University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

Vrije Universiteit Amsterdam
Faculty of Sciences

Alicja Łuszczak
Student no. 248265(UW), 2128020(VU)

Simple Solutions for Compressed Execution in Vectorized Database System

Master thesis
in COMPUTER SCIENCE

Supervisor:
Marcin Żukowski
VectorWise B.V.

First reader:
Peter Boncz
Vrije Universiteit Amsterdam
Centrum Wiskunde & Informatica

Second reader:
Jacopo Urbani
Vrije Universiteit Amsterdam

July 2011

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law. The thesis has never before been a subject of any procedure of obtaining an academic degree. Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Compressed execution is a method of operating directly on compressed data to improve database management system performance. Unfortunately, previously proposed techniques for compressed execution required major modifications of database engine and were difficult to introduce into a DBMS. In this master thesis we look for solutions that are easier to implement, unintrusive and well suited to the architecture of VectorWise. We introduce an optimization based on RLE-compressed execution that consists of *constant vectors* and *primitive swapping* mechanism. We also investigate possible benefits of operating on dictionary-encoded data and propose *on-the-fly dictionaries* — a solution for providing uniform domain-wide encoding while avoiding concurrency issues. The conducted experiments show a considerable potential of proposed techniques.

Keywords

database management systems, compression, compressed execution, query processing, database performance

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

H. Software
H.2 DATABASE MANAGEMENT
H.2.4 Systems
Query processing

Contents

1. Introduction . . . 3

2. Preliminaries . . . 5
- 2.1. Database introduction . . . 5
  - 2.1.1. Relational model . . . 5
  - 2.1.2. DBMS architecture . . . 6
  - 2.1.3. Operator trees . . . 7
  - 2.1.4. Storage organisation . . . 7
  - 2.1.5. Database workload . . . 8
- 2.2. Compression in databases . . . 9
  - 2.2.1. Compression schemes . . . 9
  - 2.2.2. CPU cost of decompression . . . 11
  - 2.2.3. Compression in column stores . . . 12
- 2.3. VectorWise architecture . . . 12
  - 2.3.1. Execution model . . . 13
  - 2.3.2. Compression in VectorWise . . . 15
- 2.4. Compressed Execution . . . 15
  - 2.4.1. Early research . . . 16
  - 2.4.2. Avoiding decompression . . . 16
  - 2.4.3. Simultaneous predicate evaluation . . . 17
  - 2.4.4. Code explosion problem . . . 18
  - 2.4.5. Compression techniques . . . 18
- 2.5. Compressed execution in context of VectorWise . . . 19
- 2.6. Summary . . . 20

3. Run-Length Encoding . . . 21
- 3.1. Challenges of RLE-compressed execution . . . 21
  - 3.1.1. Achieving compatibility with original code . . . 21
  - 3.1.2. Increase in amount of code . . . 21
  - 3.1.3. Selection vectors . . . 22
  - 3.1.4. Variation in data clustering . . . 22
- 3.2. Proposed solution . . . 22
  - 3.2.1. Detecting constant vectors . . . 23
  - 3.2.2. Changes in expressions . . . 23
  - 3.2.3. Impact on operators . . . 25
- 3.3. Experiments . . . 26
  - 3.3.1. About the benchmark . . . 26
  - 3.3.2. Results . . . 28

3.3.3. Detecting constant string vectors . . . 34
3.3.4. Filling vectors with constant value . . . 36
3.3.5. Conclusions from the experiments . . . 38
3.4. Impact of the vector size . . . 39
3.4.1. Effect on query execution performance . . . 39
3.4.2. Choosing the right vector size . . . 40
3.4.3. Variable vector size . . . 41
3.4.4. Vector borders problem . . . 41
3.4.5. Optimization algorithm . . . 42
3.5. Summary . . . 44

4. Dictionary Encoding . . . 45
4.1. Operations on dictionary-encoded data . . . 45
4.2. Global dictionaries . . . 47
4.3. The potential of dictionary compression . . . 47
4.3.1. On-Time benchmark . . . 48
4.3.2. TPC-H benchmark . . . 53
4.4. On-the-fly dictionaries . . . 57
4.5. Summary . . . 61

5. Conclusions and future work . . . 63

Chapter 1

Introduction

Database management systems provide functionality to safely store and efficiently operate on data to application developers. They are in use since 1960s and over time have become highly advanced and specialized software. Many data models, query languages and architectures were designed and implemented to achieve best possible performance in different workloads.

Compression has been employed in database systems for years, both in order to save storage space, and to improve performance. The typical approach is to decompress all the data before it is passed to the query execution layer. This way, all the knowledge about data compression can be encapsulated within the storage layer making the implementation of the database system easier. This way however any opportunity for compressed execution is wasted.

Although the idea of compressed execution was first introduced in 1991 [GS91] and was shown to create a considerable potential of performance gains, there has been relatively little research in this area. The few proposed solutions required redesign and major adaptation of the database engine [AMF06], which made them poorly suited for integrating in existing database systems. None of them fitted the architecture of VectorWise [Zuk09], a DBMS that we focus on in this project.

We decided to try to answer the following question: *Is it possible to introduce compressed execution in VectorWise, and achieve significant performance improvements, while avoiding any intrusive changes in the database engine?*

Our goal was to propose new solutions, that would avoid or overcome difficulties of previous ones, but not necessarily to implement all of them. We did not intend to make these solutions suitable for generic database systems, but instead we tried to tailor them to match the specific architecture of VectorWise. They are supposed to be non-invasive and straightforward, and we consider simplicity an advantage. We aimed to achieve a notable part of possible performance benefits, but not to exploit every drop of opportunity at the price of introducing significant extra complexity.

The main contributions of this thesis are:

- proposing a simplification of RLE-compressed execution that consist of *constant vectors* and *primitive swapping* mechanism, and evaluating its performance,
- estimating the benefits of using global dictionaries, and introducing the idea of *on-the-fly dictionaries*, that can provide similar gains, but are easier to maintain.

Not every compression technique exposes properties of data that can be used to execute queries more efficiently, but some of them do. Our work proves that these opportunities are well worth chasing. We show that it is often possible to get significant benefits by using

simple optimizations, which require just a fraction of effort that would have to be invested in order to implement full-blown compressed execution.

The reminder of this thesis is organized as follows. Chapter 2 provides basic introduction to database systems and explains how they employ compression to improve performance. It also presents the notion of compressed execution and reviews related work. Next, in Chapter 3 we discuss RLE-compressed execution, show the challenges it poses, and explain how they can be avoided with constant vectors. We provide benchmarking results for this solution. Chapter 4 explores the topic of operating on dictionary-compressed values, evaluates the benefits and discusses the problems of maintaining uniform encoding for entire columns or domains. Then we propose a possible solution of those problems called on-the-fly dictionaries. Finally, we present conclusions of this thesis and discuss future work in Chapter 5.

Chapter 2

Preliminaries

This chapter is intended to familiarize the reader with basic concepts that will be referred throughout this thesis. These include database management systems, compression and compressed execution, as well as the architecture of VectorWise.

First, we provide an overview of the basic aspects of database systems in Section 2.1. Section 2.2 discusses the role of compression in databases and introduces a number of compression schemes. The description of VectorWise database system is provided in Section 2.3. Section 2.4 presents the idea of compressed execution. Finally, in Section 2.5 we examine the usability of different approaches to compressed execution in context of VectorWise.

2.1. Database introduction

A *database management system* (DBMS) is a software package that allows applications to store and access data in a reliable and convenient way. The application developers do not have to implement specialized data manipulation procedures, but instead depend on functionality provided by DBMS. A database system accepts the description of the task expressed in a high-level language, decides how it should be performed and returns the results to the application.

The modern DBMSes are very specialized software and provide a number of advanced features, for example, they analyse many possible ways of completing a given task to choose the most effective one (query optimization), create backup copies, enforce data integrity, allow many users to modify the data in parallel while avoiding concurrency issues, and prevent unauthorized access.

2.1.1. Relational model

The *data model* determines the way that database is organised and operations that can be performed on data. Many different approaches were proposed over the years (e.g. hierarchical, object-oriented, XML, RDF), but the *relational model* remains the most widespread.

In the relational model the data is divided into *relations*. A relation has a number of *attributes*. Each attribute has a corresponding *domain*, i.e. a set of valid values. A relation is an unordered set of *tuples*. A tuple contains exactly one value for each of the attributes of appropriate relation. A relation is traditionally depicted as a table, where columns represent different attributes and rows correspond to tuples (see an example on the left side of Figure 2.3).

The *relational algebra* defines operations on data stored in the relational model. We call

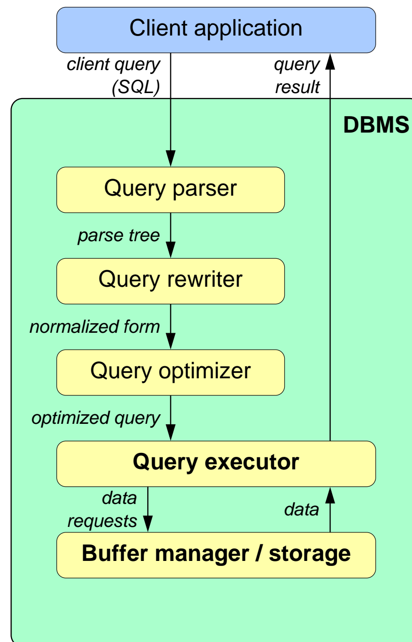


Figure 2.1: Architecture of a database management system (from [Zuk09]).

them *operators*. Some examples are: selection, projection, join, union, Cartesian product. We will not discuss them here. However, we will describe their physical counterparts in VectorWise in Section 2.3.1 and in Section 2.1.3 we will explain how operators interact with each other in the iterator model.

2.1.2. DBMS architecture

Although different database systems have different components, in general they follow a multi-layer architecture presented in Figure 2.1. We will now explain the role of each of these layers.

First, the client application has to describe the task that it wants to be performed. Such a task is called a *query* and it is expressed in a special language, e.g., SQL. The query is then passed to the database system.

The *query parser* analyses the query. If the syntax is correct, the parser will produce an internal representation of the query, called *parse tree*.

Next, the *query rewriter* ensures that the *parse tree* has correct semantics. It checks whenever the accessed attributes actually exist in proper relations, operations are performed on correct data types, and so on. The result is a tree in *normalized form*, specific for a given database system, and usually similar to relational algebra.

Then, the *query optimizer* tries to find the most efficient way of executing the query. It analyses many possibilities and modifies the tree according to its decisions. If an operator has many physical equivalents, the query optimizer has to pick the most suitable implementation.

The *query executor* is the part of database system which performs the actual query processing. It requests the data from the storage layer and operates on it according to the query plan provided by the query optimizer.

Finally, the *buffer manager* or the *storage layer* is responsible for managing in-memory buffers as well as accessing and storing the data on disks or other media.

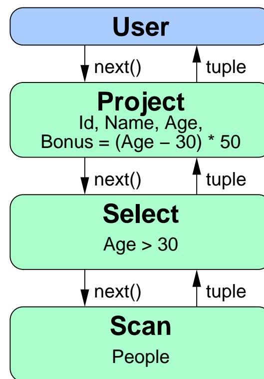


Figure 2.2: Example of an operator tree (from [Zuk09]).

2.1.3. Operator trees

We will now discuss *tuple-at-a-time* iterator model, which is the most popular way of implementing a database engine. The VectorWise engine also evolved from this model.

In the iterator model, the physical plan produced by a query optimizer consists of a number of operators. These operators are objects corresponding to relational algebra operators. Each of these objects implements the methods `open()`, `next()` and `close()`. They are organized in the operator tree, so that the result of the root operator is the result of the entire query.

The `open()` method initializes the operator. When `next()` is called, the operator produces and returns the next tuple. The `close()` method is used when the query execution is terminated to free the resources. All of those methods are called recursively on children operators.

An example operator tree for a query

```
SELECT Id,
       Name,
       Age,
       (Age - 30) * 50 AS Bonus
FROM   People
WHERE  Age > 30
```

is presented in Figure 2.2. We will now describe how this query is executed.

First, the Project operator is asked for the next tuple. Project requests a tuple from the Select operator, and Select operator request a tuple from Scan. The Scan operator reads the next tuple from the table and passes it back to Select. Select uses this tuple to evaluate its condition. If the tuple fails to meet the condition, then Select asks Scan for the next one. Otherwise, the tuple is returned to Project. Project creates a new tuple with copies of Id, Name and Age attribute values, and the computed Bonus. The new tuple is then passed to the user and the process is repeated. When the Scan operator detects that there are no tuples left to read, it returns a special value that marks the end of tuple stream.

2.1.4. Storage organisation

Choosing the database model and the execution model does not determine the organisation of physical database representation, that is, the way the data is stored on persistent media.

Relation

Id	Name	Age
101	Alice	22
102	Ivan	37
104	Peggy	45
105	Victor	25
108	Eve	19
109	Walter	31
112	Trudy	27
113	Bob	29
114	Zoe	42
115	Charlie	35

NSM representation

Page 1

101	Alice	22	102
Ivan	37	104	Peggy
45	105	Victor	
25	108	Eve	19

Page 2

109	Walter	31	112
Trudy	27	113	Bob
29	114	Zoe	
42	115	Charlie	35

DSM representation

Id	Name	Age
101	Alice	22
102	Ivan	37
104	Peggy	45
105	Victor	25
108	Eve	19
109	Walter	31
112	Trudy	27
113	Bob	29
114	Zoe	42
115	Charlie	35

Figure 2.3: Example of a relation and its NSM and DSM representations (from [Zuk09]).

The most popular method is to store the tuples of a given relation one after another in continuous manner. This approach is called *N-ary storage model* (NSM) and database systems employing it are called *row stores*. It is depicted in the middle part of Figure 2.3.

An alternative method is to store values of each attribute separately. It is called *decomposed storage model* (DSM) and DBMSes using it are *column stores*. The same relation stored in DSM is shown in the right part of Figure 2.3.

Both of these solutions have their advantages and shortcomings. Their performance differs greatly depending on the workload and characteristics of the used database engine [ZNB08, AMH08]. There are also mixed approaches that try to get the best of both worlds. For example, PAX [ADHS01] stores values of different attributes separately, but it never splits a single tuple between two or more blocks. VectorWise allows both column-wise and PAX storage.

2.1.5. Database workload

Database systems were initially intended mostly for *on-line transactional processing* (OLTP) applications. They were designed to cope with many concurrent reads and updates, with each query accessing a relatively small number of tuples. The main goal was to minimize the portion of the data that has to be locked in order to modify a tuple [MF04]. Example queries from an OLTP workload include recording a bank transaction or checking price of a product. TPC-C benchmark [TPC10] is a standard tool for comparing performance of different database systems in OLTP applications.

On-line analytical processing (OLAP), *data-mining* and *decision support system* provide an entirely different workload. The queries usually only use a few attributes, but may access a large portion of tuples. The example is computing the fraction of computer games sold to persons that are 50 or older. The most widespread OLAP benchmark is TPC-H [TPC11].

The database systems optimized for different workloads have to be designed differently to achieve best performance. The DBMSes intended for OLTP queries are typically row stores following the tuple-at-a-time iterative model. The ones intended for analytical workloads are more often column stores. They sometimes employ an alternative query execution model, for example, vectorized execution as described in Section 2.3.1.

2.2. Compression in databases

Database systems often store data in compressed form. Compression can be applied to persistent data — that is, to the actual content of a database — as well as to intermediate results of query processing.

It is widely known that compression can improve the performance of a database systems [AMF06, Aba08, GS91, RHS95, Che02]. First, the time required to transfer the data from disk to main memory is shorter because the amount of data that has to be transferred is reduced. The same applies to moving data from main memory to CPU cache (the topic of in-cache decompression was explored in [ZHNB06]). Second, a bigger fraction of database can fit in the buffer pool, so the buffer hit rate is higher. Third, data can be stored closer together, which can be beneficial when using magnetic disks because it helps to reduce seek times.

However, these advantages come at the price of decompression overhead. We will explore this topic in Section 2.2.2.

2.2.1. Compression schemes

Database systems have special requirements for the compression methods rarely met by general purpose compression schemes. For that reason many compression algorithms were designed specially for database use [ZHNB06, GRS98, RHS95, HRSD07, RH93].[1]

One of the requirements is allowing *fine-grained access* to the data, which means that in order to read just few values we don't have to decompress the entire page, but only a small fragment of it. The smaller that fragment is, the better. For example, when using dictionary or frame of reference encoding (described below), we don't have to decompress any excessive data. Contrary, when using an adaptive scheme — one where the encoding is constructed basing on previously compressed or decompressed data, e.g., LZW – the decompression of the entire page is unavoidable. Providing fine-grained access to data is an important property that plays a major role in designing a compression scheme for database use [ZHNB06, GRS98, RHS95].

We will now present short descriptions of a number of compression schemes used in database systems. For more details we refer to [RH93, Aba08].

Fixed-width dictionary encoding In dictionary encoding we map values to fixed-width codes. A data structure for translating codes into values and values into codes is called a *dictionary*. An example of compressing data using dictionary encoding is shown on Figure 2.4.

This compression scheme requires knowing the number of distinct values upfront, before the compression process can begin. In practice, we have to use two-pass method: in the first pass we count distinct values and build a dictionary, and in the second pass we compress the data.

Dictionary encoding achieves a very high decompression speed, but it's compression ratio can be ruined by outliers — values that occur extremely rarely in the input, but still have to included in the dictionary. A solution for this issue can be found for example in [ZHNB06].

[1] Compression algorithms designed for DBMS use — unlike general purpose compression methods — can exploit the fact that data in a database has specific types (e.g. decimals, strings, dates). This allows better compression ratios (defined as the ratio between the compressed data size and the original data size) and lower CPU usage during decompression.

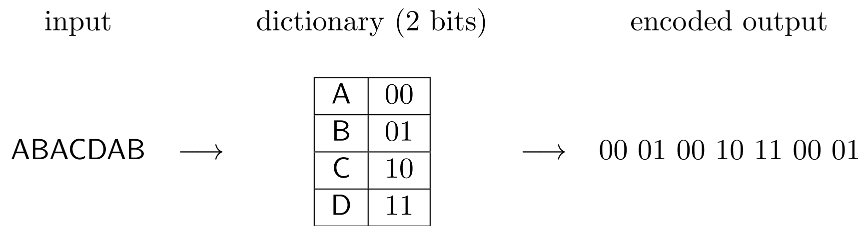


Figure 2.4: Example of dictionary encoding.

Huffman encoding Huffman encoding replaces occurrences of values with codes, just like the dictionary encoding. However, the lengths of codes are variable. The values that occur often get shorter codes, and the values that occur rarely get longer ones. To ensure no ambiguity during decompression, no code is a prefix of another one.

Just like with dictionary encoding, we first have to determine the encoding before we can start compressing data. If statistics are not available upfront, we have to examine the data and determine probabilities of occurrence of different values. The actual algorithm for deriving codes from probabilities can be found in [Huf52]. An example of Huffman-encoded data is presented on Figure 2.5.

