University of Zurich[UZH]

Department of Informatics

Robust and Scalable Content-and-Structure Indexing of Semi-Structured Hierarchical Data

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by

Kevin Wellenzohn

from South Tyrol, Italy

approved in December, 2021

at the request of

Prof. Dr. Michael H. Böhlen
Dr. Sven Helmer
Prof. Dr. Viktor Leis

Abstract

Large amounts of semi-structured, hierarchical data are generated every day. A frequent type of queries on such data are *Content-and-Structure (CAS)* queries that consist of two predicates: a value predicate on the content and a path predicate on the hierarchical structure of the data. CAS queries select data items based on their value for some attribute and their location in the hierarchical structure of the data. Dedicated CAS indexes exist to evaluate CAS queries, but they partially or completely lack two important qualities that we seek in a CAS index: *robustness* and *scalability*.

Robustness means that a CAS index optimizes the average runtime across all queries and not the runtime of individual queries. Scalability means that a CAS index can cope with big semi-structured, hierarchical data. A scalable CAS index can be efficiently bulk-loaded to create a new index for large datasets and it supports efficient updates to keep up with the influx of new data.

Together, robustness and scalability make a CAS index useful in real-world use cases. Robustness ensures that the index can deal with a wide variety of CAS queries in an efficient way, while scalability makes sure that the index can keep up with the increasing amount of semi-structured, hierarchical data. This thesis presents the first CAS index that offers robust CAS query performance and scales to big semi-structured, hierarchical data.

The robustness of our solution is rooted in the observation that existing CAS indexes fail to integrate the content and structure of the data in one index without prioritizing one of the two

dimensions (paths or values). Consequently, to offer robust CAS query performance we develop a novel interleaving scheme, called *dynamic interleaving*, that interleaves the binary representation of the paths and values of data items in a well-balanced way without prioritizing one of the dimensions. This is challenging because of the different characteristics of paths and values: while the paths are long sequences of node labels, the values are basic data types, like numbers, short strings, etc. Our dynamic interleaving accounts for these differences by adapting to the data distribution and skipping long common prefixes in the paths and values. We store dynamically-interleaved keys in our trie-based Robust Content-and-Structure (RCAS) index to query them efficiently. Its trie-based structure allows our index to efficiently evaluate the value and path predicates of CAS queries using a mix of range and prefix searches, respectively. Since the keys are dynamically interleaved, we can evaluate the two predicates simultaneously to avoid large intermediate results and offer robust query performance.

To scale our index to large datasets we store our RCAS index compactly on block-based storage devices. We propose an efficient bulk-loading algorithm for our index that uses several techniques to optimize the CPU, disk, and memory usage of the algorithm. The algorithm has a small memory footprint and optimally uses the remaining memory to curb the algorithm's disk I/O. We develop two algorithms to restructure the RCAS index when the index is updated. The first algorithm optimizes for query performance at the expense of update performance and the second algorithm optimizes for update performance while still achieving good query performance in practice.

We analytically prove the robustness and scalability of our index and confirm these results experimentally by indexing and querying, among other things, data from the Software Heritage archive, which is the world's largest publicly-available software archive.

To Katrin

Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Michael H. Böhlen, for his guidance and constructive criticism throughout my PhD. Thank you for never settling for less when more was possible.

I would like to thank Dr. Sven Helmer, who, after supervising already my Bachelor's thesis, has agreed to co-advise me again during my PhD. Thank you for introducing me to research and showing me how much fun it can be.

A special thanks to Prof. Dr. Viktor Leis for agreeing to be the co-advisor of my PhD thesis and to Prof. Dr. Thomas Fritz for chairing my PhD thesis defense.

Many thanks to Prof. Dr. Johann Gamper for always looking out for me and nudging me in the right direction at difficult crossroads in my (academic) career.

I would like to thank my co-authors for the valuable discussions and their contributions. In particular, thank you to Anton Dignös, Antoine Pietri, and Stefano Zacchiroli for their help.

Thank you to all my current and former colleagues at the DBTG group. I am deeply grateful for all the cheerful discussions and the great time that we had inside and outside the office.

Lastly, I am truly indebted to my partner, Katrin. Thank you for your love and support.

Kevin Wellenzohn
Zurich, September 2021

Contents

Abstract **iii**

Acknowledgments **vii**

I Synopsis 1

1 Introduction **3**

1.1 Running Example 8
1.2 Related Work 10
1.2.1 CAS Indexing 10
1.2.2 Linearizing Multi-Dimensional Keys 12
1.2.3 Building and Updating Indexes 15
1.3 Challenges 16

2 Contributions **19**

2.1 Dynamic Interleaving 20
2.2 Robust Content-and-Structure (RCAS) Index 22
2.3 Updating the RCAS Index 25
2.4 Scalable $RCAS^{+}$ Index 28

3 Thesis Roadmap **31**

4 Conclusion **33**
4.1 Limitations 33
4.2 Summary 35
4.3 Future Work 36

II Publications **39**

A Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data **41**
A.1 Introduction 42
A.2 Running Example 44
A.3 Related Work 45
A.4 Background 47
A.5 Dynamic Interleaving 50
A.5.1 Partitioning by Discriminative Bytes 51
A.5.2 Interleaving 54
A.5.3 Efficiency of Interleavings 55
A.6 RCAS Index 59
A.6.1 Trie-Based Structure of RCAS 59
A.6.2 Physical Node Layout 60
A.6.3 Bulk-Loading RCAS 61
A.6.4 Querying RCAS 64
A.7 Experimental Evaluation 67
A.7.1 Setup and Datasets 67
A.7.2 Impact of Datasets on RCAS's Structure 68
A.7.3 Robustness 71
A.7.4 Evaluation of Cost Model 74
A.7.5 Space Consumption and Scalability 75
A.7.6 Summary 76
A.8 Conclusion and Outlook 77

B Inserting Keys into the Robust Content-and-Structure (RCAS) Index 79

B.1 Introduction 80
B.2 Background 81
B.3 Insertion of New Keys 83
B.4 Index Restructuring during Insertion 84
B.4.1 Strict Restructuring 85
B.4.2 Lazy Restructuring 86
B.5 Utilizing an Auxiliary Index 88
B.6 Analysis 90
B.7 Experimental Evaluation 91
B.7.1 Runtime of Strict and Lazy Restructuring 91
B.7.2 Query Runtime 92
B.7.3 Merging of Auxiliary and Main Index 94
B.7.4 Summary 95
B.8 Related Work 95
B.9 Conclusion and Outlook 96

C Scalable Content-and-Structure Indexing 97

C.1 Introduction 98
C.2 Application Scenario 100
C.3 Background 101
C.3.1 Notation & Terminology 101
C.3.2 Dynamic Interleaving in the RCAS Index 103
C.4 The Scalable RCAS$^+$ Index 104
C.4.1 Depth-First Bulk-Loading 105
C.4.2 Lazy Interleaving 109
C.4.3 Node Clustering 110
C.5 Proactive Partitioning 110
C.5.1 Implementation 111
C.5.2 Properties of the Partitioning 112
C.6 Front-Loading 114
C.6.1 Implementation 115

C.6.2 Analysis 116
C.7 Analytical Evaluation 117
C.7.1 I/O Overhead 117
C.7.2 Space Overhead 119
C.8 Experimental Evaluation 119
C.8.1 Scalability of Depth-First Bulk-Loading 120
C.8.2 Query Performance 121
C.8.3 Node Clustering 122
C.8.4 Lazy Interleaving 122
C.8.5 Proactive Partitioning 123
C.8.6 Front-Loading 125
C.8.7 Cost Model 126
C.8.8 Summary 127
C.9 Related Work 127
C.10 Conclusion and Outlook 129

D Curriculum Vitae **131**

Bibliography **135**

Part I

Synopsis

CHAPTER 1

Introduction

A large part of real-world data does not follow the rigid structure of tables found in relational database management systems. Instead, a substantial amount of data is semi-structured, which means that each data item is stored with a schema that defines its structure [LH19]. Data items are marked-up with labels and annotated with attributes to make them self-descriptive such that the data can be interpreted by humans and machines alike. Since data items can be nested, this leads to a hierarchical structure. A data item has two important dimensions: its *content* and its location in the *hierarchical structure*. The content of a data item stores its actual information and its location in the hierarchical structure is the context that is required to interpret the data item. Without its structure, a data item cannot be interpreted and without its content, the item carries no information.

Semi-structured, hierarchical data can be found in a wide range of application domains. In engineering, e.g., a bill-of-materials (BOM) is semi-structured and describes the hierarchical assembly of components into a final product, where each component may have a distinct set of attributes [BFF$^+$15]. For example, the BOM of a car shows that the car is assembled of an engine and the chassis (among other things), and the chassis itself is assembled of tires, etc. Different pieces of information are recorded for different components: for the tires, e.g., the BOM records

```
<bom>
  <car>
    <body weight="325000g" />
    <chassis>
      <tire rimsize="17in" maxspeed="240kmh" />
      <axle diameter="2in" />
    </chassis>
    <engine torque="660Nm" power="615kW">
      <cylinder capacity="500cc" />
      <piston />
      <oilpan capacity="5500ml" />
    </engine>
    <interior>
      <rearmirror weight="500g" />
      <seat material="polyester" quantity="5" />
      <seatbelt type="three-point" />
      <steeringwheel material="leather" />
    </interior>
  </car>
</bom>
```

(a) A bill-of-materials (BOM) in XML format.

```
/                                 Modification
├─ arch/                              date:
│  └─ arm64/
│     └─ kernel/
│        └─ acpi.c                2020-09-30
├─ crypto/
│  ├─ ecc.c                       2021-03-19
│  └─ ecc.h                       2021-04-06
├─ fs/
│  ├─ ext2/
│  │  └─ ext2.h                   2021-04-06
│  └─ ext4/
│     ├─ inode.c                  2021-03-27
│     └─ resize.c                 2020-12-16
└─ README                         2018-09-04
```

(b) Subset of files in the Linux kernel.

```
{ "created_at": "Thu Apr 06 15:24:15 +0000 2017",
  "user": { "id": 2244994945, "name": "Twitter Dev" },
  "text": "Today we're sharing our vision for the future of the Twitter API platform! https://t
      .co/XweGngmxlP",
  "entities": {
    "urls": [
      { "url": "https://t.co/XweGngmxlP",
        "unwound": {
          "url": "https://cards.twitter.com/cards/18ce53wgo4h/3xo1c",
          "title": "Building the Future of the Twitter API Platform" }}]
  }
}
```

(c) A tweet in JSON format.

Figure 1.1: Examples of semi-structured, hierarchical data.

their profile, rim size, and maximum speed, while for the engine the BOM records its power, torque, etc. Figure 1.1a shows an example of a BOM of a car in the XML storage format. The BOM is self-descriptive due to its hierarchical, semi-structured model. The labels of the marked-up data items (e.g., <tire>) describe a component in the BOM and the nesting of tags describes the hierarchical assembly of components. Data items can have attributes (e.g., weight, capacity, etc.) that describe a property of the component and each component in the car has its own set of attributes. Each component in the BOM is described by its location in the hierarchical structure and its set of attributes. For example, the tires are part of the car's chassis, evidenced by the path /bom/car/chassis/tire, and are characterized by their attributes: they have a rim size of 17 inches and a maximum speed of 240 kilometers per hour. Another application domain of semi-structured, hierarchical data is the web since semi-structured hierarchical data-formats like

XML, and more recently, JSON are heavily used as data-interchange formats. Figure 1.1c shows a tweet retrieved through Twitter's API in the JSON format. The tweet's `created_at` attribute shows that the tweet was posted on 2017-04-06; its `user` attribute is nested and shows the ID of the user who posted it (2244994945) and its name ("Twitter Dev"). Web sites themselves are semi-structured documents since they are written in HTML, which is a hierarchical mark-up language similar to XML. File systems are another example where data items (i.e., files) are organized hierarchically. For example, Unix-like operating systems store configuration files in the folder `/etc`, personal files in `/home`, etc., and users can freely create more folders and nest them arbitrarily. Lastly, version control systems like git, svn, etc. store the hierarchical structure of software repositories and additional information such as when a file was updated, who updated it, etc. Figure 1.1b shows a subset of the files in the Linux kernel and when they were last updated at the time of writing according to the Linux git repository. The path of a file helps to explain what part of the kernel is implemented by the file. For example, just by looking at the file `resize.c` it is not clear what exactly is resized by this code, but knowing its full path `/fs/ext4/resize.c` it is clear that the code is used to resize the ext4 filesystem.

Due to the surge of semi-structured hierarchical data, database researchers and vendors have developed systems and techniques to efficiently store, index, and query this kind of data with a particular focus on XML and JSON data. Existing relational database systems like Oracle and PostgreSQL have been extended to support XML and JSON data, while systems like Natix [FHK+02] and Sedna [TSK+10] were developed from the ground up as native XML stores. Recently, a new class of NoSQL systems has emerged that addresses the need to handle big (semi-structured hierarchical) data [DCL18]. For example, new document stores like MongoDB and Couchbase Server address the need to natively manage JSON documents at scale. In this thesis we do not focus on one particular type of semi-structured, hierarchical data (e.g., XML or JSON) and we do not target one specific type of database system (e.g., native document stores). Instead, we look at the problem of indexing semi-structured hierarchical data irrespective of how and where the data is stored.

Increasing interest in semi-structured hierarchical data has lead to a new set of query languages that address the characteristics of the data. Query languages like XPath [CD99], XQuery [BCF+10], and JSONiq [FF13] have been developed that allow fine-grained, navigational access to semi-structured hierarchical data. These languages offer sophisticated constructs to filter data items based on their content and their location in the hierarchical structure of the data by specifying their relationship to other data items (e.g., parent-child, ancestor-descendant, and sometimes

sibling relationships). In this thesis we focus on *Content-and-Structure (CAS)* queries [MHSB15] that filter data items based on their location in the hierarchical structure and their value for some attribute. CAS queries consist of a *path predicate* and a *value predicate*. The path predicate is expressed as a query path that matches the paths of all wanted data items. A query path consists of node labels that are connected through parent-child or ancestor-descendant relationships to widen the search when the exact location of a data item is not known in the hierarchical structure. For example, the query path `/bom/car//bolt` contains the parent-child relationship `/` and the ancestor-descendant relationship `//` (also known as the descendant axis), where the latter can skip zero to any number of node labels. As a result, this query path matches paths like `/bom/car/bolt`, `/bom/car/engine/bolt`, etc. In addition, wildcards can be used to skip one node label fully or partially (e.g., `/ext*/app.c` matches the paths `/ext4/app.c`, `/extension/app.c`, etc.). On the other hand, the value predicate refers to some attribute A of the data and is expressed as a range predicate $x \leq A \leq y$ that matches all data items whose value for attribute A is between x and y. For example, the value predicate $1000 \leq$ `weight` ≤ 2000 selects all data items whose weight is between 1000 and 2000 grams. CAS queries can appear as building blocks for more complex twig-pattern queries [BKH+17] that specify tree-shaped patterns and that are matched against the hierarchical data stored in the database.

Answering queries on big semi-structured, hierarchical data is expensive unless the data is indexed. A number of techniques have been proposed to index the content *or* the structure of semi-structured hierarchical data, see [BKH+17, MHSB15] for good overviews. The DataGuide [GW97] and its derivatives, for example, are structural indexes that summarize all the paths in the data, but they do not index the content of the data. Pure content indexes such as B+ trees and similar index structures are used to index all the values in the data, but they ignore its structure. These indexes cannot answer CAS queries efficiently because they ignore one of the two dimensions of the semi-structured, hierarchical data (either its content or its structure). A number of approaches have been proposed that index the content *and* structure of the data (called CAS indexes [MHSB15]). Existing CAS indexes [CSF+01, KKNR04, LAAE06, MHSB15, STR+15] do not have robust CAS query performance because they either build separate indexes for the content and structure that need to be joined [KKNR04, MHSB15] or they fix the order of the dimensions a priori (i.e., they first index the content and then the structure, or vice versa) [CSF+01, LAAE06]. This fails for CAS queries that have a high selectivity[1] for their individual path and value pred-

[1] The selectivity of a predicate is the fraction of all data items for which the predicate returns true. A predicate with high selectivity matches many data items.

icates, but a low final selectivity because evaluating each predicate individually leads to large intermediate results that need to be narrowed down to a small final result.

We look for two qualities in a CAS index that we found partially or completely missing in existing CAS indexes: *robustness* and *scalability*. Since these terms are not well-defined in the literature, we provide our own definitions:

Robustness: We call a CAS index robust if – in the absence of any information about the query workload – the index optimizes the average query runtime over all queries.

Scalability: We call a CAS index scalable if (i) it is not constrained by the size of the available memory, (ii) the index supports bulk-loading for large datasets, and (iii) the index can be updated efficiently.

Robustness and scalability make a CAS index useful in practice. Robust query performance makes sure that a CAS index can efficiently answer a wide variety of CAS queries. The goal of a robust CAS index is not to be the fastest index for every single CAS query, instead we want to have the best performance, on average. Not knowing the query workload beforehand, a robust CAS index must be prepared to answer ad-hoc CAS queries efficiently. This is especially useful in exploratory data analysis when users do not know the data and pose a series of queries to familiarize themselves with it.

Scalability means that a CAS index works for large real-world datasets. We focus on large datasets that can be managed on a single machine, we do not consider distributed setups. In this thesis we work, e.g., with semi-structured, hierarchical data from the Software Heritage (SWH) archive [DCZ17, PSZ20], which is the world's largest publicly-available software archive. At the time of writing, SWH has crawled 156 million software repositories from places like GitHub, GitLab, etc., and has stored 2.1 billion commits and 10 billion unique source code files, and these numbers are growing daily. To operate a CAS index at this scale we need an index that is not constrained by the size of the available memory on a single machine, that can be efficiently created for large datasets, and that can keep up with the influx of new data.

This thesis is about developing a robust and scalable CAS index for semi-structured, hierarchical data. We analytically prove the robustness and scalability of our index and confirm these results experimentally by indexing, among other things, data from the SWH archive.

1.1 Running Example

We consider a company that stores the bills of materials (BOMs) of its products. Figure 1.2 shows the hierarchical representation of a BOM for three products (as explained above, we do not assume any particular storage format like XML, JSON, etc.). The components of each product are organized under a node with label `item`. Components can have attributes to record additional information, e.g., the weight and capacity of a battery. Attributes are represented by special nodes that are prefixed with an `@` and that have an additional value. For example, the weight of the rightmost battery is 250714 grams and its capacity is 80000 Wh.

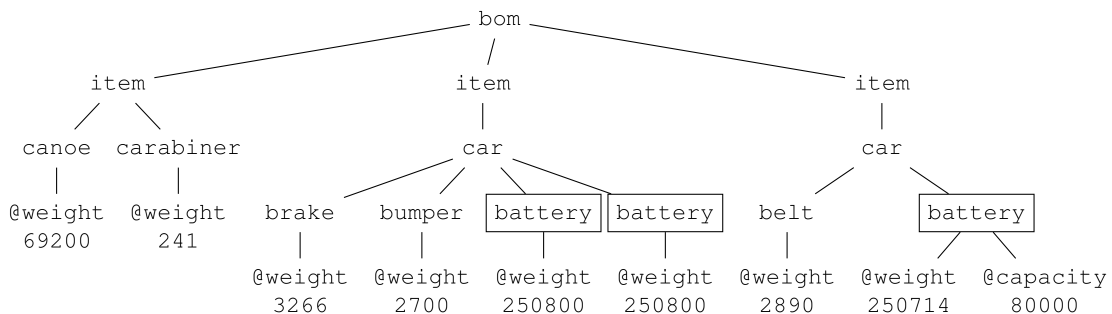


Figure 1.2: Example of a bill-of-materials (BOM).

Engineers working on the products in Figure 1.2 routinely use CAS queries to find relevant components in the BOM. Consider the following example.

Example 1.1. *The engineers are looking for ways to cut the weight of cars. Thus, they are looking for all heavy car parts that weigh at least 50 kilograms in Figure 1.2. They issue the following CAS query using a syntax similar to XQuery, where "//" is the descendant axis that matches a node and all its descendants in a hierarchical structure:*

```
Q:  for $c in /bom/item/car//
    where $c/@weight >= 50000
    return $c
```

This query consists of a path predicate expressed as the query path `/bom/item/car//` and the value predicate expressed as the range `@weight >= 50000`. The path predicate matches all car parts and there are six of them in Figure 1.2. Likewise, the value predicate matches all parts heavier than 50000 grams and there are four of them. The CAS query is the conjunction of both

predicates and returns only three data items (the three framed nodes). Evaluating these predicates individually or one after the other leads to large intermediate results, which is expensive.

In our running example fast access is needed to the location of components in the hierarchical assembly of a product and their value for the `@weight` attribute. Therefore, we want to build a CAS index on the paths of components and their value for the `@weight` attribute. We represent the data items that need to be indexed as two-dimensional keys, called *composite keys*, that account for the content and structure of a data item, see Table 1.1 for an example. A composite key k consists of a value dimension V that represents the content of a data item and a path dimension P that represents its location in the hierarchical structure of the data. The path of a key k, denoted by $k.P$, is given by the labels of all nodes from the root of the hierarchical structure to the node that this key represents; the labels are separated by a `/` and the path is terminated by the end-of-string character `$`. Similarly, the value of a key k, denoted by $k.V$, stores the content (e.g., an attribute value) of the node that k represents. Additionally, each key k stores a reference $k.R$ that points to the indexed data item. How exactly the reference is implemented depends on the system; it can be, e.g., the physical address of the data item in memory or on disk, or a unique ID generated with a node labeling scheme like OrdPath [OOP+04] (explained below). We denote a set of composite keys by K. We use a sans-serif font to refer to concrete values in our examples. Further, we use notation $\mathsf{K}^{2,5,6,7}$ to refer to $\{\mathsf{k}_2, \mathsf{k}_5, \mathsf{k}_6, \mathsf{k}_7\}$.

Table 1.1: A set $\mathsf{K}^{1..7} = \{\mathsf{k}_1, \ldots, \mathsf{k}_7\}$ of composite keys.

	Path Dimension P	Value Dimension V	R
k_1	`/bom/item/canoe$`	69200 (`00 01 0E 50`)	r_1
k_2	`/bom/item/carabiner$`	241 (`00 00 00 F1`)	r_2
k_3	`/bom/item/car/battery$`	250714 (`00 03 D3 5A`)	r_3
k_4	`/bom/item/car/battery$`	250800 (`00 03 D3 B0`)	r_4
k_5	`/bom/item/car/belt$`	2890 (`00 00 0B 4A`)	r_5
k_6	`/bom/item/car/brake$`	3266 (`00 00 0C C2`)	r_6
k_7	`/bom/item/car/bumper$`	2700 (`00 00 0A 8C`)	r_7
	1 3 5 7 9 11 13 15 17 19 21 23	1 2 3 4	

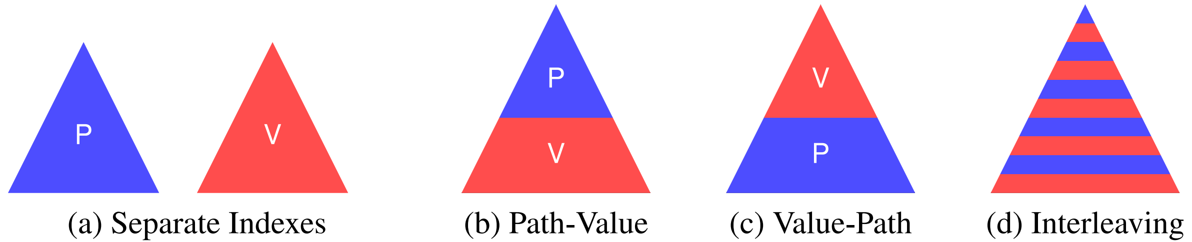
Example 1.2. *Table 1.1 shows a set $\mathsf{K}^{1..7}$ of seven composite keys taken from the BOM in Figure 1.2. The path dimension denotes the path from the root to the data item and the value dimension denotes its value for the attribute `@weight`. Composite key k_1 denotes that the canoe has a weight of 69 200 grams. k_1's path $\mathsf{k}_1.P$ = `/bom/item/canoe$` contains all node labels starting from the root node, delimited by the path separator `/`, and ending with the end-of-string*

character $\$$. The reference r_1 points to the data item that has this path and value. The CAS query in our running example matches keys $\{\mathsf{k}_3, \mathsf{k}_4\}$ in Table 1.1.

1.2 Related Work

1.2.1 CAS Indexing

In the following we outline the state-of-the-art in CAS indexing and show that existing CAS indexes are not robust and/or do not scale. Figure 1.3 shows a conceptual overview how the paths and values are indexed in existing CAS indexes. Blue denotes the paths and red the values.



(a) Separate Indexes (b) Path-Value (c) Value-Path (d) Interleaving

Figure 1.3: Conceptual overview of different approaches to index paths and values.

The CAS index by Mathis et al. [MHSB15] consists of two separate index structures: a value index and a path index (see Figure 1.3a). The value index is a regular B-tree and the path index is a structural summary (e.g., a DataGuide [GW97]) that summarizes all paths in the data. The path index assigns to each distinct path a unique identifier, termed path-class reference (PCR). The value index, i.e., the B-tree, stores tuples of the form $(\texttt{value}, \langle\texttt{nodeId}, \texttt{PCR}\rangle)$ in its leaves. The first element, `value`, is the value of the indexed attribute for a key. The next element, `nodeId`, uniquely identifies the node that has the given value. The node identifier is based on a node-labeling scheme (e.g., OrdPath [OOP+04]) that encodes the position of the node in the hierarchical structure of the data. The last element, `PCR`, uniquely identifies the path in the structural summary. In our running example we assume a CAS index is built on the `@weight` attribute. Then, the tuple $(69\,200, \langle\texttt{1.1.1}, 20\rangle)$ denotes that the node with ID `1.1.1`, referring to the first child of the first child of the root node in Figure 1.2, has the PCR 20, e.g., referring to the path `/bom/item/canoe`, and its value for attribute `@weight` is 69 200. In other words, this tuple refers to the composite key k_1 in Table 1.1. Answering a CAS query requires looking at the path and the value index. Two evaluation strategies are possible. First, the path and value

predicates of the CAS query can be evaluated independently on the respective index structures and the intermediate results are joined on the PCR. This is expensive if the intermediate results are large (i.e., at least one predicate has a high selectivity) but the final result is small (i.e., the CAS query has a low selectivity). Second, the value predicate is evaluated and for each query match its PCR is looked up in the path index to see if it satisfies the path predicate. The problem with this strategy is that if the value predicate has a high selectivity, a large number of point lookups must be made in the path index, which is expensive. Additionally, if the final result is small, a large intermediate result must be narrowed down to a small final result. With either strategy, this CAS index is not robust since its performance is dominated by the size of its intermediate results. In terms of scalability, Mathis et al. [MHSB15] design their CAS index as a scalable disk-based index. Both index structures, the DataGuide and the B-tree, can be efficiently bulk-loaded and updated. Bulk-loading B-trees is sort-based and discussed in more detail in Section 1.2.3.

IndexFabric [CSF$^+$01] is a CAS index that concatenates the paths and values of composite keys and stores them in a trie data-structure. Unlike comparison-based trees (e.g., binary search trees, B-trees etc.), a trie does not store keys in its leave nodes. Instead, in their simplest form, tries work like a thumb index: we look at the first letter in the search term and jump to all keys that begin with that letter, then we repeat this process one letter at a time until we looked at all letters in a key. Each node in a trie stores a letter and when we concatenate all the letters on a root-to-leaf path we get a key that is stored in the trie. Since IndexFabric concatenates composite keys and stores them in a trie, the index has two separate layers: the upper layer contains the paths and the lower layer contains the values. Figure 1.3b visualizes this approach. As a result, IndexFabric prioritizes the structure of the data over its values since the paths are ordered before the values. To answer a CAS query, IndexFabric must first fully evaluate the query's path predicate before it can evaluate its value predicate. This leads to large intermediate results if the path predicate has a high selectivity and consequently, IndexFabric is not robust. IndexFabric is based on a disk-optimized trie that supports updates (tough the details are lost in a technical report that is no longer accessible online). Bulk-loading is not discussed.

The hierarchical database system Apache Jackrabbit Oak [Apa20] implements the property index. This index sorts the values and for each value it stores a structural summary (e.g., a DataGuide [GW97]) of all paths in the database that have this particular value. This again separates values and paths, as shown in Figure 1.3c. The problem remains the same: when the selectivity of the dimension that is ordered first (in this case the value dimension) is high,

there can be large intermediate results that need to be narrowed down. As a result, also the property index is not robust. Jackrabbit Oak is a distributed database system with a focus on scalability. Its property index can be bulk-loaded and updated. Oak implements a node storage interface that allows for different storage backends to be used. To scale Oak to large datasets, MongoDB [Mon20] is the recommended storage backend.

More approaches for CAS indexing exist (e.g., [STR+15, KKNR04, LAAE06], etc.), but they share similar problems as the ones outlined above. Besides CAS indexes, a number of pure content and pure structure indexes exist, but they are not designed to answer CAS queries efficiently.

The problem with existing CAS indexes is that they do not integrate the paths and values of composite keys. Instead, they keep them in separate index structures or, if they keep them together in one index, they prioritize one dimension over the other. To achieve robust CAS query performance a solution is needed to integrate the content and structure in a well-balanced way.

1.2.2 Linearizing Multi-Dimensional Keys

At its core, CAS indexing is a special case of indexing multi-dimensional data with the difference that CAS indexing focuses on two dimensions and that one of these two dimensions, the paths, have a different semantics from the basic data types (numbers, strings, etc.) that are typically indexed in multi-dimensional indexes. As seen before, existing CAS indexes lack a well-balanced integration of paths and values. There exist a number of schemes that integrate the dimensions of multi-dimensional keys and map them to one-dimensional keys. A common approach is to linearize composite keys using space-filling curves, like the c-order curve [NY17], the z-order curve [Mor66, OM84], or the Hilbert curve [Hil91]. Space-filling curves map a multi-dimensional space onto a one-dimensional space and try to keep keys that are close to each other in multi-dimensional space also close to one another in the one-dimensional space.

The c-order curve [NY17] is the simplest space-filling curve and is obtained by concatenating the individual dimensions of a composite key according to a given ordering of the dimensions. In our case, there are two orderings: path followed by value, or vice versa. Table 1.2 shows various ways to integrate the dimensions of key k_6 from Table 1.1. Here, the values are stored as 32-bit unsigned integers and represented in hexadecimal. Value bytes are written in italic and shown in red, path bytes are shown in blue. The first two rows show the two possible c-order curves. The IndexFabric [CSF+01] and the property index in Apache Jackrabbit Oak [Apa20], in

essence, implement these two schemes as seen above. The *c*-order does not lead to robust query performance because the query predicates must be evaluated one after another according to the same ordering of the dimensions. If the predicate on the first dimension has a high selectivity, the keys cannot be pruned effectively and many keys must be considered for the second dimension.

Table 1.2: Key k_6 is interleaved using different approaches

Approach	Interleaving of Key k_6
Path-Value Concatenation	`/bom/item/car/brake$` *00 00 0C C2*
Value-Path Concatenation	*00 00 0C C2* `/bom/item/car/brake$`
Byte-Wise Interleaving	*00* `/` *00* `b` *0C* `o` *C2* `m/item/car/brake$`

The *z*-order curve [Mor66, OM84] is obtained by the bit-wise interleaving of the individual dimensions. Figure 1.3d shows how conceptually paths and values are interleaved. Due to its interleaving, the *z*-order leads in theory to a more robust query performance since the query predicates are no longer evaluated one after another but can be evaluated simultaneously. The *z*-order curve is a *static* interleaving scheme since the interleaving follows a pre-defined pattern (i.e., the first bit of every dimension is interleaved, then the second, etc.). The problem with the *z*-order curve is twofold. First, since the *z*-order is static, it is oblivious to the data distribution and can interleave the dimensions at common prefixes. The problem with common prefixes is that they do not partition the data and therefore interleaving at a common prefix does *not* make *progress*. Common prefixes do not help to prune the search space during a search since common prefixes are the same for all data items. This means that during a search either a query matches the common prefix, in which case we must look at all keys, or the query does not match the common prefix, in which case we must discard all keys. Discarding keys selectively is not possible at a common prefix. Interleaving a common prefix in one dimension with a non-common prefix in the other dimension means we can prune keys in one dimension but not the other (i.e., there is progress in one dimension, but not the other). As a result, the *z*-order curve can prioritize one dimension over another. The second problem is that it is unclear how to encode variable-length composite keys when the *z*-order is applied in CAS indexing. Typically, *z*-order is applied when all the dimensions of a key have the same basic data type (e.g., every dimension is stored as a 32 bit number). In our case, paths are variable-length strings that encode a variable number of node labels, while the values are basic data types like integers, floating point numbers, or simple strings. The last row in Table 1.2 shows one possible way to interleave the paths and values: namely byte-wise. This scheme makes the first problem even worse since now all bytes of the value are interleaved with a common path prefix. In practice this means the byte-wise interleaving behaves exactly like the value-path concatenation, which does not lead to robust query

performance. To better handle variable-length keys, Markl [Mar99] suggests surrogate functions that map variable-length keys to fixed-length keys. The problem here is to find a surrogate function that (i) preserves the sort order in the value dimension, (ii) maintains the hierarchical relationships between nodes in the path dimension, and (iii) can be updated when the data changes. We need points (i) and (ii) to evaluate the value and path predicates of CAS queries efficiently, and point (iii) – efficient updates – is required for scalability.

Nishimura et al. [NY17] propose an approach, called QUILTS, to design space-filling curves based on the query workload. QUILTS devises static interleaving schemes that prioritize the dimensions that lead to the fastest query execution for the given query workload. It achieves this by changing the order in which the bits of the different dimensions are interleaved. With the c-order and z-order curves as the two extreme ends of a possible range of interleaving schemes (no interleaving vs. interleaving every bit), QUILTS exploits knowing the query workload and rearranges the interleaving order to, e.g., interleave two bits of the first dimension, followed by three bits of the other dimension, etc. Applying QUILTS to CAS indexing is difficult for several reasons. First, like the z-order, also QUILTS assumes that the keys have a fixed length and, as shown above, current approaches to interleave variable-length keys are flawed. Second, QUILTS requires information about the query workload, which is often not known in advance. In this thesis we assume no knowledge of the query workload, instead we want to support a wide range of queries in a robust way, including ad-hoc queries that are used in exploratory data analysis. Another problem is that Nishimura et al. do not discuss what happens if the query workload changes. Presumably, either the query performance deteriorates if the interleaving is not updated, or, if the static interleaving scheme is updated according to the changing query workload, *all* keys must be interleaved anew. This is expensive and limits the scalability of this approach.

Interleaving keys is only the first step; we must also store keys in suitable index structures to query them efficiently. The benefit of interleaving schemes is that traditional one-dimensional index structures can be re-used to store and query multi-dimensional data. For example, the UB-tree [RMF$^+$00] is a B-tree that stores z-encoded keys and likewise the kd-trie [OM84] is a trie that stores z-encoded keys.

The problem with existing interleaving schemes is that they are static and ignore the data distribution. As a result, they are oblivious to common prefixes in the data and interleaving keys at common prefixes does not lead to robust query performance. In addition, they struggle with

variable-length keys because it is unclear how to best interleave them. In summary, static interleavings do not offer a well-balanced integration of paths and values of composite keys.

1.2.3 Building and Updating Indexes

One requirement that we defined for a scalable CAS index is that it it must be possible (i) to efficiently build the index from scratch for large datasets, and (ii) to update the index when the data changes. In the following we look at the standard techniques to create a new index using bulk-loading and how to insert/delete data efficiently. Three common approaches for bulk-loading exist: sort-based, sampling-based, and buffer-based approaches. We look at these approaches in turns.

Sort-based bulk-loading is the standard approach to build B-trees and is widely implemented in database systems (e.g., PostgreSQL, etc.). Unless the data fits into memory, the data is first sorted externally, e.g., using external sort merge, and then the index is built bottom-up, level by level [KPT91]. Sort-based bulk-loading is more difficult for multi-dimensional data since the sort-order is unclear. One solution to fix the sort-order is to linearize multi-dimensional keys using space-filling curves (see above). This approach works if keys can be efficiently linearized, which is the case for static interleaving schemes like the z-order curve that interleaves a key in constant time. If, on the other hand, computing the interleaving of a key requires more than constant time, this can become a bottleneck that dominates the runtime of sort-based bulk-loading. In this thesis we propose a dynamic, data-driven interleaving scheme that interleaves a *set* of keys as a whole and not each key in the set individually. In this dynamic interleaving scheme the cost of interleaving a key is not constant since we cannot interleave one key without looking at all other keys. This makes sort-based bulk-loading with our this dynamic interleaving scheme impractical for large datasets.

Sampling-based bulk-loading [AS10, dBS01, GSM$^+$04] draws a sample of the dataset to build a skeleton of the target index in memory and later extend that index to disk. For example, van den Bercken et al. [dBS01] build an index in memory top-down from a sample of the data, attach disk-based buffers to each leaf node, insert the remaining keys into the leaf buffers, and recursively call the algorithm on each leaf buffer. Combining sampling-based bulk-loading with interleaving-based indexing works only if the interleaving can be computed from a sample of the data. Since our dynamic interleaving scheme interleaves *all* keys at the same time, it is not

possible to derive the dynamic interleaving from a sample of the data. Doing so would lead to an incorrect index that can return wrong query results.

A third class of algorithms is based on the buffer-tree technique [Arg03, AHVV02], where each node in a tree has a buffer that accumulates updates for the subtree. Only when the buffer is full, the updates in the buffer are propagated one level down. Batching the updates allows for efficient bulk-loading and index updates. Much like in the previous case, looking at small batches is insufficient since we need to look at all the data to dynamically interleave it.

In terms of efficiently updating indexes there exist a large number of approaches. Each index structure comes with its own set of update routines that are tailored to the characteristics of the data structure. For example, B-trees use splitting and merging to implement insertion and deletion, respectively, and to ensure that the tree remains balanced, while red-black trees use rebalancing to maintain the integrity of the tree after updates. Updating a trie data-structure is simpler since it is not a balanced tree. To insert a key, the trie is traversed starting from the root node and the letters in the key are compared to the letters in the nodes. The traversal stops when the next node to traverse to cannot be found and at that position the remaining suffix of the key is inserted. Modern trie implementations [AZ09, HZW02, LKN13, Mor68] have more complex update routines, but the basic principle is the same.

Existing bulk-loading strategies are efficient and have proven their value in commercial and open-source database systems, but the problem is that combining these strategies with a dynamic interleaving scheme is difficult. Either the performance is subpar or, in the case of bulk-loading the index through small samples or in batches, the index can return incorrect query results.

1.3 Challenges

In this thesis we address four challenges: (i) a *well-balanced interleaving* of the paths and values of composite keys in semi-structured, hierarchical data, (ii) building a *robust* CAS index based on interleaving the paths and values, (iii) supporting *updates* to this index when new data arrives, and (iv) *scaling* this index to big datasets.

Challenge 1: Well-Balanced Interleaving of Paths and Values. The first challenge that we address in this thesis is to find a way to interleave the paths and values of composite keys without prioritizing either dimension. Existing static interleaving schemes struggle with variable-length

keys and are vulnerable to long common prefixes in the data because they are static and ignore the data distribution. A well-balanced integration must handle long common prefixes effectively since especially the paths in a hierarchical structure have, by their very nature, long common prefixes. We develop a novel interleaving scheme, the *dynamic interleaving*, that offers a well-balanced integration of paths and values.

Challenge 2: Robust CAS Indexing. The dynamic interleaving defines how the paths and values are interleaved, but without suitable index structure the interleaved keys cannot be queried efficiently. Therefore, the second challenge that we address in this thesis is to build a robust CAS index that utilizes the dynamic interleaving. Finding the right data structure for our CAS index is not straightforward because the data structure must support two different access methods. On the one hand, we have path predicates that can contain besides simple parent-child relationships also more complex ancestor-descendant relationships and wildcards. On the other hand, the index structure must support value predicates that are expressed as a range $[a,b]$, which enables point lookups (if $a = b$) and range searches (if $a < b$). To implement the path and value predicates, we need a data structure that supports prefix and range searches, respectively. A prefix search matches paths that have the same prefix as the query path and a range search matches values that fall within the given range. Given these requirements, a *trie*, also known as prefix tree or radix tree, is the natural choice since it efficiently supports both search methods. Therefore, we develop our *Robust CAS (RCAS)* index on top of a memory-optimized trie structure (ART [LKN13]).

Challenge 3: Updating CAS Indexes. The third challenge is how to efficiently insert/delete keys in the RCAS index while preserving RCAS's robust query performance. Updating the RCAS index is difficult due to its dynamic interleaving scheme. Since the dynamic interleaving is data-driven and the interleaving of one key depends on all other keys, inserting or deleting just one key into the index can change the dynamic interleaving of a large number of keys. This means in the worst case we need to restructure a significant part of the RCAS index to preserve the dynamic interleaving of the keys, which is expensive. Here we look for techniques to update RCAS efficiently and maintain its robust query performance.

