Efficient Indexing for Big Data in Hadoop MapReduce and Main Memory Databases

Stefan Richter

Thesis for obtaining the title of
Doctor of Engineering
of the Faculties of Natural Sciences and Technology
of Saarland University

Saarbrücken, Germany
2015

Dean of the Faculty	Prof. Dr. Frank-Olaf Schreyer
Day of Colloquium	25.05.2016

Examination Board:

Chairman	Prof. Dr. Jörg Hoffmann
Adviser and First Reviewer	Prof. Dr. Jens Dittrich
Second Reviewer	Prof. Dr. Jan Reineke
Third Reviewer	Prof. Dr. Wolfgang Lehner
Academic Assistant	Dr. Alvaro Torralba

To my family

Acknowledgements

I want to express my sincere gratitude to everybody who supported me in my doctoral research at Saarland University and contributed to the success of this work. First, I want to thank my advisor Prof. Dr. Jens Dittrich for the opportunity and encouragement to pursue my Ph.D. under his supervision. His motivation, inspiration, experience, and continuous support on many levels have been invaluable for my research and this thesis. Besides my advisor, I also want to thank Prof. Dr. Jan Reineke for reviewing my thesis.

My sincere thanks go to Jorge Quiané and Victor Alvarez for their professional and moral support. They have been role models of postdoctoral researchers to me and I greatly enjoyed working with them. I thank all current and former fellow Ph.D. students at our chair: Stefan Schuh, Endre Palatinus, Felix Martin Schuhknecht, Alekh Jindal, and Jörg Schad. In the past years, many of you have become friends to me and I enjoyed our stimulating discussions and our teamwork. I would also like to thank our secretary Angelika Scholl-Danopoulos for keeping our backs free from administrative works.

Last but not the least, I would like to thank my family: my parents, grandparents, and my sister for their endless love and understanding. You have supported me in countless ways throughout writing this thesis and my life in general.

I acknowledge that parts of the research work presented in this thesis were supported by funding from the German Ministry of Education and Science (BMBF).

Abstract

In recent years, the database community has witnessed the advent and breakthrough of many new system designs like the famous Hadoop MapReduce or main memory databases like MonetDB, Hekaton, SAP Hana, and HyPer to solve the problems of "*Big Data*". The system architectures in this generation of emerging systems often radically differ from traditional relational databases. For database research, this trend creates new challenges to design and optimize data structures for those novel architectures. Premier candidates for innovations are index structures, which are traditionally among the most crucial performance factors in databases. In this thesis, we focus on efficient indexing methods for Hadoop MapReduce and main memory databases. Our work consists of three independent parts that resulted from different research projects.

In the first part, we introduce *HAIL*, a novel approach for efficient static and adaptive indexing in Hadoop MapReduce. We believe that efficient indexing becomes increasingly important in the context of very large data sets. HAIL combines very low index build times that are often even invisible for users, with significant runtime improvements for selective MapReduce jobs. We provide extensive experiments, and show that HAIL can improve job runtimes by up to 68x over Hadoop.

In the second part of this thesis, we present an in-depth evaluation of the adaptive radix tree *ART*, a recent and very promising competitor in the domain of tree-indexes for main memory databases. ART was reported by its inventors to be significantly faster than previous tree indexes and even competitive to hash tables. However, the original evaluation of ART did not consider *Judy Arrays*, which is, to the best of our knowledge, the first data structure introducing adaptivity to radix trees. Furthermore, the hash table used in the comparison with ART was just a textbook implementation of chained hashing and not a more sophisticated state-of-the-art hash tables. We provide an extended analysis and experimental evaluation of ART, including a detailed comparison to Judy Arrays, hashing via quadratic probing, and three variants of Cuckoo hashing. Our results give a more differentiated look on ART. In particular, we present striking conceptual similarities between ART and Judy Arrays and show that well-engineered hash tables can beat the lookup throughput of adaptive radix trees by up 6x.

In the third part, motivated by our previous results, we take a closer look at *hashing methods* in main memory databases. We identify seven key factors that influence hashing performance, evaluate their impact, and discuss the implications on hashing in modern databases. Our study indicates that choosing the right hashing method and configuration can make an order of magnitude difference in insert and lookup performance. We also provide a guideline for practitioners on when to use which hashing method.

Zusammenfassung

In den letzten Jahren hat die Datenbank-Community das Aufkommen und den Durchbruch von vielen neuartigen Systementwürfen erlebt, deren übergeordnetes Ziel es ist, die Probleme im Bereich "Big Data" zu lösen. Wichtige Beispiele hierfür sind das berühmte Hadoop MapReduce oder Hauptspeicher-Datenbanken wie MonetDB, Hekaton, SAP Hana und Hyper. Die Architekturen in dieser Generation von Systemen unterscheiden sich oft grundlegend von den Ansätzen traditioneller relationaler Datenbanken. Für die Datenbankforschung schafft diese Entwicklung neue Herausforderungen im Zusammenhang mit dem Entwurf passender Datenstrukturen und deren Optimierung in diesem neuen Kontext. Zu den herausragenden Kandidaten für Innovationen zählen hierbei Indexstrukturen, die traditionell zu den wichtigsten Leistungsfaktoren in Datenbanken gehören. In dieser Arbeit konzentrieren wir uns auf effiziente Indizierungsmethoden für Hadoop MapReduce und Hauptspeicher-Datenbanken. Unser Beitrag besteht dabei aus drei unabhängigen Teilen, die jeweils aus verschiedenen Forschungsprojekten in diesem Bereich entstanden sind.

