no gaps in the codes which allow to insert new string values. Therefore the multi-version encoding scheme creates a new version 2 that has gaps at the right place where the new values can be inserted. In Figure 4.2, the codes greater equal 2 of version 1 are shifted back by the constant 3 to create the first gap and the codes greater equal 5 are

shifted back to create another gap of size 1. Logically, if we would map all codes in the dictionary to version 2 they would be order-preserving (e.g., in version 2 the code for 'New York' would be 5, the code for 'Seattle' would be 6 and so on).

The gaps that need to be created for a new version are stored in a so called *Version Mapping Table*. The mapping tables can be used to translate codes from older versions to newer ones. Figure 4.2 shows the mapping table which translates codes from version 1 into version 2. This mapping table defines that existing codes between 2 and 4 in version 1 need to be shifted by +3 and code 5 needs to be shifted by +4. With the help of the mapping tables re-encoding can be applied lazily. In the example of Figure 4.2 existing codes in the dictionary as well in the table 'Contacts' are not updated immediately (i.e., they can remain in version 1). Consequently, re-encoding these codes (i.e., mapping the codes to the new version) can be done whenever the dictionary or the table 'Contacts' is re-written anyway. In some sense, the mapping table can be seen as the variable part of the encoding scheme that makes it possible to keep the codes themselves fixed-sized.

However, updating existing codes lazily comes with costs for query processing as only codes in the same version are order-preserving (i.e., operations that need order have to map the codes into a common version). Assume we want to sort the table 'Contacts' by the city names using the compressed codes. In that case, we first need to map all codes to one common version and afterwards we can sort the table using these mapped codes.

In general, multi-version encoding is orthogonal to existing order-preserving encoding schemes. Thus, it can be combined with a very simple order-preserving encoding scheme as the one shown in Figure 4.2 which generates sequential codes or with a more sophisticated encoding scheme which leaves gaps based on statistics of the workload.

In summary, the contributions of this chapter are:

- A novel encoding scheme called *Multi-Version Encoding*.
- A description of how to integrate Multi-Version Encoding into the dictionary based compression scheme of a column store.
- New indexing methods for a dictionary to encode strings as multi-version codes and to decode multi-version codes.
- A discussion of how query processing operations can leverage multi-version codes efficiently.
- A comprehensive performance evaluation.

The remainder of this chapter is organized as follows: Section 4.2 discusses related work, Section 4.3 gives a real life use case (i.e., the dictionary-based compression scheme of a column store similar to the use case of the previous chapter) that motivated this work and derives requirements from it. Section 4.4 discusses the details of the new encoding scheme. Section 4.5 shows how multi-version encoding can be integrated into a dictionary-based compression scheme of a column store. Section 4.6 then discusses

several optimizations for re-encoding and query processing multi-versioned codes. Section 4.7 presents the results of our performance evaluation. Finally, Section 4.8 concludes the chapter.

4.2 Related Work

In the area of XML Labeling, there is a body of literature on order-preserving encoding schemes based on Dewey labels (e.g. [75, 79]) or ORDPath [60]. These encoding schemes all use variable length keys and are thus not optimal to process. Indeed, every variable-length code can be restricted to a fixed-size code representation, but then the encoding scheme is nothing else than a clever way of leaving gaps in the code space to prepare for future updates [28]. As discussed before, any of these schemes could be used as the underlying encoding schema for multi-version encoding. Multi-version encoding then would take care of the inevitable re-encoding in a lazy fashion while the underlying encoding schema aims to delay the re-encoding.

Antoshenkov et al. [6, 7] looked into order-preserving compression using dictionaries but use variable-length codes. The integration of compression in query processing has also been an area of interest, for example in the context of databases in general [27, 39] or concrete systems like C-Store [3, 44] or MonetDB [84]. However, none of this work uses a global dictionary to encode more than one column of the same domain. We will see in Section 4.3 why that scenario changes the requirements for the encoding scheme.

Finally, our work leverages ideas from different approaches to cache-conscious indexing (e.g.,[66, 67]) to provide efficient access paths to the dictionary.

4.3 Problem Statement

In this section we present the real-world application scenario that motivated the design of the multi-version encoding scheme. Furthermore, we also derive concrete requirements for the encoding scheme from that application scenario.

4.3.1 Use Case: Global Dictionary

Column stores such as C-Store [74] or Monet-DB [83] use lightweight compression schemes to compress data and to increase performance of query processing for analytical workloads in data warehousing. One such lightweight compression scheme is to use a dictionary to replace long variable-length values with shorter fixed-length integer codes (as shown in Figure 4.1). When using one dictionary per column to compress the data, certain query operations like the equi-selection operator can be executed directly on the compressed data. Moreover, when generating order-preserving codes further operations such as sorting and range-selections can also be executed on the encoded data. However, operations such as unions or joins which combine data from different columns which use different dictionaries must decompress the data before query execution. Thus, our idea is to integrate a *global dictionary* into a column store. That global dictionary generates

order-preserving fixed-length codes for all columns of the same domain to enable more query operations (e.g., joins and set operations) on the compressed data.

However, integrating an order-preserving global dictionary into a column store is not trivial because of several reasons: One reason is that the global dictionary is updated more frequently than local dictionaries because it is shared by several columns. Consequently, if new values are inserted into one column that is compressed using the global dictionary and re-encoding becomes necessary, existing codes of all other columns that use the global dictionary might need to be updated. For example, assume that the codes of the dictionary shown in Figure 4.1 are used to compress columns of different tables. Thus, when dictionary entries need to be re-encoded because new values are inserted in one table (e.g., into table 'Contacts'), all other tables that use the same global dictionary must be updated accordingly. However, this is usually too expensive for many scenarios. Especially in a data warehousing environment where column-stores are usually used to process analytical queries, cubes (i.e., fact tables and dimension tables) hold huge amounts of data and thus re-encoding has significant costs as well as negative impact on the overall performance if it must be done on-line (i.e., concurrently to running analytical queries).

4.3.2 Requirements

In the following, we list the requirements for an order-preserving encoding scheme that can be used for a global dictionary within a column store to compress variable-length values for a data warehousing environment:

Fixed-length Codes For analytical workloads, the query performance is the main optimization goal. Thus, encoding schemes should generate fixed-length codes that are more efficient to process rather than variable-length codes [3, 84].

Lazy Re-encoding Data Warehouses store huge amounts of data. Thus, re-encoding the global dictionary should not require the immediate re-encoding of all columns that use the codes of the global dictionary. Instead, re-encoding should be doable in a lazy fashion (e.g., when a column needs to be re-written anyway).

Workload Independence Workload characteristics in a real data warehousing environment are constantly changing (e.g., because new cubes are created and existing ones are updated regularly). Thus, an encoding scheme should be flexible with regard to changes in the characteristics (e.g., distribution) of the workload.

Optimized for Bulks Data is usually loaded into the cubes of a data warehouse using bulk operations. Thus, the encoding scheme for the global dictionary should be optimized for encoding bulks of values rather than inserting single values.

Query Performance Many analytical queries do not include operations that require order. Thus an order-preserving encoding scheme should have no significant impact on the performance of these queries (i.e., those queries should have a similar performance as if they where executed on non order-preserving codes).

4.3.3 Other Use Cases

The general ideas behind multi-version encoding can be used together with any kind of encoding scheme that requires flexibility in the decision when to re-encode. An example where multi-version encoding could be useful is for XML labeling [60, 79]. When inserting new nodes into an XML document, re-labeling might become necessary when a fixed-size physical representation is used for the node labels [28]. However, re-labeling a complete subtree of an XML document might not be desirable. Thus, the ideas of multi-version encoding could also be applied here to enable lazy re-encoding.

Furthermore, an extension of this work was applied in the context of cloud computing in order to improve the privacy of the user's data [46]. The idea is to have a dictionary on a trusted machine to encode the sensitive data before shipping it to the cloud. Since the dictionary encoding is order-preserving and supports updates, the cloud can still do most of the compute-intensive processing while not learning any sensitive details about the data.

4.4 Multi-Version Encoding

In this section, we first introduce the basic concepts for the novel coding scheme called multi-version encoding and show that it can efficiently use a fixed size physical code representation. Moreover, we also analyze the encoding scheme with regard to certain update patterns (i.e., encoding single values versus encoding bulks).

4.4.1 Basic Concepts

Multi-Version Encoding extends existing order-preserving encoding schemes with the possibility of doing lazy re-encoding. The basic concepts of the multi-version encoding scheme are that 1) all codes c are tagged with a *version number* v (also called versioned-code (c, v)), and 2) additional information is stored in so called *version mapping tables* which map codes from an older version into a more recent one.

Using these concepts, lazy re-encoding can be added to an existing order-preserving encoding scheme as follows: If a new value should be encoded but the code space is exceeded in the code region where we need to insert a new code, multi-version encoding creates a new code version which introduces a 'gap' into the code space (e.g., by shifting existing codes) that enables the encoding of the new value. The information about where the new gaps are introduced and what size these gaps have is encoded in a corresponding version mapping table.

Figure 4.2 shows an example where a simple order-preserving encoding scheme, which generates sequential codes, is combined with the multi-version encoding scheme. As discussed before, in this example two gaps are introduced in code version 2, one of size 3 which ranges from code $(2, 2)$ to $(4, 2)$ and one of size 1 at code $(8, 2)$. These gaps in the code version 2 are used to generate codes for the four new string values (shown in bold).

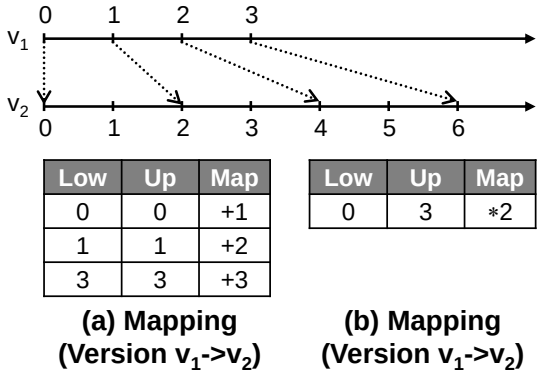


Figure 4.3: Compressing version mapping tables

With multi-version encoding codes are only order-preserving if they are in the same version.[1] Thus, if we need to order codes in different versions, the version mapping tables need to be applied to map codes to a common version. For example, if we need to order the table 'Contacts' of Figure 4.2 by city codes, the version mapping table that maps codes from version 1 to version 2 must be applied for all codes in version 1.

In general, a version mapping table defines a mapping of codes from a source version v_s to a target version v_t whereas version v_t is a more recent version than v_s. Version mapping tables that map codes back from version v_t to version v_s are not relevant for many applications (as we will see in Section 4.5 for the global dictionary) since such a mapping is not defined for all codes in v_t (i.e., more codes exist in version v_t than in version v_s). An entry of the version mapping table consists of three attributes *Low*, *Up*, and *Map*. The attributes *Low* and *Up* define the lower and upper bound of the code range in version v_s for which the mapping operation defined by the attribute *Map* should be applied.

The main map operation is shifting, i.e., adding a positive or a negative constant to the code. A positive constant opens a gap whereas a negative constant closes a gap. When using shifting as the only map operation, the size (i.e., the numbers of possible entries) of a version mapping table from a version v_s to another version v_t is bounded by n, where n is the number of codes existing in version v_s. This is because the mapping table can have at most one entry per code value.

However, since n can become large, e.g., when using 32 bits for the codes, the number of entries can be reduced by allowing more sophisticated map operations such as stretching (i.e., multiplication). Figure 4.3 shows two code versions (v_1 to v_2) and the mapping between these two versions. In version v_2 three new codes have been inserted in between each code of version v_1. Table (a) shows a version mapping table that uses shifting as the only map operation. In that case, the mapping table has size 3 (which is the upper bound). Table (b) shows an equivalent version mapping table which has only

[1] This fact is used to provide privacy in the mentioned cloud computing scenario: the data is encoded in multiple runs and the cloud does not have the mapping tables to translate between these runs. Thus the attacker cannot learn the total order of the data [46].

size 1. The only entry of that table is defined for the code range 1 to 3 and defines a multiplication operation.

If codes must be mapped over multiple versions (e.g., say from version 1 to 10), single version mapping tables can be applied consecutively. In order to optimize a mapping over multiple versions, a corresponding version mapping table could be materialized that defines directly that mapping (e.g. $1 \rightarrow 10$). Whether or not to create such a version mapping is a tradeoff between increased costs for storing the additional mappings and decreased costs for applying the mapping.

The size of a version mapping table $v_1 \rightarrow v_3$ that materializes two subsequent mappings $v_1 \rightarrow v_2$ and $v_2 \rightarrow v_3$ is bounded by $3 * n_{12}$ where n_{12} is the number of entries in the version mapping table $v_1 \rightarrow v_2$ and n_1 is the number of codes in version v_1. This is because a range of the first version mapping table can only be split into three sub-ranges if it is combined with an entry of the second version mapping table. Moreover, the upper bound for version mapping tables mentioned before holds for materialized mapping tables as well.

4.4.2 Fixed-size Code Representation

Multi-versioned codes (c, v) can be represented as fixed-size code size by reserving some bits of the code for the version number v. Say, we want to use 32 bits in total to represent multi-versioned codes. Depending on the workload that needs to be encoded, a certain number of bits should be reserved for the version number v (e.g., the first 4 bits) while the remaining bits (e.g., the last 28 bits) can be used to represent the code c that is generated by an underlying order-preserving encoding scheme. If zero bits are used for the version number (i.e., only one version is available), then the multi-version encoding scheme behaves exactly as the underlying order-preserving encoding scheme.

When using a fixed number of bits for the version, multi-version encoding will run out of version numbers at a certain point. Thus, codes in old versions should constantly be mapped into newer ones to be able to recycle old version numbers (if they are not used anymore). As mentioned before, compared to other encoding schemes multi-version encoding provides an efficient mechanism to do this lazily using the version mapping tables (e.g., when a table needs to be rewritten anyway).

Another important aspect of multi-version encoding is the representation of the version mapping tables, which basically enable the lazy re-encoding. This is not done as a part of the code (i.e., mapping tables are kept in a separate data structure), which enables the processing of multi-versioned codes if we do not require order without any overhead. Strictly speaking, multi-versioned codes have a variable part (i.e., the version mapping tables, which are applied when needed). However, the resulting codes are always fixed-length. Section 4.5 discusses a concrete data structure that can be used for the version mapping tables of the global dictionary compression scheme.

4.4.3 Analysis of Update Patterns

Multi-version encoding creates a new code version whenever a new value should be encoded and the fixed-size code space is exceeded in the region of the value. Creating a new version comes with additional costs: A version mapping table must be created and applied either during query processing if order is needed or for re-encoding. Moreover, if more versions are created, re-encoding must be applied more often to not run out of version numbers when using a fixed code representation.

Thus, in an environment where single values need to be encoded, it makes sense to combine multi-version encoding with an order-preserving encoding scheme that leaves gaps based on statistics in order to reduce the total number of code versions. This can help when multi-version encoding is used to compress data within a database that runs OLTP-like workloads.

In a data warehousing environment where bulks of values that need to be compressed are loaded into a database, combining multi-version encoding with a simple encoding scheme that generates sequential codes might make sense if no assumptions can be made about the workloads. However, if workload characteristics are known up-front, using a more sophisticated order-preserving encoding scheme always helps to reduce the number of code versions.

4.5 Global Dictionary Compression

In this section, we first give an overview of how multi-version encoding can be integrated into a global dictionary that compresses variable-length values using order-preserving fixed-length codes. Afterwards, we present the details of how encoding and decoding can be done efficiently within a global dictionary. Finally, we discuss the effects of versioned codes on individual query processing operations.

4.5.1 Overview

The two main tasks of the global dictionary are 1) encoding a bulk of string values when new data is loaded into a database, and 2) decoding query results during query processing.

For encoding string values using a global dictionary, we propose to use multi-version encoding together with a simple order-preserving encoding scheme that generates sequential codes. This encoding scheme is called *multi-version sequential encoding* further on. Using this encoding scheme has different reasons: Firstly, it fits perfectly to the update patterns of a data warehousing environment with bulk updates as discussed in Subsection 4.4.3. Secondly, it does not make any assumptions about the workload characteristics, which is one requirement discussed in Subsection 4.3.2. Finally, versioned codes can have a fixed-length representation while still being update efficient.

In general, the global dictionary can hold codes in different versions (i.e., codes for different values need not to have the same version). The most recent version of all codes in the global dictionary is called the *current version*. If we would map all codes of the

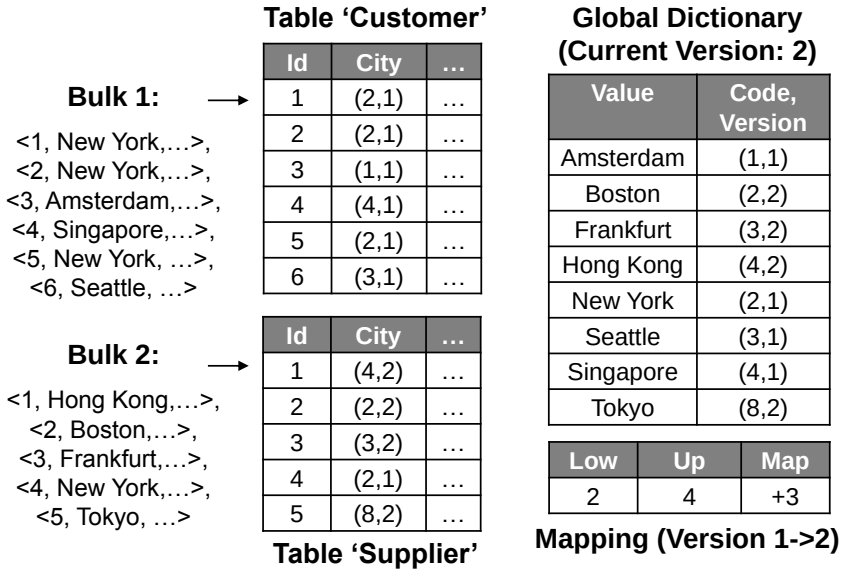


Figure 4.4: Global dictionary compression

global dictionary into the current version, we would get a list of ordered codes (more precisely sequential codes if we use multi-version sequential encoding). Figure 4.4 shows an example where two tables ('Customers' and 'Suppliers') share a global dictionary that uses multi-version sequential encoding to compress the attribute 'City' of each of these tables.

When encoding an existing value, we suggest that the global dictionary returns the same versioned code as stored. That way, query operations that do not rely on order (e.g., set-operations) do not need to pay the costs of mapping codes to a common version since a given value is always encoded using the same versioned code (if the codes in the global dictionary remain stable). For example, executing a distinct union of the city names in both tables shown in Figure 4.4 can be done without mapping codes to a common version.

The first bulk that is encoded using the global dictionary has code version 1. The versioned code that is generated for the first value in that bulk is $(1, 1)$. When encoding new values in subsequent bulks, in most cases the global dictionary needs to create a new code version since multi-version sequential encoding does not leave gaps in a code version for future insertions. Creating a new code version requires the creation of version mapping tables, which map codes of older versions to this new version to enable query operations that rely on order. Only in the case that all new values can be appended at the end of the global dictionary, the current version of the global dictionary can be used for encoding.

Assume that the bulks shown on the left side of Figure 4.4 were used to bulk load the two tables 'Customer' and 'Supplier'. Table 'Customer' was bulk loaded first using bulk 1. Thus, all codes of this bulk were inserted into the empty global dictionary and encoded using codes $(1, 1)$ to $(4, 1)$. Since this was the first bulk loaded into the

global dictionary, no version mapping table was created. Afterwards bulk 2 was loaded into table 'Supplier'. While existing values in that bulk were encoded using codes in version 1, new values like 'Boston' were inserted into the global dictionary and encoded using a new version (i.e., version 2). For that version, a version mapping table was created that maps codes from version 1 to version 2. Assume that we finally want to load another bulk that only contains a customer in 'Zurich' (not shown in the Figure) into table 'Supplier'. For encoding this value, it could be appended at the end of the global dictionary and encoded using the versioned code $(9, 2)$ (i.e., no new version needs to be created).

Another important task of the global dictionary is the decoding of query results. If we can make sure that the codes of a query result are in the same version as in the global dictionary, decoding is a pure lookup. For example, in order to decode all city codes in table 'Supplier', one only needs to lookup the string values in the global dictionary that correspond to the codes in the given version. However, if a query operation needs to order codes, it might need to map the codes in the input of the operation to a common version (e.g., for sorting table 'Supplier' by their city names). The result of mapping the codes should not overwrite the original codes in the input, because that would make decoding more complicated. Instead, if the mapped codes are used by a subsequent query operation, they should be added to the output as a temporary column.

As shown above, storing the same versioned code for a given value in all tables as well as in the global dictionary has the benefit that query operations that do not rely on order do not need to map codes at all and decoding is a pure lookup. However, this approach makes re-encoding (i.e., mapping old code versions to newer ones) more complicated since all tables and the global dictionary need to be updated at the same time. Thus, lazy re-encoding cannot be done incrementally table by table (e.g., whenever a table needs to be re-written anyway). In Section 4.6, we show how we can tackle that issue efficiently.

4.5.2 Encoding

Encoding a bulk of string values using a global dictionary includes looking up the codes for existing values and creating codes for new values. Using multi-version sequential encoding, creating new codes involves in most cases creating a new version and version mapping tables as discussed in the Section before.

Algorithm 4.1 shows a function that inserts a bulk of string values V into an existing global dictionary D and generates codes for the new values using multi-version sequential encoding.

First, this algorithm inserts the new values in bulk V into the global dictionary and its current version is extracted (lines 2-3). For example, in Figure 4.5 three new values are inserted into an existing global dictionary, which holds codes in versions 1 and 2 (i.e., the current version is 2). Inserting these new values creates two gaps in the global dictionary (e.g., the first is between 'New York' and 'Seattle'). For these gaps new codes need to be generated. This is done in Listing 4.1 by lines 6-20: For each gap G in ascending order, the first versioned code after the gap $G.up$ is mapped to a code in the current version

Listing 4.1: Bulk encoding of new values V

```
Mapping encode(values v, dictionary d) {
  d.insert(v); // Insert new values into global dictionary
  version = d.currM // Set version for encoding
  shift = 0M // accumulated shifts

  Mapping m = new mapping();
  // create empty version mapping table

  for (/* all gaps g in d in ascending order */) {
    Code up = map( g,up, d.curr );
    // Code after gap in current version
    Code code = code(up) + shift // First new code for gap
    if ( /* g is not at the end of d */) {
      if (m.isEmpty()) {
        // Create new version number
        version = next(version);
      }
      shift = shift + size(g); // Create mapping entry
      m.insert(code(up), shift);
    }
    for ( /* all entries e in gap g in ascending order */ ) {
      e = (code,version); // Generate codes
      code = code + 1;
    }
  }
  d.curr = version; // set current version of global dictionary
  return m;
}
```

of the global dictionary using the existing version mapping tables (line 7). If gap G is the last gap then $G.up$ denotes the next possible code after the last code in the global dictionary.[2] The resulting versioned code is stored in the variable *up* (line 7) and used later on (1) to determine the code for the first entry in the gap (line 8) and (2) as the lower bound to create the entry of the version mapping table m for gap G (lines 9-15). In the example in Figure 4.5, code $(3, 1)$ is the first versioned code after the first gap. This versioned code is mapped to code 6 in the current version 2.

When using multi-version sequential encoding, an entry of the version mapping table contains only the lower bound and the mapping operation (no upper bound as shown in Figure 4.4). The upper bound does not need to be stored explicitly in the version mapping table since it is given implicitly by the lower bound of the next entry. In Listing 4.1, the lower bound of an entry in the version mapping table is defined by the variable *up* while the mapping operation is defined as shifting by the accumulated gap sizes stored in variable $shift$ (line 14).[3] In Figure 3.6, the first entry of the version mapping table $2 \rightarrow 3$ shifts codes ≥ 6 and < 8 by $+2$ (i.e., the first gap size). The second entry defines that codes ≥ 8 are shifted by $+3$ (i.e., the accumulated size of the first two gaps).