Challenge 4: Scalable CAS Indexing. The fourth challenge is scaling the RCAS index to large datasets. Since RCAS is a main-memory index and based on a memory-optimized trie (ART [LKN13]), RCAS does not scale to large datasets. Scaling our dynamic interleaving and RCAS index to large datasets is not trivial. First, extending RCAS to block storage devices is not straightforward since its nodes are small and not aligned with a page-structured storage layout. Second, the RCAS bulk-loading algorithm is limited by main-memory data structures

and algorithms that do not scale. For example, dynamically interleaving a set of composite keys that does not fit into memory is difficult because the dynamic interleaving is data-driven and to dynamically interleave a single key we need to consider all other keys. In addition, during bulk-loading we need to be careful how to manage the available memory if the data size exceeds the memory size.

CHAPTER 2

Contributions

This thesis makes the following contributions:

1. *(Dynamic Interleaving)* We introduce a novel interleaving scheme, the dynamic interleaving, that is rooted in a well-balanced interleaving of the paths and values of composite keys without prioritizing either dimension.
2. *(Robust Content-and-Structure (RCAS) Index)* We propose a new in-memory, trie-based CAS index, called the *Robust CAS (RCAS)* index, that stores dynamically-interleaved composite keys.
3. *(Supporting Insertions in RCAS)* We develop several techniques to incrementally insert new keys in the RCAS index that trade insertion and query performance.
4. *(Scalable RCAS⁺ Index)* We propose the RCAS⁺ index, a scalable version of the in-memory RCAS index that maintains RCAS's excellent CAS query performance and that scales to large datasets that do not fit into memory.

In this thesis we address the challenges from Section 1.3 by starting with an application scenario that illustrates the problem. Our solutions to each problem and their properties are studied and

elaborated analytically. We implement and experimentally evaluate our solutions to confirm our analytical results and we compare them to state-of-the-art competitors on different datasets. We provide the source code, datasets, and instructions how to reproduce our results online (see the experimental evaluation section of each paper).

The remainder of this chapter elaborates each contribution in detail.

2.1 Dynamic Interleaving

Our first contribution is a novel interleaving scheme, called *dynamic interleaving*, that offers a well-balanced integration of the paths and values of composite keys. Dynamic interleaving is rooted in the observation that paths and values often have long common prefixes and that statically interleaving at a common prefix leads to an ill-balanced interleaving that prioritizes one dimension over another. While discarding keys selectively during a search is not possible at a common prefix, the first byte following a longest common prefix allows exactly this: it distinguishes different data items and allows us to narrow down the set of keys that match a query. We call such a byte a *discriminative byte*. We define the discriminative byte $\mathsf{dsc}(K,D)$ of a set K of composite keys in dimension $D \in \{P,V\}$ as the first byte for which the keys differ in dimension D.

Example 2.1. *The discriminative path byte of the keys $\mathsf{K}^{1..7}$ in Table 1.1 is $\mathsf{dsc}(\mathsf{K}^{1..7},P) = 13$. All seven keys share the same longest common path prefix `/bom/item/ca` of length 12 and the first byte after the longest common path prefix is the discriminative path byte, since key $\mathsf{k}_1.P[13] = \mathtt{n}$ while $k.P[13] = \mathtt{r}$ for all keys $k \in \{\mathsf{k}_2,\dots,\mathsf{k}_7\}$. Likewise, the discriminative value byte is $\mathsf{dsc}(\mathsf{K}^{1..7},V) = 2$.*

Intuitively, dynamically interleaving a set of keys means to always interleave the shortest prefix that can distinguish data items in one dimension (e.g., the value dimension) with the corresponding shortest prefix in the other dimension (e.g., the path dimension). The shortest prefix that can distinguish data items in the respective dimension is the sequence of bytes from the first byte up to the discriminative byte. Consequently, the dynamic interleaving interleaves the paths and values of composite keys at their discriminative path and value bytes. Interleaving at the discriminative bytes means that at each interleaving step we make *progress* in both dimensions at the same time.

To dynamically interleave a set of keys, we propose a partitioning-based approach that alternatingly partitions the data in the path and value dimensions. We introduce the partitioning operator $\psi(K,D)$ that partitions a set K of composite keys based on their value at the discriminative byte in dimension D. That is, ψ groups all keys in K that have the same value at the discriminative byte $\mathsf{dsc}(K,D)$. Formally, ψ returns a set of partitions $\psi(K,D) = \{K_1,\ldots,K_m\}$ such that (i) all keys in K_i have the same value at $\mathsf{dsc}(K,D)$, (ii) no two keys from different partitions $K_i \neq K_j$ have the same value at $\mathsf{dsc}(K,D)$, and (iii) all keys in K are assigned to some partition K_i and no partition is empty.

Example 2.2. *We ψ-partition the keys $\mathsf{K}^{1..7}$ based on their value at their discriminative value byte, which is the second byte. Consequently, the ψ-partitioning is $\psi(\mathsf{K}^{1..7},V) = \{\mathsf{K}^{2,5,6,7},\mathsf{K}^{1},\mathsf{K}^{3,4}\}$. All keys in $\mathsf{K}^{2,5,6,7}$ have value `00` at the discriminative value byte. The only key in K^{1} has value `01` at the second value byte, and the two keys in $\mathsf{K}^{3,4}$ have value `03`.*

To compute the dynamic interleaving we start with the set of all keys and ψ-partition it in the value dimension (chosen arbitrarily). Each resulting partition is itself ψ-partitioned in the alternate dimension, i.e., the path dimension. We continue to alternatingly ψ-partition the data until a partition contains only a single key and can therefore not be partitioned further. Each time we ψ-partition the data, we record three pieces of information: (i) the longest common path prefix of the current set of keys, (ii) the longest common value prefix of the current set of keys, and (iii) the dimension in which the current set of keys is ψ-partitioned (we set the dimension to $\perp$ if the set cannot be further ψ-partitioned since it contains only one key).

Table 2.1: The dynamic interleaving of the composite keys in $\mathsf{K}^{1..7}$. The values at the discriminative bytes are written in bold.

Key k	Dynamic Interleaving of key k
k_2	((`00`,`/bom/item/ca`,V),(**`00`**,`r`,P),(**`a`**`biner$`,`00 F1`,$\perp$))
k_7	((`00`,`/bom/item/ca`,V),(**`00`**,`r`,P),(**`/`**`b`,ε,V),(**`0A`** `8C`,`umper$`,$\perp$))
k_5	((`00`,`/bom/item/ca`,V),(**`00`**,`r`,P),(**`/`**`b`,ε,V),(**`0B`** `4A`,`elt$`,$\perp$))
k_6	((`00`,`/bom/item/ca`,V),(**`00`**,`r`,P),(**`/`**`b`,ε,V),(**`0C`** `C2`,`rake$`,$\perp$))
k_1	((`00`,`/bom/item/ca`,V),(**`01`** `0E 50`,`noe$`,$\perp$))
k_3	((`00`,`/bom/item/ca`,V),(**`03`**`D3`,`r/battery$`,V),(**`5A`**,ε,$\perp$))
k_4	((`00`,`/bom/item/ca`,V),(**`03`**`D3`,`r/battery$`,V),(**`B0`**,ε,$\perp$))

Example 2.3. *We start with the set $\mathsf{K}^{1..7}$ and store its longest common value prefix `00`, longest common path prefix `/bom/item/ca`, and dimension V that is used to ψ-partition the data in a tuple (`00`,`/bom/item/ca`,V). After that, we repeat the same process for each partition in*

$\psi(\mathsf{K}^{1..7}, V) = \{\mathsf{K}^{2,5,6,7}, \mathsf{K}^{1}, \mathsf{K}^{3,4}\}$ *in the alternate dimension: the path dimension P. We illustrate this process for partition* $\mathsf{K}^{2,5,6,7}$. *First, we we interleave* $\mathsf{K}^{2,5,6,7}$*'s longest common path and value prefixes but remove its parent partition's, i.e.,* $\mathsf{K}^{1..7}$*'s, longest common prefixes since they were already interleaved in the previous step. Thus, we interleave strings* `00` *in the value dimension and* `r` *in the path dimension. Next, we ψ-partition* $\mathsf{K}^{2,5,6,7}$ *in dimension P. This process continues until all partitions are narrowed down to a single key. Table 2.1 shows the dynamic interleaving of all keys in* $\mathsf{K}^{1..7}$.

Compared to the static interleavings in Table 1.2, the dynamic interleavings in Table 2.1 are well-balanced and interleave paths and values in a natural way.

2.2 Robust Content-and-Structure (RCAS) Index

Our second contribution is a new CAS index, called the *Robust CAS (RCAS)* index, that utilizes the dynamic interleaving to achieve robust CAS query performance. We embed dynamically-interleaved keys in a trie structure since tries support the range and prefix searches efficiently that we need to implement the value and path predicates of CAS queries, respectively. Crucially, tries in combination with dynamically-interleaved keys allow us to simultaneously evaluate path and value predicates of CAS queries in a robust way.

Embedding dynamically-interleaved keys in a trie is a natural fit since the dynamic interleaving organizes keys according to their longest common prefixes, exactly like tries do. Figure 2.1 shows how the dynamically-interleaved keys in Table 2.1 are stored in the RCAS index. Each longest common prefix in the dynamic interleaving is stored as a single node in the RCAS index. Storing common prefixes only once reduces the storage overhead of the index. Each node stores a value and a path substring, and a dimension D that specifies in which dimension the data is partitioned (for leaf nodes the dimension is $\perp$). The value and path substrings in a node are the longest common prefixes of all descendants of the node. Leaf nodes store a set of references that point to the location of the data item in the database. On a technical level, RCAS is an in-memory index that is built on top of a memory-optimized trie (ART [LKN13]).

To efficiently answer a CAS query with the RCAS index we traverse the index depth first. At each node that we visit during the search, we evaluate a part of the path and value predicates, which allows us to prune subtrees early if at least one of the predicates has a low selectively. Starting from the root, the query's path and value predicates are matched against the root node's

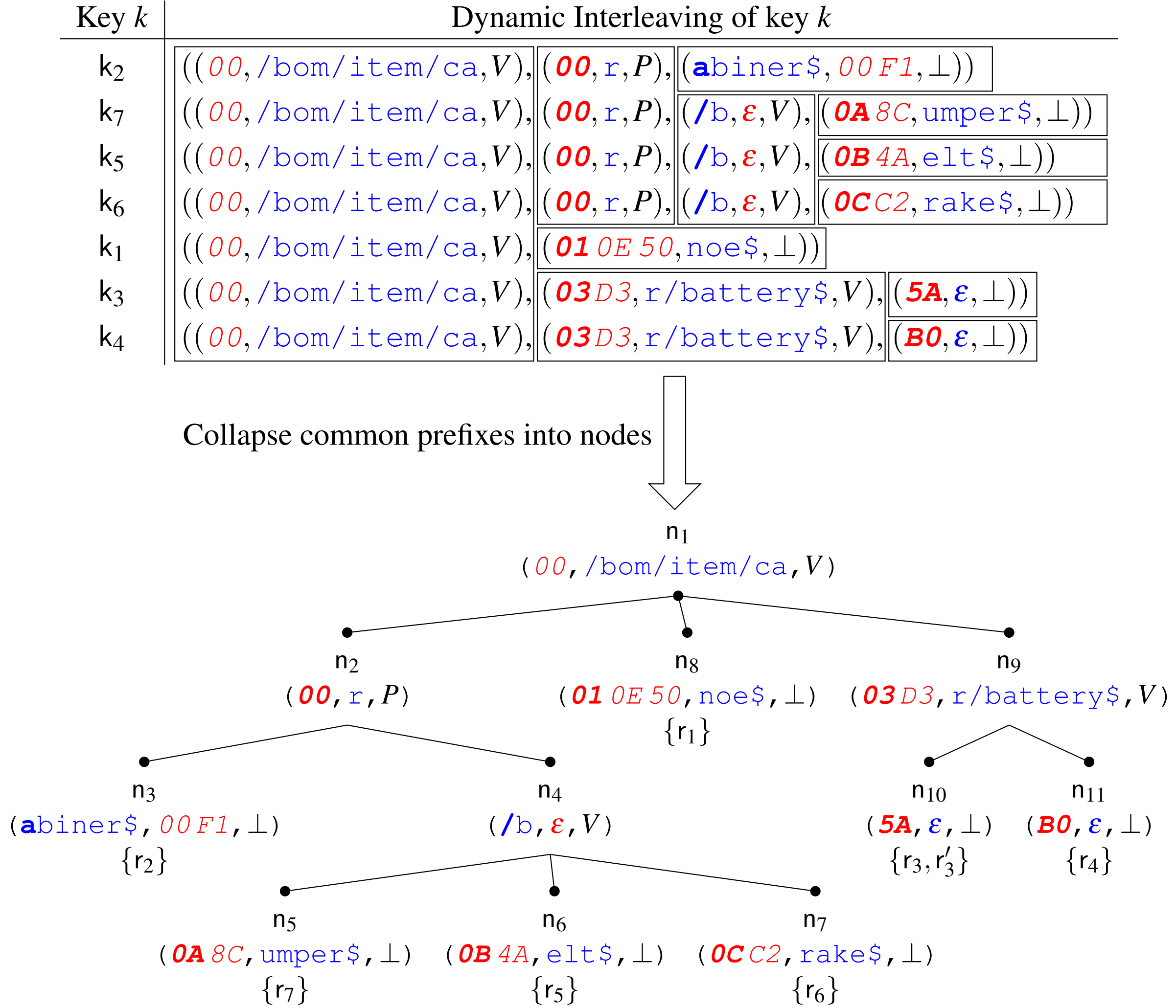


Figure 2.1: The dynamic interleaving is embedded in the trie-based RCAS index.

path and value substrings. As long as these substrings match, we descend to the appropriate children. In case the current node partitions the data in the value dimension we look at the next byte in the value predicate's lower and upper bound to decide what children we look at next (if any). Likewise, if the node partitions the data in the path dimension, we look at the next byte in the query path to find the appropriate child, or we descend to all children in case the next byte in the query path is the descendant axis or a wildcard.

Example 2.4. *We consider a CAS query with query path $q =$ `/bom/item//battery$` and a value range $[v_l, v_h]$ from $v_l = 10^5 =$ `00 01 86 A0` to $v_h = 5 \cdot 10^5 =$ `00 07 A1 20`. We show how this query is executed on the RCAS index depicted in Figure 2.1.*

- *We start at the root node* n_1 *and see that its value substring* `00` *matches the first bytes of the lower and upper bound in the value predicate. The path predicate matches the path substring* `/bom/item/ca`*, since they coincide on the prefix* `/bom/item` *and the descendant axis* `//` *in the query path matches the substring* `/ca`*. We descend to the appropriate children. Since* n_1 *is a value node* ($\mathsf{n}_1.D = V$)*, we look for all matching children whose value at the discriminative value byte is between* `01` *and* `07` *(these are the values at the second byte in the lower and upper bound on the range, respectively). Nodes* n_8 *and* n_9 *satisfy this condition.*

- *Node* n_8 *is a leaf. Neither its path nor value substrings satisfy the path and value predicates. Consequently, the search discards node* n_8 *(and its descendants – if there were any).*

- *Next we look at node* n_9*. Since the node's first value byte,* `03`*, is strictly larger (smaller) than the corresponding byte* `01` *(*`07`*) in the lower bound (upper bound), this node and all its descendants match the value predicate.* n_9*'s path substring matches the path predicate up to the very end. Consequently, we know that all keys contained in this subtree satisfy the CAS query and therefore we return all references in the subtree, i.e., references* $\{\mathsf{r}_3, \mathsf{r}'_3, \mathsf{r}_4\}$.

RCAS's robustness is rooted in its dynamic interleaving that alternatingly interleaves paths and values at their discriminative bytes. We show that RCAS has the lowest average query runtime among all interleaving-based tries.

To prove RCAS's robustness we propose a cost model and show that the cost over all queries is minimal when we use dynamically-interleaved keys. For simplicity, the cost model assumes that the index has a fixed height h and fanout o (making the index balanced). In addition, we assume that all the nodes on a level i of the index partition the node in dimension $D_i \in \{P,V\}$. Then the interleaving order can be described by a vector $\phi = (D_1, D_2, \ldots, D_h)$. Our RCAS index alternatingly partitions the data in dimension V and P, thus $\phi_{\text{DY}} = (V,P,V,P,\ldots)$. The c-order curve separates values and paths, e.g., $\phi = (V,V,\ldots,P,P)$. A query starts at the root and traverses the data structure to determine the answer set. In the case of range queries, more than one branch must be followed. A search follows a fraction of the outgoing branches o originating at this node. We call this the selectivity of a node (or just selectivity). We assume that every path node has a selectivity of ς_P and every value node has the selectivity of ς_V, where $0 \leq \varsigma_P, \varsigma_V \leq 1$. The cost of a CAS query is measured in the number of visited nodes during the search.

State-of-the-art CAS-indexes are not robust because they favor either path or value predicates. As a result they show a very good performance for one type of query but run into problems for different queries. We define *complementary queries* as two queries Q and Q' that have opposing

selectivities. That is, if Q has selectivities ς_P and ς_V, its complementary query Q' has selectivities $\varsigma'_P = \varsigma_V$ and $\varsigma'_V = \varsigma_P$. RCAS has the lowest cost for complementary queries among all interleaving-based approaches.

Theorem 2.1. *There is no interleaving ϕ that in total has a smaller cost than the dynamic interleaving ϕ_{DY} for complementary queries.*

Having the provably best average runtime for complementary queries means that the RCAS index is less sensitive to the individual selectivities of the path and value predicates than other indexes. Since for every query there exists also a complementary query, we can generalize the above result and show that it holds for all possible queries.

Theorem 2.2. *There is no interleaving ϕ that in total has a smaller cost than the dynamic interleaving ϕ_{DY} over all possible queries.*

2.3 Updating the RCAS Index

Our third contribution are methods to update the RCAS index that explore the trade-off between update performance and query performance. Updating the RCAS index is difficult because inserting or deleting a single key can affect the dynamic interleaving of other keys. Indeed, in the worst case, inserting or deleting a single key can change the dynamic interleaving of *all* other keys and, as a consequence, a single update can affect large parts of the RCAS index.

We show that not every insertion or deletion is expensive: three cases can occur during an index update and only one of them is expensive. In the remainder we focus on insertion; deletion is analogous. In the first insertion case, a duplicate key is inserted and in this case we only need to add another reference to the corresponding leaf node in RCAS. The second case occurs when the key to be inserted deviates from the keys in the index, but it does so at the very end of the trie structure. In this case, we need to add a new leaf node to RCAS. The first and second case are inexpensive since the main cost is traversing the index. The third case is the most difficult one and occurs when the key to be inserted differs from a node in the path and/or value dimension. This means, the new key shifts the position of a discriminative byte and this invalidates the dynamic interleaving of all keys that are located in the subtree rooted in the node where the mismatch occurred.

Example 2.5. *We insert the key* (`/bom/item/cassette$`, `00 00 AB 12`) *with reference* r_{10} *into the RCAS index in Figure 2.1. The insertion proceeds from the root node to node* n_2 *and finds that there is a mismatch since the first* `s` *in the key's path differs from the* `r` *in* n_2*'s path substring. This invalidates the dynamic interleaving of all keys rooted in* n_2*'s subtree.*

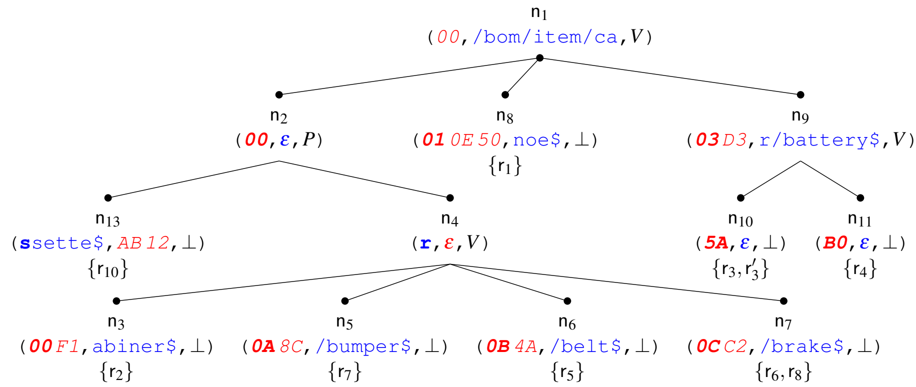
To handle the third insertion case we must restructure the RCAS index and we propose two techniques to do just that, called *strict restructuring* and *lazy restructuring*. Strict restructuring optimizes for query performance at the expense of insertion performance, while lazy restructuring trades query performance for better insertion performance.

Strict restructuring preserves the dynamic interleaving by (i) collecting all keys whose dynamic interleaving gets invalidated by inserting the new key, (ii) recomputing their dynamic interleaving, and (iii) replacing the subtree rooted in the node where the mismatch occurred with a newly created subtree that reflects the new dynamic interleaving. The cost of this approach depends on the size of the subtree that must be recreated. In the worst case, if there is a mismatch in the root node, this effectively means rebuilding the entire RCAS index.

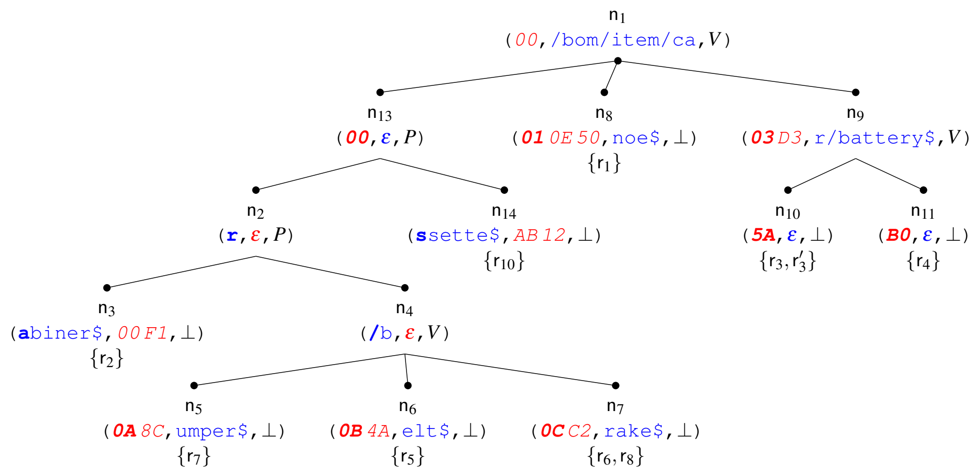
Example 2.6. *We continue the previous example. Since there is a mismatch in node* n_2*, strict restructuring rebuilds the entire subtree rooted in* n_2*. Figure 2.2a shows the RCAS index after inserting the key using strict restructuring.*

Lazy restructuring opts for a more efficient insertion method that only adds exactly two new nodes to the RCAS index. The basic idea is to add a new intermediate node that is able to successfully distinguish its children: the node where the mismatch happened and a new sibling that represents the newly inserted key. While efficient, lazy restructuring does not preserve the dynamic interleaving of paths and values at their discriminative bytes. Inserting a key with lazy restructuring introduces small irregularities that are limited to the dynamic interleaving of the keys in the subtree where the mismatch occurred. These irregularities slowly *separate* (rather than *interleave*) paths and values if insertions repeatedly force the algorithm to split the same subtree in the same dimension. On the other hand, lazy restructuring can also repair itself when an insertion forces the algorithm to split in the opposite dimension. In our experimental evaluation, lazy restructuring proves to be fast and to lead to good query performance.

Example 2.7. *We insert the same key as in the previous example using lazy restructuring, see Figure 2.2b. Since the mismatch occurred in* n_2*, we add a new node* n_{13} *above* n_2 *and a new sibling node* n_{14}*. The new parent node* n_{13} *contains all bytes that are shared between the inserted*



(a) Inserting a key with the strict restructuring method.



(b) Inserting a new key with lazy restructuring.

Figure 2.2: Comparison of strict and lazy restructuring.

key and node n_2*. It partitions the keys in the path dimension since there was only a path mismatch between the key and* n_2*. Node* n_{13} *and its child* n_2 *partition the data both in the path dimension (*$n.D = P$*) and therefore violate the strictly alternating pattern.*

In addition to strict and lazy restructuring we propose to use an auxiliary RCAS index to further improve the update performance. The idea is to apply the inexpensive Case 1 and 2 insertions directly on the (main) RCAS index and redirect the expensive Case 3 insertions to a smaller auxiliary RCAS index, where restructuring is cheaper. The auxiliary index is periodically merged back into the main RCAS index before it grows too big. Queries now have to traverse two index structures, but their combined size is comparable to the RCAS index if no auxiliary index was used, and therefore query performance does not degrade noticeably.

2.4 Scalable RCAS$^+$ Index

Our fourth contribution is to scale the RCAS index to big datasets. Many applications need quick access to large amounts of semi-structured hierarchical data, for which scalable indexing solutions are essential. For example, we collaborate with the Software Heritage (SWH) project,[1] which aims to collect and preserve *all* publicly-available software source code. The SWH archive has become the world's largest archive of its kind [ACZ18, DCZ17] and it includes code from popular forges like GitHub, GitLab, Bitbucket, etc. The SWH archive is big: it archives hundreds of millions of repositories, billions of commits, and tens of billions of source code files. Currently, the SWH archive implements only rudimentary search features, like keyword searches in the names and URLs of software repositories, but more advanced queries, like CAS queries, are not supported. We use RCAS to index the *revisions* (i.e., commits) in the SWH archive. A revision captures what is commonly referred to as a "commit" in modern version control systems. A revision references the entire source code tree of a software project at commit time, points to previous revision(s) – allowing to compute source code "diffs" between commits – and is associated to metadata such as commit time and author. We index for each revision in the SWH archive its commit time and its diff, i.e., what files are modified (added/changed/deleted). This allows, e.g., software researchers to filter and later analyze revisions that modify certain files in a given time frame. If software researchers want to perform a security audit of open-source software, they can use the RCAS index to look for revisions that modify cryptographic routines that are stored in certain directories (e.g., the `crypto` folder in a repository). In addition, they can limit the scope of the audit to recent revisions from, e.g., the past six months.

While RCAS is robust, it does not scale to large datasets like the SWH archive because of two issues. First, RCAS is an in-memory index based on a memory-optimized trie (ART [LKN13]), and second bulk-loading RCAS is limited by main-memory data structures and algorithms that do not scale.

To address these issues we propose the scalable RCAS$^+$ index that is not constrained by the available memory. While RCAS is optimized for main-memory storage, we optimize RCAS$^+$ for disk storage.[2] In addition, we develop a scalable algorithm that builds RCAS$^+$ while, at the same time, dynamically interleaving the keys. Building the index and dynamically interleaving the keys at the same time amortizes the cost of the interleaving. A salient property of the bulk-

[1] https://www.softwareheritage.org

[2] We use the term disk to refer to any generic block-storage device (HDD, SSD, etc.)

loading algorithm is that it requires only little memory, but scales nicely with the size of the available memory.

The algorithm is partitioning-based and proceeds as follows. Initially, all keys belong to one *partition* that may exceed the size of the available memory. A partition is a tuple $(g_P, g_V, \mathsf{mptr}, \mathsf{fptr})$ and stores a set of composite keys along with important meta-information, namely the discriminative path byte g_P and the discriminative value byte g_V. A key is stored either in memory or on disk. mptr and fptr are pointers to keys in memory and on disk, respectively. The bulk-loading algorithm works top-down. The first partition that contains all keys, called the root partition, is ψ-partitioned, creating a set of partitions. Each partition is recursively processed and in each step of the recursion we add one node to the RCAS$^+$ index that interleaves the longest common path and value prefixes of the current partition. The recursion stops when a partition becomes small enough to fit on a disk page. We propose five techniques that make building RCAS$^+$ at scale feasible in terms of, respectively, CPU, memory, and disk usage. We outline the five techniques in the following.

Node Clustering. The nodes in the memory-optimized RCAS index are typically small and do not fill a disk page. Mapping each node to its own page on disk is wasteful and would unnecessarily increase query runtime because many pages have to be fetched from disk during query evaluation. Hence, in RCAS$^+$ we use *node clustering* to align small nodes on block-based storage devices by grouping the nodes on pages. We use a greedy node-clustering algorithm [KM06a] that groups nodes bottom-up (i.e., from the leaves to the root). As soon as the algorithm clusters a set of nodes, we write the nodes to disk and release them from memory. Because of its greedy nature, the algorithm has a small memory footprint, which allows us to use the remaining memory to improve the performance of the bulk-loading algorithm.

Depth-first bulk-loading. Our bulk-loading algorithm is partitioning-based and creates nodes top-down, but the node-clustering algorithm groups the nodes into pages bottom-up, starting at the leaf nodes. Depth-first bulk-loading guarantees that only a small number of nodes have to be kept in memory before they can be written to disk by node clustering. Depth-first bulk-loading in combination with node clustering guarantees that the bulk-loading algorithm has a negligible memory footprint of $O(h \times B)$, where h is the height of RCAS$^+$ and B is the page size. Consequently, we can use the remaining memory to speed up bulk-loading with front-loading.

Front-Loading. Depth-first bulk-loading together with node clustering minimizes the memory footprint and we propose *front-loading* to optimally use the remaining memory. For large datasets that exceed the size of the available memory we need to decide what part of the data

is kept in memory and what is stored on disk. Our bulk-loading algorithm uses partitions that keep some keys in memory (mptr) and some on disk (fptr). When we start bulk-loading RCAS$^+$, we store as many keys as possible in the root partition's mptr and store the remaining keys in its fptr. When we break up the root partition into a set of partitions $\{K_1, \ldots, K_m\}$ we need to decide how we re-allocate the memory used by the root partition to the new partitions $K_1, \ldots, K_m$. Front-loading re-uses the memory for those partitions that are processed next by the bulk-loading algorithm. Since we process the partitions $K_1, \ldots, K_m$ in that order, we keep the first few partitions entirely memory-resident, followed by up to one hybrid partition that is stored partially in memory and partially on disk, and all remaining partitions are entirely disk-resident. Memory-resident partitions can be processed entirely in memory. After processing those partitions recursively, their memory is released and can be reused when we process the hybrid partition next (and the same applies for the following disk-resident partitions). We show that front-loading is the optimal memory placement strategy during bulk-loading.

Proactive Partitioning. To ψ-partition a partition K we first need to compute its discriminative bytes g_P and g_V. Thus, in general we need two scans over K to ψ-partition it: the first scan determines K's discriminative byte and the second scan assigns the keys to their partitions. Scanning every partition twice is expensive, especially when the partitions are disk-resident. Proactive Partitioning exploits that the data is partitioned hierarchically and pre-computes the discriminative bytes of new partitions at the next level while partitioning a set of keys. By the time the new partitions are being partitioned we have already computed the discriminative bytes. Consequently, only the root partition needs to be scanned twice, and every subsequent partition needs to be scanned only once.

Lazy Interleaving. Unlike static interleaving schemes that are typically cheap to compute (e.g., constant time per key for z-order), the dynamic interleaving is more expensive to compute. To reduce the cost of the dynamic interleaving we propose a lazy version of the dynamic interleaving, called lazy interleaving. The basic idea is to interleave only the prefixes of the keys without interleaving their suffixes. With lazy interleaving we stop the hierarchical partitioning when a partition fits on a single disk page and store the remaining suffixes of the keys un-interleaved on a leaf page. During subsequent searches all suffixes in a leaf page must be checked with a linear scan if they match a given CAS query. Since RCAS$^+$ is not designed for point queries that select only a single key in a leaf node, the overhead of scanning all keys in a leaf is acceptable. We experimentally show that lazy interleaving speeds up bulk-loading by a factor of 20 without compromising query performance.

CHAPTER 3

Thesis Roadmap

This thesis is based on the following collection of papers. The papers are reprinted in the appendix and a bibliography for all papers is given at the end of the thesis.

Appendix A Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data

Kevin Wellenzohn, Michael H. Böhlen, Sven Helmer. "Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data", in *PVLDB*, 13(10): pages 1641–1653, 2020.
doi:10.14778/3401960.3401963

Kevin Wellenzohn, Michael H. Böhlen, Sven Helmer. "Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data (Extended Version)", *Technical Report*, *CoRR*, 14 pages, 2020.
https://arxiv.org/abs/2006.05134

Appendix B Inserting Keys into the Robust Content-and-Structure (RCAS) Index

Kevin Wellenzohn, Luka Popovic, Michael H. Böhlen, Sven Helmer. "Inserting

Keys into the Robust Content-and-Structure (RCAS) Index", in *ADBIS*, pages 121–135, 2021.
doi:10.1007/978-3-030-82472-3_10

Appendix C Scalable Content-and-Structure Indexing

Kevin Wellenzohn, Michael H. Böhlen, Sven Helmer, Antoine Pietri, Stefano Zacchiroli. "Scalable Content-and-Structure Indexing", [*ready for submission*]

The first paper, Appendix A, introduces the first two contributions of this thesis: the dynamic interleaving scheme (see Section 2.1) and the trie-based RCAS index (see Section 2.2).

The second paper, Appendix B, covers the third contribution (see Section 2.3): it describes why updating the RCAS index with its dynamic interleaving is difficult and proposes several restructuring techniques that trade update and query performance.

The third paper, Appendix C, develops our fourth contribution (see Section 2.4): the paper proposes the scalable RCAS$^+$ index that scales the RCAS index to large datasets. We show-case RCAS$^+$'s scalability by indexing data from the Software Heritage archive, which is the world's largest archive of publicly available source-code.

The three papers are connected to each other. In particular, the second and third paper (Appendixes B and C) are based on the ideas proposed in the first paper (Appendix A). The second and third paper were written simultaneously and are independent. Because the papers are connected, there is limited overlap between the papers as they share some common definitions, examples, etc. The second paper re-uses the running example from the first paper. Therefore, Figures A.1 and A.5, and Table A.1 re-appear in Appendix B. The background sections of the second and third paper (Sections B.2 and C.3) re-introduce some concepts defined already in Appendix A. For example, the definitions of the discriminative byte (Definition C.2) and the ψ-partitioning (Definition C.3) are based on the corresponding definitions in Appendix A, though they are rephrased and shortened.

In the third paper we use a different syntax for the descendant axis with respect to the first paper. For example, in the first paper we write `/bom//battery` and the equivalent query path with the syntax from the third paper is `/bom/**/battery`. The third paper adopts this shell-like syntax for the descendant axis because the targeted use case is to match files in a hierarchical source code archive and the users of this system are likely more familiar with a shell-link syntax.

CHAPTER 4

Conclusion

4.1 Limitations

Query Formulation The RCAS and RCAS$^+$ indexes are designed to efficiently answer CAS queries on semi-structured, hierarchical data. CAS queries are not the only way to query such data. Twig-pattern queries [BKH$^+$17] define tree-shaped patterns that are matched against the database. Like CAS queries, twig queries can contain parent-child and ancestor-descendant relationships (among others). Indeed, CAS queries can be seen as a building block for twig queries, since twig queries can contain CAS queries as subqueries (e.g., a root-to-leaf path in a twig query can be a CAS query). As a result, RCAS and RCAS$^+$ can be used as a fast access method for twig queries. CAS and twig queries are a form of structured queries that are useful when users know the (rough) structure of the data. In contrast, unstructured queries are used to explore new datasets. Users are accustomed to search engines like Google that provide access to vast datasets through unstructured queries in the form of keyword searches. Unstructured queries give laymen a simple and intuitive search interface, but the problem with such types of queries is that they often return unrelated search results. CAS queries are more complicated to formulate since a

user needs to decide what a good query path looks like, where to put possible wildcards, etc., but this flexibility allows users to formulate more specific queries.

Placement of Descendant Axis RCAS and RCAS+ struggle with CAS queries that have a descendant axis close to the beginning of the query path, e.g., `//battery` or `/bom//brake`. RCAS and RCAS+ use prefix searches to answer the path predicates, but to answer, e.g., `//battery` efficiently we would need *suffix* searches instead of prefix searches. When our CAS indexes evaluate this path predicate they may have to traverse a large part of the tree because as soon the query executor encounters the descendant axis it cannot use the path predicate anymore to prune the search space during query evaluation. Normally, when the query path and the current node's path substring do not match, the query executor can discard the node and its subtree because no path suffix can match the query path anymore. This is no longer true after the query executor encounters the first descendant axis because even if the query path and the current node's path substring do not match, it is still possible that the descendant axis skips over the mismatch and the query path's suffix matches the remainder of the data. Since the path predicate can no longer be used to prune subtrees during query evaluation, the query executor can from here onwards only rely on the value predicate for pruning. Depending on the selectivity of the value predicate, this can be expensive.

There are several ways how to approach this problem. A solution inspired by similar techniques in information retrieval is to reorder the path labels such that the descendant axis is always at the end of the query path [MRS08]. For example, if we know that queries typically start with the descendant axis and look for example like `//battery`, it is more efficient to reorder the path labels and evaluate the query path `battery//` instead (in essence, we turn a suffix search into a prefix search that we can evaluate efficiently). This means that we have to reorder the paths in the index as well: instead of storing a path `/bom/item/car/battery` we must store the path `battery/car/item/bom/`. The disadvantage with this approach is the increased storage requirements, because we need to store the data twice: once with forward paths and once with backward paths. There exist similar techniques for when the descendant axis is at an arbitrary position in the query path, see [MRS08]. If the query workload is known in advance, the user can decide whether to store forward and/or backward paths.

Index Updates We proposed two restructuring techniques to update the RCAS index when the positions of the discriminative bytes change: strict and lazy restructuring. Strict restructuring

is slow but it preserves the dynamic interleaving that makes the index robust. In contrast, lazy restructuring is fast but it does not preserve the dynamic interleaving. Experimentally, we show that lazy interleaving ensures robust CAS query performance despite introducing small irregularities in the dynamic interleaving. However, analytically, we cannot say how much the dynamic interleaving degrades over time with lazy restructuring. Therefore, a fast restructuring technique that provably maintains robust query performance is still missing.

4.2 Summary

In this thesis, we develop new index structures to efficiently evaluate Content-and-Structure (CAS) queries on big semi-structured, hierarchical data. CAS queries consist of two predicates that select data items based on (i) their value for some attribute and (ii) their location in the hierarchical structure of the data. We look for two qualities in a CAS index: *robustness* and *scalability*. Robustness means that in the absence of any information about the query workload, an index optimizes the average query performance over all queries. A scalable CAS index is not constrained by the size of the available memory, and can be bulk-loaded and updated efficiently to keep up with the influx of large amounts of data.

We observe that to provide robust CAS query performance an access method must integrate the content and structure of the data in a well-balanced way such that the two predicates of CAS queries can be evaluated simultaneously. Consequently, we propose a novel interleaving scheme, called *dynamic interleaving*, that interleaves the paths and values of data items in a well-balanced way without prioritizing either dimension of the data. Dynamic interleaving achieves this by interleaving the binary representation of the paths and values at their first byte after their longest common prefix, known as the *discriminative byte*. Interleaving at a common prefix does not partition the data, but interleaving at the first byte after the longest common prefix does exactly this: it partitions the data and helps to prune the search space during a query.

We propose the *Robust CAS* (RCAS) index that stores dynamically interleaved keys to offer robust CAS query performance. The RCAS index is kept in main memory and is based on a memory-optimized trie (ART [LKN13]). Its trie structure allows RCAS to answer the value and path predicates of CAS queries through a mix of range and prefix searches.

We develop several techniques to update the RCAS index and its dynamic interleaving when new data is inserted. The problem is that inserting new keys into RCAS can change the dynamic

interleaving of existing keys in RCAS. Thus, there exists a trade-off between query runtime and update cost: if we want to preserve the dynamic interleaving that enables RCAS's robust query performance, we may have to restructure large parts of the index when new data inserted. We develop two algorithms that optimize for query performance and update performance, respectively. Additionally, we explore the idea of using differential files to update RCAS more efficiently.

Lastly, we propose the scalable RCAS$^+$ index that, unlike RCAS, is not constrained by the size of the available memory because we store RCAS$^+$ on disk. Additionally, we develop a new bulk-loading algorithm for RCAS$^+$. This algorithm optimizes the CPU, memory, and disk usage, respectively, making it possible to build RCAS$^+$ for large datasets.

In summary, in this thesis we propose a CAS index that offers robust query performance through a novel dynamic interleaving scheme and that scales to large datasets.

4.3 Future Work

Future work points in several directions. First, in this thesis, we consider two-dimensional keys that consist of a path and a value dimension. It would be interesting to generalize our solutions to multi-dimensional data to cover use cases where efficient access to the structure of the data and more than one content attribute is needed. That is, the composite keys would consist of a path dimension that represents the data item's location in the hierarchical structure and one or more values that represent the content of the data item. With such a solution we could, for example, index for each revision in the SWH archive the paths of the files that are modified, the commit time, and the commit author.

We have shown how to update the in-memory RCAS index and how to scale it to large datasets using our disk-resident RCAS$^+$ index. The disk-resident RCAS$^+$ index can be efficiently bulk-loaded, but it does not support updates yet. To do so we could adapt the update routines that we developed for RCAS in Appendix B. The problem with this approach is that in-place updates to disk-resident indexes are a known scalability bottleneck [OCGO96]. Therefore, we opt for out-of-place updates as pioneered by Log-Structured Merge (LSM) trees [OCGO96]. We plan to combine an in-memory RCAS index with a series of disk-resident RCAS$^+$ indexes that grow exponentially in size. In-place insertions are accumulated in RCAS and when the index grows too big, we apply all insertions at once using bulk-loading to create a read-optimized immutable RCAS$^+$ index on disk. Bulk-loading can be performed in the background such that updates to

the in-memory RCAS index do not stall, which makes sure that we can keep up with the influx of new data.