Im ersten Teil präsentieren wir *HAIL* als neue Methode für effizientes statisches und adaptives Indizieren in Hadoop MapReduce, einem Framework für verteilte Datenverarbeitung auf großen Rechnernetzwerken. Wir glauben, dass effiziente Indizierung im Hinblick auf die sehr großen Datenmengen in typischen Hadoop Systemen besonders wichtig ist. HAIL kombiniert eine sehr niedrige Indizierungszeit, die für die Nutzer zumeist sogar unsichtbar bleibt, mit erheblichen Laufzeitverbesserungen für selektive MapReduce Jobs. Wir präsentieren umfangreiche Experimente hierzu und zeigen, dass HAIL Job-Laufzeiten gegenüber Hadoop um bis zu 68x verbessert.

Im zweiten Teil dieser Arbeit präsentieren wir eine umfassende Analyse von *ART*, einem adaptiven Radix-Baum, der einen sehr vielversprechenden Wettbewerber auf dem Gebiet der Indexbäume für Hauptspeicher-Datenbanken darstellt. Die Erfinder von ART erklären in ihrer Arbeit, dass ART deutlich schneller als vorherige Indexbäume ist und sogar an die Leistung von Hash Tabellen heranreicht. Allerdings fehlt in dieser ursprünglichen Studie ein Vergleich von ART mit *Judy Arrays*, welches nach unserem Wissen der erste adaptive Radix-Baum ist. Des Weiteren basieren die gezeigten Ergebnisse im Zusammenhang mit Hash Tabellen lediglich auf einem Vergleich von ART mit einer einfachen Implementierung von Chained Hashing, anstelle von optimierten Hash Tabellen auf dem neusten Stand der Technik. Wir liefern eine erweiterte Studie zu ART,

einschließlich detaillierter Vergleiche mit Judy Arrays, mit Quadratic Probing sowie mit drei Varianten von Cuckoo Hashing. Unsere Ergebnisse erlauben eine differenziertere Bewertung von ART. Insbesondere erläutern wir auch die auffälligen konzeptionellen Ähnlichkeiten zwischen ART und Judy Arrays. Weiterhin zeigen unsere experimentellen Ergebnisse, dass der Durchsatz für Suchanfragen bei Hash Tabellen um bis zu einem Faktor von 6 höher ist als bei adaptiven Radix-Bäumen.

Im dritten Teil, zu dem uns unsere vorangehenden Ergebnisse motiviert haben, werfen wir einen genaueren Blick auf Hashing-Verfahren für Hauptspeicher-Datenbanken. Wir identifizieren sieben Schlüsselfaktoren, welche die Leistung von Hash Tabellen beeinflussen. Darüber hinaus diskutieren wir die Auswirkungen dieser Faktoren und die Konsequenzen für Hashing in modernen Datenbanken. Unsere Studie zeigt deutlich, dass die Wahl der richtigen Hash-Verfahren und deren korrekte Konfiguration einen Unterschied von einer Größenordnung bei der Leistung für Einfüge- und Suchoperationen machen kann. Abschließend bieten wir auch einen praktischen Leitfaden an, der dabei hilft, das beste Hashing-Verfahren für eine konkrete Problemstellung zu finden.

Contents

Chapter 1

Introduction

1.1 Motivation

In our modern society, the amount of data that is stored and processed grows rapidly year by year. This data is generated everywhere around us, for example on social media websites, by cell phones, financial transactions, logistics, images, videos, and by various other sensors. If we consider this huge increase in data sources in our daily life, it should not come at a surprise that by far the largest share of all digital data in the world was created in very recent years[1]. However, not only the sheer volume of data is increasing, but also its variety, veracity, and the velocity in which data must be handled. These changes imposed huge challenges on database research and demonstrated the limits of traditional approaches to data management.

Apparently, these new problems could not simply be solved by adapting the omnipresent disk-based relational database systems of the past, that have been in the focus of database research for decades. Instead, we have seen the advent of new system designs, which benefit from recent advancements in computer hardware, such as high performance multicore CPUs, huge main memory, or simply the increased hardware cost-efficiency. Maybe the most prominent example from this generation of emerging systems is Hadoop MapReduce, a massively parallel distributed system for large-scale batch processing on unstructured data. The availability of cheap but powerful commodity hardware, low upfront investments, and relative easy-of-use made Hadoop MapReduce a disruptive technology that changed the world by making data processing at petabyte scale widely available. Another important branch of new system designs are main memory (also called in-memory) relational databases. Nowadays, the memory sizes for mainstream servers reaches the scale of terabytes and most users (excluding the largest technology companies) can already store their entire data in the main memory of

[1] According to SINTEF [16], already in 2013 90% of the digital data in the world was created within the last two years.

a single machine. Main memory databases benefit from this development by removing the bottleneck of disk or network I/O from their data processing pipelines. Thus, they provide orders of magnitude improvements in latency and throughput compared to their disk-based ancestors.