Moreover, if the first entry is added to the version mapping table m, a new version number is created and stored in variable $version$ (lines 10-12). This version number is used in lines 16-19 to generate codes for all entries in the current gap G. Finally, after all gaps G are processed, the current version of the dictionary is set to be the new code version and the version mapping table m is returned (line 20).

The function *map((code, source), target)* (used in Listing 4.1 line 7) maps a code in a source version $((c_s, v_s))$ to a code in a given target version (v_t). If a code needs to be mapped over multiple versions, this function can apply several version mapping tables consecutively. For example, if we want to map a code 3 in version 1 of the global dictionary on the right side in Figure 3.6 to version 3 then two mapping tables can be applied to first map the code from version 1 to 2 and then from version 2 to 3. If multiple paths exist to map a code from the source to the given target version (e.g., if a materialized version mapping table for $1 \rightarrow 3$ would also exist) this function can use the shortest mapping path.

In Chapter 3, we presented a new approach which compresses the dictionary data while enabling efficient access paths. The basic idea is that both an encoding and decoding index are built over the dictionary whereas both indexes can be clustered as the values and codes are in the same physical sort order when generating order-preserving codes. Moreover, both indexes can thus share the same leaves that directly hold the dictionary data (called *shared-leaves indexing*). The leaves that are used to store the dictionary data are similar to slotted pages used in databases to store variable-length values but compress the string values using prefix-compression.

However, for multi-version encoding codes are only order-preserving within one ver-

[2] For the first bulk that is loaded $G.up$ is thus initialized by $(0, 1)$ and therefore the first code that is generated will be $(1, 1)$.

[3] Multi-version sequential coding only needs to shift by positive constants since the encoding scheme starts with $(1, 1)$ as first code and does not leave any gaps.

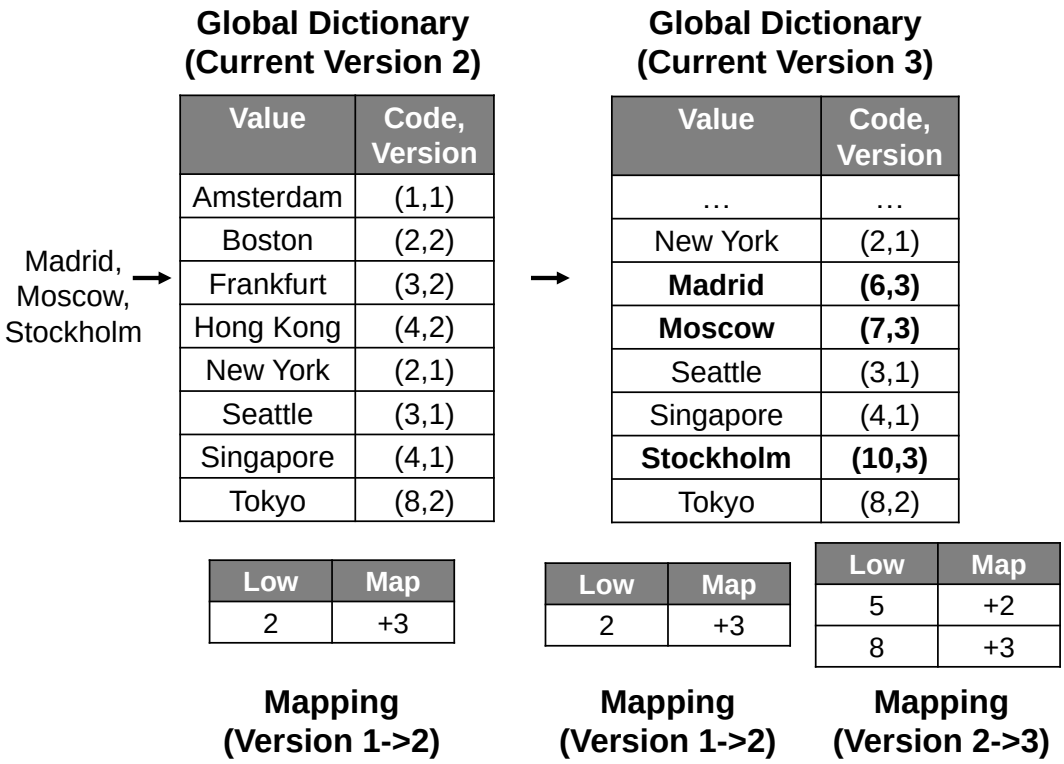


Figure 4.5: Bulk encoding

sion. Consequently, an obvious idea for indexing a global dictionary that uses multi-version sequential codes is to use one clustered and one non-clustered index. In order to enable efficient bulk encoding, the encode index should be clustered since the encoding function shown above requires to access the first code after the gap (in sort order of the string values). Other more sophisticated ideas, which also enable a clustered decode index that can share the leaves with the encode index, are discussed in the following section.

4.5.3 Decoding

As mentioned before, we suggested a leaf structure that compresses the dictionary data. These leaves store variable-length values (i.e., string values) and fixed-length integer codes in sort order using prefix-compression for the strings while each n-th string is stored uncompressed. At the end of the leaf, an offset vector is stored that has entries which point to the uncompressed string values. Moreover, each entry in the offset vector also holds additionally the corresponding codes of the uncompressed strings to enable efficient lookup operations for decoding. The encoding and decoding indexes use the first value and code per leaf as separators for the index nodes above the leaf level.

When using multi-version encoding for the global dictionary, the codes in the leaves are not order-preserving since in most cases they are in different versions. Thus, using a clustered decode index that shares the leaves with the encode index is not trivial. In the following, we present three different decoding strategies for multi-version encoding. The first two strategies use a clustered index while the third strategy uses a non-clustered index which does not need to index all versioned codes per leaf.

Mapping-based Indexing

The idea of this indexing strategy is to use codes that are in the current version v_c of the dictionary as separators of the decode index nodes, i.e., the nodes above the leaf level hold separators that are the minimal codes of each leaf mapped to the current version. In order to lookup a value for a versioned code (c_s, v_s) using such a decode index, we first need to map c_s from version v_s to code c_c in the current version v_c using the *map* function discussed in the Section before. The versioned code (c_c, v_c) can then be used to probe the decoded index to find the leaf that holds the versioned code (c_s, v_s). As discussed in Subsection 4.5.2 before, for decoding we require that only lookup keys (i.e., versioned codes (c_s, v_s)) are used that are stored in the in the global dictionary (i.e., in the leaves). Thus, to find the corresponding value for (c_s, v_s), we need to sequentially scan the leaf that is returned by probing the decode index.

Figure 4.6 shows a global dictionary, which uses three leaves to store the dictionary data. The decode index on top has only one node, which separates the leaves using the minimal codes per leaf mapped to the current version 3 (e.g., the minimal code of the second leaf is used as the first separator). In order to lookup the versioned code $(8, 2)$, this versioned code must first be mapped to version 3, which results in versioned code $(11, 3)$. This versioned code is then used to probe the decode index, which returns the

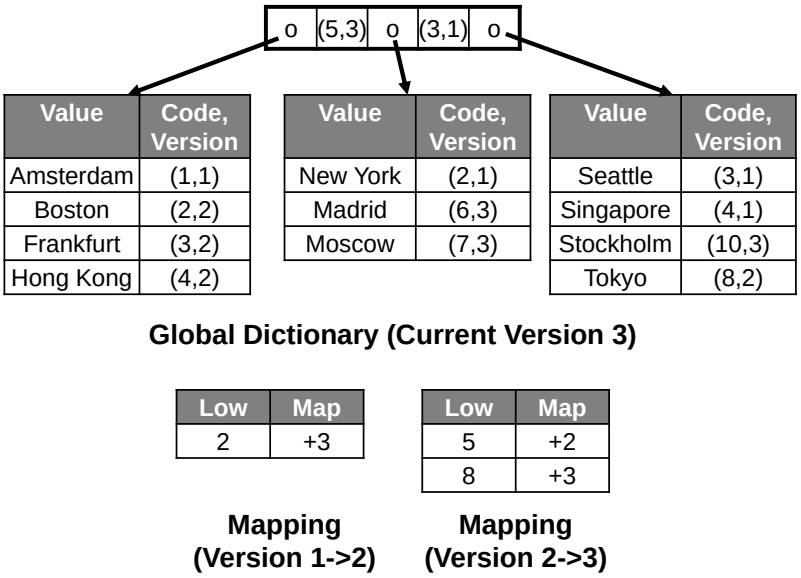


Figure 4.6: Mapping based indexing

last leaf as $(11, 3) \geq (9, 3)$ holds. Finally, the last leaf is sequentially scanned to find the value using the original versioned code $(8, 2)$ as lookup key.

Moreover, if the versioned codes in the offset vector are stored in the current version v_c (during encoding), then we can use the mapped versioned code (c_c, v_c) that we used to probe the decode index also for scanning the offset vector, which returns the optimal offset. Assume that in Figure 4.6 each other value is stored uncompressed, i.e., each other code is also stored in the offset vector of a leaf. Thus, if we map the codes in the offset vector to the most recent version (e.g., for the last leaf we map the first code from $(3, 1)$ to $(8, 3)$ in the offset vector), we can skip entries in an optimal way when using the mapped versioned code $(11, 3)$ for scanning the offset vector. Since $(11, 3) \geq (10, 3)$ holds, sequential scanning can skip the first two leaf entries.

Guideline-based Indexing

The idea of this indexing approach is that the nodes hold multiple versions of the same code for each separator (i.e., the number of branches remains the same as with the mapping-based index but there can be multiple keys for each branch). In order to lookup a value for a versioned code (c_s, v_s), we do not map the code to the current version of the global dictionary. Instead, code c_s is directly used to probe the decode index. When using multi-versioned sequential codes, the following relation holds: $(c_m, v_m) \geq (c_n, v_n)$ if $c_m \geq c_n$ and $v_m \leq v_n$ since the same code in a later version can only increase (as we use only positive constants for the version mapping operation). This relation gives only an approximation if code c_m is greater equal code c_n without mapping the codes to a common version.

Using this relation, the separators in the nodes can be used as a guideline that deter-

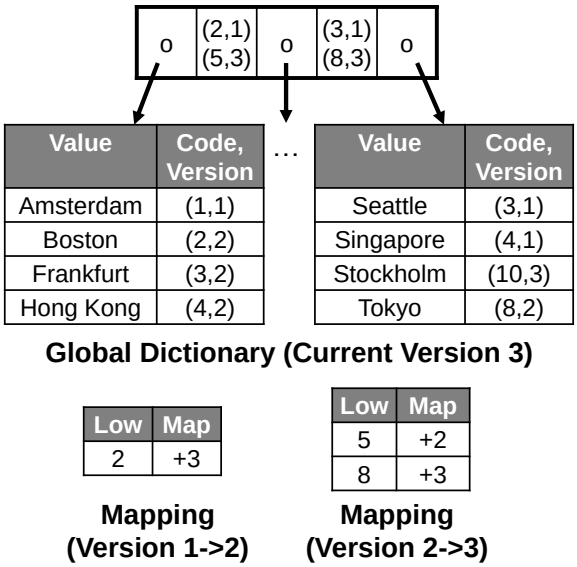


Figure 4.7: Guideline-based indexing

mines which branch to push down the key (c_s, v_s) by using the relation shown above for $(c_m, v_m) \geq (c_n, v_n)$ where (c_m, v_m) is the request and (c_n, v_n) is a versioned code stored in the separator. If this relation holds for one of the versioned codes stored as a separator, the right branch is used, otherwise the left branch is used. Thus, the request might be pushed down on the left branch while the versioned code is actually stored in the right branch. Consequently, if the lookup key (c_s, v_s) is not found during sequentially scanning the leaf, it must be propagated to the next leaf until the versioned code is found.

Figure 4.7 shows the same global dictionary as the one in Figure 4.6. However, the decode index stores two codes per separator, one in version 1 and another in version 3. When probing the index using the versioned code $(8, 2)$, the last separator holds the versioned code $(8, 3)$ for which $(8, 2) \geq (8, 3)$ holds since the for the codes $8 \geq 8$ holds and for the versions $2 \leq 3$ holds as well. Thus, the code is pushed down to the rightmost leaf. Afterwards, the leaf is sequentially scanned as for the strategy before.

Full Indexing

The idea of this strategy is that the index holds entries for each minimal code per version in each leaf. Thus, different nodes might point to the same leaf as shown in Figure 4.8. The versioned codes (v, c) used as separators in an index node are sorted ascending first by version number v and then by code value c. When probing the decode index using the key (c_s, v_s), we can always find the exact leaf using the before mentioned sort order to determine the branch where a key needs to be pushed down. As a drawback, the index size is much bigger as the index size for the first two decode strategies that only use one entry per leaf.

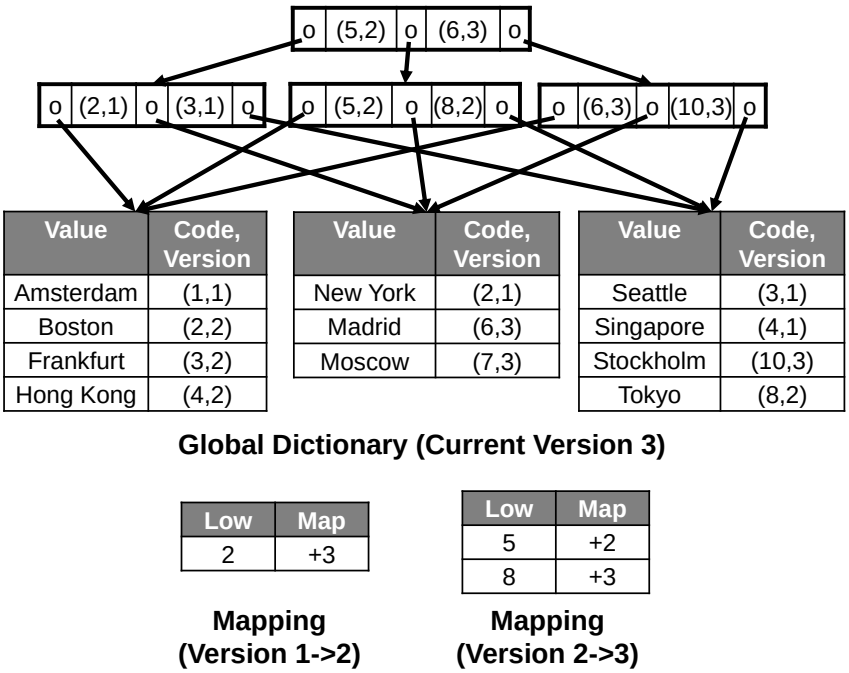


Figure 4.8: Full indexing

Guideline-based decoding is optimal if the versioned codes used for decoding are in a version close to the version of those codes used in the index (as well as to those in the offset vector of the leaves) since then comparisons of versioned codes are more exact. Mapping-based decoding works well for all versions if we can afford materialized mappings for all versions to the current version of the global dictionary (called full materialization). If not, the fall back strategy is the fully-indexed decoding strategy, which gives a constant performance over all versions while using less space than the full materialization in many cases.

When bulk encoding string values using the global dictionary, new entries might be added to the leaves and thus the decode index needs to be updated. Since the decode indexes for the first two strategies contain significantly less entries (when using a big leaf size which is optimal for data warehousing) than for the third one, it could be bulk loaded when the dictionary data changes using a read optimized index (e.g., a CSS-Tree [66]). Bulk loading the decode index makes sense in a data warehousing environment where data is loaded in bulks anyway and thus not updated too often. For the last strategy, we also propose to use a read-optimized index. However, since more updates need to be applied (in the worst case one update per leaf and code version) and since the index size is bigger, we use a more update friendly index (e.g., a CSB$^+$-Tree [67]) and instead of bulk loading the decode index we apply the updates directly (e.g., when a leaf must be split).

4.5.4 Query Processing

As discussed in Subsection 4.3.2, one requirement for multi-version encoding is that executing query operations that do not require order-preserving codes (e.g., unions) should not be significantly more expensive than executing those operations non order-preserving codes. This requirement is satisfied by multi-version encoding if a given value is always encoded using the same versioned code (as presented in Subsection 4.5.1). Thus, equality of two versioned codes holds if the version number and the code value are the same: $(c_m, v_m) == (c_n, v_n)$ if $c_m == c_n$ and $v_m == v_n$. Since multi-version encoding can have an efficient fixed-size physical representation (see Subsection 4.4.2), the costs for checking equality of two versioned codes are similar (with a minor overhead for the version number) to checking equality for any other encoding scheme that can be represented using a fixed-size physical representation.

However, when executing operations that need order, versioned codes need to be mapped to a common version. In the following, we analyze the costs for two query operations that need order: sorting and range-selection.

Sorting

A sort operation on an attribute a that is compressed using versioned codes can be executed by first mapping all codes in the input to the most recent version used by that attribute. For example, to sort the table 'Supplier' in Figure 4.4 ascending by the encoded city names, the codes first need to be mapped into version 2. The costs to map versioned codes into a target version depend on the number d of version mapping tables that need to be applied (called *mapping distance* further on) and on the size m of the mapping tables. Thus, if the input that needs to be sorted contains i versioned codes, then the costs for sorting are:

$$O(i * d * log(m) + sort\_cost(i))$$

where $O(i*d*log(m))$ are the costs for mapping i codes to the most recent version and $O(sort\_cost(i))$ are the costs for a particular algorithm to sort i elements. The mapping costs include the factor $log(m)$ since version mapping tables are sorted and thus binary search can be applied to find an entry in the table for a given code. Moreover, the average mapping distance d can be kept small if mappings that are used frequently get materialized.

Other more sophisticated algorithms for sorting versioned codes could use the relation $(c_m, v_m) \geq (c_n, v_n)$ shown in the section before, which allows to presort the input without mapping the codes. Which sort operation should be used is a decision that a query optimizer must take since the costs of these operations strongly depend on the data characteristics.

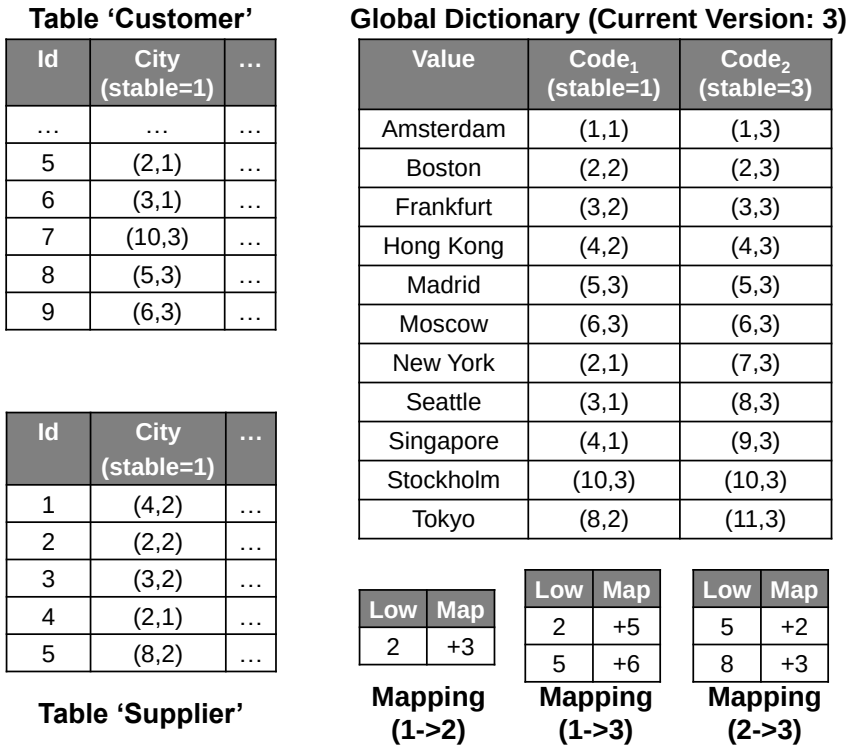


Figure 4.9: Global dictionary with stable versions

Range-Selection

Another operation that needs to map versioned codes is range-selection. For example, to search the table 'Supplier' in Figure 4.4 using the predicate $city = `S*'$ we can rewrite the filter predicate using the codes of the lower and upper bound that mark the corresponding range in the global dictionary (i.e., the codes for 'Seattle' and 'Singapore'). Thus, the rewritten query predicate would be $(3,1) \leq city \leq (4,1)$. If the range includes more than one code version in the dictionary (e.g., entries in version 2), we would need to find the lower and upper bound per version in the global dictionary and use a disjunctive predicate that combines the ranges per version.

Another option to execute the range-selection is to map the input (as for sorting) to the most recent version as well as the lower and upper bounds of the predicate. This is again a decision that a query optimizer must take.

4.6 Optimizations

In Section 4.5, we showed that re-encoding is necessary for a global dictionary when using a fixed-size code representation for multi-version encoding schemes in order to avoid running out of version numbers. However, re-encoding cannot be done incrementally if we require that a given value is encoded using the same versioned code that is also

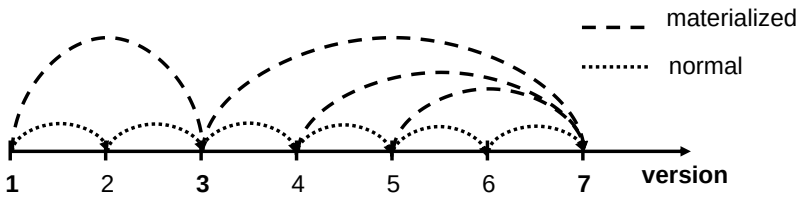


Figure 4.10: Materialization strategy for stable versions

stored in the global dictionary.

In this section, we present a solution that enables incremental re-encoding when using a global dictionary. The basic idea is that the global dictionary can hold more than one code version for a given value. Thus, an entry in the global dictionary has the form *value*, $code_1$, ..., $code_n$ where n is constant for all entries of the global dictionary. One such a code column $code_i$ $(1 \leq i \leq n)$ contains only codes that are greater equal than a given version number, which is called *stable version* of that code column. Figure 4.9 shows a global dictionary that has two code columns: code column $code_1$ has stable version 1, and the other code column $code_2$ has stable version 3.

A column of a table that uses a global dictionary with different stable versions for compression must specify which stable version (i.e., which code column) it uses for compression. In Figure 4.9 both tables 'Customer' and 'Supplier' use code column $code_1$ (i.e., stable version 1) for compressing the values of their column 'City'. If two compressed columns agree on the same stable version, they use the same versioned code to compress a given value (e.g., the versioned code $(2, 1)$ compresses the value 'New York' in both tables of Figure 4.9). Therefore query operations that only require equality can be executed without mapping.

Re-encoding a column of a table that uses a global dictionary with stable versions can be done independently from any other column that uses the same global dictionary. However, re-encoding is only allowed from one stable version to another more recent stable version to enable efficient decoding and query processing operations as we will show later in this Section. For example, re-encoding the column 'City' of table 'Customer' in Figure 4.9 is allowed if the target version is the stable version 3. In order to minimize the costs of re-encoding, version mappings tables can be materialized that map codes from all versions that are in between two subsequent stable versions to the more recent stable version. Figure 4.10 shows the materialized mapping tables (as dashed arrows) created for a global dictionary that has three stable code versions 1, 3, and 7 (shown in bold) while the current version of the dictionary is 7. Compared to a dictionary that does not have stable versions the size of the materialized mapping tables is much smaller since the mapping distances that a mapping table spans are not constantly growing.

If a stable version of the global dictionary is not used anymore to compress any column (i.e., all columns use more recent stable versions), the code column can be reused to create a new stable version. For example, assume that in Figure 4.9, no table uses the stable version 1 of the global dictionary anymore because both columns have been re-encoded and use stable version 3, then the code column $code_1$ of the global dictionary

can be overwritten. However, creating a new stable version makes only sense, if the global dictionary contains versioned codes in a version that is more recent than the latest stable version.

Re-encoding a column to a more recent stable version makes the execution of query operations that need order less expensive since the distance between the minimum and the maximum code version within that column decreases. Moreover, as less different code versions are used for compressing a certain column, the cache efficiency of mappings increases since less version mapping tables need to be loaded in the CPU caches.[4]

However, set operations or equi-joins that combine columns that use different stable versions must map the codes of these columns to a common (stable) version. These mappings can be done efficiently if materialized mappings between stable versions are provided as discussed before. For example, in order to execute an equi-join over two columns where one uses stable version 1 and the other stable version 7 of the global dictionary in Figure 4.10, we must map all codes of the column with stable version 1 to stable version 7. The maximal mapping distance in this example is when codes in version 1 or 2 are mapped to codes in version 7 which requires two version mapping tables to be applied subsequently (e.g., $2 \rightarrow 3$ and $3 \rightarrow 7$ for codes in version 2). For all other codes in version ≥ 3, only one version mapping table needs to be applied. In order to avoid the mapping completely for these operations, columns which are often accessed together should be re-encoded at the same time such that they always can use the same stable versions.