Currently, our solutions are stand-alone implementations, but it would be interesting to integrate them into actual database systems. This would bring a new set of challenges as we would have to integrate our access methods with the query optimizer, query executor, storage engine, etc.

Part II

Publications

APPENDIX A

Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data

Reprinted from:

K. Wellenzohn, M. H. Böhlen, S. Helmer. "Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data", in *PVLDB*, 13(10): pages 1641–1653, 2020. doi:10.14778/3401960.3401963

and the corresponding technical report:

K. Wellenzohn, M. H. Böhlen, S. Helmer. "Dynamic Interleaving of Content and Structure for Robust Indexing of Semi-Structured Hierarchical Data (Extended Version)", *Technical Report CoRR*, 14 pages, 2020. https://arxiv.org/abs/2006.05134

Abstract

We propose a robust index for semi-structured hierarchical data that supports content-and-structure (CAS) queries specified by path and value predicates. At the heart of our approach is a novel dynamic interleaving scheme that merges the path and value dimensions of composite keys in a balanced way. We store these keys in our trie-based Robust Content-And-Structure index, which efficiently supports a wide range of CAS queries, including queries with wildcards and descendant axes. Additionally, we show important properties of our scheme, such as robustness against varying selectivities, and demonstrate improvements of up to two orders of magnitude over existing approaches in our experimental evaluation.

A.1 Introduction

A lot of the data in business and engineering applications is semi-structured and inherently hierarchical. Typical examples are bills of materials (BOMs) [BFF$^+$15], enterprise asset hierarchies [FBK$^+$13], and enterprise resource planning applications [FBK$^+$15]. A common type of queries on such data are content-and-structure (CAS) queries [MHSB15], containing a *value predicate* on the *content* of some attribute and a *path predicate* on the location of this attribute in the *hierarchical structure*.

As real-world BOMs grow to tens of millions of nodes [FBK$^+$13], we need dedicated CAS access methods to support the efficient processing of CAS queries. Existing CAS indexes often lead to large intermediate results, since they either build separate indexes for, respectively, content and structure [MHSB15] or prioritize one dimension over the other (i.e., content over structure or vice versa) [Apa20, CSF$^+$01, STR$^+$15]. We propose a *well-balanced integration* of paths and values in a single index that provides *robust* performance for CAS queries, meaning that the index prioritizes neither paths nor values.

We achieve the balanced integration of the path and value dimension with composite keys that *interleave* the bytes of a path and a value. Interleaving is a well-known technique applied to multidimensional keys, for instance Nishimura et al. look at a family of bit-merging functions [NY17] that include the *c*-order [NY17] and the *z*-order [Mor66, OM84] space-filling curves. Applying space-filling curves on paths and values is subtle, though, and can result in poor query performance because of varying key length, different domain sizes, and the skew of the data. The

z-order curve, for example, produces a poorly balanced partitioning of the data if the data contains long common prefixes [Mar99]. The paths in a hierarchical structure exhibit this property: they have, by their very nature, long common prefixes. The issue with common prefixes is that they do not help to partition the data, since they are the same for all data items. However, the first byte following a longest common prefix does exactly this: it distinguishes different data items. We call such a byte a *discriminative byte*. The distribution of discriminative path and value bytes in an interleaved key determines the order in which an index partitions the data and, consequently, how efficiently queries can be evaluated. The z-order of a composite key often clusters the discriminative path and value bytes, instead of interleaving them. This leads to one dimension—the one whose discriminative bytes appear first—to be prioritized over the other, precluding robust query performance.

We develop a *dynamic interleaving* scheme that interleaves the discriminative bytes of paths and values in an alternating way. This leads to a well-balanced partitioning of the data with a robust query performance. Our dynamic interleaving is *data-driven* since the positions of the discriminative bytes depend on the distribution of the data. We use the dynamic interleaving to define the *Robust Content-and-Structure (RCAS)* index for semi-structured hierarchical data. We build our RCAS index as an in-memory trie data-structure [LKN13] to efficiently support the basic search methods for CAS queries: *range searches* and *prefix searches*. Range searches enable value predicates that are expressed as a value range and prefix searches allow for path predicates that contain wildcards and descendant axes. Crucially, tries in combination with dynamically interleaved keys allow us to efficiently evaluate path and value predicates simultaneously. We provide an efficient bulk-loading algorithm for RCAS that scales linearly with the size of the dataset. Incremental insertions and deletions are not supported.

Our main contributions can be summarized as follows:

- We develop a *dynamic interleaving* scheme to interleave paths and values in an alternating way using the concept of *discriminative bytes*. We show how to compute this interleaving by partitioning the data. We prove that our dynamic interleaving is robust against varying selectivities (Section A.5).
- We propose the in-memory, trie-based *Robust Content-and-Structure (RCAS) index* for semi-structured hierarchical data. The RCAS achieves its robust query performance by a well-balanced integration of paths and values via our dynamic interleaving scheme (Section A.6).

- Our RCAS index supports a broad spectrum of CAS queries that include wildcards and the descendant axis. We show how to evaluate CAS queries through a combination of range and prefix searches on the trie-based structure of the RCAS index (Section A.6.4).

- An exhaustive experimental evaluation with real-world and synthetic datasets shows that RCAS delivers robust query performance. We get improvements of up to two orders of magnitude over existing approaches (Section A.7).

A.2 Running Example

We consider a company that stores the bills of materials (BOMs) of its products. BOMs represent the hierarchical assembly of components to final products. Each BOM node is stored as a tuple in a relational table, which is common for hierarchies, see, e.g., SAP's storage of BOMs [BFF+15, FBK+13, FBK+15] and the Software Heritage Archive [DCZ17, PSZ20]. A CAS index is used to efficiently answer queries on the structure (location of a node in the hierarchy) and the content of an attribute (e.g., the weight or capacity). The paths of all nodes in the BOM that have a value for the indexed attribute as well as the value itself are indexed in the CAS index. The index is read-only, updated offline, and kept in main memory.

Figure A.1 shows the hierarchical representation of a BOM for three products. The components of each product are organized under an `item` node. Components can have attributes to record additional information, e.g., the weight of a battery. Attributes are represented by special nodes that are prefixed with an `@` and that have an additional value. For example, the weight of the rightmost battery is 250'714 grams and its capacity is 80000 Wh.

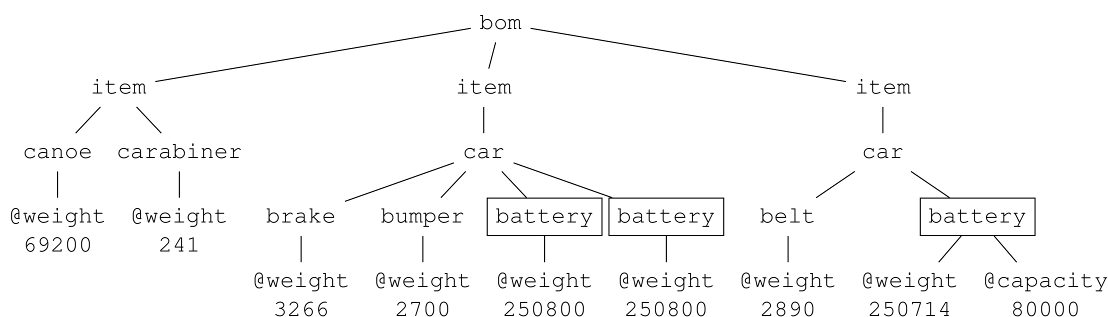


Figure A.1: Example of a bill of materials (BOM).

Next, we look at an example CAS query. We roughly follow the syntax of query languages for semistructured data, such as XQuery [KCK+03] or JSONiq [FF13], utilizing simple FLWOR expressions.

Example A.1. *To reduce the weight of cars we look for all heavy car parts, i.e., parts weighing at least 50 kilograms ("//" matches a node and all its descendants in a hierarchical structure):*

```
Q:  for $c in /bom/item/car//
    where $c/@weight >= 50000
    return $c
```

The answer to query Q are the three framed nodes in Figure A.1. Our goal is an index that guides us as quickly as possible to these nodes. Indexes on either paths or values do not perform well. An index built for only the values of weight nodes also accesses the node for the canoe. A purely structural index for the paths additionally has to look at the weight of other car parts. Our RCAS index considers values and paths together to get a good query performance.

A.3 Related Work

We begin with a review of existing CAS indexes [CSF+01, KKNR04, LAAE06, MHSB15, STR+15]. IndexFabric [CSF+01] prioritizes the structure of the data over its values. It stores the concatenated path and value of a key in a disk-optimized PATRICIA trie [Mor68] that supports incremental updates (i.e., inserts and deletes). IndexFabric does not offer robust CAS query performance since a CAS query must fully evaluate a query's path predicate before it can evaluate its value predicate. This leads to large intermediate results if the path predicate is not selective.

The hierarchical database system Apache Jackrabbit Oak [Apa20] implements a CAS index that prioritizes values over paths. Oak indexes (value v, path p)-pairs in a DataGuide-like index that supports updates. For each value v, Oak stores a DataGuide [GW97] of all paths p that have this particular value v. Query performance is poor if the value predicate is not selective because the system must search many DataGuides.

The CAS index of Microsoft Azure's DocumentDB (now Cosmos DB) concatenates paths and values [STR+15] and stores the result in a Bw-tree [LLS13] that supports updates. Depending on the expected query type(s) (point or range queries), the system either stores forward keys (e.g., `/a/b/c`) or reverse keys (e.g., `c/b/a`). To reduce the space requirements, forward and reverse

keys are split into trigrams (three consecutive node labels). During the evaluation of a CAS query these trigrams must be joined and matched against the query, which is slow. Moreover, choosing forward or reverse keys prioritizes structure over values or vice-versa.

Mathis et al. [MHSB15] propose a CAS index that consists of two index structures: a B-tree to index the values and a structural summary (e.g., a DataGuide [GW97]) to index the structure of the data. The DataGuide assigns an identifier (termed PCR) to each distinct path in the documents. The B-tree stores the values along with the PCRs. The path and value predicates of a CAS query are independently evaluated on the DataGuide and the B-tree, and the intermediate results are joined on the PCR. This is expensive if the intermediate results are large (i.e., at least one predicate is not selective) but the final result is small. Updates are supported and are executed on the B-tree as well as the DataGuide.

Kaushik et al. [KKNR04] present an approach that joins inverted lists for answering CAS queries. They use a 1-index [MS99] to evaluate path predicates and B-trees to evaluate value predicates. This approach evaluates both predicates independently and exhibits the same problems as [MHSB15]. Updates are not discussed.

FLUX [LAAE06] computes a Bloom filter for each path into which its labels are hashed and stores these Bloom filters along with the values in a B-tree. Query evaluation proceeds as follows. The value predicate is matched on the B-tree and for each matched value the corresponding Bloom filter C is compared to a Bloom filter Q built for the query path. If each bit that is set in Q is also set in C, the path is a possible match that needs to be double-checked through database accesses. Value predicates that are not selective produce large intermediate results. Updates are not discussed.

Some document databases for semi-structured, hierarchical data (MongoDB [Mon20], CouchDB [Cou20], and AsterixDB [AAA^{+}14]) use pure value indexes (e.g., standard B-trees) that index the content of documents but not their structure. They create an index on a predefined path (e.g., `/person/name`) and only index the corresponding values. They cannot answer CAS queries with arbitrary path predicates.

Besides pure value indexes there are also pure structure indexes that focus on complex twig queries with different axes (ancestor, descendant, sibling, etc.). DeltaNI [FBK^{+}13] and Order Indexes [FBK^{+}17] are recent proposals in this area. Pure structure indexes cannot efficiently answer CAS queries that also include value predicates.

Our RCAS index integrates paths and values by interleaving them. This is similar to the bit-merging family of space-filling curves that combine the binary representation of a key's dimensions. We compare our approach to two representatives: the c-order curve [NY17] and the z-order curve [Mor66, OM84]. The c-order curve is obtained by concatenating dimensions, which prioritizes one of the dimensions. The selectivity of the predicate on the prioritized dimension determines the query performance. If it is high and the other selectivity is low, the c-order curve performs badly. The z-order curve is a space-filling curve that is used, among others, by UB-trees [RMF+00] and k-d tries [NS08, OM84, Sam06]. It is obtained by the bit-wise interleaving of dimensions. The z-order curve produces an unbalanced partitioning of the data with poor query performance if the data contains long common prefixes. Markl calls this the "puff-pastry effect" [Mar99] because the query performance deteriorates to that of a c-order curve that fully orders one dimension after another. The Variable UB-tree [Mar99] uses a pre-processing step to encode the data in such a way that the puff-pastry effect is avoided. The encoding is not prefix-preserving and cannot be used in our CAS index. We need prefix searches to evaluate path predicates. The c-order and z-order curves are *static* interleaving schemes that do not take the data distribution into account. Indexes based on static schemes can be updated efficiently since insertions and deletions do not affect existing interleavings. The Variable UB-tree does not support incremental updates [Mar99] since its encoding function adapts to the data distribution and must be recomputed whenever the data changes. Similarly, our data-driven dynamic interleaving does not support incremental updates since the position of the discriminative bytes may change when keys are inserted or deleted.

QUILTS [NY17] devises a static interleaving scheme for a specific query workload. Index updates, although not discussed, would work as for other static schemes (e.g., c- and z-order). Our dynamic interleaving adapts to the data distribution rather than a specific query workload. We do not optimize a specific workload but want to support a wide range of queries in a robust way, including ad-hoc queries.

A.4 Background

Composite Keys. We use composite keys that consist of a path dimension P and value dimension V to index attributes in hierarchical data. Neither paths nor values nor the combination of paths and values need to be unique in a database. Composite keys can be extracted from popular semi-structured hierarchical data formats, such as JSON and XML.

Definition A.1. *(*Composite Key*) A composite key k states that a node with path $k.P$ in a database has value $k.V$.*

Let $D \in \{P,V\}$ be the path or value dimension. We write $k.D$ to access k's path (if $D = P$) or value (if $D = V$). The value dimension can be of any primitive data type. In the remainder of this paper we use one byte ASCII characters for the path dimension and hexadecimal numbers for the value dimension.

Example A.2. *In our running example we index attribute `@weight`. Table A.1 shows the composite keys for the `@weight` attributes from the BOM in Figure A.1. Since only the `@weight` attribute is indexed, we omit the label `@weight` in the paths in Table A.1. The values of the `@weight` attribute are stored as 32 bit unsigned integers.*

The set of composite keys in our running example is denoted by $\mathsf{K}^{1..7} = \{\mathsf{k}_1, \mathsf{k}_2, \ldots, \mathsf{k}_7\}$, see Table A.1. We use a sans-serif font to refer to concrete values. Further, we use notation $\mathsf{K}^{2,5,6,7}$ to refer to $\{\mathsf{k}_2, \mathsf{k}_5, \mathsf{k}_6, \mathsf{k}_7\}$.

Table A.1: A set $\mathsf{K}^{1..7} = \{\mathsf{k}_1, \ldots, \mathsf{k}_7\}$ of composite keys. The values are stored as 32 bit unsigned integers.

	Path Dimension P	Value Dimension V
k_1	`/bom/item/canoe$`	00 01 0E 50
k_2	`/bom/item/carabiner$`	00 00 00 F1
k_3	`/bom/item/car/battery$`	00 03 D3 5A
k_4	`/bom/item/car/battery$`	00 03 D3 B0
k_5	`/bom/item/car/belt$`	00 00 0B 4A
k_6	`/bom/item/car/brake$`	00 00 0C C2
k_7	`/bom/item/car/bumper$`	00 00 0A 8C
	1 3 5 7 9 11 13 15 17 19 21 23	1 2 3 4

Querying. Content-and-structure (CAS) queries contain a path predicate and value predicate [MHSB15]. The path predicate is expressed as a query path q that may include `//` to match a node itself and all its descendants, and the wildcard $\star$ to match all of a node's children. The latter is useful for data integrated from sources using different terminology (e.g., `product` instead of `item` in Fig. A.1).

Definition A.2. *(*Query Path*) A query path q is denoted by $q = e_1\,\lambda_1\,e_2\,\lambda_2\,\ldots\,\lambda_{m-1}\,e_m$. Each e_i, $i \leq m$, is either the path separator `/` or the descendant-or-self axis `//` that matches zero to any number of descendants. The final path separator e_m is optional. λ_i, $i < m$, is either a label or the wildcard $\star$ that matches any label.*

Definition A.3. *(CAS Query) A CAS query $Q(q,\theta)$ consists of a query path q and a value predicate θ on an attribute A, where θ is a simple comparison $\theta = A\,\Theta\,v$ or a range comparison $\theta = v_l\,\Theta\,A\,\Theta'\,v_h$ where $\Theta,\Theta' \in \{=,<,>,\leq,\geq\}$. Let K be a set of composite keys. CAS query Q returns all composite keys $k \in K$ such that $k.P$ satisfies q and $k.V$ satisfies θ.*

Example A.3. *The CAS query from Section A.2 is expressed as $Q($`/bom/item/car//`$,$ `@weight` $\geq 50000)$ and returns all car parts weighing more than 50000 grams in Figure A.1. CAS query $Q($`/bom/*/car/battery`$,$ `@capacity` $= 80000)$ looks for all car batteries that have a capacity of 80kWh. The wildcard `*` matches any child of* `bom` *(only* `item` *children exist in our example).*

Representation of Keys. Paths and values are prefix-free byte strings as illustrated in Table A.1. To get prefix-free byte strings we append the end-of-string character (ASCII code `0x00`, here denoted by `$`) to each path. This guarantees that no path is prefix of another path. Fixed-length byte strings (e.g., 32 bit numbers) are prefix-free because of the fixed length.

Let s be a byte-string, then $\mathsf{len}(s)$ denotes the length of s and $s[i]$ denotes the i-th byte in s. The left-most byte of a byte-string is byte one. $s[i] = \varepsilon$ is the empty string if $i > \mathsf{len}(s)$. $s[i,j]$ denotes the substring of s from position i to j and $s[i,j] = \varepsilon$ if $i > j$.

Interleaving of Composite Keys. We integrate path $k.P$ and value $k.V$ of a key k by interleaving them. Figure A.2 shows various interleavings of key k_6 from Table A.1. Value bytes are underlined and shown in red, path bytes are shown in blue. The first two rows show the two c-order curves: path-value and value-path concatenation (I_{PV} and I_{VP}). The byte-wise interleaving I_{BW} in the third row interleaves one value byte with one path byte. Note that none of these interleavings is well-balanced. The byte-wise interleaving is not well-balanced, since all value-bytes are interleaved with parts of the common prefix of the paths (`/bom/item/ca`). In our experiments we use the surrogate-based extension proposed by Markl [Mar99] to more evenly interleave dimensions of different lengths (see Section A.7).

Approach	Interleaving of Key k_6
Path-Value Concatenation	`/bom/item/car/brake$` *00 00 0C C2*
Value-Path Concatenation	*00 00 0C C2* `/bom/item/car/brake$`
Byte-Wise Interleaving	*00* `/` *00* `b` *0C* `o` *C2* `m/item/car/brake$`

Figure A.2: Key k_6 is interleaved using different approaches

A.5 Dynamic Interleaving

Our dynamic interleaving is a *data-driven* approach to interleave the paths and values of a set of composite keys K. It adapts to the specific characteristics of paths and values, such as varying length, differing domain sizes, and the skew of the data. To this end, we consider the distribution of *discriminative bytes* in the indexed data.

Definition A.4. (Discriminative Byte) *The discriminative byte of a set of composite keys K in dimension $D \in \{P, V\}$ is the position of the first byte in dimension D for which not all keys are equal:*

$$\begin{aligned} &\mathsf{dsc}(K,D) = m \textit{ iff} \\ &\quad \exists k_i, k_j \in K, i \neq j (k_i.D[m] \neq k_j.D[m]) \textit{ and} \\ &\quad \forall k_i, k_j \in K, l < m (k_i.D[l] = k_j.D[l]) \end{aligned}$$

If all values of dimension D in K are equal, the discriminative byte does not exist. In this case we define $\mathsf{dsc}(K,D) = \mathsf{len}(k_i.D) + 1$ for some $k_i \in K$. □

Example A.4. *Table A.2 illustrates the position of the discriminative bytes for the path and value dimensions for various sets of composite keys K.*

Table A.2: Illustration of the discriminative bytes for $\mathsf{K}^{1..7}$ from Table A.1 and various subsets of it.

Composite Keys K	$\mathsf{dsc}(K,P)$	$\mathsf{dsc}(K,V)$
$\mathsf{K}^{1..7}$	13	2
$\mathsf{K}^{2,5,6,7}$	14	3
$\mathsf{K}^{5,6,7}$	16	3
K^{6}	21	5

Discriminative bytes are crucial during query evaluation since at the position of the discriminative bytes the search space can be narrowed down. We alternate in a round-robin fashion between discriminative path and value bytes in our dynamic interleaving. Note that in order to determine the dynamic interleaving of a key k, which we denote by $I_{\mathrm{DY}}(k,K)$, we have to consider the set of keys K to which k belongs and determine where the keys in K differ from each other (i.e., where their discriminative bytes are located). Each discriminative byte partitions K into subsets, which we recursively partition further.

A.5.1 Partitioning by Discriminative Bytes

The partitioning of a set of keys K groups composite keys together that have the same value for the discriminative byte in dimension D. Thus, K is split into at most 2^8 non-empty partitions, one partition for each value (`0x00` to `0xFF`) of the discriminative byte of dimension D.

Definition A.5. *(ψ-Partitioning) $\psi(K,D) = \{K_1,\ldots,K_m\}$ is the ψ-partitioning of composite keys K in dimension D iff all partitions are non-empty ($K_i \neq \emptyset$ for $1 \leq i \leq m$), the number m of partitions is minimal, and:*

1. *All keys in partition $K_i \in \psi(K,D)$ have the same value for the discriminative byte of K in dimension D:*
   - $\forall k_u, k_v \in K_i \, (k_u.D[\mathsf{dsc}(K,D)] = k_v.D[\mathsf{dsc}(K,D)])$
2. *The partitions are disjoint:*
   - $\forall K_i, K_j \in \psi(K,D)(K_i \neq K_j \Rightarrow K_i \cap K_j = \emptyset)$
3. *The partitioning is complete:*
   - $K = \bigcup_{K_i \in \psi(K,D)} K_i$ □

Let $k \in K$ be a composite key. We write $\psi_k(K,D)$ to denote the *ψ-partitioning of k* with respect to K and dimension D, i.e., the partition in $\psi(K,D)$ that contains key k.

Example A.5. *Let $\mathsf{K}^{1..7}$ be the set of composite keys from Table A.1. The ψ-partitioning of selected sets of keys in dimension P or V is as follows:*

- $\psi(\mathsf{K}^{1..7}, V) = \{\mathsf{K}^{2,5,6,7}, \mathsf{K}^{1}, \mathsf{K}^{3,4}\}$
- $\psi(\mathsf{K}^{2,5,6,7}, P) = \{\mathsf{K}^{2}, \mathsf{K}^{5,6,7}\}$
- $\psi(\mathsf{K}^{5,6,7}, V) = \{\mathsf{K}^{5}, \mathsf{K}^{6}, \mathsf{K}^{7}\}$
- $\psi(\mathsf{K}^{6}, V) = \psi(\mathsf{K}^{6}, P) = \{\mathsf{K}^{6}\}$

The ψ-partitioning of key k_6 with respect to sets of keys and dimensions is as follows:

- $\psi_{\mathsf{k}_6}(\mathsf{K}^{1..7}, V) = \mathsf{K}^{2,5,6,7}$
- $\psi_{\mathsf{k}_6}(\mathsf{K}^{6}, V) = \psi_{\mathsf{k}_6}(\mathsf{K}^{6}, P) = \mathsf{K}^{6}$. □

The crucial property of our partitioning is that the position of the discriminative byte for dimension D increases if we ψ-partition K in D. This *monotonicity* property of the ψ-partitioning holds since every partition is built based on the discriminative byte and to partition an existing partition we need a discriminative byte that will be positioned further down in the byte-string. For the alternate dimension $\overline{D}$, i.e., $\overline{D} = P$ if $D = V$ and $\overline{D} = V$ if $D = P$, the position of the discriminative byte remains unchanged or may increase.

Theorem A.1. (Monotonicity of Discriminative Bytes) *Let $K_i \in \psi(K,D)$ be one of the partitions of K after partitioning in dimension D. In dimension D, the position of the discriminative byte in K_i is strictly greater than in K while, in dimension $\overline{D}$, the discriminative byte is equal or greater than in K, i.e.,*

$$\begin{aligned}&K_i \in \psi(K,D) \wedge K_i \subset K \Rightarrow \\ &\quad \mathsf{dsc}(K_i,D) > \mathsf{dsc}(K,D) \wedge \mathsf{dsc}(K_i,\overline{D}) \geq \mathsf{dsc}(K,\overline{D})\end{aligned}$$

Proof. The first line states that $K_i \subset K$ is one of the partitions of K. From Definition A.5 it follows that the value $k.D[\mathsf{dsc}(K,D)]$ is the same for every key $k \in K_i$. From Definition A.4 it follows that $\mathsf{dsc}(K_i,D) \neq \mathsf{dsc}(K,D)$. By removing one or more keys from K to get K_i, the keys in K_i will become more similar compared to those in K. That means, it is not possible for the keys in K_i to differ in a position $g < \mathsf{dsc}(K,D)$. Consequently, $\mathsf{dsc}(K_i,D) \not< \mathsf{dsc}(K,D)$ for any dimension D (so this also holds for $\overline{D}$: $\mathsf{dsc}(K_i,\overline{D}) \not< \mathsf{dsc}(K,\overline{D})$). Thus $\mathsf{dsc}(K_i,D) > \mathsf{dsc}(K,D)$ and $\mathsf{dsc}(K_i,\overline{D}) \geq \mathsf{dsc}(K,\overline{D})$. ☐

Example A.6. *The discriminative path byte of* $\mathsf{K}^{1..7}$ *is* 13 *while the discriminative value byte of* $\mathsf{K}^{1..7}$ *is* 2 *as shown in Table A.2. For partition* $\mathsf{K}^{2,5,6,7}$*, which is obtained by partitioning* $\mathsf{K}^{1..7}$ *in the value dimension, the discriminative path byte is* 14 *while the discriminative value byte is* 3*. The positions of both discriminative bytes have increased. For partition* $\mathsf{K}^{5,6,7}$*, which is obtained by partitioning* $\mathsf{K}^{2,5,6,7}$ *in the path dimension, the discriminative path byte is* 16 *while the discriminative value byte is* 3*. The position of the discriminative path byte has increased while the position of the discriminative value byte has not changed.*

When computing the dynamic interleaving of a composite key $k \in K$ we recursively ψ-partition K while alternating between dimension V and P. This yields a partitioning sequence $(K_1,D_1),\ldots,(K_n,D_n)$ for key k with $K_1 \supset K_2 \supset \cdots \supset K_n$. We start with $K_1 = K$ and $D_1 = V$. Next, $K_2 = \psi_k(K_1,V)$ and $D_2 = \overline{D}_1 = P$. We continue with the general scheme $K_{i+1} = \psi_k(K_i,D_i)$ and $D_{i+1} = \overline{D}_i$. This goes on until we run out of discriminative bytes in one dimension, which

means $\psi_k(K,D) = K$. From then on, we can only partition in dimension $\overline{D}$. When we run out of discriminative bytes in this dimension as well, that is $\psi_k(K,\overline{D}) = \psi_k(K,D) = K$, we stop. The partitioning sequence is finite due to the monotonicity of the ψ-partitioning (see Lemma A.1), which guarantees that we make progress in every step in at least one dimension. Below we define a partitioning sequence.

Definition A.6. *(*Partitioning Sequence*) The partitioning sequence $\rho(k,K,D) = ((K_1,D_1),\ldots,(K_n,D_n))$ of a composite key $k \in K$ denotes the recursive ψ-partitioning of the partitions to which k belongs. The pair (K_i,D_i) denotes the partitioning of K_i in dimension D_i. The final partition K_n cannot be partitioned further, hence $D_n = \bot$. $\rho(k,K,D)$ is defined as follows:*[1]

$$\rho(k,K,D) = \begin{cases} (K,D) \circ \rho(k,\psi_k(K,D),\overline{D}) & \textit{if } \psi_k(K,D) \subset K \\ \rho(k,K,\overline{D}) & \textit{if } \psi_k(K,D) = K \wedge \psi_k(K,\overline{D}) \subset K \\ (K,\bot) & \textit{otherwise} \end{cases}$$

Example A.7. *In the following we illustrate the step-by-step expansion of $\rho(\mathsf{k}_6,\mathsf{K}^{1..7},V)$ to get k_6's partitioning sequence.*

$$\begin{aligned} &\rho(\mathsf{k}_6,\mathsf{K}^{1..7},V) = \\ &\quad = (\mathsf{K}^{1..7},V) \circ \rho(\mathsf{k}_6,\mathsf{K}^{2,5,6,7},P) \\ &\quad = (\mathsf{K}^{1..7},V) \circ (\mathsf{K}^{2,5,6,7},P) \circ \rho(\mathsf{k}_6,\mathsf{K}^{5,6,7},V) \\ &\quad = (\mathsf{K}^{1..7},V) \circ (\mathsf{K}^{2,5,6,7},P) \circ (\mathsf{K}^{5,6,7},V) \circ \rho(\mathsf{k}_6,\mathsf{K}^{6},P) \\ &\quad = (\mathsf{K}^{1..7},V) \circ (\mathsf{K}^{2,5,6,7},P) \circ (\mathsf{K}^{5,6,7},V) \circ (\mathsf{K}^{6},\bot) \end{aligned}$$

Notice the alternating partitioning in, respectively, V and P. We only deviate from this if partitioning in one of the dimensions is not possible. For instance, $\mathsf{K}^{3,4}$ cannot be partitioned in dimension P and therefore we get

$$\rho(\mathsf{k}_4,\mathsf{K}^{1..7},V) = (\mathsf{K}^{1..7},V) \circ (\mathsf{K}^{3,4},V) \circ (\mathsf{K}^{4},\bot) \qquad \square$$

There are two key ingredients to our dynamic interleaving: the monotonicity of discriminative bytes (Lemma A.1) and the alternating ψ-partitioning (Definition A.6). The monotonicity

[1] Operator $\circ$ denotes concatenation, e.g., $a \circ b = (a,b)$ and $a \circ (b,c) = (a,b,c)$

guarantees that each time we ψ-partition K we advance the discriminative byte in at least one dimension. The alternating ψ-partitioning ensures that we interleave paths and values.

A.5.2 Interleaving

We determine the dynamic interleaving $I_{\text{DY}}(k,K)$ of a key $k \in K$ via k's partitioning sequence ρ. For each element in ρ, we generate a tuple containing two strings s_P and s_V and the partitioning dimension of the element. The strings s_P and s_V are composed of substrings of $k.P$ and $k.V$, ranging from the previous discriminative byte up to, but excluding, the current discriminative byte in the respective dimension. The order of s_P and s_V in a tuple depends on the dimension used in the previous step: the dimension that has been chosen for the partitioning comes first. Formally, this is defined as follows:

Definition A.7. (Dynamic Interleaving) *Let $k \in K$ be a composite key and let $\rho(k,K,V) = ((K_1,D_1), \ldots, (K_n,D_n))$ be the partitioning sequence of k. The dynamic interleaving $I_{DY}(k,K) = (t_1,\ldots,t_n)$ of k is a sequence of tuples t_i, where $t_i = (s_P,s_V,D)$ if $D_{i-1} = P$ and $t_i = (s_V,s_P,D)$ if $D_{i-1} = V$. The path and value substrings, s_P and s_V, and the partitioning dimension D are determined as follows:*

$$t_i.s_P = k.P[\mathsf{dsc}(K_{i-1},P), \mathsf{dsc}(K_i,P) - 1]$$
$$t_i.s_V = k.V[\mathsf{dsc}(K_{i-1},V), \mathsf{dsc}(K_i,V) - 1]$$
$$t_i.D = D_i$$

To correctly handle the first tuple we define $\mathsf{dsc}(K_0,V) = 1$, $\mathsf{dsc}(K_0,P) = 1$ and $D_0 = V$. □

Example A.8. *We compute the tuples for the dynamic interleaving $I_{DY}(\mathsf{k}_6,\mathsf{K}^{1..7}) = (\mathsf{t}_1,\ldots,\mathsf{t}_4)$ of key k_6 using the partitioning sequence $\rho(\mathsf{k}_6,\mathsf{K}^{1..7},V) = ((\mathsf{K}^{1..7},V), (\mathsf{K}^{2,5,6,7},P), (\mathsf{K}^{5,6,7},V), (\mathsf{K}^{6},\perp))$ from Example A.7. The necessary discriminative path and value bytes can be found in Table A.2. Table A.3 shows the details of each tuple of k_6's dynamic interleaving with respect to $\mathsf{K}^{1..7}$.*

*The final dynamic interleavings of all keys from Table A.1 are displayed in Table A.4. We highlight in bold the values of the discriminative bytes at which the paths and values are interleaved, e.g., for key k_6 these are bytes **00**, **/**, and **0C**.*

Table A.3: Computing the dynamic interleaving $I_{DY}(\mathsf{k}_6, \mathsf{K}^{1..7})$.

t	s_V	s_P	D
t_1	$\mathsf{k}_6.V[1,1] =$ `00`	$\mathsf{k}_6.P[1,12] =$ `/bom/item/ca`	V
t_2	$\mathsf{k}_6.V[2,2] =$ `00`	$\mathsf{k}_6.P[13,13] =$ `r`	P
t_3	$\mathsf{k}_6.V[3,2] = \varepsilon$	$\mathsf{k}_6.P[14,15] =$ `/b`	V
t_4	$\mathsf{k}_6.V[3,4] =$ `0C C2`	$\mathsf{k}_6.P[16,20] =$ `rake$`	$\perp$

Table A.4: The dynamic interleaving of the composite keys in $\mathsf{K}^{1..7}$. The values of the discriminative bytes are written in bold.

k	Dynamic Interleaving $I_{DY}(k, \mathsf{K}^{1..7})$
k_2	((`00`, `/bom/item/ca`, V), (**`00`**, `r`, P), (**`a`**`biner$`, `00 F1`, $\perp$))
k_7	((`00`, `/bom/item/ca`, V), (**`00`**, `r`, P), (**`/`**`b`, ε, V), (**`0A`** `8C`, `umper$`, $\perp$))
k_5	((`00`, `/bom/item/ca`, V), (**`00`**, `r`, P), (**`/`**`b`, ε, V), (**`0B`** `4A`, `elt$`, $\perp$))
k_6	((`00`, `/bom/item/ca`, V), (**`00`**, `r`, P), (**`/`**`b`, ε, V), (**`0C`** `C2`, `rake$`, $\perp$))
k_1	((`00`, `/bom/item/ca`, V), (**`01`** `0E 50`, `noe$`, $\perp$))
k_3	((`00`, `/bom/item/ca`, V), (**`03`** `D3`, `r/battery$`, V), (**`5A`**, ε, $\perp$))
k_4	((`00`, `/bom/item/ca`, V), (**`03`** `D3`, `r/battery$`, V), (**`B0`**, ε, $\perp$))

A.5.3 Efficiency of Interleavings

We introduce a cost model to measure the efficiency of different interleaving schemes. We assume that the interleaved keys are arranged in a tree-like search structure. Each node represents the partitioning of the composite keys by either path or value, and the node branches for each different value of a discriminative path or value byte. We simplify the cost model by assuming that the search structure is a complete tree with fanout o where every root-to-leaf path contains h edges (h is called the height). Further, we assume that all nodes on one level represent a partitioning in the same dimension ϕ_i and we use a vector $\phi(\phi_1, \ldots, \phi_h)$ to specify the partitioning dimension on each level. Figure A.3 visualizes this scheme.

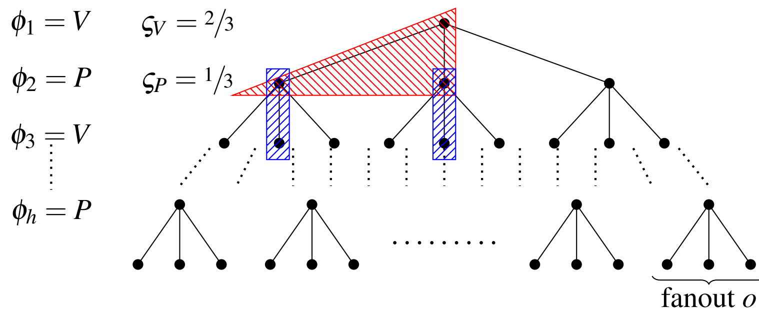


Figure A.3: The search structure in our cost model is a complete tree of height h and fanout o.

A search starts at the root and traverses the data structure to determine the answer set. In the case of range queries, more than one branch must be followed. A search follows a fraction of the outgoing branches o originating at this node. We call this the selectivity of a node (or just selectivity). We assume that every path node has a selectivity of ς_P and every value node has the selectivity of ς_V. The cost $\widehat{C}$ of a search, measured in the number of visited nodes during the search, is as follows:

$$\widehat{C}(o,h,\phi,\varsigma_P,\varsigma_V) = 1 + \sum_{l=1}^{h} \prod_{i=1}^{l} (o \cdot \varsigma_{\phi_i})$$

If a workload is known upfront, a system can optimize indexes to support specific queries. Our goal is an access method that can deal with a wide range of queries in a dynamic environment in a robust way, i.e., avoiding a bad performance for any particular query type. This is motivated by the fact that modern data analytics utilizes a large number of ad-hoc queries to do exploratory analysis. For example, in the context of building a robust partitioning scheme for ad-hoc query workloads, Shanbhag et al. [SJM$^+$17] found that after analyzing the first 80% of real-world workload traces the remaining 20% still contained 57% completely new queries.

Even though robustness of query processing performance has received considerable interest, there is a lack of unified metrics in this area [Gra11, GGKP12]. Our goal is a good performance for queries with differing selectivities for path and value predicates. Towards this goal we define the notion of *complementary queries*.

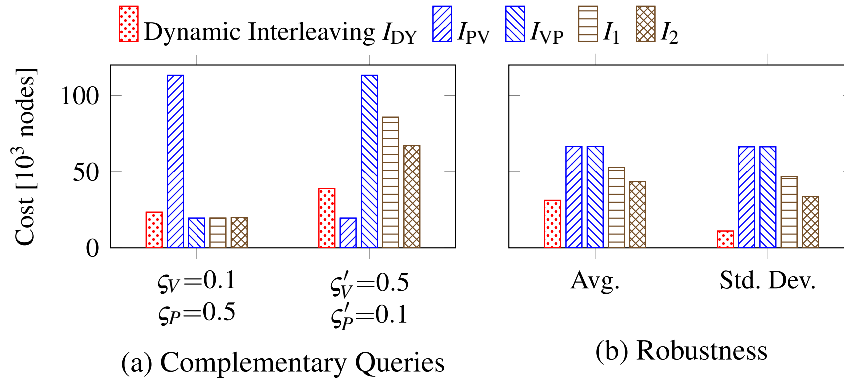
Definition A.8. *(*Complementary Query*) Given a query Q with path selectivity ς_P and value selectivity ς_V, there is a* complementary query *Q' with path selectivity $\varsigma'_P = \varsigma_V$ and value selectivity $\varsigma'_V = \varsigma_P$*

State-of-the-art CAS-indexes favor either path or value predicates. As a result they show a very good performance for one type of query but run into problems for the complementary query.

Definition A.9. *(*Robustness*) A CAS-index is* robust *if it optimizes the average performance when evaluating a query Q and its complementary query Q'.*

Example A.9. *Figure A.4a shows the costs for a query Q and its complementary query Q' for different interleavings in terms of the number of visited nodes during the search. We assume parameters $o = 10$ and $h = 12$ for the search structure. In our dynamic interleaving I_{DY} the discriminative bytes are perfectly alternating. I_{PV} stands for path-value concatenation with $\phi_i = P$ for $1 \leq i \leq 6$ and $\phi_i = V$ for $7 \leq i \leq 12$. I_{VP} is a value-path concatenation (with an*

inverse ϕ compared to I_{PV}). We also consider two additional permutations: I_1 uses a vector $\phi = (V,V,V,V,P,V,P,V,P,P,P,P)$ and I_2 one equal to $(V,V,V,P,P,V,P,V,V,P,P,P)$. They resemble, e.g., the byte-wise interleaving that usually exhibits irregular alternation patterns with a clustering of, respectively, discriminative path and value bytes. Figure A.4b shows the average costs and the standard deviation. The numbers demonstrate the robustness of our dynamic interleaving: it clearly shows the best performance both in terms of average costs and lowest standard deviation.



(a) Complementary Queries

(b) Robustness

Figure A.4: Dynamic interleaving has a robust query performance.

In the previous example we showed empirically that a perfectly alternating interleaving exhibits the best overall performance when evaluating complementary queries. In addition to this, we can prove that this is always the case.