We strongly believe that in the context of these changes, we should discuss the consequences for optimal data structures and algorithms that should be employed in those new systems. One premier example in this area are database index structures, which are traditionally among the most crucial factors for the performance of data management systems. Indexes can accelerate data retrieval dramatically, because they allow us to quickly identify and locate records that qualify with respect to a filter condition. Thus, indexes can reduce the amount of data that has to be processed to answer a query by orders of magnitude. However, this benefit is only available if a suitable index for a filter condition is available, and the availability of indexes, in turn, is limited by the fact that indexes do not come for free. Each index consumes resources, such as memory or disc space, and the footprint for most indexes typically grows linearly with the size of the indexed data. Furthermore, indexes must first be created and even maintained in the presence of updates, which is costly. As a result, designing index structures and methods that offer good tradeoffs between search features (e.g. range predicates, prefix searches, multiple dimensions), index performance, and indexing costs is a complex and important research topic. In the context of disk-based relational database systems, indexing has been researched for decades and is still not considered a solved problem. For those observations, several questions arise: How much of those previous results carry over to the new systems? What modifications to index structures should we make, and where are new potentials for innovation?

This thesis is a contribution to the discussion about indexing in a new age of data processing. In particular, we focus on exploring indexing methods in Hadoop MapReduce and main memory databases.

1.2 Overview

This work consist of three independent parts that correspond to the different projects of my doctoral research.

First, we present *HAIL* (Hadoop Adaptive Indexing Library), a novel approach for efficient static and adaptive indexing in the distributed data processing framework Hadoop together with an in-depth evaluation (Chapter 3) and an associated system demonstration (Appendix A).

Then, we shift our focus away from distributed systems towards index structures for main memory databases. In particular, we focus on the adaptive radix tree *ARTful* [76], a recent and promising competitor in the domain of main memory indexes, that was reported to be significantly faster than previous tree indexes and even competitive to hash

tables. In this work we provide an in-depth analysis and experimental comparison of ARTful and relevant prior work, i.e. *Judy Array* [12] and competitive hashing schemes, which were not considered in the original paper (Chapter 4).

Motivated by our results we finally take a closer look at hashing methods for main memory databases. We identify seven key factors that influence hashing performance, evaluate their impact, and discuss the implications for hashing as an essential building block for algorithms in modern database systems (Chapter 5).

As each chapter covers a full research project, we decided to keep all chapters self-contained. This means that there is no consolidation across chapters, for example on introductions and foundational parts. In the following Chapter 2 we provide the backgrounds, as well as details about contributions and previous publications for each of the aforementioned chapters. We also point out the connections of all parts in the context of this work as a whole.

Chapter 2

Background, Contributions, and Publications

2.1 Chapter 3 : HAIL — Hadoop Adaptive Indexing Library

2.1.1 Background

In Chapter 3 we introduce HAIL (*Hadoop Adaptive Indexing Library*), a novel approach for static and adaptive indexing in Hadoop MapReduce [111]. Recently, Hadoop MapReduce has evolved into an important industry standard for massive parallel data processing and has become widely adopted for a variety of use-cases. However, when we started our work on the HAIL project in 2011, Hadoop MapReduce was still a rather young, emerging open source project inspired by Google's famous MapReduce paper [26] that had yet to reach its adolescence. Nevertheless, Hadoop's promise of ease-of-use and scalability on commodity hardware entailed a rapidly growing user base, including important companies like Facebook, IBM, eBay, Twitter, AOL, Adobe and Yahoo!. Hadoop was considered a key technology that could make large scale data processing widely available and affordable for the first time. With this technology at their fingertips, researchers as well as practitioners started to explore the potential of Hadoop as a foundation for various data intensive applications, such as graph algorithms [9] and RDF [57], machine learning [89, 80], data warehousing [108] and a distributed database [47, 1] — to name a few. However, Hadoop MapReduce was originally designed as a framework for parallel and distributed batch computations, scanning over large amounts of unstructured data that is stored on a cluster of commodity machines. It is obvious that this initial design of Hadoop is not well suited or "lacks" certain features for many of the aforementioned applications and, in turn, such applications often have to stretch or extend the boundaries of the system. This also contributes to the unsurpris-

ing observations that Hadoop, despite being very scalable, is rather resource-inefficient compared to more specialized systems.

In 2009, an experimental study by Pavlo et. al. [92] compared Hadoop MapReduce against two parallel DBMSs and found that Hadoop was slower by a large factor for most analytical queries. They concluded that MapReduce is "a major step backwards" from parallel DBMSs. In a follow-up article [27], the authors of the initial MapReduce paper disagreed with this assessment, mentioning that those conclusions were "*(...) based on implementation*[1] *and evaluation shortcomings not fundamental to the MapReduce model*". Both works, however, agree that the lack of indexes and the commitment to "schema-later", which results in parsing overhead at runtime and suboptimal data layout, significantly contributed to the low performance of Hadoop MapReduce. As a result, several papers tried to solve those problems and have shown that indexes [1, 35, 79, 64, 78] and storing data in binary format using optimized layouts [24, 66, 81, 55, 45] can improve the performance of several classes of MapReduce jobs dramatically. However, one major weakness of all previous approaches to indexing and data layouts are high parsing and index creation costs. This shortcoming has inspired our work on HAIL, an approach for indexing in Hadoop that focuses on minimizing and hiding the parsing and index creation costs. HAIL creates different clustered indexes over terabytes of data with only minimal, often invisible overhead and can dramatically improve runtimes of selective MapReduce jobs. This is a huge improvement over previous work where index creation costs are significant and can become almost prohibitive for large datasets. This way we also reduce the performance gap to parallel relational databases.