If a new value should be inserted into a table that uses a global dictionary with stable versions, the new code is written to all code columns of that entry. For example, in order to insert a new tuple into table 'Customer' of Figure 4.9 with the value 'Istanbul' for the attribute 'City', we insert that value into the global dictionary after 'Hong Kong' and generate the versioned code $(5, 4)$. This code is then written to both code columns ($code_1$ and $code_2$) of that new entry. If later on another column with stable version 1 or 3 needs to encode the value 'Istanbul' again, the global dictionary returns the code $(5, 4)$.

Decoding versioned codes of a column that has a certain stable version is a lookup of that versioned code in the global dictionary using the code column that has the same stable version. For example, to decode the column 'City' of table 'Customer' in Figure 4.9, we would need to execute the lookup on code column $code_1$ since it has the same stable version. In order to decode a column of a query result, we can also use a stable version for the lookup. If the column in the result contains codes from only one column of a table then it has the same version as this column. Moreover, if the query result is produced by a set operation (e.g., union), this operation needs to agree on a common stable version when producing its output anyway.

Moreover, for decoding we must build an index over all code columns (i.e., stable versions) of the global dictionary. The naive approach would be to build one decode index per stable version. However, the costs for storing and maintaining would increase by factor n when using n stable versions. Therefore, we only build one decode index for

[4]Cache efficiency is an important aspect to optimize algorithms for modern CPUs [84].

all stable version and store only the most recent stable version in the offset vector of the leaves. This can be done as follows for the decode strategies shown in Section 4.5.3:

- For the mapping-based indexing strategy, in which the decode index holds codes in the most recent version, we can use the most recent stable version to build the index.

- For the guideline-based indexing strategy, in which the decode index holds different versions of the same code, the different stable versions can be used directly as input to build the decode index.

- For the fully-indexed version, in which the decode index contains the minimal code per leaf and version, we need to sequentially scan all stable version of a leaf.

4.7 Evaluation

This section shows the results of our performance experiments with the prototype of our global dictionary that uses multi-version sequential encoding as discussed in Section 4.5.

We executed three experiments: The first experiment (Subsection 4.7.1) shows the efficiency of query processing on multi-versioned codes. The second experiment (Subsection 4.7.2) presents the costs for version mapping that is used during query execution and re-encoding. The third experiment (Subsection 4.7.3) examines the costs of encoding string values and decoding multi-versioned codes using a global dictionary.

We implemented the data structures in C++ (using gcc 4.1.2) and optimized them for a 64-bit Suse Linux Server (kernel 2.6.18) with two Intel Xeon 5450 CPUs (each having four cores) and 16 GB of main memory. All experimental results reported here are using non-parallelized algorithms. However, we discussed the potential for parallelization in the individual sections of this and the previous chapter. We tested parallel implementation of the algorithms (mainly using partitioning) which resulted in an increased performance. We do not report these numbers here as the dictionary should not consume many cores in a database system and leave the computing power to other parts of the system.

In order to generate meaningful workloads, we used the same string generator as in the experiments of the previous chapter (see Section 3.6). That string generator allows to tune different parameters like the number of strings, string length, alphabet size, the distribution, and others. We did not use the TPC-H data generator *dbgen*, for example, because most string attributes either follow a certain pattern (e.g., the customer name is composed of the prefix 'Customer' and a unique number) or the domain size of such attributes is small (e.g., the name of a country). Thus the data generated by *dbgen* does not allow us to generate workloads that let us analyze our global dictionary with workloads that have certain interesting properties. We will show the properties of the workloads that we generated for each experiment individually.

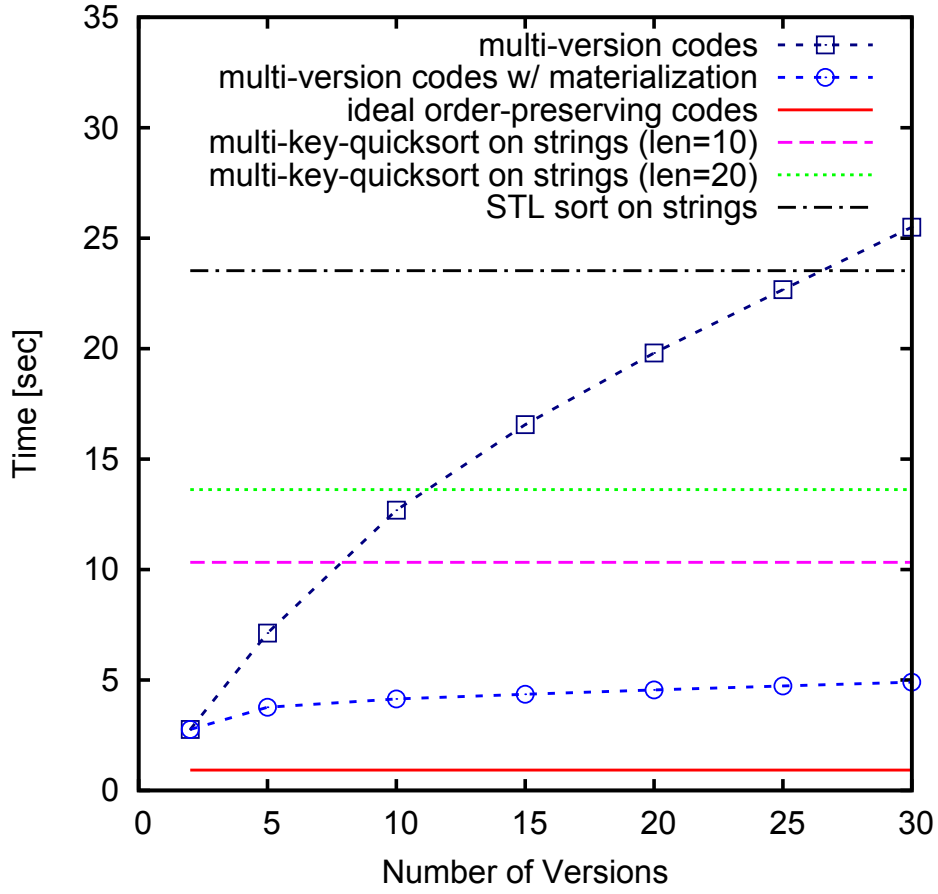


Figure 4.11: Sorting costs

4.7.1 Query Processing

In this experiment, we analyze the efficiency of multi-versioned codes for executing a sort operation as an example for a typical query operation that requires order. We compare the costs of the sort operation for multi-versioned codes using different numbers of versions in the input to the costs for sorting strings and sorting ideal order-preserving codes. In this experiment, we do not analyze operations that do not require order (e.g., equi-selection, union) since these operations do not need to pay the costs for mapping in most cases and thus have similar performance as on ideal codes. Only if set operations need to process inputs that have different stable versions, mapping becomes necessary. The costs for mapping are analyzed in detail in the next experiment using different materialization strategies.

For this experiment, we first generated unordered string workloads using different characteristics for the string length (10, 20), for the sizes of the workload ($1M$, $10M$), and for the frequency distribution of each string (uniform distinct, zipf). Afterwards, we split each of these workloads into $2 - 30$ bulks of the same size and encoded these bulks using multi-versioned sequential codes (i.e., when encoding the workload using n bulks the encoded workload contains n different versions). Moreover, we also encoded each string workload the same way as before using an order-preserving encoding scheme that leaves equi-distant gaps as baseline. All encoded workloads use a fixed-size physical representation of 64 bit per code while multi-versioned codes use 8 bit thereof for the version number. While multi-version codes could encode the workload using a much smaller fixed-size physical representation, the encoding scheme which leaves gaps requires 64 bit per code to avoid re-encoding for this workload.

Finally, we executed different sort algorithms for sorting the string workloads as well as for their encoded counterparts, which use multi-version codes as well as ideal order-preserving codes:

- For sorting the string workloads, we used the sort algorithm implemented in the gcc STL library (i.e., quick-sort [51]) as well as an optimized multi-key quick-sort [14].
- For sorting multi-versioned codes, we first mapped the encoded workload into the most recent version (using no materializations as well as ideal materializations) and then sorted multi-versioned codes using the sort algorithm implemented in the gcc STL library.
- For sorting ideal order-preserving codes, we also used the sort algorithm implemented in the gcc STL library.

Figure 4.11 shows the results for sorting $10M$ distinct strings of length 20 (and for multi-key quick sort also string length 10) as well as for sorting the 64 bit encoded counterparts. Other workloads produce similar results while those workloads with distinct strings are the worst-case for multi-versioned codes since the size of the version mapping tables is close to their upper bounds. The x-axis of Figure 4.11 shows the number of versions in the encoded workloads that use multi-version sequential codes and the y-axis

shows the time that is needed to sort all workloads. While sorting strings and ideal order-preserving codes is independent of the version number, we can see that for an increasing number of versions in the input workload, the costs for sorting the multi-version encoded workloads increase. For example, the time for sorting $10M$ multi-versioned codes with 5 versions is only a factor 4 slower than ideal order-preserving codes, while sorting $10M$ strings is a factor $10 - 24$ slower. When the number of versions increases to 10, sorting multi-versioned codes is also factor 10 slower. However, when using an ideal materialization strategy, which only needs one version mapping table to map each code version to the most recent one, the costs for sorting multi-version codes are growing much slower (i.e., they are faster than sorting strings in all cases). Moreover, for sorting strings multi-key quick-sort is faster than the standard quick-sort. However, the costs of multi-key quick-sort depend on the string length while all other sort operations are independent of the string length.

Consequently, operations which need order can leverage multi-version codes efficiently if either the mapping distance is small (because the input only contains codes in versions that are close together) or materializations exist that enable efficient mappings.

4.7.2 Version Mapping

In this experiment, we show the costs for the version mapping using different materializing strategies. The goal is to show the costs for these strategies using bulks that have a constant size but the mapping needs to span a different number of versions.

For this experiment, we used the encoded distinct workload of the previous experiment of size $10M$ that uses multi-version sequential encoding with 20 versions (i.e., 20 bulks were used to encode the data). Thus, each encoded bulk contains $10m/20 = 500k$ distinct codes whereas all codes are in one particular version (i.e., all codes in bulk 0 have version 1). Again, using the encoded workloads for distinct strings shows the worst-case size for multi-versioned codes since all version mapping tables have a size close to their upper bound. In practice, the time and memory spend for the mapping will be less.

Figure 4.12 shows the costs of mapping individual bulks to the most recent version 20. The x-axis shows the bulk number and the y-axis shows the time used for the mapping. For example, when mapping bulk 0 which is in version 1 to version 20 the mapping function must span 20 versions which results in different costs depending on the materialization strategies:

Plain This is the base case where no materialized version mapping tables exist (i.e., all version mapping tables only span one version number).

Full This is the ideal case, a materialized mapping exists that allows us to map codes from a given source to any target version using only one mapping table. However, this strategy has the highest costs for storing the mapping tables.

Stable This strategy simulates stable versions where the code versions that are between two stable versions can be mapped to the more recent one using only one mapping

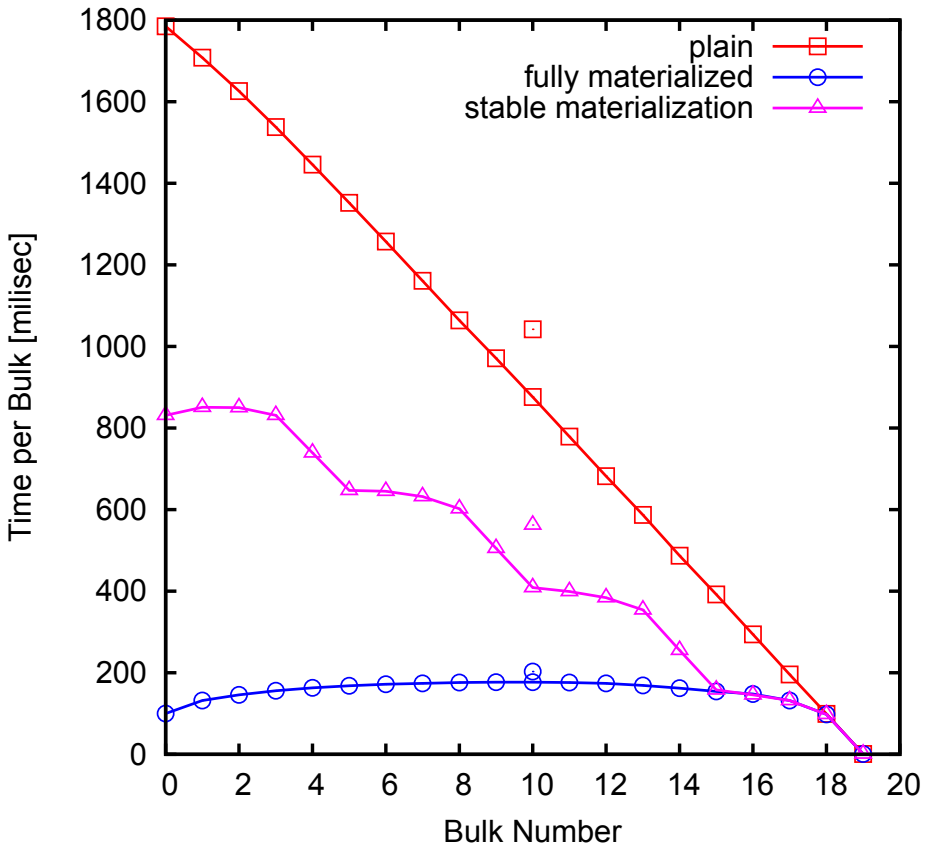


Figure 4.12: Mapping costs

(as shown in Figure 4.10). For the experiment, we create a stable version every 5 bulks (i.e., version 5, 10, 15, and 20 are stable versions).

While the time for the fully materialized strategy is almost independent of the bulk number, it has the highest storage costs: The mapping tables which map versions $1-19$ to the target versions 20 use $\sim$ 206 MB, while mappings to all possible target versions would use $\sim$ 3 GB.[5] As mentioned before, these sizes represent the upper bound (i.e., in practice the storage overhead will be much lower).

Compared to the fully materialized strategy, the stable materialization uses only $\sim$ 216 MB. The stabilization has only a small overhead over directly mapping to the target version ($\sim$ 206 MB) but in order to be prepared to sort at any point in time, the fully materialized strategy uses up to $\sim$ 3 GB for mapping tables. However, with stable materialization, the time for the mapping depends on the number of versions that the mapping function must span whereas mapping codes to the next stable version (e.g., from versions $\geq$ 15 to version 20) has the same performance as the fully-materialized strategy. Thus, for re-encoding it makes sense to constantly map codes to the next stable version. Moreover, query operations that must map codes to one common version can also use this strategy efficiently. Finally, the plain strategy uses the least amount of memory ($\sim$ 128 MB) but the time for the mapping function increases linearly with the number of versions that need to be spanned. Consequently, when using materialization carefully, mapping introduces only a minor overhead for query processing and re-encoding.

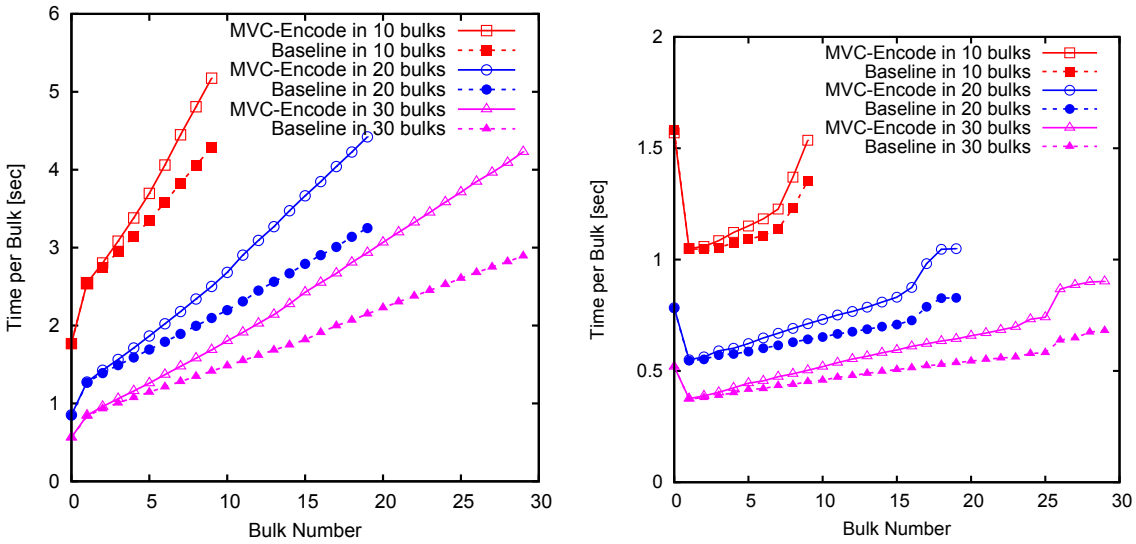
Moreover, for each strategy we additionally mapped a mixed bulk of size $500k$ which contains codes in versions 1 to 20 (uniformly distributed). The time for mapping that bulk is shown in Figure 4.12 by the single data points at the position of bulk 10 since the mixed bulks need to be mapped over 10 versions in average (which is comparable to bulk 10). However, for a mixed bulk more version mapping tables need to be applied which decreases cache efficiency and thus the mapping for mixed bulks takes a little longer as compared to mapping a bulk that consists only of codes in the same version.

4.7.3 Encoding and Decoding

In this experiment, we show the costs for encoding and decoding different workloads using a global dictionary that is based on multi-version sequential encoding as discussed in Section 4.5 and compare the costs to a global dictionary that uses an ideal order-preserving encoding scheme. As encoding index we use the CS-Prefix-Tree as described in Subsection 3.5.2 with a leaf size of 512 KB. For decoding, we used the strategies presented in Subsection 4.5.3.

For the experiment, we used the $10M$ string workload and a string length of 20 as described in Subsection 4.7.1. We split these workloads into 10, 20, and 30 bulks of the same size (i.e., the bulks sizes are $1M$, $500k$, and $333k$) and encoded them using the global dictionary. Figure 4.13(a) shows the time for a workload with only distinct

[5] Currently, a mapping table is stored uncompressed, i.e., two 64 bit numbers are stored per entry if we use a fixed-size code representation of 64 bit. Since mapping tables are stable (once they are created), using bit-compression could reduce the size significantly.



(a) distinct strings

(b) zipf distributed strings

Figure 4.13: Encoding costs

strings and Figure 4.13(b) for a workload where the frequency distribution of strings is a zipf distribution. The x-axis shows the bulk number that is encoded and the y-axis shows the time used for encoding a single bulk. Therefore the total time to encode the workload with 30 bulks is longer than with 10 bulks even though the time per bulk decreases.

For encoding these workloads, we see that the costs for later bulks increases since the number of entries in the global dictionary increase over time. Thus, when encoding bulk 10, the global dictionary already holds all values of bulks 0 to 9. Moreover, compared to ideal codes the costs for multi-versioned codes is growing faster since (1) versioned codes need to be mapped to the current version during encoding, and (2) version mapping tables need to be created. In the experiment, we did not materialize any mappings. This would increase the mapping performance during encoding but also increase the overhead to create these materialized mappings. However, materializing mappings can can be done asynchronously (i.e., not during encoding). In general, the workloads which use a zipf distribution have lower costs as the distinct workloads since much less new values need to be encoded per bulk (while the first bulk is the most expensive since all values are new and must be inserted into the global dictionary).

For decoding, we use the global dictionary that was created by encoding the distinct workload with 20 bulks mentioned before. We then decode each encoded bulk of this workload using the different decoding strategies presented in Subsection 4.5.3 and compare them to decoding ideal order-preserving codes that can use a clustered CSS-Tree [66] for decoding as baseline. Figure 4.14 shows the results:

- For the mapping-based indexing, we see that the decoding time increases with increased mapping distance (e.g., bulk 0 must be mapped from version 1 to 20).

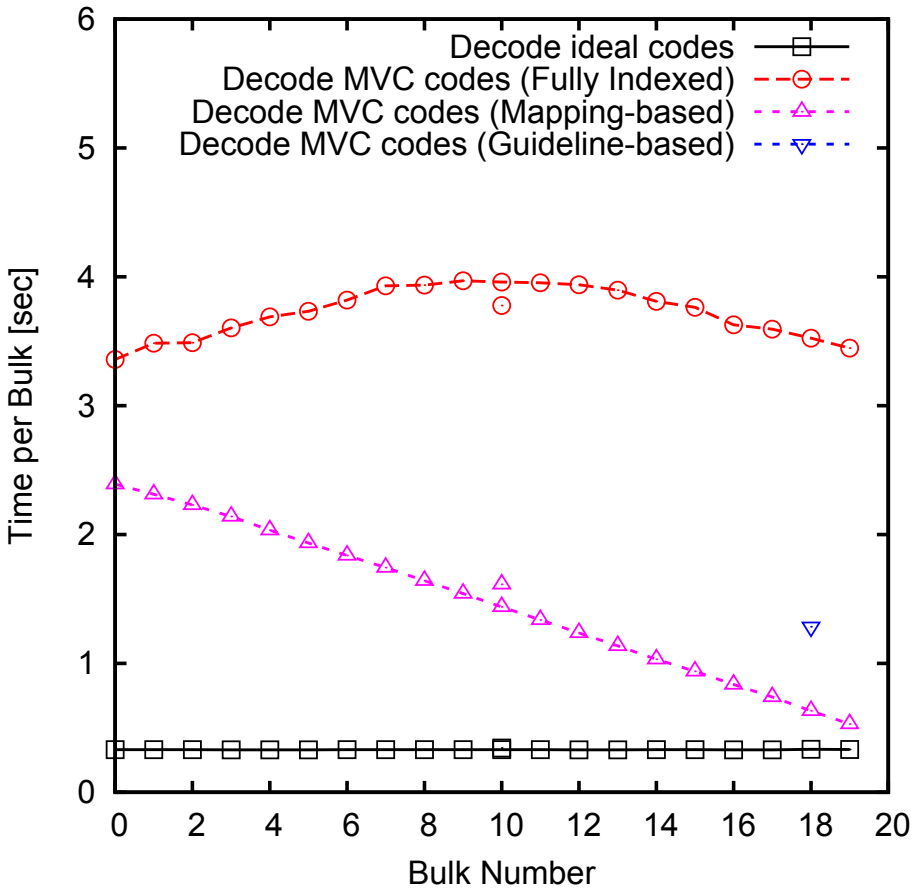


Figure 4.14: Decoding costs

This could be improved by using materialized mappings as presented in the previous experiment. However, if not done carefully the storage overhead for those mappings is significant.

- For the fully-indexed strategy, we see that the decoding time is almost independent of the bulk number since this strategy does not need to map for decoding. However, the costs for updating this index during encoding (which is not shown in Figure 4.13(a) and (b) grows linearly with the number of versions while the other strategies are almost independent of the number of versions). This strategy can help when using a lot of versions while we can not afford storing materialized mappings.
- For the guideline-based indexing, we see that this only works well if the bulk contains versions that are close to the versions that are used in the decode index as separators (e.g., version 18 in our experiment). For all other versions, the performance decreases significantly and is not shown in Figure 4.14. The extreme case of guideline-based indexing is to store codes in all versions per separator in each index node which is then the same as the fully-indexed strategy.

Moreover, decoding a mixed bulk of size $500k$ is also shown here as a single data point at bulk 10. The decoding time is similar to the one for bulk 10.