Theorem A.1. *Consider a query Q with selectivities ς_P and ς_V and its complementary query Q' with selectivities $\varsigma'_P = \varsigma_V$ and $\varsigma'_V = \varsigma_P$. There is no interleaving that on average performs better than the dynamic interleaving that has a perfectly alternating vector ϕ_{DY}, i.e., $\forall \phi : \widehat{C}(o,h,\phi_{DY},\varsigma_P,\varsigma_V) + \widehat{C}(o,h,\phi_{DY},\varsigma'_P,\varsigma'_V) \leq \widehat{C}(o,h,\phi,\varsigma_P,\varsigma_V) + \widehat{C}(o,h,\phi,\varsigma'_P,\varsigma'_V)$.*

Proof. We begin with a brief outline of the proof. We show for a level l that the costs of query Q and complementary query Q' on level l is smallest with the dynamic interleaving. That is, for a level l we show that $\prod_{i=1}^{l}(o \cdot \varsigma_{\phi_i}) + \prod_{i=1}^{l}(o \cdot \varsigma'_{\phi_i})$ is smallest with the vector $\phi_{\mathrm{DY}} = (V,P,V,P,\dots)$ of our dynamic interleaving. Since this holds for any level l, it also holds for the sum of costs over all levels l, $1 \leq l \leq h$, and this proves the theorem.

We only look at search trees with a height $h \geq 2$, as for $h = 1$ we do not actually have an interleaving (and the costs are all the same). W.l.o.g., we assume that the first level of the search

tree always starts with a discriminative value byte, i.e., $\phi_1 = V$. Let us look at the cost for one specific level l for query Q and its complementary query Q'. We distinguish two cases: l is even or l is odd.

l **is even:** The cost for a perfectly alternating interleaving for Q for level l is equal to $o^l(\varsigma_V \cdot \varsigma_P \dots \varsigma_V \cdot \varsigma_P)$, while the cost for Q' is equal to $o^l(\varsigma'_V \cdot \varsigma'_P \dots \varsigma'_V \cdot \varsigma'_P)$, which is equal to $o^l(\varsigma_P \cdot \varsigma_V \dots \varsigma_P \cdot \varsigma_V)$. This is the same cost as for Q, so adding the two costs gives us $2o^l \varsigma_V^{l/2} \varsigma_P^{l/2}$

For a non-perfectly alternating interleaving with the same number of ς_V and ς_P multiplicands up to level l we have the same cost as for our dynamic interleaving, i.e., $2o^l \varsigma_V^{l/2} \varsigma_P^{l/2}$. Now let us assume that the number of ς_V and ς_P multiplicands is different for level l (there must be at least one such level l). Assume that for Q we have r multiplicands of type ς_V and s multiplicands of type ς_P, with $r + s = l$ and, w.l.o.g., $r > s$. This gives us $o^l \varsigma_V^s \varsigma_P^s \varsigma_V^{r-s} + o^l \varsigma_V^s \varsigma_P^s \varsigma_P^{r-s} = o^l \varsigma_V^s \varsigma_P^s (\varsigma_V^{r-s} + \varsigma_P^{r-s})$ for the cost.

We have to show that $2o^l \varsigma_V^{l/2} \varsigma_P^{l/2} \leq o^l \varsigma_V^s \varsigma_P^s (\varsigma_V^{r-s} + \varsigma_P^{r-s})$. As all values are greater than zero, this is equivalent to $2\varsigma_V^{l/2-s} \varsigma_P^{l/2-s} \leq \varsigma_V^{r-s} + \varsigma_P^{r-s}$. The right-hand side can be reformulated: $\varsigma_V^{r-s} + \varsigma_P^{r-s} = \varsigma_V^{l-2s} + \varsigma_P^{l-2s} = \varsigma_V^{l/2-s} \varsigma_V^{l/2-s} + \varsigma_P^{l/2-s} \varsigma_P^{l/2-s}$. Setting $a = \varsigma_V^{l/2-s}$ and $b = \varsigma_P^{l/2-s}$, this boils down to showing $2ab \leq a^2 + b^2 \Leftrightarrow 0 \leq (a-b)^2$, which is always true.

l **is odd:** W.l.o.g. we assume that for computing the cost for a perfectly alternating interleaving for Q, there are $\lceil l/2 \rceil$ multiplicands of type ς_V and $\lfloor l/2 \rfloor$ multiplicands of type ς_P. This results in $o^l \varsigma_V^{\lfloor l/2 \rfloor} \varsigma_P^{\lfloor l/2 \rfloor} (\varsigma_V + \varsigma_P)$ for the sum of costs for Q and Q'.

For a non-perfectly alternating interleaving, we again have $o^l \varsigma_V^s \varsigma_P^s (\varsigma_V^{r-s} + \varsigma_P^{r-s})$ with $r + s = l$ and $r > s$, which can be reformulated to $o^l \varsigma_V^s \varsigma_P^s (\varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_V + \varsigma_P^{\lfloor l/2 \rfloor - s} \varsigma_P^{\lfloor l/2 \rfloor - s} \varsigma_P)$.

What is left to prove is $o^l \varsigma_V^{\lfloor l/2 \rfloor} \varsigma_P^{\lfloor l/2 \rfloor} (\varsigma_V + \varsigma_P) \leq o^l \varsigma_V^s \varsigma_P^s (\varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_V + \varsigma_P^{\lfloor l/2 \rfloor - s} \varsigma_P^{\lfloor l/2 \rfloor - s} \varsigma_P)$, which is equivalent to $\varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_P^{\lfloor l/2 \rfloor - s} (\varsigma_V + \varsigma_P) \leq \varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_V^{\lfloor l/2 \rfloor - s} \varsigma_V + \varsigma_P^{\lfloor l/2 \rfloor - s} \varsigma_P^{\lfloor l/2 \rfloor - s} \varsigma_P$. Substituting $a = \varsigma_V$, $b = \varsigma_P$, and $x = \lfloor l/2 \rfloor - s$, this means showing that $a^x b^x (a+b) \leq a^{2x+1} + b^{2x+1} \Leftrightarrow 0 \leq a^{2x+1} + b^{2x+1} - a^x b^x (a+b)$. Factorizing this polynomial gives us $(a^x - b^x)(a^{x+1} - b^{x+1})$ or $(b^x - a^x)(b^{x+1} - a^{x+1})$. We look at $(a^x - b^x)(a^{x+1} - b^{x+1})$, the argument for the other factorization follows along the same lines. This term can only become negative if one factor is negative and the other is positive. Let us first look at the case $a < b$: since $0 \leq a, b \leq 1$, we can immediately follow that $a^x < b^x$ and $a^{x+1} < b^{x+1}$, i.e., both factors are negative. Analogously, from $a > b$ (and $0 \leq a, b \leq 1$) immediately follows $a^x > b^x$ and $a^{x+1} > b^{x+1}$, i.e., both factors are positive. □

Note that in practice the search structure is not a complete tree and the fraction ς_P and ς_V of children that are traversed at each node is not constant. In Section A.7.4 we experimentally evaluate the cost model on real-world datasets. We show that the estimated cost and the true cost of a query are off by a factor of two, on average. This is a good estimate for the cost of a query.

A.6 RCAS Index

We propose the Robust Content-And-Structure (RCAS) index to efficiently query the content and structure of hierarchical data. The RCAS index uses our dynamic interleaving to integrate the paths and values of composite keys in a trie-based index.

A.6.1 Trie-Based Structure of RCAS

The RCAS index is a trie data-structure that efficiently supports CAS queries with range and prefix searches. Each node n in the RCAS index includes a dimension $n.D$, path substring $n.s_P$, and value substring $n.s_V$ that correspond to the fields $t.D$, $t.s_P$ and $t.s_V$ in the dynamic interleaving of a key (see Definition A.7). The substrings $n.s_P$ and $n.s_V$ are variable-length strings. Dimension $n.D$ is P or V for inner nodes and $\perp$ for leaf nodes. Leaf nodes additionally store a set of references r_i to nodes in the database, denoted n.refs. Each dynamically interleaved key corresponds to a root-to-leaf path in the RCAS index.

Definition A.10. *(*RCAS Index*) Let K be a set of composite keys and let R be a tree. Tree R is the RCAS index for K iff the following conditions are satisfied.*

1. *$I_{DY}(k,K) = (t_1,\ldots,t_m)$ is the dynamic interleaving of a key $k \in K$ iff there is a root-to-leaf path $(n_1,\ldots,n_m)$ in R such that $t_i.s_P = n_i.s_P$, $t_i.s_V = n_i.s_V$, and $t_i.D = n_i.D$ for $1 \le i \le m$.*
2. *R does not include duplicate siblings, i.e., no two sibling nodes n and n', $n \neq n'$, in R have the same values for s_P, s_V, and D, respectively.*

Example A.10. *Figure A.5 shows the RCAS index for the composite keys $\mathsf{K}^{1..7}$. We use blue and red colors for bytes from the path and value, respectively. The discriminative bytes are highlighted in bold. The dynamic interleaving $I_{DY}(\mathsf{k}_6,\mathsf{K}^{1..7}) = (\mathsf{t}_1,\mathsf{t}_2,\mathsf{t}_3,\mathsf{t}_4)$ from Table A.4 is mapped to the path $(\mathsf{n}_1,\mathsf{n}_2,\mathsf{n}_4,\mathsf{n}_7)$ in the RCAS index. For key k_2, the first two tuples of $I_{DY}(\mathsf{k}_2,\mathsf{K}^{1..7})$ are also mapped to n_1 and n_2, while the third tuple is mapped to n_3.*

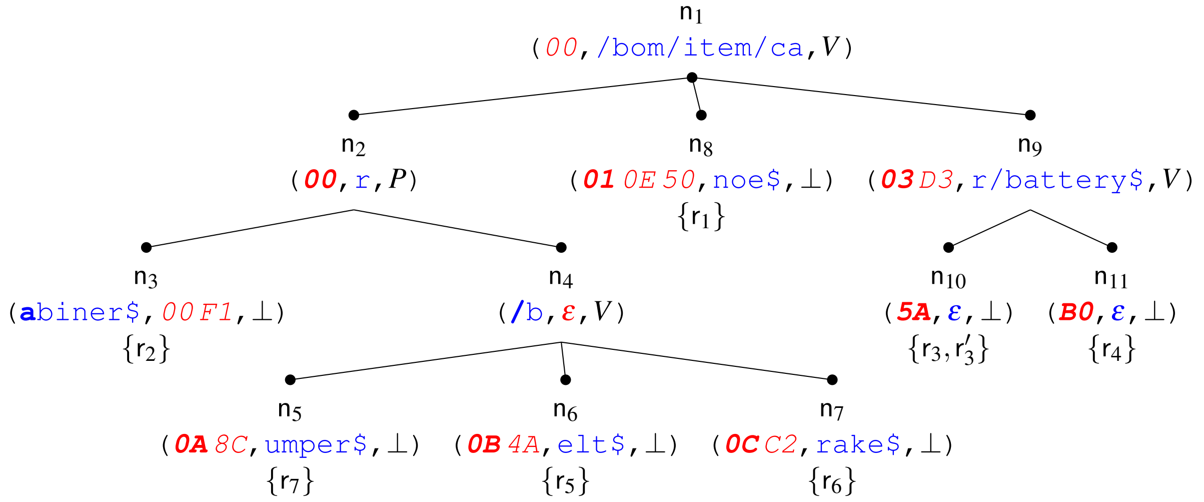


Figure A.5: The RCAS index for the composite keys $\mathsf{K}^{1..7}$.

A.6.2 Physical Node Layout

Figure A.6 shows the physical structure of an inner node. The `header` field contains meta information, such as the number of children. Fields s_P and s_V (explained above) are implemented as variable-length byte vectors (C++'s `std::vector<uint8_t>`). Dimension D (P or V, or $\perp$ if the node is a leaf) is the dimension in which the node partitions the data. The remaining space of an inner node (gray-colored in Figure A.6) is reserved for child pointers. Since ψ partitions at the granularity of bytes, each node can have at most 256 children, one for each possible value of a discriminative byte from `0x00` to `0xFF` (or their corresponding ASCII characters in Figure A.5). For each possible value b there is a pointer to the subtree whose keys all have value b for the discriminative byte of dimension D. Typically, many of the 256 pointers are `NULL`. Therefore, we implement our trie as an Adaptive Radix Tree (ART) [LKN13], which defines four node types with a physical fanout of 4, 16, 48, and 256 child pointers, respectively. Nodes are resized to adapt to the actual fanout of a node. Figure A.6 illustrates the node type with an array of 256 child pointers. For the remaining node types we refer to Leis et al. [LKN13].

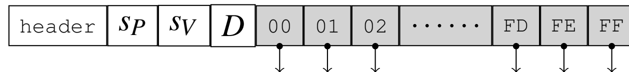


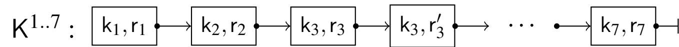
Figure A.6: Structure of an inner node with 256 pointers.

The structure of a leaf node is similar to that shown in Figure A.6, except that instead of child pointers the leaf nodes have a variable-length vector with references to nodes in the database.

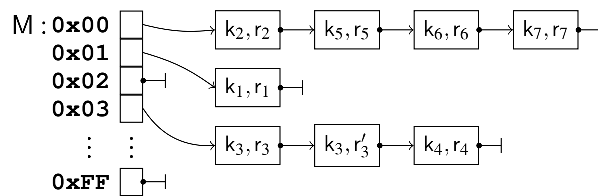
A.6.3 Bulk-Loading RCAS

This section gives an efficient bulk-loading algorithm for RCAS that is linear in the number of composite keys $|K|$. It simultaneously computes the dynamic interleaving of all keys in K. We implement a partition K as a linked list of pairs (k,r), where r is a reference to a database node with path $k.P$ and value $k.V$. In our implementation the keys in K need not be unique. There can be pairs (k,r_i) and (k,r_j) that have the same key but have different references $r_i \neq r_j$. This is the case if there are different nodes in the indexed database that have the same path and value (thus the same key). A partitioning $M = \psi(K,D)$ is implemented as an array of length 2^8 with references to (possibly empty) partitions K. The array indexes `0x00` to `0xFF` are the values of the discriminative byte.

Example A.11. *Figure A.7a shows the linked list for set* $\mathsf{K}^{1..7}$ *from our running example. Two nodes, pointed to by* r_3 *and* r'_3*, have the same key* k_3*. They correspond to the batteries in Figure A.1 that have the same path and value. Figure A.7b shows the partitioning* $\psi(\mathsf{K}^{1..7},V)$ *for our running example. Three partitions exist with values* **`0x00`***,* **`0x01`***, and* **`0x03`** *for the discriminative value byte.*



(a) Partition $\mathsf{K}^{1..7}$ is represented as a list of (k,r) pairs.



(b) The partitioning $\mathsf{M} = \psi(\mathsf{K}^{1..7},V)$ is an array with 2^8 partitions.

Figure A.7: Data structures used in Algorithm A.3.

Algorithm A.1 determines the discriminative byte for a partition K. Note that $\mathsf{dsc_inc}$ looks for the discriminative byte starting from position g, where g is a lower bound for $\mathsf{dsc}(K,D)$ as per Lemma A.1. Also, looping through the bytes of the first key of K is correct even if there are shorter keys in K. Since we use prefix-free keys, any shorter keys differ at some position, terminating the loop correctly.

Algorithm A.1: dsc_inc(K, D, g)

```
1 Let (k_i, r_i) be the first key in K;
2 while g ≤ len(k_i.D) do
3 |  for (k_j, r_j) ∈ K do
4 |  |  if k_j.D[g] ≠ k_i.D[g] then return g;
5 |  g++;
6 return g
```

Algorithm A.2 illustrates the computation of the ψ-partitioning $M = \psi(K, D, g)$. We pass the position g of the discriminative byte for dimension D as an argument to ψ. The discriminative byte determines the partition to which a key belongs.

Algorithm A.2: $\psi(K, D, g)$

```
1 Let M be an array of 2^8 empty lists;
2 for (k_i, r_i) ∈ K do
3 |  Move (k_i, r_i) from partition K to partition M[k_i.D[g]];
4 return M
```

Algorithm A.3 recursively ψ-partitions K and alternates between the path and value dimensions. In each call of BulkLoadRCAS one new node n is added to the index. The algorithm takes four parameters: (a) partition K, (b) dimension $D \in \{P, V\}$ by which K is partitioned, and the positions of the previous discriminative (c) path byte g_P and (d) value byte g_V. For the first call, when no previous discriminative path and value bytes exist, we set $g_P = g_V = 1$. BulkLoadRCAS$(K, V, 1, 1)$ returns a pointer to the root node of the new RCAS index. We start by creating a new node n (line 1) and determining the discriminative path and value bytes g'_P and g'_V of K (lines 3-4). Lemma A.1 guarantees that the previous discriminative bytes g_P and g_V are valid lower bounds for g'_P and g'_V, respectively. Next, we determine the current node's substrings s_P and s_V in lines 5-6 (see Definition A.7). In lines 7-10 we check for the base case of the recursion, which occurs when all discriminative bytes are exhausted and K cannot be partitioned further. In this case, all remaining pairs $(k, r) \in K$ have the same key k. Leaf node n contains the references to nodes in the database with this particular key k. In lines 11 we check if K can be partitioned in dimension D. If this is not the case, since all keys have the same value in dimension D, we ψ-partition K in the alternate dimension $\overline{D}$. Finally, in lines 14-16 we iterate over all non-empty partitions in M and recursively call the algorithm for each partition $M[b]$, alternating dimension D in round-robin fashion.

Algorithm A.3: BulkLoadRCAS(K, D, g_P, g_V)

```
1  Let n be a new RCAS node;
2  Let (k_i, r_i) be the first key in K;
3  g'_P ← dsc_inc(K, P, g_P);
4  g'_V ← dsc_inc(K, V, g_V);
5  n.s_P ← k_i.P[g_P, g'_P − 1];
6  n.s_V ← k_i.V[g_V, g'_V − 1];
7  if g'_P > len(k_i.P) ∧ g'_V > len(k_i.V) then                /* n is a leaf */
8      n.D ← ⊥;
9      for (k_j, r_j) ∈ K do append r_j to n.refs ;
10     return n;
11 if g'_D > len(k_i.D) then D ← D̄ ;
12 n.D ← D;
13 M ← ψ(K, D, g'_D);
14 for b ← 0x00 to 0xFF do
15     if partition M[b] is not empty then
16         n.children[b] ← BulkLoadRCAS(M[b], D̄, g'_P, g'_V);
17 return n
```

Theorem A.2. *Let K be a set of composite keys and let $l = \max_{k \in K}\{len(k.P) + len(k.V)\}$ be the length of the longest key. The time complexity of Algorithm A.3 is $O(l \cdot |K|)$.*

Proof. We split the computations performed in function BulkLoadRCAS in Algorithm A.3 into two groups. The first group includes the computations of the discriminative bytes across *all* recursive invocations of BulkLoadRCAS (lines 1–12). The second group consists of the ψ-partitioning (line 13) across *all* recursive invocations of BulkLoadRCAS.

Group 1: BulkLoadRCAS exploits the monotonicity of the discriminative bytes (Lemma A.1) and passes the lower bound g to function dsc_inc(K, D, g). As a result, we scan each byte of $k.P$ and $k.V$ only once for each k in K to determine the discriminative bytes. This amounts to one full scan over all bytes of all keys in K across all invocations of BulkLoadRCAS. The complexity of this group is $O\left(\sum_{k \in K}(\text{len}(k.P) + \text{len}(k.V))\right) = O(l \cdot |K|)$.

Group 2: Given the position g of the discriminative byte computed earlier, $\psi(K, D, g)$ must only look at the value of this byte in dimension D of each key $(k, r) \in K$ and append (k, r) to the proper partition $M[k.D[g]]$ in constant time. Thus, a single invocation of $\psi(K, D)$ can be performed in $O(|K|)$ time. The partitioning $\psi(K, D)$ is disjoint and complete (see Definition A.5), i.e., $|K| = \sum_{K_i \in \psi(K,D)} |K_i|$. Therefore, on each level of the RCAS index at most $|K|$ keys

need to be partitioned, with a cost of $O(|K|)$. In the worst case, the height of the RCAS index is l, in which case every single path and value byte of the longest key is discriminative. Therefore, the cost of partitioning K across all levels of the index is $O(l \cdot |K|)$.

Although we partition K recursively for every discriminative byte, the partitions become smaller and smaller and on each level add up to at most $|K|$ keys. Thus, the costs of the operations in group 1 and group 2 add up to $O(2 \cdot l \cdot |K|) = O(l \cdot |K|)$. □

The factor l in the complexity of Algorithm A.3 is typically much smaller than $\mathsf{len}(k.P) + \mathsf{len}(k.V)$ of the longest key k. For instance, assuming a combined length of just six bytes would already give us around 280 trillion potentially different keys. So, we would need a huge number of keys for every byte to become a discriminative byte on each recursion level.

A.6.4 Querying RCAS

Algorithm A.4 shows the pseudocode for evaluating a CAS query on an RCAS index. The function CasQuery gets called with the current node n (initially the root node of the index), a path predicate consisting of a query path q, and a range $[v_l, v_h]$ for the value predicate. Furthermore, we need two buffers $\texttt{buff}_P$ and $\texttt{buff}_V$ (initially empty) that hold, respectively, all path and value bytes from the root to the current node n. Finally, we require state information s to evaluate the path and value predicates (we provide details as we go along) and an answer set W to collect the results.[2]

Algorithm A.4: CasQuery$(n, q, [v_l, v_h], \texttt{buff}_V, \texttt{buff}_P, s, W)$

```
1 UpdateBuffers(n, buffV, buffP)
2 matchV ← MatchValue(buffV, vl, vh, s, n)
3 matchP ← MatchPath(buffP, q, s, n)
4 if matchV = MATCH and matchP = MATCH then
5 |   Collect(n, W)
6 else if matchV ≠ MISMATCH and matchP ≠ MISMATCH then
7 |   for each matching child c in n do
8 |   |   s' ← Update(s)
9 |   |   CasQuery(c, q, [vl, vh], buffV, buffP, s', W)
```

[2]The parameters n, W, q, and $[v_l, v_h]$ are call-by-reference, the parameters $\texttt{buff}_V$, $\texttt{buff}_P$, and s are call-by-value.

First, we update the buffers $\texttt{buff}_V$ and $\texttt{buff}_P$, adding the information in the fields s_V and s_P of the current node n (line 1). Next, we match the query predicates to the current node. Matching values (line 2) works differently to matching paths (line 3), so we look at the two cases separately.

To match the current (partial) value $\texttt{buff}_V$ against the value range $[v_l, v_h]$, their byte strings must be binary comparable (for a detailed definition of binary-comparability see [LKN13]). Function `MatchValue` proceeds as follows. We compute the longest common prefix between $\texttt{buff}_V$ and v_l and between $\texttt{buff}_V$ and v_h. We denote the position of the first byte for which $\texttt{buff}_V$ and v_l differ by `lo` and the position of the first byte for which $\texttt{buff}_V$ and v_h differ by `hi`. If $\texttt{buff}_V[\texttt{lo}] < v_l[\texttt{lo}]$, we know that the node's value lies outside of the range and we return `MISMATCH`. Similarly, if $\texttt{buff}_V[\texttt{hi}] > v_h[\texttt{hi}]$, the node's value lies outside of the upper bound and we return `MISMATCH` as well. If n is a leaf node and $v_l \leq \texttt{buff}_V \leq v_h$, we return `MATCH`. If n is not a leaf node and $v_l[\texttt{lo}] < \texttt{buff}_V[\texttt{lo}]$ and $\texttt{buff}_V[\texttt{hi}] < v_h[\texttt{hi}]$, we know that all values in the subtree rooted at n match and we also return `MATCH`. In all other cases we cannot make a decision yet and return `INCOMPLETE`. The values of `lo` and `hi` are kept in the state to avoid recomputing the longest common prefix from scratch for each node. Instead we can resume the search from the previous values of `lo` and `hi`.

Function `MatchPath` matches the query path q against the current path prefix $\texttt{buff}_P$. It supports the wildcard symbol `*` and the descendant-or-self axis `//` that match any child and descendant node, respectively. As long as we do not encounter any wildcards in the query path q, we directly compare (a prefix of) q with the current content of $\texttt{buff}_P$ byte by byte. As soon as a byte does not match, we return `MISMATCH`. If we are able to successfully match the complete query path q against a complete path in $\texttt{buff}_P$ (both terminated by `$`), we return `MATCH`. Otherwise, we need to continue and return `INCOMPLETE`. When we encounter a wildcard `*` in q, we match it successfully to the corresponding label in $\texttt{buff}_P$ and continue with the next label. A wildcard `*` itself will not cause a mismatch (unless we try to match it against the terminator `$`), so we either return `MATCH` if it is the final label in q and $\texttt{buff}_P$ or `INCOMPLETE`. Matching the descendant-axis `//` is more complicated. We note the current position where we are in $\texttt{buff}_P$ and continue matching the label after `//` in q. If at any point we find a mismatch, we backtrack to the next path separator after the noted position, thus skipping a label in $\texttt{buff}_P$ and restarting the search from there. Once $\texttt{buff}_P$ contains a complete path, we can make a decision between `MATCH` or `MISMATCH`.

The algorithm continues by checking the outcomes of the value and path matching (lines 4 and 6). If both predicates match, we descend the subtree and collect all references (line 5 and function

Collect in Algorithm A.5). If at least one of the outcomes is `MISMATCH`, we immediately stop the search in the current node, otherwise we continue recursively with the matching children of n (lines 7–9). Finding the matching children depends on the dimension $n.D$ of n and follows the same logic as described above for `MatchValue` and `MatchPath`. If node $n.D = P$ and we have seen a descendant axis in the query path, all children of the current node match.

Algorithm A.5: Collect(n, W)

```
1 if n is a leaf then
2 |   add references r in n.refs to W
3 else
4 |   for each child c in n do
5 |   |   Collect(c, W)
```

Example A.12. *We consider an example CAS query with path $q =$ `/bom/item//battery$` and a value range from $v_l = 10^5 =$ `00 01 86 A0` to $v_h = 5 \cdot 10^5 =$ `00 07 A1 20`. We execute the query on the index depicted in Figure A.5.*

- *Starting at the root node n_1, we load `00` and `/bom/item/ca` into $\texttt{buff}_V$ and $\texttt{buff}_P$, respectively. Function `MatchValue` matches `00` and returns `INCOMPLETE`. `MatchPath` also returns `INCOMPLETE`: even though it matches `/bom/item`, the partial label `ca` does not match `battery`, so `ca` is skipped by the descendant axis. Since both functions return `INCOMPLETE`, we have to traverse all matching children. Since n_1 is a value node ($n_1.D = V$), we look for all matching children whose discriminative value byte is between `01` and `07`. Nodes n_8 and n_9 satisfy this condition.*

- *Node n_8 is a leaf. $\texttt{buff}_P$ and $\texttt{buff}_V$ are updated and contain complete paths and values. Byte `01` matches, but byte $\texttt{buff}_V[3] =$ `0E` $<$ `86` $= v_l[3]$. Thus, `MatchValue` returns a `MISMATCH`. So does `MatchPath`. The search discards n_8.*

- *Next we look at node n_9. We find that $v_l[2] < \texttt{buff}_V[2] < v_h[2]$, thus all values of n_9's descendants are within the bounds v_l and v_h, and `MatchValue` returns `MATCH`. `MatchPath` skips the next bytes `r/` due to the descendant axis and resumes matching from there. After skipping `r/`, it returns `MATCH`, as `battery$` matches the query path until its end. Both predicates match, invoking Collect on n_9, which traverses n_9's descendants n_{10} and n_{11} and adds references r_3, r'_3, and r_4 to W.*

Twig queries [BKH+17] with predicates on multiple attributes are broken into smaller CAS queries. Each root-to-leaf branch of the twig query is evaluated independently on an appropriate RCAS index and the resulting sets W are joined to produce the final result. The join requires that the references $r \in W$ contain structural information about a node's position in the tree (e.g., an OrdPath [OOP+04] node-labeling scheme). A query optimizer can use our cost model to choose which RCAS indexes are used in a query plan.

A.7 Experimental Evaluation

A.7.1 Setup and Datasets

We use a virtual Ubuntu 18.04 server with eight cores and 64GB of main memory. All algorithms are implemented in C++ by the same author and were compiled with clang 6.0.0 using -O3. Each reported runtime measurement is the average of 100 runs. All indexes are kept in main memory. The code[3] and the datasets[4] used in the experimental evaluation can be found online.

Datasets. We use three datasets, the ServerFarm dataset that we collected ourselves, a product catalog with products from Amazon [HM16], and the synthetic XMark [SWK+02] dataset at a scale factor of 500. The ServerFarm dataset mirrors the file system of 100 Linux servers. For each server we installed a default set of packages and randomly picked a subset of optional packages. In total there are 21 million files. For each file we record the file's full path, size, various timestamps (e.g., a file's change time `ctime`), file type, extension etc. The Amazon dataset [HM16] contains products that are hierarchically categorized. For each experiment we index the paths in a dataset along with one of its attributes. We use the notation `$dataset:$attribute` to indicate which attribute in a dataset is indexed. E.g., ServerFarm:size contains the path of each file in the ServerFarm dataset along with its size. The datasets do not have to fit into main memory, but we assume that the indexed keys fit into main memory. Table A.5 shows a summary of the datasets.

[3] https://github.com/k13n/rcas
[4] https://doi.org/10.5281/zenodo.3739263

Table A.5: Dataset Statistics

Dataset	Size	Attribute	No. of Keys	Unique Keys	Size of Keys
ServerFarm	3.0GB	size	21’291’019	9’345’668	1.7GB
XMark	58.9GB	category	60’272’422	1’506’408	3.3GB
Amazon	10.5GB	price	6’707’397	6’461’587	0.8GB

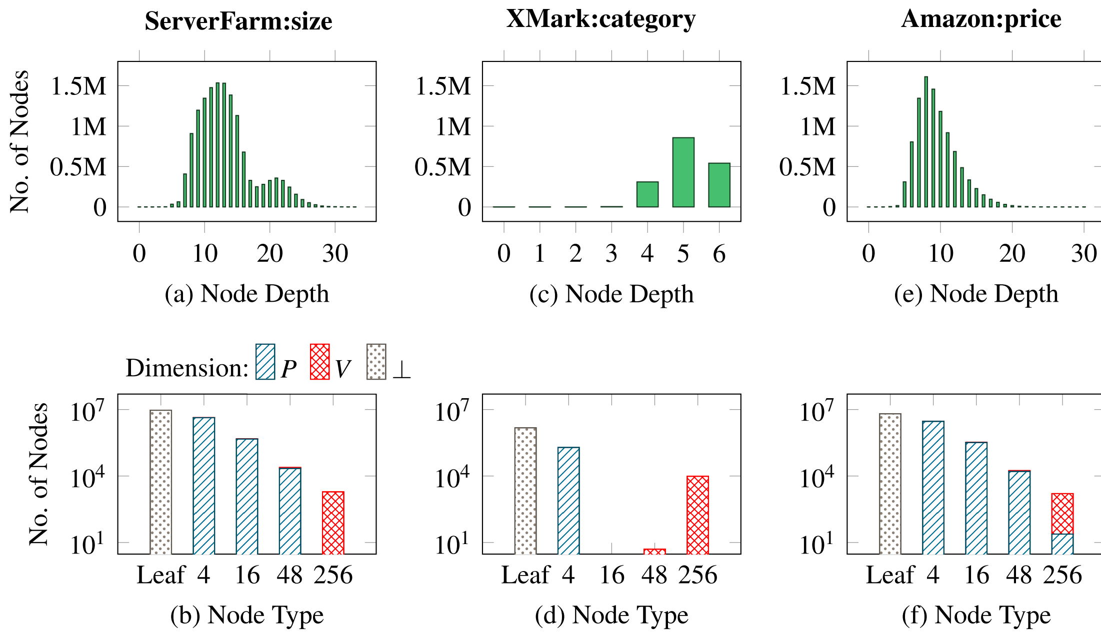
Compared Approaches. We compare our RCAS index based on dynamic interleaving with the following approaches that can deal with variable-length keys. The path-value (PV) and value-path (VP) concatenations are the two possible c-order curves [NY17]. The next approach, termed ZO for z-order [Mor66, OM84], maps variable-length keys to a fixed length as proposed by Markl [Mar99]. Each path label is mapped to a fixed length using a surrogate function. Since paths can have a different number of labels, shorter paths are padded with empty labels to match the number of labels in the longest path in the dataset. The resulting paths have a fixed length l_P and are interleaved with values of length l_V such that $\lceil l_V/l_P \rceil$ value bytes are followed by $\lceil l_P/l_V \rceil$ path bytes. The label-wise interleaving (LW) interleaves one byte of the value with one label of the path. We utilize our RCAS index to identify the dimension of every byte of the variable-length interleaved keys. The same underlying data-structure is also used to store the keys generated by PV, VP, and ZO. Finally, we compare RCAS against the CAS index in [MHSB15] that builds a structural summary (DataGuide [GW97]) and a value index (B+ tree[5]) and joins them to answer CAS queries. We term this approach XML as it was proposed for XML databases.

A.7.2 Impact of Datasets on RCAS’s Structure

In Figure A.8 we show how the shape (depth and width) of the RCAS index adapts to the datasets. Figure A.8a shows the distribution of the node depths in the RCAS index for the ServerFarm:size dataset. Because of the trie-based structure not every root-to-leaf path in RCAS has the same length (see also Figure A.5). The deepest nodes occur on level 33, but most nodes occur on levels 10 to 15 with an average node depth of 13.2. This is due to the different lengths of the paths in a file system. Figure A.8b shows the number of nodes for each node type. Recall from Section A.6.2 that RCAS, like ART [LKN13], uses inner nodes with a physical fanout of 4, 16, 48, and 256 pointers depending on the actual fanout of a node to reduce the space consumption.

[5] We use the tlx B+ tree (https://github.com/tlx/tlx), used also by [BZP+18, ZLL+18] for comparisons.

The type of a node n and its dimension $n.D$ are related. Path nodes typically have a smaller fanout than value nodes. This is to be expected, since paths only contain printable ASCII characters (of which there are about 100), while values span the entire available byte spectrum. Therefore, the most frequent node type for path nodes is type 4, while for value nodes it is type 256, see Figure A.8b. Leaf nodes do not partition the data and thus their dimension is set to $\perp$ according to Definition A.7. The RCAS index on the ServerFarm:size dataset contains more path than value nodes. This is because in this dataset there are about 9M unique paths as opposed to about 230k unique values. Thus, the values contain fewer discriminative bytes than the paths and can be better compressed by the trie structure.



	ServerFarm:size	XMark:category	Amazon:price
Avgerage Depth	13.2	5.1	9.5
Size RCAS (GB)	1.5	0.6	1.2

Figure A.8: Structure of the RCAS index

Figures A.8c and A.8d show the same information for dataset XMark:category. The RCAS index is more shallow since there are only 7 unique paths and ca. 390k unique values in a dataset of 60M keys. Thus the number of discriminative path and value bytes is low and the index less deep. Nodes of type 256 are frequent (see Figure A.8d) because of the larger number of unique values. While the XMark:category dataset contains 40M more keys than the ServerFarm:size dataset, the RCAS index for the former contains 1.5M nodes as compared to the 14M nodes for the latter.

Table A.6: CAS queries with their result size and the number of keys that match the path, respectively value predicate.

Query definitions

Dataset: **ServerFarm:size**	
Q_1	$Q($`/usr/include//`$,$`@size`$\geq 5000)$
Q_2	$Q($`/usr/include//`$,3000 \leq$ `@size` $\leq 4000)$
Q_3	$Q($`/usr/lib//`$,0 \leq$ `@size` $\leq 1000)$
Q_4	$Q($`/usr/share//Makefile`$,1000 \leq$ `@size` $\leq 2000)$
Q_5	$Q($`/usr/share/doc//README`$,4000 \leq$ `@size` $\leq 5000)$
Q_6	$Q($`/etc//`$,$`@size`$\geq 5000)$
Dataset: **XMark:category**	
Q_7	$Q($`/site/people//interest`$,0 \leq$ `@category` $\leq 50000)$
Q_8	$Q($`/site/regions/africa//`$,0 \leq$ `@category` $\leq 50000)$
Dataset: **Amazon:price**	
Q_9	$Q($`/CellPhones&Accessories//`$,10000 \leq$ `@price` $\leq 20000)$
Q_{10}	$Q($`/Clothing/Women/*/Sweaters//`$,7000 \leq$ `@price` $\leq 10000)$

Query selectivities

Q	Result size (σ)		Matching paths (σ_P)		Matching values (σ_V)	
Dataset: **ServerFarm:size**						
Q_1	142253	(0.6%)	434564	(2.0%)	7015066	(32.9%)
Q_2	46471	(0.2%)	434564	(2.0%)	1086141	(5.0%)
Q_3	512497	(2.4%)	2277518	(10.6%)	8403809	(39.4%)
Q_4	1193	($<$ 0.1%)	6408	($<$ 0.1%)	2494804	(11.7%)
Q_5	521	($<$ 0.1%)	24698	(0.1%)	761513	(3.5%)
Q_6	7292	($<$ 0.1%)	97758	(0.4%)	7015066	(32.9%)
Dataset: **XMark:category**						
Q_7	1910524	(3.1%)	19009723	(31.5%)	6066546	(10.0%)
Q_8	104500	(0.1%)	1043247	(1.7%)	6066546	(10.0%)
Dataset: **Amazon:price**						
Q_9	2758	($<$ 0.1%)	291625	(4.3%)	324272	(4.8%)
Q_{10}	239	($<$ 0.1%)	4654	($<$ 0.1%)	269936	(4.0%)

The RCAS index for the Amazon:price dataset has similar characteristics as the ServerFarm:size dataset, see Figures A.8e and A.8f.

A.7.3 Robustness

Table A.6 shows a number of typical CAS queries with their path and value predicates. For example, query Q_4 looks for all Makefiles underneath `/usr/share` that have a file size between 1KB and 2KB. In Table A.6 we report the selectivity σ of each query as well as path selectivity σ_P and value selectivity σ_V of the queries' path and value predicates, respectively. The RCAS index avoids large intermediate results that can be produced if an index prioritizes one dimension over the other, or if it independently evaluates and joins the results of the path and value predicates. Query Q_5, e.g., returns merely 521 matches, but its path and value predicates return intermediate results that are orders of magnitudes larger.

Figures A.9a to A.9f show that RCAS consistently outperforms its competitors for queries Q_1 to Q_6 from Table A.6 on the ServerFarm:size dataset. On these six queries ZO and LW perform similarly as PV, which is indicative for a high "puff-pastry effect" (see Section A.3) where one dimension is prioritized over another. In the ServerFarm:size dataset ZO and LW prioritize the path dimension. The reasons are twofold. The first reason is that the `size` attribute is stored as a 64 bit integer since 32 bit integers cannot cope with file sizes above $2^{32} \approx 4.3$GB. The file sizes in the dataset are heavily skewed towards small files (few bytes or kilo-bytes) and thus have many leading zero-bytes. Many of these most significant bytes do not partition the values. On the other hand, the leading path bytes immediately partition the data: the second path byte is discriminative since the top level of a file system contains folders like `/usr`, `/etc`, `/var`. As a result, ZO and LW fail to interleave at discriminative bytes and these approaches degenerate to the level of PV. The second reason is specific to ZO. To turn variable-length paths into fixed-length strings, ZO maps path labels with a surrogate function and fills shorter paths with trailing zero-bytes to match the length of the longest path (see Section A.7.1). We need 3 bytes per label and the deepest path contains 21 labels, thus every path is mapped to a length of 63 bytes and interleaved with the 8 bytes of the values. Many paths in the dataset are shorter than the deepest path and have many trailing zero-bytes. As explained earlier the values (64-bit integers) have many leading zero-bytes, thus the interleaved ZO string orders the relevant path bytes before the relevant value bytes, further pushing ZO towards PV. Let us look more closely at the results of query Q_1 in Figure A.9a. VP's runtime suffers because of the high value selectivity σ_V. XML

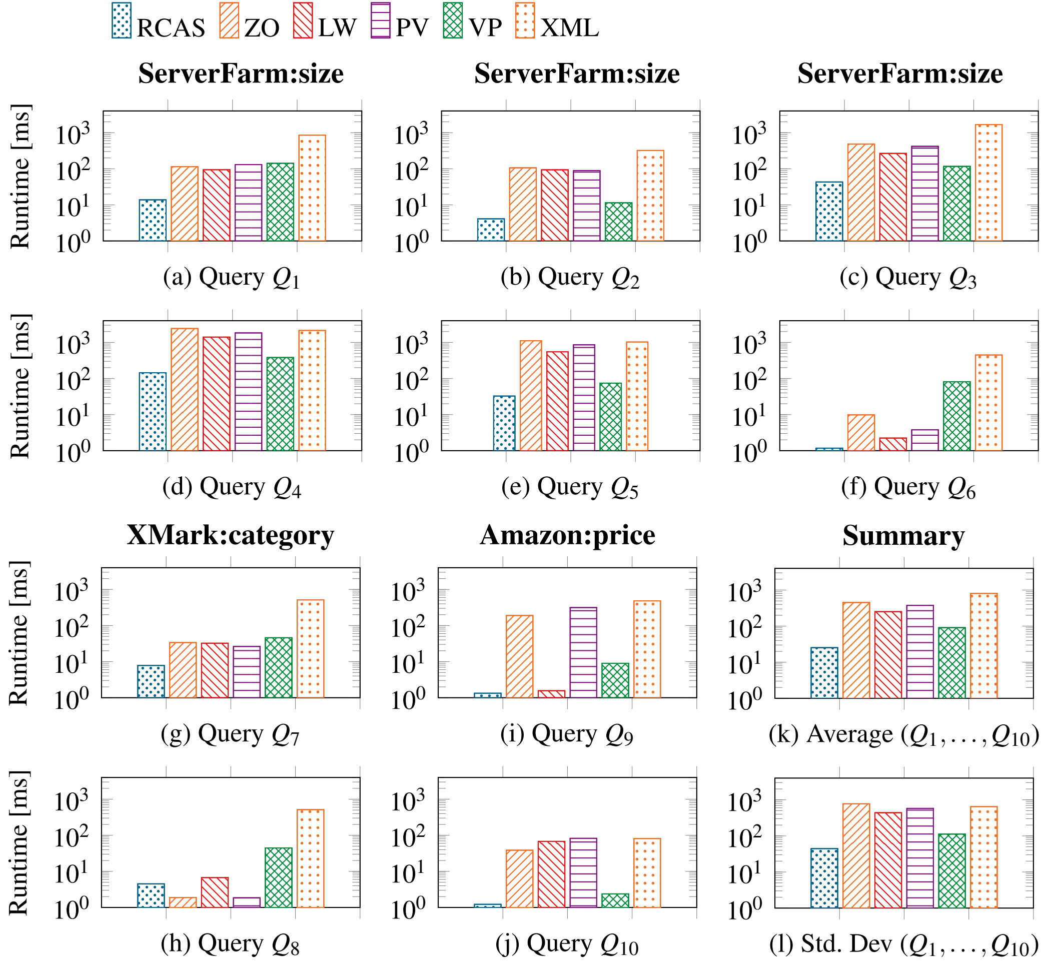


Figure A.9: (a)–(f): Runtime measurements for queries Q_1 to Q_6 from Table A.6 on dataset ServerFarm:size. (g)–(h) queries Q_7 to Q_8 on XMark:category. (i)–(j) queries Q_9 and Q_{10} on Amazon:price. (k)–(l): Average and standard deviation for queries Q_1 to Q_{10}.

performs badly because the intermediate result sizes are one to two orders of magnitude larger than the final result size. RCAS with our dynamic interleaving is unaffected by the puff-pastry effect in ZO, LW, and PV because it only interleaves at discriminative bytes. In RCAS the value selectivity (32%) is counter-balanced by the low path selectivity (2%), thus avoiding VP's pitfall. Lastly, RCAS does not materialize large intermediate results as XML does.