Research on indexing in Hadoop is still ongoing, and after the publication of HAIL there have been several related works in this area which we will briefly summarize in the following. While HAIL offers record-level indexes per HDFS block, the authors of E3 (Eagle Eyed Elephant) [42] propose split-oriented indexing, i.e. building one index for each file that allows for excluding complete blocks of the file from processing. SpatialHadoop [40] and ScalaGIST [83] extend Hadoop to support multi-dimensional and spatial indexing. However, none of these works focuses on minimizing index creation costs. In this respect, Instant Loading [87] is related to HAIL as it transfers two core ideas, invisible parsing and index creation at insert time to the main memory database HyPer [70]. AIR (Adaptive Index Replacement) [104] is an enhancement for automatic index selection and replacement in HAIL's adaptive indexing pipeline.

2.1.2 Contributions

In the following, we present a detailed lists of contributions for HAIL:

[1]The term "implementation" here refers to Hadoop MapReduce, in contrast to, e.g., Google's undisclosed implementation.

1. **Zero-Overhead Indexing.** We show how to effectively piggyback sorting and index creation on the existing HDFS upload pipeline. This way it is not needed to run a MapReduce job to create those indexes, nor to read the data a second time in any other way for the purpose of indexing. In fact, the HAIL upload pipeline is so effective when compared to HDFS that the additional overhead of sorting and index creation is hardly noticeable in the overall process. Therefore, we offer a win-win situation over Hadoop MapReduce and even over Hadoop++ [35]. We give an overview of HAIL and its benefits in Section 3.2.

2. **Per-Replica Indexing.** We show how to exploit the default replication of Hadoop to support different sort orders and indexes for each block replica (Section 3.3). For a default replication factor of three, up to three different sort orders and clustered indexes are available for processing MapReduce jobs. Thus, the likelihood to find a suitable index increases and the runtime for a workload improves. Our approach benefits from the fact that Hadoop is only used for appends: there are no updates. Thus, once a block is full, it will never be changed again.

3. **Job Execution.** We show how to effectively change the Hadoop MapReduce pipeline to exploit existing indexes (Section 3.4). Our goal is to do this without changing the code of the MapReduce framework. Therefore, we introduce optional annotations for MapReduce jobs that allow users to enrich their queries by explicitly specifying their selections and projections. HAIL takes care of performing MapReduce jobs using normal data block replicas or pseudo data block replicas (or even both).

4. **HAIL Scheduling.** We propose a new task scheduling, called *HAIL Scheduling*, to fully exploit statically and adaptively indexed data blocks (Section 3.7). The goal of HAIL Scheduling is twofold: (i) to reduce the scheduling overhead when executing a MapReduce job, and (ii) to balance the indexing effort across computing nodes to limit the impact of adaptive indexing.

5. **Zero-Overhead Adaptive Indexing.** We show how to effectively piggyback adaptive index creation on the existing MapReduce job execution pipeline (Section 3.5). The idea is to combine adaptive indexing and zero-overhead indexing to solve the problem of missing indexes for evolving or unpredictable workloads. In other words, when HAIL executes a map reduce job with a filter condition on an unindexed attribute, HAIL creates that missing index for a certain fraction of the HDFS blocks in parallel.

6. **Adaptive Indexing Strategies.** We propose a set of adaptive indexing strategies that makes HAIL aware of the performance and the selectivity of MapReduce jobs (Section 3.6). We present:

(a) *lazy* adaptive indexing, a technique that allows HAIL to adapt to changes in the users' workloads at a constant indexing overhead.

(b) *eager* adaptive indexing, a technique that allows HAIL to quickly adapt to changes in the users' workloads with a robust performance.

(c) We then show how HAIL can decide which data blocks to index based on the selectivities of MapReduce jobs.

7. **Exhaustive Validation.** We present an extensive experimental comparison of HAIL with Hadoop and Hadoop++ [35] (Section 3.9 and Section 3.10). We use seven different clusters including physical and virtual EC2 clusters of up to 100 nodes. A series of experiments shows the superiority of HAIL over both Hadoop and Hadoop++. Another series of scalability experiments with different datasets also demonstrates the superiority of using adaptive indexing in HAIL. In particular, our experimental results demonstrate that HAIL: (i) creates clustered indexes at upload time almost for free; (ii) quickly adapts to query workloads with a negligible indexing overhead; and (iii) only for the very first job does HAIL have a small overhead over Hadoop when creating indexes adaptively: all the following jobs are faster in HAIL.

