Master Thesis

Dictionary compression for a scan-based, main-memory database system

Author(s):
Bernet, Janick

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006050496

Rights / License:
In Copyright - Non-Commercial Use Permitted
Dictionary Compression for a Scan-Based, Main-Memory Database System

Master Thesis

Janick Bernet
ETH Zurich
jabernet@student.ethz.ch

October 1, 2009 - March 31, 2010

Supervised by:
Prof. Donald Kossmann
Philipp Unterbrunner
Abstract

The advent of the No-SQL movement and the emergence of innovative, new database technology over the last years reflect the increasing diversification of workloads and the shortcomings of traditional approaches in supporting them. Compression has been studied extensively on traditional systems, yet little has been done to determine if results from those studies can be reproduced in the context of these new workloads and database technologies. This thesis investigates the benefits of dictionary compression in a novel scan-based, in-memory row-store called Crescando. The findings it makes carry over to comparable systems.

It is shown how to integrate dictionary compression transparently into Crescando, without breaking with the strong predictability and scalability guarantees the original system provides. This is achieved by exploiting the advantages of a scan-based architecture in combination with attribute-level compression. For the actual dictionary implementation, a range of data structures for storing string-values are considered, including various forms of hash-tables and tries. Based on these data structures, different dictionary candidates are implemented and evaluated on both real-world and synthetic data sets and workloads. The results recommend the use of a novel data structure developed as part of the thesis: the extendible bidi-map. It performs well under a wide range of workloads, due to its small memory footprint and cache-conscious design. In conclusion, adding dictionary compression to Crescando enables significant space savings for many use cases, as well as performance improvements for certain workloads.
Acknowledgments

I would like to thank my advisor Donald Kossmann for giving me the opportunity to work on the Crescando project and my supervisor Philipp Unterbrunner for his support throughout the thesis. I would also like to thank Giorgos Giannikis, who is also working on the Crescando project, for helping me at many instances when I got stuck. Furthermore, I thank all people I shared the office with for providing such great working atmosphere. And last but not least I would like to give my special thanks to all people of the group that participated in often engaged, sometimes exhausting, but always fun table football matches.
# Contents

1. **Introduction** 6  
   1.1 Background & Motivation 6  
   1.2 Contribution 8  
   1.3 Outline & Approach 8  
   1.4 Design Principles 9  
   1.5 Terminology 9  

2. **Related Work** 11  
   2.1 State of the Art in Traditional Database Systems 11  
   2.2 Database Compression and Performance 12  
   2.3 Order Preserving Dictionary Compression 13  
   2.4 Data Structures for Storing String Values 13  

3. **Dictionary Integration** 15  
   3.1 The Crescando Storage System 15  
   3.2 Design Decisions 21  
   3.3 Architecture 26  
   3.4 Summary of Changes 34  

4. **Dictionary Design & Data Structures** 35  
   4.1 Dimensions 36  
   4.2 Influencing Factors 36  
   4.3 Design Decisions 41  
   4.4 Data structures 43  
   4.5 Dictionary Candidates 50  

5. **Evaluation** 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Questions</td>
<td>54</td>
</tr>
<tr>
<td>5.2 Approach</td>
<td>56</td>
</tr>
<tr>
<td>5.3 Results</td>
<td>61</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>79</td>
</tr>
<tr>
<td>6.1 Future Work</td>
<td>80</td>
</tr>
<tr>
<td>A Appendix</td>
<td>83</td>
</tr>
<tr>
<td>A.1 Amadeus Itinerary Schema</td>
<td>83</td>
</tr>
<tr>
<td>Bibliography</td>
<td>85</td>
</tr>
</tbody>
</table>
1

Introduction

1.1 Background & Motivation

Compression has been around for many years in disk-based database systems and studied extensively [33, 11, 29, 5, 21, 10]. It has only been recently however that the I/O savings provided could outweigh the increase in CPU-consumption and thus increase overall system performance [28, 12]. This is especially true for light-weight dictionary compression. Not surprisingly, this kind of compression has therefore seen a surge in popularity in recent years and has been propagated by leading database vendors and the research community alike [1, 3, 2]. Despite that interest, little has been investigated into whether improvements in performance also apply to in-memory row-stores.

In this thesis we evaluate dictionary compression in the context of Crescando, a scan-based, in-memory row-store. Crescando was developed by the Systems group at ETH Zurich as an answer to the increasing demand for database systems to provide "predictable performance under unpredictable workloads" [31]. This is a requirement that many decision support systems have to fulfill, as queries tend to get more complicated and diverse, but users still expect answers to be available in a matter of seconds.

Amadeus, a world-leading service provider for managing travel-related bookings, operates such system. Their implementation is based on a combination of a commercial relational database and a main-frame key-value store. The
system performs well for simple primary key lookups, but is ill-suited for decision support queries that include non-key columns. To be able to support such queries, the database contains many materialized views, each tailored at supporting a very specific kind of query. This approach does not scale well and only works when the workload is known beforehand. If a query which has not been optimized for is added to the system, the materialized views can not be used. In that case an expensive full table scan is needed.

Crescando is built around this observation: If with more and more diverse queries a full table scan becomes the most likely access path, why not optimize the system towards full table scans and make it the only method of data access? At the core of Crescando hence is an in-memory table that is scanned continuously. Instead of requiring the user to define indexes on the data, Crescando indexes the incoming operations on the fly. This yields to predictable latency, as operations are answered after a full scan cycle, which takes a fixed amount of time. A scan is done by many threads in parallel, each operating on a segment of the horizontally partitioned table. The scan threads do not communicate with each other providing a strict shared-nothing architecture that allows Crescando to scale up nearly linearly.

Adding compression to such system is not straightforward and the benefits are not clear beforehand. In this thesis we show how to implement dictionary compression in Crescando without affecting strong guarantees in predictability and scalability. For this purpose we will employ attribute-level compression that has been studied mainly in the context of column stores. To find a fitting dictionary implementation, existing data structures are evaluated on several dimensions. Based on that evaluation we motivate our own novel data struc-
ture, the bidi-map. It provides sub-linear key and value-lookup performance, is cache-conscious and has a low memory footprint. We compare it to other dictionary implementations based on existing data structures and evaluate the benefits of compression under various workloads.

1.2 Contribution

Dictionary compression is known to not only provide considerable space savings, but to also improve performance in disk-based systems due to reduced I/O. It has not been studied so far however for scan-based, in-memory row-stores. In this thesis we evaluate the gains of dictionary compression in such a system, by example of Crescando. Our main contributions are:

- We compare existing data structures to be used in a compression dictionary. We present our own novel data structure as a well-suited candidate and compare it to those other data structures (Chapter 4).

- We show how to adapt an existing system to support dictionary compression. We explain which factors and influences have to be considered in that process (Chapter 3). At this we especially focus on how to integrate compression in a way that does not interfere with Crescando’s predictability guarantees and shared-nothing architecture. We argue that attribute-level dictionary compression made popular by column stores does also fit a scan-based row-oriented systems like Crescando.

- We show that compression can pay off in regard to space and performance. We furthermore show that it enables workloads that were not feasible without it and argue that this strengthens Crescando’s predictability (Chapter 5).

1.3 Outline & Approach

The work of this thesis has been undertaken as follows: First, a minimal dictionary interface was defined. Based on that, a naive dictionary was implemented according to that interface and unit tested. Then, this dictionary was integrated into the existing Crescando code base. As a final step, different dictionary implementations were implemented and compared to the baseline and each other.
The structure of this thesis reflects that approach. In the upcoming chapter, Chapter 2, we present work related to dictionary compression. In Chapter 3 we show how the dictionary was integrated into Crescando and what design decisions were made beforehand. In that chapter we also present Crescando’s architecture in more detail, with a special focus on the parts affected by compression. Chapter 4 is about decisions designing the dictionary itself and the evaluation of various data structures as candidates for such dictionary. In there we present our own novel data structure, the bidi-map. After the two design chapters we provide benchmarking results of our work (Chapter 5). Based on those results we draw conclusions in regard to the usefulness of compression and the individual data structures. In the final chapter (Chapter 6) we draw overall conclusions about the thesis and present future work.

1.4 Design Principles

The work described in this thesis is motivated by the following design principles:

Stay True to the Architecture  Compression shall well integrate into Crescando’s current program flow, philosophy and architecture. This especially means to keep Crescando’s current guarantees in predictability and scalability.

Do Not Regress  When compression is not used, the system shall not exhibit different behavior than the old system without compression support. Nor should performance be impacted in a significant way.

Compartmentalize  Compression shall be transparent to the client and the higher levels of the Crescando systems. It shall be implemented in an encapsulated and easily maintainable way.

1.5 Terminology

In many places in this document, Crescando or parts of the Crescando storage system are compared to traditional systems. This term refers to disk-based relational systems such as IBM DB2, Oracle, MS SQL Server, MySQL, etc.

When we talk about order preserving compression, we mean compression schemes where values in the compressed representation are in the same mathematical order as the uncompressed values.
Throughout the thesis we will refer to the uncompressed representation of data as the value. Its compressed counterpart we refer to as the key. We call the process of transforming such a value into the corresponding key compression. The reverse process we call decompression. A compression dictionary has to provide both directions and when no specific direction is addressed, we refer to simple dictionary access as (dictionary-)lookup.

If the data structure used to implement a dictionary can serve as an index allowing both compression and decompression in sub-linear time, we call such structure bidirectional. (As opposed to data structures that only allow one direction in sub-linear time and the other one in linear or super-linear time.)
Related Work

2.1 State of the Art in Traditional Database Systems

All three leading database vendors (IBM, Microsoft and Oracle) have implemented compression in their database products in the last years. Even though IBM has had compression based on Huffman coding and similar compression schemes in DB2 for the mainframe for many years [10, 21], Oracle was the first vendor to introduce dictionary compression for their database for the x86-architecture [28]. Their implementation works on the page-level, with the possibility of having mixed compressed and uncompressed pages. If a page is compressed, frequent patterns are replaced by a key and a key-value pair is stored in a page’s dictionary. When predicates are evaluated, certain operations can directly be performed on that dictionary (e.g. if a value is not present in the dictionary, no equality predicate on that value will match any record in the page). It was not until Oracle 11g however, that the dictionary could be updated through normal DML statements. Before it was statically initialized during an initial bulk loading.

Unlike Oracle’s and also Microsoft’s compression scheme, IBM’s DB2 compresses at the table level [3]. Similar to Oracle, a table can be mixed with compressed and uncompressed data: Only frequently occurring patterns (over one or multiple columns) are stored in a dictionary and replaced by keys. According to numbers published in [27] table-level compression can yield much better compression ratios than page-level compression (up to 60% for DB2
Related Work - Database Compression and Performance

compared to up to 40% for Oracle). Oracle on the other hand claims that page-level compression is superior in regard to performance. They argue that a record-lookup using a table-wide dictionary always needs an additional disk-access to read the dictionary. Their implementation on the other hand only needs to access one page containing the record and the dictionary. This is only true, however, if the dictionary cannot be cached in main memory, as discussed in [13]. Results from [12] seem to indicate that this is not often the case.

Poess and Potapov [28] further argue that page-level compression is easier to maintain and may adapt more dynamically to changes in distribution. To be able to quickly adapt the dictionary to those changes, Oracle associates reference counters with each value stored in the dictionary, indicating how many times the value appears in a page. A table-wide dictionary may become very static with increased table sizes. This is especially true, if dictionaries are on the attribute level as they are often used in column stores [1]. Therefore Oracles high adaptability may not be an advantage but a necessity of its design.

2.2 Database Compression and Performance

In 1995 Gautam, Jayant and Seshadri [29] argued, that database compression can not only be used to minimize the data size, but also to improve query performance. They studied different compression algorithms on five granularities: file, page, table, record and attribute level. They found that, while yielding the worst compression ratio, attribute level compression is best in regard to performance, as predicates can directly be evaluated on compressed values [18]. Later research in the context of column stores [1] however showed that attribute-level compression not only yields better performance, but also better compression ratios. This is due to the values of an attribute belonging to the same domain. If compression is performed across attribute boundaries, the domain generally increases.