Queries Q_1 and Q_2 have the same path predicate, but Q_2's value selectivity σ_V is considerably lower. The query performance of ZO, LW, and PV is unaffected by the lower σ_V since σ_P remains unchanged. This is further evidence that ZO and LW prioritize the paths, just like PV. The runtime of VP and XML benefit the most if the value selectivity σ_V drops. RCAS's runtime still halves with respect to Q_1 and again shows the best overall runtime.

Query Q_3 has the largest individual path and value selectivities, and therefore produces the largest intermediate results. This is the reason for its high query runtime. RCAS has the best performance since the final result size is an order of magnitude smaller than for the individual path and value predicates.

Queries Q_4 and Q_5 look for particular files (`Makefile`, `README`) within large hierarchies. Their low path selectivity σ_P should favor ZO, LW, and PV, but this is not the case. Once Algorithm A.4 encounters the descendant axis during query processing, it needs to recursively traverse all children of path nodes ($n.D = P$). Fortunately, value nodes ($n.D = V$) can still be used to prune entire subtrees. Therefore, approaches that alternate between discriminative path and values bytes, like RCAS, can still effectively narrow down the search, even though the descendant axis covers large parts of the index. Instead, approaches that prioritize the paths (PV, ZO, LW) perform badly as they cannot prune subtrees high up during query processing.

Query Q_6 has a very low path selectivity σ_P, but its value selectivity σ_V is high. This is the worst case for VP as it can evaluate the path predicate only after traversing already a large part of the index. This query favors PV, LW, and ZO. Nevertheless, RCAS outperforms all other approaches.

The results for queries Q_7 and Q_8 on the XMark:category dataset are shown in Figures A.9g and A.9h. The gaps between the various approaches is smaller since the number of unique paths is small (see Section A.7.2). As a result, the matching paths are quickly found by ZO, LW, and PV. RCAS answers Q_8 in 4ms in comparison to 2ms for ZO and PV. VP performs worse on query Q_8 because of Q_8's low σ_P and high σ_V.

Query Q_9 on dataset Amazon:price searches for all phones and their accessories priced between \$100 and \$200. Selectivities σ_P and σ_V are roughly 4.5% whereas the total selectivity is two orders of magnitude smaller. Figure A.9i confirms that RCAS is the most efficient approach. Query Q_{10} looks for all women's sweaters priced between \$70 and \$100. Sweaters exist for various child-categories of category `Women`, e.g., `Clothing`, `Maternity`, etc. Query Q_{10} uses the wildcard $\star$ to match all of them.

Figures A.9k and A.9l show the average runtime and the standard deviation for queries Q_1 to Q_{10}. RCAS is the most robust approach: it has the lowest average runtime and standard deviation.

A.7.4 Evaluation of Cost Model

This section uses the cost function $\widehat{C}(o,h,\phi,\varsigma_P,\varsigma_V)$ from Section A.5.3 to estimate the cost of answering a query with RCAS. We explain the required steps for query Q_1 on dataset ServerFarm:size (see Table A.6). First, we choose the alternating pattern of discriminative bytes in RCAS's dynamic interleaving by setting ϕ to $(V,P,V,P,\ldots)$. For determining h and o we consider $|K|$ unique keys. Since each leaf represents a key, there are $o^h = |K|$ leaves. We set h to the average depth of a node in the RCAS index (truncated to the next lower integer) and fanout o to $\sqrt[h]{|K|}$. For dataset ServerFarm:size we have $|K| = 9.3\text{M}$ and $h = 13$ (see Table A.5 and Figure A.8), thus $o = \sqrt[13]{9.3\text{M}} = 3.43$. This is consistent with Figure A.8a that shows that the most frequent node type in RCAS has a fanout of at most four. Next we look at parameters ς_P and ς_V. In our cost model, a query traverses a constant fraction ς_D of the children on each level of dimension D, corresponding to a selectivity of $\sigma_D = \varsigma_D \cdot \varsigma_D \cdot \ldots = \varsigma_D^N$ over all N levels of dimension D. N is the number of discriminative bytes in dimension D. Thus, if a CAS query has a value selectivity of σ_V we set $\varsigma_V = \sqrt[N]{\sigma_V}$. The value selectivity of query Q_1 is $\sigma_V = 32.9\%$ (see Table A.6); the number of discriminative value bytes in ϕ is $N = \lceil h/2 \rceil = \lceil 13/2 \rceil = 7$ (we use the ceiling since we start partitioning by V in ϕ), thus $\varsigma_V = \sqrt[7]{0.329} = 0.85$. ς_P is determined likewise and yields $\varsigma_P = \sqrt[\lfloor 13/2 \rfloor]{0.02} = 0.52$ for Q_1.

We refine this cost model for path predicates containing the descendant axis `//` or the wildcard $\star$ followed by further path steps. In such cases we use the path selectivity of the path predicate up to the first descendant axis or wildcard. For example, for query Q_4 with path predicate `/usr/share//Makefile` and $\sigma_P = 0.03\%$, we use the path predicate `/usr/share//` with a selectivity of $\sigma_P = 44\%$. This is so because the low path selectivity of the original predicate is not representative for the number of nodes that RCAS must traverse. As soon as RCAS hits a descendant axis in the path predicate it can only use the value predicate to prune nodes during the traversal (see Section A.6.4).

Table A.7: Estimated cost $\widehat{C}$ and true cost C for queries Q_1 to Q_{10}.

	$\widehat{C}$	C	E
Q_1	105 793	83 190	1.27
Q_2	19 157	28 943	1.51
Q_3	542 458	273 824	1.98
Q_4	710 128	784 068	1.10
Q_5	111 139	146 124	1.31

	$\widehat{C}$	C	E
Q_6	9920	3062	3.24
Q_7	34 513	30 365	1.14
Q_8	18 856	38 247	2.03
Q_9	20 421	4219	4.84
Q_{10}	17 993	10 698	1.68

In Table A.7 we compare the estimated cost $\widehat{C}$ for the ten queries in Table A.6 with the true cost of these queries in RCAS. In addition to the estimated cost $\widehat{C}$ and the true cost C, we show the factor $E = {}^{\max(\widehat{C},C)}/_{\min(\widehat{C},C)}$ by which the estimate is off. On average the cost model and RCAS are off by only a factor of two.

Figure A.10 illustrates that the default values we have chosen for the parameters of $\widehat{C}$ yield near optimal results in terms of minimizing the average error $\overline{E}$ for queries Q_1 to Q_{10}. On the x-axis, we plot the deviation from the default value of a parameter. The values for o and h are spot on. We overestimate the true path and value selectivities by a small margin; decreasing $\Delta\varsigma_P$ and $\Delta\varsigma_V$ by 0.04 improves the error marginally.

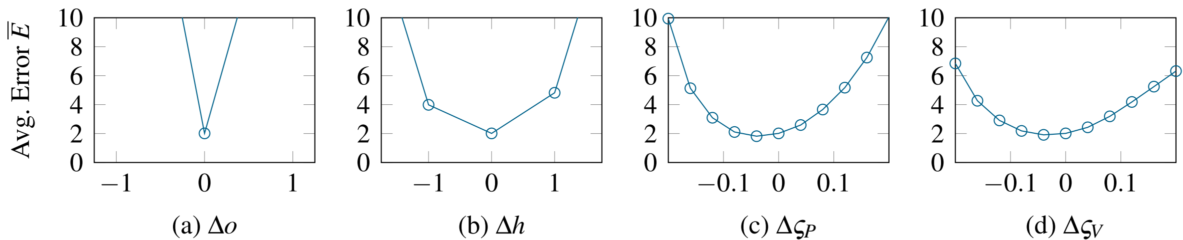


Figure A.10: Calibrating the cost model

A.7.5 Space Consumption and Scalability

Figure A.11 illustrates the space consumption of the indexes for our datasets. RCAS, ZO, LW, PV, and VP all use the same underlying trie structure and have similar space requirements. The XMark:category dataset can be compressed more effectively using tries because the number of unique paths and values is low (see Section A.7.2) and common prefixes need to be stored only once. The trie indexes on Amazon:price do not compress the data as well as on the other two datasets since the lengthy titles of products do not share long common prefixes. The XML index needs to maintain two structures, a DataGuide and a B+ tree. Consequently, its space consumption is higher. In addition, prefixes are not compressed as effectively in a B+ tree as they are in a trie.

In Figure A.12 we analyze the scalability of the indexes in terms of their space consumption and bulk-loading time as we increase the number of (k,r) pairs to 100M. We scale the ServerFarm:size dataset as needed to reach a certain number of (k,r) pairs. The space consumption (Figure A.12a) and the index bulk-loading time (Figure A.12b) are linear in the number of keys. The small drop for RCAS, ZO, LW, PV, and VP in Figure A.12a is due to their trie structure.

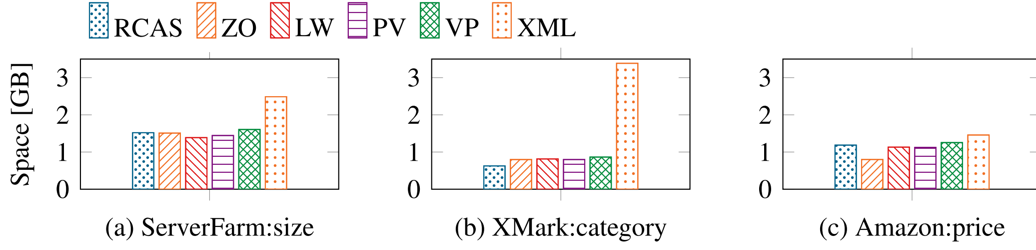


Figure A.11: Space consumption

These indexes compress the keys more efficiently when we scale the dataset from originally 21M keys to 100M keys (we do so by duplicating keys). They store the path $k.P$ and value $k.V$ only once for pairs (k, r_i) and (k, r_j) with the same key k. Figure A.12b confirms that the time complexity of Algorithm A.3 to bulk-load the RCAS index is linear in the number of keys (see Lemma A.2). Bulk-loading RCAS is a factor of two slower than bulk-loading indexes for static interleavings, but a factor of two faster than bulk-loading the XML index. While RCAS takes longer to create the index it offers orders of magnitude better query performance as shown before. Figure A.12 shows that the RCAS index for 100M keys requires 2GB of memory and can be bulk-loaded in less than 5 minutes.

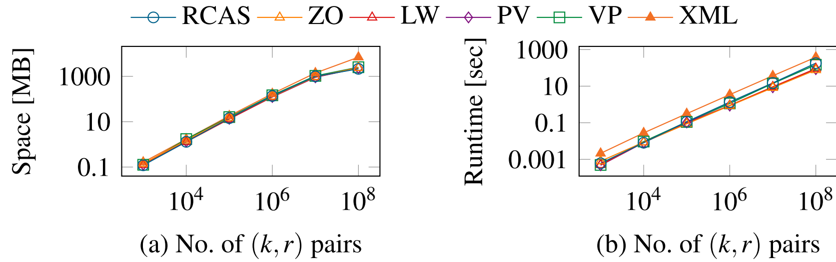


Figure A.12: Space consumption and bulk-loading time

A.7.6 Summary

We conclude with a summary of the key findings of our experimental evaluation. First, RCAS shows the most robust query performance for a wide set of CAS queries with varying selectivities: it exhibits the most stable runtime in terms of average and standard deviation, outperforming other state-of-the-art approaches by up to two orders of magnitude. Second, our cost model yields a good estimate for the true cost of a CAS query on the RCAS index. Third, because of the trie-based structure the RCAS index consumes less space than its competitors, requiring only

a third of the space used by a B+ tree-based approach. The space consumption and bulk-loading time are linear in the number of keys, allowing it to scale to a large number of keys.

A.8 Conclusion and Outlook

We propose the Robust Content-and-Structure (RCAS) index for semi-structured hierarchical data, offering a well-balanced integration of paths and values in a single index. Our scheme avoids prioritizing a particular dimension (paths or values), making the index robust against queries with high individual selectivities producing large intermediate results and a small final result. We achieve this by developing a novel dynamic interleaving scheme that exploits the properties of path and value attributes. Specifically, we interleave paths and values at their *discriminative bytes*, which allows our index to adapt to a given data distribution. In addition to proving important theoretical properties, such as the monotonicity of discriminative bytes and robustness, we show in our experimental evaluation impressive gains: utilizing dynamic interleaving our RCAS index outperforms state-of-the-art approaches by up to two order of magnitudes on real-world and synthetic datasets.

Future work points into several directions. We plan to apply RCAS on the largest archive of software source code, the Software Heritage Dataset [DCZ17, PSZ20]. On the technical side we are working on supporting incremental insertions and deletions in the RCAS index. It would also be interesting to explore a disk-based RCAS index based on a disk-based trie [AZ09]. Further, we consider making path predicates more expressive by, e.g., allowing arbitrary regular expressions as studied by Baeza-Yates et al. [BG96]. On the theoretical side it would be interesting to investigate the length of the dynamic interleavings for different data distributions.

APPENDIX B

Inserting Keys into the Robust Content-and-Structure (RCAS) Index

Reprinted from:

K. Wellenzohn, L. Popovic, M. H. Böhlen, S. Helmer. “Inserting Keys into the Robust Content-and-Structure (RCAS) Index”, in *ADBIS*, pages 121–135, 2021.
doi:10.1007/978-3-030-82472-3_10

Abstract

Semi-structured data is prevalent and typically stored in formats like XML and JSON. The most common type of queries on such data are Content-and-Structure (CAS) queries, and a number of CAS indexes have been developed to speed up these queries. The state-of-the-art is the RCAS index, which properly interleaves content and structure, but does not support insertions of single keys. We propose several insertion techniques that explore the trade-off between insertion and query performance. Our exhaustive experimental evaluation shows that the techniques are efficient and preserve RCAS's good query performance.

B.1 Introduction

A large part of real-world data does not follow the rigid structure of tables found in relational database management systems (RDBMSs). Instead, a substantial amount of data is semi-structured, e.g., annotated and marked-up data stored in formats such as XML and JSON. Since mark-up elements can be nested, this leads to a hierarchical structure. A typical example of semi-structured data are bills of materials (BOMs), which contain the specification of every component required to manufacture end products. Figure B.1 shows an example of a hierarchical representation of three products, with their components organized under a node `bom`. Nodes in a BOM can have attributes, e.g., in Figure B.1 attribute `@weight` denotes the weight of a component in grams.

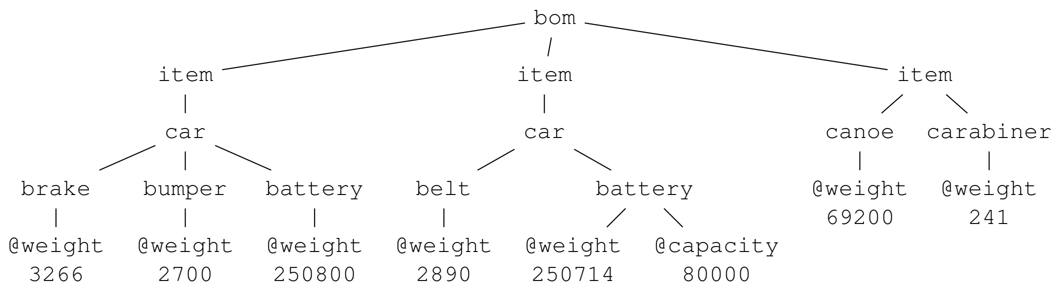


Figure B.1: Example of a bill of materials (BOM).

Semi-structured hierarchical data is usually queried via content-and-structure (CAS) queries [MHSB15] that combine a value predicate on the content of some attribute and a path predicate on the location of this attribute in the hierarchical structure. An example query for the BOM

depicted in Figure B.1 that selects all car parts with a weight between 1000 and 3000 grams has the form: Q (/bom/item/car//, [1000, 3000]), with "//" matching a node and all its descendants. To speed up this type of query, the Robust Content-and-Structure (RCAS) index has been proposed [WBH20]. RCAS is based on a new interleaving scheme, called dynamic interleaving, that adapts to the distribution of the data and interleaves path and value dimension at their discriminative bytes.

So far, the RCAS index supports bulk-loading but it cannot be updated incrementally. We present efficient methods to insert new keys into RCAS without having to bulk-load the index again. We make the following contributions:

- We develop two different strategies for inserting keys into an RCAS index: strict and lazy restructuring.
- With the help of an auxiliary index, we mitigate the effects of having to restructure large parts of the index during an insertion. We propose techniques to merge the auxiliary index back into the main index if it grows too big.
- Extensive experiments demonstrate that combining lazy restructuring with the auxiliary index provides the most efficient solution.

B.2 Background

RCAS is an in-memory index that stores composite keys k consisting of two components: a path dimension P and a value dimension V that are accessed by $k.P$ and $k.V$, respectively. An example of a key (representing an entity from Figure B.1) is (/bom/item/car/bumper$, *00 00 0A 8C*), where the blue part is the key's path and the red part is the key's value (in hexadecimal). Table B.1 shows the keys of all entities from Figure B.1; the example key is k_7.

The RCAS index interleaves the two-dimensional keys at their discriminative path and value bytes. The discriminative byte $\mathsf{dsc}(K, D)$ of a set of keys K in a given dimension D is the position of the first byte for which the keys differ. That is, the discriminative byte is the first byte after the keys' longest common prefix in dimension D. For example, the discriminative path byte $\mathsf{dsc}(\mathsf{K}^{1..7}, P)$ of the set of keys $\mathsf{K}^{1..7}$ from Table B.1 is the 13th byte. All paths up to the 13th byte share the prefix /bom/item/ca and for the 13th byte, key k_1 has value n, while keys $\mathsf{k}_2, \ldots, \mathsf{k}_7$

have value `r`. The dynamic interleaving is obtained by interleaving the keys alternatingly at their discriminative path and value bytes.

Table B.1: Set $\mathsf{K}^{1..7} = \{\mathsf{k}_1, \dots, \mathsf{k}_7\}$ of composite keys.

	Path Dimension P	Value Dimension V
k_1	`/bom/item/canoe$`	69200 (*00 01 0E 50*)
k_2	`/bom/item/carabiner$`	241 (*00 00 00 F1*)
k_3	`/bom/item/car/battery$`	250714 (*00 03 D3 5A*)
k_4	`/bom/item/car/battery$`	250800 (*00 03 D3 B0*)
k_5	`/bom/item/car/belt$`	2890 (*00 00 0B 4A*)
k_6	`/bom/item/car/brake$`	3266 (*00 00 0C C2*)
k_7	`/bom/item/car/bumper$`	2700 (*00 00 0A 8C*)
	1 3 5 7 9 11 13 15 17 19 21 23	1 2 3 4

The dynamic interleaving adapts to the data: when interleaving at a discriminative byte, we divide keys into different partitions. If we instead use a byte that is part of the common prefix, all keys will end up in the same partition, which means that during a search we cannot filter keys efficiently. Our scheme guarantees that in each interleaving step we narrow down the set of keys to a smaller set of keys that have the same value for the discriminative byte. Eventually, the set is narrowed down to a single key and its dynamic interleaving is complete. Switching between discriminative path and value bytes gives us a robust query performance since it allows us to evaluate the path and value predicates of CAS queries step by step in round-robin fashion.

We embed the dynamically interleaved keys $\mathsf{K}^{1..7}$ from Table B.1 into a trie data structure as shown in Figure B.2, building the final RCAS index. Each node n stores a path substring s_P (blue), a value substring s_V (red), and a dimension D. s_P and s_V contain the longest common prefixes in the respective dimensions of all the nodes in the subtree rooted at n. Dimension D determines the dimension that is used for partitioning the keys contained in the subtree rooted in n; D is either P or V for an inner node and $\perp$ for a leaf node. Leaf nodes store a set of references that point to nodes in the hierarchical document. In Figure B.2 node n_9 stores the longest common prefixes $s_P =$ `r/battery$` and $s_V =$ *03 D3*. $\mathsf{n}_9.D = V$, which means the children of n_9 are distinguished according to their value at the discriminative value byte (e.g., ***5A*** for n_{10} and ***B0*** for n_{11}). For more details on dynamic interleaving, building an RCAS index, and querying it efficiently, see [WBH20]. Here we focus on inserting new keys into RCAS indexes.

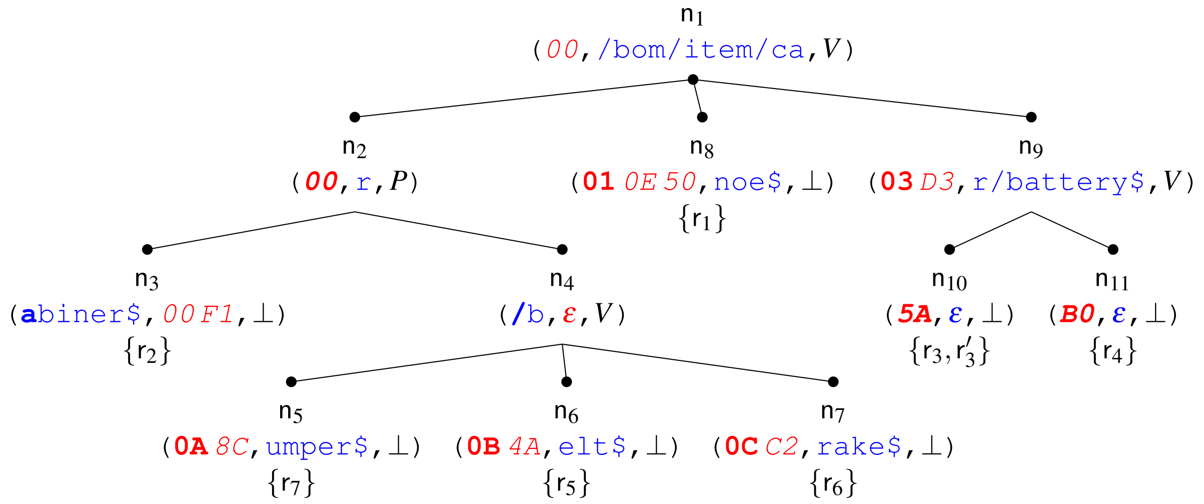


Figure B.2: The RCAS index for the composite keys $\mathsf{K}^{1..7}$.

B.3 Insertion of New Keys

We distinguish three insertion cases for which the effort varies greatly:

Case 1. The inserted key is a duplicate, i.e., there is already an entry in the index for the same key. Thus, we add a reference to the set of references in the appropriate leaf node. For instance, if we insert a new key k_3' that is identical to k_3, we only add the reference r_3' to the set of references of node n_{10} (see Figure B.2).

Case 2. The key to be inserted deviates from the keys in the index, but it does so at the very end of the trie structure. In this case, we add one new leaf node and a branch just above the leaf level. In Figure B.3 we illustrate RCAS after inserting key (`/bom/item/car/bench$`, *00 00 19 64*) with reference r_9. We create a new leaf node n_{12} and add a new branch to its parent node n_4.

Case 3. This is the most complex case. If the path and/or the value of the new key results in a mismatch with the path and/or value of a node in the index, the index must be restructured. This is because the position of a discriminative byte shifts, making it necessary to recompute the dynamic interleaving of a potentially substantial number of keys in the index. For example, if we want to insert key (`/bom/item/cassette$`, *00 00 AB 12*) with reference r_{10}, due to its value *00* at the discriminative value byte (the second byte), it has to be inserted into the subtree rooted at node n_2 (see Figure B.3). Note that the discriminative path byte has decreased by one position since the first `s` in `cassette` differs from `r` in the path substring $\mathsf{n}_2.s_P$. This

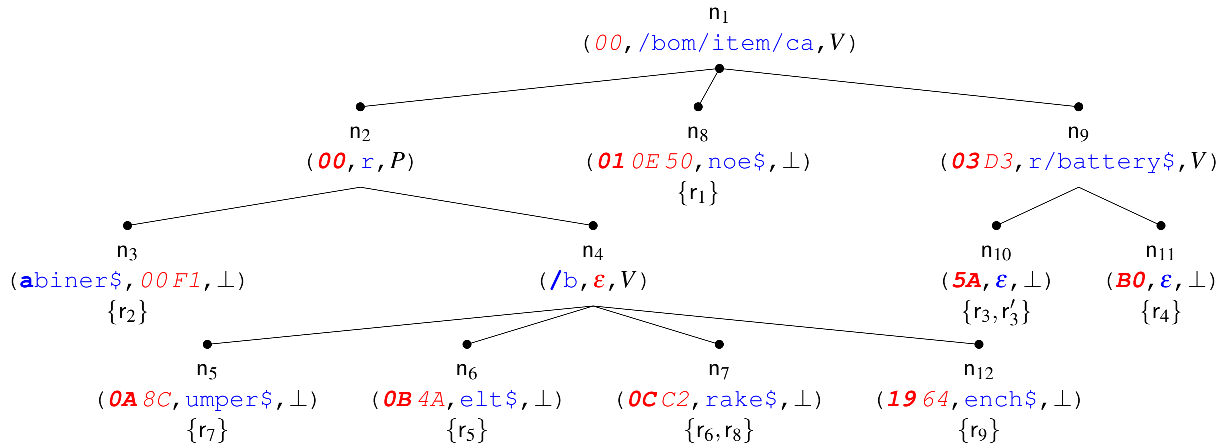


Figure B.3: Inserting a new key just above leaf level.

invalidates the current dynamic interleaving of the keys in the subtree rooted at n_2. Consequently, the whole subtree has to be restructured. As this is the most complicated case and there is no straightforward answer on how to handle it, we look at it in Section B.4.

Insertion Algorithm. Algorithm B.1 inserts a key into RCAS. The input parameters are the root node n of the trie, the key k to insert, and a reference r to the indexed element in the hierarchical document. The algorithm descends to the insertion point of k in the trie. Starting from the root node n, we compare the current node's path and value substring with the relevant part in k's path and value (lines 2–3). As long as these strings coincide we proceed. Depending on the current node's dimension, we follow the edge that contains k's next path or value byte. The descent stops once we reach one of the three cases from above. In Case 1, we reached a leaf node and add r to the current node's set of references (lines 4–6). In Case 2, we could not find the next node to traverse, thus we create it (lines 13–19). The new leaf's substrings s_P and s_V are set to the still unmatched bytes in $k.P$ and $k.V$, respectively, and its dimension is set to $\perp$. In Case 3 we discovered a mismatch between k and the current node's substrings (lines 7–10).

B.4 Index Restructuring during Insertion

Algorithm B.1: Insert(n, k, r)

```
1  while true do
2      Compare n.s_P to relevant part of k.P
3      Compare n.s_V to relevant part of k.V
4      if k.P and k.V have completely matched n.s_P and n.s_V then      // Case 1
5          Add reference r to node n
6          return
7      else if mismatch between k.P and n.s_P or k.V and n.s_V then     // Case 3
8          // detailed description later
9          Insert k into restructured subtree rooted at n
10         return
11     Let n_p = n
12     Let n be the child of n with the matching discriminative byte
13     if n = NIL then                                                   // Case 2
14         Let n = new leaf node
15         Initialize n.s_P and n.s_V with remainder of k.P and k.V
16         Set n.D to ⊥
17         Insert r into n
18         Insert n into list of children of n_p
19         return
```

B.4.1 Strict Restructuring

The shifting of discriminative bytes in Case 3 invalidates the current dynamic interleaving and if we want to preserve it we need to recompute it. An approach that achieves this collects all keys rooted in the node where the mismatch occurred (in the example shown above, the mismatch occurred in node n_2), adds the new key to it, and then applies the bulk-loading algorithm to this set of keys. This creates a new dynamic interleaving that is embedded in a trie and replaces the old subtree. We call this method *strict restructuring*. It guarantees a strictly alternating interleaving in the index, but the insertion operation is expensive if a large subtree is replaced. Figure B.4 shows the RCAS index after inserting the key (`/bom/item/cassette$`, *00 00 AB 12*).

Strict restructuring (Algorithm B.2) takes four input parameters: the root node n of the subtree where the mismatch occurred, its parent node n_p (which is equal to NIL if n is the root), the new key k, and a reference r to the indexed element in the hierarchical document. See Section B.6 for a complexity analysis.

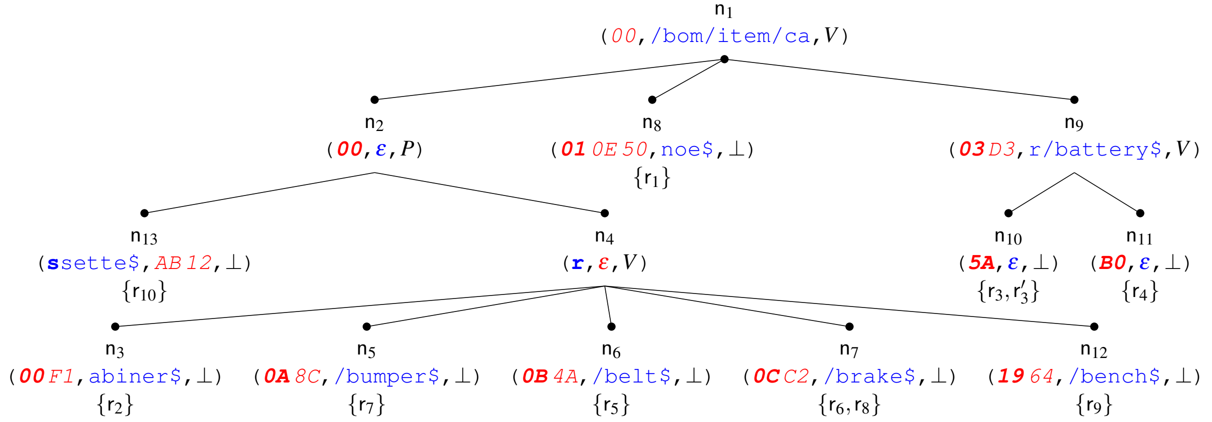


Figure B.4: Inserting a key with the strict restructuring method.

Algorithm B.2: StrictRestructuring(n, n_p, k, r)

1 Let c = the set of all keys and their references rooted in n
2 Let $c = c \cup \{(k,r)\}$
3 Let $D = n.D$ `// dimension used for starting interleaving`
4 Let n' = bulkload(c, D)
5 **if** n_p = NIL **then** replace original trie with n'; `// n is root node`
6 **else** replace n with n' in n_p;

B.4.2 Lazy Restructuring

Giving up the guarantee of a strictly alternating interleaving allows us to insert new keys more quickly. The basic idea is to add an intermediate node n'_p that is able to successfully distinguish its children: node n, where the mismatch happened and a new sibling n_k that represents the new key k. The new intermediate node n'_p will contain all path and value bytes that are common to node n and key k. Consequently, path and value substrings of n and n_k contain all bytes that are not moved to n'_p. Node n is no longer a child of its original parent n_p, this place is taken by n'_p. We call this method *lazy restructuring*. While it does not guarantee a strictly alternating interleaving, it is much faster than strict restructuring, as we can resolve a mismatch by inserting just two nodes: n'_p and n_k. Figure B.5 shows RCAS after inserting the key (/bom/item/cassette$, *00 00 AB 12*) lazily. Node n_{13} and its child n_2 partition the data both in the path dimension ($n.D = P$) and therefore violate the strictly alternating pattern.

Inserting a key with lazy interleaving introduces small irregularities that are limited to the dynamic interleaving of the keys in node n's subtree. These irregularities slowly *separate* (rather

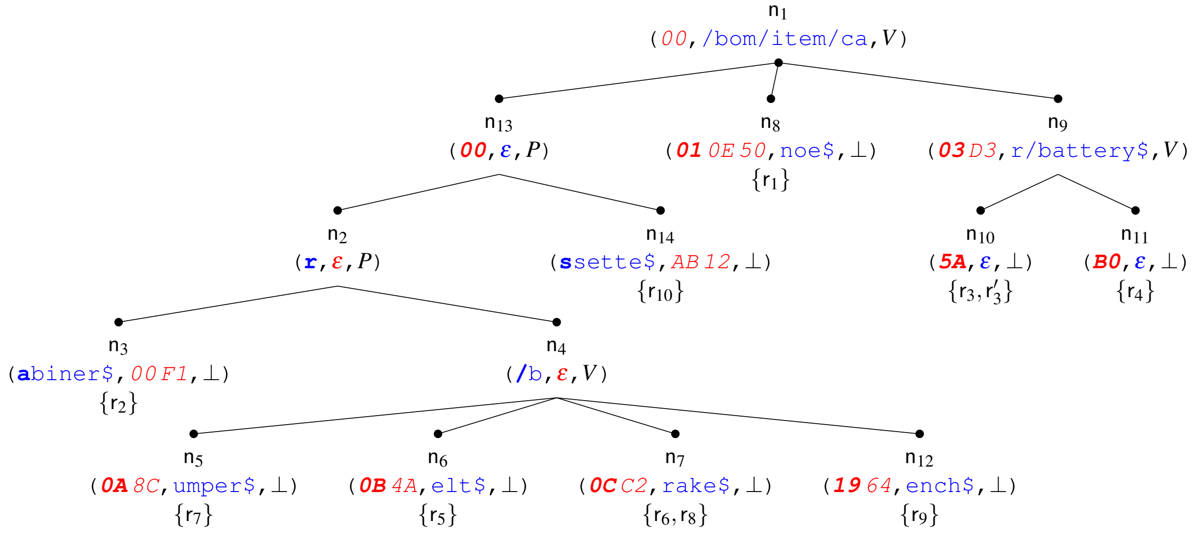


Figure B.5: Inserting a new key with lazy restructuring.

than *interleave*) paths and values if insertions repeatedly force the algorithm to split the same subtree in the same dimension. On the other hand, lazy restructuring can also repair itself when an insertion forces the algorithm to split in the opposite dimension. We show experimentally in Section B.7 that lazy restructuring is fast and offers good query performance.

Algorithm B.3 shows the pseudocode for lazy restructuring, it takes the same parameters as Algorithm B.2. First, we create a new inner node n'_p and then determine which dimension to use for partitioning. If only a path mismatch occurred between n and k, we have to use P. In case of a value mismatch, we have to use V. If we have mismatches in both dimensions, then we take the opposite dimension of parent node n_p to keep up an alternating interleaving as long as possible. The remainder of the algorithm initializes s_P and s_V with the longest common prefixes of n and k and creates two partitions: one containing the original content of node n and the other containing the new key k. The partition containing k is stored in a new leaf node n_k. We also have to adjust the prefixes in the nodes n and n_k. Finally, n and n_k are inserted as children of n'_p, and n'_p replaces n in n_p. See Section B.6 for a complexity analysis.

Algorithm B.3: LazyRestructuring(n, n_p, k, r)

```
1  Let n'_p = new inner node
2  Let n'_p.D = determineDimension()
3  Let n'_p.s_P = longest common path prefix of n and k
4  Let n'_p.s_V = longest common value prefix of n and k
5  Let n_k = new leaf node
6  Insert n and n_k as children of n'_p
7  Adjust s_P and s_V in n and n_k
8  Insert r into n_k
9  if n_p = NIL then replace original trie with n'_p;        // n is root node
10 else replace n with n'_p in n_p;
```

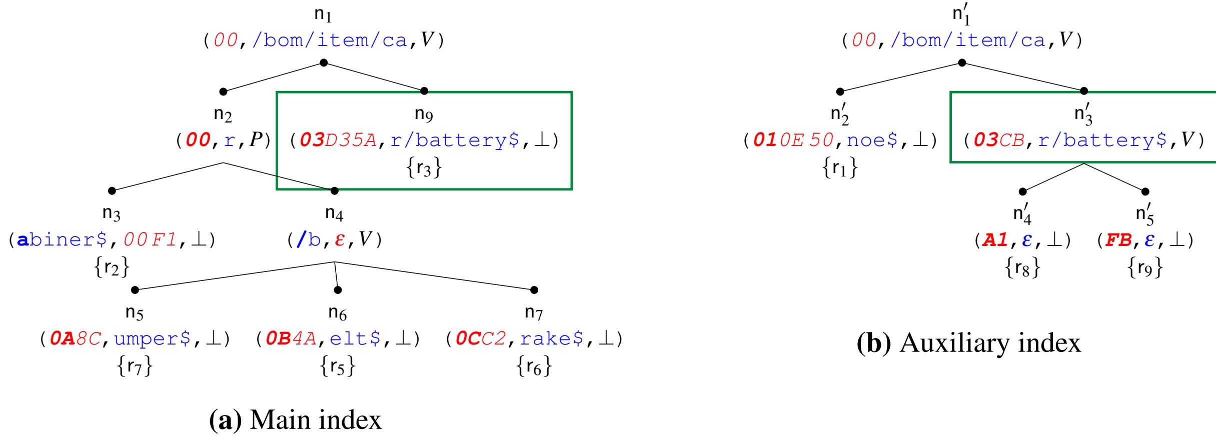
B.5 Utilizing an Auxiliary Index

Using differential files to keep track of changes in a data collection is a well-established method (e.g., LSM-trees [OCGO96]). Instead of updating an index in-place, the updates are done out-of-place in auxiliary indexes and later merged according to a specific policy. We use the general idea of auxiliary indexes to speed up the insertion of new keys into an RCAS index. However, we apply this method slightly differently: we insert new keys falling under Case 1 and Case 2 directly into the main RCAS index, since these insertions can be executed efficiently. Only the keys in Case 3 are inserted into an auxiliary RCAS index. As the auxiliary index is much smaller than the main index, the strict restructuring method can be processed more efficiently on the auxiliary index. Sometimes a Case 3 insertion into the main index even turns into a Case 1/2 insertion into the auxiliary index, as it contains a different set of keys. For an even faster insertions we can use lazy restructuring in the auxiliary index.

There is a price to pay for using an auxiliary index: queries now have to traverse two indexes. However, this looks worse than it actually is, since the total number of keys stored in both indexes is the same. We investigate the trade-offs of using an auxiliary index and different insertion strategies in Section B.7.

Using an auxiliary index only makes sense if the expensive insertion operations (Case 3) can be executed much more quickly on the auxiliary index. To achieve this, we have to merge the auxiliary index into the main index from time to time. This is more efficient than individually inserting new keys into the main index, though, as the restructuring of a subtree in the main index during the merge operation usually covers multiple new keys in one go rather than restructuring a subtree for every individual insertion. We consider two different merge strategies. The simplest,

but also most time-consuming, method is to collect all keys from the main and auxiliary index and to bulk-load them into a new index.



(a) Main index

(b) Auxiliary index

Figure B.6: A more sophisticated method merges only subtrees that differ.