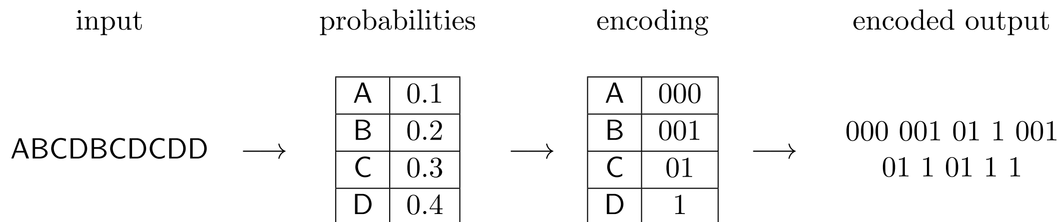


Figure 2.5: Example of Huffman encoding.

Huffman encoding results in better compression ratios than dictionary encoding, but it is also slower to decompress, because we first have to determine the length of the code. This problem was addressed for example in [HRSD07].

Run-length encoding The run-length encoding replaces sequences of identical values by a pair: a value and the number of its repetitions. Such a group of equal consecutive vales is called a *run*. The groups of values that cannot be effectively compressed using RLE are marked with negative run length. An example of RLE compression is shown on Figure 2.6.

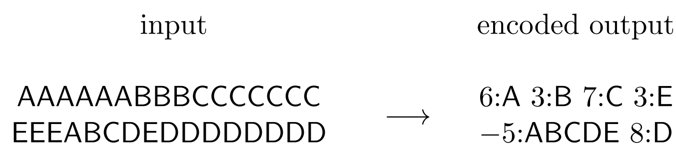


Figure 2.6: Example of run-length encoding.

RLE is useful for columns that contain long runs. This is typical for ones that data is sorted on or ones that contain repeated zeroes or nulls.

Frame of reference (prefix compression) FOR is a method for compressing numeric data. To encode values $v_1, v_2, \ldots, v_n$, it first finds the minimum and maximum value. It records $\min_i(v_i)$ followed by $v_k - \min_i(v_i)$ for $k = 1, 2, \ldots, n$. Each of those differences is encoded using $\lceil \log_2(\max_i(v_i) - \min_i(v_i) + 1) \rceil$ bits [ZHNB06]. An example in presented in Figure 2.7.

input 8-bit integers		determining encoding		computing offsets from base		encoded output	
						01101100	= base
112				4		100	offsets
108		$\min_i(v_i) = 108$		0		000	
113	$\longrightarrow$	$\max_i(v_i) = 113$	$\longrightarrow$	5	$\longrightarrow$	101	
110		$\lceil \log_2(6) \rceil = 3$		2		010	
112				4		100	
109				1		001	

Figure 2.7: Example of frame of reference compression.

2.2.2. CPU cost of decompression

In the recent years we observed a growing disparity between the performance of disks and CPUs [BMK99, Che02, ZHNB06]: the available processing power grows at much faster rate than the disk bandwidth. The advent of multicore processors makes this disproportion even stronger. As a consequence, database systems often have to be deployed on machines with RAID storage constructed of tens or hundreds of disks in order to provide the required I/O bandwidth [ZHNB06].

However, the issue of insufficient I/O bandwidth can be to some extent mitigated by applying proper compression schemes. We will explain this idea using a model proposed in [ZHNB06].

Let B denote I/O bandwidth, Q — query bandwidth (that is, the maximal rate at which data could be processed, assuming that I/O is not a bottleneck), and R — resulting actual bandwidth. We have:

$$R = \min(B, Q)$$

Now, assume that the data is compressed so that the proportion between size of raw data and size of compressed data is $r > 1$, and let the decompression bandwidth be C. The data required for query execution can now be provided faster (the physical bandwidth is obviously unchanged, but the information density is greater):

$$B_c = Br$$

On the other hand, the additional CPU time is used for decompression. One can check that:

$$Q_c = \frac{QC}{Q + C}$$

Finally, we can determine the resulting bandwidth.

$$R_c = \min(B_c, Q_c) = \min\left(Br, \frac{QC}{Q + C}\right)$$

Using these equations, we will analyse the impact of compression on different queries.

I/O bound queries If the query was I/O bound, then we have $R = B < Q$. Now, if the query is still I/O bound after applying compression, then we have

$$R_c = Br = Rr$$

In other words, the performance was improved by a factor of r (usually more than threefold).

On the other hand, if the query is now CPU bound, then we have:

$$R_c = \frac{QC}{Q + C}$$

We can't clearly say if that's better or worse performance than before. If the difference between Q and B was big and a light-weight compression scheme was used (that is, C is significantly larger than B), then the query will be executed faster. However, too expensive decompression will surely ruin the performance.

The bottom line is that if the decompression bandwidth C is much greater than the I/O bandwidth B, then we can expect shorter execution times of I/O bound queries. Researchers tried to exploit this effect in a number of studies, e.g., [HRSD07, GRS98, ZHNB06].

CPU bound queries We have

$$\frac{QC}{Q + C} < Q < B < Br$$

So we can conclude that

$$R_c = \frac{QC}{Q + C} < Q = R$$

Therefore, using compression will hurt the performance of I/O bound queries. However, with light-weight schemes this effect is much less severe that with heavy-weight ones.

2.2.3. Compression in column stores

Column stores are considered to create better opportunities for reaching high compression ratios [Che02], because they keep values of a single attribute close together. Such values are much more likely to expose regularities useful for compression than values of different attributes.

Unlike in row stores, the column stores omit reading from disk attributes that are not needed to execute a given query. This way the I/O traffic is limited to necessary data. Therefore it often happens that either queries are not I/O but CPU bound [MF04] or at least the difference between I/O and query bandwidth is not very high. For that reason, only light-weight decompression methods can be used in a column stores without hindering query execution performance.

2.3. VectorWise architecture

VectorWise [Ing] is a state-of-the-art database system [Zuk09] for analytical applications that employs a wide range of optimizations designed specifically to exploit capabilities of modern hardware. In this section we discuss its architecture, focusing on aspects most relevant for this thesis.

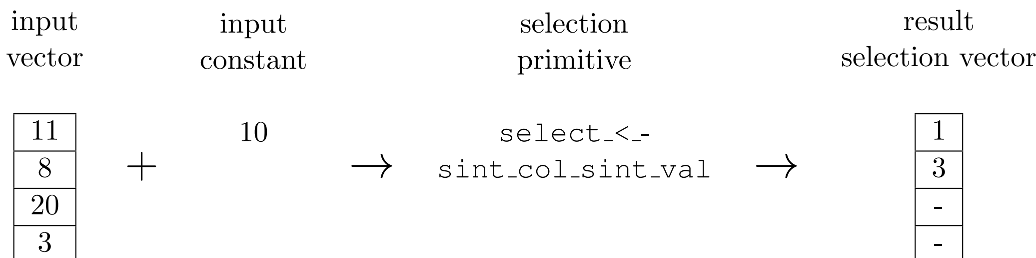


Figure 2.8: Example of selection vector produced by selection primitive. Elements of input vector are compared one by one with input constant. Resulting selection vector contains indices of elements smaller than 10.

2.3.1. Execution model

Vectors

VectorWise employs the *vectorized execution model*. Here, data is not passed in tuples like in query execution engines of most other database systems, but instead it is partitioned vertically into *vectors*. Each vector holds values of a single attribute from many consecutive tuples. The number of elements in each vector is fixed and is usually in range 100 . . . 10000.

In the traditional tuple-at-a-time model, operators have to acquire tuples to process one by one. To get next input tuple, they have to call method `next()` of their children operators. Those function calls cannot be inlined, and may require preforming complex logic. As a result, tuple-at-a-time model suffers from high interpretation overhead, that can easily dominate query execution. In vectorized execution model, operations are always performed on all values in a vector at once. A single call of `next()` method produces not a single tuple, but a collection of vectors representing multiple tuples. Hence, the interpretation overhead is amortized over hundreds or thousands of values, and the queries are executed more efficiently [Zuk09].

Apart from the ordinary vectors, there is a special kind of vectors used to denote that part of tuples must not be used during operator evaluation (e.g. because they were eliminated by an underlying selection). These vectors are called *selection vectors* and hold offsets of elements that should be used. An example of how a selection vector is created is shown in Figure 2.8.

Primitives

The operations working directly on data stored in vectors are usually executed using *primitives*. Primitives are simple functions, that are easy to optimize for a compiler and achieve high performance on modern CPUs. They are designed to allow SIMDization (using special instructions simultaneously performing the same operation on multiple input values) and out-of-order execution (reordering instructions by the CPU to avoid being idle while the data is retrieved), and to efficiently use CPU cache.

Primitives are identified by signatures, which describe the performed operation and types of input and output. For example, the signature `map_add_sint_col_sint_val` represents signed integer (`sint`) addition (`map_add`) of a vector (`col` — column) and a constant (`val` – value). Such a primitive is implemented with code similar to this [Zuk09]:

```
int
map_add_sint_col_sint_val(int n, sint *result, sint *param1, sint *param2)
```

```
{
   for (int i = 0; i < n; i++)
      result[i] = param1[i] + *param2;
   return n;
}
```

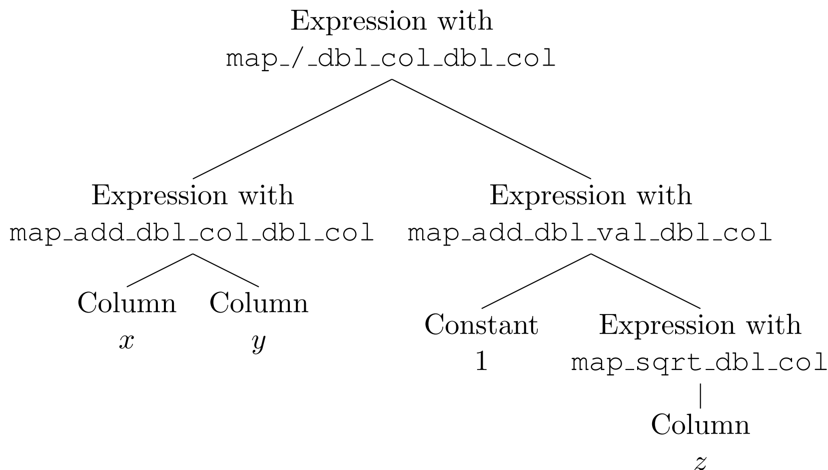
Expressions

Expressions are objects that are responsible for executing primitives. They bind a primitive to its arguments and provide a vector to store the results. Expressions are organized in a tree structure and they are evaluated recursively. Hence, the evaluation of an expression results in evaluation of its children expressions.

For example, in order to compute

$$\frac{x + y}{1 + \sqrt{z}}$$

where x, y and z are attributes of type `dbl`, we would have to construct following tree of expressions:



Operators

As we explained earlier, query plans are composed of *operators* that loosely match the relational algebra operators. They contain generic logic for performing different operations like joins or selections. Operators do not manipulate data directly, but instead use appropriate primitives, often wrapped in expressions. More information about this topic is available in [Zuk09].

The following list contains descriptions of VectorWise operators. It does not feature all of the operators, but only ones that are used later in this thesis.

- `Scan` is responsible for communicating with the storage layer to obtain table data required for query processing.
- `Project` can introduce new columns with values computed from input columns. It can also rename or remove columns.
- `Select` returns the tuples that match a given predicate, and discard the others.

- `Aggr` divides tuples into groups and compute statistical functions (max, min, sum, count) for each of the groups. We call it *aggregation*, the columns used for dividing tuples are *grouping columns* or *group-by columns*, and the results of statistical functions are *aggregates*. `Aggr` can also be used for duplicate elimination.
- `Sort` returns exactly the same tuples as it consumes, but in different ordering, determined by the values in *sort columns*.
- `TopN` also returns sorted tuples, but only first N of them, while the rest is discarded.
- `HashJoin` processes two streams of tuples. It matches and returns pairs of tuples from different streams that have equal values in *key columns*. For that purpose, it uses a hash table.
- `CartProd` also processes two streams of tuples, but returns all the possible combinations of pairs of tuples from different streams.

2.3.2. Compression in VectorWise

VectorWise is intended for analytical workloads, so it uses column-wise storage [Zuk09] (see Section 2.1.4). That is, it partitions data vertically and stores values of each attribute separately. As we explained, this creates much better opportunities for using compression effectively than the row-wise storage, because the data from the same domain and with similar distribution is stored near each other.

In VectorWise, compression is used to obtain better query execution performance, rather than to save storage space, but there is no notion of compressed execution. All the data is decompressed before passing it to the scan operator [ZHNB07].

Different parts of a single column may be compressed using different compression schemes. In VectorWise, the most appropriate scheme is chosen for each part automatically by analysing samples of data, and picking the scheme that offers best compression ratio. It is also possible to assign a fixed compression scheme to a column.

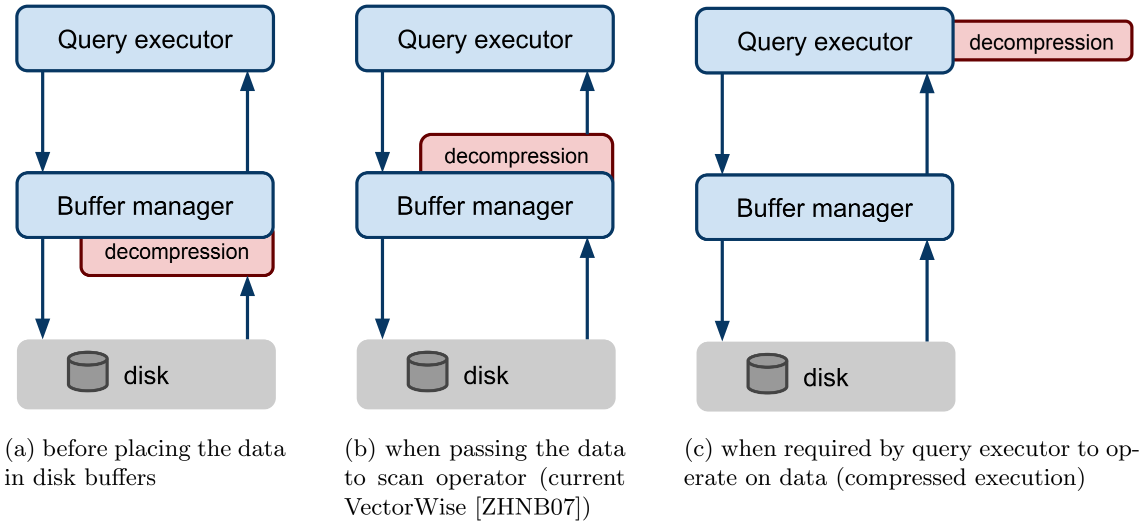
To match the vectorized execution model, the decompression process is performed with a granularity of a vector. Scan does not request the next tuple or value from the storage layer, but instead it asks for an entire vector. This way the decompression can be performed more effectively, because the overheads are amortized across a large number of values.

VectorWise employs a number of light-weight compression methods. Some of them were designed specifically to exploit the capabilities of super-scalar CPUs [ZHNB06].

2.4. Compressed Execution

The traditional approach to compression in database systems is to eagerly decompress all the data. The decompression may take place before the data is placed in I/O buffer (see Figure 2.9a), or before it is passed to processing layer (see Figure 2.9b). This way the execution engine can be kept completely oblivious of data compression, but also many optimization opportunities are wasted.

An alternative approach is to employ *compressed execution*, that is to allow passing the data from storage layer and between operators in compressed form. The decompression is then performed only when it's needed to operate on data (see Figure 2.9c). This way the amount of memory required for query execution can be reduced, unnecessary decompression can be avoided and even some of the operations can be performed directly on compressed data. However, this method also creates its own new challenges.



(a) before placing the data in disk buffers

(b) when passing the data to scan operator (current VectorWise [ZHNB07])

(c) when required by query executor to operate on data (compressed execution)

Figure 2.9: Different stages at which decompression can be performed.

2.4.1. Early research

The idea of compressed execution was first proposed by Goetz Graefe and Leonard D. Shapiro in their work from 1991 [GS91]. They focused mainly on reducing the usage of memory space, because spilling data to disks during joins, aggregations and sorting was a major issue at the time. Hence, they proposed to delay the decompression as long as possible. We call this approach *lazy decompression*, as opposed to traditional *eager decompression*.

Graefe and Shapiro assumed that each value is encoded separately, and that a fixed encoding scheme is used for each domain. They pointed out that a number of operations can be performed directly on the values compressed this way. The codes can be used for equality checks, as join attributes, for duplicate elimination and as grouping attributes in aggregations. Moreover, when values have to be copied, for example during projections or materialization of join results, the decompression of values would even be harming, since it would increase the amount of data to copy.

We should notice that this work had a purely theoretical character. The authors did not implement the proposed solutions, but just made estimations based on a simple model taking into account the number of I/O operations.

2.4.2. Avoiding decompression

There were many studies that focused on delaying or avoiding decompression. Their authors did not aim to actually perform operations on compressed data, but rather to analyse when and how the data should be decompressed.

Westmann et al. proposed in [WKHM00] to modify the standard `next()` method, so that it returns not only a pointer to the tuple, but also a set of values from this tuple. This set would initially be empty, but whenever a value from a given tuple would be decompressed, it would be added to the set. This way no value would have to be decompressed twice.

Chen et al. in [CGK01] suggested to use *transient operators*, that is, ones that internally work on decompressed values, but return compressed results. They argued that lazy decompression can lead to suboptimal execution times, since once the data gets decompressed, it

Predicate to evaluate `A = 0 AND C=2`

Placement of codes in a bank

empty	A	B	C
1 bit	2 bits	2 bits	3 bits

Translating constants into codes

A_dictionary[0] = `11`
C_dictionary[2] = `101`

Resulting expression to evaluate `(x BIT-AND 01100111) == 01100101`

Figure 2.10: Example of converting conjuction of predicates into bit-wise operations on banks.

stays decompressed, which can increase the size of intermediate results and the number or required disk accesses for following operations like joins and sorting.

They also stressed the importance of taking compression into account while searching for the best query plan. The query optimizer should decide when to use eager, lazy or transient decompression, and adjust its estimates accordingly.

2.4.3. Simultaneous predicate evaluation

Johnson et al. argued in [JRSS08] that queries in decision support systems usually consist of an aggregation over a subset of tuples which have to meet tens of predicates. They give an example of a `WHERE` clause of a query from customer workload which illustrates this property:

```
WHERE A = a AND B = b AND C IN (c, d) AND D = 0
      AND E = 0 AND F <= 11 AND G >= 2
      AND H IN (e,f) AND (I IN (g,h) OR I > j)
      AND { nine other clauses }
```

They pointed that efficient evaluation of those predicates is critical for query performance and proposed a method of compressed execution designed for that purpose.

Johnson et al. proposed to use an order-preserving dictionary encoding for all of the attributes. To make the compression more efficient, they partitioned the relations horizontally, and used a separate set of dictionaries for each of the partitions.

The codes were packed into 128-bit wide banks, each containing values from many columns. The banks for a given set of columns were stored one after another. In other words, this solution does not use row-wise or column-wise, but a special bank-wise storage.

Johnson et al. decided to translate the predicates on attributes to bit-wise operations on banks. A single CPU instruction operates on the entire bank, with many encoded attributes. This way a number of predicates can be evaluated in parallel. The translation rules are in general much too complicated to describe here, but we provide an example of one of the rules in Figure 2.10 to give the reader an idea of how such parallel evaluation can be executed.