Note that this evaluation is deliberately missing any discussion of how many versions are to be expected in a real world scenario. This is for two reasons: 1) The number of versions highly depends on the underlying encoding scheme. As mentioned before, *multi-version encoding* can be combined with any order-preserving encoding scheme. New versions are created once the underlying scheme cannot encode incoming updates anymore. Furthermore it also depends on the underlying scheme and the used mapping functions how good a new version can fix issues in the code domain (e.g., if we can open a large enough gap to prevent the next updates to cause a new version again, we can reduce the overall number of versions). 2) The number of versions highly depends on the scenario. There is no representative scenario for a large enough class of workloads available in order to derive meaningful conclusions.

4.8 Conclusion

In this chapter, we introduced a new order-preserving encoding scheme called multi-version Encoding which does not make any assumptions on where to leave gaps for future insertions. Furthermore, it provides means to re-encode existing codes lazily. That way, compared to other order-preserving encoding schemes it can use a fixed-size physical representation while still enabling efficient updates. We showed that this encoding scheme can be applied for a dictionary-based compression scheme within column-oriented databases. The experiments show that multi-version encoding can improve the query execution for operations that require order. Moreover, query operations which do not

rely on order show similar performance as for any other non-order preserving encoding scheme that generates fixed-length codes.

As mentioned in Section 3.7, internal evaluation with SAP HANA DB showed that for current workloads order-preserving encoding does not provide enough performance gain to justify the development effort of a global order-preserving dictionary in a column store. Nevertheless, the idea of lazy re-encoding can be applied in other context where re-encoding is expensive. One example is improving the privacy in a cloud computing scenario [46].

5 Incremental Snapshots for Distributed Snapshot Isolation

The topic of this chapter is transaction handling, in particular, snapshot isolation in distributed databases. We enhance existing solutions for federated databases to make them suitable for the distributed scenario. Furthermore, this chapter presents a novel approach for snapshot isolation in distributed databases called *incremental*. Compared to the enhanced existing approaches, the incremental approach requires less knowledge about the workload while providing the same consistency guarantees than existing approaches. Performance experiments with the TPC-C benchmark show that the incremental approach provides better scalability with the number of nodes than the traditional approach. Moreover, it outperforms the other approaches for distributed snapshot isolation if no a-priori knowledge about the workload is available.

5.1 Motivation

Column stores show good performance in OLAP-style workloads [44]. Column stores traditionally do support transactional processing, but this use case was not the focus until now. Furthermore, column stores usually expect updates to be executed in bulks (e.g., during periodic replication). This has changed: the development efforts inside of SAP HANA DB [64] and also in other column stores aim for a wider applicability of their systems. One direction in which column stores are extending are OLTP-style workloads. Traditionally, updates are a weak point in column stores but ideas like delta indexing [48] or Positional Delta Trees (PDT) [45] alleviate this issue. Furthermore, real-time warehousing or operational business intelligence (BI) [26] get more and more interesting for enterprises and therefore the capabilities of column stores in the warehousing area are

required on live data. Thus, it seems reasonable to have only a column store as the main database in an enterprise instead of a row store combined with a periodically replicated warehouse organized as a column store. It follows that column stores need to improve the existing support for transactional guarantees and also for updates on single rows. In this chapter, we focus on performance improvements on the transaction handling, more concrete on distributed snapshot isolation.

The ultimate correctness criteria for transaction handling is a serializable schedule. Such a schedule does not contain any anomalies. As observed in [15], snapshot isolation allows certain non-serializable schedules, but this caveat seems to be tolerable for most applications. On the positive side, snapshot isolation can be implemented efficiently to enable a high transaction throughput. Snapshot isolation allows to execute read-only transactions in a non-blocking way. This is important for so-called operational BI workloads which involve long-running decision support queries on transacted data. Many commercial and open source database systems support snapshot isolation (e.g., Oracle, Microsoft SQL Server). Centralized snapshot isolation is understood quite well. Section 2.4 explained how it can be implemented in a column store.

In a distributed database system, transactions may read and update data items from multiple nodes. In that context, snapshot isolation has not been explored much. However, such distributed database systems are becoming increasingly important with the emerging trend to deploy databases in the cloud [16] and to keep all data in main memory [62]. For in-memory database systems, for instance, distributed snapshot isolation is critical because a database may not fit into the main memory of a single machine, but it is likely to fit into the aggregate main memory provided by a cluster of machines.

Today, the only commercial database system that supports distributed snapshot isolation and that we are aware of, is Oracle [61]. Unfortunately, no details of the implementation and isolation properties have been published by Oracle. In the academic community, distributed snapshot isolation was first studied in [69] (mainly in the context of federated databases) and it has recently gained attention in various different contexts; e.g., [8, 32, 49, 54]. Building upon that previous work, the goal of this chapter is to present new approaches to implement distributed snapshot isolation more efficiently.

In current systems, distributed transaction processing is usually implemented with a centralized coordinator. All transactions are coordinated by that central instance to guarantee consistency. But in many scenarios, central coordination is an unnecessary overhead. If the data can be partitioned in an optimal way for a given workload, many transactions only access a single node in the distributed landscape. Therefore, these transactions do not need a globally consistent view (or snapshot). One example for such a workload are multi-tenant databases, i.e., databases that are shared among multiple customers (i.e., tenants). Tenants only access their data, which leads to a natural partitioning.

In this chapter, we present two approaches, *pessimistic* and *optimistic*, that extend on the ideas for federated databases from [69] and make them work in a distributed scenario. As opposed to federated databases, in the distributed scenario, local transactions also

have to be considered.[1] Furthermore, we introduce a new technique, called *incremental*, that can be implemented transparently to the application, i.e., the application does not need to provide the database with additional knowledge about the transactions. This is in contrast to the other two approaches, *pessimistic* and *optimistic*, which only achieve optimal performance if the application provides a-priori knowledge about the transactions.

Experiments with the TPC-C benchmark show that for a low fraction of distributed transactions (i.e., when most transactions only access one node), the centralized solution is indeed a bottleneck. Solutions that utilize the difference between local and distributed transactions can improve performance by a factor of two in small setups. Even with high fractions (up to 70%) of distributed transactions, the proposed solutions are not worse than the centralized approach. Furthermore, *incremental* achieves that performance improvement without requiring any additional knowledge from the application.

The rest of this chapter is structured as follows:

- Section 5.2 presents related work.
- Section 5.3 introduces the background and terms used in this chapter.
- Section 5.4 discusses distributed snapshot isolation from a theoretical perspective. It presents:
  - a precise definition of distributed snapshot isolation
  - two anomalies that can arise in a distributed setting
  - a criteria that, if met by an implementation, guarantees correctness in the distributed setting
  - an informal proof and a more formal proof that the correctness criteria is sufficient.
- Sections 5.5 to 5.8 present four different approaches to implement distributed snapshot isolation:
  - *centralized*, the state of the art currently implemented in SAP HANA DB
  - *pessimistic* and *optimistic*, two approaches extending the ideas from [69] to distributed databases
  - *incremental*, a novel approach which requires less knowledge about a transaction than the other approaches while still providing better or the same performance
- Section 5.9 presents the results of the empirical evaluation of the different approaches.
- Section 5.10 concludes the chapter.

[1] [69] investigates the issue in federated databases and treats the individual nodes of the database as black boxes. Therefore only distributed transactions are considered by their system.

5.2 Related Work

The topic of distributed snapshot isolation received attention recently. The renewed interest started with early work on snapshot isolation in a federated setup [69] and also in a restricted setup [32, 49] or in the context of replication [8]. Snapshot isolation in column stores received only limited attention so far [80, 81].

5.2.1 Concepts of Snapshot Isolation

Similar to some ideas of this thesis is the idea of *generalized snapshot isolation* [34] where the begin of the transaction (i.e., the selection of the snapshot) is decoupled from the first actual operation. The incremental approach of this thesis uses a similar freedom by placing the begin of transaction on a certain node when it is suitable instead of when the first operation takes place. Different from [34] where the begin may only be earlier, the incremental approach of this thesis also allows the begin of transaction to be later (in terms of wall clock time) to ensure the most recent possible snapshot that is still correct according to the definition of distributed snapshot isolation (see Section 5.4).

Furthermore the definition of *session snapshot isolation* [33] applies to the system presented in this thesis. A system ensures session snapshot isolation if consecutive transactions of the same client (i.e., the same session) see what previous transactions wrote and snapshot isolation holds. This definition is useful since normal snapshot isolation does not require that a client gets the most recent snapshot. [33] call it *strong snapshot isolation* if every transaction gets the most recent snapshot. The goal of the incremental approach is similar since it tries to provide every transaction with the most recent snapshot that is still correct according to the definition of distributed snapshot isolation. In some cases this can even be a more recent snapshot than the definition of strong snapshot isolation in [33] would allow. This is because local transactions can be considered (i.e., a more recent snapshot can be used) as long as that view does not break global consistency.

Since the database on which the approaches are implemented and tested is organized as a column store, the work of Zhang and de Sterck [80, 81] on snapshot isolation on HBase, an open-source column store, is related. Their approach is close to what is identified as central coordination in this chapter since all information about transactions is kept in a bunch of (conceptually) centralized tables. The column store used in this thesis implements snapshot isolation with centralized coordination successfully for some time now.

5.2.2 Distributed SI in Restricted Setups

Some of the approaches in this thesis are inspired by ideas from Schenkel et al. [69]. In their paper, Schenkel et al. present algorithms to achieve snapshot isolation for federated databases. The databases are treated as black boxes and local transactions are not considered. For such a federation of snapshot isolation databases, they derive algorithms to achieve global serializability. Due to the black box approach, they have limited

possibilities but also do not need to care about local transactions (or more precisely, the federation layer is not aware of any local transactions). This leads to potential violations of the correctness criteria given in Subsection 5.4.3: in the optimistic approach in [69], the begins of distributed transactions are not ordered the same on all nodes and therefore the cross-anomaly (see Figure 5.3) can occur. This thesis extends their approaches to work in the distributed setting where local transactions have to be considered.

Optimizing for local transactions is important, for example [57] presents a system for a cloud database where transactions that share data from different transaction managers can only operate at a lower consistency level. Snapshot isolation is mentioned as future work for their system.

In [49], Jones et al. discuss concurrency control in partitioned main memory databases. They restrict the possible transactions to executions of predeclared stored procedures without user interaction which represents a higher dimension of a-priori knowledge than the one considered in this thesis. Throughput is improved by making use of wait cycles while a remote node executes its part of the current transaction. This is demonstrated using a similar experimental setup than used in this thesis.

Restricting the possible operations in order to simplify concurrency control is taken to an extreme in [32] where a fixed group of operations from small set of operators with limited connection among each other, called minitransactions, are submitted all at once. Another approach is taken in [31]: the database is automatically partitioned with the goal to minimized the amount of distributed transactions. In that sense, this thesis can be seen as an extension since the approaches presented in this thesis work best if the fraction of distributed transactions is low, but still cannot be neglected. One interesting aspect in [32] is the concept of hierarchical transaction managers. As future work, the ideas in this thesis could be extended in a similar way to have a hierarchy of transaction coordinators in order to reduce the pressure on a single central coordinator. This could for example be useful in the so called "multi-tenancy scenario" where the database is provided as a service and shared with multiple customers (i.e., tenants). If a tenant is distributed over multiple nodes, one could introduce a "tenant-global" transaction manager in order to avoid the need to go to the central coordinator for the transactions accessing the nodes of the tenant.

Another dimension, which has been investigated, is to relax the constraints of snapshot isolation. [73] introduce *parallel snapshot isolation* where the order of transactions is only maintained if the later transaction reads from the preceding transaction, e.g., the response is only propagated after the question has been propagated to other sites. As opposed to that, goal of this chapter is to provide exactly snapshot isolation without compromises in terms of the definition.

5.2.3 Snapshot Isolation in Replicated DBMS

The *pessimistic* approach is used in [9] to build a partial database replication protocol, i.e., on begin of a transaction, all potentially involved nodes are informed.

A similar idea to the incremental technique is used in [70] but instead of actually reconstructing the snapshot on other nodes, their system just opens dummy transactions

after each commit, which can then later be used to process requests from other nodes that need an older snapshot. They are more concerned about the aspect of replication[2] and less about the distinction between local and distributed transactions. In order to keep the nodes synchronized, they use group communication to commit all transactions on all nodes while the incremental technique uses a centralized coordinator to keep track of the global counters.

There are many solutions for concurrency control in replicated databases using group communication, for example [23, 33, 34, 50, 54, 63], or see [8] for an overview of snapshot-isolation-oriented approaches. But the issue of scheduling is different from the issue considered in this thesis since in these systems all data is fully replicated.

Our work differs in one important aspect from other work that uses snapshot isolation-based replicas but achieves a higher isolation level in the overall system (e.g. [23, 50]): our approaches minimize the information about the transactions shipped between nodes (e.g., read- or write-sets). This is an essential design decision in our work and one of the reasons for the improved performance. At the same time, this also implies that our approaches cannot directly achieve a higher isolation level than what the underlying systems provide since that would require additional information.

Some approaches support partial replication (e.g. [36, 47]) but aim for serializability instead of snapshot isolation as concurrency requirement. Sprint [25] uses similar techniques, i.e., group communication and predefined transactions, and also aims for serializability. Relevant to the scenario in this thesis is that Sprint makes a distinction between distributed and local transactions. A local transaction can access data from other nodes by informing all other nodes using total order multicast. If the order of the operations is wrong on any node, the transaction is aborted. This is similar to the design decision made in this thesis not to monitor actual conflicts between transactions since this is expensive. Just as in Sprint, we abort a transaction as soon as the order of the transaction begin or commit operations is wrong.

5.3 Background

5.3.1 Architecture

The basic system used in this chapter is a distributed database that consists of nodes that implement snapshot isolation. The architecture is shown in Figure 5.1. It consists of several clients that access the database and includes a central coordinator. *Distributed* means that the database is hosted on multiple nodes, which can be virtually or physically separated. We have full control over all nodes and are able to implement new protocols within those nodes. This is in contrast to a federated system that was used by [69] in which the nodes operate autonomously and cannot be changed.

[2]Replication is not solved in this thesis, but support of eager replication is straight-forward.

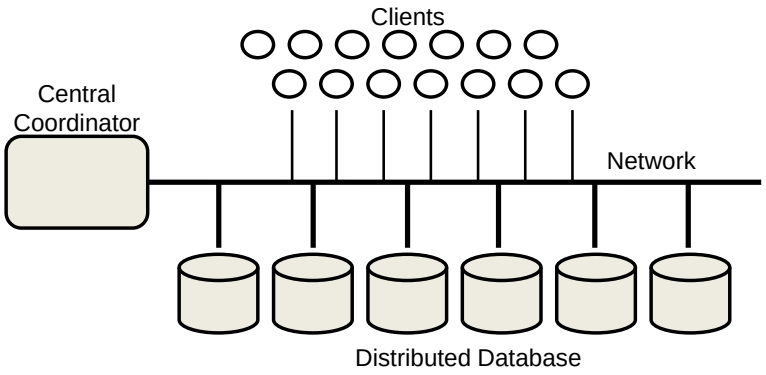


Figure 5.1: Conceptual architecture of the databse system

5.3.2 Expected Workload

The data is *partitioned* in such a way that most transactions only access data from a single node. For example, this is the case in the multi-tenancy scenario where one database system is shared across many clients (tenants). The data of such clients typically fits into one node. Additionally, some system data is shared across nodes. Thus, a few transactions require consistent access to more than one node or even the whole database, e.g., for updates to the mentioned system data. Another example for such a partitionable workload is the TPC-C benchmark. In that benchmark, the data can be partitioned (i.e., sharded) by warehouse, and most transactions access only data from one warehouse (i.e., one node). Thus, only a few transactions require data from more than one node.

To achieve proper isolation for those distributed transactions that access data from multiple nodes, the system provides distributed snapshot isolation. The basic idea of snapshot isolation is that every transaction gets its own (virtual) "copy" of the database to work with. In principle, a transaction gets never blocked by concurrent transactions.[3] At the end of a transaction, the database allows only transactions to commit if their write sets have no conflicts with other concurrent transactions [15].

Our system supports eager replication of data (e.g., by replicating the writes in the same transaction to multiple nodes or by using hardware solutions). Lazy replication (e.g., log shipping) is currently not supported. Investigating the conflict potential between lazy replication and distributed snapshot isolation is left as future work.

The approaches presented in this thesis use a central coordinator for certain tasks. Such a central coordinator is a single point of failure. To improve fault-tolerance any technique (e.g., replication) can be applied since the coordinator keeps only a limited amount of state. Studying fault-tolerance techniques is outside of the scope of this thesis. Furthermore, a central coordinator is also a potential bottleneck. For the approaches presented in this thesis, it is possible to distribute the central coordinator itself over

[3]Some implementations of snapshot isolation use locking, in which case write transactions may get blocked.

multiple nodes to improve performance even more. But again, this is not part of this thesis.

5.3.3 Conceptual Database Model

We use a conceptual database model as presented in [4]. That model is intended for local systems but also works in our scenario since we assume a partitioned (i.e., eagerly replicated or not replicated) database. Below we add a few details to handle the peculiarities of the distributed scenario.

> The database consists of objects that can be read or written by transactions. Each transaction reads and writes objects and indicates a total order in which these operations occur; thus, our transactions are sequential in nature. An object has one or more versions. Transactions interact with the database only in terms of objects; the system maps each operation on an object to a specific version of that object. The last operation of a transaction is a commit or abort operation to indicate whether the transaction's execution was successful or not; there is at most one commit or abort operation for a transaction. When transaction x commits, each version o_x created by x becomes a part of the committed state and we say that x *installs* o_x.
>
> A schedule S over a set of transactions consists of two parts—a partial order of events E that reflects the operations (e.g., read, write, abort, commit) of those transactions, and a version order, $\ll$, that is a total order on committed object versions. [4][4]

To simplify the discussion, we also model the begin of a transaction. In [4] this operation is implicit in the first read or write operation of a transaction. We model this explicitly, i.e., the first operation of a transaction on a node is a begin operation.

The following *terms* refer to different kinds of transactions: A *local transaction* is a transaction that accesses (reads or writes) only data from a single node. A *distributed transaction* is a transaction that accesses (reads or writes) data from more than one node. Such a distributed transaction is partitioned into subsets of operations that are executed on individual nodes. These subsets are not independent, thus the semantics of a transaction applies to the entire set of operations (e.g., if the transaction is aborted, all its operations on all nodes have to be reverted). Furthermore, we extend the model from Adya [4] for distributed transactions as follows: a distributed transaction has begin and commit operations on every node that it accesses, but again, a distributed transaction either commits on all involved nodes or it aborts on all nodes.

Consequently, a *local schedule* is a schedule that contains the operations from transactions executed on a single node. The *global schedule* is a schedule for all operations from both local and distributed transactions on all nodes.

In terms of *notation*, this thesis uses x, y, z for the transactions currently discussed, s, t, for transactions that are required to complete the schedule but are not in the

[4]Pages 34ff in [4] adapted to our notation.

focus and i, j for nodes. b_x represents the begin of transaction x while c_x represents the commit of transaction x. A superscript on an operation means that the given operation happens on the specified node, e.g., b_x^i is the begin of transaction x on node i. $N(x)$ contains all nodes accessed by transaction x, i.e., all nodes on which x has accessed an object. $SN(x, y)$ contains all nodes accessed (shared) by both transactions x and y, i.e., $SN(x, y) = N(x) \cap N(y)$. A transaction x is before a transaction y on node i if $c_x^i < b_y^i$. We use $x < y$ as a short notation. Two transactions x, y are concurrent on node i if $b_x^i < c_y^i$ and $b_y^i < c_x^i$. We use $x||y$ as a short notation.

5.3.4 Local Snapshot Isolation

The implementation of snapshot isolation in a centralized, single-node system has been excessively studied in the past, and many major database products already support snapshot isolation. Therefore we can build on the fact that a system properly implements local snapshot isolation, i.e., a read of transaction x is mapped to the corresponding write of the transaction that committed last before the begin of transaction x on one node. Therefore, for our considerations, only the begin and commit operations matter. A transaction x does not read from transaction y if $b_x < c_y$. Note that aborted transactions do not change the scheduling or the correctness of a snapshot isolation scheme [15]. Thus this thesis will not consider aborts caused by the user or application.

In order to show that a schedule is correct according to local snapshot isolation, we use the formal definition of snapshot isolation from [4].[5] For that formal definition, we first need to define dependencies between transactions. Then, we construct the so-called *direct serialization graph (DSG)* and *start-ordered serialization graph (SSG)* using these dependencies. Finally, a set of phenomena is given that must not occur in a schedule that is correct according to local snapshot isolation.

Dependencies between transactions:[6]

- Directly Read-Depends. A transaction y directly read-depends on transaction x (denoted by $x \xrightarrow{wr} y$) if x installs some object version o_x and y reads o_x.
- Directly Anti-Depends. A transaction y directly anti-depends on transaction x (denoted by $x \overset{rw}{\dashrightarrow} y$) if x reads some object version o_z and y installs o's next version (after o_z) in the version order. Note that the transaction that wrote the later version directly anti-depends on the transaction that read the earlier version.
- Directly Write-Depends. A transaction y directly write-depends on x (denoted by $x \xrightarrow{ww} y$) if x installs a version o_x and y installs o's next version (after o_x) in the version order.

[5] The formalization from [4] also covers other isolation levels and therefore is more general than just snapshot isolation. This leads to increased complexity in the description if just used for our scenario. We did not attempt to simplify the description or use another formalization because this formalism helped us to discover certain flaws in earlier versions of our approaches.

[6] Pages 40ff and 78ff in [4]

- Start-Depends. A transaction y start-depends on x (denoted by $x \xrightarrow{s} y$) if $c_x < b_y$, i.e., if it starts after x commits.

Definition of the DSG:[7] We define the direct serialization graph arising from a schedule S, denoted DSG(S), as follows. Each node in DSG(S) corresponds to a committed transaction in S and directed edges correspond to different types of direct conflicts. There is a read-/ write-/ anti-dependency edge from transaction x to transaction y if y directly read-/ write-/ anti-depends on x.

Definition of the SSG:[8] For a schedule S, SSG(S) contains the same nodes and edges as DSG(S) along with start-dependency edges.

Definition of Local Snapshot Isolation:[9] First, we define a set of phenomena that can occur in a history. Then we define snapshot isolation as the absence of these phenomena.

- G1a: Aborted Reads. A schedule S exhibits phenomenon G1a if it contains an aborted transaction x and a committed transaction y such that y has read some object modified by x.
- G1b: Intermediate Reads. A schedule S exhibits phenomenon G1b if it contains a committed transaction y that has read a version of object o written by transaction x that was not x's final modification of o.
- G1c: Circular Information Flow. A schedule S exhibits phenomenon G1c if DSG(S) contains a directed cycle consisting entirely of [read- and/or write-][10]dependency edges.
- G-SIa: Interference. A schedule S exhibits phenomenon G-SIa if SSG(S) contains a read- or write-dependency edge from x to y without there also being a start-dependency edge from x to y.
- G-SIb: Missed Effects. A schedule S exhibits phenomenon G-SIb if SSG(S) contains a directed cycle with exactly one anti-dependency edge.

A schedule S consisting of committed transactions executes under snapshot isolation iff G1(a-c) and G-SI(a-b) are disallowed. [4]

Examples of how this definition is used follow in the next section, for example in Figure 5.2.

[7] Page 43 in [4]
[8] Page 79 in [4]
[9] Pages 48ff and 80ff in [4]
[10] Adya [4] defines the term *depends* as read- or write-depends. Thus the term *dependency edge* refers only to read- and write-dependency edges. If anti-dependency edges are also included, it is called *conflict-depends*.

5.4 Distributed Snapshot Isolation

This section defines distributed snapshot isolation based on the existing definition of local snapshot isolation. Furthermore, it presents anomalies that can occur in a distributed setting if only local snapshot isolation is enforced on the individual nodes, thus motivating the need for a global coordination. The section concludes with a set of correctness criteria that is sufficient to enforce schedules that do not suffer from the presented anomalies (i.e., are correct according to distributed snapshot isolation).