Algorithm B.4: Merge(n_m, n_a)

1. **if** $n_a \neq$ NIL **then**
2.   **if** nodes n_m and n_a match **then**
3.   **foreach** child c_a of n_a **do**
4.   Find corresponding child c_m of n_m
5.   **if** c_m does not exist **then** relocate c_a to n_m;
6.   **else** merge(c_m, c_a);
7.   **else**
8.   Let $K =$ all keys in subtrees n_m and n_a
9.   Let $n'_m =$ bulkload(K)
10.   Replace n_m with n'_m in main index

A more sophisticated method traverses the main and auxiliary index in parallel and only if the path and/or value substrings of two corresponding nodes do not match, we restructure this subtree, bulk-load the keys into it, and insert it into the main index. Figure B.6 illustrates this method. Since root nodes n_1 and n'_1 match, the algorithm proceeds to its children. For child n'_2 we cannot find a corresponding child in the main index, hence we relocate n'_2 to the main index. For child n'_3 we find child n_9 in the main index and since they differ in the value substring, the subtrees rooted in these two trees are merged. Notice that child n_2 in the main index is not affected by the merging. Algorithm B.4 depicts the pseudocode. It is called with the root of the main index n_m and the root of the auxiliary index n_a. If n_m's and n_a's values of substrings s_P, s_V and dimension

D match, we recursively merge the corresponding children of n_m and n_a. Otherwise, we collect all keys rooted in n_m and n_a and bulk-load a new subtree.

B.6 Analysis

We first look at the complexity of Case 1 and 2 insertions, which require no restructuring (see Section B.3). Inserting keys that require no restructuring takes $O(h)$ time, where h is the height of RCAS, since Algorithm B.1 descends the tree in $O(h)$ time and in Case 1 the algorithm adds a reference in $O(1)$ time, while in Case 2 the algorithm adds one leaf in $O(1)$ time. The complexity of Case 3, which requires restructuring, depends on whether we use lazy or strict restructuring.

Theorem B.1. *Inserting a key into RCAS with lazy restructuring takes $O(h)$ time.*

Proof. The insertion algorithm descends the tree to the position where a path or value mismatch occurs in $O(h)$ time. To insert the key, lazy restructuring adds two new nodes in $O(1)$ time. □

Theorem B.2. *Inserting a key into RCAS with strict restructuring takes $O(l \cdot N)$ time, where l is the length of the longest key and N is the number of keys.*

Proof. Descending the tree to the insertion position takes $O(h)$ time. In the worst case, the insertion position is the root node, which means strict restructuring collects all keys in RCAS in $O(N)$ time, and bulk-loads a new index in $O(l \cdot N)$ time using the bulk-loading algorithm from [WBH20]. □

The complexity of Case 3 insertions into the auxiliary index (if it is enabled) depends on the insertion technique and is $O(h)$ with lazy restructuring (Lemma B.1) and $O(l \cdot N)$ with strict restructuring (Lemma B.2). In practice, insertions into the auxiliary index are faster because it is smaller. This requires that the auxiliary index is merged back into the main index when it grows too big.

Theorem B.3. *Merging an RCAS index with its auxiliary RCAS index using Algorithm B.4 takes $O(l \cdot N)$ time.*

Proof. In the worst case, the root nodes of the RCAS index and its auxiliary RCAS index mismatch, which means all keys in both indexes are collected and a new RCAS index is bulk-loaded in $O(l \cdot N)$ time [WBH20]. □

B.7 Experimental Evaluation

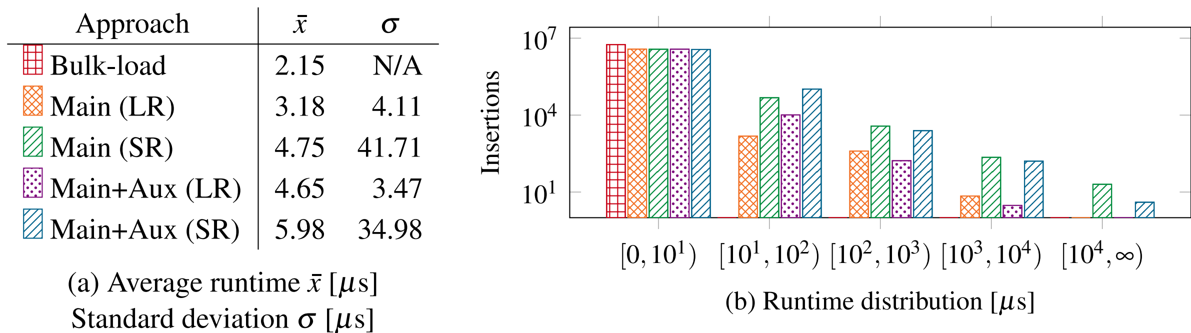
Setup. We use a virtual Ubuntu server with 8GB of main memory and an AMD EPCY 7702 CPU with 1MB L2 cache. All algorithms are implemented in C++ and compiled with g++ (version 10.2.0). The reported runtime measurements represent the average time of 1000 experiment runs.

Dataset. We use the ServerFarm dataset from [WBH20] that contains information about the files on a fleet of 100 Linux servers. The path and value of a composite key denote the full path of a file and its size in bytes, respectively. We eliminate duplicate keys because they trigger insertion Case 1, which does not change the structure of the index (see Section B.3). Without duplicates, the ServerFarm dataset contains 9.3 million keys.

Reproducibility. The code, dataset, and instructions how to reproduce our experiments is available at: https://github.com/k13n/rcas_update.

B.7.1 Runtime of Strict and Lazy Restructuring

We begin by comparing the runtime of lazy restructuring (LR) and strict restructuring (SR) either applied on the main index directly, or applied on the combination of main and auxiliary index (Main+Aux). When the auxiliary index is used, insertion Cases 1 and 2 are performed on the main index, while Case 3 is performed on the auxiliary index with LR or SR. In this experiment we bulk-load 60% of the dataset (5 607 400 keys) and insert the remaining 40% (3 738 268 keys) one-by-one. Bulk-loading RCAS with 5.6M keys takes 12 seconds, which means 2.15μs per key. Figure B.7a shows the average runtime ($\bar{x}$) and the standard deviation (σ) for the different insertion techniques. We first look at LR and SR when they are applied to the main index only.



Approach	$\bar{x}$	σ
Bulk-load	2.15	N/A
Main (LR)	3.18	4.11
Main (SR)	4.75	41.71
Main+Aux (LR)	4.65	3.47
Main+Aux (SR)	5.98	34.98

(a) Average runtime $\bar{x}$ [μs]
Standard deviation σ [μs]

(b) Runtime distribution [μs]

Figure B.7: Runtime of insertions.

LR is very fast with an average runtime of merely 3μs per key. This is expected since LR only needs to insert two new nodes into the index. SR on the other hand, takes on average about 5μs and is thus not significantly slower than LR, on average. The runtime of SR depends greatly on the level in the index where the mismatch occurs. The closer to the root the mismatch occurs, the bigger is the subtree this technique needs to rebuild. Therefore, we expect that even if the average runtime is low, the variance is higher. Indeed, the standard deviation of SR is 41μs compared to 4μs for LR. This is confirmed by the histogram in Figure B.7b, where we report the number of insertions that fall into a given runtime range (as a reference point, we report for bulk-loading that all 5.6M keys have a runtime of 2.15μs per key). While most insertions are quick for all methods, SR has a longer tail and a significantly higher number of slow insertions (note the logarithmic axes). Applying LR and SR to the auxiliary index slightly increases the average runtime since two indexes must be traversed to find the insertion position, but the standard deviation decreases since there are fewer expensive updates to the auxiliary index, see Figure B.7b.

B.7.2 Query Runtime

We look at the query performance after updating RCAS with our proposed insertion techniques. We simulate that RCAS is created for a large semi-structured dataset that grows over time. For example, the Software Heritage archive [ACZ18, DCZ17], which preserves publicly-available source code, grows ca. 35% to 40% a year [RCZ20]. Therefore, we bulk-load RCAS with the at least 60% of our dataset and insert the remaining keys one by one to simulate a year worth of insertions. We expect SR to lead to better query performance than LR since it preserves the dynamic interleaving, while LR can introduce small irregularities. Further, we expect that enabling the auxiliary index does not significantly change the query runtime since the main and auxiliary indexes, when put together, are of similar size as the (main) RCAS index when no auxiliary index is used.

In Figure B.8a we report the average runtime for the six CAS queries from [WBH20]. For example, the first query looks for all files nested arbitrarily deeply in the `/usr/include` directory that are at least 5KB large, expressed as (`/usr/include//`, [`5000`, ∞]). The results are surprising. First, SR in the main index leads to the worst query runtime and the remaining three approaches lead to faster query runtimes. To see why, let us first look at Figure B.8b that shows the number of nodes traversed during query processing (if the auxiliary index is enabled, we sum the number of nodes traversed in both indexes). The queries perform better when the aux-

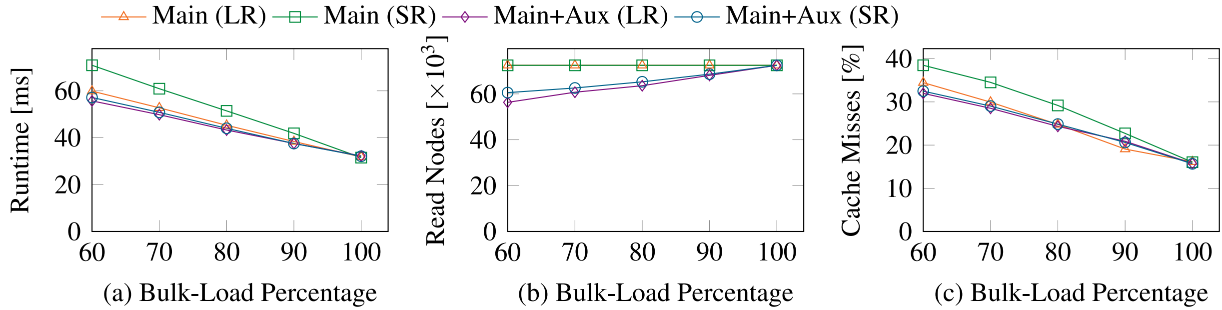


Figure B.8: Query performance.

iliary index is enabled because fewer nodes are traversed during query processing, which means subtrees were pruned earlier.

Figure B.8b does not explain why LR leads to a better query performance than SR since with both approaches the queries need to traverse almost the same number of nodes. To find the reason for the better query runtime we turn to Figure B.8c, which shows that query runtime and the CPU cache misses[1] are highly correlated. SR leads to the highest number of cache misses due to memory fragmentation. When the index is bulk-loaded its nodes are allocated in contiguous regions of the main memory. The bulk-loading algorithm builds the tree depth-first in pre-order and the queries follow the same depth-first approach (see [WBH20]). As a result, nodes that are traversed frequently together have a high locality of reference and thus range queries typically access memory sequentially, which is faster than accessing memory randomly [PHD16]. Inserting additional keys fragments the memory. Strict restructuring (SR) deletes and rebuilds entire subtrees, which can leave big empty gaps between contiguous regions of memory and as a result experiences more cache misses in the CPU during query processing. LR causes fewer cache misses in the CPU than SR because it fragments the memory less. This is because LR always inserts two new nodes whereas SR inserts and deletes large subtrees, which can leave big gaps in memory.

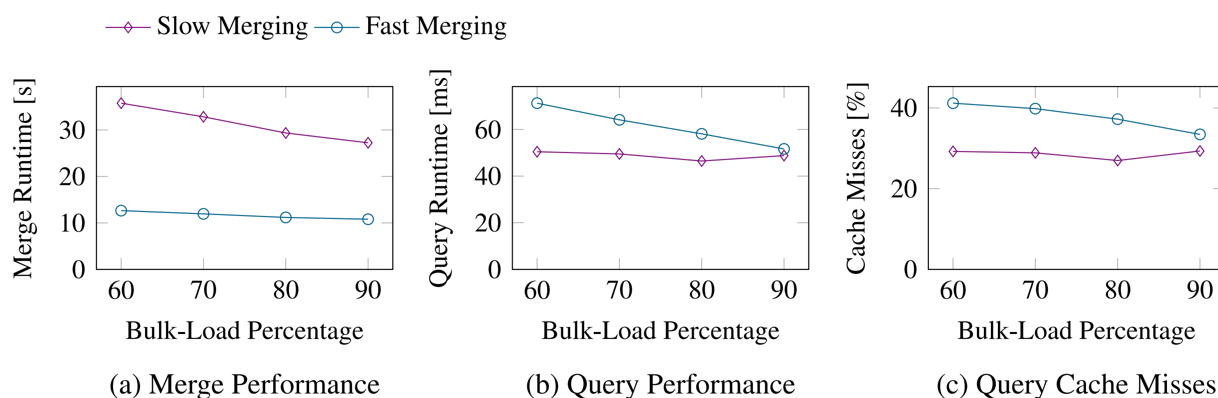
The query runtime improves for all approaches as we bulk-load a larger fraction of the dataset because the number of cache misses decreases. Consider the strict restructuring method in the main index (green curve). By definition, SR structures the index exactly as if the index was entirely bulk-loaded. This can also be seen in Figure B.8b where the number of nodes traversed to answer the queries is constant. Yet, the query runtime improves as we bulk-load more of the

[1] We measure the cache misses with the perf command on Linux, which relies on the Performance Monitoring Unit (PMU) in modern processors to record hardware events like cache accesses and misses in the CPU.

index because the number of cache misses is reduced due to less memory fragmentation (see explanation above). Therefore, it is best to rebuild the index from scratch after inserting many new keys.

B.7.3 Merging of Auxiliary and Main Index

We compare two merging techniques: (a) the slow approach that takes all keys from both indexes and replaces them with a bulk-loaded index, and (b) the fast approach that descends both indexes in parallel and only merges subtrees that differ. In the following experiment we bulk-load the main index with a fraction of the dataset, insert the remaining keys into the auxiliary index, and merge the two indexes with one of the two methods. Figure B.9a shows that fast merging outperforms the slow technique by a factor of three. This is because the slow merging needs to fully rebuild a new index from scratch, while the fast merging only merges subtrees that have actually changed. In addition, if fast merging finds a subtree in the auxiliary index that does not exist it the main index, it can efficiently relocate that subtree to the main index.



(a) Merge Performance (b) Query Performance (c) Query Cache Misses

Figure B.9: Merging and querying performance.

After merging the auxiliary index into the main index, we look at the query runtime of the main index in Figure B.9b. Slow merging leads to a better query performance than fast merging because slow merging produces a compact representation of the index in memory (see discussion above), while fast merging fragments the memory and leads to cache misses in the CPU (Figure B.9c).

B.7.4 Summary

Our experiments show that RCAS can be updated efficiently, but to guarantee optimal query performance it is recommended to rebuild the index occasionally. The best way to insert keys into the RCAS index is to use an auxiliary index with lazy restructuring (LR). LR is faster and leads to better query performance than strict restructuring since it causes fewer cache misses in the CPU during query processing. When the auxiliary index becomes too large, it is best to merge it back into the main RCAS index with the slow merging technique, i.e., the main index is bulk-loaded from scratch including all the keys from the auxiliary index.

B.8 Related Work

Updating RCAS is difficult due to its dynamic interleaving scheme that adapts to the data distribution [WBH20]. Inserting or deleting keys can invalidate the position of the discriminative bytes and change the dynamic interleaving of other keys.

Interleaving bits and bytes is a common technique to build multi-dimensional indexes, e.g., the z-order curve [OM84] interleaves the dimensions bit-wise. These schemes are *static* since they interleave at pre-defined positions (e.g., one byte from one dimension is interleaved with one byte from another dimension). Because the interleaving is static, individual keys can be inserted and deleted without affecting the interleaving of other keys. QUILTS [NY17] devises static interleavings that optimize for a given query workload. However, Nishimura et al. [NY17] do not discuss what happens if the query workload changes and with it the static interleaving scheme, which affects the interleavings of *all* keys.

Existing trie-based indexes, e.g., PATRICIA [Mor68], burst tries [HZW02], B-tries [AZ09], and ART [LKN13], solve insertion Case 3 by adding a new parent node to distinguish between the node where the mismatch happened and its new sibling node. Lazy restructuring is based on this technique, but we must decide in which dimension the parent node partitions the data since we deal with two-dimensional keys.

Using auxiliary index structures to buffer updates is a common technique [JNS+97, OCGO96, SL76]. Log-structured merge trees (LSM-trees [OCGO96]) have been developed to ingest data arriving at high speed, see [LC19] for a recent survey. Instead of updating an index in-place, i.e., overwriting old entries, the updates are done out-of-place, i.e., values are stored in an auxiliary

index and later merged back. We redirect Case 3 insertions to a small auxiliary RCAS index that would otherwise require an expensive restructuring of the main RCAS index.

The buffer tree [Arg03] amortizes the cost of updates by buffering *all* updates at inner nodes and propagating them one level down when the buffers overflow. Instead, we apply the inexpensive Case 1 and 2 insertions immediately on the main RCAS index and redirect Case 3 insertions to the auxiliary RCAS index.

B.9 Conclusion and Outlook

We looked at the problem of supporting insertions in the RCAS index [WBH20], an in-memory, trie-based index for semi-structured data. We showed that not every insertion requires restructuring the index, but for the cases where the index must be restructured we proposed two insertion techniques. The first method, called strict restructuring, preserves RCAS's alternating interleaving of the data's content and structure, while the second method, lazy restructuring, optimizes for insertion speed. In addition, we explore the idea of using an auxiliary index (similar to LSM-trees [LC19]) for those insertion cases that would require restructuring the original index. Redirecting the tough insertion cases to the auxiliary index leaves the structure of the main index intact. We proposed techniques to merge the auxiliary index back into the main index when the auxiliary index grows too big. Our experiments show that these techniques can efficiently insert new keys into RCAS and preserve its good query performance.

For future work we plan to support deletion. Three deletion cases can occur that mirror the three insertion cases. Like for insertion, the first two cases are simple and can be solved by deleting a reference from a leaf or the leaf itself if it contains no more references. The third case occurs when the dynamic interleaving is invalidated because the positions of the discriminative bytes shift. Deletion algorithms exist that mirror our proposed insertion techniques for the third case.

APPENDIX C

Scalable Content-and-Structure Indexing

Reprinted from:

K. Wellenzohn, M. H. Böhlen, S. Helmer, A. Pietri, S. Zacchiroli. "Scalable Content-and-Structure Indexing", [*ready for submission*]

Abstract

Frequent queries on semi-structured data are Content-and-Structure (CAS) queries that filter data items based on their location in the hierarchical structure and their value for some attribute. Existing CAS indexes either do not have robust CAS query performance or do not scale. We propose the scalable and robust CAS (RCAS+) index that tackles both issues. To build RCAS+ at scale we propose *lazy interleaving* to reduce the CPU load, *node clustering* and *proactive partitioning* to curb disk accesses, and *depth-first bulk-loading* and *front-loading* to optimally use the available memory. Our detailed analytical and experimental evaluation shows that the combination of these five techniques yields a scalable CAS index. We show-case RCAS+'s scalability and practical applicability by indexing data from the Software Heritage archive, which is the world's largest, publicly-available source code archive.

C.1 Introduction

The field of *mining software repositories* [Has08] has seen growing interest in the past decades since studying software artifacts at scale is important for software engineering applications. The way researchers study large collections of software artifacts, and source code in particular, is to narrow down the subset of artifacts (i.e., commits, directories, and files) that are relevant to the study and to analyze this subset.

Offering a platform that allows researchers to efficiently perform the selection step in software archives is an important research goal that various recent efforts like Boa, World of Code, and Software Heritage have been trying to tackle [DNRN15, MDB+21, PSZ20]. We look at Content-and-Structure (CAS) queries [MHSB15] that allow researchers to locate *revisions* (i.e., commits) in a large body of source code artifacts based on the commit time of the revisions and the names and paths of the files that they modified (added/changed/deleted). CAS queries consist of a path and a value predicate. The path predicate filters revisions based on the location and name of the modified files in the file system; it allows wildcards to widen the search when the exact locations or names of the files are not known. The value predicate filters revisions based on an attribute, e.g., commit time, author, etc.

Systems like Boa, World of Code, and Software Heritage cannot answer CAS queries. We aim to implement this missing piece in the software mining infrastructure by providing an index that

efficiently answers CAS queries. As a case study, we compute this index for all GitLab repositories maintained by the Software Heritage (SWH) archive, which is the largest publicly-available software archive [ACZ18, DCZ17] with more than two billion revisions from 150 million repositories retrieved from GitHub, GitLab, etc.

To efficiently evaluate CAS queries we use the Robust CAS (RCAS) index [WBH20], which tightly integrates the path and value dimensions by interleaving the paths and values of two-dimensional keys into a single byte-string without favoring one of the dimensions. This is achieved by RCAS's dynamic interleaving scheme that interleaves keys at their discriminative path and value bytes, which are the first bytes for which the keys differ in the respective dimension. RCAS does not scale to large datasets since it is an in-memory index based on a memory-optimized trie (ART [LKN13]). Scaling RCAS to large datasets is challenging. First, extending RCAS to block storage devices is not straightforward since its nodes are not aligned with a page-structured storage layout. Second, the RCAS construction algorithm is limited by main-memory data structures and algorithms that do not scale. For instance, on a machine with 400 GB main memory it is not possible to index datasets larger than 100 GB with RCAS.

We propose the scalable RCAS$^+$ index that is not constrained by the available memory. We develop a scalable algorithm that builds RCAS$^+$ while, at the same time, dynamically interleaving the keys. It is partitioning-based and in each partitioning step we interleave the longest common path and value prefixes of the keys and store them in a new node in RCAS$^+$. Our partitioning is *order-* and *prefix-preserving*, which means that the partitions are totally ordered and the keys in a partition have a longer common prefix than keys from different partitions. These properties make it possible to efficiently evaluate the value and path predicates of CAS queries.

We propose five techniques that make building RCAS$^+$ at scale feasible in terms of, respectively, CPU, memory, and disk[1] usage. **CPU**: *Lazy interleaving* stops to dynamically interleave a set of keys when they fit on a disk page and stores their un-interleaved suffixes in a single leaf node. This speeds up bulk-loading by an order of magnitude without compromising query performance. **Disk**: *Node clustering* aligns nodes with pages of block-based storage devices to compactly store and efficiently search the RCAS$^+$ index. *Proactive partitioning* exploits that the data is partitioned hierarchically and pre-computes the discriminative bytes of new partitions at the next level while partitioning a set of keys. By the time the new partitions are being partitioned we have already computed the discriminative bytes. **Memory**: *Depth-first bulk-loading* guarantees that only a small number of nodes have to be kept in memory before they can be written to disk by

[1] We use the term disk to refer to any generic block-storage device (HDD, SSD, etc.)

node clustering. *Front-loading* optimally uses the remaining memory to buffer partitions. When a set of keys $\mathscr{K}$ is broken up into partitions $\{\mathscr{K}_1, \mathscr{K}_2, \ldots\}$, front-loading keeps those partitions in memory that are processed next during bulk-loading.

Our main technical contributions are as follows:

- We propose the scalable RCAS$^+$ index for large datasets that is not constrained by the available memory. RCAS$^+$ is based on a hierarchical partitioning that is *order*- and *prefix-preserving*, which allows us to evaluate CAS queries efficiently. We propose a new bulk-loading algorithm for large datasets that dynamically interleaves the keys and builds RCAS$^+$ at the same time.

- We propose five techniques to make our bulk-loading algorithm scalable. *Lazy interleaving* stops to dynamically interleave keys when they fit into a leaf page. *Node clustering* aligns nodes with block-based storage devices. *Proactive partitioning* pre-computes the discriminative bytes needed at the next level of the hierarchical partitioning. *Depth-first bulk-loading* curbs the memory footprint of the algorithm so that most of the memory can be used by *front-loading* to optimally buffer partitions that are processed next.

- We evaluate our bulk-loading algorithm analytically and experimentally. In our analytical evaluation we develop a cost model, and prove best and worst case bounds on the algorithm's disk I/O. In our experiments we demonstrate the scalability and performance of our algorithm by indexing all GitLab projects archived by Software Heritage with a total of 120 million unique commits that modify 6.9 billion files.

C.2 Application Scenario

The SWH archive[2] lets users look up repositories by keyword search on their URLs or metadata or look up specific software artifacts by cryptographic identifiers that must be known in advance. More complex CAS queries, such as the following, are not supported:

CAS Query. Find all revisions in the SWH archive that changed a C file in June 2020 (expressed as time range [2020-06-01, 2021-07-01)). The file must be in a folder that begins with name `ext`

[2] `https://archive.softwareheritage.org`

and can be located anywhere in the directory structure of a project (expressed as query path `/**/ext*/*.c`). □

This CAS query consists of a path predicate expressed as a query path (blue) and a value predicate expressed as a range (red). Often the exact location of a file is unknown, but parts of its path are known. We use a shell-like syntax with wildcards to widen the search. For instance, `ext*` matches any label starting in `ext`, e.g., `ext3`, `extension`, etc. The wildcard `**` denotes the descendant-or-self axis and matches any file nested arbitrarily deeply in a folder.

Table C.1: A set $\mathcal{K}^{1..9} = \{k_1, \ldots, k_9\}$ of composite keys

	Path Dimension P	Value Dimension V	R
k_1	/Sources/Scheduler.swift$	00 00 00 00 5D BD 97 8B	r_1
k_2	/crypto/ecc.h$	00 00 00 00 5F BD 94 C4	r_2
k_3	/crypto/ecc.c$	00 00 00 00 5F BD 94 C4	r_2
k_4	/Sources/Signal.swift$	00 00 00 00 5D A8 94 8C	r_3
k_5	/fs/ext3/inode.c$	00 00 00 00 5E F2 9C 59	r_4
k_6	/fs/ext4/inode.c$	00 00 00 00 5E FB 23 C2	r_5
k_7	/fs/ext4/inode.c$	00 00 00 00 5F BD 3D 5A	r_6
k_8	/Sources/Bag.swift$	00 00 00 00 5D A8 94 2A	r_7
k_9	/Sources/Map.swift$	00 00 00 00 5D A8 94 2A	r_7
	1 5 9 13 17 21	1 2 3 4 5 6 7 8	

Answering CAS queries on billions of files is difficult since we have to dynamically interleave and index two-dimensional keys from the SWH archive to answer CAS queries efficiently. Table C.1 shows such composite keys. Key k_7 denotes that file `inode.c` in the `/fs/ext4` directory was modified on 2020-11-24 in revision r_6. The values are stored as 64 bit Unix timestamps and are represented in hexadecimal. Revisions are identified by 20 byte SHA1 hashes in the SWH archive. Our example query returns revisions $\{r_4, r_5\}$ since keys $\{k_5, k_6\}$ match both the path and value predicate.

C.3 Background

C.3.1 Notation & Terminology

A *revision* in the SWH archive captures what is commonly referred to as a "commit" in modern version control systems. A revision references the entire source code tree of a software project at

commit time, points to previous revision(s)—allowing to compute source code “diffs” between commits—and is associated to metadata such as commit time and author.

We index for each revision in the SWH archive its commit time and its diff, i.e., what files are added/changed/deleted. If a revision's diff and more than one attribute (e.g., commit time and author) must be indexed, multiple indexes are created. We store dynamically-interleaved composite keys in our index that consist of a path dimension P (the full path of the modified file) and a value dimension V (the commit time of the revision). Additionally, a key stores a revision as payload. We write $k.P$, $k.V$, and $k.R$ to access k's path, value, and revision, respectively. We write $k.D$ to access k's path (if $D = P$) or value (if $D = V$). We denote a set of keys by $\mathscr{K}$ and write, e.g., $\mathscr{K}^{2,5,6}$ to refer to $\{k_2, k_5, k_6\}$. The set of nine (un-interleaved) keys in Table C.1 is denoted by $\mathscr{K}^{1..9}$.

Paths and values are prefix-free byte strings that we access byte-wise. In the paper we visualize them with one byte ASCII characters for the path dimension and hexadecimal numbers written in italic for the value dimension, see Table C.1. To guarantee that no path is a prefix of another we append the end-of-string character `$` (ASCII code `0x00`) to each path. Fixed-length byte strings (e.g., 64 bit numbers) are prefix-free because of the fixed length. We assume that the path and value dimensions are binary-comparable, i.e., two paths or values are $<$, $=$, or $>$ iff their corresponding byte strings are also $<$, $=$, or $>$, respectively [LKN13]. For example, big-endian integers are binary-comparable while little-endian integers are not.

Let s be a byte-string, then $|s|$ denotes the length of s and $s[i]$ denotes the i-th byte in s. The left-most byte of a byte-string is byte one. $s[i] = \varepsilon$ is the empty string if $i > |s|$. $s[i, j]$ denotes the substring of s from position i to j and $s[i, j] = \varepsilon$ if $i > j$.

Given a set of keys $\mathscr{K}$ we define its *longest common prefix* $\mathsf{lcp}(\mathscr{K}, D)$ and *discriminative byte* $\mathsf{dsc}(\mathscr{K}, D)$ in dimension D.

Definition C.1 (Longest Common Prefix)**.** *The longest common prefix $\mathsf{lcp}(\mathscr{K}, D)$ of keys $\mathscr{K}$ in dimension D is the longest prefix s that all keys $k \in \mathscr{K}$ share in dimension D, i.e.,*

$$\begin{aligned} &\mathsf{lcp}(\mathscr{K}, D) = s \textit{ iff } \forall k \in \mathscr{K}\,(k.D[1, |s|] = s) \wedge \\ &\nexists l (l > |s| \wedge \forall k, k' \in \mathscr{K}\,(l \leq \min(|k.D|, |k'.D|) \wedge k.D[1, l] = k'.D[1, l])) \end{aligned}$$

Definition C.2 (Discriminative Byte)**.** *The discriminative byte $\mathsf{dsc}(\mathscr{K}, D)$ of keys $\mathscr{K}$ in dimension D is the first byte for which the keys differ in dimension D, i.e., $\mathsf{dsc}(\mathscr{K}, D) = |\mathsf{lcp}(\mathscr{K}, D)| + 1$.*

Example C.1. *Consider the composite keys $\mathcal{K}^{1..9}$ in Table C.1. The longest common path and value prefixes are* $\mathsf{lcp}(\mathcal{K}^{1..9}, P) = $ `/` *and* $\mathsf{lcp}(\mathcal{K}^{1..9}, V) = $ `00 00 00 00`*. The discriminative path and value bytes are* $\mathsf{dsc}(\mathcal{K}^{1..9}, P) = 2$ *and* $\mathsf{dsc}(\mathcal{K}^{1..9}, V) = 5$. □

C.3.2 Dynamic Interleaving in the RCAS Index

RCAS is a trie-based index that dynamically interleaves sets of composite keys at their discriminative path and value bytes [WBH20]. Interleaving at the discriminative bytes makes RCAS robust against long common prefixes that are common in alphanumeric keys. To interleave keys we start with the set of all keys $\mathcal{K}$ and partition them based on the value at the discriminative byte in dimension D.

Definition C.3 (ψ-Partitioning *[WBH20]*)**.** *The ψ-partitioning of a set of keys $\mathcal{K}$ in dimension D is $\psi(\mathcal{K}, D) = \{\mathcal{K}_1, \ldots, \mathcal{K}_m\}$ iff*

- (Correctness) *All keys in a set $\mathcal{K}_i$ have the same value at $\mathcal{K}$'s discriminative byte in D. Keys from different sets $\mathcal{K}_i \neq \mathcal{K}_j$ do not have the same value at $\mathcal{K}$'s discriminative byte in D.*
- (Completeness) *Every key in $\mathcal{K}$ is assigned to a set $\mathcal{K}_i$. All $\mathcal{K}_i$ are non-empty.*

Example C.2. *The ψ-partitioning of $\mathcal{K}^{1..9}$ in dimension V is $\psi(\mathcal{K}^{1..9}, V) = \{\mathcal{K}^{1,4,8,9}, \mathcal{K}^{5,6}, \mathcal{K}^{2,3,7}\}$. The keys in these three sets have, respectively, the values* `0x5D`*,* `0x5E`*, and* `0x5F` *at $\mathcal{K}^{1..9}$'s discriminative value byte, which is the fifth byte.* □

RCAS alternatingly ψ-partitions the keys in dimensions V and P and each index node represents one of the partitions. The longest common prefixes of each partition are stored in the corresponding node as path substring s_P and value substring s_V. Thus, a node in RCAS is a tuple (s_P, s_V, D) where s_P and s_V denote the longest common path and value prefix of all composite keys contained in its descendants, and D is the dimension in which this node ψ-partitions the data ($D = \perp$ for leaves that do not partition the data further). Note that if we concatenate substrings s_P and s_V from the root to a leaf we obtain a dynamically-interleaved composite key.

Example C.3. *Consider the index in Figure C.1 for the keys $\mathcal{K}^{1..9}$ in Table C.1 (ignore the dotted lines for the moment). The root node's substrings $s_P = $* `/` *and $s_V = $* `00 00 00 00` *are the longest common path and value prefixes of all keys in $\mathcal{K}^{1..9}$. The root node ψ-partitions the data*

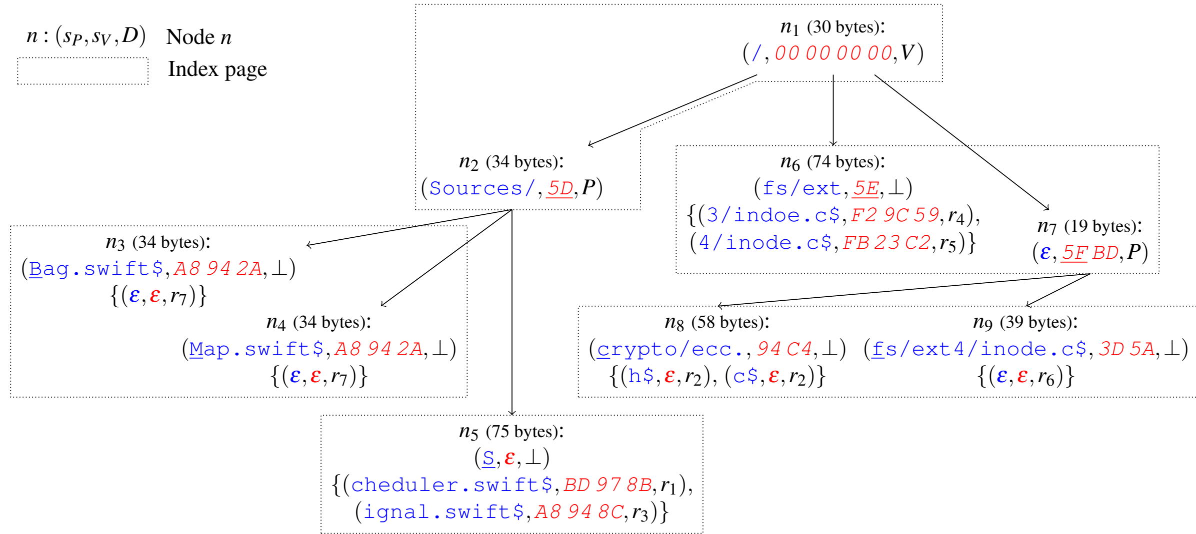


Figure C.1: RCAS+ index for the keys $\mathscr{K}^{1..9}$ from Table C.1. The page size is 100 bytes in this example.

in the value dimension since $n_1.D = V$. If we follow the path from the root to leaf node n_3 we find key k_8 with its reference r_7. Its dynamic interleaving is:
"`/ 00 00 00 00 Sources/ 5D Bag.swift$ A8 94 2A`". □

C.4 The Scalable RCAS+ Index

RCAS+ combines *depth-first bulk-loading* with *node clustering* and *lazy interleaving* to achieve good scalability. Depth-first bulk-loading ensures a small memory footprint. Node clustering aligns nodes to a page-structured disk since nodes are typically small and do not fill a disk page. *Lazy interleaving* combines multiple keys in the same leaf node. Specifically, leaf nodes store a set refs $= \{(s_P, s_V, r), \ldots\}$ of un-interleaved key suffixes s_P and s_V along with their payload (a reference r).

Example C.4. *Leaf n_5 in Figure C.1 contains the suffixes of keys $\{k_1, k_4\}$. The dotted boxes show how nodes are clustered into pages. RCAS+ for keys $\mathscr{K}^{1..9}$ is stored in five pages.* □

C.4.1 Depth-First Bulk-Loading

Building RCAS+ at scale is hard because a set of keys must be considered together to dynamically interleave the keys. The reason is that the discriminative bytes depend on all keys in a set. This makes inserting keys one by one (or in batches) impossible and sort-based bulk-loading impractical since the initial computation of the interleaving is too expensive. We propose a partitioning-based algorithm that simultaneously interleaves the keys and builds RCAS+. The algorithm creates nodes in RCAS+ top-down and clusters them into pages bottom-up. *Depth-first bulk-loading* ensures that the number of nodes kept in memory is small.

Initially, all keys belong to the *root partition*. We use partitions during bulk-loading to temporarily store keys along with meta-information. Once a partition has been processed, it is deleted.

Definition C.4 (Partition)**.** *A partition* $K = (g_P, g_V, \mathsf{mptr}, \mathsf{fptr})$ *stores a set* $\mathscr{K}$ *of composite keys. A key is stored either in memory or on disk.* mptr *and* fptr *are pointers to keys in memory and on disk, respectively.* $g_P = \mathsf{dsc}(\mathscr{K}, P)$ *and* $g_V = \mathsf{dsc}(\mathscr{K}, V)$ *denote the partition's discriminative path and value byte, respectively. We write* $\mathscr{K}$ *to denote the set of keys stored in* K. □

Example C.5. *Root partition* $K^{1..9} = (2, 5, \bullet, \bullet)$ *in Figure C.2a stores keys* $\mathscr{K}^{1..9}$ *from Table C.1.* $K^{1..9}$*'s longest common prefixes are written in bold-face and the first bytes after these prefixes are* $K^{1..9}$*'s discriminative bytes* $g_P = 2$ *and* $g_V = 5$*. We use the placeholder* $\bullet$ *for pointers* mptr *and* fptr*; in Section C.6 we describe which keys are stored, respectively, in memory and on disk.* □

Bulk-loading starts with the root partition K and repeatedly breaks it into smaller partitions using the ψ-partitioning until a partition fits on a page. The ψ-partitioning $\psi(K, D)$ takes a partition K as input and returns a *partition table* where each entry in this table points to a partition K_i. We apply ψ alternatingly in dimensions V and P to interleave the keys at their discriminative bytes. Before we apply $\psi(K, D)$ on a partition K, we add a new node to RCAS+ that extracts K's longest common path and value prefixes. This new node resides temporarily in memory before it is written to disk. RCAS+ nodes are typically small (tens or hundreds of bytes) and do not fill a page (thousands of bytes). This is because inner nodes have at most 2^8 children since we partition at the granularity of bytes, but in practice the fanout is often lower. We use *node clustering* to align nodes to page-structured storage (see Section C.4.3).

We build RCAS+ depth-first in pre-order and cluster the nodes in post-order (i.e., we build the index subtree by subtree rather than level by level) to keep the number of memory-resident nodes

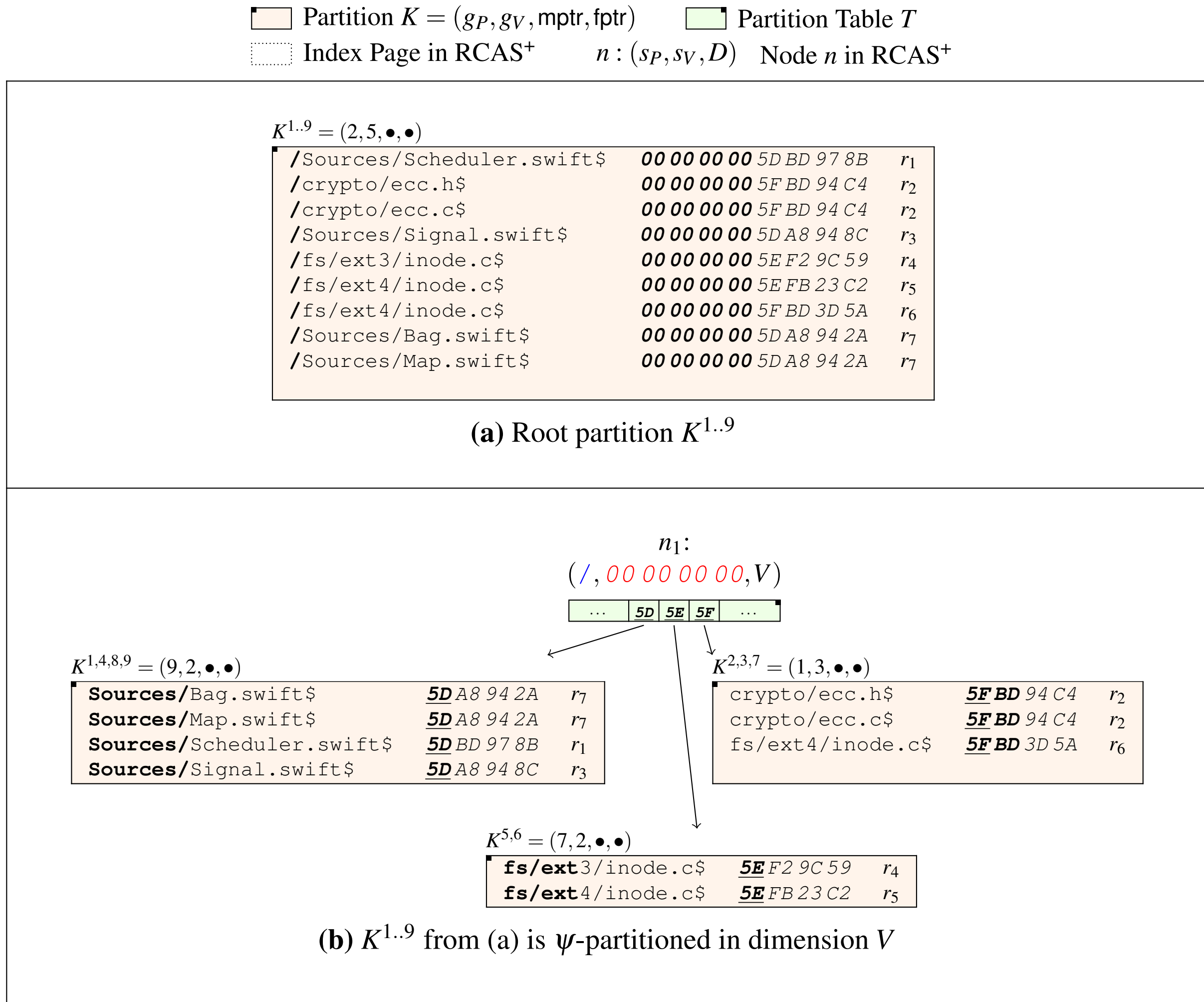


Figure C.2: (Part 1) The keys are recursively ψ-partitioned depth-first, creating new RCAS+ nodes in pre-order. A node represents the longest common path and value prefixes of its corresponding partition. Nodes are clustered into pages in post-order.