Unfortunately, this approach is designed for a relatively small class of queries, and it is not clear whether it can perform well in more general workloads. Moreover, it implies very serious modifications in the execution and storage layer. It allows only a single compression scheme and needs a specialized bank-wise storage. Also, the series of operations on banks have to be compiled for each of the partitions separately.

2.4.4. Code explosion problem

Abadi et al. focused on integrating a number of compression schemes into a column store [AMF06]. They wanted to actually operate on the compressed data whenever possible, not just avoid the decompression. However, they observed that if no preventive steps are taken, then with each new compression technique the operators code will become more and more complex, since new special cases will have to be implemented for different combinations of compression schemes in the input attributes. Also, they wanted to avoid exposing the details of compression to operators as much as possible.

To solve those problems, Abadi et al. decided to wrap the compressed data in special objects — *compressed blocks* — that would provide a uniform interface for accessing the data and its properties. This way the code of operators was not depending on implementation of individual compression schemes, but only on compressed blocks interface.

Abadi et al. proposed three Boolean properties of compressed blocks. The `isOneValue` property was set, if all values in a given block were identical. The `isValueSorted` denoted that values in a given block were stored in a sorted order. The `isPosContig` was true, if the values formed a consecutive subset of a column. These three properties were used to decide on the optimal handling of data by an operator.

As an example, consider properties of RLE-compressed blocks, each containing a single run. All values in a run are identical, and equal values are naturally sorted, so `isOneValue = TRUE` and `isValueSorted = TRUE`. Moreover, each run is a continuous fragment of a column, hence `isPosContig = TRUE`.

Access to the data was possible through one of two functions: `getValue()` and `asArray()`. The first one transiently decompressed a value, and returned it together with its position in the table. The latter one decompressed all the values in a compressed block and returned them as an array.

2.4.5. Compression techniques

We will now discuss the compression methods chose for compressed execution.

- Definitely the most popular encoding scheme was *dictionary encoding.* It was used in all the above approaches because of ease of decompression and possibility of checking equality of values without decompression. If the encoding was order-preserving, the range comparisons were possible as well. The study of Abadi et al. recommends this scheme for all the attributes with anywhere between 50 and 50.000 values, that do not have runs that could be exploited by RLE.

- The *RLE* was considered by Abadi et al. the best choice for all the attributes that have runs of average length at least 4. A number of operations — comparisons, hashing, computing aggregates, duplicate elimination — could be performed once per run, instead of once per value. However, this scheme was useful only for column stores, because of difficult decompression in row stores.

- Abadi et al. pointed that *bit-vector encoding* is well suited for attributes with small number of distinct values that does not compress well with RLE. Similar to RLE, the bit-vector encoding provides groups of identical values. However, it is only useful if we accept processing values out of order.

- The *Lempel-Ziv* compression techniques were used as well, but the only benefit that can be achieved with them is by delayed decompression, because they do not expose any useful properties of data.

In addition, for some of the compression schemes special variants were designed, to make them more useful in compressed execution. For example, Holloway et al.proposed in [HRSD07] a method of reordering the nodes of Huffman tree, which allows range comparisons without decompression.

2.5. Compressed execution in context of VectorWise

Our objective was to integrate compressed execution into the VectorWise DBMS. We wanted to achieve significant performance improvements, but also to avoid any major and invasive modifications of the database engine. We wished the changes to be as transparent and non-intrusive as possible, and likely to be implemented in a limited time. We focused on benefiting in the most common cases, rather than on chasing every bit of optimization opportunity. We reviewed the solutions for compressed execution described in previous section to determine if they can be successfully adapted to our needs. We summarize our observations below.

Chen et al. and Westmann et al. focused on lazy and transient decompression. However, the savings in decompression time are becoming less important with the advent of multi-core processors and growing available CPU power. The disk spills during joins or aggregations are problems of small concern in VectorWise, since these operations are usually executed within main memory. Moreover, the decompression in VectorWise is performed at the granularity of a vector, because decoding each value separately would introduce a significant overhead. Since it is relatively unlikely that no value in a vector will be used, a vectorized database system creates much fewer opportunities to avoid decompression. Hence, we do not anticipate any notable performance improvements to be achieved using these methods.

The solution proposed by Johnson et al., that is, simultaneous evaluation of predicates using banks, is not well suited for VectorWise. First of all, to employ it, the storage layer would have to be converted to store the data not column-wise, but bank-wise, which is a major modification to the DBMS. Second, it allows only dictionary compression and demands that encoding is fixed for horizontal partitions. This in not in line with the design of VectorWise storage layer, where each column is independent from the others. Most importantly, this approach only covers a very limited set of queries. VectorWise is intended for more general workloads, and turning to the bank-wise storage would most probably hurt the performance of other queries.

The approach presented by Abadi et al. poorly fits the architecture of VectorWise engine. Implementing it would require a major reengineering effort. It is also not clear how to make this solution compatible with vectorized execution model without hindering the performance. The compressed blocks may contain arbitrary number of values, while the operators in VectorWise expect identical number of values in each of the input vectors. Hence, the vectors would have to be represented by a collection of compressed blocks and some of the compressed blocks would have to be split between the vectors. Operating on such collections would introduce prohibitive overheads.

Moreover, Abadi et al. assumed that a large number of compression schemes would be used and focused on preventing code explosion. However, for most data distribution only a very small subset of compression algorithms provides optimal or close to optimal benefits. Our aim is not to employ as many compression techniques as possible, but to find and employ the ones that are really useful.

Our investigation suggests that compressed execution provides significant performance optimization potential, but the techniques proposed so far does not seem to fit the VectorWise system. For that reason, we decided to search for alternative methods. In the reminder of

this thesis we will describe techniques that achieve most of possible benefits, and at the same time are non-intrusive and easy to implement.

2.6. Summary

This chapter explained the concept and architecture of database management systems. We discussed the role of data compression in DBMSes and presented the idea of compressed execution. We also described the basic aspects of VectorWise — an analytical database system on which we decided to focus in this project. Then, previous research on compressed execution was reviewed in context of VectorWise. We concluded that none of solutions proposed so far fits well the architecture of this DBMS. Therefore, in order to integrate compressed execution in VectorWise and avoid major reengineering, we have to seek for new methods.

Chapter 3

Run-Length Encoding

Run-Length Encoding, described in Section 2.2.1, allows compressing sequences of repeating values. This encoding method was shown to be particularly useful for compressed execution purposes in previous research [AMF06].

In this section, we will discuss the difficulties of making a vectorized database engine capable of operating on RLE-compressed data. We will then propose a solution inspired by RLE, that allows achieving significant performance improvements while having a minimal impact on the database engine.

3.1. Challenges of RLE-compressed execution

Introducing execution on RLE-compressed vectors into a database engine is a task with many pitfalls. We will now describe some of the basic challenges that must be faced during this process.

3.1.1. Achieving compatibility with original code

The use of RLE-compressed data during query execution requires making the whole engine able to deal with a new data representation. Instead, one could try reusing existing code. This can be achieved by providing a function that converts an RLE-compressed vector into an ordinary one and then making sure that each operator or expression that cannot work directly on RLE-compressed data will use this function before accessing the content of a vector. Depending on how the database engine is actually implemented, the second part may or may not be a trivial task.

3.1.2. Increase in amount of code

In order to benefit from RLE-compressed execution, we have to extend the engine with new, RLE-aware code. The volume of code that has to be introduced can prove surprisingly large.

For example, let us consider a simple function that subtracts two vectors (a primitive `map_-_sint_col_sint_col()`). To make it RLE-aware, we have to provide four versions of this function: one operating on two ordinary vectors, one operating on two RLE-compressed vector, and two operating on one ordinary and one RLE-compressed vector. If we want to have a choice between producing a result that is either ordinary or RLE-compressed vector, the number of versions doubles.

The different variants can often be automatically generated, but nonetheless such an increase in the amount of code is undesired and should be avoided if possible.

3.1.3. Selection vectors

Selection vectors, described in Section 2.3.1, create a challenge for RLE-compressed execution. Due to how the selection vectors are represented in VectorWise it cannot be determined in a constant time if at least one value in a given run is selected. Hence, without searching through the selection vector we cannot tell if computations for a given run should be performed at all.

For some operations this problem can be evaded by ignoring the selection vector. The computations are then performed for all runs, as if all values were selected. We call this method *full computation*.

Unfortunately, full computation is only suitable for a limited set of operations which cannot result in failures. For example, execution of primitive `map_==_sint_col_sint_col` will always succeed, because comparing any two numbers will produce a valid result. On the other hand, many operations does not have this property. For example, full computation for primitive `map_/_sint_col_sint_col` may result in a 'division by zero' error.

3.1.4. Variation in data clustering

It may happen that some parts of a column are well suited for RLE-compression (identical values are clustered together), while others are not. Using RLE-compressed vectors to represent data from the first ones can bring performance improvements, while using them for the latter ones will only create an additional overhead.

Therefore, we should allow a scan operator to produce a mix of ordinary and RLE-compressed vectors, depending on their content. As a result, each and every expression and operator must be capable of handling any sequence of different kinds of vectors. A proper implementation cannot be chosen upfront while building an operator tree — it has to be selected dynamically during query execution.

3.2. Proposed solution

Our objective was to keep part of the benefits of RLE-compressed execution while limiting the impact on a database engine, simplifying implementation and avoiding the challenges described above. The RLE-compressed execution achieves reduced CPU-cost by performing operations once per run — a group of consecutive equal values — instead of once per value, and we aimed to exploit the same mechanism.

We propose not to introduce a new data representation for fully RLE-compressed data. Instead we shall focus entirely on vectors that hold values that are all equal, i.e. vectors that consist of a single run. We will call them *constant vectors*. In order to denote this property, we only have to add a single Boolean flag to a vector data structure. However, unlike with RLE, even a constant vector will still have to contain all values in order to ensure that it can be safely used in the same way as all the other vectors.[1]

This approach makes it easier to overcome the challenges described before. The problem of code compatibility is evaded by using a single data representation. The original code will be oblivious to whether vectors are constant or not. These parts will not achieve any performance improvements, but at least will not lose correctness. The increase of amount of code can be limited by reusing code that is already a part of the system. This process is described in Section 3.2.2. The presence of selection vector is no longer a problem: in

[1]This assumption is made to avoid engineering difficulties emerging due to how VectorWise was implemented. We will later show that it actually causes a significant overhead and discuss abandoning it.

a constant vector all values are equal, so it only matters *how many* values are selected, but not *which ones* exactly. Finally, since there is no structural difference between constant and non-constant vectors, this method is not spoiled by variations in data clustering.

3.2.1. Detecting constant vectors

The scan operator should mark some of produced vectors as constant. It is not necessary to detect all of the constant vectors, as false negative may only harm performance, but not correctness. We should aim at marking as many as possible while keeping the CPU-cost low.

Non-string attributes We first decided to only try to find constant vectors for data stored in RLE-compressed disk blocks. There are two reasons for this:

1. Checking if a vector is constant during RLE decompression is very cheap in terms of CPU-time, while the same is not true for other compression schemes in VectorWise.
2. For a column that can produce a significant fraction of constant vectors (i.e. has plenty of very long runs) RLE is most probably the best suited compression scheme. Then columns for which a different compression scheme is better suited are less likely to give us a lot of constant vectors.

String attributes A workaround was required for string attributes, because there is currently no implementation of RLE-compression for strings in VectorWise.

We decided to manually check each vector produced by scan operator, i.e. compare its values one-by-one. This introduces a significant overhead, that is unacceptable in production code. However, this overhead is easy to accurately measure. It is also easy to avoid by implementing an appropriate compression scheme. Hence, we decided that this method can be used for the purpose of this thesis.

3.2.2. Changes in expressions

We will now discuss how to modify the expression object, so that it can seize the opportunities provided by constant vectors.

Primitives

As we explained in Section 2.3.1, expressions have an assigned primitive that they are responsible for calling. Each kind of primitive comes in many variations, depending on what kind of arguments it is intended for. The arguments may be vectors (represented by `col` in primitive's signature) or constants (represented by `val`).

Primitive swapping

Let us notice that constant vectors can often be treated as if they were just a single value.

For example, consider primitive `map_+_dbl_col_dbl_col`. It can be intuitively written as:

```
for(i = 0; i < n; i++)
   result[i] = param1[i] + param2[i];
```

Assume that the second argument is a constant vector. There is no need to actually read the content of this vector. Instead we can do:

```
for(i = 0; i < n; i++)
   result[i] = param1[i] + *param2;
```

This approach should be beneficial, as it requires moving less data from memory to CPU registers and can improve cache utilisation.

The second listing is simply the `map_+_dbl_col_dbl_val` primitive. Hence, a similar primitive — one that has a `val` instead of `col` in its name — can be called instead of the default one. The same idea can be applied to many different primitives.

We propose to choose the most appropriate primitives dynamically during query execution. This decision should depend entirely on whenever argument vectors are constant or not. We call this process *primitive swapping.*

Implementation

In order to make primitive swapping possible, we have to modify the expression object. Those changes can be divided in two categories: those that come into effect when the operator tree is constructed, and those that come into effect whenever a primitive has to be executed.

- *Build phase*

  During the construction of the operator tree, the expressions determine which primitives they should use. Normally they only choose one primitive, and use it during the whole query execution.

  However, in our proposal they have to retrieve not only a single primitive, but also all the primitives that it can be swapped for. Determining which primitives should be selected is not difficult. After finding the default primitive, we can easily construct their names through simple string operations (substituting `col` for `val` in the primitive name).

  This process might be laborious, but it only takes place once per expression.

- *Execute phase*

  Every time an expression is evaluated and a primitive has to be called, the argument vectors should be examined. Depending on whether any or all of those vectors are constant, an appropriate primitive has to be chosen: the more `val`s in the name (instead of `col`s) the better, but each new `val` must match an argument that is a constant vector.

  Making this decision has to introduce some additional interpretation overhead. However, it can be effectively implemented. We propose to prepare a table with choices for all possible combinations of constant and non-constant argument vectors at build time. Each argument vector would correspond to one bit in table index. In order to choose the right primitive, we'd only have to iterate over all argument vectors and set a proper bit for every non-constant one. An example of this procedure is presented on Figure 3.1.

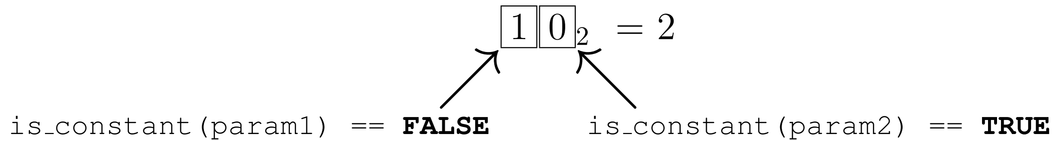
Caveats

The described approach allows us to 'recycle' code that is already a part of the system for compressed execution. However, it also has a certain disadvantage: the result type (a vector or a value) of a primitive is determined implicitly from the types of its arguments. This

(a) Table of primitive choices (created at build time):

Index	Primitive
$00_2 = 0$	map_add_sint_**val**_sint_**val**
$01_2 = 1$	map_add_sint_**val**_sint_**col**
$10_2 = 2$	map_add_sint_**col**_sint_**val**
$11_2 = 3$	map_add_sint_**col**_sint_**col**

(b) Computing index (performed for each expression evaluation):



(c) Selected primitive:

$10_2 = 2$	map_add_sint_**col**_sint_**val**

Figure 3.1: Choosing best primitve in expression with `map_add_sint_col_sint_col` function.

means that, for example, the result of the primitive `map_+_dbl_val_dbl_col` is a vector, while the result of `map_+_dbl_val_dbl_val` is just a single value.

We decided to keep an original data representation, and therefore each constant vector actually has to contain all those equal values. Hence, whenever swapping of primitives affects the result type (changing it from a vector to a value), we have to manually 'fill' the result vector with copies of value returned by a primitive. Luckily, those situations are easy to identify.

The necessity of 'filling' constant vectors introduces a significant overhead, but it does not make using constant vectors pointless. We will examine this overhead and explain how to remove it in Section 3.3.4.

The performance of constant vectors optimization strongly depends on choosing correct vector size for a given data. We will explain this problem in Section 3.4.

3.2.3. Impact on operators

We discussed the modifications of expressions needed to make them aware of constant vectors. Those changes alone can bring notable performance benefits. However, not all optimization opportunities of constant vectors can be exploited by altering only the way expressions are evaluated. While some operators — e.g. Project — can benefit fully from constant vectors without any changes to their code, others require some modifications.

As an example we will now examine the aggregation operator.

The work of Aggr operator proceeds in two phases:

- a *build phase* when it consumes the input data and computes all aggregates,

- and a *produce phase* when it outputs the results of the previous phase.

There are no opportunities to use constant vectors in the latter, so we will focus on the build phase.

The first step in the build phase is to divide the input into groups that refer to certain values of attributes from `GROUP BY` clause. Depending on the type of those attributes, this operation can be performed in one of two ways:

- *Direct aggregation* — the group number is computed arithmetically from the values of group-by attributes. This computation is performed using expressions, so it benefits from constant vectors without any modifications.

- *Hash aggregation* — the group number is determined using a hash table. First, the hash values for all group-by attributes are computed through a proper expression. Then a lookup-or-insert function working on a hash table is called. For constant vectors it is enough to perform this operation once for entire vector, instead of repeating it for each value in a vector. Therefore an additional `if` statement should be added to hash aggregation code.

The second and final step is updating the aggregates. Here primitives computing sum, minimum, maximum, count, etc. are called. For this purpose expressions are used and constant vectors optimizations are applied automatically.

3.3. Experiments

3.3.1. About the benchmark

We decided to test the constant vector implementation against real-life data. We used records of USA domestic flights from October 1988 to August 2009 [Res11]. The data consisted of a single table `Ontime` with ca. 125 millions of rows.

We run a set of 10 queries, that we will refer to as *On-Time benchmark*. They were previously used to compare performance of different analytical DBMSs [Tka09, Ker10].

Ontime table

The `Ontime` table had 93 columns of various types, but only a small subset of them was used during the benchmark. Below we present the schema definition of `Ontime` table with unused columns omitted.

```
CREATE TABLE "Ontime" (
   "Year"             SMALLINT     DEFAULT NULL,
   ...
   "Month"            TINYINT      DEFAULT NULL,
   ...
   "DayOfWeek"        TINYINT      DEFAULT NULL,
   ...
   "Carrier"          CHAR(2)      DEFAULT NULL,
   ...
   "Origin"           CHAR(5)      DEFAULT NULL,
   "OriginCityName"   VARCHAR(100) DEFAULT NULL,
   ...
   "DestCityName"     VARCHAR(100) DEFAULT NULL,
   ...
   "DepDelay"         DECIMAL(8,2) DEFAULT NULL,
```

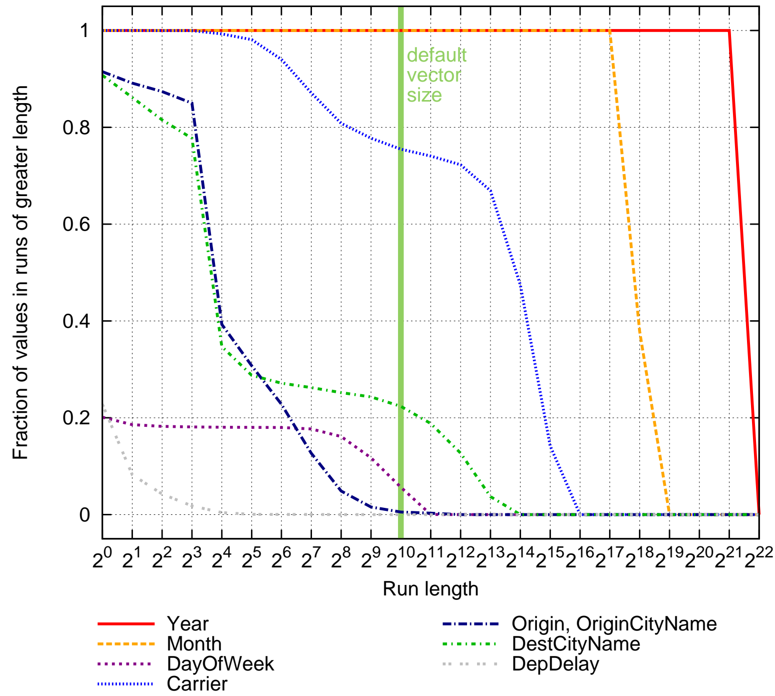


Figure 3.2: The value runs in On-Time dataset.

```
    ...
);
```

Value runs

We decided to investigate to what extent different columns of the `Ontime` table are suitable for RLE-compression. We examined lengths of value runs in each of them. The data was not sorted, but instead kept in the same order in which it was provided. The results are presented in Figure 3.2.