Abadi et al. [1] also highlight that integrating the compression transparently to the higher levels of the DBMS is important and that there is a need to have compressed and uncompressed representation of the data in the engine. Their tests show, that opposed to row-based systems, the I/O savings of compression do not translate to increased overall system performance. They attribute this to their system already being much less I/O bound on workloads and the added overhead in CPU-cycles becoming the limiting factor. Similarly, Westmann et al. [33] identified that to increase overall performance compression has to be
Related Work - Order Preserving Dictionary Compression

very light-weight. Furthermore, the paper highlights that for compression to be most effective the optimizer must be aware of compression and the advantages and disadvantages of applied compression schemes. This has also been studied extensively in \[11\].

We will use the findings about having to add light-weight compression, making compression transparent to the higher levels of the system and providing the optimizer with knowledge about compression in Crescando. We will evaluate if I/O savings from disk-based row-stores translate to an in-memory, scan-based row-store.

2.3 Order Preserving Dictionary Compression

Order preserving compression schemes can either generate variable \[5\] or fixed-size keys \[9\]. Variable length keys are easier to extend, but cannot provide the same performance gains as can be achieved using fixed length keys. Those performance gains though, can be negated when keys have to be re-assigned. Binnig et al. \[9\] argue, that the performance gains outweigh, especially for analytical workload applied to column stores. Their dictionary implementation uses shared leaves for the decoding and encoding data structures. It is further optimized towards bulk-loading and bulk-decompression, which is the most performance-critical functionality needed to materialize results during query execution.

The dictionary presented in \[9\] is a global one.

2.4 Data Structures for Storing String Values

Over the years many data structures for storing string data were proposed. Most of them focus either on space and/or time efficiency. Apart from simple, multi-purpose structures such as hash-tables and binary trees, different kinds of tries \[20, 7, 26, 30\] have been suggested.

The trie was first presented by Edward Fredking in \[16\]. The original trie was fast at string lookups, but not very space-efficient. Burst tries, as presented in \[20\], and patricia tries \[26\] are two different approaches to improve upon the original trie in that regard. Both approach the problem by using alternative leave nodes. Leave nodes in burst tries are a linked list of values stored under
a trie node. Some heuristic determines when such a leave node is split into multiple leave nodes that are attached to a new trie node. Patricia tries work by merging single leave nodes with their parent, thus eliminating pointer overhead. Other approaches to saving space were also suggested, such as to use linked lists to store pointers in a trie node instead of fixed-size arrays [22]. This idea is orthogonal to the ones in burst and patricia tries.

All presented approaches, while improving upon the original trie in regard to space, are not very efficient in regard to performance on modern hardware due to their use of linked lists and pointers. It was shown in [32], that linked lists perform very poorly on current CPUs because of pointer-chasing. It would therefore make sense to replace the linked lists used in the structures above by linear arrays. Such solutions have been proposed for burst tries (HAT-trie, [7]) and patricia tries (CS-array trie, [9]). The HAT-trie has also been shown to not only improve upon other trie implementations in regard to access and update performance, but also in regard to space consumption. This at least is true, if the linked lists are replaced by compact array structures, that are resized according to the elements needed. Even though such arrays may enforce copying of their data when resized, in practice they do not perform worse than paged implementations [8].

Apart from tries, simple hash tables have been shown [19, 6] to also perform well for storing and retrieving string data. This is especially true, if cache conscious hash tables as proposed in [8] are used.

For Crescando we will evaluate hash table and trie based dictionary implementations.
3

Dictionary Integration

In this chapter we describes the integration of dictionary compression into the Crescando storage engine. For this purpose we give a short introduction to the Crescando system with a special focus on the areas affected by compression. Then we describe in detail how existing behavior was changed and augmented. We show the data flow through the system.

The integration was conducted using a simple interface and a naive dictionary implementation. More advanced implementations are studied in the next chapter.

3.1 The Crescando Storage System

Crescando is a novel relational table implementation developed at ETH during the last years. It was built to provide predictable performance under unpredictable workloads and to be highly scalable in the number of processor cores.

Predictability is achieved through a novel collaborative-scan algorithm called Clock Scan, that interleaves select and update operations. During a Clock Scan cycle, the data is continuously scanned. Operations are batched together and applied to the data under the scan cursor. Clock Scan can index operations on the fly, and eliminates the need of indexes on the data used by traditional systems.
3.1.1 Architecture Overview

At the core of the Crescando system is the Crescando storage engine, which is used by the client in the form of a C-library (see Figure 3.1). On top of this C-Interface there is a distributed system consisting of different storage nodes that are orchestrated by so called aggregators. A storage node runs an instance of the Crescando engine. Data is horizontally distributed amongst these nodes. A detailed description of the distributed architecture can be found in [17]. For the purpose of this thesis we will focus on the stand-alone Crescando engine, as compression is implemented on that level, transparently to distribution.

Further details about Crescando can be found in the original paper [31] and in the paper about distribution [17].

3.1.2 Operations & Predicates

Crescando supports select, insert, update and delete operations. Those operations are divided into read (select) and write (insert, update, delete) operations and in conditional (select, update, delete) and non-conditional (insert) operations. Conditional operations have a predicate list that contains zero or more predicates. Such predicates are triples consisting of an attribute, an operator and a value. A predicate list could, for instance, contain the predicates...
Firstname = “Hans” and Location = “Zurich”. For a predicate list to be satisfiable, all predicates have to be satisfiable (i.e. conjunction). Select operations have in addition a list of projected attributes, update operations a list of update attributes and update values. Insert operations only consist of list of records.

### 3.1.3 Storage Node & Partitioning

A storage node is an individual machine running the Crescando engine and storing part of the table in a row-wise fashion. On one such machine, the data is horizontally partitioned (Figure 3.3) into fixed size memory segments, which are further divided into chunks (they correspond to pages in a traditional disk-based system and fit into the CPU’s L1-cache). Those chunks contain a number of slots that can be occupied by records or empty. The memory segments are managed by so-called scan threads (Figure 3.2) that have hard affinity (run on a specific core). If supported by the hardware, the segments are allocated on the main-memory region that has minimal distance from the core the corresponding scan thread is running on. Individual scan threads do not communicate with each other or share state, but are orchestrated by a control-thread (the controller).

The controller communicates with the scan threads using queues. For this purpose, each scan thread has its own input queue and result queue. The input queues are fed by the controller and the results queues are periodically flushed by the scan threads to the controller’s result queue.

![Figure 3.2: Overview of a Crescando storage node running several scan threads (taken from [31])](image)
Figure 3.3: Partitioning and distribution of a Crescando table
3.1.4 Scan Cycle

In addition to its input queue a scan thread also has two queues for operations currently being executed: one for writes and one for reads. Those queues are allocated with minimal distance from the CPU core.

A scan cycle is divided in the following steps:

**Activation** Operations are copied from the input queue into the write and read queues.

**Optimizing** The optimizer creates read and write query plans for the operations in the respective queues. Those query plans may contain indexing structures, as shown in Figure 3.4.

![Figure 3.4: The index union optimizer (taken from [31])](image)

**Execution** Iterate through all the data, chunk by chunk and apply the write and read query plans to the chunks. Indexed and non-indexed predicates are evaluated in separate code paths. Both iterate through the chunk slot by slot.

When a slot is empty and there is an *insert* operation in the write queue, the slot is filled with data from the insert. When a slot is occupied, predicate lists of conditional operations are evaluated.

When a record matches an *update* operation’s predicate list, the record’s values are updated according to the update’s definition. When a record matches a *delete* operation’s predicate list, the slot the record is contained in is set to empty.

When a record matches a *select* operation’s predicate-list, the projected values are copied into a result-object which is then added to the result-queue. When the queue fills up, results are flushed to the controllers result queue.
Deactivation Operations are deactivated and end-of-result indicators are added to the result queue. Remaining results in the result queue are flushed to the controller’s result queue.

Most of the time the engine spends in the execution phase, which is the heart of the actual Clock Scan and therefore the most performance critical phase.

3.1.5 Schema, Compiler & Interface

The Crescando storage library is accessed through a C-interface. The main methods provided by that interface are `enqueueOperation` and `dequeueResult`. The `enqueueOperation` function is used to add operations to the storage system and the `dequeueResult` is used to fetch results from the engine’s result queue. In the engine, those functions map to functions provided by the controller.

A Crescando schema is defined in an SQL-like create table statement (for an example see Section A.1). This table definition gets translated into C++ code by a schema compiler. The generated C++ code then has to be compiled and linked against the Crescando base library. This process results in a custom library with schema dependent code (Figure 3.1).

Such approach has the advantage of generating very fast and efficient code, but the downside of having a statically defined schema (as opposed to one that could adapt during runtime). The code of a schema compiler is also harder to maintain. Only a few classes are therefore extended by schema-dependent sub-classes or instantiated with schema dependent generic parameters. Our rule is to only make that code schema dependent that is highly performance sensitive. Those classes are mainly classes related to optimization and classes related to execution.

3.1.6 Recovery

Recovery in Crescando is based on logical logging and heavy-weight checkpoints. The checkpoints are created by a special checkpoint select that matches any record. On crash recovery, such a checkpoint is loaded into the segment and unfinished operations are replayed. Recovery will be touched by compression, as the dictionary must able to be reconstructed from recovery data.
3.2 Design Decisions

Before implementing the dictionary, some choices along various dimensions in regard to Crescando had to be made. In this section, we will study those dimensions and present our design decisions.

3.2.1 Granularity & Partitioning

As discussed in Chapter 2, research and commercial products from recent years [1, 29, 3, 13, 27] have shown that dictionary compression can be very effective not only for saving storage space, but also for increasing query performance. There is no agreement however, on which level of granularity such compression is most effective. Traditionally, database vendors of row-stores applied their compression horizontally on the page or table level, mixing data from different attributes in the same dictionary. Research on column stores has shown, that applying compression vertically can yield to better compression ratios due to reduced domain ranges. In the following we will analyze the partitioning possibilities in Crescando and comparable systems, and decide upon a fitting compression scheme based on that existing research.

As explained before and depicted in Figure 3.3, the data stored in Crescando is horizontally distributed onto several storage nodes. Those nodes consist of multiple segments, which are further divided into chunks. A compression dictionary could be added on any such level.

Compression could be done adaptively or statically. An adaptive scheme only compresses frequently occurring values or even only enables compression at all when indicated by certain heuristics. Such scheme is desirable, if the domain of values is large. Adaptive compression however requires dynamic schema or variable sized fields, which both are not available in Crescando at this point.

As we want to preserve Crescando’s current shared-nothing architecture, a distributed dictionary on the system level is not a viable solution. Such dictionary would induce considerable communication overhead. Additional logic for caching parts of the dictionary locally would have to be added. Equally, a dictionary on the storage node level, that is accessed by multiple scan threads, would have to be implemented in a thread-safe manner such that individual scan threads can access them concurrently. This is still not shared-nothing and may hinder Crescando from scaling in the future.
Dictionary Integration  -  Design Decisions

Adding dictionaries to individual chunks would very likely exhibiting best locality. It however induces an additional space overhead compared to a dictionary on the segment level due to overlapping dictionary contents. As a result, compression ratios may be much worse compared to a segment-wide dictionary. The segment level is therefore the best fit in regard to Crescando for placing compression dictionaries.

As we ruled out adaptive compression schemes before, if we would add only one dictionary to the segment level, all the data on that level would have to be compressed. This is not feasible, as the resulting domain from values over all attributes is to large for dictionary compression to pay off. We therefore have to divide a segment further and employ multiple dictionaries. As explained before and shown in [1], a vertical partitioning, where dictionaries are added to individual attributes, can result in considerably better compression ratios.

Theoretically, a dictionary could span multiple attributes. This may be feasible, if the values in those attributes are strongly correlated. Otherwise, the size of the resulting domain would again be too large for dictionary compression to pay off. In addition, combining multiple attributes will make the evaluation of predicates directly on keys much harder to achieve. Consequently, we decided against combining multiple attributes in a dictionary.