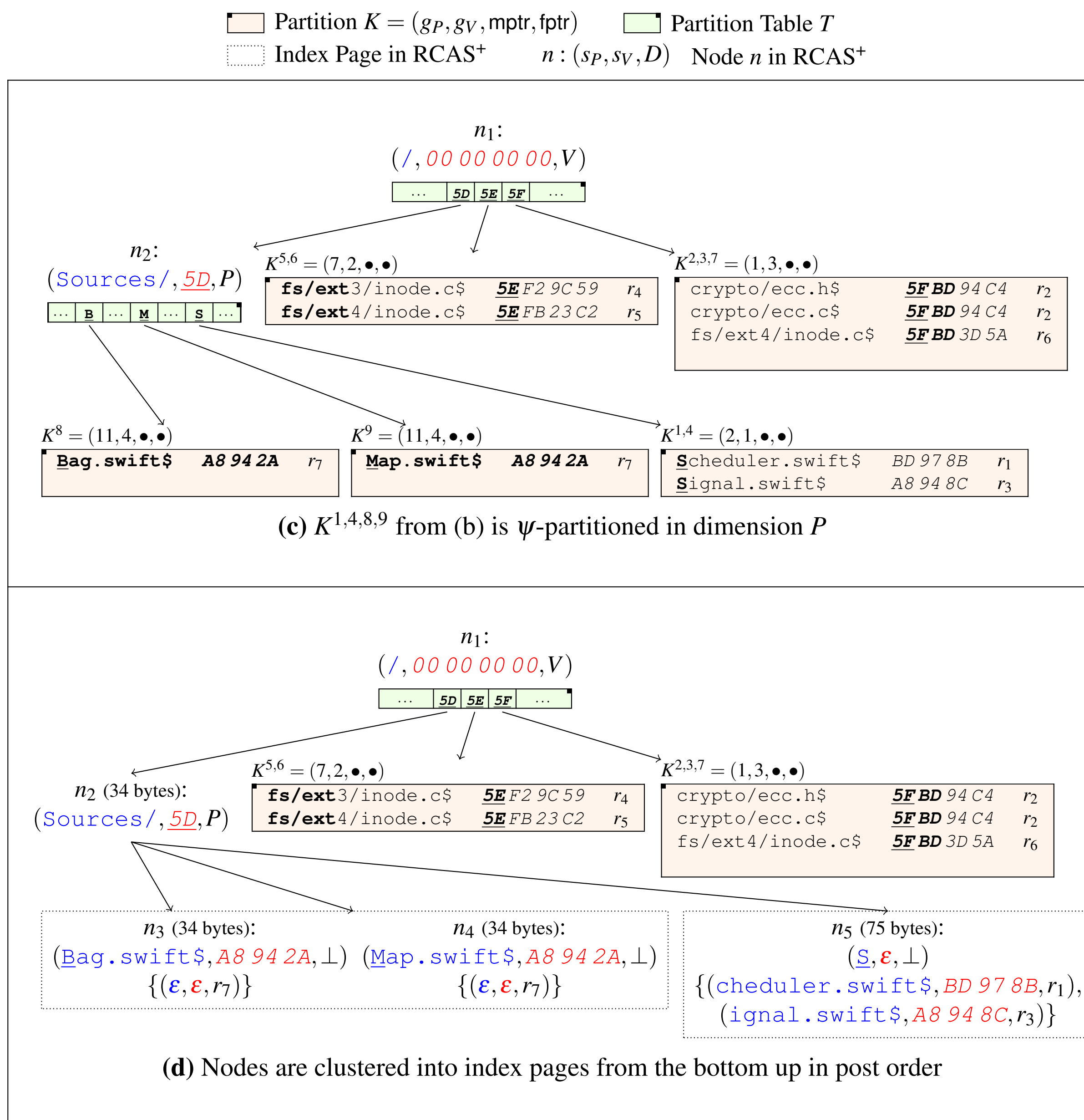


Figure C.2: (Part 2) The keys are recursively ψ-partitioned depth-first, creating new RCAS+ nodes in pre-order. A node represents the longest common path and value prefixes of its corresponding partition. Nodes are clustered into pages in post-order.

small. With depth-first bulk-loading and node clustering we only need to hold $O(h)$ nodes in memory, where h is the height of the tree. Specifically, we need to keep all of the current node's ancestors and a limited number of its siblings in memory that do not yet fill a page (see Lemma C.5). This leaves most of the memory to curb disk I/O by buffering partitions during partitioning steps (see Section C.6).

Algorithm C.1: DepthFirstBulkLoading(K,D)

1 Let n be a new node, k a key in K;
2 $n.s_P \leftarrow k.P[1, K.g_P - 1]$;
3 $n.s_V \leftarrow k.V[1, K.g_V - 1]$;
4 **if** K fits on one page **then** LazyInterleaving(K,n);
5 **else**
6   **if** $D = P \wedge K.g_P > |k.P|$ **then** $D \leftarrow V$;
7   **else if** $D = V \wedge K.g_V > |k.V|$ **then** $D \leftarrow P$;
8     $n.D \leftarrow D$;
9     $T \leftarrow \psi(K,D)$;
10   **for** $b \leftarrow$ `0x00` **to** `0xFF` **do**
11   **if** $T[b] \neq$ NIL **then** n.ptrs$[b] \leftarrow$ DepthFirstBulkLoading$(T[b],\overline{D})$;
12 NodeClustering(n);
13 **return** n;

Algorithm C.1 shows the bulk-loading algorithm. It has two parameters: a partition K (initially the root partition) and the partitioning dimension D (initially dimension V). The algorithm first creates a new node n and sets its longest common prefixes $n.s_P$ and $n.s_V$, which are extracted from a key $k \in K$ from the first byte up to, but excluding, the positions of K's discriminative bytes $K.g_P$ and $K.g_V$ (lines 2–3). If K fits on one disk page, we apply lazy interleaving (Algorithm C.2). Otherwise, we ψ-partition K in dimension D. In lines 6–7 we check if we can indeed ψ-partition K in D and switch to the alternate dimension $\overline{D}$ otherwise ($\overline{D} = P$ if $D = V$ and vice versa). This happens if all keys are equal in dimension D and therefore must be partitioned in $\overline{D}$. In line 9 we apply $\psi(K,D)$ and obtain a partition table T, which is a 2^8-long array that maps the 2^8 possible values b of a discriminative byte (`0x00` $\leq b \leq$ `0xFF`) to partitions. We write $T[b]$ to access the partition for value b ($T[b] =$ NIL if no partition exists for value b). We apply Algorithm C.1 depth-first and recursively call it on each partition in T with the alternate dimension $\overline{D}$. This returns a pointer to each new child node of n that we store in n.ptrs. At this point, in line 12, the current node n is complete: its contents (substrings $n.s_P, n.s_V$ and dimension $n.D$) have been determined and its children, if any, have been computed. Hence, we can now assign node n to an index page (Algorithm C.3).

Example C.6. *Figure C.2 shows how RCAS+ from Figure C.1 is built. In Figure C.2b we create its root node n_1 from root partition $K^{1..9}$ by extracting $K^{1..9}$'s longest common path and value prefix. Then, we ψ-partition $K^{1..9}$ in dimension V and obtain a partition table (light green) that points to three new partitions: $K^{1,4,8,9}$, $K^{5,6}$, and $K^{2,3,7}$. They contain all keys k for which $k.V[K^{1..9}.g_V]$ is `0x5D`, `0x5E`, or `0x5F`, respectively (the values of the discriminative byte are underlined). We drop $K^{1..9}$'s longest common prefixes from these new partitions. We proceed depth-first with partition $K^{1,4,8,9}$ in Figure C.2c. We create node n_2 as before and this time we ψ-partition in dimension P and obtain three new partitions. Assuming a page can hold up to two keys, the three new partitions K^8, K^9, and $K^{1,4}$ each fit on one page and are not partitioned further. In Figure C.2d we create the corresponding leaf nodes n_3, n_4, n_5 for these three partitions. After clustering leaves n_3,n_4,n_5 (see Section C.4.3) we write them to disk, release the memory, and proceed with $K^{5,6}$.* □

C.4.2 Lazy Interleaving

Lazy interleaving stops to dynamically interleave the keys in a partition K if K is small enough to fit on a page. Lazy interleaving stores the un-interleaved suffixes in the set n.refs of a leaf node n, see Algorithm C.2. During subsequent searches all suffixes must be checked if they match the given CAS query. In Section C.8.4 we show that lazy interleaving speeds up bulk-loading by a factor of 20 without compromising query performance.

Algorithm C.2: LazyInterleaving(K,n)

```
1 n.D ← ⊥;
2 foreach key k ∈ K do
3 |   s_P ← k.P[K.g_P, |k.P|];
4 |   s_V ← k.V[K.g_V, |k.V|];
5 |_  n.refs ← n.refs ∪ {(s_P, s_V, k.R)};
6 Delete K;
```

Example C.7. *Assuming two keys fit on a page, partitions K^8, K^9, and $K^{1,4}$ in Figure C.2c are lazily interleaved. We create the three leaf nodes n_3, n_4, and n_5 in Figure C.2d. While n_3 and n_4 each represent a single key, node n_5 stores the suffixes of keys $\{k_8,k_9\}$.* □

C.4.3 Node Clustering

We cluster nodes into pages with the greedy right-sibling algorithm [KM06a] that, given a node n, starts clustering at n's rightmost child and grows a cluster leftwards until a page is full. Once full, a page is written to disk and the memory used by the nodes is released. The leftmost siblings are placed in the same page as n if they do not fill a page on their own. These nodes are clustered with n's siblings when node clustering is invoked on n's parent. The pseudo code is shown in Algorithm C.3.

Algorithm C.3: NodeClustering(n)

```
1  Compute n's size and store it in n.size;
2  cluster, size ← ({}, 0);
3  for b ← 0xFF down to 0x00 do
4      if n.ptrs[b] ≠ NIL then
5          if size + n.ptrs[b].size > PAGE_SIZE then
6              page_nr ← WriteIndexPage(cluster);
7              for (b′, _) ∈ cluster do n.ptrs[b′] ← page_nr;
8              cluster, size ← ({}, 0);
9          cluster ← cluster ∪ {(b, n.ptrs[b])};
10         size ← size + n.ptrs[b].size;
11 if size + n.size > PAGE_SIZE then
12     page_nr ← WriteIndexPage(cluster);
13     for (b′, _) ∈ cluster do n.ptrs[b′] ← page_nr;
14 else n.size ← n.size + size;
```

Example C.8. *Assume the page size is 100 bytes and the size of a pointer is 8 bytes. The size of node n_3 in Figure C.2d is 34 bytes; it consists of 10B for s_P, 3B for s_V, 1B for D, and 20B for reference r_7. We execute Algorithm C.3 on node n_2. Its rightmost child n_5 is written to its own page since nodes $\{n_4, n_5\}$ cannot be put in the same page ($34 + 75 > 100$). Nodes n_3 and n_4 are clustered together, but n_2 cannot be added since $34 \cdot 3 > 100$.* □

C.5 Proactive Partitioning

Proactive partitioning eliminates explicit scans to compute discriminative bytes during the ψ-partitioning. Remember that we have to know K's discriminative byte in dimension D to compute $\psi(K, D)$. Thus, in general we need two scans over K to compute $\psi(K, D)$: the first scan

determines K's discriminative byte and the second scan assigns the keys to their partitions. It is not possible to combine these two scans since first we need to look at *every* key in K to compute its discriminative byte $\mathsf{dsc}(\mathscr{K}, D)$ and only then we can ψ-partition K according to this byte. To see why, consider a key k in K and assume $\mathsf{dsc}(\mathscr{K} \setminus \{k\}, D) = g$. If k differs from all other keys at byte $g-1$ in D, the discriminative byte of $\mathscr{K}$ is $g-1$. Thus, without looking at every key we cannot compute $\mathsf{dsc}(\mathscr{K}, D)$ and without that we cannot ψ-partition K.

To avoid scanning K twice we make the ψ-partitioning *proactive* by exploiting that $\psi(K,D)$ is applied hierarchically. This means we pre-compute the discriminative bytes of every new partition $K_i \in \psi(K,D)$ as we ψ-partition K. As a result, by the time K_i itself is ψ-partitioned, we already know its discriminative bytes and can directly compute the partitioning. We only need to explicitly compute the root partition's discriminative bytes. The discriminative bytes of subsequent partitions are computed proactively during the partitioning. This halves the scans over the data during bulk-loading.

C.5.1 Implementation

Algorithm C.4 implements $\psi(K,D)$ and returns a partition table T. We organize the keys in a partition K at the granularity of pages so that we can seamlessly transition between keys in memory (K.mptr) and on disk (K.fptr). A page is a fixed-length buffer that contains a variable number of keys. K.mptr points to the head of a singly-linked list of pages, while K.fptr points to a page-structured file on disk (see Figure C.3). Algorithm C.4 iterates first over the pages in K.mptr and then those in K.fptr. For each key in a page line 7 determines the partition $T[b]$ to which k belongs by looking at its value b at the discriminative byte. Next we drop the longest common path and value prefixes from k (lines 8–9). We proactively compute $T[b]$'s discriminative bytes whenever we add a key k to $T[b]$ (lines 10–17). Two cases can arise. If k is $T[b]$'s first key, we initialize g_P and g_V with one past the length of k in the respective dimension (lines 10–12). These values are valid upper-bounds for the discriminative bytes since keys are prefix-free. We store k as a reference key for partition $T[b]$ in refkeys$[b]$. If k is not the first key in $T[b]$, we update the upper bounds (lines 13–17) as follows. Starting from the first byte, we compare k with reference key refkeys$[b]$ byte-by-byte in both dimension until we reach the upper-bounds $T[b].g_P$ and $T[b].g_V$, or we find new discriminative bytes and update $T[b].g_P$ and $T[b].g_V$. As we iterate over K we incrementally release its memory-resident pages and re-use them in T (see Section C.6).

Algorithm C.4: $\psi(K,D)$

1 Let T be a new partition table;
2 Let outpages be an array of 2^8 pages for output buffering;
3 Let refkeys be an array to store 2^8 composite keys;
4 **while** K contains more pages **do**
5   page $\leftarrow$ **if** K.mptr $\neq$ NIL **then** pop(K.mptr) **else** read(K.fptr);
6   **foreach** key $k \in$ page **do**
7     $b \leftarrow$ **if** $D = P$ **then** $k.P[K.g_P]$ **else** $k.V[K.g_V]$;
8     $k.P \leftarrow k.P[K.g_P, |k.P|]$;
9     $k.V \leftarrow k.V[K.g_V, |k.V|]$;
10   **if** $T[b] =$ NIL **then**
11       $T[b] \leftarrow (|k.P|+1, |k.V|+1, \text{NIL}, \text{NIL})$;
12   refkeys$[b] \leftarrow k$;
13   **else**
14       $k', g_P, g_V \leftarrow$ (refkeys$[b]$, 1, 1);
15   **while** $g_P < T[b].g_P \wedge k.P[g_P] = k'.P[g_P]$ **do** g_P++;
16   **while** $g_V < T[b].g_V \wedge k.V[g_V] = k'.V[g_V]$ **do** g_V++;
17       $T[b].g_P, T[b].g_V \leftarrow (g_P, g_V)$;
18   **if** outpages$[b]$ is full **then**
19   outpages$[b] \leftarrow$ FrontLoadingInsertion(T, b, outpages$[b]$);
20   Clear contents of page outpages$[b]$;
21   Add k to outpages$[b]$;
22   Delete page;
23 **for** $b \leftarrow$ `0x00` **to** `0xFF` **do**
24   **if** $T[b] \neq$ NIL **then** FrontLoadingInsertion(T, b, outpages$[b]$);
25 Delete K;
26 **return** T;

(Lines 10–17 highlighted: proactivley compute discriminative bytes)

C.5.2 Properties of the Partitioning

RCAS+ supports CAS queries with a value and a path predicate. We use range and prefix searches to efficiently evaluate the predicates. With a range search we choose subtrees that intersect the range predicate and with a prefix search we choose subtrees that match the path predicate. The partitioning is *order-* and *prefix-preserving* to efficiently support range and prefix searches.

Property C.1 (*Order-Preserving*)**.** *The ψ-partitioning $T = \psi(K,D)$ is order-preserving in dimension D. That is, all keys in a partition are either strictly greater or smaller in dimension D*

than all keys from another partition:

$$\forall K_i, K_j \in T, K_i \neq K_j : (\forall k \in \mathscr{K}_i, \forall k' \in \mathscr{K}_j : k.D < k'.D) \vee$$
$$(\forall k \in \mathscr{K}_i, \forall k' \in \mathscr{K}_j : k.D > k'.D)$$

Example C.9. *The ψ-partitioning $\psi(K^{1..9}, V) = \{K^{1,4,8,9}, K^{5,6}, K^{2,3,7}\}$ is order-preserving in dimension V. The value predicate $[2018\text{-}07, 2018\text{-}09)$ only needs to consider partition $K^{1,4,8,9}$, which spans keys from June to December, 2018 (range* `[0x5D 00 00 00, 0x5E 00 00 00)`*).* □

Property C.2 (*Prefix-Preserving*)**.** *The ψ-partitioning $T = \psi(K, D)$ is prefix-preserving in dimension D. That is, keys in the same partition have a longer common prefix in dimension D than keys from different partitions:*

$$\forall K_i, K_j \in T, K_i \neq K_j : |\mathit{lcp}(\mathscr{K}_i, D)| > |\mathit{lcp}(\mathscr{K}_i \cup \mathscr{K}_j, D)| \wedge$$
$$|\mathit{lcp}(\mathscr{K}, D)| = |\mathit{lcp}(\mathscr{K}_i \cup \mathscr{K}_j, D)|$$

Example C.10. *The ψ-partitioning $\psi(K^{1..9}, P) = \{K^{1,4,8,9}, K^{2,3}, K^{5,6,7}\}$ is prefix-preserving in dimension P. For example, $K^{1,4,8,9}$ has a longer common path prefix $\mathit{lcp}(\mathscr{K}^{1,4,8,9}, P) =$* `/Source/` *than keys across partitions, e.g., $\mathit{lcp}(\mathscr{K}^{1,4,8,9} \cup \mathscr{K}^{2,3}, P) =$* `/`*. Because of this, query path* `/Source/S*.swift` *only needs to consider partition $K^{1,4,8,9}$.* □

Properties C.1 and C.2 guarantee a total ordering among partitions. The nodes in RCAS+ are ordered by the value at the discriminative byte such that queries can quickly choose the correct subtree (see [WBH20] for details about the query algorithm).

The third property ensures that ψ actually partitions the data (unlike, e.g., radix partitioning).

Property C.3 (*Guaranteed Progress*)**.** *Let K be a partition for which not all keys are equal in dimension D. $\psi(K, D)$ guarantees progress, i.e., ψ splits K into at least two partitions: $|\psi(K, D)| \geq 2$.*

The ψ-partitioning ensures these three properties because we partition K at its discriminative byte. If we partitioned the data *before* this byte we would not make progress, because every byte before the discriminative byte is part of the longest common prefix. If we partitioned the data *after* the discriminative byte the partitioning would no longer be order- and prefix-preserving.

Example C.11. *$K^{1..9}$'s discriminative value byte is the fifth byte. If we partitioned $K^{1..9}$ at value byte four we get $\{K^{1..9}\}$ and there is no progress since all keys have* `0x00` *at value byte four. If*

we partitioned $K^{1..9}$ at value byte six we get $\{K^{4,8,9}, K^{1,2,3,7}, K^5, K^6\}$, which is neither order- nor prefix-preserving in V. Consider keys $k_1, k_2 \in K^{1,2,3,7}$ and $k_6 \in K^6$. The partitioning is not order-preserving in V since $k_1.V < k_6.V < k_2.V$. The partitioning is not prefix-preserving in V since the longest common value prefix in $K^{1,2,3,7}$ is `00 00 00 00`, which is not longer than the longest common value prefix of keys from different partitions since, e.g., $\mathsf{lcp}(\mathscr{K}^{1,2,3,7} \cup \mathscr{K}^5, V) =$ `00 00 00 00`. □

C.6 Front-Loading

Depth-first bulk-loading together with node clustering minimizes the memory footprint and *front-loading* uses the remaining memory to optimally buffer partitions. Front-loading keeps the partitions at the beginning of a partition table in memory to minimize the overall disk I/O during bulk-loading. It exploits that Algorithm C.1 (lines 10–11) processes the partitions in a partition table depth-first in ascending order. Thus, we allocate memory for the partitions at the beginning of the partition table. A front-loading partition table contains a sequence of *memory-resident* partitions, followed by zero or one *hybrid* partition (i.e., partially in memory and on disk), followed by a sequence of *disk-resident* partitions.

Definition C.5 (Front-Loading)**.** *A partition table T is front-loading iff there exists an index i, `0x00` $\leq i \leq$ `0xFF`, such that*

- $\forall b < i, T[b] \neq \textsc{Nil} : T[b].\mathit{mptr} \neq \textsc{Nil} \wedge T[b].\mathit{fptr} = \textsc{Nil}$
- $\forall b > i, T[b] \neq \textsc{Nil} : T[b].\mathit{mptr} = \textsc{Nil} \wedge T[b].\mathit{fptr} \neq \textsc{Nil}$
- $T[i].\mathit{mptr} \neq \textsc{Nil} \vee T[i].\mathit{fptr} \neq \textsc{Nil}$

Example C.12. *Partition table $T^{1..9}$ in Figure C.3 is front-loading since it has one memory-resident partition $T^{1..9}[$`0x5D`$]$ followed by two disk-resident partitions $T^{1..9}[$`0x5E`$]$ and $T^{1..9}[$`0x5F`$]$ (there is no hybrid partition).* □

Assume we compute $\psi(K, D)$ for a partition K and obtain the front-loading partition table $T = \{K_1, K_2, \ldots\}$, where K_1 is the first partition. Once the bulk-loader has processed K_1, it can re-use the memory occupied by K_1 since K_1 is not used again.

Example C.13. *Figure C.4 shows how front-loading is applied in a hierarchical partitioning. We assume that two pages fit into memory. The root partition contains eight pages, two in memory*

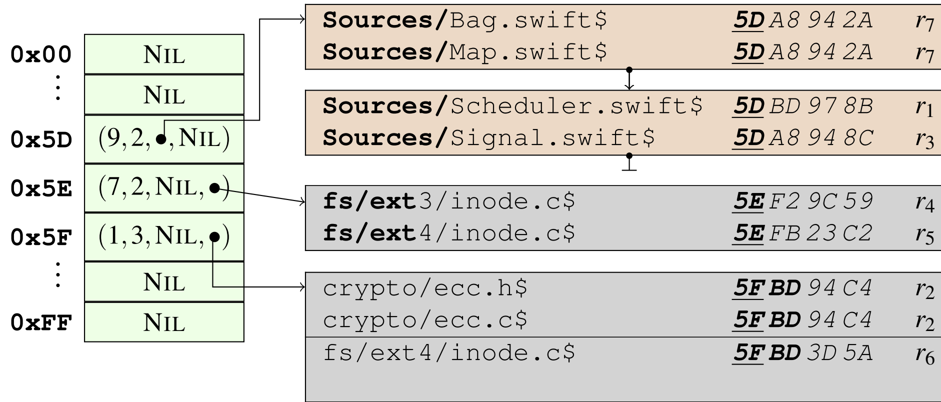


Figure C.3: Front-loading partition table $T^{1..9} = \psi(K^{1..9}, V)$. Orange (gray) pages are stored in memory (on disk).

and six on disk. We identify partitions by their pre-order number, which is the order in which the partitions are recursively processed. For the illustration, we assume that ψ always creates two partitions. When we ψ-partition **1***, we obtain partitions* **2** *and* **9***. The two memory pages from* **1** *are used in* **2***, while* **9** *is disk-resident. Once the bulk-loader recursively processed* **2***, it has two free pages available in memory. They are used when its sibling* **9** *is partitioned and, as a result,* **10** *is memory-resident. Had the bulk-loader initially used the two memory pages for* **9** *instead of* **2***, all of* **2***'s descendants would be disk-resident, which would incur a higher disk I/O.* □

C.6.1 Implementation

Algorithm C.5 inserts a page into a front-loading partition table T. It takes three parameters: table T, the position b in the table where the page is to be inserted, and a pointer ptr to the page in the output buffer of the caller (Algorithm C.4). Algorithm C.5 inserts the page into $T[b]$ and returns a pointer to a page that replaces the old page in the caller's output buffer (this can be the same page or a new page). The algorithm decides whether to keep the page in memory and add it to $T[b]$.mptr or write the page to disk at $T[b]$.fptr. We first check if there exists a free page in memory that can replace ptr in the caller's output buffer. In this case, we add the page to $T[b]$.mptr and return a pointer to this free page in memory (lines 1–3). If no such free page exists, we try to reclaim a memory-resident page from a partition $T[b']$, $b' > b$. Starting from the last partition in T, we walk towards partition $T[b]$ and look for a partition $T[b']$ that contains a memory-resident page (i.e., $T[b']$.mptr $\neq$ NIL). If we find such a partition $T[b']$, we add ptr to $T[b]$.mptr and we move one memory-resident page from $T[b']$.mptr to disk at $T[b']$.fptr (lines 7–10). This frees a

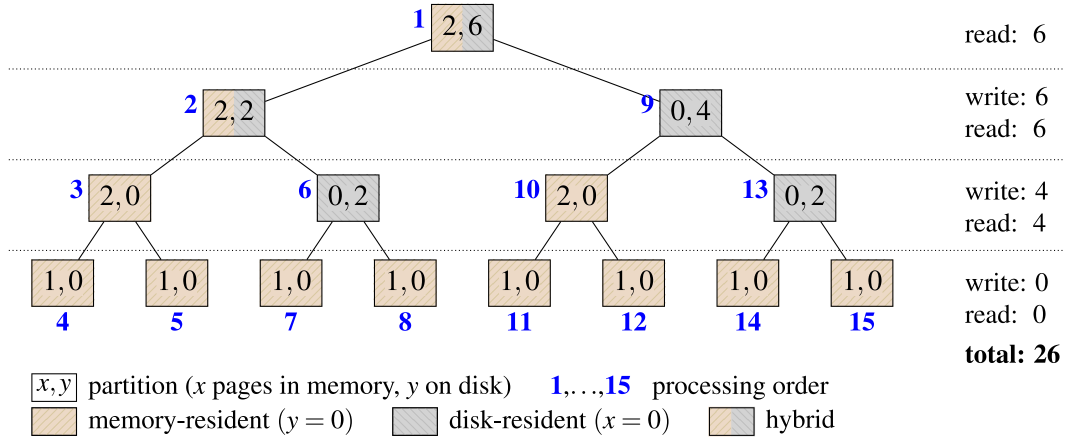


Figure C.4: Front-loading during bulk-loading

page in memory that is returned to the caller. Otherwise, we write the page to disk at $T[b]$.fptr and ptr itself is returned to the caller (line 11).

Algorithm C.5: FrontLoadingInsertion(T, b, ptr)

```
1  if a free page in memory exists then
2  |  push(T[b].mptr, ptr);
3  |  return pointer to a free page in memory;
4  else
5  |  b' ← 0xFF;
6  |  while b' > b ∧ (T[b'] = NIL ∨ T[b'].mptr = NIL) do −−b';
7  |  if b' > b then
8  |  |  push(T[b].mptr, ptr);
9  |  |  ptr ← pop(T[b'].mptr);
10 |  |  write(T[b'].fptr, ptr);
11 |  else write(T[b].fptr, ptr);
12 |  return ptr;
```

C.6.2 Analysis

Theorem C.1. *Algorithm C.5 inserts a page into a front-loading partition table in $O(1)$ time.*

Proof. If there exists a free page in memory we add the page at the front of the linked list $T[b]$.mptr in $O(1)$ time. Otherwise, we try to reclaim a memory-resident page from a partition $T[b']$, $b' > b$, in $O(1)$ time since there are at most 2^8 partitions. □

Theorem C.2. *Front-loading minimizes the disk I/O of Algorithm C.1 if memory-resident partitions are processed in memory.*

Proof. We consider one invocation of $T = \psi(K,D)$ and look at T's memory-resident, hybrid, and disk-resident partitions. Since memory-resident partitions are processed in memory they incur no disk I/O other than writing the final index to disk. Let the hybrid partition (if one exists) be K_j. To decrease K_j's and its descendants' disk I/O, we need to steal memory from a memory-resident partition K_i that appears before K_j in T. This turns K_i into a hybrid or disk-resident partition and thus its disk I/O (and that of its descendants) *increases*. The disk I/O incurred by K_j's descendants remains *unchanged* since by the time they are processed, Algorithm C.1 has already reclaimed the memory used by K_i and its descendants. The same is true for disk-resident partitions. Thus, stealing memory from a partition that appears before cannot decrease disk I/O. □

C.7 Analytical Evaluation

C.7.1 I/O Overhead

We analyze the I/O overhead of Algorithm C.1 for a uniform data distribution where RCAS⁺ is balanced and for a maximally skewed distribution where RCAS⁺ is unbalanced. The I/O overhead is the number of page I/Os without reading the input and writing the output (the index). We use N, M, and B for the number of input keys, the number of keys that fit into memory, and the number of keys that fit into a page, respectively [AV88].

If the data is uniformly distributed, the ψ-partitioning splits a partition into equally sized partitions. With a fixed fanout f the ψ-partitioning splits a partition into f, $2 \leq f \leq 2^8$, partitions.

Theorem C.3. *The I/O overhead to build RCAS⁺ with Algorithm C.1 from uniformly distributed data is*

$$\left\lceil\frac{N}{B}\right\rceil - \left\lceil\frac{M}{B}\right\rceil + 2 \times \sum_{i=1}^{\lceil \log_f \lceil \frac{N}{M} \rceil \rceil} \left(\left\lceil\frac{N}{B}\right\rceil - f^{i-1}\left\lceil\frac{M}{B}\right\rceil\right)$$

Proof. The term $\lceil \frac{N}{B} \rceil - \lceil \frac{M}{B} \rceil$ is the cost to read the root partition during the first ψ-partitioning, with $\lceil \frac{M}{B} \rceil$ pages being buffered during the first scan when computing the root partition's discriminative bytes. There are $\lceil \log_f \lceil \frac{N}{M} \rceil \rceil$ levels before partitions fit completely into memory. At level

i we have f^i partitions due to f^{i-1} ψ-partitionings on the upper level. Front-loading guarantees that each run of $\psi(K,D)$ can keep $\lceil \frac{M}{B} \rceil$ pages in memory (see Figure C.4). Therefore, we read and write $\lceil \frac{N}{B} \rceil - f^{i-1} \lceil \frac{M}{B} \rceil$ pages on level i. □

Example C.14. *We compute the disk I/O for the tree in Figure C.4 with* $N = 16$, $M = 4$, $B = 2$, *and* $f = 2$. *At level* $i = 0$ *we read* $\frac{16}{2} - \frac{4}{2} = 6$ *pages to* ψ*-partition the root partition. There are* $\lceil \log_2 \lceil \frac{16}{4} \rceil \rceil = 2$ *intermediate levels where we recursively partition the data. The partitions at level* $i = 1$ *are created through* $f^{i-1} = 1$ ψ*-partitionings in the upper level, thus the disk I/O on level 1 is* $2(\frac{16}{2} - 1 \times \frac{4}{2}) = 12$. *The partitions at level* $i = 2$ *are created through* $f^{i-1} = 2$ ψ*-partitionings in the upper level, thus the disk I/O on level 2 is* $2(\frac{16}{2} - 2 \times \frac{4}{2}) = 8$. *In total, the disk I/O is* $6 + 12 + 8 = 26$. □

For maximally skewed data RCAS⁺ deteriorates to a tree whose height is linear in the number of keys in the dataset.

Theorem C.4. *The I/O overhead to build RCAS⁺ with Algorithm C.1 from maximally skewed data is*

$$\left\lceil \frac{N}{B} \right\rceil - \left\lceil \frac{M}{B} \right\rceil + 2 \times \sum_{i=1}^{N-M+B} \left(\left\lceil \frac{N-i}{B} \right\rceil - \left\lceil \frac{M}{B} \right\rceil + 1 \right)$$

Proof. The term $\lceil \frac{N}{B} \rceil - \lceil \frac{M}{B} \rceil$ is the same as before. We assume that $\psi(K,D)$ returns two partitions where the first contains one key and the second contains all other keys. Thus, on each level of the partitioning we have two partitions $K_{i,1}$ and $K_{i,2}$ such that $|\mathscr{K}_{i,1}| = 1$ and $|\mathscr{K}_{i,2}| = N - i$. Partition $K_{i,1}$ occupies one page in memory and no page on disk. $K_{i,2}$ occupies $\lceil \frac{M}{B} \rceil - 1$ pages in memory and $\lceil \frac{N-i}{B} \rceil - \lceil \frac{M}{B} \rceil + 1$ pages on disk. Setting the latter to zero and solving for i shows that there are are $i = N - M + B$ levels before partition $K_{i,2}$ fits completely into memory. □

Example C.15. *We use the same parameters as in the previous example but assume the data is maximally skewed. The cost* $\frac{16}{2} - \frac{4}{2} = 6$ *to* ψ*-partition the root partition stays the same. There are* $16 - 4 + 2 = 14$ *levels before the partitions fit into memory. For example, at level* $i = 1$ *we write and read* $\lceil \frac{16-1}{2} \rceil - \lceil \frac{4}{2} \rceil + 1 = 7$ *pages. In total, the I/O overhead is* 104 *pages.* □

Theorem C.1. *The I/O overhead to build RCAS⁺ with Algorithm C.1 depends on the data distribution and is lower-bounded by* $O(\frac{N-M}{B} \log(\frac{N}{M}))$ *and upper-bounded by* $O(\frac{N-M}{B}(N - M + B))$.

Proof. The lower-bound follows from Lemma C.3. In each level $O(\frac{N-M}{B})$ pages are transferred and there are $O(\log(\frac{N}{M}))$ levels in the partitioning. The base of the logarithm is upper-bounded by

$f = 2^8$ since we ψ-partition at the granularity of bytes. The upper-bound follows from Lemma C.4. In each level $O(\frac{N-M}{B})$ pages are transferred and there are $O(N-M+B)$ levels in the worst case. □

Note that, since RCAS⁺ is trie-based and keys are encoded by the path from the root to the leaves, the height of the index is bounded by the length of the keys. The worst-case is unlikely in practice because it requires that the lengths of the keys is linear in the number of keys. Typically, the key length is logarithmic in the number of keys and at most tens or hundreds of bytes. We show in Section C.8 that building RCAS⁺ performs close to the best case on real world data.

C.7.2 Space Overhead

Algorithm C.1 needs memory for nodes before the nodes have been clustered into pages and written to disk, and for other temporary data structures. This is the space overhead of Algorithm C.1.

Theorem C.5. *Algorithm C.1's space overhead is $O(h \cdot b)$, where h is the height of RCAS⁺ and b is the page size.*

Proof. Since Algorithm C.1 processes partitions depth-first, it requires $O(h \cdot b)$ memory for nodes, partition tables, and I/O buffers. Since nodes are clustered in post-order, we need to keep the $O(h)$ ancestors of the current node n in memory and the size of a node is at most $O(b)$. In addition, we need to keep some of n's siblings and their descendants in memory before they are clustered. The number of siblings is upper-bounded by 2^8 and each sibling with its descendants requires at most $O(b)$ memory because if they required more than that, the greedy node-clustering algorithm would have written them to disk already. There exist at most $O(h)$ partition tables at the same time, each requiring $O(1)$ memory. At most one ψ-partitioning is executed at the same time and it uses one input page and 2^8 output pages, each of size $O(b)$. □

C.8 Experimental Evaluation

Setup. We use a Debian 10 server with 80 cores and 400 GB main memory. The machine has six hard disks, each 2 TB big, that are configured in a RAID 10 setup. We use a page size of 16 KB. The code is written in C++ and compiled with g++ 8.3.0.

Dataset. We use the GitLab data from the SWH archive, which contains all archived copies of publicly available GitLab repositories up to 2020-12-15. The GitLab dataset contains 914 593 archived repositories, which correspond to a total of 120 071 946 unique revisions and 457 839 953 unique files. For all revisions in the GitLab dataset we index the commit time and the modified files (equivalent to "commit diffstats" in version control system terminology). In total, we index 6 891 972 832 composite keys similar to Table C.1. The average length of a composite key is 80 bytes. The GitLab dataset (without the contents of the source code files) is 1.6 TB big and the 6.9 billion keys consume 550 GB. In the following we refer to the keys that are indexed as the GitLab dataset.

Reproducibility. The code, the dataset, and instructions how to reproduce our experiments is available online.[3]

C.8.1 Scalability of Depth-First Bulk-Loading

We compare RCAS⁺ to RCAS [WBH20], Postgres (version 13.2), and GNU sort (version 8.3). To compare with Postgres we create a table `data`(P,V,R), similar to Table C.1, and create a composite B+ tree on attributes path and value. Postgres creates a B+ tree by sorting the data and then building the index bottom up, level by level. We compare our bulk-loading algorithm with GNU sort since sorting is a pre-requisite of many bulk-loading algorithms. All approaches are run single-threaded and we configure RCAS⁺, Postgres, and GNU sort to use 300 GB of memory. The memory size of RCAS cannot be configured; it uses the available memory (400 GB) and crashes once it runs out of memory and the swap space provided by operating system (OS).

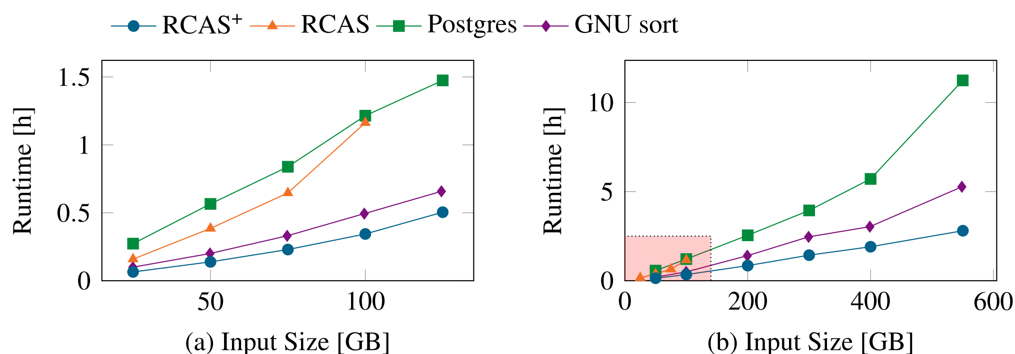


Figure C.5: Scaling RCAS⁺

[3] https://github.com/k13n/scalable_rcas

In Figure C.5a we increase the input size until RCAS crashes and in Figure C.5b we continue until we reach the full GitLab dataset. Note that building RCAS$^+$ is faster than RCAS even though RCAS$^+$ is written to disk. One reason is that RCAS$^+$ uses lazy interleaving, which stops the hierarchical partitioning early, while RCAS breaks partitions down until a partition contains a single key. Additionally, RCAS$^+$ proactively computes discriminative bytes, which reduces the number of scans over the data. Clearly, RCAS does not scale. It crashes for inputs larger than 100 GB since it runs out of memory. The sharp increase in RCAS's runtime before it crashes is due to swapping when the OS runs out of memory.

RCAS$^+$ scales near-linearly and does not deteriorate when the input size exceeds the available memory (300 GB). We observed that for Postgres and GNU sort the OS started swapping pages from memory to disk for inputs $>$ 400 GB, despite limiting the available memory to 300 GB. This explains Postgres' and GNU sort's sharp increase for inputs $>$ 400 GB.

On the full GitLab dataset, the RCAS$^+$ index is 464 GB big and the B+ tree with fill-factor 100% is 451 GB big. RCAS$^+$ is slightly bigger than the B+ tree because it stores the 20 byte long revisions in the index, while B+ tree stores pointers of size 6 bytes to tuples in table `data` that contain the revisions.