Since the data is naturally sorted on `Year` and `Month`, these two columns provide the longest runs of equal values. The scan operation on those columns should produce almost exclusively constant vectors. Therefore, we decided to use RLE-compression for `Year` and `Month`.

`Carrier` is a string column that consists of relatively long runs — 67% of all values are in runs of length at least 2^{13}. We can anticipate that a scan of `Carrier` will return a significant fraction of constant vectors if only the vector size is not greater than 2^{12}.

Finally, `DestCityName` could provide a bit more than 20% of constant vectors for vector size 2^{10}. The other columns do not create an opportunity to use our optimization.

Note that the plot lines for `Carrier` and `DestCityName` have middle sections that are notably less steep. It is a consequence of non-homogeneous data clustering in the dataset. All data is sorted on `Year` and `Month`, but then within a single month it may be clustered in many ways. This results in dramatically different run lengths in parts of columns belonging to different months.

Column name	Constant vectors
`Carrier`	87.3–96.2%
`DayOfWeek`	0.0%
`DepDelay`	0.0%
`DestCityName`	41.7%
`Month`	99.8%
`Origin`	1.0%
`OriginCityName`	0.3%
`Year`	99.9%

Table 3.1: Fraction of constant vectors produced during execution of On-Time queries. Note that these values differ from ones presented in Figure 3.2 due to non-homogenous data clustering.

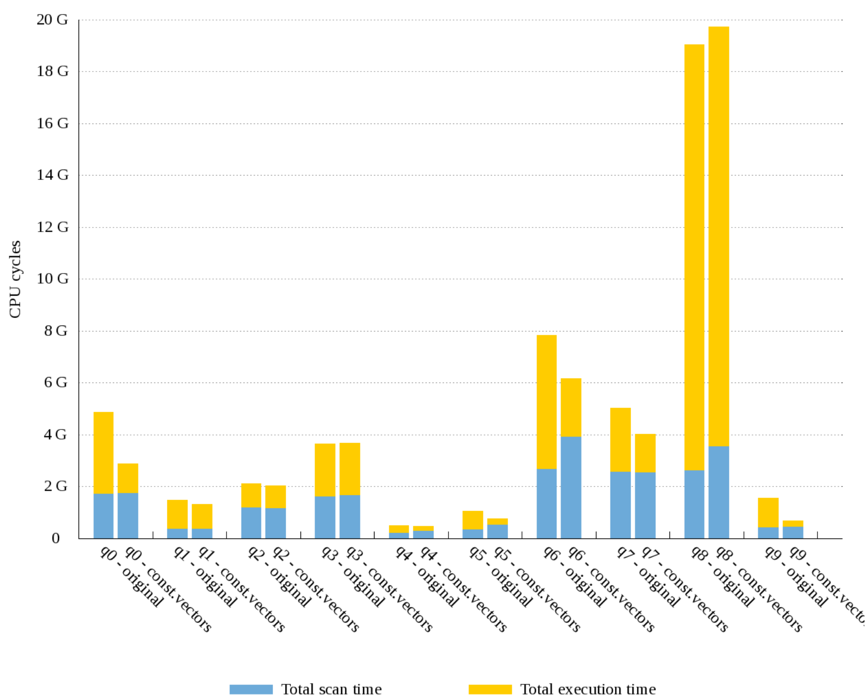


Figure 3.3: Execution times of queries from On-Time benchmark with and without constant vectors optimization.

3.3.2. Results

All test were executed on Intel(R) Quad Core CPU Q6600 @ 2.40GHz platform with 8 GB RAM. The reported results are mean times of query execution with hot disk buffers, i.e. no data had to be read from disk. We used the default vector size of 1024.

The results for entire benchmark are presented in Figures 3.3 and 3.4, as well as listed in Table 3.2. The queries and their operator trees are shown in Figures from 3.5 to 3.14. The fractions of constant vectors produced from each column while executing those queries are listed in Table 3.1. We will now discuss benchmark results in detail.

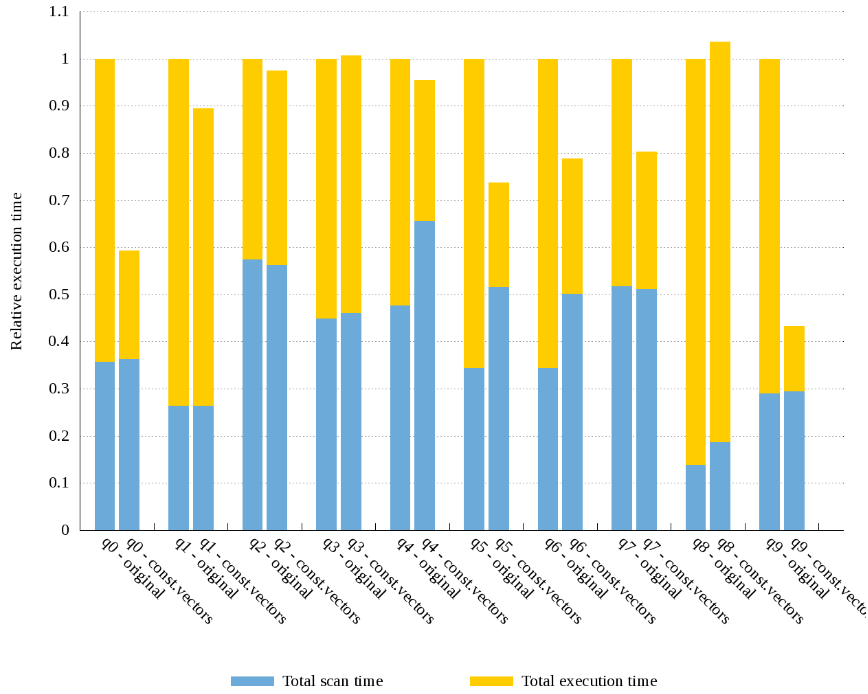


Figure 3.4: Execution times of queries from On-Time benchmark while using constant vectors compared to original VectorWise results.

Query	Q0	Q1	Q2	Q3	Q4
Execution time:					
– original	4869.6	1489.8	2106.2	3662.0	496.6
– with constant vectors	2884.5	1332.0	2051.5	3685.7	474.2
– change	−40.7%	−10.6%	−2.6%	+0.6%	−4.5%

Query	Q5	Q6	Q7	Q8	Q9
Execution time:					
– original	1053.0	7838.3	5020.2	19049.9	1570.6
– with constant vectors	778.8	6213.2	4111.7	19813.0	723.9
– change	−26.0%	−20.7%	−18.1%	+4.0%	−53.9%

Table 3.2: Execution times of queries from On-Time benchmark.

Query 0

The majority of query execution time is spent in direct aggregation Aggr[1] that uses `Year` and `Month` as group-by columns, and consumes ca. 125 millions of tuples. The use of constant vectors reduces the aggregation time by 65%.

As we described in Section 3.2.3, the direct aggregation consists of two steps: (a) computing identifiers of groups to which each tuple belongs (through `map_directgrp*` primitives), and (b) updating appropriate aggregates (through `aggr*` primitives). The first part can be only slightly improved by use of constant vectors, because vector filling is necessary. However, the second part benefits greatly, because for `aggr*` primitives there is no such need.

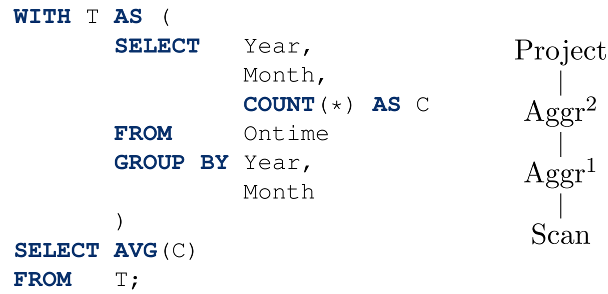


Figure 3.5: Query 0 from On-Time benchmark and its operator tree.

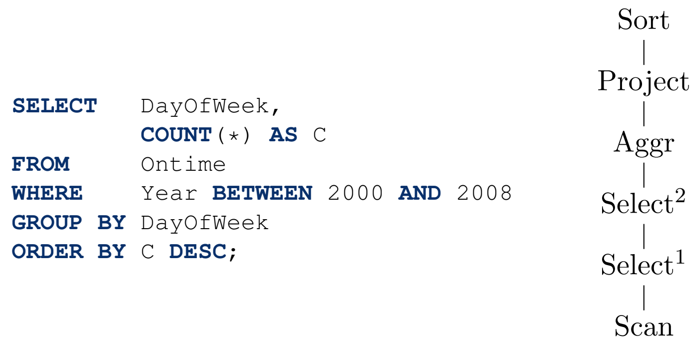


Figure 3.6: Query 1 from On-Time benchmark and its operator tree.

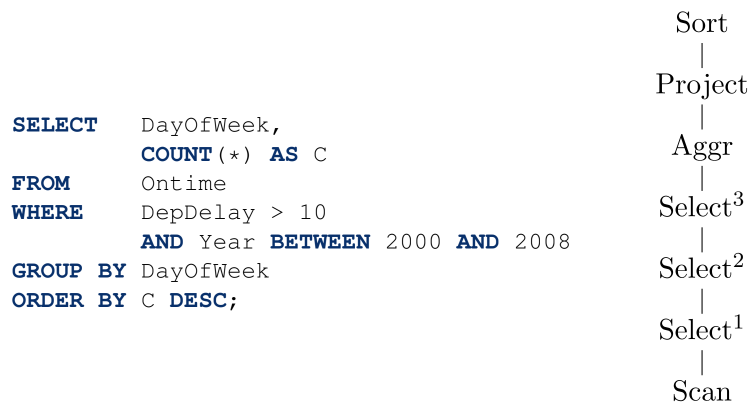
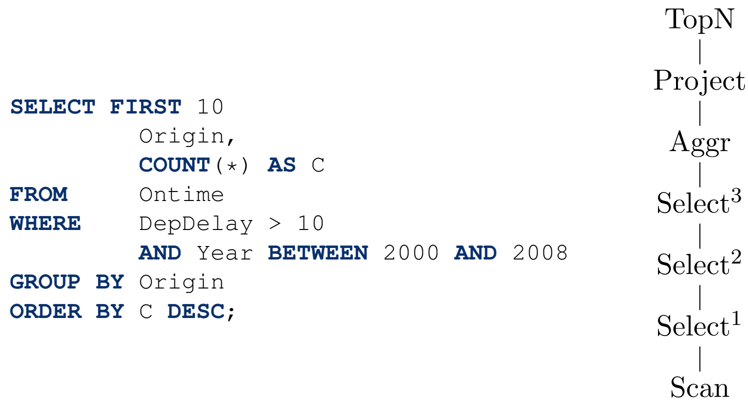
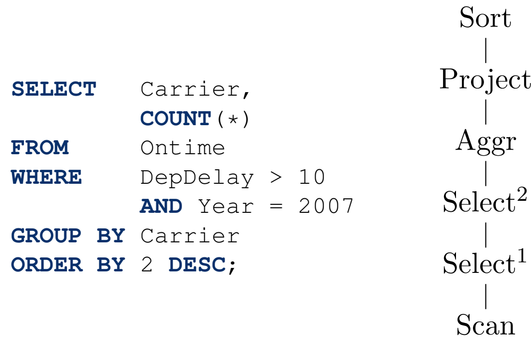


Figure 3.7: Query 2 from On-Time benchmark and its operator tree.



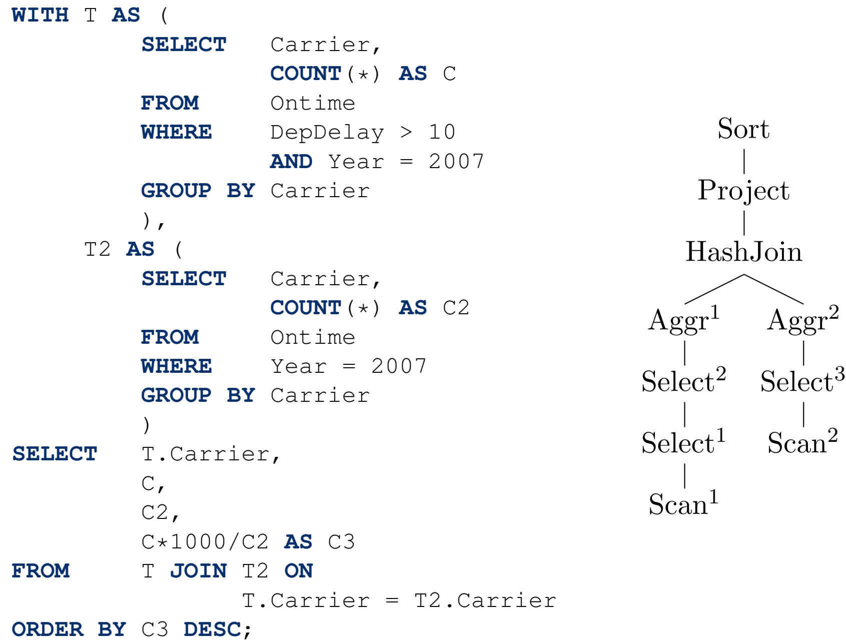
```sql
SELECT FIRST 10
           Origin,
           COUNT(*) AS C
FROM       Ontime
WHERE      DepDelay > 10
           AND Year BETWEEN 2000 AND 2008
GROUP BY Origin
ORDER BY C DESC;
```


Figure 3.8: Query 3 from On-Time benchmark and its operator tree.



```sql
SELECT     Carrier,
           COUNT(*)
FROM       Ontime
WHERE      DepDelay > 10
           AND Year = 2007
GROUP BY Carrier
ORDER BY 2 DESC;
```


Figure 3.9: Query 4 from On-Time benchmark and its operator tree.



```sql
WITH T AS (
         SELECT     Carrier,
                    COUNT(*) AS C
         FROM       Ontime
         WHERE      DepDelay > 10
                    AND Year = 2007
         GROUP BY Carrier
         ),
     T2 AS (
         SELECT     Carrier,
                    COUNT(*) AS C2
         FROM       Ontime
         WHERE      Year = 2007
         GROUP BY Carrier
         )
SELECT   T.Carrier,
         C,
         C2,
         C*1000/C2 AS C3
FROM     T JOIN T2 ON
                T.Carrier = T2.Carrier
ORDER BY C3 DESC;
```


Figure 3.10: Query 5 from On-Time benchmark and its operator tree.

```sql
WITH T AS (
         SELECT   Carrier,
                  COUNT(*) AS C
         FROM     Ontime
         WHERE    DepDelay > 10
                  AND Year BETWEEN 2000 AND 2008
         GROUP BY Carrier
         ),
     T2 AS (
         SELECT   Carrier,
                  COUNT(*) AS C2
         FROM     Ontime
         WHERE    Year BETWEEN 2000 AND 2008
         GROUP BY Carrier
)
SELECT   T.Carrier,
         C,
         C2,
         C*1000/C2 AS C3
FROM     T JOIN T2 ON
                T.Carrier = T2.Carrier
ORDER BY C3 DESC;
```

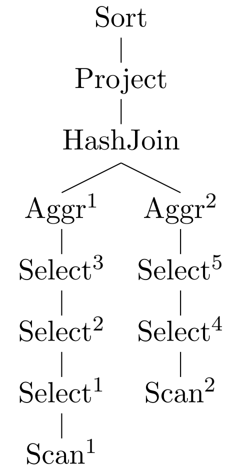


Figure 3.11: Query 6 from On-Time benchmark and its operator tree.

```sql
WITH T AS (
       SELECT   Year,
                COUNT(*) * 1000 AS C1
       FROM     Ontime
       WHERE    DepDelay > 10
       GROUP BY Year
       ),
     T2 AS (
       SELECT   Year,
                COUNT(*) AS C2
       FROM     Ontime
       GROUP BY Year
       )
SELECT T.Year,
       C1/C2
FROM   T JOIN T2 ON
            T.Year = T2.Year;
```

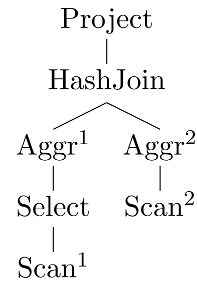


Figure 3.12: Query 7 from On-Time benchmark and its operator tree.

```sql
SELECT FIRST 10
         DestCityName,
         COUNT(DISTINCT OriginCityName)
FROM     Ontime
WHERE    Year BETWEEN 1990 AND 2000
GROUP BY DestCityName
ORDER BY 2 DESC;
```

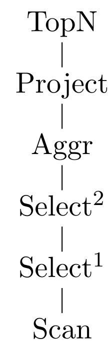


Figure 3.13: Query 8 from On-Time benchmark and its operator tree.

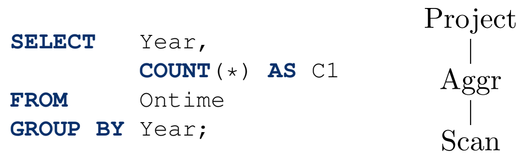


Figure 3.14: Query 9 from On-Time benchmark and its operator tree.

Query 1

The aggregation in query 1 takes over 30% of execution time. Unfortunately, it uses the `DayOfWeek` column which does not produce constant vectors. Thus our optimization does not give any benefits for this operator. However, the two selections on `Year` that constitute together 41% of execution time are improved by 18%. This results in a 10.6% shorter total time.

Query 2

The only operators that can benefit from constant vectors in this query are selections on `Year`, that is Select^2 and Select^3. They take 11% less time. However, Scan and Select^1 are much more expensive operators, so the impact on total time is only 2.6%.

Query 3

Both the aggregation operator and Select^1 are using columns that do not provide constant vectors, i.e. `Origin` and `DepDelay`. They cannot benefit from our optimization. The Select^2 and Select^3 work on the `Year` column. When using constant vectors they become 16% and 14% faster. However, only a very small part of execution time is spent on those operators, so it has negligible impact on total time. The query is now 0.6% slower.