2.1.3 Personal Contributions

Our work on HAIL initially started as my Master thesis and eventually became a larger team project that involved several members of our research group. As a member of this team developing HAIL I was involved in almost all aspects of the system from the start to the end of the project. I declare my personal contributions in Table 2.1.

2.1.4 Publications

- [96] Stefan Richter. HAIL: Hadoop Aggressive Indexing Library. Master's thesis, Saarland University, Germany, 2012
- [36] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.
- [99] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Adaptive Indexing in Hadoop. *CoRR*, abs/1212.3480, 2012.
- [100] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Static and Adaptive Indexing in Hadoop. *VLDB Journal*, 23(3):469–494, 2013.

- [37] Jens Dittrich, Stefan Richter, and Stefan Schuh. Efficient OR Hadoop: Why Not Both? *Datenbank-Spektrum*, 13(1):17–22, 2013.
- [98] Stefan Richter, Jens Dittrich, Stefan Schuh, and Tobias Frey. Mosquito: Another one bites the Data Upload STream. *PVLDB*, 6(12):1274–1277, 2013, demo paper, see Section A.
- *Patent:* Replicated data storage system and methods WO 2013139379 A1.

Contribution	Involvement	Details
Zero-Overhead Static Indexing.	major	I developed and implemented most of this contribution. SS and JQ were involved in discussions and supported me in coding and debugging.
Per-Replica Indexing.	major	see above.
Job Execution.	major	see above.
HAIL Scheduling	no	SS developed this together with JQ.
Zero-Overhead Adaptive Indexing.	major	I developed and implemented most of this contribution. SS and JQ were involved in discussions and supported me in coding and debugging.
Lazy Adaptive Indexing Strategy.	full	I developed this on my own as default strategy of the Adaptive Indexer.
Eager Adaptive Indexing Strategy.	minor	I was involved in the discussion of this idea with SS, but the implementation was done by him.
Selectivity-based Adaptive Indexing Strategy.	major	I developed this on my own and SS was involved in the discussion.
Exhaustive validation.	minor	I was involved in designing, writing, and running jobs for the experiments and particularly in the evaluation and interpretation of results. Most work for writing and running the jobs was done by SS, JQ, and JS.

Table 2.1: *Personal contributions to Chapter 3. Contributions by Stefan Schuh (SS), Jorge-Arnulfo Quiané-Ruiz (JQ), and Jörg Schad (JS) are mentioned in the "Details" column.*

Initial results of the HAIL project were published as my Master Thesis [96]. An extended version containing the HAIL Splitting strategy was published in PVLDB [36]. Afterwards, we introduced adaptive indexing in Hadoop and published the results in

a technical report [99]. A combined article including the description of the original HAIL system and the adaptive indexing of LIAH was then published in the VLDB Journal [100]. Chapter 3 of this thesis is an extended version of that journal article. I decided to include the full article, as it is very hard to understand my personal contributions for this thesis in isolation.

2.2 Chapter 4 : A Comparison of Adaptive Radix Trees and Hash Tables

2.2.1 Background

MapReduce started the wide and general adoption of data analytics on unstructured data at large scale in both research and industry. With this use-case in mind, Hadoop was clearly designed for long running queries in batch fashion and to address the problem of ever growing dataset sizes. Hadoop offered a solution that could handle many problems related to this dimension of dataset size. With this focus on the dimension of dataset size, Hadoop does not really offer a solution for applications that are difficult in other dimensions like query latency or throughput, and it became quickly clear that Hadoop MapReduce was not a solution that fits all problems [92, 105]. For example, OLTP (Online Transaction Processing) applications for businesses are typically not problematic in terms of pure dataset size as the amount of customers, products and orders per year obviously do not follow Moore's law. However, real-time transactions and analytics over data in the size of up to a few terabytes can bring huge benefits and enable new ways in which companies run their businesses. With prices for main memory constantly decreasing, it became possible to run database applications for most companies completely in main memory, and leave high I/O-costs that come along with traditional, disk-based approaches out of the equation. This combination of technical progress and customer demands triggered the development of a new generation of main memory databases, like MonetDB, H-Store/VoltDB [68], SAP HANA [43], Microsoft Hekaton [28], and HyPer [70]. A common design trait of this generation is heavy optimization for modern hardware features that mitigate the impact of the ever growing performance gap between the storage medium main memory and CPU, thus aiming for getting as close to becoming CPU-bound as possible.

With the paradigm shift from disk-based to memory-based systems there is also a shift in the relevant optimizations and their impact. For example, arguably the most important optimization in disk-based system is to avoid disk I/O (and random I/O in particular) by keeping hot data in main memory wherever possible. Interestingly, the essence of this idea can be transferred to main memory systems as well and is again the foundation to one of the most important optimizations. In this analogy, main memory databases try to keep hot data in the cache hierarchy, as close to the CPU as possible

and try to avoid accesses to the slower memory (again in particular random accesses) at all costs. While the basic idea of cache optimization already finds application in older disk based systems, it is now playing a more fundamental role in main memory where the latencies are orders of magnitude smaller. As a result, optimizations for main memory databases present new challenges to the way data should be stored and accessed in order to be processed efficiently. For example, such challenges occur when designing new tree index structures, which are crucial for the performance of database systems, on disk as well as in main memory. Tree-shaped index structures have already come a long way from binary search trees and their variants (e.g., AVL-tree, Red-Black-Tree, Splay-Tree). While binary search trees are interesting from a theoretical perspective, they have become less and less popular in practice over years. The main reason for this development is the ever-growing gap in performance between CPU and memory, which entailed increasing cache sizes and the addition of more and more layers to the memory hierarchy over time. As a result, the assumption of uniform memory access costs is invalid in practice for modern hardware architectures.