To summarize, compression dictionaries are added for each individual attribute on the segment-level. This corresponds to a vertical partitioning per attribute and a horizontal partitioning per segment. Which attributes are compressed is right now a choice the user can specify.

3.2.2 Garbage Collection

Values that are removed from the dictionary do not have to be physically removed from the underlying data structure immediately. Instead, removal can be delayed to a dedicated garbage collection phase that reorganizes the data structure and removes values that are not referenced anymore.

The simplest form of garbage collection is reference counting: Every value added to the dictionary is augmented with a counter of how many records in the database reference that value. The counter is increased every time a value is added to the dictionary and decreased when a value is removed. If the counter becomes 0, the value is not used anymore and can be removed from the dictionary. This does not have to happen instantly, but may be delayed to a dedicated garbage-collection phase.
Reference counters can also be used for other purposes. A reference counting dictionary provides exact information about the data distribution in the segment. This information can be used to optimize the dictionary to exhibit better cache locality for frequently occurring values or to help the optimizer in finding a better query plan.

Due to its scan nature, a mark-and-sweep like garbage collection scheme is also straightforward to integrate into Crescando. For this purpose, every value is augmented with a bit indicating whether it is in use or not. When garbage collection is triggered, a special dictionary select that matches any record is added to the scan thread. For each record the dictionary select extracts the compression-key and marks it as being in use in the dictionary. This mark phase can be executed in parallel to the normal scan operations. When the dictionary select finishes, the sweep phase is triggered and all values that have not been marked are removed from the dictionary.

Another aspect of garbage collection is, when to trigger it: when memory runs out, periodically, or at other conditions. Using reference-counting, one could theoretically perform garbage collection and reorganize the dictionary every time a counter becomes 0. To achieve reasonable performance, this is very likely not a good scheme. Therefore a periodical execution or a lazy execution, where garbage collection only happens when memory runs out, may be more appropriate.

Given that reference counters may be useful for future optimizations, we do not settle for one garbage collection scheme or the other, but will implement both and perform preliminary evaluations.

### 3.2.3 Data Types & Ranges

Dictionary compression pays off, when the number of unique values is significantly smaller than the number of instances in the database and the average length of a value is considerably larger than the length of a key. Even thought there are numerical or date fields those qualities apply to, the gains often are much smaller than for string values. In the scope of this thesis we therefore only support compression on string attributes.

We further impose the restriction to only support strings of up to 255 characters, as this will allow the use of 1 byte to store a string’s length. This restriction is reasonable, as longer strings are very unlikely to be in a domain small enough for efficient dictionary compression.
Dictionary keys are of fixed size. We choose a size of 32bit which allows over 4 billion unique values. It was decided against having keys of only 16bit, as the possible 65,536 values may run out for certain domains (our test data provided by Amadeus, for instance, has close to 500,000 unique first names, see 4.2.1).

### 3.2.4 Predicate Evaluation

It has been shown that evaluating predicates directly on the compressed representation can improve query performance [18, 29, 33]. This can easily be done for equality and inequality predicates, as the equality of two keys that result from dictionary compression implies the equality of the corresponding values. The same is true for is-null and is-not-null predicates. To be able to evaluate range predicates directly however, the compression scheme must be order preserving.

There are many existing algorithms for generating and maintaining order preserving keys [9, 4, 25], but they all have in common that eventually there is the need for re-assigning keys to keep them ordered or for increasing the length of a key. The later is not possible in Crescando, due to the absence of dynamic schema or variable length fields. Re-assigning of keys may be cheaper in Crescando as in traditional systems, as this could simply be performed by adding a specific key-update operation to the scan, that executes before any other operation. Still, reorganizing an order-preserving dictionary is expensive. Given the decision to only support compression on string values, we do not see to much value in supporting range operations directly on keys, as arbitrary range operations on strings are rare. The main use case are prefix searches such as “return all first names starting with A”, which can be translated into two range predicates. Those queries however, can be processed differently, as discussed in Subsection 6.1.5.

Due to those reasons, we decided against order preserving keys. In consequence, range predicates can not directly be processed on compressed values, but will enforce decompression. Other operations (equality, is-null, etc.) however will be evaluated directly on keys.

### 3.2.5 Null-values

The dictionary shall provide support for null-values. This means, that a null-value will always be compressed to a specific null-key. This way, an is-null
predicate can directly be evaluated on that static null-key. On decompression, a null-key has to result in the corresponding null-value.

### 3.2.6 Dictionary Interface

Given our desire to support compression and decompression of data, the dictionary has to support those two operations. We also introduce two methods to insert new values into the dictionary and to remove them. We allow removal, as discussed in Subsection 3.2.2 to be performed lazily.

#### Minimal Interface

- **insert(value)**: Insert new value into dictionary, and assign key (or increase reference counter).
- **remove(key)**: Remove key-value-pair from dictionary (or decrease reference counter).
- **compress(key)**: Get value for specified key.
- **decompress(value)**: Get key for specified value.

#### Extended Interface

Depending on the choice of garbage collection method and data structure, the dictionary may need to be extended by additional methods. E.g. the dictionary may need to be reorganized or optimized at some point, due to defragmentation or changes in distribution. One may also want query statistical information such a size and number of stored values from the dictionary.

- **getValues()**: Returns number of values stored in dictionary.
- **getSize()**: Returns the amount of memory consumed by the dictionary.
- **addRef(key)**: Increase reference count on key-value pair (precondition that value exists).
- **reorganize(bool)**: Reorganize dictionary to improve performance or remove old values. Parameter whether to force reorganize, or to only reorganize if certain criteria are met.
- **startGC()**: Start garbage collection (mark phase).
- **endGC()**: End garbage collection (sweep phase).
3.3 Architecture

This section describes the general architecture used to integrate dictionary compression into Crescando. It shows what changes had to be made to the existing architecture to support compression.

3.3.1 Interface & Schema

One of our design principles is to introduced compression as transparently as possible, i.e. with minimal changes to the C-interface and to higher levels of the system. The only extensions we added are to provide the system and the client with information about compression. For this purpose, the meta data provided by the engine has been augmented by a boolean value for varchar-attributes indicating whether the attribute is compressed or not. Furthermore, the existing statistics was extended with information on compression efficiency, number of unique values and dictionary sizes.

To allow a user to specify which attributes are compressed, the existing grammar for the Crescando schema has been augmented by a new keyword COMPRESS and a new grammar rule for varchar attributes:

\[
\text{string_type} = \text{VARCHAR (n) COMPRESS (bits)}
\]

The parameter \text{bits} to the COMPRESS keyword defines the number of bits used for dictionary keys. Right now the only supported value is 32.

3.3.2 Engine Overview

As most of the running time of the Crescando engine is spent in the execution phase of each scan thread, reducing dictionary lookups to a minimum during this phase is crucial. Not only does the such lookup result in added CPU-cycles, it also increases the working set of execution. Therefore, decompression during execution is only performed for result creation and range predicates on compressed attributes. To further reduce the fill-up of the cashes with dictionary-data, decompression for result creation is delayed until the result-queue is full and has to be flushed to the controller. This process is explained in Subsection 3.3.4.

To allow predicates to be evaluated directly on compressed values during execution, they have to be preprocessed during the activation phase (Figure 3.5).
Dictionary Integration  -  Architecture

For this to work, it has to be further ensured that the keys are not reassigned during the execution phase. This is guaranteed by also preprocessing update and insert operations (see Figure 3.6 and Figure 3.7). To support operations that cannot directly be executed on keys, a new kind of predicate was added. During preprocessing, predicates on operations that cannot be performed directly on keys are converted into such a dictionary-lookup predicate.

The detailed preprocessing steps for all operations are explained in Subsection 3.3.3.

Given this architecture and to integrate compression transparently according to our design principles, it was necessary to introduce two different representations of records inside the Crescando storage engine. A compressed repre-
sentation is generally used inside the scan thread, whereas an uncompressed representation is used inside the controller and to communicate with the client application.

Similarly, a new meta data structure had to be introduced that passes information about compression to the scan threads. The alternative would have been to make schema dependent scan thread classes that are generated by the schema compiler. As the creation of dictionaries is not performance critical, see Subsection 3.1.5, and as such change would result in a considerable amount of code changes, we decided against such approach.

The dictionaries are created by the scan thread on initialization, when the memory segment is allocated. The dictionaries however are stored in memory separately from the segment, and not necessarily in the memory closest to the scan thread. This may be improved in the future (Subsection 6.1.2).

### 3.3.3 Scan Cycle

As described earlier and depicted in Figure 3.8, most of the dictionary operations during scan are performed in the activation phase. For non-reference-counting dictionary implementations, all dictionary access during execution will be read only (except if garbage collection is running, see Subsection 3.2.2). The following will show in detail how the scan was changed and how dictionaries are accessed during scan.

**Activation**

During the activation phase, scan operations are copied from the input queue to the scan threads internal execution queues. Whereas in the system without compression this was a simple clone operation, now steps to convert those operations into a compressed form (or internal form) are performed:

Predicates on compressed attributes of conditional operations are preprocessed. If such predicate’s operator can directly be applied to keys (such as equality, inequality, is-null or is-not-null), then the predicates value is compressed. If the predicate-value is not contained in the dictionary, the predicate and therefore the predicate list can not be satisfied by any record in the segment. Consequently the corresponding operation of such unsatisfiable predicate list is never added to the execution queue. If a predicate is on a compressed attribute, but of an operation that can not be applied directly on keys (such
Figure 3.8: Overview of dictionary access during a scan cycle
Dictionary Integration - Architecture

Figure 3.9: Dictionary access during activation phase
as range-operations), then the predicate is converted into a special dictionary-
lookup predicate.

The new data in \textit{insert} and \textit{update} statements is compressed and new values
are inserted into the corresponding dictionaries. This early compression can
theoretically lead to values never actually added to the segment being added
to the dictionary. Good heuristics however can ensure that garbage collection
is triggered when many such values exist in the dictionary.

\textbf{Optimizing}

The existing optimizer does not index any range attributes on string values.
This has not been changed for compression, as the advanced indexing structure
(Subsection 6.1.5) we propose to be implemented in the future is out of the
scope of this thesis.

Equality predicates on strings, which would not have been indexed in the
uncompressed case if above a certain size, have been enabled for indexing.

Furthermore, compression invalidates certain assumptions of the optimizer
that will have to be adapted in the future.

\textbf{Execution}

Figure 3.10 shows a simplified version of the processing during execution. It
omits that write operations are strictly executed before read operations and
indexing structures are used instead of simple iterations through operations
and segments. For the purpose of showing how dictionary compression has
been integrated, however, we can abstract from those details.

Execution is essentially the same as shown for the uncompressed case in Sub-
section 3.1.4. The main change is, that to distinct between a normal predicate
and dictionary-lookup predicate that enforces decompression, a branch had to
be added to the execution path of non-indexed operations.

If a predicate lists contains a dictionary-lookup predicate, the key for the at-
ttribute the predicate operates on stored in the current record is decompressed.
Due to the way this is implemented at this time, such lookup has to happen
for every predicate of every operation anew. We did not invest time in opti-
mizing this (e.g. by caching the decompressed value), as there are better ways
Figure 3.10: Dictionary access during execution phase, simplified (operation-loop has been omitted and select and write operations, in reality executed individually, merged)
to support range predicates (Subsection 6.1.5), which are out of the scope of this thesis however.

When predicates match, an operation is executed. In the case of a select, this means that the matching record is copied into a result object that is enqueued to the result queue. It is only when this result queue fills up, that the result object’s values are decompressed (see Subsection 3.3.4).

For reference counting dictionary implementations, executing a delete or update operations results in the reference counters of affected compressed attributes to be updated.

**Deactivation**

During the deactivation phase end result markers for all operations are created. Then all remaining result in the result queue are flushed to the controllers result queue (see Subsection 3.3.4).

### 3.3.4 Result Creation

To improve instruction and data cache locality, the data in result objects is only decompressed once the result queue is full. Then execution is interrupted, results are dequeued, decompressed and added to the controllers result queue. Once the scan threads result queue is empty, execution is resumed.