Query 4

The Select^2 operator (originally 11% of total time) works on `Year` column. Our optimization reduces its time by 27%. The aggregation (originally 26% of total time) uses `Carrier` as a group-by column. The data for year 2007 in this column forms long runs (as we mentioned before, data clustering is different in different parts of data set), so it provides 96.2% of constant vectors. The aggregation time is now reduced by over 70%. The other operators do not benefit from constant vectors. Due to a slowdown of scan operator (we will discuss it in Section 3.3.3) we only achieve a 4.5% improvement in query execution time.

Query 5

There are two selections on `Year` column: Select^1 and Select^3, that constitute 10% of original query time. They were improved by 43%. The Select^2 operator works on `DepDelay` and it did not benefit from our optimization. The two aggregations originally took almost half of execution time. They use `Carrier` as a group-by column which provides 96.2% of constant vectors. Now Aggr^1 is 71.5% faster and Aggr^2 is 85.4% faster. However, the query performance suffers from spending more time in the scan operator. The total time was reduced by 26.0%.

Query 6

There are four selections working on `Year` column in this query: Select^2, Select^3, Select^4 and Select^5. They were taking 11% of query time. They were improved by 16.3%. The Select^1 operator uses `DepDelay` column and cannot benefit from constant vectors. Both aggregation operators use `Carrier` as group-by column. It provides 87.3% of constant vectors. The aggregations are now 60.4% and 77.5% faster than before. Query execution time is now 20.7% shorter, even though query performance suffers due to slower scan.

Query 7

There are two aggregations using `Year` as a group-by column. They were previously taking 29% of query execution time. With our optimizations they are now 49.2% and 80.6% faster. The selection cannot be improved, because it works on column `DepDelay`, that does not include long value runs. This query is now running 18.1% faster.

Query 8

The query execution time is completely dominated by scan and aggregation operators. The aggregation cannot benefit from our optimization, because it first computes hashes for column `OriginCityName` that does not provide constant vectors. The query optimizer could choose different order of computing hashes: first for `DestCityName` (which gives over 40% of constant vectors), and then for `OriginCityName`. This way benefiting from constant vectors would be possible. The scan operation is now slower, which results in 4% longer total time.

Query 9

The aggregation using `Year` column constituted 70% of original query time. While using constant vectors this operator is over 80% faster. The total time was reduced by 53.9%.

Conclusions

We showed that 8 out of 10 queries of On-Time benchmark benefit from using constant vectors, and 6 of them are more than 10% faster than before. The optimization proved especially useful in a situation where constant vectors appear in group-by column of aggregation operator.

However, we also witnessed slowdown of scan operator in 4 queries. For this reason, query 8 takes 4.0% longer than before.

We now know that constant vectors can bring significant benefits, but this approach also has shortcomings that have to be eliminated. We will examine those flaws and methods of removing them in the following sections.

3.3.3. Detecting constant string vectors

We observed that the scan operators in queries 4, 5, 6 and 8 took much more time when using constant vectors than before.

As we stated in Section 3.2.1, each string vector produced by scan operator had to be examined, in order to determine whenever it is constant or not. The values had to be compared one-by-one, until either unequal values were found, or all values were checked.

This approach resulted in notable slowdown of scan operator in queries that use string columns, and especially those which provide a significant amount of constant vectors.

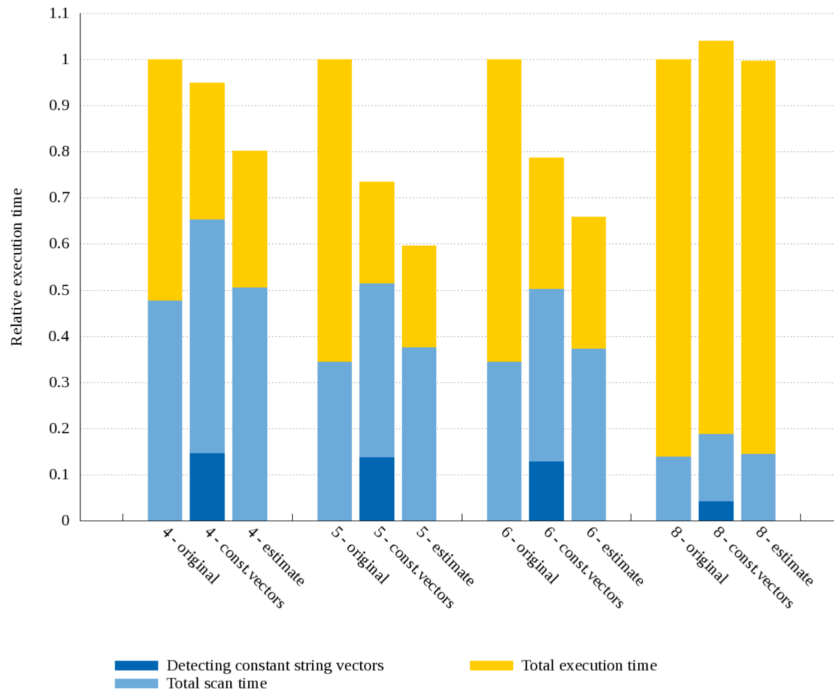


Figure 3.15: The overhead introduced by detection of constant string vectors and the estimated relative execution times after eliminating this overhead.

Impact on the benchmark results

We measured the amount of time the queries spend on examining string vectors in order to detect constant ones. The results are presented in Figure 3.15

Queries 4, 5 and 6 operate on `Carrier` column, that has a lot of very long runs. Detecting constant vectors contributed from 15.5% to 18.9% of execution time and had a big impact on query performance.

Query 8 uses columns `DestCityName` and `OriginCityName`, which do not provide so many constant vectors, and suffered less slowdown than other queries. It spend 4.1% of time on examining string vectors.

Solution

We propose to implement an additional compression scheme for VectorWise: string RLE with in-block duplicate elimination. It would provide a good compression ratios for columns like `Carrier`, as well as effective way to detect constant string vectors. We will describe how such compression scheme can be implemented.

Block organisation The data block should be divided in two parts: a *string segment* and a *RLE segment*.

The unique string values would be stored one-after-another in the string segment. Each string would have assigned a certain *offset*, i.e. number of bytes between the beginning of the block and the place where a copy of a given string is stored in the string segment. The offsets would be RLE-compressed and placed in the RLE segment.

We propose to place the string segment at the beginning of a block, and RLE segment at its end. The data in RLE segment should be stored starting from the end of the block, and moving backwards towards its beginning. This way, we do not have to know the final sizes of segments upfront in order to use all the available space effectively.

Compression During the compression phase we should only store unique string values in the string segment. Therefore we should keep a hash table that maps the strings to the offsets. A value would be placed in the string segment only if there is no entry for this value in a hash table. Then it would be inserted into the hash table with a proper offset. The offsets would be RLE-compressed just like ordinary numbers, and placed in the RLE segment.

Decompression During the decompression of a block, we would add the memory address of the first byte in this block to each offset. This way, we convert offsets to valid string pointers.

Detecting constant vectors This procedure can be performed effectively during vector decompression. This way it is much less time-consuming than examining all the values in a vector, because the number of equalities to check is smaller.

3.3.4. Filling vectors with constant value

As we described in Section 3.2.2, we decided that constant vectors must be completely compatible with original VectorWise code. For that reason, we had to ensure that constant vectors contain identical values in all their fields by duplicating the value in all positions. This method created an overhead that could be largely avoided.

Impact on benchmark results

There are two major moments when we are forced to do extra work in order to set all fields of constant vector to correct values:

- during decompression of a constant vector from an RLE-compressed block,
- when an expression swaps primitive so that only a single value is returned instead of a whole vector, e.g. when `map_+_dbl_col_dbl_val` is swapped to `map_+_dbl_val_dbl_val`.

We measured time spent on filling vectors in both cases. The results are presented at Figure 3.16. We see that the queries that benefit most from constant vectors — query 0 and query 9 — also suffer most from vector filling overhead. It takes them about 65% of time. The impact on query 8 is smallest (2.6%), because most of time it operates on data that does not form a lot of constant vectors. The other seven queries spend between 7.2% and 25.8% of time on vector filling.

Solution

In order to avoid vector filling it is necessary to make all the operators aware of constant vectors. Ideally, operators should use constant vectors to perform their tasks in the most efficient way. Implementing this is much easier than if we used RLE-compressed vectors, because constant vectors are a much simpler concept and do not introduce a new data representation.

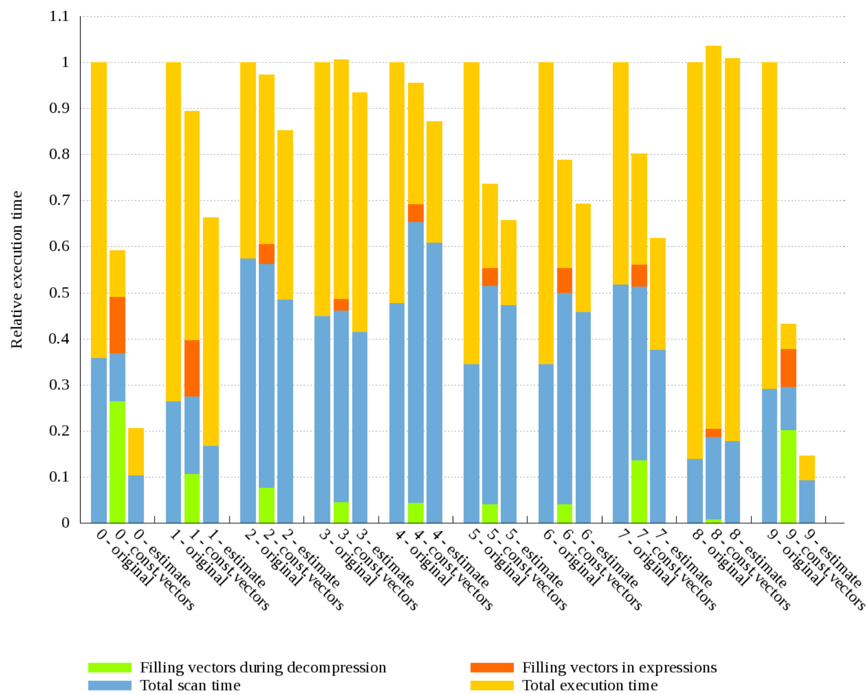


Figure 3.16: The overhead introduced by filling constant vectors with values and the estimated relative execution times after eliminating this overhead.

A simpler approach, that should still provide part of the benefits, is filling vectors lazily. In the current implementation an expression or operator that produces a constant vector is also responsible for filling it immediately. This way many vectors are filled unnecessarily.

An alternative is to identify all parts of code where appearance of not filled constant vectors could lead to failure. At the beginning of each of those parts, we have to place code that will fill vectors if necessary, e.g.

```
if (is_constant(vec) && !is_filled(vec))
   fill_vector(vec);
```

To demonstrate usefulness of this approach, consider following example:

```
SELECT   COUNT(*)
FROM     Ontime
GROUP BY Year, Month, Carrier;
```

In this query we have to perform a hash aggregation. This requires using a number of expressions to compute proper hashes.

First, an expression with primitive `map_hash_sint_col` is evaluated with vector from column `Year` as an argument. If a primitive was swapped to `map_hash_sint_val`, the result vector has to be filled. Then, an expression with primitive `map_rehash_uidx_col_schr_col` is evaluated with the previously computed hashes and a vector from the `Month` column as arguments. Again, if the primitive was swapped to `map_rehash_uidx_val_schr_val`, the resulting vector has to be filled. Finally, the same procedure applies third time.

Since all of the group-by columns provide the majority of constant vectors, very often vector filling with happen three times in above example. However, both expressions and aggregation operator are aware of constant vectors, so all the filling was unnecessary and in lazy approach, it would not be performed.

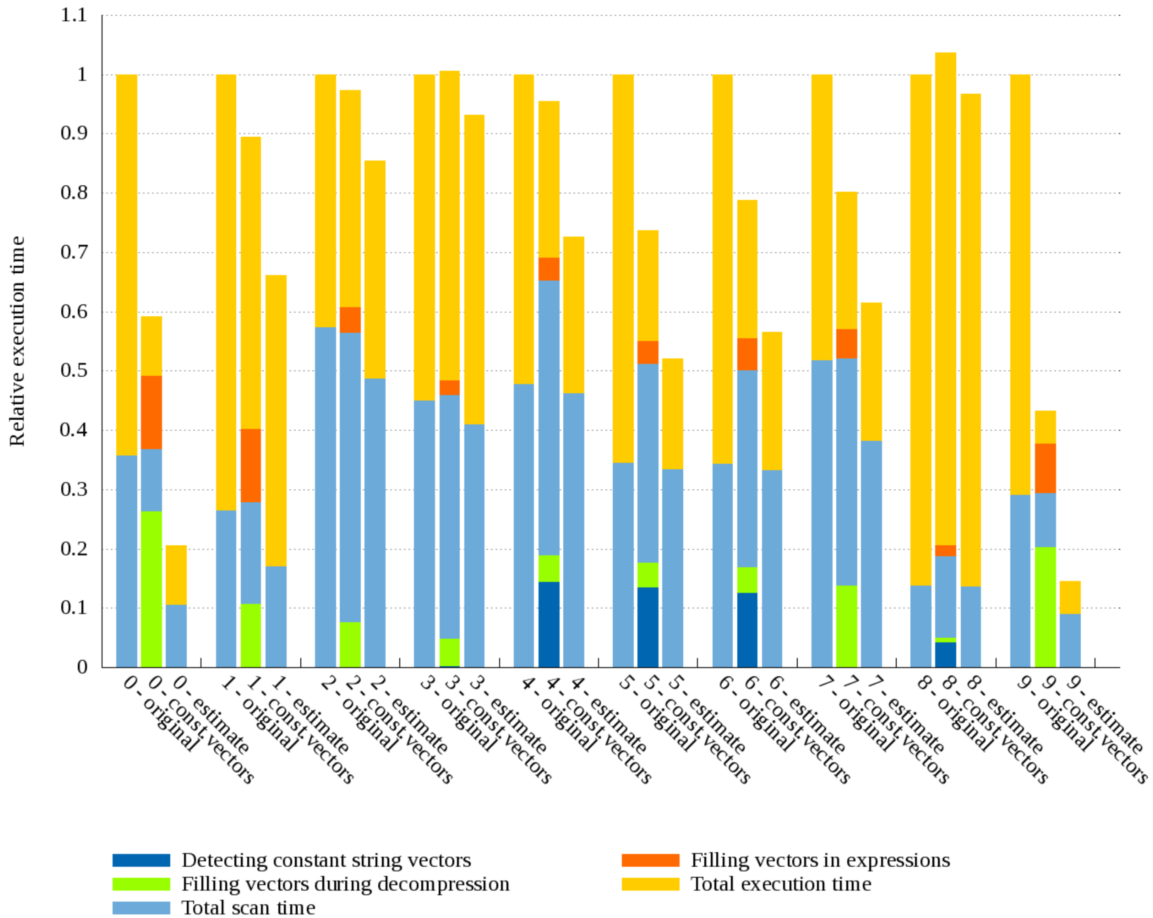


Figure 3.17: The performance improvements achieved with constant vectors optimization compared to estimated possible improvements after removing overheads described in Sections 3.3.3 and 3.3.4.

3.3.5. Conclusions from the experiments

The results of experiments with both described overheads included are listed in Table 3.3. They are also presented on Figure 3.17.

The benchmarks show big potential of constant vectors. Despite the simplicity of this solution, it manages to lower execution times of queries 0 and 9 by over 40%, and noticeably reduce execution times of queries 1, 5, 6 and 7.

The performance of constant vectors is hindered by overheads introduced by detecting constant string vectors and filling vectors with values. However, they can be removed as we described in previous sections. Those overheads can be precisely measured, so we can estimate the potential gains after eliminating them. The results demonstrate that removing the overheads would greatly improve the benefits of constant vectors optimization.

We expect to see execution times of queries 0 and 9 to be reduced fivefold. Queries 1, 4, 5, 6 and 7 should be improved by 25% to 50%. Some benefits should also be present for other queries.

After reviewing the benchmark results, we believe that constant vectors can be a valuable addition to VectorWise. They provide significant performance benefits while causing relatively small changes to the system.

Query	Q0	Q1	Q2	Q3	Q4
Exec. time with constant vectors [a]	−40.7%	−10.6%	−2.6%	+0.6%	−4.5%
Overheads: [b]	65.3%	26.0%	12.4%	7.4%	23.9%
– filling during decompression [c]	44.5%	12.1%	7.9%	4.5%	4.6%
– filling in expressions [c]	20.8%	14.0%	4.4%	2.6%	4.0%
– detecting const. string vectors [c]	0.0%	0.0%	0.0%	0.3%	15.2%
Est. exec. time w/o overheads [d]	−79.5%	−33.9%	−14.6%	−6.8%	−27.3%

Query	Q5	Q6	Q7	Q8	Q9
Exec. time with constant vectors	−26.0%	−20.7%	−18.1%	+4.0%	−53.9%
Overheads:	29.4%	28.2%	23.4%	6.7%	66.4%
– filling during decompression	5.5%	5.3%	17.3%	0.9%	47.2%
– filling in expressions	5.4%	6.8%	6.1%	1.7%	19.3%
– detecting const. string vectors	18.5%	16.2%	0.0%	4.1%	0.0%
Est. exec. time w/o overheads	−48.0%	−43.5%	−38.5%	−3.3%	−85.5%

[a] change between execution time of original VectorWise and VectorWise with constant vectors

[b] fraction of execution time with constant vectors spent on any of the overheads

[c] fraction of execution time with constant vectors spent on given overhead

[d] change between execution time of original VectorWise and estimated time for VectorWise with constant vectors after removing all overheads

Table 3.3: Performance of VectorWise with constant vectors optimization including overheads described in Sections 3.3.3 and 3.3.4.

3.4. Impact of the vector size

3.4.1. Effect on query execution performance

The performance of VectorWise depends strongly on the vector size [Zuk09]. We will explain this relationship by analysing the execution times (see Figure 3.18) of the following query:

```
SELECT   DestCityName,
         COUNT(*) AS C
FROM     Ontime
WHERE    Year BETWEEN 1996 AND 2000
GROUP BY DestCityName
```

Original VectorWise results First we will discuss the execution times of VectorWise without constant vectors optimization.

When the vector size becomes too small, VectorWise behaviour is similar to processing data tuple-at-a-time. Most of the CPU time is spent on traversing the operator tree, and not on actually operating the data. This results in performance degradation for vector sizes 2^8 and smaller (A).

When the vector size becomes too large, the vectors used during query execution do not fit in the CPU cache any more. The amount of data that has to be transferred between the RAM memory and the CPU cache grows significantly and damages query performance. This is reflected by growing execution times starting from vector size 2^{15} (B).