This observation triggered the development of index structures like the venerable B-tree and its variants, which were developed to improve cache and memory efficiency over binary search trees and are still widely used in many current database systems. However, B-trees were originally designed to accelerate disk-based systems, but they are no longer the way to go in main memory systems because, at least in their original form, their cache utilization is suboptimal. In addition to that, the algorithms on B-trees rely heavily on key comparisons, which are hard do predict for the hardware and therefore introduce significant amounts of branch mispredictions. Branch mispredictions, in turn, entail pipeline stalls which are expensive on modern CPUs that often feature long execution pipelines.

Because of the aforementioned shortcomings, the last decade has seen a considerable amount of research on tree-structured indexes for main memory systems [72, 102, 8, 15, 69, 85, 71, 76]. Among the most recent and most interesting data structures for main memory systems there is the recently-proposed adaptive radix tree ARTful [76] (ART for short). ART was integrated into the state-of-the-art main memory database HyPer [70] and, for example, reported to boost the performance of OLTP by a factor of two over previous approaches. Furthermore, the authors of ART presented experiments that indicate that ART was clearly a better choice over other recent tree-based data structures like FAST [71] and CSB$^+$-trees [95]. However, ART was not the first adaptive radix tree. To the best of our knowledge, the first was Judy Array (Judy for short), and a comparison between ART and Judy was not shown. This raised the question for us, whether a performance that is comparable to ART was already available for years through Judy, a data structure that is more than a decade old. Moreover, the same set of experiments indicated that *only* a hash table was competitive to ART. The hash table used by the authors of ART in their study was a chained hash table, but this kind of

hash tables can be suboptimal in terms of space and performance due to their potentially high use of pointers.

2.2.2 Contributions

In Chapter 4 we *extend* the discussion of related work and the experimental comparison offered by the authors of ART in the original paper. This includes the following major contributions:

1. **Conceptual comparison of ART and Judy Array.** We provide an in-depth comparison for both, concepts and implementation of ART and Judy Array, which is missing in the original paper [76] on ART. We show striking similarities between both structures, although the design of Judy Array is more than a decade older than ART. As Judy Array is a patented technology, this information is valuable to (re-)assess the novelty of ART and the possible risks of integrating ART into commercial databases.

2. **Experimental comparison of ART and Judy Array.** Similar to the conceptional comparison, the authors of ART also did not include Judy Array in their experimental comparison of ART with other index structures. We close this gap and provide a detailed comparison between ART and Judy Array w.r.t. performance and memory consumption.

3. **Experimental comparison of ART and well-engineered hashing schemes.** In the original paper, the authors found that the performance of ART is competitive to hash tables. However, they only compared against a textbook implementation of chained hashing with Murmur [7] as hash function. This combination is arguably a suboptimal representative for hash tables in their benchmark. We investigate how the picture changes when we use more elaborate, well-engineered hashing tables, such as Google's sparse and dense hash tables [48] and our own implementations of Cuckoo hashing, as well as different hash functions. We simulate two different types of workloads in our experiments, OLAP (bulk inserts and bulk lookup) and OLTP (a mix of inserts, deletes and lookups) and provide a micro-architectural analysis of our results using CPU performance counters.

4. **Evaluation of range query performance.** In the original work on ART, the authors explain that efficient support for range queries is one big advantage of tree structured indexes over hash tables. However, they do not provide an experimental evaluation of the range query performance of ART. In fact, range query support was not implemented in the published ART code. We implemented this missing feature and compared the range query performance of ART, e.g. against Judy Array and a B^+-tree.

5. **Evaluation for covering and non-covering indexing scenarios.** In our experiment, we differentiate two relevant use-cases for our indexes. The first use-case is *covering* index, where both keys and values are completely stored inside the index structure. Here, the index is used as a stand-alone data structure. The second use-case is *non-covering* index, where the index itself does not contain keys and values. Instead a non-covering index operates on top of a primary store that contains the keys and values, e.g. a database table.