### 3.3.5 Table Statistics

To gather statistics about data-distribution and null-values, Crescando uses a special statistic-select operation. Such operation is added by a statistics manager for each attribute individual. Each statistic-select then estimates distribution on its attribute using probabilistic counting [15]. The statistic-select has been extended to read information for compressed attributes directly from the compression dictionary.

### 3.3.6 Recovery

To make recovery work with compressed values, we implemented a simple approach, under which each record is decompressed before being written to disk.
This is of course not very efficient and shall be replaced in the future by a serialization of the dictionary to disk.

3.4 Summary of Changes

The following summarizes the main changes that had to be made to the system to support compression:

- New meta data was added to provide the scan threads and other parts of the system with information about compression.
- Operations were extended by methods to allow preprocessing to convert values into their compressed form.
- Predicate lists were extended by methods to convert predicate values into a compressed form.
- A new predicate type (dictionary-lookup predicate) was added. It is used for predicates on compressed attributes that cannot be evaluated on the attribute’s compressed form.
- A branch had to be added to the execution core to support dictionary-lookup predicates.
- Statistics and meta data provided through the C-interface were augmented by information about compression.
In this chapter we study data structures for use in a dictionary and compare them to each other on several dimensions. We list those dimensions and the factors that influence the decisions along them. Based on this, we compare some existing data structures against each other and motivate our own, novel bidirectional data structure, the bidi-map. We provide implementations based on the bidi-map and other data structures which will be evaluated as part of the next chapter.

As discussed in Section 3.2, the dictionary shall have non-order preserving keys of 32bit length. It shall support string sizes of up to 255 and provide methods defined in Subsection 3.2.6.

Such a dictionary can essentially be implemented in two ways: using two data structures, each providing efficient lookup in one direction, or using a single data structure that efficiently supports lookups in both directions. For the first implementation we compare existing data structures to be used as dedicated compression and decompression indexes. For the second implementation we propose a bidirectional data structure we call the bidi-map. To be able to evaluate individual data structures and to motivate our own, we first look at dimensions to be considered when designing a compression dictionary and then look at the factors that influence those dimensions.
4.1 Dimensions

**Number of unique values** How many unique values can the dictionary hold? Can it support the whole key range under all circumstances (i.e. a hash table of fixed size is depending on a good hash algorithm).

**Size of dictionary** How much memory does the dictionary consume? What is the fixed size overhead, what is its variable overhead?

**Performance of insert/deletes and lookups** How fast can we insert new values or remove existing ones? How fast can we increase the reference count or look up values?

**Cache consciousness** Is the dictionary cache conscious or can it be implemented in a cache conscious manner? Can the dictionary be optimized to exploit locality based on certain properties (e.g. value distribution)?

**Degeneration/Fragmentation** Does fragmentation occur? Do we need to execute maintenance tasks?

**Adaptiveness** Can the dictionary dynamically adapt to the given data or do we have to statically define some parameters beforehand? How sensitive is the dictionary to variations in the data distribution?

**Bulk optimized** Is the dictionary optimized towards bulk operations?

**Symmetry** Does the dictionary perform better at decompression then at compression or vice versa?

4.2 Influencing Factors

4.2.1 Data Distribution & Value Lengths

An important factor is the distribution of data stored in the dictionary. As explained in Subsection 3.2.3, we currently only support compression on string attributes. Hence, in the following, we have to study string data only.

String values in the “real world” generally exhibit a skewed, zipf-like distribution [23], where some few values are much more frequent than the others. The string attributes from the data provided by Amadeus exhibit such distribution (4.2.1). A dictionary may make use of a distribution’s skew by providing shorter access paths for the most frequent values. The more frequent those most frequent values, the better such optimization pays off. To visualize this,
Dictionary Design & Data Structures - Influencing Factors

we plotted the frequency distributions over a specific domain and the integral over those frequencies.

Another factor in regard to the efficiency of compression is the total number of unique values. Even if a value distribution is heavily skewed, dictionary compression may not pay off when there are many unique values with low frequency (for an example of such data see 4.2.1). An important indicator of the feasibility of compression is therefore the ratio of number of unique values compared to the number of value instances. Only if the number of instances is significantly higher than the number of unique values dictionary compression will pay off.

Amadeus Itinerary Schema

The Amadeus Itinerary schema is a denormalized table of flight bookings containing one record for every person on a plane. It has been studied extensively in [17], Section 1.2 and Chapter 4. A record in the Amadeus Itinerary schema is approximately 350 bytes in size and consists of 47 attributes (Section A.1). The schema contains four string-attributes that are suitable for dictionary compression: office, firstname, lastname and sgtVendorValue. Other string-attributes have either values that are too small or a domain that is too large to significantly benefit from compression.

![Figure 4.1: Value distribution of attribute firstname](image-url)

The attributes firstname and lastname contain the passengers name and
titles (i.e. Mrs, Prof, etc.). The value distribution is depicted in Figure 4.1. From the 8 million records from British Airways provided by Amadeus there are 700’000 unique first names of average length 9.0 and maximum length 56 and 694’000 unique last names of average length 7.4 and maximum length 55. The last names distribution is visible in Figure 4.2.

![Figure 4.2: Value distribution of attribute lastname](image)

The attribute office identifies the travel agency that generated a certain booking and sgtVendorValues is used as a key into another table. office has 43’000 unique values of average length 8.99 and maximum length 9. sgtVendorValues has 40’000 unique values of average length 12.93 and maximum length 15.

**Swiss Phone Book Names**

We also studied another data-set, similar to the first and last names in the Amadeus data. For this purpose we extracted all surnames from the Swiss phone book. Those names include maiden names and titles and the longest entry is 75 characters. The average length is 10.51. The data set consists of 3 million values, 1.1 million of which are unique. The value distribution is depicted in Figure 4.5. Compared to the data sets from Amadeus, there are much fewer unique names and even more names are needed to cover a large part of the data.
4.2.2 Projected Workload

The more queries project on a compressed attribute, the more likely it is that decompression is needed to generate a result. In the case of the Amadeus workload, 93.3\% of the queries project on \texttt{firstname}, \texttt{name} and \texttt{office}. Only 5\% project on \texttt{sgtVendorValues}. 
Similarly, the frequency of predicates on compressed values can influence performance and has to be taken into account. In the case of the Amadeus workload, there are very few queries that feature equality predicates on name and firstname (see 5.2.2). There are at this point no range predicates on compressed values, but there is the desire for support of prefix-predicates.

### 4.2.3 Segment Size & Segmentation Strategy

On our current hardware (see 5.2.1) Crescando exhibits latencies of around 1 second with a segment size of 1gb. A larger segment size may result in more unique values to be stored in the dictionary. What is even more important than the size of a segment, however, is the segmentation strategy: if a compressed attribute heavily correlates with the attribute according to which data is divided into the segments, there will be fewer unique values to be stored in a dictionary than when there is no correlation between the segmenting attribute and the compressed one.

### 4.2.4 Implementation & Architecture

One of the most influencing elements is Crescando’s scan-based architecture. Due to the predominance of the execution phase (see Chapter 3 for details), dictionary access during this phase is most performance critical. In this phase
the only dictionary access is to decompress result tuples or to evaluate range predicates. Therefore, the decompression performance of the dictionary is much more critical than the compression performance and cache-consciousness is important.

On the other hand, it is less important for a dictionary not to exhibit fragmentation or other kind of degeneration over time, as at the end of the scan, a degenerated dictionary can be optimized with low impact to the overall system performance.

### 4.3 Design Decisions

#### 4.3.1 Overview

There is always a trade-off between the dimensions presented before and no data structure can be perfect in all regards: To increase performance one often has to add to a data structure’s complexity, which generally results in additional memory overhead. This, on the other hand, may lead to worse cache consciousness and negatively impact performance. Therefore, one has to carefully take into account the necessities of a specific system and analyze the available data, as we did in the last section, to prioritize the importance of certain dimensions to chose from.

As indicated, decompression performance is much more important than compression performance, due to Crescando’s scan-based architecture. We therefore do not need symmetry in a dictionary. Using compression on the attributes presented in 4.2.1, we can fit the amount of data on the Amadeus Itinerary schema equal to an uncompressed 1gb segment using only 626mb (excluding the amount of memory used by the dictionaries). As the actual unique data to be store in all the dictionaries is very small (less than 10mb) compared to the segment size, we may accept some memory overhead in the dictionary. This is especially true for a variable overhead in the number of values: if there are that many values that the dictionary size becomes significant, compression may not be indicated in the first place. However, a larger dictionary will use up more cache during execution, which may impact overall system performance. In this regard, it would also be beneficial, if the chosen data structure was small and itself cache conscious.

Ideally, our data structure would not degenerate over time and should automatically adjust to changes in data distribution. It is not always possible to
adapt a data structure to support such properties however. Even when it is possible, adding logic to prevent from degeneration may impact performance. As we can delay reorganization tasks that deal with degeneration to the end of the scan, we can accept a data structure that does degenerate, but which can be easily reorganized. The same holds for optimizations towards a skewed distribution: such optimizations do not have to be performed eagerly, but can be delayed to the end of the scan.

Closely related to the above is the question whether a data structure needs any parameterization (e.g. the size of the array of a hash table) to perform reasonable under specific conditions. Given our design principles such parameterization is not desirable, as predictability cannot be guaranteed if data is inserted that diverges from the assumptions used to set the parameters. An adaptive data structure, that may not be optimal under all conditions, but will behave predictably under any conditions, is preferable.

There is also the question of whether to optimize for bulk operations or not. Due to Crescando’s architecture, where record for record is processed, bulk operations are rare and targeting a data structure towards fast bulk operations shall not be a primary goal.

4.3.2 Summary

**Number of unique values** The dictionary shall be able to hold the full range that can be provided by the 32bit keys.

**Size of dictionary** The dictionary should be as small as possible, so the CPU cache can be used for the segment data during the execution phase.

**Performance of insert/deletes and lookups** Key lookups (decompression) shall be very fast. Insertion and compression performance is less important.

**Cache consciousness** The dictionary shall be implemented in a cash conscious manner, with pointers kept to a minimum and ideally exploiting locality.

**Degeneration/Fragmentation** The dictionary may degenerate, if it can be re-organized to an optimal form.

**Adaptiveness** The dictionary should perform well under value distributions as shown in 4.2.1 and not significantly degenerate under other distributions.

**Bulk optimized** The dictionary need not be optimized for bulk operations.

**Symmetry** The dictionary need not be symmetric, decompression performance is more important than compression.
4.4 Data structures

4.4.1 Evaluation of Existing Data Structures

Given the decisions presented in the last section, we are looking for a data structure that features fast decompression, is cache-conscious, supports prefix searches, may degenerate over time and is preferably small in size.

An analysis of data structures for storing string values was presented in Section 2.4. In [7, 6, 8] it was shown that hash tables with exact-fit array buckets and the HAT-trie perform best in regard to performance and space. We argue that both those data structures are also very suited in regard to the dimensions discussed before.

Tries and hash-tables have different and sometimes orthogonal characteristics. They both have in common that they are sensitive to the values stored. Tries are further also sensitive to the order of values inserted. For hash-tables, this only holds true for certain implementations and is mainly depending on the choice of collision resolution scheme. Even though the proposed hash-table using exact-fit array buckets experiences such sensitivity, we argue that it can be reorganized more easily, e.g. by reordering the entries in a bucket. Optimizations in a trie are less obvious and may need more complex heuristics.

Hash-tables are more universal than tries in the sense that they can be used as indexes for strings as well as for integers. They are therefore suitable for compression and decompression alike and may even be combined in a way to server as both, which we present in Subsection 4.4.2. Tries on the other hand work by merging common prefixes of the stored values and are consequently mainly tailored at storing string data.

If many of the strings stored share common prefixes, a trie may be more space efficient than a hash-table. The common prefix can further be used for efficient prefix-lookups to the trie, which may be used in preprocessing of prefix-predicates using the dictionary. Consequently, to evaluate if a trie will consume less space due to mentioned merging of prefixes, we implemented a trie-based dictionary. We present our trie implementation in Subsection 4.4.3.