The optimal vector size depends not only on the size of CPU cache, but also on the number of vector objects that have to be allocated in order to execute a given query. In this

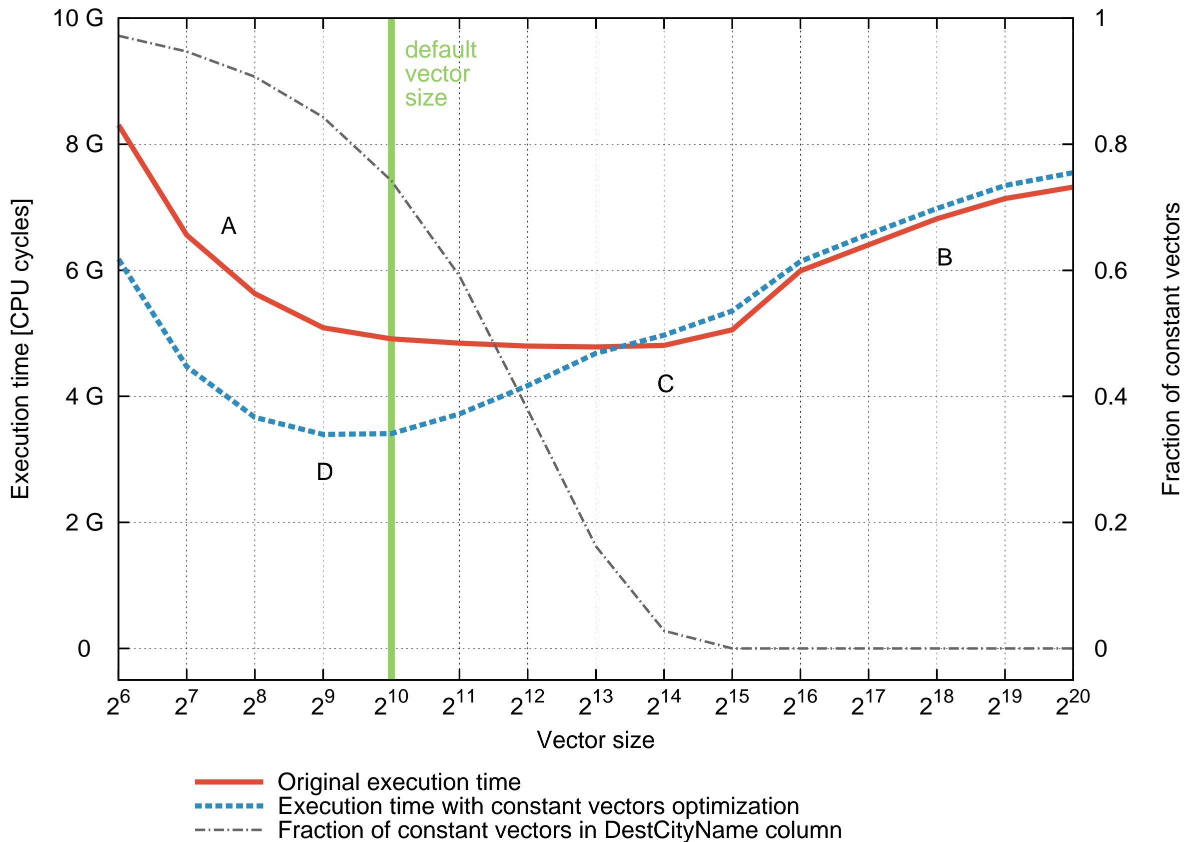


Figure 3.18: Execution times of query from page 39 with different vector sizes.

experiment the best results were achieved with vector size 2^{14} (C).

VectorWise with constant vectors results When using constant vectors optimization, we observe that too small or too large vector size hinders performance just as before.

However, now the vector size determines not only the amount of space required to store vector objects, but also the amount of constant vectors produced by scan operators. There is a clear trade-off between avoiding high interpretation overhead and creating the opportunities for applying constant vectors optimization.

The optimal vector size is now 2^9 (D). The hardware used and the number of required vectors was exactly the same as before, so growing execution times for vector sizes 2^{10}—2^{14} cannot be attributed to poor cache utilization. They are a consequence of missing a chance to benefit from constant vectors optimization.

3.4.2. Choosing the right vector size

It is relatively easy to choose a good vector size in VectorWise without constant vectors optimization. The decision depends entirely on available hardware and the complexity of executed queries. Since a wide range of vector sizes gives similar execution times — in Figure 3.18 this applies to range from 2^{10} to 2^{14}, and other queries show similar behaviour — setting a fixed system-wide vector size has proved to be a good enough solution.

The problem becomes much more complicated when we are dealing with constant vectors. Now the vector size determines not only the number of vectors that can fit in the CPU cache

and how well the interpretation overhead is amortized. It also influences the fraction of constant vectors produced by the scan operators. However, we should only be concerned to achieve a high portion of vectors marked as constant, if we can use this information to improve query execution performance, because otherwise it is completely worthless. While using constant vectors can speed up some operations substantially, others show little or no benefit.

There is a clear trade-off between reducing interpretation overhead and having more constant vectors. The optimal solution depends both on the characteristics of data, and on the operations that have to performed. To help us make the right decision, we may either store some statistics about the data or gather them incrementally during query execution and adjust our choices accordingly. To estimate benefits that can be achieved with different operations we can perform micro-benchmarks.

3.4.3. Variable vector size

In VectorWise different vectors produced by a scan operator in a single call of the `next()` method have to contain exactly the same number of values. This restriction comes from the fact that although such vectors hold values from different columns, those values belong to the same tuples. However, subsequent call of `next()` may result in producing vectors with different number of values, and also different scan operators are independent in this matter.

This observation leads to an agile solution of the problem of choosing best vector size: instead of reducing system-wide vector size, we can allow scan operators to produce smaller vectors, but only if it results in larger fraction of constant ones. This way we can seize the optimization opportunities, while avoiding heavy interpretation overhead.

3.4.4. Vector borders problem

Definitions and notation We will say that

$$v = (v_0, v_1, \ldots, v_{N_v})$$

is a *valid partitioning* for N tuples and vector lengths $m, m+1, \ldots, M$ if

$$\begin{array}{rl} (i) & v_0 = 0 \\ (ii) & v_{N_v} = N \\ (iii) & \forall_{1 \le i \le N_v} m \le v_i - v_{i-1} \le M \end{array}$$

In other words, v represents borders of N_v adjoining vectors. Each of those vectors contains between m and M values and together they contain all N values.

We will denote the number of values in constant vectors (assume that all such vectors are being detected and correctly marked) for column[2] $c = c_0, c_1, \ldots$ and valid partitioning v_i by $\phi(c, v)$. More exactly,

$$\phi(c, v) = \sum_{1 \le i \le N_v} (v_i - v_{i-1})[c_{v_{i-1}} = c_{v_{i-1}+1} = \ldots = c_{v_i - 1}]$$

where $[\ldots]$ is the Iverson bracket.[3]

[2] We only consider here scanning first N values of a given column. This simplifies the description, but the same reasoning is valid for any other subset of a column.

[3] Iverson bracket is defined as

$$[P] = \begin{cases} 1 & \text{if } P \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Problem statement Assume there are K columns $c^1, c^2, \ldots, c^K$ scanned together and K weights $w_1, w_2, \ldots, w_K$ associated with respective columns. Find a valid partitioning v for N tuples and vector lengths $m, m+1, \ldots, M$ that maximizes

$$\Phi(v) = \sum_{i=1}^{K} w_i \phi(c^i, v)$$

3.4.5. Optimization algorithm

Vector border problem can be broken down into smaller subproblems, that is, we can construct solution for problem size N using solutions for $N - M, N - (M - 1), \ldots, N - m$. We have to append vector of size M to solution for $N - M$, vector of size $M - 1$ to solution for $N - (M - 1)$, and so on. This way we get $M - m + 1$ candidate solutions for N. We just have to compute value of Φ function for each of them and pick the one that maximizes it. It can be easily proved that solution found this way is optimal.

The above reasoning can be applied recursively to further reduce the problem size. However, purely recursive solution would result in exponential complexity, which is clearly unacceptable. The complexity can be reduced to $O(N(M - m + 1)K)$ by using the methods of dynamic programming.

Algorithm Below we present pseudocode for algorithm solving vector borders problem.

```
1: v[0] ← (0)
2: Φ[0] ← 0
3: for n = 1 to N do
4:     Φ[n] ← −∞
5:     for s = m to M do
6:         if n − s ≥ 0 then
7:             P = Φ[n − s]
8:             for k = 1 to K do
9:                 if c^k_{n−s} = c^k_{n−s+1} = c^k_{n−1} then
10:                    P ← P + w_k ∗ len
11:                end if
12:            end for
13:            if P > Φ[n] then
14:                Φ[n] ← P
15:                v[n] ← v[n − s]
16:                append n to v[n]
17:            end if
18:        end if
19:    end for
20: end for
21: return v[N]
```

The algorithm uses arrays $\Phi[0 \ldots N]$ and $v[0 \ldots N]$ to memorize previously computed solutions. We start by filling in values for problem of size 0 (lines 1—2). Then we successively solve problems of sizes 1 to N. Fist, we set value of $\Phi[n]$ to $-\infty$ (line 4), which corresponds to lack of valid partitioning. Then, for each vector size between m and M, we try to extend a previously found solution for smaller problem size with a single vector. We compute value

of Φ function for such candidate solution in variable P (lines 7—12). Since Φ is defined as a sum over all vectors, we only have to examine the newly added vector in each column. This can be done in constant time. Next, if the candidate solution is better than any tested before, we save it as current best (lines 13—17). Finally, after all the loops are completed, we return the partitioning for problem size N.

Drawbacks Even though the dynamic programming approach reduces the complexity of this algorithm, it still has many important shortcomings which make it unusable in practice. Below we discuss some of them.

1. Size of the first vector is not known until the last iteration of the outer loop is completed. As a consequence, scan operator cannot interleave reading data from disk, determining vector sizes and producing vectors. It has to read all the data in advance and compute vector sizes upfront, before any actual query execution can happen.

   This requirement is clearly unacceptable. Isolating data transfer from query processing would have a significant negative impact on database performance. Even more importantly, in many cases memory buffers would be not sufficient to store all the data, so it would have to be evicted and read again.

2. Assume that N is much bigger than M, that is, number of values to read is much bigger than maximum vector size. We will try to answer following question: how many iterations of inner loop (lines 8—12) are required for each value in vectors produced by scan operator, that is, what is the cost of algorithm amortized over all the values?

   First, we start by computing the total number of iterations of inner loop.

$$
\begin{aligned}
I =& \sum_{n=1}^{N} \sum_{s=m}^{M} [n - s \geq 0]K \\
=& \sum_{n=1}^{N} \sum_{s=m}^{M} (1 - [n - s < 0])K \\
=& \sum_{n=1}^{N} \sum_{s=m}^{M} K - \sum_{n=1}^{N} \sum_{s=m}^{M} [n - s < 0]K \\
=& N(M - m + 1)K - K \sum_{s=m}^{M} \sum_{n=1}^{N} [n < s] \\
=& N(M - m + 1)K - K \sum_{s=m}^{M} s - 1 \\
=& N(M - m + 1)K - K \frac{(M-1)M - (m-2)(m-1)}{2}
\end{aligned}
$$

   Now, we divide it by the total number of produced values.

$$
\begin{aligned}
\frac{I}{KN} =& \frac{N(M - m + 1)K - K\frac{(M-1)M-(m-2)(m-1)}{2}}{NK} \\
=& M - m + 1 - \frac{(M-1)M - (m-2)(m-1)}{2N}
\end{aligned}
$$

With $N \gg M$ the last part of this formula is negligible. Hence, our result is:

$$\frac{I}{KN} \approx M - m + 1$$

Again, this result unacceptable. We can expect $M - m + 1$ to be in the ballpark of thousands and we cannot spare so many iterations on each produced value.

Although the algorithm described above cannot be applied in practice, we can find methods that provide good enough results, have reasonable cost and can be interleaved with reading the data and processing the query. They will not provide optimal solutions, but still can significantly improve the performance of current constant vectors optimization. There are many possible heuristics that could be used for that purpose, but finding effective ones requires real-life data testing.

3.5. Summary

This chapter discussed a simplification of RLE-compressed execution. It introduced constant vectors and the mechanism of primitive swapping. It also presented the results of conducted experiments. Finally, we explained two kinds of encountered overheads and the impact of vector size on query execution times.

The idea we proposed meets our design goals: it is simple, unintrusive and does not require major implementation effort, but at the same time it provides significant performance gains. Furthermore, additional improvements can be achieved by removing described overheads and creating mechanism for choosing optimal vector size.

Chapter 4

Dictionary Encoding

Dictionary encoding, described in Section 2.2.1, allows representing values by fixed-width integer codes. As we explained in Section 2.4, it is considered particularly useful for compressed execution. In this chapter we will discuss different properties of dictionaries and explain how they determine the opportunities of operating directly on codes. We will show challenges of maintaining global dictionaries and propose a method to avoid them. We will also perform a number of experiments to estimate the possible benefits of dictionary-compressed execution.

4.1. Operations on dictionary-encoded data

A dictionary can be characterized by a number of properties that determine its usefulness for compressed execution. These properties include:

Scope a scope defines part of data for which a given encoding is used. We will consider three possible scopes: domain-wide, column-wide and partition-wide (we define a partition as a subset of consecutive values of a given column).

Unique entries a dictionary has unique entries if any two distinct codes represent two distinct values.

Order preservation We say that a dictionary is order-preserving when for every pair of entries $(code_1, value_1)$, $(code_2, value_2)$ we have

$$code_1 < code_2 \Rightarrow value_1 < value_2$$

We will now discuss what operations can be performed directly on codes, and which of above properties are required to make it possible. Note that some of those requirements could be made less restrictive at the cost of extra effort.

1. **Equality checks** In order to check equality of encoded values and a given constant, we start by looking up corresponding entry for that constant in the dictionary. If there is a match, we use the code we found for comparisons. Otherwise, we may either use a special code or simply always return false. In case of checking equality of values from two sequences, both of them should be compressed using the same encoding scheme.

   Properties. *Scope:* column-wide for equality check with a constant, domain-wide for two sequences. *Unique entries:* yes, or else multiple comparisons would be required. *Order preservation:* no.

It is also possible to perform equality check of two sequences, each encoded using its own column-wide dictionary. Codes matching identical keys in different dictionaries have to be found upfront. These precomputed results can be used to determine if a given pair of codes corresponds to a pair of equal values or not.

2. **Range comparisons** The range comparisons — between an encoded sequence and a given value, as well as between two encoded sequences — are performed in exactly the same manner as equality checks.

   Properties. *Scope:* column-wide for comparisons with a constant, domain-wide for comparisons of two sequences. *Unique entries:* it is not required for comparisons with a constant value, but otherwise — yes. *Order preservation:* yes.

   Comparison between two sequences encoded using different column-wide schemes can be executed using precomputed code mapping (as in case of equality checks).

3. **Aggregations** We can use encoded grouping attributes, by treating them in exactly the same way as integer grouping attributes.

   Properties. *Scope:* at least column-wide. *Unique entries:* yes. *Order preservation:* no.

   The values for computing aggregates other than `COUNT` usually have to be decoded. The exception is finding `MIN` or `MAX` while using order-preserving dictionary.

4. **Duplicate elimination** The duplicate elimination can be performed directly on codes, in the same way it would be performed on integers.

   Properties. *Scope:* at least column-wide. *Unique entries:* yes. *Order preservation:* no.

5. **Joins** If we use an encoded join attribute, we treat it in the same way, as if it was an ordinary integer join attribute.

   Properties. *Scope:* domain-wide, since the encoding on both sides of the join has to be identical. *Unique entries:* yes. *Order preservation:* no.

   Assume that we use different column-wide encodings for corresponding join attributes. We can first perform a mini-join on their dictionaries, so that we can map codes from one encoding to codes from the other one. The fact that some of the entries may have no counterpart in the other dictionary is not a problem here.

   The attributes that only have to be copied during join operation can remain encoded, unless they are using partition-wide dictionaries and it is possible that the codes from different partitions will get mixed in join result.

6. **Results caching** When using dictionary encoding we can save the results of evaluating certain functions, for example, extracting substring or computing the `LIKE` predicate. They can be stored in an array indexed by dictionary codes, and later easily fetched. This way we avoid evaluating these functions multiple times for a single dictionary entry.

   Properties. *Scope:* at least partition-wide. *Unique entries:* not required. *Order preservation:* not required.

   In results caching we do not perform any operations on codes, but instead use dictionary encoding to save CPU time by avoiding repeated execution of expensive predicates with identical arguments.

This list suggest that in order to perform dictionary-compressed execution, it is crucial to provide at least column-wide encoding with unique entries. Having a domain-wide or order-preserving dictionary is useful as well, and allows us to perform some extra operations, but it is not as essential.

4.2. Global dictionaries

Normally in order to provide a uniform column-wide or domain-wide encoding we have to use a global dictionary. Such a dictionary is a persistent data structure, but we can assume that it is available in the main memory at all times. At every moment, it keeps a uniform encoding for all the values in a given column or group of columns, and allows different queries to lookup and insert entries.

Maintaining a global dictionary makes it possible to perform operations on codes, as we described in the previous section. However, it also poses a number of challenges, that we will now discuss.

Concurrency problems A global dictionary can be used by many transactions in parallel. This creates concurrency problems, as the actions of different transactions cannot be managed separately. Overcoming those problems may require meticulous tracking of which entries are used by which transactions.

As an example, assume that two concurrent transactions try to insert identical values into a dictionary. We must ensure that both of them receive identical codes, or otherwise the dictionary would no longer have unique entries. Moreover, if one of the transactions is aborted, the new entry cannot automatically be removed from a dictionary, because it is still being used by the other transaction. However, if the other transaction is aborted as well, we do not want to leave unused dictionary entries behind, so we should know that they are now available for deletion.

Expensive modifications Some of the operations on a global dictionary require adjusting codes across the entire column or domain. For example, when the dictionary grows too large more bits are required to represent an encoded value.

Moreover, a global dictionary may need a reorganisation every once in a while, when the obsolete entries — ones that no longer match any value — are removed, and later the entries are condensed. Such process would also require updating the whole encoded domain, and could create an interruption in database system work.

Compression efficiency Using column-wide or domain-wide encoding for compression purposes may result in worse compression ratios than ones that can be achieved by compressing partitions of a column separately. If values of a given column have a good locality, that is, number of distinct values in a partition is much smaller than in entire column, they could be encoded using fewer bits, which would result in better compression ratio. The problem is even more visible if a global dictionary contains 'holes' or has obsolete entries.

4.3. The potential of dictionary compression

We decided to estimate the possible performance benefits of using global dictionaries, to check whether they justify the effort of implementation. We used a method that does not

Stores		
StoreId	City	...
1001	*'Atlanta'*	...
1002	*'Boston'*	...
1003	*'Atlanta'*	...
1004	*'Chicago'*	...
1005	*'Denver'*	...
1006	*'Boston'*	...
1007	*'Denver'*	...
1008	*'Atlanta'*	...

(a) Original table with string column City.

Stores		
StoreId	City	...
1001	1	...
1002	2	...
1003	1	...
1004	3	...
1005	4	...
1006	2	...
1007	4	...
1008	1	...

CityDict	
Id	City
1	*'Atlanta'*
2	*'Boston'*
3	*'Chicago'*
4	*'Denver'*

(b) Modified table with column City replaced by integer column and dictionary table mapping codes to original strings.

Figure 4.1: Example of schema modification for simulating dictionaries with tables.

require any code modifications, and can be carried out by using only the SQL interface of the database system. We simulated global dictionaries with ordinary tables.