5.4.1 Definition

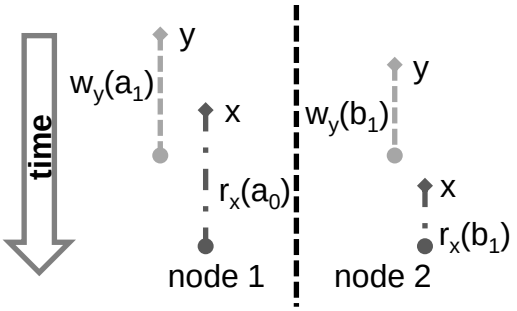
In the following, the existing definition of local snapshot isolation is extended to the distributed setting. The local transactions from one node and the subsets of operations of the distributed transactions accessing data on the same node form a local schedule. The correctness of this schedule is enforced by the local node (i.e., local snapshot isolation holds).

Definition 1 (Distributed Snapshot Isolation)**.** *A set of multiple locally correct schedules is correct under distributed snapshot isolation if and only if there exists a view-equivalent global schedule that combines all local schedules and that global schedule is correct according to local snapshot isolation (i.e., it does not contain any of the phenomena G1(a-c) and G-SI(a-b) presented in Section 5.3).*

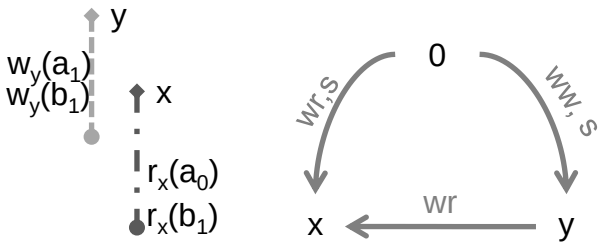
This definition is similar to what Lin et al. [55] call *1-copy-snapshot-isolation* but is not restricted to replicated databases. Other definitions in the literature, e.g. [33, 34], only work for replicated databases.

The global schedule has to contain all operations from committed transactions from all nodes. And the schedule has to be view-equivalent, i.e., the read operations have to return the same result and the result of executing the global schedule has to be equal to the union of the databases resulting from executing the local schedules.

According to Definition 1, the global schedule has to be view-equivalent to the local schedules. In theory, this implies that in some cases the order of certain operations in the local schedules can be changed, e.g., if transaction x does not read or write any object written by transaction y, the begin of transaction x can be moved to either before or after the commit of y without changing the view or the effects of the transactions. Thus there may be more than one view-equivalent global schedule for a set of local schedules. From a practical point of view, this "freedom" does not help much since it always requires to check the reads-from-relationship [18], i.e., whether or not a transaction x reads or writes something read or written by transaction y. As argued in [69], this relationship is expensive to monitor, especially in a distributed setting. Thus, the correctness criteria in Subsection 5.4.3 and the approaches presented in this chapter only allow or construct schedules where the local order of the operations is not changed.

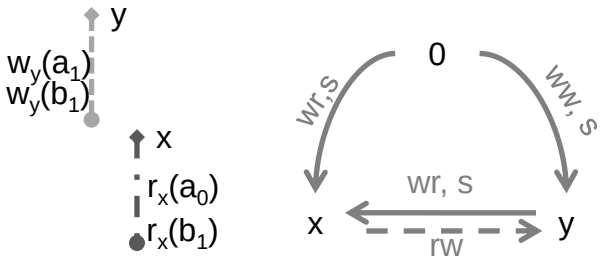


(a) Local Schedules



(b) Global Schedule 1

(c) Start-ordered Serialization Graph 1



(d) Global Schedule 2

(e) Start-ordered Serialization Graph 2

Figure 5.2: Serial-concurrent-pattern example

5.4.2 Anomalies in the Distributed Setting

In addition to the anomalies in the local setting [15], two more anomalies exist in the distributed setting if the individual nodes only enforce local snapshot isolation. One anomaly that can occur was identified in [69]. This anomaly is called *serial-concurrent-pattern*. As shown in Figure 5.2, the anomaly occurs if a transaction x runs concurrent to another transaction y on node 1 (i.e., $x||y$) and serial to y on node 2 (i.e., $x < y$). Therefore, transaction x does not read what transaction y writes on node 1 but reads what y wrote on node 2. This happens if transaction y commits before transaction x starts on node 2.

There are two relevant types of global schedules for the example in Figure 5.2(a): the two transactions x, y are concurrent (one possibility is shown in Figure 5.2(b)) or serial (shown in Figure 5.2(d)). The other possibilities for a global schedules in which the two transactions are concurrent do not change anything relevant for the discussion (i.e., the SSG is the same for all these possibilities). The other possibility in which the two transactions are serial is not relevant for the discussion because then transaction x would read data that is not written yet.

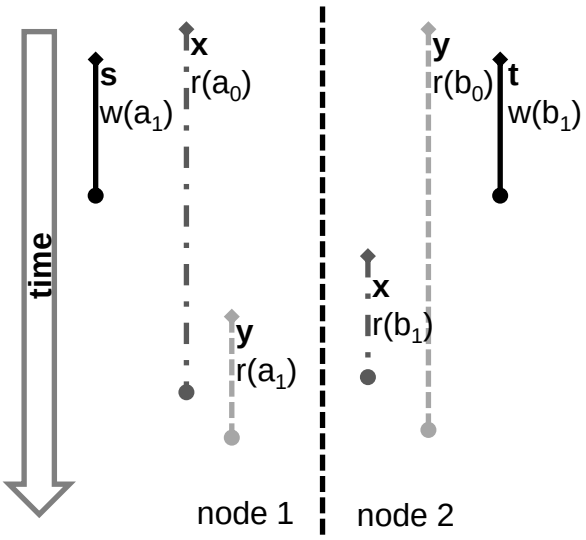
Neither of the two types of global schedules is correct according to snapshot isolation. To see why, we construct the Start-ordered Serialization Graph (SSG) according to [4] as quoted in Subsection 5.3.4 for the two types and find a phenomena that must not occur in a correct schedule.

Figure 5.2(c) shows the Start-ordered Serialization Graph (SSG) for the concurrent case. Transaction 0 in the Figure is the initial transaction that created the initial database. The formal definition of snapshot isolation requires inter alia that there is no interference (G-SIa on page 80 in [4]). It follows that if there is a read- or write-dependency (wr/ww) edge from transaction y to transaction x, there has to be a start-dependency (s) edge from transaction y to transaction x. This is not the case in Figure 5.2(c) since there is a read dependency from transaction y to transaction x (transaction x reads the state of object b that transaction y wrote) but there is no start dependency from transaction y to transaction x (because transaction x begins before transaction y commits).

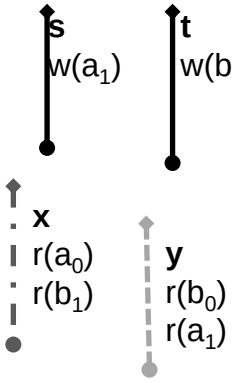
Figure 5.2(e) shows the Start-ordered Serialization Graph (SSG) for the serial case. Now the schedule contains missed effects (G-SIb on page 80 in [4]) because there is a cycle consisting of exactly one anti-dependency edge: x directly read-depends on y because it reads the version of b that x wrote but at the same time y directly anti-depends on x because x reads the version of a that is immediately before the version that y installs.

Thus, none of the possible global schedules is correct according to snapshot isolation. This is because the same distributed transaction reads different snapshots on different nodes. The approaches presented in the following sections will provide mechanisms to avoid such situations.

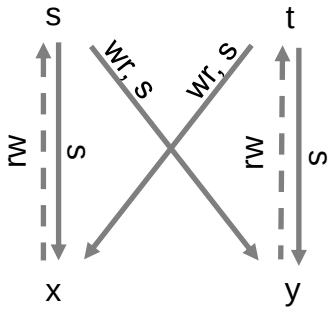
Another anomaly that can arise in a distributed setting was not considered in the literature before. Figure 5.3 presents this anomaly, called *cross-anomaly*. In the example, the distributed transaction x does not read from the local transaction s on node 1 and reads from the local transaction t on node 2. At the same time, the distributed



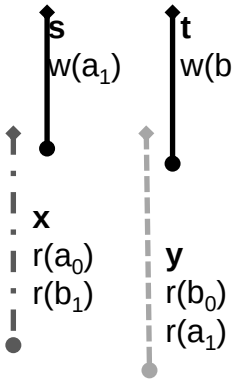
(a) Local schedules



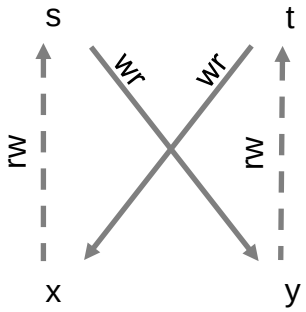
(b) Global schedule 1



(c) SSG for schedule 1



(d) Global schedule 2



(e) SSG for schedule 2

Figure 5.3: Cross-anomaly example

transaction y reads from transaction s on node 1 and does not read from transaction t on node 2. This implies that $x||s$, $x||y$ and $s < y$ on node 1 and $y||t$, $y||x$ and $t < x$ on node 2. Figure 5.3(b) shows one way to combine the local schedules and Figure 5.3(c) shows the corresponding SSG (edges from the initial transaction 0 are omitted for readability). This variant is not a correct schedule according to snapshot isolation because it has missed effects (G-SIb on page 80 in [4]): The SSG contains a directed cycle with exactly one anti-dependency (rw) edge (shown as dashed arrows). This is the case for any schedule with $s < x$ or $t < y$.

In order to get rid of this cycle, one could rearrange the global schedule as shown in Figure 5.3(d). Again, Figure 5.3(e) shows the corresponding SSG. In the new schedule, the start-dependency edges that caused the cycle disappeared. There are no more s-edges from transaction x to transaction t and from transaction y to transaction s. But, at the same time, the start-dependency edges from transaction x to transaction s and transaction y to transaction t disappeared as well. Now the schedule contains interference (G-SIa on page 80 in [4]) because there are read-dependency edges from transaction s to transaction x and transaction t to transaction y without start-dependency edges. This problem occurs in any schedule with $s||y$ or $t||x$.

Note that a mixture of the two discussed global schedules does not work either since in that case both issues occur at the same time: we did not get rid of the missed effects completely while introducing interference. Furthermore, any schedule where $x < t$ or $y < s$ does not make sense because then transaction x (respectively transaction y) reads a version that is not yet written. These three classes cover all possible schedules, thus none of the possible variants are correct according to distributed snapshot isolation.

Furthermore, it is important that transactions s and t are local transactions (or at least only access nodes outside of the scope of the example). If transaction s or t would access the other node, the example would simplify to the serial-parallel anomaly presented before.

The cross-anomaly is only an issue if the transactions x, y actually read something that transactions s, t write. However, as mentioned before, monitoring the reads-from relationship is expensive. Thus, in order to avoid the *cross-anomaly*, the system has to make sure that the order of the begin-of-transaction of distributed transactions is the same on all involved nodes. This anomaly was not considered in [69] since the local transactions are not visible in a federated system. The approaches presented in the following sections will avoid this anomaly, including enhanced versions of the two approaches presented in [69].

5.4.3 Correctness Criteria

As a first step towards implementing a protocol for snapshot isolation in distributed databases, we develop a set of sufficient correctness criteria that serves as guideline for the next steps, i.e., all the approaches to implement distributed snapshot isolation presented in this paper are based on alternative ways to meet these rules.

Assuming that the data is partitioned (or eagerly replicated) and the local schedules are correct according to snapshot isolation, a sufficient criteria is that there has to be

a partial order of all begin and commit operations of all distributed transactions. This partial order has to be adhered to on all local nodes that a distributed transaction accesses.

If the following rules are met by schedules that are correct according to local snapshot isolation, then we can show that distributed snapshot isolation is guaranteed:The following rules show how the partial global order is constructed:

$$\exists i : c_x^i < c_y^i \rightarrow c_x < c_y \tag{5.1}$$
$$\exists i : b_x^i < c_y^i \rightarrow b_x < c_y \tag{5.2}$$
$$\exists i : c_x^i < b_y^i \rightarrow c_x < b_y \tag{5.3}$$
$$\exists i : b_x^i < b_y^i \rightarrow b_x < b_y \tag{5.4}$$
$$c_x < c_y \rightarrow \forall j \in SN(x,y) : c_x^j < c_y^j \tag{5.5}$$
$$b_x < c_y \rightarrow \forall j \in SN(x,y) : b_x^j < c_y^j \tag{5.6}$$
$$c_x < b_y \rightarrow \forall j \in SN(x,y) : c_x^j < b_y^j \tag{5.7}$$
$$b_x < b_y \rightarrow \forall j \in SN(x,y) : b_x^j \leq b_y^j \tag{5.8}$$

$SN(x,y)$ is the set of nodes that both transactions x and y access.

Rules (1)-(4) construct the partial global order and Rules (5)-(8) enforce it on the local nodes.

Note that it is not sufficient to require a set of combined rules of the form $\exists i : c_x^i < c_y^i \rightarrow \forall j \in SN(x,y) : c_x^j < c_y^j$ in all four variants of begins and commits. The intermediate step of the global partial order is necessary. To see why, consider three nodes with three transactions and the following order: on node 1: $x < y$, on node 2: $y < z$, and on node 3: $z < x$. Since there is always only one node that two transactions access together, the combined rules are trivially true. But from a global perspective there is a cycle since $x < y < z$ (transitive from nodes 1 and 2) and at the same time $z < x$ (on node 3). Thus only the complete set of eight rules above construct a partial global order in which all distributed transactions are registered and basically constructs the transitive closure over all distributed transactions on all nodes.

One small deviation from these formal rules is allowed: if there is no commit operation (either local or global) between two begin operations, the order of these begin operations may be changed. Consider this example: transaction x begins on node 1 and transaction y begins on node 2. Then transaction x also begins on node 2. If now transaction y also accessed node 1, the order of the begin operations is not the same on the two nodes. This is allowed if on one of the two nodes no commit happened between the two begin operations (because the two transactions read the same snapshot if no commit happened in between and therefore this does not influence the correctness).

Proof Based on the Schedules

Proof. To proof that the rules above are sufficient, we first construct a global schedule from a set of local schedules that adhere to the rules stated above. Then we show that

the constructed global schedule is view-equivalent to the local schedules and is correct according to snapshot isolation.

Construction of global schedule. In order to construct a global schedule from a set of local schedules, we do a (sorted) merge of the schedules. The begin and commit operations of the distributed transactions are fix-points. Since they are ordered the same on every node (some care is required in case of begins at the same time), the merge is straight forward and we can construct a partial order of the global operations. Based on this order (and by arbitrarily enforcing a total order consistent with the partial order), we can merge the local schedules as follows: merge all local operations in the local order from all local schedules that contain the global operation that is the first in the global order up until (and excluding) that global operation. Then, consume that global operation from all local schedules and add it to the global schedule. Continue this procedure with the next global operation in the global order until all global operations are consumed. At the end, add any remaining local operations in the local order to the global schedule.

Based on this global schedule, we now show that this construction leads to a global schedule that is view-equivalent to the local schedules. Then we show that it is valid according to snapshot isolation.

Schedule is view-equivalent. To show that the global schedule is view-equivalent, we have to show that a) all reads return the same values and b) the final state of the database is the same after executing the schedules, i.e., the last write to an object is the same. Since the begin and commit operations of distributed transactions are ordered the same on all nodes, the order in the local schedules is not changed when the schedules are combined according to the steps described above. This is because the process is a merge of "sorted" schedules. Furthermore since the data is partitioned (or eagerly replicated), we can distinguish between local and distributed transactions to show the view-equivalence: for a local transaction x the global schedule is view-equivalent since the order of the operations from the same node is not changed, and operations from other nodes cannot change data items accessed by the local transaction x. Thus, a local transaction x reads the same data in both the local and global schedule.

For a distributed transaction y, the argument is similar, but a bit more complex as is involves multiple nodes: Since the distributed operations are always the last to be inserted in the global schedule, all operations relevant for transaction y have already been executed from all nodes when b_y is inserted in the schedule. And if transaction y does access data from a node i, there is a begin operation b_y^i on that node i which implies that the procedure above does not insert local operations that occurred after the begin of transaction y on node i and that could change the view seen by transaction y. Thus, transaction y gets the same view in both the local and global schedule. Moreover, since the order of the commits is not changed, the last write in the local schedules is also the last write in the global schedule. Therefore the global schedule is view-equivalent for both local and distributed transactions.

Schedule is correct according to snapshot Isolation. In order to show that the constructed schedule is correct according to snapshot isolation, we need to show that a) every transaction reads the most recent version committed at the beginning of the transaction of every accessed data item and that b) two concurrent transactions do not write on the same data item. To show that, we again use that the data is partitioned (or eagerly replicated).

For the first part: every operation that changes a data object relevant to transaction x must have accessed at least one node that transaction x accessed (because the object is on that node). Since the order of operations is not changed, the local system will apply snapshot isolation and provide the transaction with the proper view.

For the second part, we observe that all transactions that potentially write to the same object (i.e., have at least one accessed node in common) are either serial on all nodes or parallel on all nodes since the order of begin and commit operations is the same on all nodes. Thus a write conflict in the global schedule can always be mapped to a write conflict in a local schedule. This cannot occur since the local schedules are correct according to snapshot isolation.

This concludes the proof that the constructed schedule is correct according to snapshot isolation. □

5.4.4 Proof Based on SSGs

Proof. To proof that the criteria given in this thesis ensures distributed snapshot isolation, we can work on the level of the SSGs. We first discuss how the SSG of the global schedule can be constructed based on the SSGs of the local schedules. Then we show that the SSG of the global schedule does not contain any of the relevant anomalies.

Construction of the global SSG. First, we construct the SSG of the global schedule. The SSG of the global schedule contains all read-/ write-/ anti-/ start-dependency edges of all local SSGs. Since the data is partitioned (or eagerly replicated), the global SSG does not contain any additional read-/ write-/ anti-dependency edges because all reads and writes occur on the local nodes. The global SSG may contain some additional start-dependency edges: For any two transactions x, y that only access data from completely separate sets of nodes (e.g., local transactions from different nodes), start-dependency edges have to be added to the SSG if $c_x < b_y$. This is because the transitive closure of the global order adds relations between transactions x, y that did not exist in the local schedules.

Global SSG is correct according to Snapshot isolation. Second, we show that none of the relevant anomalies can occur in the global SSG. As stated in [4] and quoted in Subsection 5.3.4, (local) snapshot isolation holds if G1 and G-SI cannot occur. We discuss the phenomenas relevant for snapshot isolation in turn:

G1a: Aborted Reads. The local nodes provide proper snapshot isolation. Thus, any read of any transaction x that appears in the global schedule is mapped to a write

of a transaction y that committed before transaction x started. The order of these operations is the same in the global schedule as in the local schedule where the read originated from. Furthermore, distributed transactions either commit on all nodes or abort on all nodes. Thus, all reads in the global schedule return only values written by committed transactions.

G1b: Intermediate Reads. Using a similar argument as above: the local nodes provide proper snapshot isolation, thus a transaction x always reads the latest version written by the transaction that committed last before transaction x started. The order of operations is not changed, thus, transaction x reads the proper version.

G1c: Circular Information Flow. The local nodes provide proper snapshot isolation. Therefore, all dependency edges (read-/ write-dependency) from transaction x to transaction y in the SSGs corresponding to the local schedules imply that $c_x < b_y$. For read-dependencies this is because only committed data is read, and for write-dependencies this is because two concurrent transactions must not write to the same object. In other words, dependency edges can only go in the direction of commit order. Since the commit order is the same on all local nodes and in the global schedule, there is no cycle of dependency edges possible in the SSG of the global schedule.

G-SIa: Interference. The local nodes provide proper snapshot isolation. Therefore, a read-/ write-dependency edge in an SSG corresponding to a local schedule also implies that there is a corresponding start-dependency edge (i.e., G-SIa holds in the local SSG). Such an start-dependency edge from transaction x to transaction y implies that $c_x < b_y$. In the global schedule, the order of operations is not changed. Thus, for every start-dependency edge in the SSGs corresponding to the local schedules, there is also a start-dependency edge in the global SSG.

G-SIb: Missed Effects. If there is an anti-dependency edge from transaction y to transaction x in a local SSG, this implies that $b_y < c_x$. This is because the local node provides snapshot isolation, i.e., if it were $c_x < b_y$, the local SSG would contain a start-dependency from transaction x to transaction y and therefore contain a cycle (i.e., it would contain G-SIb). If the global schedule would contain a cycle with the anti-dependency edge from transaction y to transaction x and otherwise only read-/ write-/ start-dependencies, the correctness criteria is not met. The proof is by contradiction. Assume there is a path from transaction x to transaction y consisting only of read-/ write-/ start-dependencies. Any read-/ write-/ start-dependency from transaction s to transaction t implies that $c_s < b_t$ (see above). Using the transitional closure of the $c_s < b_t$ constraints and the fact that $b_s < c_s$ for any transaction, we can derive that $c_x < b_y$. This is a contradiction to the anti-dependency edge that requires $b_y < c_x$ and the correctness criteria that requires the same order of all begin and commit operations on all nodes.

This concludes the proof that any schedule that satisfies the correctness criteria is correct according to distributed snapshot isolation. □

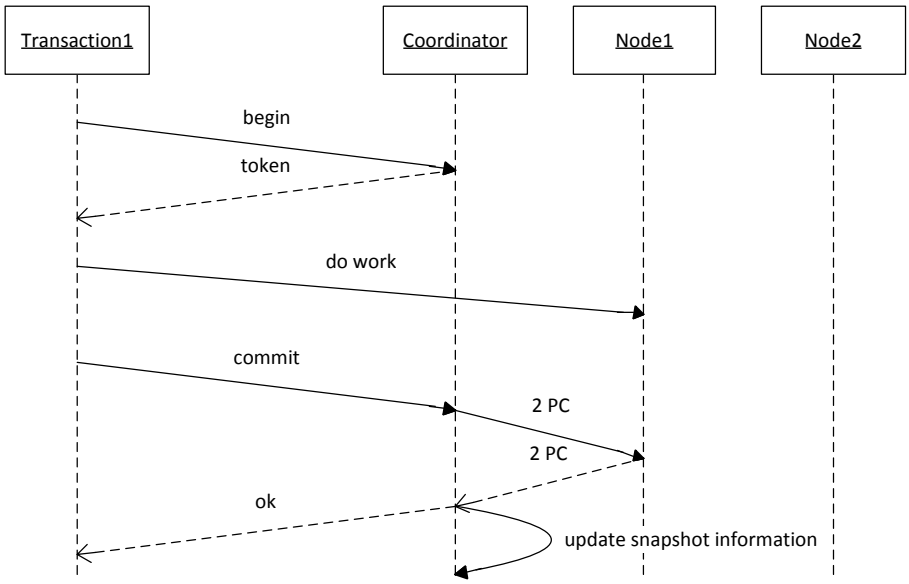


Figure 5.4: Sequence diagram for one transaction in the centralized approach

In the next sections, we present approaches to build systems that adhere to these rules.

5.5 Centralized Coordination

The simplest approach to enforce distributed snapshot isolation is with a centralized coordinator. The idea is that both local and distributed transactions are coordinated by a central instance. That way, the begin and commit operations are atomic as there is only one node that issues the information about snapshots and all nodes directly use that information. The coordinator simply issues a so called *token* that contains all required information in compressed form. This token is then used on every node to present the transaction with the proper view. In order to coordinate the commits, the system employs an atomic commit protocol like two phase commit. Figure 5.4 gives a high-level overview of how one transaction is executed.

This system trivially enforces an order of the distributed begin and commit operations as there is only one order, namely the one coordinated by the central instance. Therefore, the system provides distributed snapshot isolation according to Definition 1.