To summarize, we decided upon evaluating tries and hash-tables. We use hash tables as they are universal and simple to implement and extend. We implement tries to evaluate if they can provide space-savings and because they may be used to support prefix-predicates in the future.
4.4.2 The Bidi-map

Any data structure used to store key-value pairs can be thought as being bidirectional. Most data structures however, only allow fast (i.e. sub-linear) access in one direction. The bidi-map provides sub-linear access in both directions. This is achieved by generating keys in a way that they convey information about the location of the value in the data structure. Such scheme is possible due to the fact that the key-value mapping is not given externally, but can be generated by the dictionary itself. A key can be seen as a handler to identify the value it was generated from.

![Diagram of bidi-map](image)

**Figure 4.6:** A dictionary using the bidi-map

The most naive implementation of above idea is to store all values in a huge, fixed size hash-table and to set the key to the value’s position in that table. This may be a good solution for 16bit keys, but for the 32bit keys we decided to use (Subsection 3.2.3), the resulting array would be over 4gb in size. We therefore propose a two-staged implementation with a hash table containing a secondary data structure in the individual hash buckets. Such 2-staged approach enforces the key to be split into two parts: one that serves as an index in the hash-
table and the other one that is used to locate the value in the 2nd stage data structure. We will call the first part *bucket index* and the second part *intra-bucket index*. This is illustrated in Figure 4.6, using exact-fit array buckets [7] as the 2nd stage. We argue that simply storing pairs of intra-bucket indexes and values consecutively in such an exact-fit array performs well enough for real-world data sets.

In our implementation we set the 16 low bits of the generated 32bit key to the bucket index and the high bits to the intra-bucket index. This is motivated by the fact that the keys may be used in hash-indexes created during the optimization phase. Those hash indexes work by taking the low bits of a key as the hash-value. When those low bits are already used as an index into another hash-table, i.e. when they are generated by a solid hash-function, there will be few collisions in the generated index. If, on the other hand, the low bits would be set to the intra-bucket index, which in our implementation is a simple counter, significantly more collisions would occur.

**Compression**

As illustrated in Figure 4.7(a), compression from a value into a key works as follows: The value is fed to a hash function, that generates a 16bit hash value. This hash value is then used to fetch the bucket from the hash table. The bucket is then queried using a linear scan for the intra-bucket index using the value. When the value is found, the according intra-bucket index is fetched and combined with the bucket index to generate the key. The key is then returned as a result of compression.

**Decompression**

As illustrated in Figure 4.7(b), decompression from a key into a value works like follows: First, the key is split into its intra-bucket and bucket index parts. The bucket index is then used to fetch the bucket from the hash table. The intra-bucket index is used to locate the value in that bucket, by linearly scanning over the pairs of values and intra-bucket indexes. When the intra-bucket index is found, the corresponding value is fetched and returned as the result of decompression.
Figure 4.7: Data flow for compression and decompression in a bidi-map
Insertion

Insertion works similar to compression: The value to be inserted is fed to a hash function, that generates a 16bit hash value. This hash value is then used to fetch the bucket in the hash table. If the value can be found in the bucket, either the reference counter of that value is increased (for reference counting implementations) or no action is performed. If the value is not yet contained, a new intra-bucket index is created by increasing a counter that is contained in each bucket.

Space Considerations

A fixed size hash-table not only has a static memory overhead, but also scatters the data all over the main memory, thus providing bad locality. To improve upon this, we propose the use of dynamic schemes such as extendible [14] or linear hashing [24]. This makes the data structure more cache conscious and adaptive, and therefore better suited for the “worst case” of a schema containing many compressed attributes that would accumulate the static overhead of the fixed size hash-tables.

In our implementation we use a form of extendible hashing to grow the hash table as new entries are added. Only the first $k$-bits of the bucket index are used as an index into the hash table of size $2^k$. A simple heuristic can be used to decide when to increase the size of the hash table. On such resize, key-value pairs are redistributed from $2^k$ to $2^{k+1}$ buckets. After resizing, each bucket will contain on average half as many entries as before.

When a resize is triggered is currently determined by the size of a bucket. When a bucket’s number of entries grows above a certain number, resize is triggered.

Hash Functions

We did some preliminary evaluations of hash functions, but could not find any significant differences in distribution of our test data into the different buckets. We therefore went with a simple yet efficient hash function called $djh$, first posted on the comp.lang.c newsgroup. More extensive comparisons may be performed in the future if tests show that this hash function will create a degenerated data structure for some real-world data.
Under worst conditions, if data was generated specifically targeted to exploit this hash function, the dictionary could be reduced to a maximum capacity of only 64k values (the size of a bucket).

**Reference Counting**

We provide two variations of the bidi-map: A reference counting one and a non-reference counting one. The reference counting one stores a 32bit reference count with each key-value-pair and features methods to increase and decrease reference counts. When a reference count becomes 0 however, no data is removed until garbage collection is triggered.

The non-reference counting bidi-map uses a bit of the key’s intra-bucket index (thus reducing the maximum number of values in a bucket to $2^{15}$) to mark whether a value is in use in the segment or not. This implementation also provides methods to set those bits and a method to remove values that have not been marked.

**Buckets**

Our proposed implementations uses exact-fit arrays to store the values, but any data structure can be used.

In the current implementation, pairs of values and intra-bucket indexes are stored consecutively in the exact-fit array bucket. To not waste space, the values are stored in variable length, using a 1-byte char indicating their length.

**4.4.3 The Linear-trie**

We implemented a trie data structure similar to the HAT-trie as presented in [7]. The HAT-trie improves upon a standard trie in two areas: it is more cache conscious and it is more space efficient. This is achieved by having data stored linearly inside simple exact-fit array buckets. A heuristic decides, when such linear bucket is split (Figure 4.8) into several smaller buckets that are attached to a standard trie node. In our implementation such a split happens when there are a certain number of elements in the bucket.

A linear bucket contains the common prefix of all strings inside the buckets and the remaining characters (suffix) of a value combined with its key. The
prefix is needed so the value can be reconstructed for decompression. All buckets are linked with each other forming a linked list. This is used for linearly querying for keys on decompression.

A trie node in our current implementation is always of fixed size, containing the full range of 256 bytes. This could be optimized in future implementations by having different trie nodes depending on the value range or by using a compact list of character-pointer pairs.

**Compression**

Starting from the root node, the trie nodes are traversed until a linear bucket is reached. It is then iterate through the buckets key-suffix pairs till the value-suffix is found. The corresponding key is returned as the result of compression.

**Decompression**

Starting from the first linear bucket, the linked list of buckets is traversed. Each bucket is queried for the key until that key is found. The corresponding value is reconstructed from the value suffix and the bucket-prefix and returned as the result of decompression.
Insertion

Starting from the root node, the trie nodes are traversed until a linear bucket is reached. The value suffix is then inserted together with the key. If the heuristic for node-split is met, the linear bucket is split as depicted in Figure 4.8.

4.5 Dictionary Candidates

To evaluate compression we provide the following dictionary candidate implementations (for an overview see Table 4.1). We focused our attention on implementations based on hash tables and tries, as explained in Subsection 4.4.1.

For all hash table implementations we use exact-fit arrays for buckets, as presented in [8]. Such arrays only occupy the exact amount of space needed to hold the elements in a hash-bucket. The hash table itself can be dynamic or static (fixed-size). A dynamic hash-table increases in size as more items are added (e.g. using extendible hashing) to the dictionary whereas a fixed-size one remains the same, initial size (generally 64k buckets).

As we mainly evaluate tries for space reasons, we only implemented dictionaries using tries based on a single data structure for compression and decompression. If space savings are considerable enough and decompression performance turns out to be infeasible, a trie could in the future be combined with a dedicated decompression data structure.

All implementations provide the same interface, as specified in Subsection 3.2.6.

4.5.1 Two: Implementation Using Two Independent Hash-Tables With Reference Counting

This is a dictionary that uses two fixed size hash maps of 64k buckets each, as seen in Figure 4.9. One hash map, used for compression, stores the values and the associated keys consecutively in exact-fit array buckets. The other hash map, used for decompression, stores the key, a pointer to the value stored in the other map and a reference counter, also consecutively in an exact-fit array bucket.

The reference counter is stored in the map for decompression, as the performance critical reference-counter updates during the execution phase are by key and not by value. We further store a pointer to the value instead of the
Dictionary Design & Data Structures - Dictionary Candidates

<table>
<thead>
<tr>
<th>Dict.</th>
<th>Ref. counting</th>
<th>Fixed overhead</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two</td>
<td>Yes</td>
<td>Yes</td>
<td>Size of hash-maps</td>
</tr>
<tr>
<td>Bidi1</td>
<td>Yes</td>
<td>Yes</td>
<td>Size of hash-map</td>
</tr>
<tr>
<td>Bidi2</td>
<td>No</td>
<td>Yes</td>
<td>Size of hash-map</td>
</tr>
<tr>
<td>Bidi3</td>
<td>No</td>
<td>No</td>
<td>No. of entries for resize</td>
</tr>
<tr>
<td>Trie1</td>
<td>No</td>
<td>No</td>
<td>Size of hash map, no. of entries for split</td>
</tr>
<tr>
<td>Trie2</td>
<td>No</td>
<td>Yes</td>
<td>Size of hash map, no. of entries for split</td>
</tr>
</tbody>
</table>

Table 4.1: Dictionary candidates overview

actual value itself in the decompression-map. We argue that this is the more predictable solution, given that the pointer always occupies a fixed space, even though it may be sub-optimal for strings smaller than the pointer-size. Additionally, having fixed size structure makes for faster access due to fixed-size offsets, which is again the most crucial for decompression.

There is no garbage collection heuristics in this implementation.

4.5.2 Bidi1: Dictionary Using Reference-Counting Bidi-map

This is a dictionary that uses a fixed-size bidi-map to store the data. Dictionary interface methods map to the equivalent method in the bidi-map. The bidi-map contains reference counters.

Garbage collection is performed based on a simple heuristic: As soon as a specific percentage of reference counters have become 0, garbage collection is triggered and the dictionary reorganized. Garbage collection is performed by iterating through the whole bidi-map, allocating new buckets for each bucket and copying the data into those new buckets, omitting values with reference count of 0.
4.5.3 Bidi2: Dictionary Using Non-Reference-Counting Bidi-map

This is a dictionary that uses a fixed size bidi-map to store the data. Dictionary interface methods map to the equivalent method in the bidi-map. The bidi-map does not contain reference counters.

There are currently no heuristics for garbage collections, those will have to be evaluated.

4.5.4 Bidi3: Dictionary Using Extendible Non-Reference-Counting Bidi-map

This is a dictionary that uses a dynamic bidi-map (using extendible hashing) to store the data. Dictionary interface methods map to the equivalent method in the bidi-map. The bidi-map does not contain reference counters.

There are currently no heuristics for garbage collections, those will have to be evaluated.
4.5.5 Trie1: Dictionary Using Linear-trie

This is a dictionary that uses a linear-trie to store its data. Compression and decompression are both handled by the linear-trie data structure. Evaluation will show, if the linear list of buckets can provide fast enough decompression performance.

The linear-trie dictionary does not contain reference counters. There are currently no heuristics for garbage collections, those will have to be evaluated.

4.5.6 Trie2: Dictionary Using Bidi-trie-map

This is a dictionary that uses linear-tries instead of exact-fit arrays as buckets in a fixed-size bidi-map. This is motivated by the fact that decompression performance of tries may be insufficient on large data sets. Using them as buckets in a hash-table reduces the linear search to at most 64k entries. However, the hash-table should not consist of too many slots, but instead focus on fewer buckets with many values, so the advantages from the trie buckets can be used and it does not just degenerate in a linear-bucket. The later is also influenced by the choice of bucket-split. Ideally, this would happen in combination with an extendible bidi-map, however we wanted to be able to tune the bidi-map size statically first.

This data structure is less efficient in regard to prefix-lookups than the normal trie.