First, a dictionary table has to be created for a given column, by executing a query like

```
CREATE TABLE CityDict AS
             SELECT __ids__ AS Id,
                    DISTINCT(City) AS City
             FROM   Stores
```

where `__ids__` provides sequence $1, 2, 3, \ldots$, and can be obtained in different ways depending on the database system used. Next, the original column is replaced by integer column that references the dictionary table (see Figure 4.1). The original values can be retrieved by performing join with dictionary table. Finally, the queries have to be altered to match the new data organisation. When possible, the operations should be performed directly on codes.

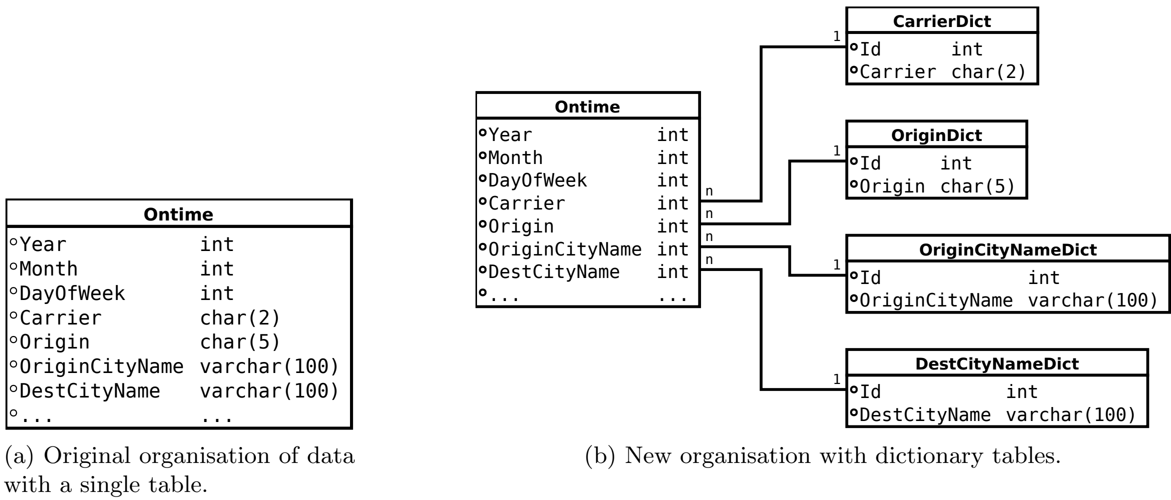
This approach gives us an approximation of performance improvements that could be obtained by implementing global dictionaries. Note that it provides a pessimistic estimate: adding extra tables significantly complicates query plans and increases the probability of choosing suboptimal ones. Moreover, retrieving original values by performing join operation is usually less efficient than fetching them from a dictionary.

We tested dictionary tables with the On-Time benchmark and chosen queries from TPC-H benchmark. We examined query execution times and observed the amount of time spent on scan operators (On-Time) and changes in memory usage (TPC-H). The results are presented in the following sections. Note that in each case the plans for modified queries were similar to the plans for original queries.

4.3.1. On-Time benchmark

We created dictionary tables for all string attributes accessed by On-Time queries (see Sections 3.3.1 and 3.3.2), that is, `Carrier`, `Origin`, `OriginCityName` and `DestCityName`. The new database organisation in shown on Figure 4.2. There are five queries in On-Time benchmark that use string attributes, that is, queries 3, 4, 5, 6, and 8. We did not modify the remaining queries in any way.

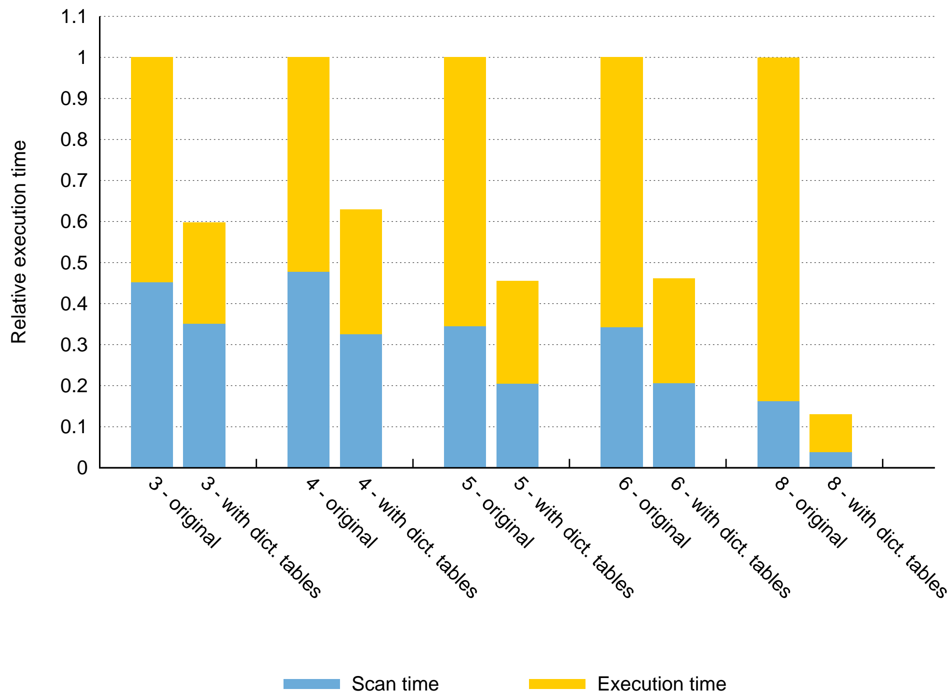
In a row store changing data types of some attributes can influence the performance of



(a) Original organisation of data with a single table.

(b) New organisation with dictionary tables.

Figure 4.2: The original and modified schema for On-Time data.



Query	Q3	Q4	Q5	Q6	Q8
Execution time change	−40.3%	−37.1%	−54.5%	−53.9%	−87.1%
Scan time change	−22.3%	−31.7%	−40.5%	−39.5%	−76.9%

Figure 4.3: Execution times of chosen On-Time queries with and without dictionary tables.

any query, because it alters the size and organisation of tuples. However, in a column store modifying attributes that are not used by a given query should not change its performance at all. The execution times of queries 0, 1, 2, 7 and 9 are identical with and without dictionary tables, which confirms this statement.

The results of conducted experiments are shown in Figure 4.3. We will now present the queries of the On-Time benchmark that were modified to use dictionary tables and discuss the impact of those changes on query performance. For convenience, we indicated columns containing codes with italics.

Query 3

We altered query 3 to use codes for `Origin` as a grouping attribute, and later retrieve the original values through join with `OriginDict`.

```
WITH t AS (
          SELECT   Origin,
                   COUNT(*) AS c
          FROM     Ontime
          WHERE    DepDelay > 10
                   AND Year BETWEEN 2000 AND 2008
          GROUP BY Origin
)
SELECT FIRST 10
         OriginDict.Origin,
         t.c
FROM     t LEFT OUTER JOIN OriginDict ON
                t.Origin = OriginDict.Id
ORDER BY c DESC
```

The total time was reduced by slightly over 40%. The majority of this improvement comes from a much more efficient aggregation. The direct aggregation on codes is now used instead of hash aggregation on strings.[1] This operator is now 7.5 times faster, while originally it contributed almost 35% of execution time. Also, the scan operator was improved by one fifth because reading an integer column with codes is faster than reading a corresponding string column.

Query 4

Query 4 was changed to use codes for `Carrier` as a grouping attribute and afterwards perform join with dictionary table.

```
WITH t AS (
         SELECT   Carrier,
                  COUNT(*) AS col2
         FROM     Ontime
         WHERE    DepDelay > 10 AND Year = 2007
         GROUP BY Carrier
)
SELECT   CarrierDict.Carrier,
         t.col2
FROM     t LEFT OUTER JOIN CarrierDict ON
                t.Carrier = CarrierDict.Id
ORDER BY col2 DESC
```

The execution time while using dictionary tables is 37% shorter than original. Again, this can be attributed mostly to faster aggregation. The time previously spent on this operator was 26% of total time, and now it is reduced five-fold. The scan is 31% faster.

[1] We explained the difference between direct and hash aggregation in Section 3.2.3.

Query 5

In query 5 we have two aggregations grouped by the `Carrier` attribute, which are now performed using codes. Moreover, the join of two subqueries also uses codes. Finally, the original values of the `Carrier` attribute are retrieved.

```
WITH t1 AS (
         SELECT    Carrier,
                   COUNT(*) AS c
         FROM      Ontime
         WHERE     DepDelay > 10 AND Year = 2007
         GROUP BY  Carrier
), t2 AS (
         SELECT    Carrier,
                   COUNT(*) AS c2
         FROM      Ontime
         WHERE     Year = 2007
         GROUP BY  Carrier
), t3 AS (
         SELECT    t1.Carrier,
                   c,
                   c2,
                   c*1000/c2 AS c3
         FROM      t1 JOIN t2 ON t1.Carrier = t2.Carrier
)
SELECT   CarrierDict.Carrier,
         t3.c,
         t3.c2,
         t3.c3
FROM     CarrierDict RIGHT OUTER JOIN t3 ON
                   CarrierDict.Id = t3.Carrier
ORDER BY c3 DESC
```

Query 5 with dictionary tables takes 55% less time than before. The two aggregations together contributed almost half of original execution time. When replaced by direct aggregations on codes, they are $5 - 6$ times faster than before. The join operation performed on codes could not bring notable improvement, because the sizes of input relations are too small. The scan time was reduced by 40%.

Query 6

Query 6 is very similar to query 5. Again two aggregation grouped by `Carrier` attribute and join of subqueries' results were altered to work directly on codes.

```
WITH t1 AS (
         SELECT    Carrier,
                   COUNT(*) AS c
         FROM      Ontime
         WHERE     DepDelay > 10
                   AND Year BETWEEN 2000 AND 2008
         GROUP BY  Carrier
), t2 AS (
         SELECT    Carrier,
                   COUNT(*) AS c2
         FROM      Ontime
         WHERE     Year BETWEEN 2000 AND 2008
         GROUP BY  Carrier
), t3 AS (
         SELECT    t1.Carrier,
```

```
                c,
                c2,
                c*1000/c2 AS c3
        FROM    t1 JOIN t2 ON t1.Carrier = t2.Carrier
)
SELECT  CarrierDict.Carrier,
        t3.c,
        t3.c2,
        t3.c3
FROM    CarrierDict RIGHT OUTER JOIN t3 ON
                CarrierDict.Id = t3.Carrier
ORDER BY c3 DESC
```

The execution time of query 6 is now 54% shorter than before. The two aggregations used to take almost half of total time, but after turning them into direct aggregations on codes they can be executed 5.5 times faster. The results of subqueries are too small to allow any substantial gains in join operator. The scan is almost 40% faster.

Query 8

Query 8 was altered to perform duplicate elimination for `OriginCityName` attribute directly on codes. The aggregation grouped by `DestCityName` also used integers instead of strings. Note that an additional join operation was inserted to retrieve original values of `DestCityName` attribute, but we never decode values of `OriginCityName`, because they aren't really needed to execute this query.

```
WITH t AS (
        SELECT   DestCityName,
                 COUNT(DISTINCT OriginCityName) AS col2
        FROM     Ontime
        WHERE    Year BETWEEN 1990 AND 2000
        GROUP BY DestCityName
)
SELECT  DestCityNameDict.DestCityName,
        t.col2
FROM    t LEFT OUTER JOIN DestCityNameDict ON
               DestCityNameDict.Id = t.DestCityName
ORDER BY col2 DESC
```

The time required to execute query 8 was reduced almost eight-fold using dictionary tables. The aggregation in this query is performed by VectorWise using both `DestCityName` and `OriginCityName` as grouping attributes. Originally over 80% of time was spend on this operator. Again, after converting it into direct aggregation on integer values, we observe that it is 14 times faster than before. Also, the scan time was reduced by over 76%.

Conclusion

These results show that using dictionary tables significantly improves performance in On-Time benchmark (see Figure 4.3). All the queries that use string columns are now over one third faster than before. The most spectacular gains can be observed for query 8, which can be executed almost eight times faster, and it is the slowest query in the whole On-Time benchmark. We can also notice a great improvement for aggregations that use strings as grouping attributes.

4.3.2. TPC-H benchmark

TPC-H is an industry standard benchmark for decision support systems. Its specification can be found in [TPC11] and we will not describe it here. We chose scale factor 10, which means that table lineitem contained ca. 60.000.000 rows.

Some of the string attributes accessed by TPC-H queries are unsuitable for using with dictionaries, because they have way too many distinct values. This applies, for example, to column s_comment appearing in query 16. We decided to create dictionary tables for attributes o_orderpriority, n_name, l_shipmode, p_type and p_brand. The schema modifications were analogical to those presented in the previous sections.

We omitted queries that use only not dictionary-compressible string attributes, or no string attributes at all, and ones that spend only a marginal fraction of execution time on operations using strings. In the end, we conducted experiments with queries 4, 9, 12, 14, 16 and 19.

Query 4

Query 4 was modified to use codes for o_orderpriority as a group-by column and prevent values of this attribute from taking part in join with lineitem table.

```
SELECT   o_orderpriority_dict.o_orderpriority,
         order_count
FROM     (
         SELECT     o_orderpriority,
                    COUNT(*) AS order_count
         FROM       orders
         WHERE      o_orderdate >= date '1993-07-01'
                    AND o_orderdate <
                          date '1993-07-01' + interval '3' month
                    AND EXISTS (
                          SELECT *
                          FROM   lineitem
                          WHERE  l_orderkey = o_orderkey
                          AND l_commitdate < l_receiptdate
                          )
          GROUP BY  o_orderpriority
          ) AS q,
          o_orderpriority_dict
WHERE     q.o_orderpriority = o_orderpriority_dict.id
ORDER BY  o_orderpriority_dict.o_orderpriority;
```

The execution time of query 4 was reduced by almost one third. A hash aggregation that contributed 20% of total time was removed, and a direct aggregation that is 20 times faster and takes 6 times less memory is now used instead. A merge join of orders and lineitem was also improved. It is now 58% faster and uses only one tenth of memory that it required before. The total memory usage of query 4 was reduced by 72%.

Query 9

In query 9 we use codes for n_name as a grouping attribute.

```
SELECT   n_name_dict.n_name AS nation,
         o_year,
         sum_profit
FROM     (
         SELECT     n_name,
                    o_year,
```

```
                SUM(amount) AS sum_profit
        FROM    (
                SELECT n_name,
                       extract(year FROM o_orderdate) AS o_year,
                       l_extendedprice * (1 - l_discount) -
                            ps_supplycost * l_quantity AS amount
                FROM   part,
                       supplier,
                       lineitem,
                       partsupp,
                       orders,
                       nation
                WHERE  s_suppkey = l_suppkey
                       AND ps_suppkey = l_suppkey
                       AND ps_partkey = l_partkey
                       AND p_partkey = l_partkey
                       AND o_orderkey = l_orderkey
                       AND s_nationkey = n_nationkey
                       AND p_name LIKE '%green%'
                ) AS profit
         GROUP BY n_name,
                  o_year
         ) AS q,
         n_name_dict
WHERE    q.n_name = n_name_dict.id
ORDER BY nation,
         o_year DESC;
```

Query 9 is now 7% faster. Originally a hash aggregation took 8% of total time. It was replaced by a much more efficient direct aggregation. Also, removing string attribute from join with `lineitem` table reduced the memory usage by 9%.

Query 12

The expressions using `o_orderpriority` and `l_shipmode` are evaluated on dictionary tables (subqueries `high`, `low` and `shipmodes`). The codes for `l_shipmode` are also used as a grouping attribute.

```
SELECT   l_shipmode_dict.l_shipmode AS l_shipmode,
         high_line_count,
         low_line_count
FROM     (
         SELECT   shipmodes.id,
                  SUM(line_count.high) AS high_line_count,
                  SUM(line_count.low) AS low_line_count
         FROM     (
                  SELECT id,
                         (
                         CASE
                         WHEN o_orderpriority = '1-URGENT'
                              OR o_orderpriority = '2-HIGH'
                         THEN 1
                         ELSE 0
                         END
                         ) AS high,
                         (
                         CASE
                         WHEN o_orderpriority <> '1-URGENT'
                              AND o_orderpriority <> '2-HIGH'
```

```
                        THEN 1
                        ELSE 0
                        END
                        ) AS low
                   FROM o_orderpriority_dict
                   ) AS line_count,
                   orders,
                   lineitem,
                   (
                   SELECT id
                   FROM   l_shipmode_dict
                   WHERE  l_shipmode = 'SHIP' OR l_shipmode = 'MAIL'
                   ) AS shipmodes
          WHERE    o_orderkey = l_orderkey
                   AND l_shipmode = shipmodes.id
                   AND l_commitdate < l_receiptdate
                   AND l_shipdate < l_commitdate
                   AND l_receiptdate >= date '1994-01-01'
                   AND l_receiptdate <
                          date '1994-01-01' + interval '1' year
                   AND o_orderpriority = line_count.id
          GROUP BY shipmodes.id
          ) AS q,
          l_shipmode_dict
WHERE     q.id = l_shipmode_dict.id
ORDER BY l_shipmode_dict.l_shipmode;
```

Query 12 now takes 26% less time to execute. The majority of this change can be attributed to more efficient evaluation of condition `l_shipmode IN ('MAIL', 'SHIP')` that previously took 37% of total time. The performance of aggregation is also improved, but the size of its input relation is too small to allow a significant impact on execution time.

Query 14

In query 14 we evaluate the relatively expensive expression `p_type LIKE 'PROMO%'` directly on the dictionary table. This way it is computed exactly once for each unique value of `p_type` and also this string column is prevented from participating in the join with the `lineitem` table.

```
WITH promo_types AS (
       SELECT id,
              (
              CASE
              WHEN p_type LIKE 'PROMO%'
              THEN 1
              ELSE 0
              END
              ) AS ispromo
       FROM   p_type_dict
), promo_parts AS (
       SELECT p_partkey,
              ispromo
       FROM   part_d,
              promo_types
       WHERE  p_type = id
)
SELECT 100.00 * SUM (
                    CASE
                    WHEN ispromo = 1
```

```
                        THEN l_extendedprice * (1 - l_discount)
                        ELSE 0
                        END
                        ) / SUM(l_extendedprice * (1 - l_discount))
                                                   AS promo_revenue
FROM    lineitem,
        promo_parts
WHERE   l_partkey = p_partkey
        AND l_shipdate >= date '1995-09-01'
        AND l_shipdate < date '1995-09-01' + interval '1' month
```

The execution time of query 14 is now 14% shorter. The cost of joins that dominates query performance (previously 75% of total time) is reduced by 24%. Also, the evaluation of `p_type LIKE 'PROMO%'` was almost 40% faster. Query used only half of the memory it required before.