Obviously, the central coordinator is a bottleneck. All transactions have to access the coordinator to begin, commit and abort. In SAP HANA DB, a number of optimizations have been implemented which involve caching the token at the local node and using the cached token for read-only transactions to improve performance of the begin operation. For write transactions, a new token has to be fetched from the coordinator. This is not

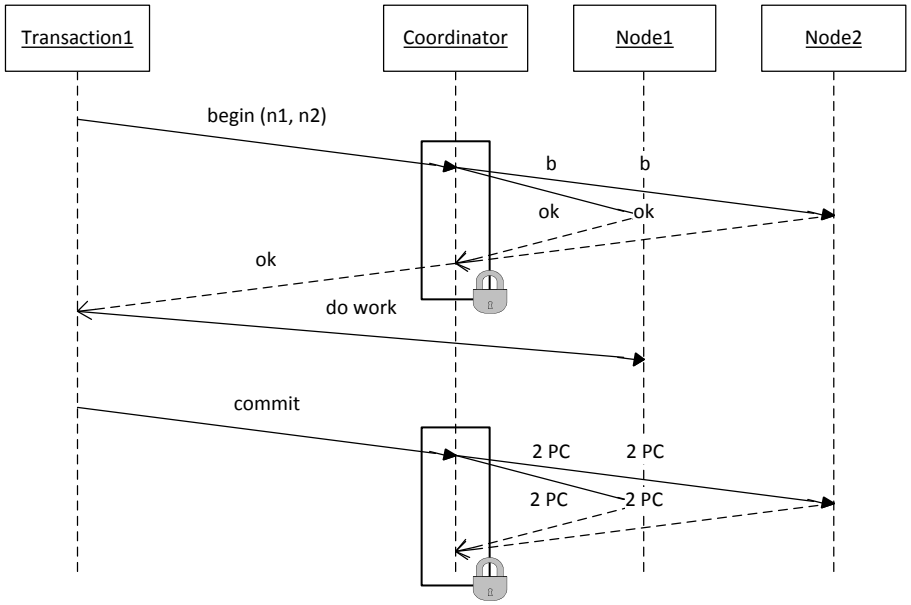


Figure 5.5: Sequence diagram for a distributed transaction in the pessimistic approach

sufficient to make the system scale, therefore in the next sections, we present approaches that separate the local transactions from the distributed transactions and reduce the load of the central coordinator.

5.6 Pessimistic Coordination

In the *pessimistic* approach, the local transactions access only their local node and are not coordinated by the central coordinator. The distributed transactions tell the system in advance that they are distributed and which nodes they will access. Based on this information, the central coordinator prepares the transaction context (i.e., executes a begin operation) on every involved node when a new distributed transaction enters the system. This step happens in a synchronized way, i.e., while the coordinator prepares the context for one distributed transaction, no other distributed transaction can begin or commit. For distributed commits, the system uses an atomic commit protocol like in the other approaches. Figure 5.5 gives a high-level overview of how one transaction is executed.

This approach is inspired by the pessimistic approach for snapshot isolation on federated databases in [69] that synchronizes distributed commits. But as opposed to the proposition in [69] where distributed begins may happen simultaneously, the pessimistic approach for distributed snapshot isolation also synchronizes distributed begins.

The pessimistic approach enforces an order of the distributed begin and commit oper-

ations by explicitly ordering them on the central coordinator. Since no other distributed transaction can begin or commit while the transaction context is prepared for a distributed transaction x or during the commit phase of a distributed transaction, this ensures that the operations are in the same order on every node.

The advantage of this approach is that it separates the local and distributed transactions and allows the local transactions to run without overhead on the local nodes. The disadvantage is that the system requires more knowledge about the transactions in advance. Even if this knowledge is available, which in many scenarios is not the case, the protocol (and therefore the application code) needs to be changed to have the application communicate this knowledge to the database. Such a change is difficult to introduce in existing systems.

5.7 Optimistic Coordination

In the *optimistic* approach, local transactions can run independently from distributed transactions as with the pessimistic approach. Distributed transactions tell the system in advance, that they are distributed, but no exact knowledge about the accessed nodes is required (as opposed to the pessimistic approach).

This approach is inspired by the optimistic approach for snapshot isolation on federated databases in [69], too. As in the federated scenario of [69], the central coordinator keeps a matrix of all currently active distributed transactions that indicates for each pair of distributed transactions whether they are serial or concurrent or still in an undefined relation to each other. To ensure correctness in the distributed scenario, we add a second data structure that stores for each node which distributed transaction started last on that node. If a distributed transaction accesses a new node i, the system performs two checks: First, it uses the list of last begins to ensure that the order of begins is the same on every node. Second, it uses the matrix to check that the serial-parallel anomaly cannot occur (by ensuring that if two transactions are already serial or concurrent on another node, the same relationship is valid for the new node i). If the checks are successful, the data structures are updated. Listing 5.1 shows the steps in more precise notation. For commit, the optimistic approach uses an atomic commit protocol. Figure 5.6 gives a high-level overview of how a distributed transaction is executed.

The optimistic approach in [69] uses the same matrix but does not need to keep track of the begins to enforce the order of distributed begins. On the other hand, the original algorithm adds pseudo-operations on every node that was not accessed by a transaction on commit. This is not necessary in the algorithm presented in this thesis because the issues solved with these pseudo-operations are already tackled with the more restrictive set of rules for ordered begins of transactions in Listing 5.1.

We now show that the algorithm enforces an order among all distributed operations: as stated above, in the optimistic approach, a distributed transaction has to tell the system, that it is distributed beforehand. Thus, if a distributed transaction arrives at the system, a unique timestamp for its begin can be assigned (line 8 in Listing 5.1). Using this timestamp, the system enforces that the same order for begins is valid on all

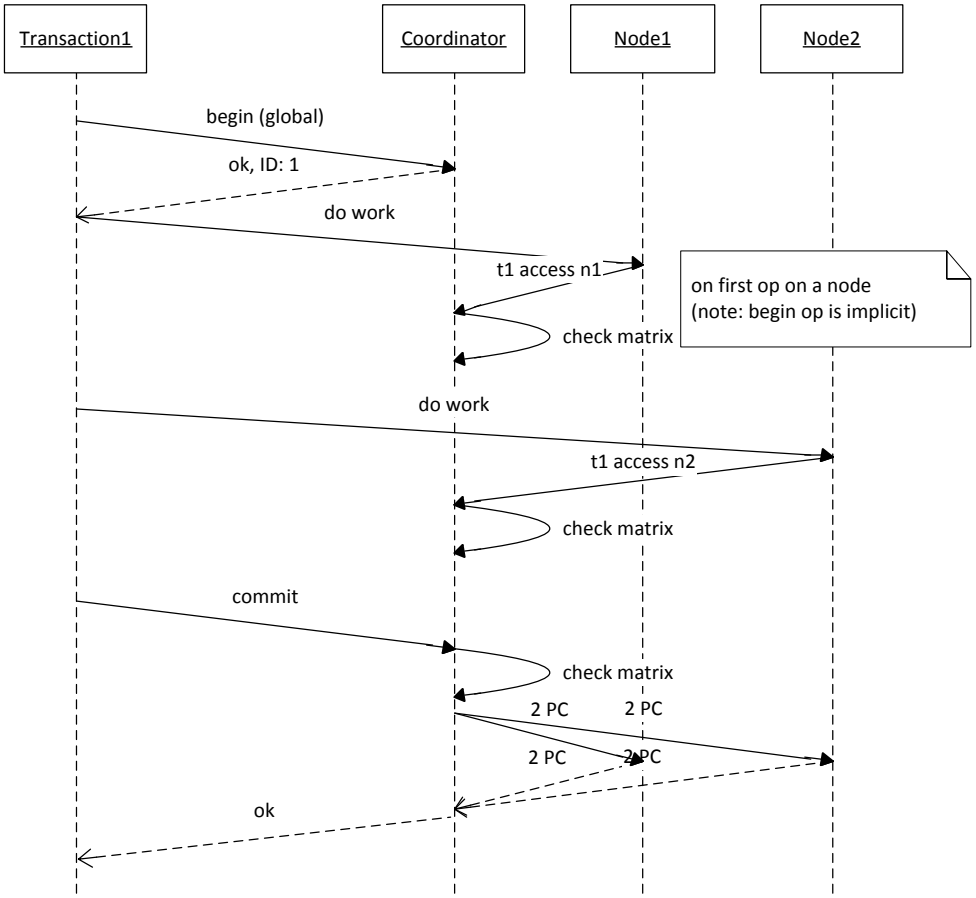


Figure 5.6: Sequence diagram for a distributed transaction in the optimistic approach

Listing 5.1: Optimistic approach: begin of transaction

```
// data
timestamp = 0; // synchronized access
rel[][]; // two dimensional matrix
    // that contains relations between all TAs
last_begin[];
commit_ts[];

bool begin(transaction x, node i) {
  b = timestamp++; // synchronized!
  commit_ts[x] = MAX_INT;
  access(x, i); // store that x accessed i
  // make sure begins are in the proper order
  if (last_begin[i] > b) {
    abort(x);
    return false;
  } else {
    last_begin[i] = b;
  }

  // check matrix
  for ( /* all other active TAs y */ ) {
    if (accessed(y, i)) {
      if (commit_ts[y] < b) {
        // x is serial to y
        if (rel[x,y] == concurrent) {
          abort(x);
          return false;
        } else {
          rel[x,y] = rel[y,x] = serial;
        }
      } else {
        // y did not commit yet
        // therefore x is concurrent to y on i
        rel[x,y] = rel[y,x] = concurrent;
        // note that it is not possible that x
        // and y are serial on any node since
        // neither of the two transactions
        // committed yet
      }
    } // end if accessed(y,i)
  } // end for all other active TAs y
  return true;
} // end function begin
```

nodes by rejecting begins of older transactions once a more recent transaction accessed a node (lines 11-17). Furthermore the atomic commit protocol enforces order among all commit operations. Finally, the matrix check enforces the order between begin and commit operations as follows: If a begin of a transaction x is before the commit of another transaction y, the system marks the two transactions as concurrent (line 33). Based on this information, the system denies transaction x access to any new node once transaction y committed (lines 22-27). It follows that once an order between a begin and a commit is established, it is obeyed on all accessed nodes. It follows that the system enforces an order among all distributed operations.

The advantage of the optimistic approach is that the transactions can try to access additional nodes without having to tell the system in advance. Furthermore, local transactions can run independently from distributed transactions. The disadvantage is that the expansion to more nodes for a distributed transaction does not work in all cases, which leads to more aborts.

5.8 Incremental Snapshots

The *incremental* approach is a new technique to enforce distributed snapshot isolation in distributed databases. Local transactions are completely independent from distributed transactions. The main idea is that if a local transaction wants to become distributed, it detects the restrictions imposed on its snapshot locally. That information is then used on other nodes to assign the transaction a snapshot that adheres to these restrictions. No knowledge about the intentions of a transaction are required beforehand. As in the other approaches, an atomic commit protocol (i.e., two phase commit) is used.

The approach works as follows: Assume transaction x started locally on node i. When transaction x decides to access another node j, the system collects the required information about the state on node i. This information consists of an interval $[c_y, c_z)$ of commit IDs. The transaction requires a snapshot within that interval on the other nodes. Figure 5.7 gives a high-level overview of how a distributed transaction is executed.

The interval of commit IDs is based on local information only. The collection works as follows: the lower bound of the interval is quite easy. Based on the local transaction token (see Section 2.4), the local node knows which distributed transaction committed last before the begin of transaction x (because it is the most recent distributed transaction that transaction x reads from). For the upper bound, two cases are possible: if there is another distributed transaction that committed on node i since transaction x started, the upper bound is the commit ID of the oldest of these transactions. If no distributed transaction committed since (which is the more common case), a bit more care is required to select the upper bound. The idea is to ask the coordinator what the *next* global commit ID will be that it is going to assign. This commit ID is then the upper bound if no other distributed transaction committed on node i during the communication with the coordinator. This communication is asynchronous, i.e., other transactions continue working which may include a commit on node i. Therefore it is necessary to check again once the reply from the coordinator is received and use the commit ID of the

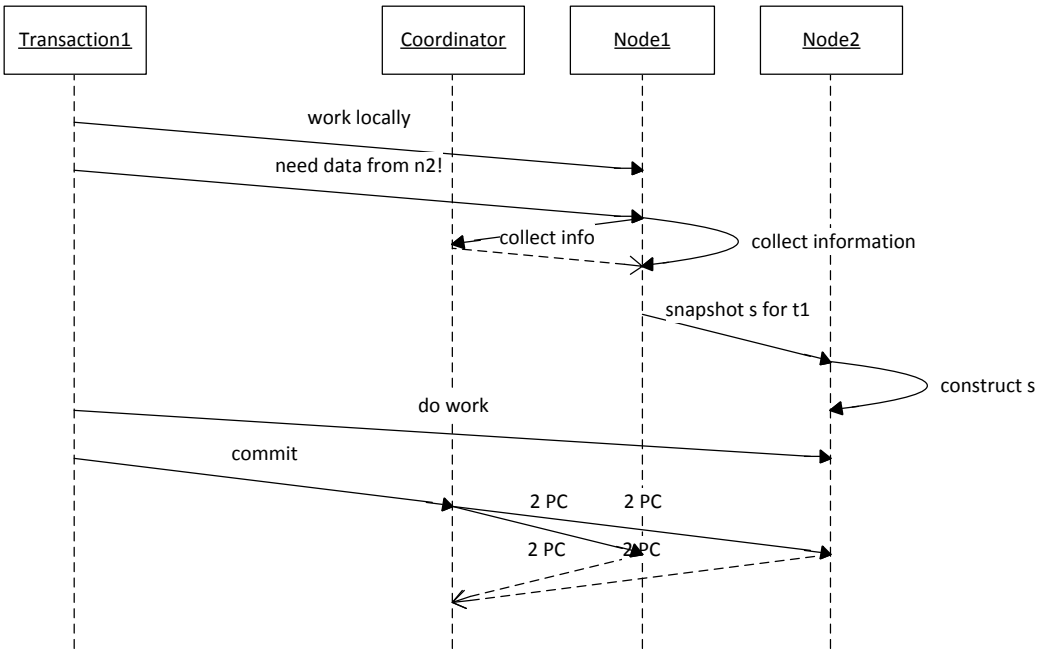


Figure 5.7: Sequence diagram for a distributed transaction in the incremental approach

distributed transaction that committed on that node in the meantime as upper bound if such a transaction exists. If a distributed transaction that did not access the local node commits, this does not need to be considered because the upper bound is still valid.

Furthermore, the system needs to ensure that all begins are properly ordered. The incremental approach does this in a simple but restrictive way: within one interval (or more precisely, for each distinctive *end* of the interval) only one snapshot per node is used. For each node, the first distributed transaction with a given end of the interval selects the snapshot to be used and all other distributed transactions with the same end of the interval have to use the same snapshot on that node. This has two consequences: first, no information about the begins within the interval needs to be collected. It is sufficient to know the oldest transaction from which a transaction does not read (i.e., the end of the interval) to assign the proper snapshot on another node. Second, it may happen that a local transaction z cannot become distributed because it happens to be in an interval for which the snapshot is already chosen (by another transaction) and the local transaction z does not use that exact snapshot. Therefore, the local transaction z is not allowed to become distributed but it may continue locally, if it wants to.

This restriction is necessary to avoid the cross anomaly. Consider the following example: if transaction x from node i that has a different snapshot than the designated one is allowed to access data from another node, its begin of transaction is different than all begins of other distributed transactions on node i (for the sake of presentation, assume it is earlier) but equal on all other nodes. If now another transaction y is allowed to do the same on a different node j and later transaction x accesses node j (i.e., $b_y^j < b_x^j$)

and transaction y accesses node i (i.e., $b_x^i < b_y^i$), the begins are not ordered the same. Of course this is very restrictive but monitoring which node a transaction may access or not is expensive. If a transaction informs the system on its begin that it wants to access data from another node, the system can ensure that this is possible by assigning the designated snapshot to the transaction, i.e., the transaction does not start as a local transaction but is a distributed transaction from the beginning.[11]

Finally, to assign an "older" snapshot to a transaction, the transaction token can be tweaked. This cannot be explained in detail as it involves specific information of the token implementation in SAP HANA DB but it basically involves reconstructing the older token step by step from the current token. For the prototype used to evaluate the approaches a simplified approach was used: Every node *caches* a token representing a snapshot right *before* every commit of a distributed transaction. To assign a snapshot that does not read from transaction y to a transaction, the system can use the token cached when transaction y committed. This is similar to the idea of opening dummy transactions at specific points and use them later to get an old snapshot used in [70].

Once the information is collected, the local node translates the information into the global namespace. To achieve this, every node keeps a mapping table to map between local and global commit IDs as follows: for every commit of a distributed transaction the coordinator issues a global commit ID. The nodes involved in that transaction store that global commit ID along with the local commit ID they assigned.

In summary, the complete steps executed if a local transaction x from node i wants to access data from another node j are as follows:

(1) Transaction x calculates its distributed snapshot by calculating the interval $[y, z)$ as described above and designates its own snapshot as the one to be used by other distributed transactions if they have the same end of the interval.

- if the snapshot before transaction z is already designated, transaction x must not become distributed.

(2) The interval is transformed into an interval in the global CID space by means of the mapping tables on node i.

(3) The global interval is transformed into the local CID space of node j by means of the mapping tables on that node.

(4) With that information the system constructs the snapshot for transaction x on node j as follows:

- if there exists a snapshot that was designated as the local part of the distributed snapshot for transactions with transaction z as their upper end of the interval, transaction x uses that snapshot.
- if there is no such snapshot, the system designates the snapshot produced by the latest local transaction that committed before transaction z as the one to

[11] This optimization is currently not implemented in our prototype.

be used by all distributed transactions with transaction z as the upper end of their interval.

- if there is no commit of transaction z on node j, the system takes the latest local snapshot right before the commit of the next distributed transaction on node j or the most recent snapshot if no other distributed transaction committed on node j since. This is still a snapshot not reading from transaction z since the distributed commits are ordered and it is certain that transaction z did not access node j.

Note that the information about which snapshot to select remains the same for transaction x at all time. If transaction x gets the snapshot not reading from a later transaction t on node j (because transaction z did not commit on node j), transaction x still asks for the snapshot not reading from transaction z on other nodes.

The *incremental* approach enforces an order among all distributed operations for the following reasons: the order among distributed commits is enforced using the atomic commit protocol. The order between distributed begins and commits is enforced because all begins of one transaction are scheduled before the same distributed commit (or if that commit does not exist, before the next commit in the order, which still obeys to the order). The order between begins is not directly enforced but the number of possible begins between two distributed commits is restricted to one and therefore the order among all operations holds (since rule 5.8 explicitly allows simultaneous begins if no other distributed commit is between two distributed begins).

The *incremental* approach solves the issue of the order of the begins in a very restrictive way. This may lead to an increased abort rate because not all local transactions can become distributed. The experiments show that this is not an issue in many scenarios. For scenarios where this increased abort rate is an issue, we briefly sketch a possible extension of the incremental technique. The idea is to enumerate the possible snapshots between two distributed commits on a node in an order-preserving way. To achieve this with only local information, a combined enumeration is used: the commits of the distributed transactions serve as the major clock tick and are available on all nodes. The minor clock tick uses a simple order-preserving enumeration of the local commits. Not all commits are enumerated but only those that are used by distributed transactions. If the enumeration scheme supports updates (e.g., as used in the previous chapters, by leaving gaps), the system can still allow local transactions to become distributed. The combined identifier of global commit ID and local enumeration can then be used on another node. The only requirement is that on all accessed nodes, the order of snapshots is the same. Depending on how well the used enumeration scheme supports updates, almost all local transaction can now become distributed. The integer scheme with gaps described here is of course quite limited but should be sufficient for a number of scenarios. For more demanding scenarios, other more flexible order-preserving enumeration schemes can be used, e.g., Dewey Codes [60]. We do not discuss this extension further in this thesis as the experiments show that the incremental approach with a single snapshot per interval and node performs quite well.

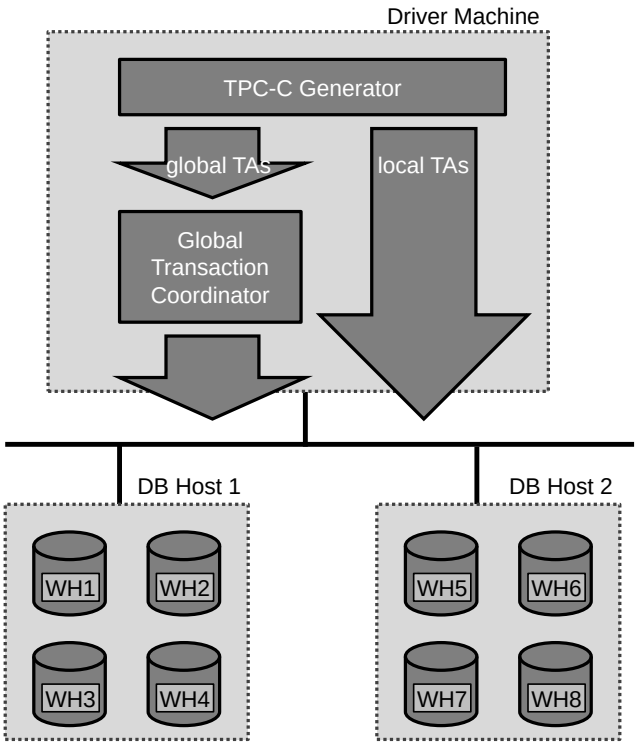


Figure 5.8: Benchmark setup

5.9 Evaluation

This section gives the results of an evaluation of the approaches presented in this chapter using a variant of the TPC-C benchmark [76]. First, the benchmark details are introduced, then evidence is presented that the *incremental* approach a) scales well with the number of instances, b) has comparable performance to the other approaches in many scenarios (i.e., if a-priori knowledge is available), c) behaves reasonable if the distributed transactions get more complex and d) outperforms the other approaches if no a-priori knowledge about the transactions is available.

5.9.1 Benchmark Details

For the performance evaluation, a variant of the TPC-C benchmark [76] is used. The setup is depicted in Figure 5.8. A number of instances of a SAP HANA DB run on servers with 32 cores (4 Intel Xeon X7560 CPUs with hyper-threading) and 256 GB main memory each (e.g., if 8 instances and 2 machines are used, four instances share one machine) are used as distributed database. Each instance runs independently from the others locally enforcing snapshot isolation. Note that there is no communication between the instances, thus the fact that the nodes are only virtually separated does not impact the performance results since the only communication is going on between the coordinator and the individual nodes and that is a real physical link in our setup.

Another server with 8 cores (2 Intel Xeon X5450 CPUs without hyper-threading) and 16 GB of main memory is used as the driver machine: it runs a Java implementation of the TPC-C framework based on prepared statements (i.e., no stored procedures are used). The driver machine also includes the required transaction coordination facilities. The coordination of the transactions is centralized on that machine and implemented using normal OS locks. This setup decreases the penalty for approaches that use a lot of coordination since the clients are usually on many different machines and the coordinator sits on a different machine than the clients.

The database is populated with a number of warehouses (32 in the first experiment, 16 in the third experiment, 8 in the rest of the experiments) and partitioned by warehouse (i.e., if possible one warehouse per instance, otherwise round-robin among the available instances) according to the TPC-C benchmark specification. The number of terminals (i.e., simulated clients) is fixed to 128 in the first experiment and 32 in all other experiments. Every point in the graphs that report throughput is the average number of successful new order transactions per minute (tpmC) based on a 10 minute run of the normal TPC-C benchmark mix including five different types of transactions.

The parameter usually varied in the experiments is the fraction of transactions that are distributed. This is effected by slightly changing the TPC-C benchmark in the following way: according to the specification, 10% of the new orders contain at least one item that is shipped from a remote warehouse. This factor is used to scale the fraction of distributed transactions since every warehouse is on a different instance, an "item from a remote warehouse" implies that the transaction is distributed. A distributed new order transaction involves a random number of nodes: a new order contains 10 items on average (uniformly random between 5 and 15) and for each item the supplying warehouse is selected at random. Therefore most of the distributed new order transactions will span all 8 nodes. The payment transaction can also be distributed, but it is not changed except that in the 0% distributed transactions case, all payment transactions are local. In all other cases, the default setting of the specification is used, i.e., 15% of all payment transactions are distributed. Furthermore a distributed payment transaction only involves two nodes.

To simplify the graph axis and benchmark description, the fraction of new order transactions that are distributed is reported since the new order transaction is what is measured, 45% of all transactions are new order transactions, and the other types of transactions (except the payment transaction) are local only anyway. A similar variant of TPC-C where the fraction of distributed transactions is varied was used in [49].

Note that none of the transactions actually involve distributed query processing like joining data from different instances. As future work, an integration of the ideas presented in this chapter is planned, such that the database can provide full distributed SQL processing capabilities without conceptual changes to the presented approaches.

Table 5.1 contains a summary of the different workloads and machine setups.