The bidi-trie-map does not contain reference counters.
In this chapter we evaluate the work from the previous two chapters on various workloads and data sets using benchmarks in the large and in the small. We first derive the main questions that have to be answered to provide insights into the benefits of compression and the differences between the dictionary candidates presented in the last chapter. Using the benchmarking methods presented after that, we generated results that are then used to answer our questions.

5.1 Questions

As described in Chapter 3, adding compression to Crescando resulted in many code changes, especially in the core of Crescando’s execution engine, the scan thread. Therefore we want to evaluate if the new system with support for compression performs worse on uncompressed workloads than the old system. Consequently our first question is:

**Question 1:** Does supporting compression impact performance of uncompressed workloads?

Even though we aimed at adding compression in a way that the uncompressed case is affected as little as possible (as part of our design principles), adding new functionality often impacts performance. Such impacts are acceptable,
Evaluation - Questions

as long as they are outbalanced by the added benefits. Yet, before we start evaluating those benefits by investigating the performance differences between compressed and uncompressed, we first would like to find out which of our dictionary candidates, as presented in Section 4.5, performs best:

**Question 2:** Which dictionary candidate is the best?

This is a question which needs different dimensions to be taken into consideration. Data structures may trade space savings for performance efficiency and vice versa. They can also have different performance characteristics for insertion, compression and decompression. We therefore evaluate data structures on all those dimensions using an isolated micro-benchmark and intend to find a clear winner overall, or at least rule out some definite losers that do not have to be considered for further testing.

The declaration of a winning data structure should however not only rely on micro benchmark results alone, but the individual dictionary candidates must also be evaluated integrated into the whole Crescando system, to take into account caching and other influences. This leads us to our next question:

**Question 3:** Does the dictionary implementation matter “in the real world”?

When we have answers to above questions, we are ready to evaluate our main one:

**Question 4:** Does compression pay off?

As Question 2, this is a question that can not be answered by yes or no without exception. There are again many dimensions the answer may depend on. Compression may pay off in regard to space savings, but may introduce performance penalties. Or it could be the other way round: Space savings are minor, but performance can be improved due to predicates evaluated directly on keys. The gains in time and space can be dependent on the distribution of values and the range of the domain (Subsection 4.2.1). And they can be different for read only workloads than for update-heavy ones. We intend to be able to give a clear answer, given those dimensions, under which circumstances compression pays off.

Apart from the rather hypothetical question, if compression pays off and under which circumstance, we are also interested in knowing if those payoffs can be made to use in terms of saving of hardware. Or formulated slightly differently:

**Question 5:** Can compression reduce hardware costs?
Evaluation - Approach

5.1.1 Summary

1. Does supporting compression impact performance of uncompressed workloads?
2. Which dictionary candidate is the best?
   • in regard to size?
   • in regard to insertion performance?
   • in regard to compression performance?
   • in regard to decompression performance?
   • overall?
3. Does the dictionary implementation matter “in the real world”?
4. Does compression pay off?
   • in regard to size?
   • in regard to performance?
   • under read only / update workloads?
   • overall?
5. Can compression reduce hardware costs?

5.2 Approach

To answer above questions we used two kinds of benchmarks: a Java-based Crescando benchmark developed in [17] and a compression micro benchmark developed as part of this thesis. For all system benchmarks we used the Itinerary schema provided by Amadeus (4.2.1).

5.2.1 Test System

All benchmarks were run on the same 16-way machine built from 4 quad-core AMD Opteron 8354 (“Barcelona”) processors with 32 GB of DDR2 667 RAM. Each core has a 2.2 GHz clock frequency, 64 KB + 64 KB (data + instruction) L1 cache, and 512 KB L2 cache. The machine was running a 64-bit Linux SMP kernel, version 2.6.31. For all test, we run only 1 scan thread and not a full scale system-benchmark, as compression is implemented transparently to higher levels of the system.
5.2.2 Crescando Benchmark

The Crescando benchmark was developed as part of [17]. It is a Java-based benchmark that was mainly created to be used in conjunction with the Amadeus flight-data workload based on the Itinerary schema presented in 4.2.1, but it can also be used on other schema. The benchmark allows to be extended by custom workloads, which can be implemented in the form of Java-classes. We provide a few such workloads to test various aspects of compression.

The benchmark has an initial bulk loading phase, in which a segment is filled with data to a specified percentage. The distribution of values is loaded from plain-text files for each individual attribute or generated randomly if the attribute value is unique. After that initial bulk loading phase the benchmark creates a workload based on the specified workload-generator class. This workload is applied during the benchmarks execution at a specified rate of throughput for a set amount of time.

![Figure 4.1: Amadeus Workload: Predicated Attributes in select operations (taken from [17])](image-url)
Evaluation - Approach

Workloads

The following shows a list of workloads used for testing. The *Amadeus workload* was created in [17], whereas the other workloads were implemented as part of this thesis.

**Amadeus Workload**  This is a workload based on the real-world data provided by Amadeus. The average select has 8.5 distinct predicate attributes and projects 27 of the 47 attributes in the schema. 99.5% of the selects contain equality predicates on *flightNumber* and *departureDate* and 99.7% select on *productId* and *provider*. The detailed predicate-distribution is illustrated in Figure 5.1.

More detailed information can again be found in [17], Chapter 2 and Chapter 4.

This workload is meant to measure real-world performance.

**Equality Workload**  This is a workload that creates selects containing an equality predicate on the attribute *name*. The predicate values are randomly picked from the attribute’s domain according to the same distribution exhibited by the data bulk-loaded into the table. Using aggregation ensures that no result records have to be fetched.

This workload is meant to measure performance of equality predicates on compressible string attributes. It can also be used to measure index lookup performance.

**Range Workload**  This workload creates selects containing two range predicates on the attribute *name*. The predicate values are randomly picked from the attribute’s domain according to the same distribution exhibited by the data in the segment. An additional predicate on *cityFrom* with a fixed predicate value ensures that only 1% of the data is touched, otherwise such range predicates are not feasible neither in compressed nor uncompressed schema. Using aggregation ensures that no result records have to be fetched.

This workload is meant to measure performance of prefix predicates (translated to two range predicates) on compressible string attributes. As ranges on string are not indexed, it can also be used to measure non-indexed predicate-evaluation performance.
Evaluation - Approach

Result Creation Workload This workload creates selects that match 500 records on average. This is achieved by creating predicates on uncompressed attributes from and to and productid. The predicate values are randomly picked from the attribute’s domain according to the same distribution exhibited by the data in the segment. Each select contains projections on all compressed attributes.

This workload is meant to measure performance of result creation.

Update Workload This workload creates update operations that update on average of 500 records per operation. This is achieved by creating predicates on uncompressed attributes from and to and productid. The predicate values are randomly picked from the attribute’s domain according to the same distribution exhibited by the data in the segment. Each update operation updates all compressed attributes to new random values.

This workload is meant to measure performance of update operations on compressible string attributes.

Metrics

The following metrics are provided by the Crescando benchmark and the Crescando engine respectively:

- The 50, 90 and 99 percentiles of the systems latency.
- The sizes and number of unique values of compression dictionaries per attribute.

Benchmarking Setup

All benchmarks were performed on the Amadeus Itinerary schema, with segment sizes of 1gb for the uncompressed schema and 626mb for the compressed schema. The segments were filled with 3.3 million records.

In the compressed case the attributes office, firstname, lastname and sgtVendorValue were compressed (4.2.1). The respective dictionaries contain 37’000, 450’000, 397’000 and 27’000 unique values.

An individual record is 313 bytes in uncompressed an 192 bytes in compressed form.
5.2.3 Micro Benchmark

We developed a small micro benchmark in C++ to measure performance of basic dictionary operations. The benchmark program takes two parameters: the data file to be read and the maximum string length.

The benchmark operates as follows: the data file provided is loaded line by line into an array in memory. The values in this array are used to benchmark the three main dictionary operations: insertion, decompression and compression. Therefore, the benchmark consists of three phases. Before each phase, the input arrays are permuted to prevent caching effects.

During the first phase, dictionary values from the values-array are inserted into the dictionary and the generated keys are added to a key-array. During the second phase we query the dictionary for all values in the values-array and in the third and last phase we query the dictionary for all keys in the keys-array.

Metrics

The benchmark provides 4 metrics: time taken to insert values into the dictionary, time taken for compression and time taken for decompression, and the size of the dictionary.

Data Sets

We derived six data sets to be used in the micro benchmark. Those data sets are defined by the number of total values, the number of unique values, the value distribution and the average and maximum value lengths. Those are important to consider for reasons given in Subsection 4.2.1.

A: Amadeus First Names   This is a subset of first names from the ticketing data provided by Amadeus (4.2.1). The data set consists of 10 million values, 370’000 of which are unique. The average length of a value is 10.51 and the maximum is 75. The value distribution of the full set is illustrated in Figure 4.1.

S: Swiss Phone Book Names   This data set contains swiss surnames (including maiden names and titles) from the Swiss phone book. The data set consists of 3 million values, 1.1 of which are unique. The average length of a
value is 10.51 and the maximum is 56. The value distribution of the full set is illustrated in Figure 4.5.

**N1-N4: Normally Distributed Values** Data sets N1-N4 each consist of 50 million normally distributed values with 1000, 10'000, 100'000 and 1 million unique values respectively. The standard deviation was set to one sixth of the number of unique values, so there are less high frequency values as in the other two data sets. The maximum length of a value is 20 and the average is for each set close to 10, as the individual value lengths were generated from an uniform distribution. This was again done to have data different from the other sets, where the average length is much smaller than the total length.

### 5.2.4 Detailed Approach

For each workload in the Crescando benchmark we evaluated the maximum possible throughput for that workload under compressed and uncompressed workloads. Based on this maximum throughput we derived data points representing high, medium, low load and very low load. Very low load was set to 10 operations, low load to 100 and medium load to around half the maximum load. We measured the latency of the system under those loads. Where needed, we added further data points to see specific changes during load increase.

The benchmarks were run 5 times for each load. We plot the median over those 5 runs of the individual runs 90 percentile latency. Where the measurement error is significant, the plots also contain error bars based on the standard deviation over the 5 runs. If we provide percentages in the form of “the latency for compressed is X% higher than for uncompressed”, those percentages are based on the average of the 90 percentile latencies over the 5 runs (as opposed to the median used in the plots).

All benchmarks were run on the Amadeus itinerary schema.

### 5.3 Results

In the following we present answers to the questions asked in the beginning of this chapter. For each question we present the individual approach to answer that specific question, present and comment the results and summarize our findings.
5.3.1 Question 1: Does supporting compression impact performance of uncompressed workloads?

Approach

To find out whether compression introduces an overhead, we run the Crescendo benchmark with the following workloads: Range Workload, Result Creation Workload and Amadeus Workload. If there is an overhead introduced in our real world workload (Amadeus Workload), then the other workloads help us to analyze if it stems from result creation (Result Creation Workload) or predicate evaluation (Range Workload). We additionally tested using the update-only workload (Update Workload), to evaluate if the addition of a new virtual method in the update logic impacts performance.

Results

![Figure 5.2: Results of Amadeus Workload](image)

From the results of the Amadeus Workload (Figure 5.2), we can see that supporting compression does not affect its performance significantly. The average latency of the code without compression differs at most 2% from the new code with compression support. The same is true for the update workload Update Workload (Figure 5.3).

Looking at results from the Range Workload (Figure 5.4) and Result Creation...
Evaluation - Results

![Write Latency and Throughput](image1)

**Figure 5.3:** Results of Update Workload

![Select Latency and Throughput](image2)

**Figure 5.4:** Results of Range Workload

Workload (Figure 5.5), it is evident that under extreme conditions the added complexity to predicate evaluation and result creation can lead to increased latency. The predicates in Range Workload are not indexed, and the un-index path of predicate evaluation has been extended by an additional branch to support dictionary lookup predicates 3.3.3. Due to this, we can experience up to 18% increased latency under those extreme conditions. The impacts for result creation are less severe with only up to 9% increase latency.
Evaluation - Results

Figure 5.5: Results of Result Creation Workload

Conclusion

Under extreme conditions, where no predicate can be indexed, supporting compression can result in up to 18% increase in latency. Additionally, there is an up to 8% increase in latency due to result creation. Both however, do not translate in significantly increased latency for the real-world Amadeus workload.