Query 16

The dictionary tables for `p_brand` and `p_type` attributes were used. Selections on these columns were replaced by selections on corresponding dictionary tables (subqueries `types` and `brands`) followed by join. We also used codes as grouping attributes in aggregation.

```
SELECT    p_brand_dict.p_brand,
          p_type_dict.p_type,
          p_size,
          supplier_cnt
FROM      (
          SELECT    p_brand,
                    p_type,
                    p_size,
                    COUNT(DISTINCT ps_suppkey) AS supplier_cnt
          FROM      partsupp,
                    part,
                    (
                    SELECT id
                    FROM   p_type_dict
                    WHERE  p_type NOT LIKE 'MEDIUM POLISHED%'
                    ) AS types,
                    (
                    SELECT id
                    FROM   p_brand_dict
                    WHERE  p_brand <> 'Brand#45'
                    ) AS brands
          WHERE     p_partkey = ps_partkey
                    AND p_brand = brands.id
                    AND p_type = types.id
                    AND p_size IN (49, 14, 23, 45, 19, 3, 36, 9)
                    AND ps_suppkey NOT IN (
                          SELECT s_suppkey
                          FROM   supplier
                          WHERE  s_comment LIKE '%Customer%Complaints%'
                          )
          GROUP BY p_brand,
                    p_type,
                    p_size
          ) AS q,
          p_brand_dict,
          p_type_dict
WHERE     p_brand_dict.id = q.p_brand
```

Query	Q4	Q9	Q12	Q14	Q16	Q19
Execution time change	−31.8%	−7.8%	−26.7%	−14.3%	−48.2%	−30.5%
Query memory:						
– original	8.5MB	38.1MB	6.5MB	108MB	111MB	24.2MB
– with dictionary tables	2.4MB	34.6MB	7.5MB	54MB	70MB	26.5MB
– change	−71.8%	−9.2%	+15.4%	−50.0%	−36.9%	+9.5%

Table 4.1: Execution times and memory usage of chosen queries form TPC-H benchmark while using dictionary tables.

```
        AND p_type_dict.id = q.p_type
ORDER BY supplier_cnt DESC,
        p_brand_dict.p_brand,
        p_type_dict.p_type,
        p_size;
```

Query 16 is almost two times faster with dictionary tables. An aggregation originally contributed 73% of total time. Now, a hash aggregation is still used to execute this query, but it is 68% faster than before. Joining the results of selections performed on dictionaries creates an overhead of ca. 3% of total time. The memory usage of query 16 was reduced by one third.

Query 19

We used dictionary tables for attributes `l_shipinstruct`, `l_shipmode`, `p_brand` and `p_container`. Instead of selections on these columns, we used selections on dictionary tables followed by join operation.

Unfortunately, query optimizer was fooled by a large number of additional tables and created suboptimal query plan. For that reason, we had to provide hand-written query plan using internal representation called VectorWise algebra (see X100 algebra in [Zuk09]).

Modified query 19 takes over 30% less execution time than original one. This benefit comes from reducing the number of string comparisons. For example, the selection on `l_shipinstruct` originally contributed almost 40% of total time. In the new query corresponding selection on dictionary and following join are one third faster. Similar improvements can be observed for selections on `l_shipmode`, `p_brand` and `p_container`.

Due to large number of join operations, the modified query uses about 10% more memory.

Conclusion

The summary of experiments with TPC-H queries can be found in table 4.1. As in On-Time benchmark, we see that using dictionary tables allows us to reduce execution times of many queries, and often it is a significant change. It also turns out, that frequently modified queries require much less memory than the original ones, and join operations become cheaper, because there is less data to copy.

4.4. On-the-fly dictionaries

As discussed in Section 4.2, global dictionaries, while providing good properties, cause problems and require significant changes to the system. This goes against our desire to make changes as unintrusive as possible. As a result, in this section we propose a solution that

avoids the difficulties associated with global dictionaries, but still allows us to benefit from using a uniform column-wide or dictionary-wide encoding. The idea is to maintain separate dictionaries for partitions of a column, and combine them during the scan operation. We call the result of this procedure *on-the-fly dictionary*, because it is built incrementally, with new entries added each time a data from previously unvisited partition is requested. Such a dictionary would serve a single query, and would be dropped as soon as the query execution is completed.

Partial dictionaries

A *partial dictionary* is a list of unique values, that represents some dictionary encoding and is stored on disk along with a set of encoded values. It is fetched with compressed data when requested by the scan operator, stays for some time in disk buffer, and later it is evicted. The content of partial dictionary can be altered only when some updates are flushed to disk. Thus, a partial dictionary is very different from a global dictionary, that is a memory-resident data structure, that can be accessed and modified by many transaction in parallel. For this reason, it is much simpler to create and to manage.

In VectorWise each block is compressed separately from the others, which creates a natural partitioning of a column. Hence, every block of a dictionary-compressed column can be supplied with its own *in-block dictionary*.

Combining dictionaries

Partial dictionaries would be incorporated into on-the-fly dictionary one-by-one. First, each entry from a partial dictionary would have to be looked up in on-the-fly dictionary, and if no match was found, it would have to be inserted. a map of old and new codes would be created in the process. Second, each compressed value would be translated using this code map, so that the encoding scheme fits the on-the-fly dictionary.

This process will create an overhead related to performing dictionary lookups and inserts. The scale of this overhead depends on the size of partial dictionaries, and not on the number of encoded values. In other words, the less unique values are in each partition, the smaller the overhead will be. Typically, a dictionary-compressible column has much more values than *unique* values, so the overhead would be amortized.

We will now discuss an example of how the on-the-fly dictionary is created from a number of in-block dictionaries. This example is illustrated on Figure 4.4. On the left side we can see the content of compressed block that is currently being read. Each of those blocks has its own independent dictionary and stores a number of encoded values. The entries in on-the-fly dictionary are listed in the center. The mapping from in-block codes to column-wide codes is shown on the right.

(a) At the beginning of query execution on-the-fly dictionary is empty.

(b) The first block contains two unique values: *'Atlanta'* and *'Boston'*. Obviously, neither of them is present in the on-the-fly dictionary, so both have to be inserted. Since the codes remained unchanged, there is no extra translation required. The encoded values can be now passed to the scan operator.

(c) In the second block we have three unique values, and two of them can already be found in on-the-fly dictionary. We only have to insert one additional value — *'Chicago'*. Now the

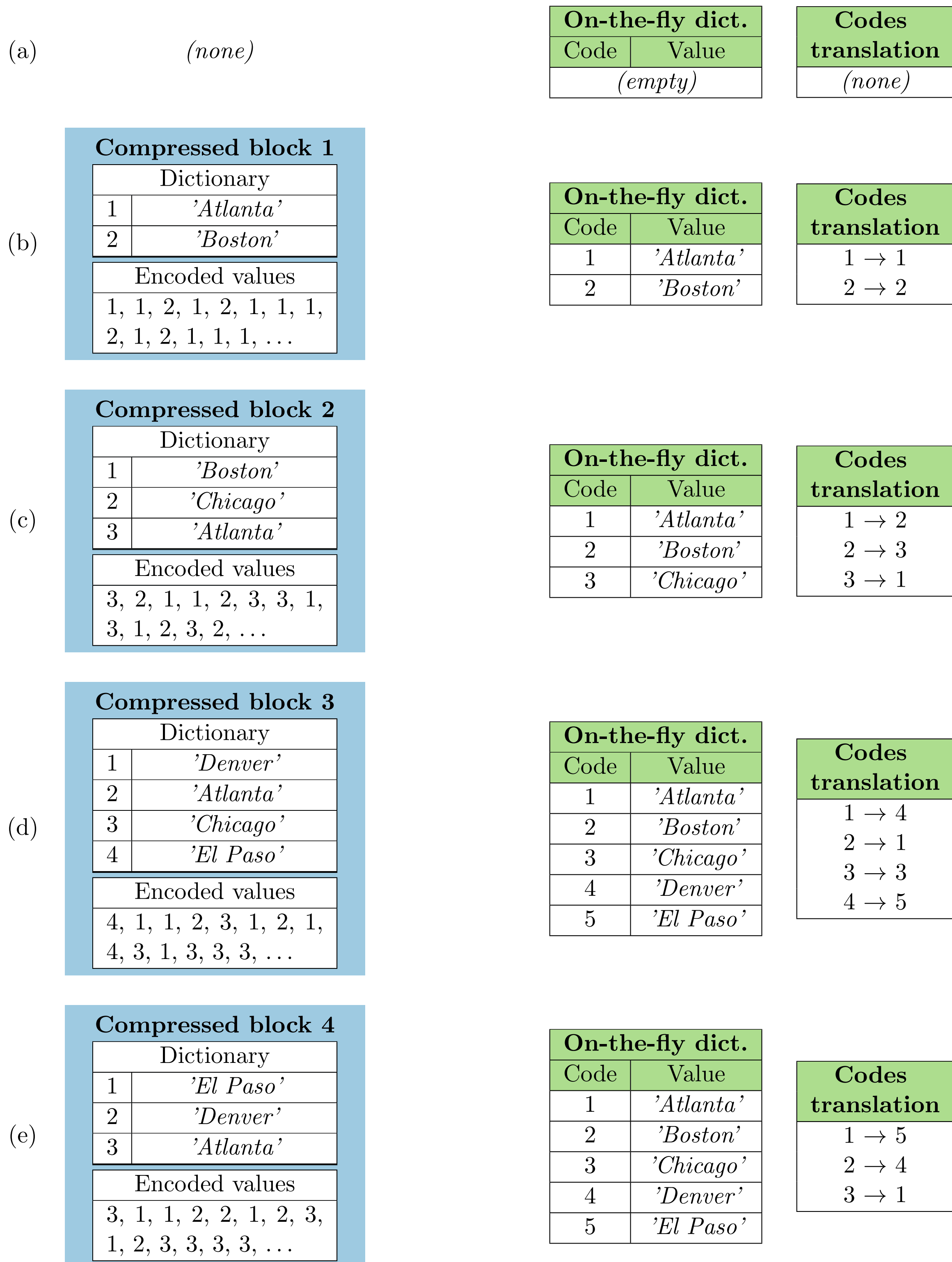


Figure 4.4: Example of combining partial in-block dictionaries in order to create on-the-fly dictionary.

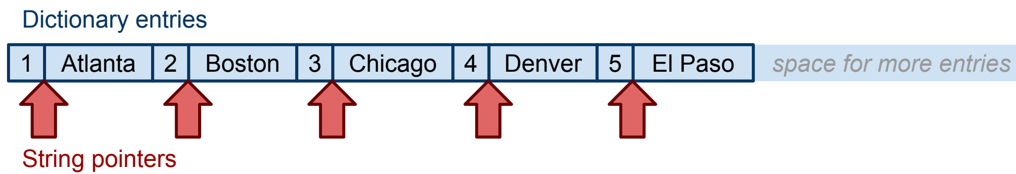


Figure 4.5: Example of on-the-fly dictionary content.

entries in in-block and on-the-fly dictionaries are identical, but the encoding is different. For example, *'Boston'* has code 1 in in-block dictionary, but it corresponds to 2 in on-the-fly dictionary. Hence, all the 1s have to be replaced by 2s. The same process has to be applied to other codes.

(d) The third block contains four unique values. Half of them is already present in on-the-fly dictionary, but we still have to insert entries *'Denver'* and *'El Paso'*. As before, the codes have to be translated to match the new encoding.

(e) All the values from fourth block are already present in on-the-fly dictionary, so we do not insert any new entries at this time, only provide an appropriate code mapping.

Implementation

We propose to create empty on-the-fly dictionaries during the construction of the operator tree. The dictionary handles would be attached to vectors to denote that their values are contained in a respective dictionary. Then they would be propagated through expression and operator trees so that, for example, if the vectors serving as input of join operator have attached dictionary handles, then the respective vectors holding join results will have attached the same handles. Other preparations would include, for example, making sure that matching join attributes use the same dictionaries (that is, ensuring that domain-wide encoding is being used), or inserting constant values into respective dictionaries (for example, to make equality checks with constant possible). Finally, aggregation and join operators may choose to use codes as grouping or join attributes, which can alter their behaviour and used data structures. The dictionary should allow efficient lookups of string values, so we propose to implement it using a hash table.

Data representation

We propose that vectors contain pointers to strings stored in a dictionary rather than actual codes. The reasons are as follows. First, if a dictionary has unique entries, then the string pointers hold the same property as codes: two strings from a dictionary are equal if and only if their pointers are equal. The scope also remains the same. Second, this way decoding is never required, because such vectors can always be used just like ordinary string vectors. Finally, we can store the codes together with string values in the dictionary, and fetch them easily whenever they are needed, for example, in order to use them for direct aggregation. An example of such organisation is presented on Figure 4.5. In the upper part we can see the content of a dictionary: codes and values stored one-by-one. The arrows represent string pointers which would be placed in vectors. We can find a matching code for pointer `x` by computing `*((int*)x - 1)`.

Dictionary overflows

If the on-the-fly dictionary grows too large, such that it is impossible to add new entries, we should nevertheless continue query execution. The appropriate dictionary handle must be marked as invalid. The equality checks and range comparisons have to be performed in traditional way, and the result caching has to be abandoned. The joins and aggregations are more difficult to handle if they use codes as grouping or join attributes. Then it may be necessary to rebuild a hash table in order to make them usable with non-dictionary values.

Updates

In VectorWise, when an attribute value has to be updated, inserted or deleted, the changes are not applied immediately to data stored on disks. Instead, a representation of such modification is recorded in a dedicated main memory data structure called *positional delta tree* [HNZB08]. The scan operator is responsible for applying those memory-resident changes to data it receives from buffer manager. When the number of modifications grows too large, they have to be flushed to disks for performance reasons.

The process of applying memory-resident changes has to be adjusted in order to use on-the-fly dictionaries. We have to ensure that a new value — that appears as a result of insertion or update — is represented by an appropriate dictionary entry. For that purpose we have to perform lookup-or-insert operation on on-the-fly dictionary for each such value. This process introduces some additional overhead, but as we stated before, the number of memory-resident modifications has to be kept low, so this overhead will also be limited. However, if the overhead grows too large and applying updates becomes a bottleneck, we may decide to stop using on-the-fly dictionary (as in the case of dictionary overflow).

Domain-wide encoding

As we explained at the beginning of this chapter, for some operations — for example, joins or equality checks — domain-wide encoding is much more useful than column-wide encoding. While using global dictionaries we have to know upfront which attributes have a common domain, so that we can use a single dictionary for all of them and provide uniform encoding. However, with on-the-fly dictionaries such prior knowledge in not necessary, but instead we can adapt to the needs of query that is currently processed. We can easily detect situations when two or more columns should follow the same encoding scheme and use a common dictionary handler for all of them.

Note that using on-the-fly dictionary by many operators concurrently is — unlike with global dictionaries — not a difficult task. The only available operation is lookup-and-insert, and there is no way to alter or delete an entry. Thus, all we need to ensure correctness is a single mutex.

4.5. Summary

This chapter presented dictionary-compressed execution. The possible operations on dictionary-encoded data were described and their impact on database performance was estimated using dictionary tables. We concluded that dictionary-compressed execution can allow processing the queries much faster while using less memory and disk bandwidth. Unfortunately, providing uniform column-wide or domain-wide encoding by maintaining a global dictionary is a challenging task. We described on-the-fly dictionaries that could provide similar gains while avoiding the difficulties of global dictionaries.

Chapter 5

Conclusions and future work

In our work we tried to improve the performance of VectorWise with compressed execution. We examined previously proposed solutions and found out that they require massive adaptation of database engine and poorly match the VectorWise architecture. It inspired us to search for simpler methods, that are less complicated and invasive in implementation, but still manage to seize significant performance benefits.

Contributions

We introduced *constant vectors* optimization which is a variant of RLE-compressed execution. We implemented it in VectorWise and tested its performance. We also investigated how it influences the impact of vector size on query execution times.

We examined the possible benefits of dictionary-compressed execution, and proposed *on-the-fly dictionaries*, that offer similar gains as global dictionaries, but are much easier to maintain.

Conclusions

Our work shows that compressed execution can greatly reduce query execution times. The full-blown solutions are cumbersome and invasive, but we managed to find alternatives less painful to integrate in a database system. These optimizations can provide visible performance improvements. The usefulness of the proposed methods was confirmed by benchmarking results.

Future work

Our work may be complemented and extended in future in the following ways:

(a) An attempt can be made to reduce or remove the overheads of current implementation of constant vectors. We presented a rough estimate of what impact it could have on VectorWise performance, but it requires verification with actual measurements.

(b) The mechanism for producing vectors with variable size can be implemented. We would like to find out whether it gives visibly better results than adjusting vector size query-wide.

(c) We can implement on-the-fly dictionaries along with operations on data stored in them. The experiments with dictionary tables showed significant improvements of query exe-

cution speed, but on-the-fly dictionaries would introduce their own overheads. We shall find out to what extent those overheads hinder the performance gains.

(d) Alternatively, we could implement global dictionaries. This would require more work, but would allow better analysis of on-the-fly dictionaries.

(e) Frame of reference (described in Section 2.2.1) is another compression method that could be used for non-intrusive compressed execution. It allows remapping large integers into smaller ones, so that they can be stored using less bytes. This can have beneficial influence on performance. Note that there are many opportunities for frame of reference compression, because it often happens that only a small range of values from large integer domain is used (e.g. dates). Implementing and testing compressed execution based on frame of reference encoding is a possible future extension of this work.

Bibliography

[Aba08] Daniel J. Abadi. *Query Execution in Column-Oriented Database Systems*. PhD thesis, MIT, 2008.

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance, 2001.

[AMF06] 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*, 2006.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really. In *In SIGMOD*, 2008.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 54–65, 1999.

[CGK01] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. Query optimization in compressed database systems. *SIGMOD Rec.*, 30, May 2001.

[Che02] Zhiyuan Chen. *Building compressed database systems*. PhD thesis, Cornell University, 2002.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing Relations and Indexes. In *ICDE'98*, pages 370–379, 1998.

[GS91] Goetz Graefe and Leonard D. Shapiro. Data Compression and Database Performance. In *In Proc. ACM/IEEE-CS Symp. On Applied Computing*, 1991.

[HNZB08] S. Héman, N. J. Nes, M. Zukowski, and P. A. Boncz. Positional Delta Trees To Reconcile Updates With Read-Optimized Data Storage. CWI Technical Report INS-E0801, CWI, August 2008.

[HRSD07] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[Huf52] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[Ing] Ingres VectorWise. http://www.ingres.com/products/vectorwise.

[JRSS08] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *Proc. VLDB Endow.*, 1, August 2008.

[Ker10] Martin Kersten. On-Time Report. http://homepages.cwi.nl/~mk/ontimeReport, December 2010.

[MF04] Roger MacNicol and Blaine French. Sybase IQ Multiplex - Designed For Analytics. In *Proceedings of the 30th VLDB Conference*, pages 1227–1230, 2004.

[Res11] Research and Innovative Technology Administration, Bureau of Transportation Statistics. On-Time Performance. http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time, February 2011.

[RH93] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Record*, pages 31–39, 1993.

[RHS95] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *Proc. of 7th Intl. Conf. on Management of Data*, 1995.

[Tka09] Vadim Tkachenko. Analyzing air traffic performance with InfoBright and MonetDB. http://www.mysqlperformanceblog.com/2009/10/02/analyzing-air-traffic-performance-with-infobright-and-monetdb, October 2009.

[TPC10] TPC Benchmark C Standard Specification. http://www.tpc.org/tpcc/, February 2010.

[TPC11] TPC Benchmark H (Decision Support) Standard Specification. http://www.tpc.org/tpch/, June 2011.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29, September 2000.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[ZHNB07] M. Zukowski, S. Heman, N. J. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing In A DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 723 – 734. A.C.M., 2007.

[ZNB08] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *In DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware*, pages 47–54. ACM, 2008.

[Zuk09] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, Universiteit van Amsterdam, 2009.