C.8.2 Query Performance

We show that RCAS$^+$ maintains RCAS's excellent query performance by comparing RCAS$^+$ to RCAS and Postgres' B+ tree. For an in-depth evaluation of the query performance of RCAS, and by extension RCAS$^+$, we refer to [WBH20]. For Postgres we create two composite B+ trees: one on attributes (P,V), and one on attributes (V,P). We limit ourselves to CAS queries that can be expressed in SQL. We use the `%` wildcard in the query path to match all files in a certain subdirectory, e.g., the query path `/a/b/%` matches all files located arbitrarily deeply in directory `/a/b`.

```
SELECT COUNT(*) FROM data
WHERE P LIKE '@path' AND V BETWEEN @start AND @end;
```

Table C.2 shows the runtime of 100 CAS queries that on average match about 180k keys (0.002%) in the GitLab dataset. We evaluate these queries on the GitLab dataset and a 100 GB subset since RCAS cannot be built for the full dataset. For RCAS$^+$ and the B+ trees we look at the runtime with cold caches, where data must be read from disk, and warm caches. Since RCAS

is memory-based, we report the runtime for warm caches only. RCAS$^+$ outperforms the B+ trees since it simultaneously evaluates the path and value predicate of a CAS query, while a composite index evaluates one after the other. This allows RCAS$^+$ to prune subtrees early and gives it its robust query performance. RCAS$^+$ is slightly slower than RCAS when the data fits into memory since RCAS is based on a memory-optimized trie.

Table C.2: CAS Query Performance

	100 GB GitLab subset		Full GitLab dataset	
	cold cache	warm cache	cold cache	warm cache
RCAS$^+$	0.82 s	0.03 s	1.54 s	0.05 s
RCAS	N/A	0.01 s	N/A	N/A
B+ Tree (P, V)	27.41 s	0.50 s	34.85 s	1.32 s
B+ Tree (V, P)	8.39 s	0.26 s	9.52 s	1.05 s

C.8.3 Node Clustering

We use node clustering to align clustered nodes to pages. Figures C.6a and C.6b show the internal and external depth of leaf nodes, respectively. The internal (external) depth of a leaf is the number of nodes (pages) on the path from the root to the leaf. Clearly, node clustering is effective: it reduces the average depth from 7 (internal) to 5 (external) and the maximum depth from 64 (internal) to 14 (external). Figures C.6c and C.6d show the fanout of nodes and pages, respectively. The fanout of a page is the number of outgoing pointers, which is five for the root page in Figure C.1. Figure C.6c shows that on average a partitioning yields nodes with a fanout of 10 and node clustering increases the fanout to 15 per page. Node clustering groups on average eight nodes into one page, see Figure C.6e. Figure C.6f shows the page utilization of RCAS$^+$, i.e., what percentage of a page is occupied. The average page utilization is 76% and half of all pages have a page utilization of 85% or more.

C.8.4 Lazy Interleaving

Lazy interleaving stops to dynamically interleave keys in a partition when the partition fits on a page. This speeds up bulk-loading by a factor of 20 (see Figure C.7a) because about 40 times fewer partitions are created.

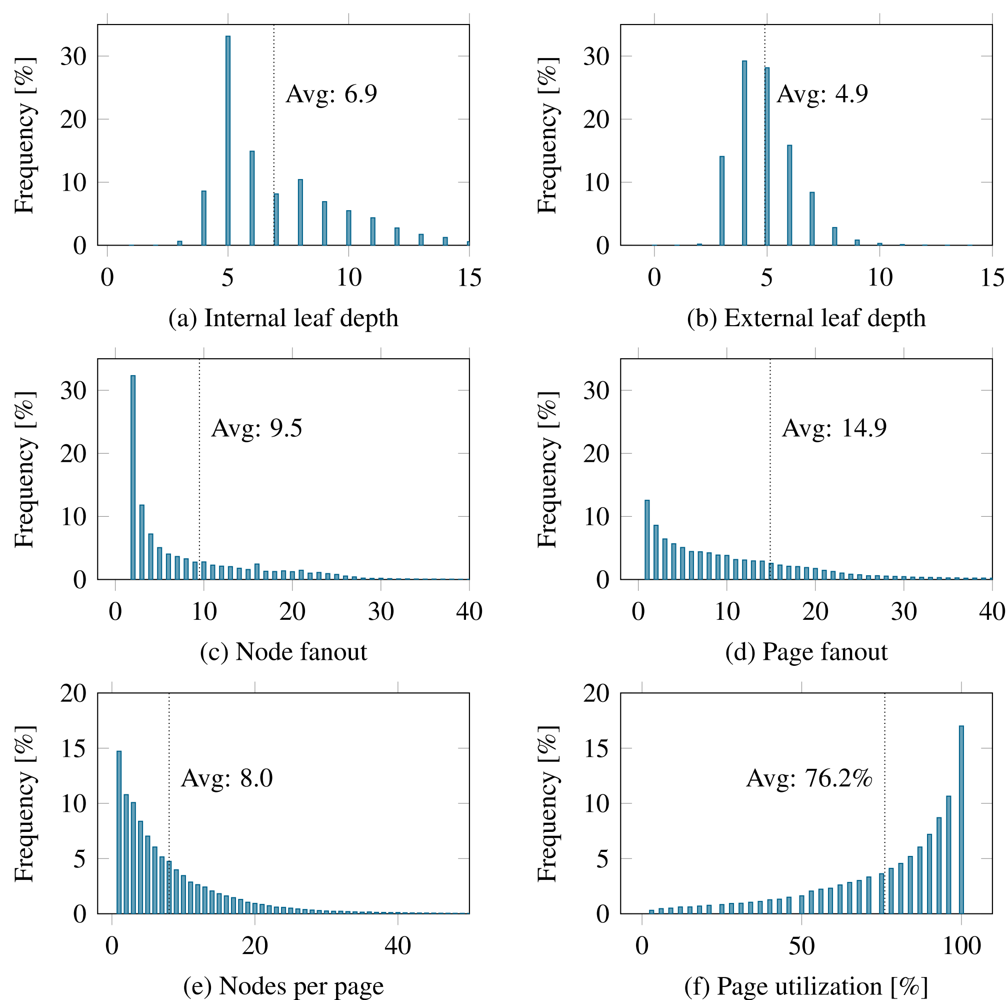
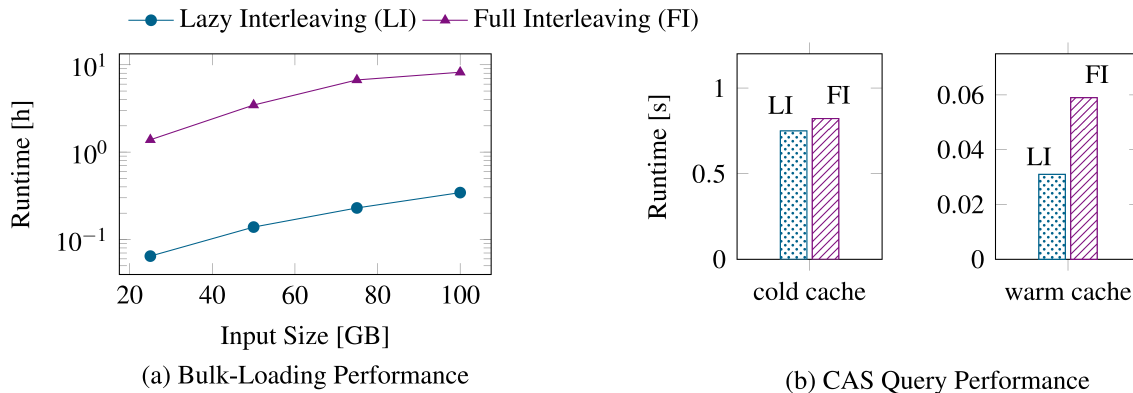


Figure C.6: Structure of the RCAS+ index

Lazy interleaving improves the performance of the CAS queries from Section C.8.2 on cold caches by 8% and on warm caches by 45% (see Figure C.7b), because once we reach a leaf page, often all suffixes in a leaf node match the query and a linear scan of these suffixes is faster than traversing the corresponding subtree that is created without lazy interleaving. Thus, lazy interleaving improves bulk-loading time *and* query performance.

C.8.5 Proactive Partitioning

RCAS+ proactively computes the discriminative bytes during the ψ-partitioning to reduce disk I/O. We compare proactive partitioning with the two-pass approach, which scans a partition once to compute the discriminative bytes and a second time to partition the data. Similar to Algorithm



(a) Bulk-Loading Performance

(b) CAS Query Performance

Figure C.7: Lazy interleaving improves bulk-loading runtime

C.4 the two-pass approach computes the discriminative bytes by picking a reference key and comparing it to all other keys.

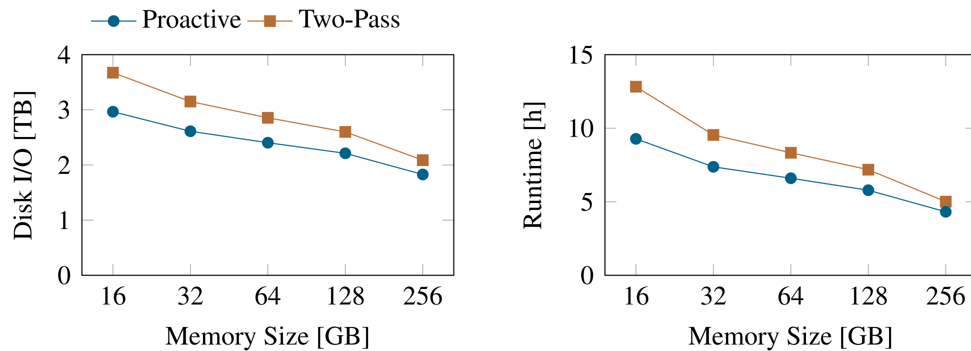


Figure C.8: Discriminative byte computation

In Figure C.8a we measure the disk I/O during bulk-loading, which consists of reading the input data, writing and reading intermediate partitions, and writing the final RCAS$^+$ index. Increasing the memory size reduces the disk I/O because more intermediate partitions can be kept in memory. Proactively computing the discriminative bytes outperforms two-pass because it reads only the root partition twice, while two-pass needs to scan *every* partition twice. The difference is more pronounced if the memory size is small because there are more disk-resident partitions.

We compare the bulk-loading runtime of the two approaches in Figure C.8b. We use direct I/O that reads and writes partitions directly from/to disk and bypasses OS caches to avoid measuring caching effects. Building RCAS$^+$ with proactive partitioning is faster than with the two-pass approach because bulk-loading is I/O-bound and the disk I/O of proactive partitioning is lower.

C.8.6 Front-Loading

Front-loading buffers partitions in memory during bulk-loading. Figure C.9 shows how memory is distributed in the root partition table when we bulk-load RCAS+ with 100 GB memory. The root partition table is obtained by ψ-partitioning the root partition in dimension V. The x-axis shows the index of the partition and the y-axis shows the percent of all pages that are located in a partition. Front-loading keeps the first partitions in the table memory-resident, followed by one hybrid partition (`0x51`) and a sequence of disk-resident partitions. The partitions are not equally big because the size of a partition depends on the data distribution. In the GitLab dataset dimension V corresponds to the commit time of a revision and, e.g., the hybrid partition `0x51` contains revisions from January to August, 2013. Since the amount of archived software artifacts grows over time [RCZ20], the size of partitions also increases over time.

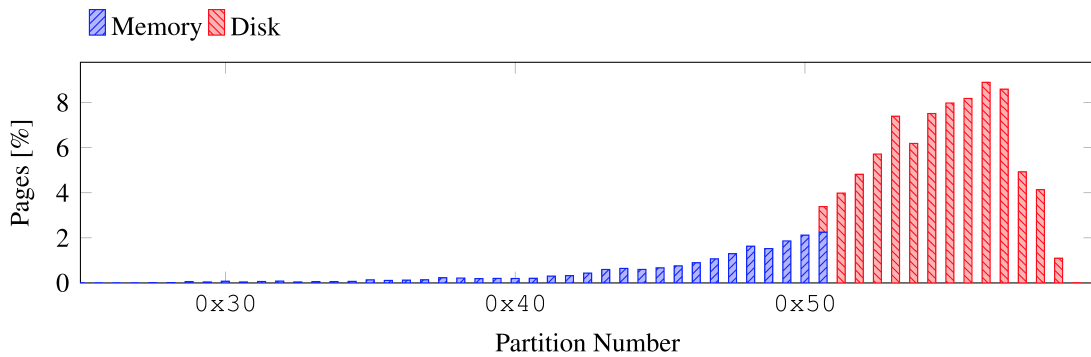


Figure C.9: Memory distribution in root partition table

We compare front-loading with a baseline, termed all-or-nothing, that keeps partitions completely in memory or on disk. With all-or-nothing, the ψ-partitioning $\psi(K,D) = \{K_1, K_2, \ldots\}$ reads K from disk and writes all K_i to disk as long as K does not fit into memory. Once K fits into memory, it is processed there.

In Figure C.10a we measure the disk I/O as we increase the memory size. While front-loading is able to use the additional memory to reduce disk I/O, all-or-nothing does not improve as we increase the memory size beyond 64 GB. This is because the GitLab dataset (550 GB) does not fit into memory, but every subsequent partition is smaller than 64 GB. Thus, all-or-nothing reads the root partition and writes all resulting partitions do disk, whereas front-loading keeps some of them in memory. For the bulk-loading runtime in Figure C.10b we observe a similar pattern as in Figure C.8b. The runtime improves proportionally with the disk I/O and thus front-loading outperforms all-or-nothing.

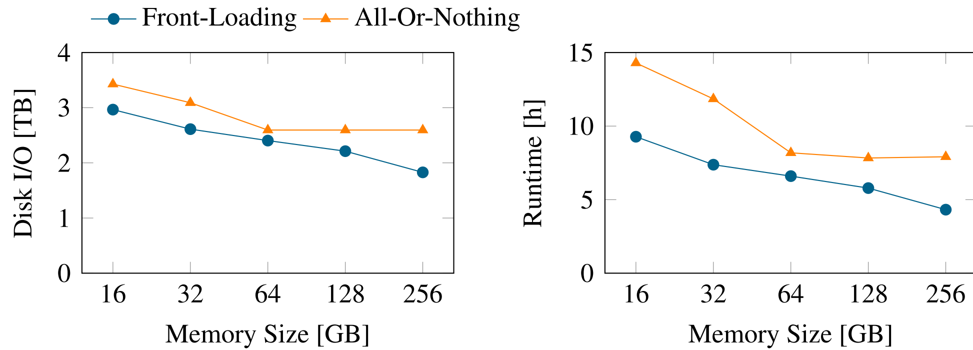


Figure C.10: Memory management

C.8.7 Cost Model

We evaluate the cost model from Lemma C.3 that measures the I/O overhead of our bulk-loading algorithm for a uniform data distribution. The I/O overhead is the number of page transfers to read/write intermediate results during bulk-loading. We multiply the I/O overhead with the default page size of 16 KB to get the number of bytes that are transferred to and from disk. The cost model in Lemma C.3 has four parameters: N, M, B, and f (see Section C.7). We set fanout $f = 10$ since this is the average fanout of a node in RCAS$^+$ for the GitLab dataset, see Figure C.6c. The cost model assumes that M (B) keys fit into memory (a page). Therefore, we set $B = \lceil \frac{16\,\text{KB}}{80\,\text{B}} \rceil = 205$, where 16 KB is the page size and 80 is the average key length (see Section C.8.3). Similarly, if the memory size is 50 GB we can store $M = \lceil \frac{50\,\text{GB}}{80\,\text{B}} \rceil = 625$ million keys in memory.

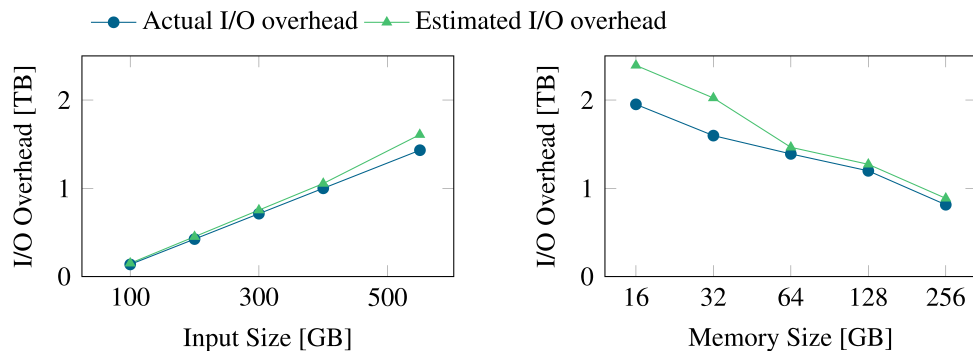


Figure C.11: Cost model evaluation

In Figure C.11a we compare the actual and the estimated I/O overhead to bulk-load RCAS$^+$ as we increase the input size, keeping the memory size fixed at 50 GB. The estimated and actual cost are

within 10% of each other. In Figure C.11b we vary the memory size and fix the GitLab dataset as input. Increasing the memory size makes the estimate more accurate because the ratio N/M between input and memory size in the cost model decreases. This ratio determines the number of levels in the hierarchical partitioning before partitions fit completely into memory. If there are only few levels before we reach small enough partitions that can be processed entirely in memory, the hierarchical partitioning is more balanced and hence the estimated I/O matches the actual I/O better (our cost model in Lemma C.3 assumes a balanced tree). The actual I/O overhead is lower than the estimated overhead because the cost model assumes that B (the number of keys that fit into a page) is constant, but in practice B increases since we drop the longest common prefixes from each key during the partitioning and hence more keys fit into a page as we progress.

C.8.8 Summary

RCAS$^+$ can be built faster and for larger input sizes than RCAS because it does not deteriorate when the input size exceeds the memory size. RCAS$^+$ scales because of the following techniques. Node clustering aligns nodes on page-structured disks and shows a page utilization of 76%, on average. Lazy interleaving improves the bulk-loading runtime by a factor of 20 and improves CAS query performance by up to 45%. Proactive partitioning reduces the disk I/O during bulk-loading by 20% w.r.t. two-pass approach since it avoids unnecessary scans over the data to compute the discriminative bytes. Front-loading cuts disk I/O by up to 30% by buffering intermediate partitions. Lastly, our cost model for uniform data yields a good estimate that is within 20% of the true I/O overhead during bulk-loading even if the data is not uniformly distributed.

C.9 Related Work

Existing CAS indexes [CSF$^+$01, KKNR04, LAAE06, MHSB15, STR$^+$15] are not robust for CAS queries because they either build separate indexes for content and structure that need to be joined [KKNR04, MHSB15] or they fix the order of the dimensions a priori [CSF$^+$01, LAAE06]. This fails for CAS queries with large intermediate results and a small final result. The in-memory RCAS index [WBH20] is robust since it tightly integrates the content and structure with its dynamic interleaving scheme but it is not scalable. RCAS$^+$ is the first robust and scalable CAS index. The following five core techniques make this possible: lazy interleaving reduces the cost of the dynamic interleaving; node clustering aligns nodes on pages to store RCAS$^+$ on disk;

proactive partitioning pre-computes the discriminative bytes during the partitioning; depth-first bulk-loading has a small memory overhead; and front-loading manages what data is stored in the remaining memory and what on disk. In the following we review relevant related techniques to build index structures.

The standard technique to build B-trees is to sort the data with external sorting [Knu98] and build the index bottom up [KPT91]. Ma et al. [MF14] use sort-based bulk-loading for the B-trie [AZ09]. Sort-based bulk-loading does not work for RCAS$^+$ because the dynamic interleaving of all keys has to be computed before sorting, which is as expensive as building RCAS$^+$ in the first place. Space-filling curves like z-order are used to linearize keys [FKM$^+$00, HS02, KF93, KDZP19, QTCZ20]. These interleavings are *static* since they are applied to each key individually and ignore the data distribution. They prioritize one dimension over another and have poor query performance if keys contain long common prefixes since interleaving at a common prefix does not partition the data [WBH20].

Arge et al. [Arg03, AHVV02] equip each node in a tree with a buffer. Insertions are first batched at the root node's buffer and they are flushed one level down when a buffer overflows. Batching insertions allows I/O efficient bulk-loading. Since we need to look at *all* keys to dynamically interleave them, inserting keys in small batches as in the buffer tree and related approaches [AS13, dBSW97] is not possible.

Van den Bercken et al. [dBS01] build an index in memory top-down from a sample of the data, attach disk-based buffers to each leaf node, insert the remaining keys into the leaf buffers, and recursively call the algorithm on each leaf buffer. This and other sampling-based approaches [AS10] do not work for RCAS$^+$ because we cannot determine the dynamic interleaving from a sample of the keys.

Ghanem et al. [GSM$^+$04] allocate half the memory to build a tree in memory and half the memory is used as buffer that extends to disk if necessary. Keys are inserted one by one as long as the tree has still space, after that keys are added to the buffer and each buffer is recursively processed. Inserting keys one by one is not possible due to the dynamic interleaving, which is applied to all keys at once. While Ghanem et al. use half the memory for nodes and half for buffering, our algorithm uses only $O(h \cdot b)$ memory for nodes, where h is the height of the tree and b is the page size (see Lemma C.5), and front-loading uses the remaining memory for buffering.

Leis et al. [LKN13] build their trie-based main-memory index top-down by radix-partitioning the data one byte after the other (starting from the most-significant byte). Radix partitioning

groups keys together that have the same value for some of the keys' bits/bytes and it is used, e.g., for radix sorting [CBB+15, Knu98, OKFS19, PR14] and aggregation [MSL+15]. Radix partitioning is order-preserving if keys are binary comparable [LKN13] and prefix-preserving since it groups keys that have a common prefix. Radix partitioning does not guarantee progress since partitioning at a byte that is the same for all keys does not partition the data.

Range partitioning splits a set of keys based on a given set of range splitters [GD90], which can be derived, e.g., from quantiles [DNS91] or samples [MRL98]. It is used to, e.g., horizontally partition (shard) the data in distributed database systems [HCZZ21]. Range partitioning is order-preserving since the keys are split into non-overlapping ranges, but it is not prefix-preserving because multiple partitions can share the same longest common prefix.

Hash partitioning uniformly maps keys from a large domain to a smaller domain using a pre-defined hash function. It is widely used, e.g., in hash joins [BTAÖ15, KTM83] and aggregation [MSL+15]. Standard hash functions do not consider the data distribution and hash partitioning is neither order- nor prefix-preserving.

We cluster nodes to align them with a page-structured storage layout. Many approaches exist to cluster nodes such that no cluster exceeds a given size limit [BZP+18, KM06a, KM06b, KK04, KM77]. [KM77] minimizes the number of clusters, while [KK04] minimizes the height of the tree. Kanne et al. [KM06a] cluster sibling subtrees with their right-sibling (RS) algorithm. We use the RS algorithm since it is fast and scales to large datasets. RS does not guarantee a minimal number of clusters, but in practice the clustering is more compact than [KK04, KM77]. A dynamic-programming algorithm has been proposed to find the minimal clustering but it is five orders of magnitude slower than RS and only provides 10% fewer clusters [KM06b]. We cannot use this dynamic programming algorithm since it loads the tree into memory before clustering.

C.10 Conclusion and Outlook

We propose RCAS+ that scales the RCAS index [WBH20] to large datasets. To build RCAS+ at scale we propose five techniques that optimize CPU, memory, and disk usage. *Lazy interleaving* reduces the cost of the dynamic interleaving to interleave two-dimensional keys into a one-dimensional byte-string. *Node clustering* aligns nodes on page-structured devices. *Proactive partitioning* reduces disk I/O by pre-computing information in one level of the hierarchical partitioning that is needed in the next level. *Depth-first bulk-loading* has only a small memory

overhead and the remaining memory is used by *front-loading* to optimally buffer partitions during the hierarchical partitioning. We evaluate our algorithm analytically and experimentally, and we show-case RCAS$^+$'s scalability by indexing the revisions of all public GitLab repositories archived by Software Heritage, for a total of 6.9 billion modified files in 120 million commits across 0.9 million repositories.

In terms of future work we are working on supporting updates in RCAS and RCAS$^+$ [WPBH21]. We also plan to build RCAS$^+$ in parallel to better utilize the CPU. Each partition created during the partitioning can be processed in parallel, but the challenge is to assign the partitions equally to the available CPU cores since the partitioning can be unbalanced depending on the data distribution. In addition, memory management becomes more difficult since partitions in a partition table are no longer processed sequentially from the first to the last partition.

APPENDIX D

Curriculum Vitae

Kevin Wellenzohn

Personal Data

Date of Birth:	10 May 1991
Citizenship:	Italian
Address:	Staatsstrasse 66 39020 Kastelbell-Tschars (BZ) Italy
Phone:	+39 349 527 25 46
email:	wellenzohn@ifi.uzh.ch

Education

10.2015 to *present*	PhD Student at the Database Technologies Group **University of Zurich**, Switzerland Supervisor: Prof. Michael Böhlen Expected Graduation: February 2022
10.2013 to 07.2015	Master in Computer Science Curriculum: Data and Knowledge Engineering **Free University of Bolzano**, Italy 113/110 cum laude Thesis: “Imputation of Missing Values in Highly Correlated Streams of Time Series Data” Supervisor: Prof. Johann Gamper
10.2010 to 07.2013	Bachelor in Computer Science and Engineering **Free University of Bolzano**, Italy 113/110 cum laude Thesis: “An Efficient Heuristic for Orienteering with Categories” Supervisor: Prof. Sven Helmer
05.2005 to 06.2010	**Gewerbeoberschule Schlanders**, Italy Final Grade: 100/100

Work Experience

10.2021 to *present*	Data Engineer Smart-Dato GmbH Project: Development of a blockchain-based a e-voting system.
10.2015 to *present*	PhD Student at University of Zurich Project: The aim of this work is to index the content and Structure of semi-structured, hierarchical data. Responsible: Prof. Johann Gamper
12.2013 to 10.2015	Research Assistant at Free University of Bolzano *Center for Information and Database Systems Engineering* Project: DASA. The aim of this work is the design, development, implementation and evaluation of algorithmic solutions for the analysis of time series data in the agriculture sector. A specific focus is on the imputation of missing values and similarity search of time series data. Responsible: Prof. Johann Gamper

07.2012 to 12.2013	Research Assistant at Free University of Bolzano *Center for Information and Database Systems Engineering* Project: OSTAR. The aim of this work is to implement, test and evaluate efficient algorithms for itinerary planning in the presence of categories and opening hours for the points of interest. The algorithms are to be tested on both real world data and artificial data. Responsible: Prof. Johann Gamper, Prof. Sven Helmer
06.2011 to 09.2011 06.2010 to 09.2010 06.2009 to 09.2009	Summer Intern at Selimex GmbH, Latsch Assisting the Head of Accounting in an internship at the accounting department of a large food and fruit trading company in South Tyrol, Italy.

Publications

[1] K. Wellenzohn, M. H. Böhlen, and S. Helmer, “Dynamic interleaving of content and structure for robust indexing of semi-structured hierarchical data,” *PVLDB*, vol. 13, no. 10, pp. 1641–1653, 2020.

[2] K. Wellenzohn, L. Popovic, M. H. Böhlen, and S. Helmer, “Inserting keys into the robust content-and-structure (RCAS) index,” in *ADBIS*, 2021.

[3] K. Wellenzohn, M. H. Böhlen, A. Dignös, J. Gamper, and H. Mitterer, “Continuous imputation of missing values in streams of pattern-determining time series,” in *EDBT*, pp. 330–341, 2017.

[4] P. Bolzoni, S. Helmer, K. Wellenzohn, J. Gamper, and P. Andritsos, “Efficient itinerary planning with category constraints,” in *SIGSPATIAL*, pp. 203–212, 2014.

Languages

German:	Mother tongue
English:	C1 (CAE certified)
Italian:	B1

Bibliography

[AAA+14] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source bdms. *PVLDB*, 7(14):1905–1916, 2014. doi:10.14778/2733085.2733096.

[ACZ18] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Commun. ACM*, 61(10):29–31, 2018. doi:10.1145/3183558.

[AHVV02] Lars Arge, Klaus H. Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002. doi:10.1007/s00453-001-0107-6.

[Apa20] Apache. Apache Jackrabbit Oak. https://jackrabbit.apache.org/oak/, 2020. [Online; accessed May 2020].

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/s00453-003-1021-x.

[AS10] Lior Aronovich and Israel Spiegler. Bulk construction of dynamic clustered metric trees. *Knowl. Inf. Syst.*, 22(2):211–244, 2010. doi:10.1007/s10115-009-0195-1.

[AS13] Daniar Achakeev and Bernhard Seeger. Efficient bulk updates on multiversion b-trees. *PVLDB*, 6(14):1834–1845, 2013. URL: http://www.vldb.org/pvldb/vol6/p1834-achakeev.pdf, doi:10.14778/2556549.2556566.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

[AZ09] Nikolas Askitis and Justin Zobel. B-tries for disk-based string management. *VLDB J.*, 18(1):157–179, 2009. doi:10.1007/s00778-008-0094-1.

[BCF+10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language (second edition). W3C recommendation, W3C, December 2010. URL: https://www.w3.org/TR/xquery/.

[BFF+15] Robert Brunel, Jan Finis, Gerald Franz, Norman May, Alfons Kemper, Thomas Neumann, and Franz Färber. Supporting hierarchical data in SAP HANA. In *ICDE*, pages 1280–1291, 2015. doi:10.1109/ICDE.2015.7113376.

[BG96] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, 1996. doi:10.1145/235809.235810.

[BKH+17] Radim Baca, Michal Krátký, Irena Holubová, Martin Necaský, Tomás Skopal, Martin Svoboda, and Sherif Sakr. Structural XML query processing. *ACM Comput. Surv.*, 50(5):64:1–64:41, 2017. doi:10.1145/3095798.

[BTAÖ15] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *TKDE*, 27(7):1754–1766, 2015. doi:10.1109/TKDE.2014.2313874.

[BZP+18] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. Hot: A height optimized trie index for main-memory database systems. In *SIGMOD*, pages 521–534, 2018. doi:10.1145/3183713.3196896.

[CBB+15] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent KulandaiSamy, and Ruchir Puri. PARADIS: an efficient parallel algorithm for in-place radix sort. *PVLDB*, 8(12):1518–1529, 2015. doi:10.14778/2824032.2824050.

[CD99] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. W3C recommendation, W3C, 1999. URL: https://www.w3.org/TR/xpath/.

[Cou20] CouchDB. CouchDB. http://couchdb.apache.org/, 2020. [Online; accessed May 2020].

[CSF+01] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, 2001. URL: http://www.vldb.org/conf/2001/P341.pdf.

[dBS01] Jochen Van den Bercken and Bernhard Seeger. An evaluation of generic bulk loading techniques. In *VDLB*, pages 461–470, 2001. URL: http://www.vldb.org/conf/2001/P461.pdf.

[dBSW97] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB*, pages 406–415, 1997. URL: http://www.vldb.org/conf/1997/P406.PDF.

[DCL18] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on NoSQL stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018. doi:10.1145/3158661.

[DCZ17] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage: Why and how to preserve software source code. In *iPRES*, 2017.

[DNRN15] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, 2015. doi:10.1145/2803171.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, pages 280–291, 1991. doi:10.1109/PDIS.1991.183115.

[FBK+13] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Franz Färber, and Norman May. Deltani: an efficient labeling scheme for versioned hierarchical data. In *SIGMOD*, pages 905–916, 2013. doi:10.1145/2463676.2465329.

[FBK+15] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. Indexing highly dynamic hierarchical data. *PVLDB*, 8(10):986–997, 2015. doi:10.14778/2794367.2794369.

[FBK+17] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Färber. Order indexes: supporting highly dynamic hierarchical data in relational main-memory database systems. *VLDB J.*, 26(1):55–80, 2017. doi:10.1007/s00778-016-0436-3.

[FF13] Daniela Florescu and Ghislain Fourny. Jsoniq: The history of a query language. *IEEE Internet Comput.*, 17(5):86–90, 2013. doi:10.1109/MIC.2013.97.

[FHK+02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native XML base management system. *VLDB J.*, 11(4):292–314, 2002. doi:10.1007/s00778-002-0080-y.

[FKM+00] Robert Fenk, Akihiko Kawakami, Volker Markl, Rudolf Bayer, and Shunji Osaki. Bulk loading a data warehouse built upon a ub-tree. In *IDEAS*, pages 179–187, 2000. doi:10.1109/IDEAS.2000.880576.

[GD90] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *VLDB*, pages 481–492, 1990. URL: http://www.vldb.org/conf/1990/P481.PDF.

[GGKP12] Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. Robust query processing (Dagstuhl Seminar 12321). *Dagstuhl Reports*, 2(8):1–15, 2012. doi:10.4230/DagRep.2.8.1.

[Gra11] Goetz Graefe. Robust query processing. In *ICDE*, page 1361, 2011. doi:10.1109/ICDE.2011.5767961.

[GSM+04] Thanaa M. Ghanem, Rahul Shah, Mohamed F. Mokbel, Walid G. Aref, and Jeffrey Scott Vitter. Bulk operations for space-partitioning trees. In *ICDE*, pages 29–40, 2004. doi:10.1109/ICDE.2004.1319982.

[GW97] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997. URL: http://www.vldb.org/conf/1997/P436.PDF.

[Has08] Ahmed E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM)*, pages 48–57, 2008. doi:10.1109/FOSM.2008.4659248.

[HCZZ21] Xialong He, Peng Cai, Xuan Zhou, and Aoying Zhou. Continuously bulk-loading over range partitioned tables for large scale historical data. In *ICDE*, pages 960–971, 2021.

[Hil91] David Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[HM16] Ruining He and Julian J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, pages 507–517, 2016. doi:10.1145/2872427.2883037.

[HS02] Gísli R. Hjaltason and Hanan Samet. Speeding up construction of PMR quadtree-based spatial indexes. *VLDB J.*, 11(2):109–137, 2002. doi:10.1007/s00778-002-0067-8.

[HZW02] Steffen Heinz, Justin Zobel, and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002. doi:10.1145/506309.506312.

[JNS+97] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, pages 16–25, 1997.

[KCK+03] Howard Katz, Don Chamberlin, Michael Kay, Philip Wadler, and Denise Draper. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley, Boston, 2003.

[KDZP19] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *VLDB J.*, 28(6):847–869, 2019. doi:10.1007/s00778-019-00573-w.

[KF93] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *CIKM*, pages 490–499, 1993. doi:10.1145/170088.170403.

[KK04] András Kovács and Tamás Kis. Partitioning of trees for minimizing height and cardinality. *Inf. Process. Lett.*, 89(4):181–185, 2004. doi:10.1016/j.ipl.2003.11.004.

[KKNR04] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD*, pages 779–790, 2004. doi:10.1145/1007568.1007656.

[KM77] Sukhamay Kundu and Jayadev Misra. A linear tree partitioning algorithm. *SIAM J. Comput.*, 6(1):151–154, 1977. doi:10.1137/0206012.

[KM06a] Carl-Christian Kanne and Guido Moerkotte. The importance of sibling clustering for efficient bulkload of XML document trees. *IBM Syst. J.*, 45(2):321–334, 2006. doi:10.1147/sj.452.0321.

[KM06b] Carl-Christian Kanne and Guido Moerkotte. A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in natix. In *VLDB*, pages 91–102, 2006. URL: http://dl.acm.org/citation.cfm?id=1164137.

[Knu98] Donald Ervin Knuth. *The art of computer programming, Volume III, Sorting and Searching, 2nd Edition*. Addison-Wesley, 1998.

[KPT91] Timothy M. Klein, Kenneth J. Parzygnat, and Alan L. Tharp. Optimal b-tree packing. *Inf. Syst.*, 16(2):239–243, 1991. doi:10.1016/0306-4379(91)90017-4.

[KTM83] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Gener. Comput.*, 1(1):63–74, 1983. doi:10.1007/BF03037022.

[LAAE06] Hua-Gang Li, S. Alireza Aghili, Divyakant Agrawal, and Amr El Abbadi. FLUX: content and structure matching of xpath queries with range predicates. In *XSym*, pages 61–76, 2006. doi:10.1007/11841920_5.

[LC19] Chen Luo and Michael Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29, 07 2019. doi:10.1007/s00778-019-00555-y.

[LH19] Jiaheng Lu and Irena Holubová. Multi-model databases: A new journey to handle the variety of data. *ACM Comput. Surv.*, 52(3):55:1–55:38, 2019. doi:10.1145/3323214.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013. doi:10.1109/ICDE.2013.6544812.

[LLS13] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, 2013. doi:10.1109/ICDE.2013.6544834.

[Mar99] Volker Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. PhD thesis, Technical University of Munich, 1999.

[MDB+21] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. World of code: enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empir. Softw. Eng.*, 26(2):22, 2021. doi:10.1007/s10664-020-09905-9.

[MF14] Dongzhe Ma and Jianhua Feng. A generic approach for bulk loading trie-based index structures on external storage. In *Web-Age Information Management WAIM*, volume 8485, pages 55–66, 2014. doi:10.1007/978-3-319-08010-9_8.

[MHSB15] Christian Mathis, Theo Härder, Karsten Schmidt, and Sebastian Bächle. XML indexing and storage: fulfilling the wish list. *Computer Science - R&D*, 30(1), 2015. doi:10.1007/s00450-012-0204-6.

[Mon20] MongoDB. MongoDB Indexing. https://docs.mongodb.com/v4.0/indexes, 2020. [Online; accessed May 2020].

[Mor66] G.M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[Mor68] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968. doi:10.1145/321479.321481.

[MRL98] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pages 426–435, 1998. doi:10.1145/276304.276342.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. doi:10.1017/CBO9780511809071.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999. doi:10.1007/3-540-49257-7_18.

[MSL+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*, pages 1123–1136, 2015. doi:10.1145/2723372.2747644.

[NS08] Bradford G. Nickerson and Qingxiu Shi. On k-d range search with patricia tries. *SIAM J. Comput.*, 37(5):1373–1386, 2008. doi:10.1137/060653780.

[NY17] Shoji Nishimura and Haruo Yokota. QUILTS: multidimensional data partitioning framework based on query-aware and skew-tolerant space-filling curves. In *SIGMOD*, pages 1525–1537, 2017. doi:10.1145/3035918.3035934.

[OCGO96] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. doi:10.1007/s002360050048.

[OKFS19] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. Theoretically-efficient and practical parallel in-place radix sorting. In *SPAA*, pages 213–224, 2019. doi:10.1145/3323165.3323198.

[OM84] Jack A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS*, pages 181–190, 1984. doi:10.1145/588011.588037.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly XML node labels. In *SIGMOD*, pages 903–908, 2004. doi:10.1145/1007568.1007686.

[PHD16] Danila Piatov, Sven Helmer, and Anton Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016. doi:10.1109/ICDE.2016.7498316.

[PR14] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014. doi:10.1145/2588555.2610522.

[PSZ20] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset: Large-scale analysis of public software development history. In *MSR*, pages 138–142, 2020. doi:10.1145/3379597.3387510.

[QTCZ20] Jianzhong Qi, Yufei Tao, Yanchuan Chang, and Rui Zhang. Packing r-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability. *TODS*, 45(3):14:1–14:47, 2020. doi:10.1145/3397506.

[RCZ20] Guillaume Rousseau, Roberto Di Cosmo, and Stefano Zacchiroli. Software provenance tracking at the scale of public source code. *Empir. Softw. Eng.*, 25(4):2930–2959, 2020. doi:10.1007/s10664-020-09828-5.

[RMF+00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the ub-tree into a database system kernel. In *VLDB*, pages 263–272, 2000. URL: http://www.vldb.org/conf/2000/P263.pdf.

[Sam06] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann series in data management systems. Academic Press, 2006.

[SJM+17] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge-Arnulfo Quiané-Ruiz, and Aaron J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In *SoCC*, pages 229–241, 2017. doi:10.1145/3127479.3131613.

[SL76] Dennis G. Severance and Guy M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976. doi:10.1145/320473.320484.

[STR+15] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin J. Levandoski, and David B. Lomet. Schema-agnostic indexing with azure DocumentDB. *PVLDB*, 8(12):1668–1679, 2015. doi:10.14778/2824032.2824065.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002. doi:10.1016/B978-155860869-6/50096-2.

[TSK+10] Ilya Taranov, Ivan Shcheklein, Alexander Kalinin, Leonid Novak, Sergei D. Kuznetsov, Roman Pastukhov, Alexander Boldakov, Denis Turdakov, Konstantin Antipin, Andrey Fomichev, Peter Pleshachkov, Pavel Velikhov, Nikolai Zavaritski, Maxim Grinev, Maria P. Grineva, and Dmitry Lizorkin. Sedna: native XML database management system (internals overview). In *SIGMOD*, pages 1037–1046, 2010. doi:10.1145/1807167.1807282.

[WBH20] Kevin Wellenzohn, Michael H. Böhlen, and Sven Helmer. Dynamic interleaving of content and structure for robust indexing of semi-structured hierarchical data. *PVLDB*, 13(10):1641–1653, 2020. doi:10.14778/3401960.3401963.

[WPBH21] Kevin Wellenzohn, Luka Popovic, Michael Böhlen, and Sven Helmer. Inserting keys into the robust content-and-structure (rcas) index. In *ADBIS*, pages 121–135, 2021. doi:10.1007/978-3-030-82472-3_10.

[ZLL+18] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In *SIGMOD*, pages 323–336, 2018. doi:10.1145/3183713.3196931.