2.2.3 Publications

[6] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. A Comparison of Adaptive Radix Trees and Hash Tables. In *31st IEEE ICDE*, April 2015

2.3 Chapter 5 : A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing

2.3.1 Background

Our work in Chapter 5 addresses a problem that we encountered during our previous studies from Chapter 4. Many researchers and practitioners are tempted to consider hashing a solved problem and there is noticeably less work published on hash based indexing compared to, e.g. tree indexes in recent database research. In the database community, hashing is often considered as a simple way to achieve lookups in constant time, where it is safe to use an arbitrary method as a black box and expect good performance. Moreover, it seems like we expect that optimizations to hashing can only improve it by a negligible delta. For example, textbook chained hashing is still used as the default implementation in the code bases of many research prototypes, database implementations, and the standard libraries of popular programming languages. Chained hashing became popular already in the 50s as, perhaps, the very first iteration of hashing — robust and easy to implement. However, nowadays, plain chained hashing is often a suboptimal choice in practice w.r.t. performance and memory efficiency for many use-cases on modern hardware due to its use of pointers. We found it remarkable, that relatively little attention is given to a technique like hashing, that is ubiquitous in databases, e.g. as hash index structure and as the backbone of many algorithms like hash-joins, deduplication, and hash-based aggregation. Rather recently, there has been a considerable amount of interesting theoretical research on new hashing algorithms, e.g. on Cuckoo hashing [90], as well as interesting reconsiderations about the properties of well-established methods like linear probing in combination with certain hash

functions [93]. Those lines of research with promising theoretical results made us curious, whether or not the proposed techniques could also make an impact in practice on real hardware where constant factors are sometimes more important for performance than asymptotic behavior, and if there is application for those ideas in databases. In the end, we were interested how different hash table implementations would compare to each other in this context, what tradeoffs different approaches could provide, how much this choice would affect performance, and how the theoretical results will match practical observations. On top of that, between the lines in recent work, we found evidence that hashing performance in databases is severely impacted by more factors than just the hash table. For example, a recent work on hash joins [75] reported that just using a different hash function with lower computational complexity already improved the end-to-end runtime of their hash join by 36%. This observation made us wonder, what dimensions impact the performance of hashing in databases and what the relationship and impact of those dimensions are in a modern system. In addition to hash table organization and hash function, we identified that five further dimensions have an important influence on hashing performance. To put things in perspective, we carefully study this, in total, seven-dimensional parameter space and we also offer a glimpse about the effect of different memory layouts and the use of SIMD instructions. Our main goal is to produce enough results that can guide practitioners, and potentially the query optimizer, towards choosing the most appropriate hash table for their use-case at hand.

In this study, we decided to focus on hash tables in a single-threaded context to isolate the impact of the aforementioned dimensions. We believe that a thorough evaluation of concurrency in hash tables is a research topic that deserves a complete study on its own. However, our observations still play an important role for hash maps in multi-threaded algorithms. For partitioning-based parallelism — which has recently been considered in the context of (partition-based hash) joins [10, 11, 75] — single-threaded performance is still a key parameter: each partition can be considered an isolated unit of work that is only accessed by exactly one thread at a time, and therefore concurrency control inside the hash tables is not needed. Furthermore, all hash tables we present in the paper can be extended for thread safety through well-known techniques such as striped locking or compare-and-swap. Here, the dimensions we discuss still impact the performance of the underlying hash table.

We focus on 64-bit integer keys and values, which is arguably the most important mapping in modern and future main memory databases for large datasets. In particular, main memory databases typically make heavy use of dictionary compression to substitute complex or variable-length datatypes (e.g. strings) with integer values.

2.3.2 Contributions

In Chapter 5 we make the following contributions:

1. **Overview on hashing schemes and hash functions.** We give an overview on five of the most important hashing schemes (including linear probing, quadratic probing, Robin Hood hashing [21], Cuckoo hashing [90], and chained hashing) and four hash functions (Multiply-shift [30], Multiply-add-shift [29], Tabulation hashing [114, 110, 93], and Murmur hashing [7]) that we considered for our study.

2. **Identification of five additional dimensions and evaluation of their impact on hashing performance.** Given a hashing scheme and a hash function, we identify five additional dimensions that can significantly impact the performance of hash tables (key distribution, load factor, hash table size, read/write ratio, un/successful lookup ratio). Together with hashing schemes and hash functions, we consider in total a seven dimensional space w.r.t. the question *which hash table performs best in which situation and why?*

3. **Experiments for static workload.** We present the essential results of our experiments that explore the seven dimensional parameter space with respect to static workload (think of OLAP). In this scenario, hash tables are loaded once with a fixed and previously known amount of data and after that lookups are performed. In particular, in this setting, hash tables never have to resize.

4. **Experiments for dynamic workload.** Similar to the static workload, we also investigate dynamic workload (think of OLTP). Here, the hash tables are preloaded with a certain amount of data. Afterwards, a mixed workload of inserts, deletes, lookups, and updates is executed. In this setting we also observe how the hash tables handle resizing (rehash operation).

5. **Experiments on hash table memory layout and vectorization.** We also present experiments that explore the impact of different memory layouts (Array-of-Structs vs Struct-of-Arrays) by the example of a linear probing hash table. Furthermore, we also investigate the effects of vectorization (SIMD), e.g. comparing multiple keys in one instruction.

6. **Lessons learned and decision diagram for practitioners.** We report our lessons learned and condense the results of all experiments in a flow diagram that can guide practitioners towards the right hash table configuration for their use-case at hand.

2.3.3 Publications

[97] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *PVLDB*, 9(3):96–107, 2015

Chapter 3

Towards Zero-Overhead Static and Adaptive Indexing in Hadoop

3.1 Introduction

MapReduce has become the de facto standard for large scale data processing in many enterprises. It is used for developing novel solutions on massive datasets such as web analytics, relational data analytics, machine learning, data mining, and real-time analytics [52]. In particular, log processing emerges as an important type of data analysis commonly done with MapReduce [14, 82, 41].