Experiment	Warehouses	Instances	Machines	Terminals
Scalability (5.9.2)	32	1-32	1-4	128
Comparison (5.9.3 & 5.9.5)	8	8	2	32
Complexity (5.9.4)	16	16	2	32

Table 5.1: Summary of the workloads

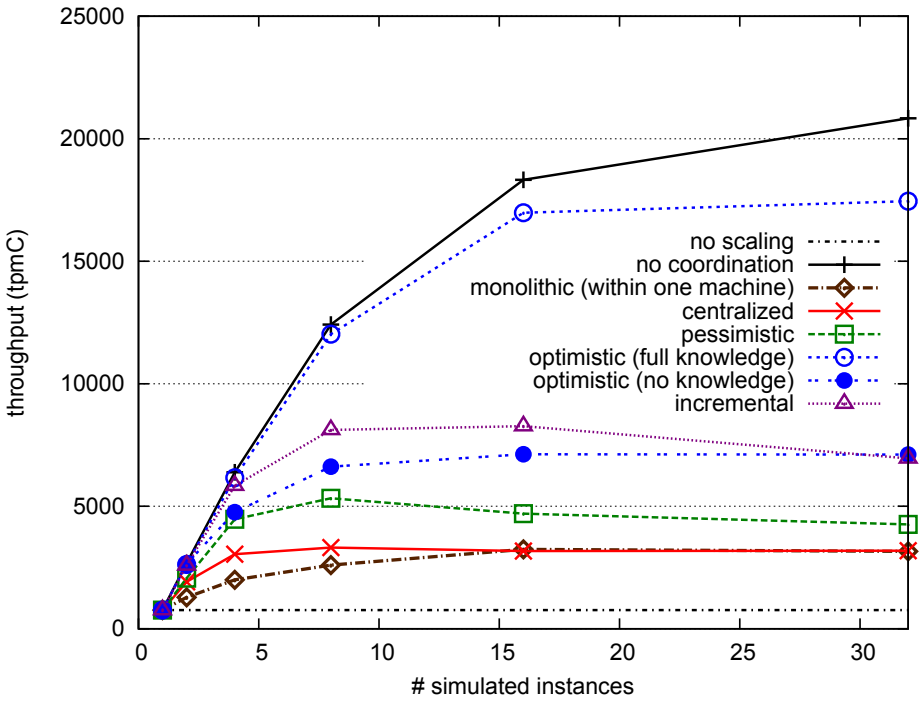


Figure 5.9: throughput (tpmC): vary # instances

5.9.2 Horizontal Scalability

Figure 5.9 shows the results of a scalability experiment with 32 instances running on 4 server machines and storing 32 warehouses. 20% of all new order transactions are distributed. One instance of the database is artificially restricted to use 4 threads for each class of processing tasks, corresponding to the power of a weak commodity machine with 4 cores. This means that 4 threads per instance are working on the main processing during the experiment.[12] Therefore this experiment is not directly comparable to the other experiments in this section, since there the default settings for the thread limits were used.

If there is no way to distribute the database (i.e., no distributed consistency protocol), the system is stuck with one machine. This is what the lower baseline (*no scaling*) shows. The other baseline is *no coordination*, in which case the system does not enforce distributed snapshot isolation. This represents the maximum throughput that the system can achieve and shows the overhead of communicating with multiple nodes. The following lines represent the different approaches, *centralized*, *pessimistic*, *optimistic* (two runs: once with full a-priori knowledge, once without any a-priori knowledge) and *incremental*. These lines show how the system scales if the database is distributed over multiple instances with the same computing power assigned (i.e., the same artificial thread limits per instance). We use the corresponding approach to enforce distributed snapshot isolation. The 32 warehouses are distributed round-robin among the available instances. Figure 5.9 contains an additional line, *monolithic*, that shows how the system would perform if the amount of computing power would be available within one machine, i.e., it represents a single instance with corresponding thread limits (i.e., $4 \times x$ threads) on a powerful machine[13] that does not need a distributed consistency protocol (but enforces local snapshot isolation).

As Figure 5.9 shows, the new approaches enable the system to better utilize the additional computing power as with the centralized or monolithic approaches. This is because the local transactions bypass the central coordinator, which reduces the impact of the bottleneck at the coordinator. In contrast to that, the *centralized* approach scales about the same as *monolithic*, which illustrates the large coordination overhead for this approach due to the fact that both local and distributed transactions involve the coordinator.

The overhead for consistency with the *optimistic* approach is small (i.e., the optimistic approach scales almost as well as *no coordination*) if the client provides a-priori knowledge. As the next experiment shows, this high performance is paid for with a high abort rate. If no a-priori knowledge is available, *optimistic* has a similar performance as *incremental*. The overhead for *incremental* is significant but it scales better than

[12] Unfortunately this does not directly correspond to 4 cores working since other tasks of the database (e.g., index maintenance) use their own threads and expand to more cores. Nevertheless, in the steady state of the experiment, the CPU usage corresponds roughly to 4 cores.

[13] The machine is of the same type as all other machines, i.e., it has 32 cores with hyper-threading. Therefore the machine is not powerful enough to accurately simulate 32 instances ($32 \times 4 = 128$ threads). This is part of the reason why there is no increase in performance between 16 and 32 instances. We show that point for completeness only.

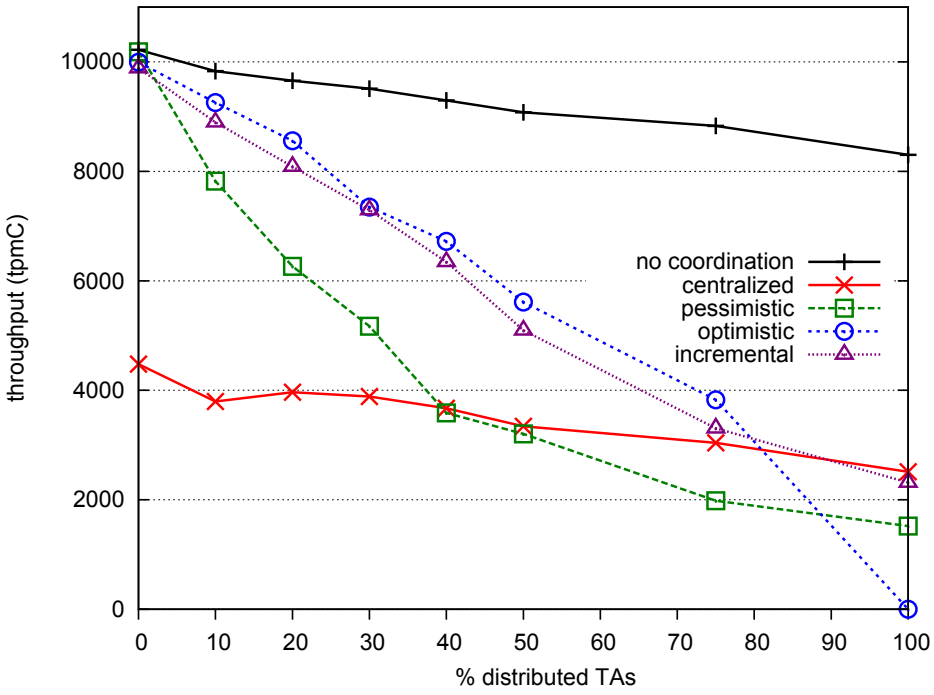


Figure 5.10: throughput (tpmC): vary % distributed TAs

the *pessimistic* approach. The *pessimistic* approach has more overhead if more nodes are available in the system since it has to open transactions on more nodes. Thus, performance degrades quickly for a larger number of simulated instances. To improve readability, the graph only shows the performance for *pessimistic* with full a-priori knowledge. A more detailed analysis of the impact of a-priori knowledge (or the lack thereof) follows in Subsection 5.9.5.

5.9.3 Comparison Between the Approaches

Figure 5.10 shows how the different approaches behave when the ratio of distributed to local transactions changes. This experiment uses 8 warehouses on 8 instances on 2 server machines. All approaches have full a-priori knowledge (i.e., with the begin of a transaction, the set of nodes that will be accessed is provided). All three more sophisticated approaches (*pessimistic*, *optimistic* and *incremental*) significantly outperform the *centralized* approach for up to 50% distributed transactions. For higher fractions of distributed transactions the *centralized* approach works better because the coordination is done once and for all on the coordinator while the other approaches have high overhead if many transactions are distributed. The *optimistic* approach degrades for high fractions of distributed transactions since the consistency check is based on a matrix that contains all active distributed transactions. The size of this matrix grows quadratically with the number of distributed transactions. This has severe impact on performance. The reason for the performance drop to 0 for 100% distributed transaction is the restrictive ordering

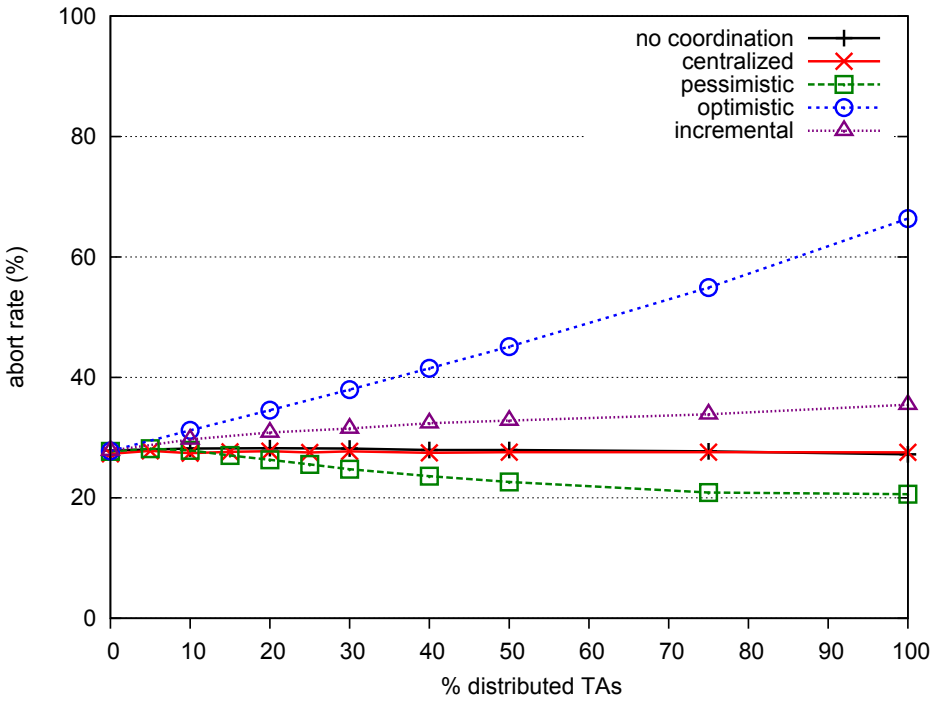


Figure 5.11: abort rate (%): vary % distributed TAs

of the begin operations. Two concurrent transactions that started on different nodes and then access the other node will both be aborted since the system cannot undo the registration of the begin of a transaction. The *pessimistic* approach opens local transactions on every involved node for each transaction in a serialized manner which is much more expensive than talking to one coordinator in a serialized manner.

Comparing the *incremental* approach to the others, Figure 5.10 shows that it always has similar throughput without requiring any a-priori knowledge like the other two approaches.

Furthermore, Figure 5.11 shows an analysis of the abort rates of the different approaches. Since the database of the benchmark also contains an up-to-date field for the revenue and other derived metrics which are updated with almost every transaction, snapshot isolation leads to a high abort rate. Because of these fields a schedule that is correct according to snapshot isolation does not contain any serializability anomalies when running TPC-C [35]. Despite this high abort rate even without distributed coordination, TPC-C is a suitable benchmark for our analysis because without conflicts, any isolation scheme can perform well. Despite the high abort ratem the throughput is still sufficient.

As expected, the *optimistic* approach has the highest abort rate since it allows all transactions to run and checks at the end whether the snapshot was valid.[14] This high

[14]Note that a throughput of 0 at 100% distributed transactions (see Figure 5.10) does not imply that the abort rate has to be 100% as the abort rate is for all transactions while the throughput considers only new order transactions.

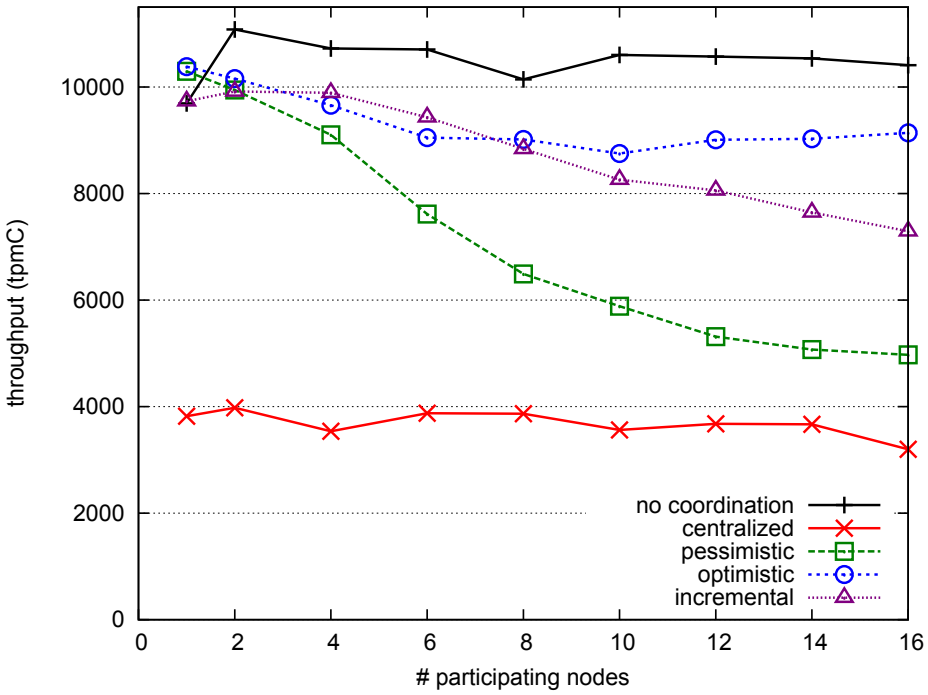


Figure 5.12: throughput (tpmC): vary # participating nodes

abort rate is a liveness issue for distributed transactions: the chance that the same transaction gets aborted over and over and never commits is quite high, especially for long-running distributed transactions that access additional nodes towards the end of the transaction. If the workload contains such transactions, the *optimistic* approach is not a good choice as isolation mechanism. The *incremental* approach also has a slightly increased abort rate compared to the centralized approach. For low fractions of distributed transactions this is due to aborts caused by locally detected issues. For higher fractions, another effect is causing aborts: transactions that started as local cannot access data from another node because another transaction already selected the distributed snapshot for its node. At 100% distributed transactions, about 50% of the aborts are caused by this restriction of the approach. We mentioned a possible extension to alleviate this issue (see Section 5.8), but for our aimed scenario, namely fractions of up to 10% distributed transactions, the current variant of the incremental approach seems sufficient.

5.9.4 Scaling Number of Nodes in a Transaction

Another important factor for the performance of a concurrency protocol is the number of nodes participating in a transaction. Figure 5.12 shows the results of the corresponding experiment with 16 warehouses distributed over 16 instances on 2 server machines. The number of nodes that participate in a new order transaction is scaled by selecting the items from the specified number of warehouses. During the entire experiment, the actual

computing power remains the same (i.e., 16 instances on 2 server machines), just the effort for one transaction changes as more nodes are accessed during one distributed transaction. The fraction of distributed transactions is fixed at 20%. Therefore only the point at 20% distributed transactions from Figure 5.10 and the point at 8 participating nodes in Figure 5.12 are roughly comparable.

As expected, the *centralized* approach is largely unaffected by the number of nodes that participate in a transaction as its coordination overhead is done at a central place for all transactions. The *pessimistic* approach does not scale well with the number of participating nodes as in the beginning of a distributed transaction the overhead increases with every node that has to be contacted in order to set up the transaction context. Since this part has to be serialized, performance suffers. The *incremental* approach performs better than the *pessimistic* approach, but the number of participating nodes is an important factor because the reconstruction of the snapshot is a noticeable overhead per node. Finally, the *optimistic* approach is largely unaffected by the number of participating nodes. This is because the size of the matrix does not depend on the number of participating nodes and the checks on commit are fast even with many participating nodes.

Note that scaling beyond 16 participating nodes is not meaningful with this experiment because every order consists of a random number of items between 5 and 15, thus adding more than 16 participating nodes does not increase the complexity of the transactions. This is also part of the reason why the downward slope of all approaches decreases for more than 12 participating nodes (since the average order consists of 10 items).

5.9.5 Impact of Insufficient Knowledge

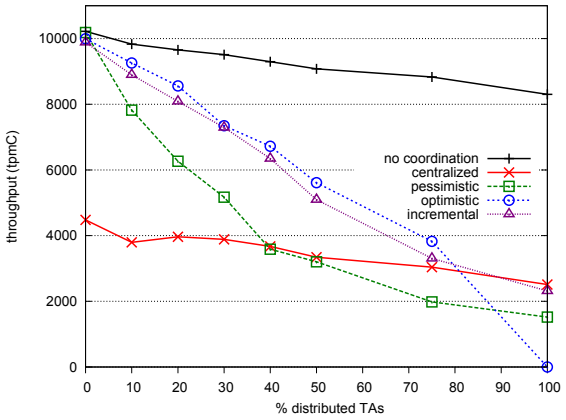
Figure 5.13 shows the same experiment as Figure 5.10 (i.e., 8 warehouses, 8 instances, 2 machines) but for different levels of a-priori knowledge. If less a-priori knowledge is available, the pessimistic and optimistic approaches have to act conservative.[15]

The *pessimistic* approach opens transactions on all nodes if the exact set of nodes that a transaction is going to access is not known. Figure (b) shows that this significantly impacts the performance. Furthermore if it is not known whether a transaction is going to be distributed or not, the *pessimistic* approach opens transactions on all nodes for all transactions. This leads to the very bad performance in Figure (c).

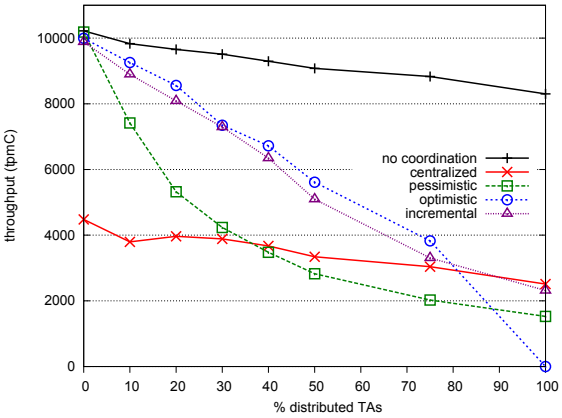
The *optimistic* approach extends the bookkeeping to all transactions if it is not known whether or not a transaction will become distributed. While this has impact on the performance (as shown in Figure (c)), the impact is not as severe as expected. This is most likely because the optimistic approach of this thesis enforces the same order of begins on all nodes. A problem with this order can be detected early and the transaction is immediately aborted, which alleviates the impact on throughput but leads to a high abort rate (see Figure 5.11).

The *incremental* approach does not require (or use, since potential optimizations are

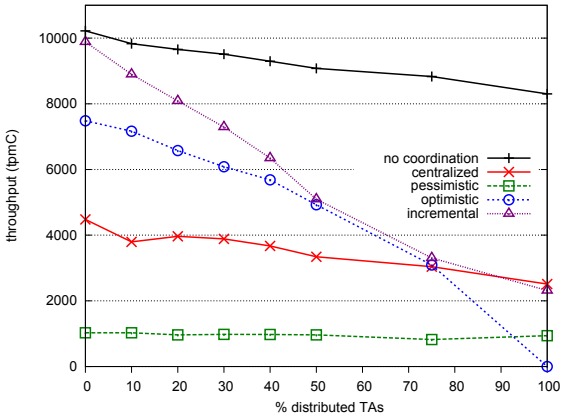
[15]Future work is to evaluate other strategies, e.g., starting all transactions as local only, aborting and restarting as distributed if required.



(a) full a-priori knowledge (same as Fig 5.10)



(b) only local/distributed distinction



(c) no a-priori knowledge

Figure 5.13: throughput (tpmC): vary % distributed transactions for different levels of a-priori knowledge

not implemented) any a-priori knowledge and performs the same in all three scenarios.

5.10 Conclusion

This chapter introduced three approaches to improve distributed snapshot isolation in partitioned databases: *pessimistic* and *optimistic* are extended from ideas in federated databases; *incremental* is a new approach. The main difference between the approaches is the amount of a-priori knowledge about transactions that they need. Pessimistic only works well if the client provides the list of accessed nodes at the begin of a transaction. Optimistic works well if the client tells the system whether a transaction is distributed or local. Incremental finally does not need any a-priori knowledge. Incremental is as efficient as the other approaches in scenarios where a lot of information about the intentions of a transaction are available at the beginning of the transaction. In the general scenario, i.e., if no information is available beforehand, the incremental approach outperforms the other approaches. Furthermore, the incremental approach is completely transparent to the application. All approaches provide clear consistency guarantees, i.e., the system behaves as if it were on a single node.

The new techniques can be used to scale-out column stores if transactional guarantees offered by snapshot isolation are desirable. Especially the fact that the *incremental* approach can be implemented transparently allows adoption in current databases. It does not change any of the properties provided by a normal database while improving performance in certain scenarios, e.g., when scaling the database to many nodes.

6 Conclusions

6.1 Summary

This thesis presented solutions to three issues that arise in distributed in-memory column stores: 1) efficient data structures for a global order-preserving dictionary to improve query performance, 2) an order-preserving encoding scheme that supports updates, and 3) a new technique to improve distributed snapshot isolation.

First, we introduced the idea of a global order-preserving dictionary to leverage the advantages of dictionary compression on a distributed scale. To that end, we introduced the *shared-leaves* approach that allows to efficiently store and access the data of such an order-preserving dictionary while still sustaining the increased load of a global dictionary. The main idea is to avoid duplicating the data as with traditional direct indexes while still allowing fast access as opposed to the indirect indexes that use references to access the data. This is possible because the data has the same order in both indexes since an order-preserving encoding is used. In addition to the leaf structure, cache-conscious indexes were presented that can handle string data.

Second, we introduced *multi-version encoding*, an order-preserving encoding scheme that supports lazy re-encoding. The main idea is to "outsource" the variable part of the codes into so called mapping tables. These mapping tables can then be used to construct the order of the codes if necessary, i.e., in a lazy fashion. With this idea, we can produce fixed-length codes that are fast to process while still supporting arbitrary update workloads. We also showed how multi-version codes can be used with the shared-leaves approach to implement a scalable global dictionary. An order-preserving dictionary allows the system to execute certain query operations directly on the compressed data that would require decompression of the data if the dictionary is not order-preserving (e.g., sorting). This is especially attractive in a distributed scenario to reduce the amount

of data shipped between nodes. Distribution is a key ingredient to enable scalability of an in-memory column store.

Third, we introduced three approaches to improve the performance of snapshot isolation in a distributed database. State of the art is to coordinate all transactions on a central coordinator. This is an inherent bottleneck. The proposed solutions separate local and distributed transactions from each other while still guaranteeing consistency. Two approaches are extended approaches from snapshot isolation in federated databases. The basic idea of the new approach, *incremental*, is to "reconstruct" the right snapshot for a distributed transaction once it accesses data from other nodes. This is in contrast to the pessimistic approach that prepares everything in advance and the optimistic approach that aborts if something conflicts with snapshot isolation.

The global order-preserving dictionary aims to improve performance of column stores in their traditional area of success, namely OLAP-style workloads. This is an area where the optimizations are quite advanced. Consequently, the improvements of using a global order-preserving dictionary were minor.

On the other hand, distributed snapshot isolation is required in new application areas for column stores, for example operational business intelligence [26]. The general industry trend for column stores is towards a mix of OLAP and OLTP workloads. Thus, transaction handling becomes more important in column stores. Our new technique for distributed snapshot isolation, *incremental*, is one building block for that scenario. But the ideas behind incremental are not restricted to column stores. Thus, incremental can be applied in any data store that provides fast access to older versions of the data (e.g., by using multi-version concurrency control (MVCC)). In any case, we strongly believe that the idea of separating local and distributed transactions while still providing strong overall consistency should be adopted by any distributed data store.