5.3.2 Question 2: Which dictionary candidate is the best?

Approach

To select some winning data structures we run our micro benchmark using all the dictionary candidates specified in Section 4.5 on all the data sets presented in the last section (5.2.3).

If we can identify dictionary candidates that are outperformed by other implementations in all dimensions, we eliminate them from further benchmarks.

Parameter Tuning

Many of the dictionary candidates have parameters, such as size of the hash table or number of elements in a bucket that trigger reorganization. We tuned
Evaluation - Results

those parameters using the micro benchmarks and other benchmarking tests beforehand; where no clear winning parameters could be determined, the candidates are presented here in several parameter combinations. The size of the hash table in dictionary candidates using fixed size hash-tables (Two, Bidi1 and Bidi2) was always set to the maximum of 64k entries. The suitable size for a given data set can be read from the size from extendible bidi-map based dictionary (Bidi3). The parameter for the resize-heuristic of Bidi3 was determined to be set to 80 entries, i.e. resize is triggered when a bucket contains more than 80 entries.

We tested the linear trie candidate Trie1 with split points of 2000 and 8000. We further tested the bidi-trie map (Trie2) with hash-table sizes of 16k and 1k entries. We did not test the Trie2 with split points larger than 512, as then the trie would degenerate into a linear bucket.

Results

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>S</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two</td>
<td>232%</td>
<td>195%</td>
<td>6556%</td>
<td>1118%</td>
<td>416%</td>
<td>219%</td>
</tr>
<tr>
<td>Bidi1</td>
<td>151%</td>
<td>126%</td>
<td>5172%</td>
<td>770%</td>
<td>243%</td>
<td>135%</td>
</tr>
<tr>
<td>Bidi2</td>
<td>122%</td>
<td>105%</td>
<td>5107%</td>
<td>713%</td>
<td>197%</td>
<td>107%</td>
</tr>
<tr>
<td>Bidi3</td>
<td>100%</td>
<td>100%</td>
<td>180%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Trie2, 16k, 512</td>
<td>207%</td>
<td>302%</td>
<td>4237%</td>
<td>678%</td>
<td>196%</td>
<td>426%</td>
</tr>
<tr>
<td>Trie2, 1k, 512</td>
<td>244%</td>
<td>316%</td>
<td>416%</td>
<td>130%</td>
<td>301%</td>
<td>383%</td>
</tr>
<tr>
<td>Trie1,8000</td>
<td>127%</td>
<td>130%</td>
<td>101%</td>
<td>101%</td>
<td>119%</td>
<td>122%</td>
</tr>
<tr>
<td>Trie1,2000</td>
<td>192%</td>
<td>229%</td>
<td>100%</td>
<td>135%</td>
<td>239%</td>
<td>122%</td>
</tr>
</tbody>
</table>

Table 5.1: Relative dictionary sizes on all datasets (5.2.3) and dictionary candidates (Section 4.5)

The are a few things we can read from the relative sizes of the dictionary candidates listed in Table 5.1. First, for every data set except one, the dictionary implementation Bidi3 proves to take up least space. The only exception is N1. Here the overhead induced by the hash table results in a slightly larger dictionary than Trie1. This is due both split points (2000 and 8000) being below the number of unique values and the linear bucket of the trie never being split. On all other data sets, Trie1 does not provide smaller sizes than Bidi3 as we anticipated.

On the other end of the scale, there is Two, which proves to be the largest
Evaluation - Results

implementation on most of the data sets. It is generally twice the size of the comparable bidi-map implementation \textit{Bidi1}. Only on \textit{S} and \textit{N4} \textit{Two} is exceeded by \textit{Trie2} with a 16k hash-table. The same implementation with a 1k hash-table performs considerably better. This indicates that the memory overhead induced by trie nodes considerably reduces the benefits from the tries ability to merge common prefixes.

Comparing the bidi-map based implementations against each other, we can see that the reference counters in \textit{Bidi1} induce an overhead of around 20\% compared to the equivalent implementation without reference counters (\textit{Bidi2}). The relative overhead of the fixed size hash-table in \textit{Bidi2} compared to the extendible \textit{Bidi3} becomes smaller with increasing number of unique values, but is still considerable for 100\,000 (97\%, \textit{N1}) and 400\,000 values (22\%, \textit{A}).

![Figure 5.6: Insertion performance (seconds) of the different dictionary candidates on all data sets](image)

The picture becomes less clear in regard to performance. \textit{Two} performs worse in regard to insertion performance (Figure 5.6) than all three bidi-maps. This was to be expected, as values in \textit{Two} have to be added to two data structures. The extendible bidi-map \textit{Bidi3} is able to outperform the fixed size bidi-map \textit{Bidi2} on some data sets, despite the overhead of value-redistribution on resize. We attribute this to better cache-locality due to its smaller size. The reference counters that have to be maintained in \textit{Bidi1} seem to induce a slight overhead compared to the non-reference counting version \textit{Bidi2}.

Overall, all hash-table based implementations perform relatively predictable: the more data is inserted, the longer insertion takes. The tries seem to be much more sensitive to the actual data. Depending on the values inserted, they may perform more costly bucket-split operations or generate large linear buckets.
Evaluation - Results

If a bucket is split too often, insertion performance drops, but compression performance is not affected. If a bucket is split too late, insertion performance and compression performance are reduced, as a large linear bucket has to be scanned to fetch a value. The spike that can be seen for data set $S$ and $Trie1$ with a bucket-split parameter of 2000 is likely caused by the bucket being split too often. The other spike on data set $N2$ for $Trie1$ with a bucket-split parameter of 8000, however, is very likely caused by splitting too late. This can be seen by looking at the compression performances of the above data points in Figure 5.7. Those results indicate, that there should be a more sophisticated heuristic determining when to split a linear bucket.

Figure 5.7: Compression performance (seconds) of the different dictionary candidates on all data sets

Further investigating the differences between the parameters for $Trie1$, a bucket-split point of 2000 values seems to perform better in regard to compression performance than a split point of 8000.

The bidi-maps perform relatively well on compression (Figure 5.7), with again a slight visible overhead for the reference-counting implementation $Bidi1$ compared to its non-reference counting counterpart $Bidi2$. The difference between the extendible bid-map $Bidi3$ and the fixed-size one $Bidi2$ observed before also apply here: for fewer unique values $Bidi3$ performs better whereas the fixed size version is superior when the number of unique values increases. We attribute the higher performance of $Bidi3$ on fewer unique values to better cache locality. The overhead of $Two$ compared to the bidi-maps is much smaller than for insertion, as only one of the two hash-maps has to be touched.

When it comes to decompression performance (Figure 5.8) the fixed-size bidi-map $Bidi2$ is faster for all data sets than its extendible counterpart $Bidi3$. Here
Evaluation - Results

The locality benefits from above seem not to apply. This is likely due to the set of keys that are decompressed being much smaller than the set of strings used for compression, making cache locality less important. As for insertion and compression, the reference counters stored in Bidi1 slightly impact performance compared to the other bidi maps. Similarly, Two is again slower than the equivalent, fixed-size bidi-map.

We completely omitted all trie-based implementations from Figure 5.8, as they took above 300 seconds to decompress data. This had to be expected, as they are optimized for compression and decompression is done by scanning the buckets linearly. Our main purpose for evaluating tries was to find out if they can provide smaller dictionary size.

The bidi-trie-map implementation Trie2 with a hash-table size of 64k and a bucket-split size of 512 did not perform as bad as the purely trie-based implementations, but still significantly worse than Bidi3, which partitions the data in larger linear buckets than the trie-map.

Conclusion

Two and Trie2 are outperformed in space and time in nearly every data set by the bidi-map implementations. They will therefore not be included in further
Evaluation - Results

testing. Tries perform, as expected, very poorly in regard to decompression performance and lead to no gain in regard to compression performance or space savings. They in general behave in unpredictable ways, sensitive to the input data, indicating that better bucket-split heuristics may be needed.

*Bidi3* proves to be the smallest dictionary implementation on 5 out of the 6 data sets. In regard to compression and decompression performance it is sometimes faster and sometimes slower than the fixed-size equivalent *Bidi2*, depending on the size of the data set, indicating that cache-locality seems to be playing a role. Reference counters in *Bidid1* induce a slight overhead not only in size, but also in regard to performance.

5.3.3 Question 3: Does the dictionary implementation matter “in the real world”?

**Approach**

To answer whether the dictionary implementation matters in the real world we compare the bidi-map based dictionary implementations against each other using the Crescando benchmark on all our workloads. *Two* and *Trie2* were omitted for reasons explained in the last question. *Trie1* was tested, but not depicted as the latency on some decompression-heavy workloads (*Result Creation Workload, Range Workload*) was over 2 minutes.

One of the main things to evaluate is whether caching effects observed in the micro benchmarks also play a role in the whole system (i.e. *Bidi3* can outperform *Bidi2* due to better locality).

**Results**

There are two aspects to this answer: does the implementation matter regarding size and does it matter regarding performance? If we look at the total size of the dictionaries over all four compressed attributes on our 626mb segment size using the Itinerary schema (*Figure 5.9*), we can declare the extendible bidi-map to be the most space-efficient for this schema. If one compares the total size consisting of the segment size plus the size of the dictionary however, the added size of even the largest dictionary is negligible compared to the total segment size. We would therefore argue, that the size of the dictionary does not really matter for our test data. This said though, for a schema containing a
significant amount of compressed attributes, the accumulated static overhead of the dictionary implementations using a fixed-size hash-table (Bidi1 & Bidi2) may result in a significant increases in total size. Furthermore, performance may be affected by such static overhead due to an increased working-set size and resulting worse cache-utilization.

In regard to performance the picture is similar. On all workloads the three bidi-maps are never more than 3% apart. We therefore only show the Amadeus Workload (Figure 5.10) representative for other workloads. The better locality of Bidi3 seems not to have any effect on the overall performance. This could be because the locality benefits are undone by the increased bucket, indicating the choice of smaller parameter for the resize-heuristic.

The only significant difference is measured under the Compressed Update Workload (Figure 5.11). Here the implementation using reference counting (Bidi1) needs to perform two dictionary operations for each update, whereas the non-reference counting implementations need not perform any operation. We hence see a linear overhead in the number of updates put into the system with Bidi1 compared to the other two dictionary implementations.
Evaluation - Results

Figure 5.10: Latency of bidi-maps on Amadeus Workload

Figure 5.11: Latency of bidi-maps on Update Workload

Conclusion

On our real-world data from Amadeus, the differences of dictionary sizes is negligible compared to the total segment size. The extendible-bidi map (Bidi3) however is most true to Crescando, as it adapts best to the data provided.

In regard to performance, the added overhead of reference counters is only
significant when it comes to update latency. The locality advantages of \textit{Bidi3} compared to \textit{Bidi2} observed in the micro benchmarks could not be replicated.

Tries exhibit the same infeasible decompression performance as observed in the micro benchmarks. They could again not provide better compression ratios than \textit{Bidi3}. Trie based implementations will therefore be excluded from further testing.

### 5.3.4 Question 4: Does compression pay off?

#### Approach

To answer this question we look at all of the available workloads and compare the latencies of the uncompressed with the compressed case. As shown in the last question, all bidi-map dictionary implementations perform nearly identical, except on update workloads. We therefore use the results of the fixed-size, non-reference counting bidi-map implementation \textit{Bidi2} representing the compressed case.

#### Results

![Bar chart showing results](image)

\textbf{Figure 5.12: Results of Amadeus Workload}

As explained, the answer to this question is very involved. First, we evaluate
Evaluation - Results

if compression pays off in regard to size. As listed in 5.2.2 the compressed segment is 626mb for the same amount of records as a 1gb uncompressed segment. The dictionary adds up to 20mb to that, depending on the specific dictionary implementation (Figure 5.9). Hence, using compression we can reduce the size of a 1gb uncompressed segment to around 650mb, thus saving 350mb of main memory per segment. This equals a relative saving of 35%. Even though this is nowhere near the 80% savings that are claimed for disk-based systems [3], we argue that a reducing the table by a third of its size is a significant payoff. Moreover, those published numbers are under ideal conditions, and the average payoff appear to be lower [13], especially for Oracle’s page-level compression.