6.2 Future Work

Once the global dictionary is adopted in a column store, it is likely to become the next bottleneck. Thus, the distribution of the dictionary over different nodes to efficiently support the scale-up of the column store is part of future work.

Furthermore, we believe that the basic idea of multi-version encoding can be applied to other use cases, e.g., XML-labeling or a general compression service for analytical databases that provide higher level functionalities such as [29] or map-reduce based analytical platforms such as Hadoop. An end-to-end performance evaluation of a column store implementing the techniques introduced in this thesis would provide further insight.

Finally, the incremental technique was introduced to improve performance of distributed snapshot isolation. A possible avenue of future work is to extend the technique to also support serializable snapshot isolation [24], if that is possible at all.

Acknowledgements

First of all, I am truly grateful to my supervisor, Prof. Donald Kossmann. He shared some of his visions with me and inspired many aspects of this thesis. He constantly challenged me by looking at things from a different angle. His connections made this thesis a fruitful collaboration with an excellent partner from the industry.

I am grateful to Prof. Carsten Binnig, my second supervisor. He was always there for me, starting as a post-doc at ETH, then as a co-worker at SAP and finally as a professor at DHBW Mannheim. The discussions with him helped me to organize my thoughts and large parts of this work would not have been possible without his support.

I am also grateful for the work and availability of the other members of my committee; Prof. Thomas Neumann and Prof. Ken A. Ross. They provided me with interesting comments and questions that helped me to improve my work. I thank Prof. Thomas Gross for chairing my examination committee.

Throughout this thesis it was a great pleasure to collaborate and to be in contact with many colleagues from the Systems Group at ETH. They provided a good working atmosphere and supported me in different ways, be it with code for a benchmark, or feedback on my work; relaxing lunch and coffee breaks, or great ski-trips. A special thanks goes to Andreas Morf, part of this thesis is based on the results of his master's thesis.

The collaboration with an industry partner was challenging but very fruitful. I am very grateful to the people behind SAP HANA (formerly and inofficially known as the TREX team). It was amazing what that team can achieve while still being available for my questions. A special thanks goes to Franz Färber. The discussions with him were an amazing experience: he has a talent to ask the right questions and precisely put the finger on the weaknesses of an approach. I would like to thank Dr. Norman May. He has spent a lot of time to improve my work and directed me to the right people inside SAP.

My special appreciation goes to my parents, family and friends for all the support and love they have given me throughout my life and the believe in me.

Bibliography

[1] Daniel J. Abadi. "Query Execution in Column-Oriented Database Systems". PhD thesis. Massachusetts Institute of Technology MIT, 2008 (cit. on pp. 7, 8).

[2] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. "Column-oriented database systems". In: *Proceedings of the VLDB Endowment* 2 (2 Aug. 2009), pp. 1664–1665. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1687553.1687625 (cit. on pp. 7, 8).

[3] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. "Integrating Compression and Execution in Column-Oriented Database Systems". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. SIGMOD '06. Chicago, IL, USA, 2006, pp. 671–682. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142548 (cit. on pp. 11, 15, 16, 20, 48, 51, 52).

[4] Atul Adya. "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions". PhD thesis. Massachusetts Institute of Technology, 1999. URL: http://www.pmg.lcs.mit.edu/~adya/pubs/phd.pdf (cit. on pp. 88–90, 93, 95, 98).

[5] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, et al. "Weaving Relations for Cache Performance". In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 169–180. ISBN: 1-55860-804-4. URL: http://dl.acm.org/citation.cfm?id=645927.672367 (cit. on p. 8).

[6] Gennady Antoshenkov. "Dictionary-Based Order-Preserving String Compression". In: *The VLDB Journal* 6.1 (Feb. 1997), pp. 26–39. ISSN: 1066-8888. DOI: 10.1007/s007780050031 (cit. on pp. 48, 51).

[7] Gennady Antoshenkov, David B. Lomet, and James Murray. "Order Preserving String Compression". In: *Proceedings of the Twelfth International Conference on Data Engineering*. ICDE '96. 1996, pp. 655–663. DOI: 10.1109/ICDE.1996.492216 (cit. on pp. 16, 20, 48, 51).

[8] J. E. Armendáriz-Iñigo, J. R. Juárez-Rodríguez, J. R. González de Mendívil, et al. "A formal characterization of SI-based ROWA replication protocols". In: *Data and Knowledge Engineering* 70 (1 Jan. 2011), pp. 21–34. ISSN: 0169-023X. DOI: 10.1016/j.datak.2010.07.012 (cit. on pp. 82, 84, 86).

[9] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. González de Mendívil, et al. "SIPRe: a partial database replication protocol with SI replicas". In: *Proceedings of the 2008 ACM symposium on Applied computing*. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 2181–2185. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1364207 (cit. on p. 85).

[10] Nikolas Askitis and Ranjan Sinha. "HAT-Trie: A Cache-Conscious Trie-Based Data Structure For Strings". In: *Proceedings of the thirtieth Australasian conference on Computer science*. ACSC '07. 2007, pp. 97–105. URL: http://dl.acm.org/citation.cfm?id=1273749.1273761 (cit. on p. 20).

[11] Nikolas Askitis and Justin Zobel. "Cache-Conscious Collision Resolution in String Hash Tables". In: *String Processing and Information Retrieval*. 2005, pp. 91–102. DOI: 10.1007/11575832_11 (cit. on p. 41).

[12] D. S. Batory. "On searching transposed files". In: *ACM Transactions on Database Systems* 4 (4 Dec. 1979), pp. 531–544. ISSN: 0362-5915. DOI: 10.1145/320107.320125 (cit. on p. 8).

[13] Rudolf Bayer and Karl Unterauer. "Prefix B-Trees". In: *ACM Transactions on Database Systems* 2.1 (1977), pp. 11–26. DOI: 10.1145/320521.320530 (cit. on pp. 19, 20, 31, 35).

[14] Jon Louis Bentley and Robert Sedgewick. "Fast Algorithms for Sorting and Searching Strings". In: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. SODA '97. 1997, pp. 360–369. URL: http://dl.acm.org/citation.cfm?id=314161.314321 (cit. on pp. 36, 72).

[15] Hal Berenson, Phil Bernstein, Jim Gray, et al. "A critique of ANSI SQL isolation levels". In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. SIGMOD '95. San Jose, California, United States: ACM, 1995, pp. 1–10. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223785 (cit. on pp. 12, 82, 87, 89, 93).

[16] Philip A. Bernstein, I. Cseri, N. Dani, et al. "Adapting microsoft SQL server for cloud computing". In: *IEEE 27th International Conference on Data Engineering*. ICDE '11. Apr. 2011, pp. 1255 –1263. DOI: 10.1109/ICDE.2011.5767935 (cit. on p. 82).

[17] Philip A. Bernstein and Nathan Goodman. "Concurrency Control in Distributed Database Systems". In: *ACM Computing Survey* 13 (2 June 1981), pp. 185–221. ISSN: 0360-0300. DOI: 10.1145/356842.356846 (cit. on p. 12).

[18] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 0-201-10715-5. URL: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx (cit. on p. 91).

[19] Bishwaranjan Bhattacharjee, Lipyeow Lim, Timothy Malkemus, et al. "Efficient index compression in DB2 LUW". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1462–1473. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1687553.1687573 (cit. on p. 20).

[20] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. "Dictionary-based Order-preserving String Compression for Main Memory Column Stores". In: *Proceedings of the 35th SIGMOD international conference on Management of data*. SIGMOD '09. 2009, pp. 283–296. DOI: 10.1145/1559845.1559877 (cit. on p. 15).

[21] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi. "Main-Memory Index Structures with Fixed-Size Partial Keys". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. SIGMOD '01. 2001, pp. 163–174. DOI: 10.1145/375663.375681 (cit. on pp. 20, 22).

[22] P. A. Boncz. "Monet: A Next-Generation Database Kernel For Query-Intensive Applications". PhD thesis. Universiteit van Amsterdam, 2002. URL: http://oai.cwi.nl/oai/asset/14832/14832A.pdf (cit. on p. 7).

[23] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, et al. "One-copy serializability with snapshot isolation under the hood". In: *IEEE 27th International Conference on Data Engineering*. ICDE '11. IEEE Computer Society, 2011, pp. 625–636. DOI: 10.1109/ICDE.2011.5767897 (cit. on p. 86).

[24] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. "Serializable isolation for snapshot databases". In: *ACM Transactions on Database Systems* 34 (4 Dec. 2009), 20:1–20:42. ISSN: 0362-5915. DOI: 10.1145/1620585.1620587 (cit. on p. 120).

[25] Lásaro Camargos, Fernando Pedone, and Marcin Wieloch. "Sprint: a middleware for high-performance transaction processing". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 385–398. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273036 (cit. on p. 86).

[26] Surajit Chaudhuri, Umeshwar Dayal, and Vivek Narasayya. "An overview of business intelligence technology". In: *Communications of the ACM* 54 (8 Aug. 2011), pp. 88–98. ISSN: 0001-0782. DOI: 10.1145/1978542.1978562 (cit. on pp. 2, 81, 120).

[27] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. "Query Optimization In Compressed Database Systems". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. SIGMOD '01. 2001, pp. 271–282. DOI: 10.1145/376284.375692 (cit. on pp. 20, 51).

[28] Edith Cohen, Haim Kaplan, and Tova Milo. "Labeling Dynamic XML Trees". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '02. 2002, pp. 271–281. DOI: 10.1145/543613.543648 (cit. on pp. 48, 51, 53).

[29] Jeffrey Cohen, Brian Dolan, Mark Dunlap, et al. "MAD Skills: New Analysis Practices for Big Data". In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1481–1492. URL: http://dl.acm.org/citation.cfm?id=1687553.1687576 (cit. on p. 120).

[30] George P. Copeland and Setrag N. Khoshafian. "A decomposition storage model". In: *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*. SIGMOD '85. Austin, Texas, United States: ACM, 1985, pp. 268–279. ISBN: 0-89791-160-1. DOI: 10.1145/318898.318923 (cit. on p. 8).

[31] Carlo Curino, Evan Jones, Yang Zhang, et al. "Schism: a workload-driven approach to database replication and partitioning". In: *Proceedings of the VLDB Endowment* 3 (1-2 Sept. 2010), pp. 48–57. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1920841.1920853 (cit. on p. 85).

[32] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. "ElasTraS: an elastic transactional data store in the cloud". In: *HotCloud*. San Diego, California: USENIX Association, 2009. URL: http://dl.acm.org/citation.cfm?id=1855533.1855540 (cit. on pp. 82, 84, 85).

[33] Khuzaima Daudjee and Kenneth Salem. "Lazy database replication with snapshot isolation". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 715–726. URL: http://portal.acm.org/citation.cfm?id=1182635.1164189 (cit. on pp. 84, 86, 91).

[34] Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. "Database Replication Using Generalized Snapshot Isolation". In: *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, Oct. 2005, pp. 73–84. ISBN: 0-7695-2463-X. DOI: 10.1109/RELDIS.2005.14 (cit. on pp. 84, 86, 91).

[35] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, et al. "Making snapshot isolation serializable". In: *ACM Transactions Database Systems* 30 (2 June 2005), pp. 492–528. ISSN: 0362-5915. DOI: 10.1145/1071610.1071615 (cit. on p. 114).

[36] Udo Jr. Fritzke and Philippe Ingels. "Transactions on Partially Replicated Data based on Reliable and Atomic Multicasts". In: *International Conference on Distributed Computing Systems* (2001), p. 0284. DOI: 10.1109/ICDSC.2001.918958 (cit. on p. 86).

[37] John Gantz and David Reinsel. *The 2011 Digital Universe Study: Extracting Value from Chaos*. 2011. URL: http://idcdocserv.com/1142 (cit. on p. 2).

[38] Hector Garcia-Molina and Kenneth Salem. "Main Memory Database Systems: An Overview". In: *IEEE Transactions on Knowledge and Data Engineering* 4.6 (1992), pp. 509–516. DOI: 10.1109/69.180602 (cit. on pp. 20, 22).

[39] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. "Compressing Relations and Indexes". In: *Proceedings of the 14th International Conference on Data Engineering*. ICDE '98. 1998, pp. 370–379. DOI: 10.1109/ICDE.1998.655800 (cit. on pp. 19, 20, 51).

[40] G. Graefe and L.D. Shapiro. "Data Compression and Database Performance". In: *Proceedings of the 1991 Symposium on Applied Computing*. Apr. 1991, pp. 22 –27. DOI: 10.1109/SOAC.1991.143840 (cit. on p. 20).

[41] Goetz Graefe. "Efficient columnar storage in B-trees". In: *SIGMOD Rec.* 36.1 (Mar. 2007), pp. 3–6. ISSN: 0163-5808. DOI: 10.1145/1276301.1276302 (cit. on p. 20).

[42] Goetz Graefe and Per-Åke Larson. "B-Tree Indexes and CPU Caches". In: *Proceedings of the 17th International Conference on Data Engineering*. ICDE '01. 2001, pp. 349–358. DOI: 10.1109/ICDE.2001.914847 (cit. on p. 20).

[43] Richard A. Hankins and Jignesh M. Patel. "Data morphing: an adaptive, cache-conscious storage technique". In: *Proceedings of the 29th international conference on Very large data bases*. VLDB '2003. Berlin, Germany: VLDB Endowment, 2003, pp. 417–428. ISBN: 0-12-722442-4. URL: http://dl.acm.org/citation.cfm?id=1315451.1315488 (cit. on p. 8).

[44] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, et al. "Performance Tradeoffs in Read-Optimized Databases". In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB '06. 2006, pp. 487–498. URL: http://dl.acm.org/citation.cfm?id=1182635.1164170 (cit. on pp. 8, 16, 18, 20, 51, 81).

[45] Sándor Héman, Marcin Zukowski, Niels J. Nes, et al. "Positional update handling in column stores". In: *Proceedings of the 2010 international conference on Management of data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 543–554. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807227 (cit. on p. 81).

[46] Stefan Hildenbrand, Donald Kossmann, Tahmineh Sanamrad, et al. *Query Processing on Encrypted Data in the Cloud*. Tech. rep. 735. Department of Computer Science, ETH Zurich, Sept. 2011 (cit. on pp. 53, 54, 79).

[47] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. "Partial Database Replication using Epidemic Communication". In: *International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 485–. ISBN: 0-7695-1585-1. URL: http://dl.acm.org/citation.cfm?id=850928.851857 (cit. on p. 86).

[48] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, et al. "Incremental Organization for Data Recording and Warehousing". In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. 1997, pp. 16–25. URL: http://dl.acm.org/citation.cfm?id=645923.671013 (cit. on pp. 49, 81).

[49] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. "Low overhead concurrency control for partitioned main memory databases". In: *Proceedings of the 2010 international conference on Management of data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 603–614. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807233 (cit. on pp. 82, 84, 85, 110).

[50] Hyungsoo Jung, Hyuck Han, Alan Fekete, et al. "Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios". In: *PVLDB* 4.11 (2011), pp. 783–794. URL: http://www.vldb.org/pvldb/vol4/p783-jung.pdf (cit. on p. 86).

[51] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN: 0-201-03803-X (cit. on pp. 32, 72).

[52] Jens Krueger, Martin Grund, Christian Tinnefeld, et al. "Optimizing Write Performance for Read Optimized Databases". In: *Database Systems for Advanced Applications*. Ed. by Hiroyuki Kitagawa, Yoshiharu Ishikawa, Qing Li, et al. Vol. 5982. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 291–305. ISBN: 978-3-642-12097-8. DOI: 10.1007/978-3-642-12098-5_23 (cit. on p. 12).

[53] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, et al. "Speeding Up Queries in Column Stores". In: *Data Warehousing and Knowledge Discovery*. Ed. by Torben Bach Pedersen, Mukesh Mohania, and A Tjoa. Vol. 6263. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 117–129. ISBN: 978-3-642-15104-0. DOI: 10.1007/978-3-642-15105-7_10 (cit. on p. 11).

[54] Yi Lin, Bettina Kemme, Ricardo Jiménez-Peris, et al. "Snapshot isolation and integrity constraints in replicated databases". In: *ACM Transactions Database Systems* 34 (2 July 2009), 11:1–11:49. ISSN: 0362-5915. DOI: 10.1145/1538909.1538913 (cit. on pp. 82, 86).

[55] Yi Lin, Bettina Kemme, Marta Patiño Martínez, et al. "Middleware based data replication providing snapshot isolation". In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 419–430. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066205 (cit. on p. 91).

[56] David B. Lomet. "The Evolution of Effective B-tree: Page Organization and Techniques". In: *SIGMOD Record* 30.3 (2001), pp. 64–69. DOI: 10.1145/603867.603878 (cit. on p. 20).

[57] David B. Lomet, Alan Fekete, Gerhard Weikum, et al. "Unbundling Transaction Services in the Cloud". In: *Fourth Biennial Conference on Innovative Data Systems Research*. CIDR '09. 2009. URL: http://arxiv.org/abs/0909.1768 (cit. on p. 85).

[58] Stefan Manegold, Martin L. Kersten, and Peter Boncz. "Database architecture evolution: mammals flourished long before dinosaurs became extinct". In: *Proceedings of the VLDB Endowment* 2 (2 Aug. 2009), pp. 1648–1653. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1687553.1687618 (cit. on pp. 7, 8).

[59] Donald R. Morrison. "PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric". In: *Journal of the ACM* 15.4 (1968), pp. 514–534. DOI: 10.1145/321479.321481 (cit. on p. 32).

[60] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, et al. "ORDPATHs: insert-friendly XML node labels". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 903–908. ISBN: 1-58113-859-8. DOI: 10.1145/1007568.1007686 (cit. on pp. 51, 53, 108).

[61] Oracle Corporation. *Oracle Database Concepts, 11g Release 2 (11.2)*. 2011. URL: http://docs.oracle.com/cd/E11882_01/server.112/e25789.pdf (cit. on p. 82).

[62] John Ousterhout, Parag Agrawal, David Erickson, et al. "The case for RAMCloud". In: *Commununications of the ACM* 54 (7 July 2011), pp. 121–130. ISSN: 0001-0782. DOI: 10.1145/1965724.1965751 (cit. on p. 82).

[63] Christian Plattner and Gustavo Alonso. "Ganymed: scalable replication for transactional web applications". In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Middleware '04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 155–174. ISBN: 3-540-23428-4. URL: http://dl.acm.org/citation.cfm?id=1045658.1045671 (cit. on p. 86).

[64] Hasso Plattner. "A common database approach for OLTP and OLAP using an in-memory column database". In: *Proceedings of the 35th SIGMOD international conference on Management of data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 1–2. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559846 (cit. on p. 81).

[65] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. "A case for fractured mirrors". In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB '02. Hong Kong, China: VLDB Endowment, 2002, pp. 430–441. URL: http://dl.acm.org/citation.cfm?id=1287369.1287407 (cit. on p. 8).

[66] Jun Rao and Kenneth A. Ross. "Cache Conscious Indexing for Decision-Support in Main Memory". In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. 1999, pp. 78–89. URL: http://dl.acm.org/citation.cfm?id=645925.671362 (cit. on pp. 5, 19, 20, 31, 35, 41, 51, 65, 76).

[67] Jun Rao and Kenneth A. Ross. "Making B^+-Trees Cache Conscious in Main Memory". In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. SIGMOD '00. 2000, pp. 475–486. DOI: 10.1145/342009.335449 (cit. on pp. 19, 20, 38, 51, 65).

[68] D. P. Reed. "Naming and Synchronization in a Decentralized Computer System". PhD thesis. Massachusetts Institute of Technology, 1978. URL: http://publications.csail.mit.edu/lcs/specpub.php?id=773 (cit. on p. 12).

[69] Ralf Schenkel, Gerhard Weikum, Norbert Weißenberg, et al. "Federated Transaction Management with Snapshot Isolation". In: *Transactions and Database Dynamics*. Vol. 1773. Springer Berlin / Heidelberg, 2000, pp. 1–25. ISBN: 978-3-540-67201-2. DOI: 10.1007/3-540-46466-2_1 (cit. on pp. 82–86, 91, 93, 95, 101, 102).

[70] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, et al. "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation". In: *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 290–297. ISBN: 0-7695-3054-0. DOI: 10.1109/PRDC.2007.39 (cit. on pp. 85, 107).

[71] Dennis G. Severance and Guy M. Lohman. "Differential files: their application to the maintenance of large databases". In: *ACM Transactions on Database Systems* 1 (3 Sept. 1976), pp. 256–267. ISSN: 0362-5915. DOI: 10.1145/320473.320484 (cit. on p. 10).

[72] Ranjan Sinha, Justin Zobel, and David Ring. "Cache-Efficient String Sorting Using Copying". In: *ACM Journal of Experimental Algorithms* 11 (2006). DOI: http://doi.acm.org/10.1145/1187436.1187439 (cit. on p. 36).

[73] Yair Sovran, Russell Power, Marcos K. Aguilera, et al. "Transactional storage for geo-replicated systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 385–400. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043592 (cit. on p. 85).

[74] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, et al. "C-Store: A Column-oriented DBMS". In: *Proceedings of the 31st international conference on Very large data bases*. VLDB '05. 2005, pp. 553–564. URL: http://dl.acm.org/citation.cfm?id=1083592.1083658 (cit. on pp. 11, 15, 51).

[75] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, et al. "Storing and querying ordered XML using a relational database system". In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. SIGMOD '02. 2002, pp. 204–215. DOI: 10.1145/564691.564715 (cit. on p. 51).

[76] Transaction Processing Performance Council. *TPC Benchmark C, revision 5.11*. 2010. URL: http://www.tpc.org/tpcc/ (cit. on p. 109).

[77] Steven P. Vanderwiel and David J. Lilja. "Data Prefetch Mechanisms". In: *ACM Computing Surveys* 32.2 (June 2000), pp. 174–199. DOI: 10.1145/358923.358939 (cit. on p. 32).

[78] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2nd ed.): Compressing and Indexing Documents and Images*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1-55860-570-3 (cit. on p. 26).

[79] Liang Xu, Tok Wang Ling, Huayu Wu, et al. "DDE: from dewey to a fully dynamic XML labeling scheme". In: *Proceedings of the 35th SIGMOD international conference on Management of data*. SIGMOD '09. 2009, pp. 719–730. DOI: 10.1145/1559845.1559921 (cit. on pp. 51, 53).

[80] Chen Zhang and H. de Sterck. "Supporting multi-row distributed transactions with global snapshot isolation using bare-bones HBase". In: *11th IEEE/ACM International Conference on Grid Computing*. GRID '10. Oct. 2010, pp. 177 –184. DOI: 10.1109/GRID.2010.5697970 (cit. on p. 84).

[81] Chen Zhang and Hans de Sterck. "HBaseSI: Multi-Row Distributed Transactions with Global String Snapshot Isolation on Clouds". In: *Scalable Computing: Practice and Experience* 12 (2 2011). ISSN: 1895-1767. URL: http://www.scpe.org/index.php/scpe/article/view/715 (cit. on p. 84).

[82] Jingren Zhou and Kenneth A. Ross. "Buffering Accesses to Memory-Resident Index Structures". In: *Proceedings of the 29th international conference on Very large data bases*. VLDB '2003. 2003, pp. 405–416. URL: http://dl.acm.org/citation.cfm?id=1315451.1315487 (cit. on pp. 20, 33).

[83] Marcin Zukowski, Peter A. Boncz, Niels Nes, et al. "MonetDB/X100 - A DBMS In The CPU Cache". In: *IEEE Data Engineering Bulletin* 28.2 (2005), pp. 17–22. URL: ftp://ftp.research.microsoft.com/pub/debull/A05june/mzukowski.ps (cit. on pp. 15, 51).

[84] Marcin Zukowski, Sándor Héman, Niels Nes, et al. "Super-Scalar RAM-CPU Cache Compression". In: *Proceedings of the 22nd International Conference on Data Engineering*. ICDE '06. 2006, p. 59. DOI: 10.1109/ICDE.2006.150 (cit. on pp. 15, 18, 20, 51, 52, 69).