Next, we study if compression pays off in regard to performance. Can latency be reduced or throughput be increased using compression? Looking at the Amadeus benchmark results (Figure 5.12), there is no significant difference in performance with compression enabled compared to the uncompressed case. Under low and average load uncompressed is up to 7% faster, whereas under high load compression seems to improve performance slightly by 4%. With increasing load performance becomes dependent on predicate-evaluation. The faster evaluation of firstname and name may therefore contribute to this gain. We attribute the performance overhead under average and low load to result-creation and preprocessing of operations. The result creation overhead can be up to 40% on average load, as depicted in Figure 5.13.

A much more extreme overhead is experienced for range predicates (Fig-
Evaluation - Results

Figure 5.14: Results of Range Workload

Figure 5.15: Results of Equality Workload

ure 5.14). This has several reasons, the most important one being the way range predicates are handled right now. As explained in Subsection 3.3.2, range predicates on compressed values are not indexed and decompression has to be performed for each predicate on such value of each select. The overhead experienced is therefore linear in the number of selects. Range predicates in this form are therefore not feasible at this point; improvements to change this are discussed in Subsection 6.1.5.
Evaluation - Results

Unlike with range predicates, compression pays off for equality predicates on compressed values. As visible in Figure 5.15 for the Equality Workload. The uncompressed overhead seems to grow linearly with the number of selects. After 500 selects per second, the uncompressed system is saturated while the one using compression scales up to 1500 selects per second. Compression in this case enables workloads that were not feasible without it. We attribute this to two facts: First, in the uncompressed case strings up to 56 characters have to be compared while in the compressed case this is only a 4 byte key. Second, the index generated for the string is much larger than the one for the keys. At 500 queries that index appears to be too large to fit into the L1 cache and performance drops severely to a point, where additional queries cannot be processed anymore.

![Figure 5.16: Results of Update Workload](image)

Finally, we can conclude from Figure 5.16 that compression does not impact update performance. This is under the assumption that the dictionary implementation is a non-reference counting one. The added overhead of reference counting for updates was discussed before and is depicted in Figure 5.11.

Conclusion

Compression provides 35% of space savings on the Amadeus schema.

Performance on the Amadeus workload is not severely impacted and slightly improves on high load by using compression. The impact likely stems from...
result creation that can impact performance of up to 40% in extreme cases. This can however be optimized in the future. The gains under high load likely stem from predicates on compressed values. In general, compression can significantly improve performance when predicates are evaluated directly on keys. In this case compression can enable workloads that were not feasible without it.

Range predicates in their current form are infeasible, but there are also ways to improve upon that in the future (Subsection 6.1.5).

Update performance is not impacted by compression, as long as the dictionary does not use reference counters.

### 5.3.5 Question 5: Can compression reduce hardware costs?

**Approach**

We use previous results on compression ratios to evaluate reductions in memory cost. We furthermore run the *Amadeus Workload* and the *Compression Equality Workload* on a compressed 1gb segment. If performance on such segment is comparable to the uncompressed case, the system could run using not only less memory, but also using less CPUs.

**Results**

As show before, using compression we can reduce the size of a 1gb uncompressed segment to around 650mb, thus saving 350mb of main memory per segment. On our 16x test-machine this would result in a total of 5.6gb of memory being saved. Those savings however can not that easily be translated into savings of hardware. Due to the demands of NUMA, it may not be possible to translate those theoretical savings in less RAM that has to be installed into the machine. It would therefore be desirable to use the same amount of RAM per segment, but less CPUs, and in the larger scale, less machines. The results depicted in Figure 5.17 indicate that this is not feasible for the *Amadeus Workload* on our current hardware. The latency is severely worse than in the uncompressed case. This indicates that on this workload, Crescando is bandwidth and not CPU-bound.

Since compression can also reduce CPU-time, as is evident from Figure 5.15, the goal of reducing the number of CPUs to run a Crescando system can be
Evaluation - Results

![Graph showing results](image)

**Figure 5.17:** Results on full 1gb segment using Amadeus Workload

![Graph showing results](image)

**Figure 5.18:** Results on full 1gb segment using Compression Equality Workload

reached for specific workloads. This can be seen from the results of running the Equality Workload on the 1gb segment depicted in Figure 5.18. Even though the compressed case is still slightly faster for low and average workloads, it scales well with increased throughput (up to over 1000 queries) exhibiting constant overhead, whereas the uncompressed case does not. Furthermore, one can find a sweet spot where compression pays off well in regard to performance and space by using a segment size between the tested 626mb (Figure 5.12,
Evaluation - Results

Figure 5.15) and 1gb (Figure 5.17, Figure 5.18).

Conclusion

RAM may be saved by enabling compression. CPU savings on current hardware under our real-world workload may not possible, due to the system being CPU-bound on that workload. For workloads containing a significant amount of equality predicates on compressed attributes, those savings can be achieved. A definite answer requires more detailed benchmarking on a full system using several CPUs.
Conclusions

As a result of this thesis, we added support for dictionary compression to the Crescando system. Compression has been integrated transparently to the higher levels of the systems and with a focus on maintaining Crescando’s predictability and scalability guarantees. This was made possible by adding compression dictionaries on the segment level and compressing on the attribute-level. The properties of Crescando’s scan-based architecture and attribute-level compression could be used to reduce dictionary access during scan phase has been kept to a minimum by preprocessing operations and evaluating predicates directly on the compressed values.

Several implementations for the compression dictionary were evaluated. It was shown that the bidi-map, developed as part of this thesis, performs better overall than other implementations in regard to space and time. The evaluated trie-based implementations could not provide the space-savings we anticipated.

We showed that dictionary compression in a scan-based, in-memory system such as Crescando can pay off in regard to space savings. In the case of our real-world data from Amadeus, we identified 4 attributes as benefiting from dictionary compression. By applying compression to those attributes, space-savings of 35% are achieved on a segment size of 1gb.

The picture was less clear in regard to performance. Under a workload that features mainly equality predicates on compressed values, the performance using compression can increase significantly (up to 300%) from the uncompressed
Conclusions - Future Work

case. In this case compression enables workloads that were not feasible without it.

On a result-heavy workload, where a lot of data has to be decompressed, there was a measurable decrease of performance when enabling compression. We could measure impacts of up to 40% compared to the uncompressed case, the average impact however were considerably smaller. On our real-world workload based on the Amadeus Itinerary ticketing data, we did not measure significant difference running with or without compression. Similarly, compression did not affect update performance, as long as a non-reference counting dictionary implementation is used.

Range predicates in the current implementation can slow the system down severely (at a throughput of 200 selects/second the latency is around 10 seconds). We argue however, that this can be overcome by appropriate indexing structures.

From above results we conclude that compression in a scan-based, in-memory system does not provide the same performance benefits due to reduced I/O as it does for disk-based systems. Compression may only increase performance, when predicates can directly be evaluated on keys. This suggests that the attribute-level compression, commonly used in column-stores, is also a fitting choice for a scan-based, in-memory row-store.

6.1 Future Work

6.1.1 Dictionary Serialization for Recovery

The recovery schema currently in used has an overhead of up to 100% compared to the uncompressed case. In future versions, one should therefore serialize the dictionary and store the records in compressed form on disk. This may turn the current performance impact into performance savings compared to the uncompressed case, due to reduces disk-I/O to serialize the smaller, compressed records. This will have to be evaluated.

6.1.2 Custom Memory Management for Dictionaries

Right now, the dictionaries data is allocated anywhere in the RAM using standard malloc and realloc facilities. Preferably, the dictionary would be on the
same memory region closest to the core as the segment. For this purpose custom memory management for the dictionary shall be evaluated in the future.

6.1.3 Key Sizes Other Than 32bit

The system right now is constrained to keys of 32bit sizes. Even though the dictionary candidates have been implemented in a generic way, also allowing key sizes other than 32, adding variable key sizes is not as easy to be implemented in the rest of the Crescando system. Therefore, undergoing such an effort should be preceded by tests and calculations, if smaller key sizes will result in much gain in performance and/or space savings. Furthermore, the bidi-map in its form is not suitable for 16bit keys (a simple hash-map may be more appropriate) and would have to be replaced by a different data structure.

6.1.4 Reference Counting Evaluation

We could not answer which presented garbage collection scheme is better: reference counting or a mark-and-sweep. This has been due to the reference counters not being used to their full potential and due to the fact that non-reference counting dictionaries reorganization and fragmentation overhead could not be measured as part of this thesis.

Reference counters can be used to optimize a data structure to provide shorter access path for more frequent values. Such optimization could for instance be implemented in the bidi-map by sorting the key-value pairs in a bucket according to the reference counts. This would result in values with higher reference counts to be at the beginning of a linear search. The gain of this optimization would have to be evaluated on different kind of data-distributions.

A reference counting dictionary can further be seen as a histogram on an attribute domain. Such histogram could be used during optimization to better decide upon the choice of query plan. It should be evaluated what can be gained by using such information.

Similarly, one could do experiments with move-to-front or moving a key-value pair one slot forward in a non-reference counting bidi-map bucket. The order of values generated this way could also be used as an estimate for the frequencies. It should be evaluated if such scheme is comparable to the exact information provided by reference-counters.
Conclusions - Future Work

Finally, the overhead of a dictionary select, used for garbage-collection of non-reference counting dictionaries, has to be evaluated and different heuristics for starting garbage collection should be tested.

Doing above experiments and using the reference counters for optimization should give a much more definite answer, if maintaining them pays off overall.

6.1.5 Range Indexing

In the current implementation, range predicates are evaluated by decompressing during the execution phase for each predicate individually. We propose two other approaches that could be studied in the future.

One approach is to preprocess range predicates using the dictionary and gather a set of matching keys for a range in the activation phase. During execution, the attribute value of a record would then be compared to the set of keys to determine if its in the range or not.

The previous approach works for any kind of ranges. We observe however, that the only use case for ranges on string attributes are prefix and suffix searches. We can support those kind of predicates efficiently by indexing predicates in a prefix or suffix trie. This index would then be probed using the decompressed value during executing. As there is still decompression needed for this scheme, the dictionary has to be more cash-conscious than the current implementation.
A

Appendix

A.1 Amadeus Itinerary Schema

CREATE TABLE Itinerary (  
  provider VARCHAR(3),  
  productId UINT(16),  
  alphaSuffix CHAR,  
  dateInFirstLeg DATE,  
  dateIn DATE,  
  dateOut DATE,  
  cityFrom VARCHAR(3),  
  cityTo VARCHAR(3),  
  cancelEnvelope INT(32),  
  cancelInitiator VARCHAR(3),  
  rloc VARCHAR(6),  
  paxTattoo INT(32),  
  segmentTattoo INT(32),  
  purgeDate DATE,  
  office VARCHAR(9) COMPRESS(32),  
  creationDate DATE,  
  modificationDate DATE,  
  nip INT(16),  
  unassigned INT(16),  
)
pnrQualifier UINT(64),
paxQualifier UINT(64),
sgtQualifier UINT(64),
name VARCHAR(57) COMPRESS(32),
firstname VARCHAR(56) COMPRESS(32),
sex BOOLEAN,
cabin CHAR,
classOfService CHAR,
bookingStatus VARCHAR(2),
codeShareType VARCHAR(2),
bookingDate DATE,
subclass UINT(8),
posCrs VARCHAR(3),
posCountry VARCHAR(2),
cancelFlag CHAR,
mariage VARCHAR(3),
yieldValue INT(32),
rvIndicator CHAR,
rvValue INT(32),
cnxNumber INT(8),
did VARCHAR(16),
id VARCHAR(16),
indexingVersion INT(16),
sgtVendorFormat VARCHAR(2),
sgtVendorValues VARCHAR(15) COMPRESS(32),
inboundCnxTime INT(32),
inboundSgtTattoo INT(32),
outboundCnxTime INT(32),
outboundSgtTattoo INT(32)
);
Bibliography