In fact, Facebook and Twitter use Hadoop MapReduce (the most popular MapReduce open source implementation) to analyze the huge amounts of web logs generated every day by their users [107, 51, 79]. Over the last years, a lot of research works have focused on improving the performance of Hadoop MapReduce [24, 56, 63, 66]. When improving the performance of MapReduce, it is important to consider that it was initially developed for large aggregation tasks that scan through huge amounts of data. However, nowadays Hadoop is often also used for selective queries that aim to find only a few relevant records for further consideration[1]. For selective queries, Hadoop still scans through the complete dataset. This resembles the search for a needle in a haystack.

For this reason, several researchers have particularly focused on supporting efficient index access in Hadoop [112, 35, 79, 64]. Some of these works have improved the performance of selective MapReduce jobs by orders of magnitude. However, all these indexing approaches have three main weaknesses. First, they require a high upfront cost for index creation. This translates to long waiting times for users until they can actually start to run queries. Second, they can only support one physical sort order (and hence one clustered index) per dataset. This becomes a serious problem if the work-

[1] A simple example of such a use-case would be a distributed grep.

load demands indexes for several attributes. Third, they require users to have a good knowledge of the workload in order to choose the indexes to create. This is not always possible, e.g., if the data is analyzed in an exploratory way, or queries are submitted by customers.

3.1.1 Motivation

Let us see through the eyes of a data analyst, say Bob, who wants to analyze a large web log. The web log contains different fields that may serve as filter conditions for Bob like visitDate, adRevenue, sourceIP, and so on. Assume Bob is interested in all sourceIPs with a visitDate from 2011. Thus, Bob writes a MapReduce program to filter out exactly those records and discard all others. Bob is using Hadoop, which will scan the entire input dataset from disk to filter out the qualifying records. This takes a while. After inspecting the result set, Bob detects a series of strange requests from sourceIP 134.96.223.160. Therefore, he decides to modify his MapReduce job to show all requests from the entire input dataset having that sourceIP. Bob is using Hadoop. This takes a while. Eventually, Bob decides to modify his MapReduce job again to only return log records having a particular adRevenue. Yes, this again takes a while.

In summary, Bob uses a sequence of different filter conditions, each one triggering a new MapReduce job. He is not exactly sure what he is looking for. The whole endeavor feels like going shopping without a shopping list. This example illustrates an exploratory usage (and a major use-case) of Hadoop MapReduce [14, 41, 88]. But, this use-case has one major problem: *slow query runtimes.* The time to execute a MapReduce job based on a scan may be very high: it is dominated by the I/O for reading all input data [91, 64]. While waiting for his MapReduce job to complete, Bob has enough time to pick a coffee (or two) and this happens every time Bob modifies the MapReduce job. This will likely kill his productivity and make his boss unhappy.

Now, assume the fortunate case that Bob remembers a sentence from one of his professors saying "*full-table-scans are bad; indexes are good*"[2]. Thus, he reads all the recent VLDB papers (including [64, 24, 56, 63]) and finds a paper that shows how to create a so-called *trojan index* [35]. A trojan index is an index that may be used with Hadoop MapReduce and yet does not modify the underlying Hadoop MapReduce and HDFS engines.

Zero-Overhead indexing

Bob finds the trojan index idea interesting and hence decides to create a trojan index on sourceIP before running his MapReduce jobs. However, using trojan indexes raises two other problems:

[2]Nowadays, Bob is already aware that for some situations, the opposite is true.

1. *Expensive index creation.* The time to create the trojan index on sourceIP (or any other attribute) is even much longer than running a scan-based MapReduce job. Thus, if Bob's MapReduce jobs use that index only a few times, the index creation costs will never be amortized. So, why would Bob create such an expensive index in the first place?

2. *Which attribute to index?* Even if Bob amortizes index creation costs, the trojan index on sourceIP will only help for that particular attribute. So, which attribute should Bob use to create the index?

Bob is wondering how to create several indexes at very low cost to solve those problems.

Per-Replica indexing

One day in autumn 2011, Bob reads about another idea [66] where some researchers looked at ways to improve vertical partitioning in Hadoop. The researchers in that work realized that HDFS keeps three (or more) physical copies of all data for fault-tolerance. Therefore, they decided to change HDFS to store each physical copy in a *different* data layout (row, column, PAX, or any other column grouping layout). As all data layout transformation is done per HDFS data block, the failover properties of HDFS and Hadoop MapReduce were not affected. At the same time, I/O-times improved. Bob thinks that this looks very promising, because he could possibly exploit this concept to create different clustered indexes almost invisible to the user. This is because he could create one clustered index per data block replica when uploading data to HDFS. This would already help him a lot in several query workloads.

However, Bob quickly figures out that there are cases where this idea still has some annoying limitations. Even if Bob could create one clustered index per data replica at low cost, he would still have to determine which attributes to index when uploading his data to HDFS. Afterward, he could not easily revise his decision or introduce additional indexes without uploading the dataset again. Unfortunately, it sometimes happens that Bob and his colleagues navigate through datasets according to the properties and correlations of the data. In such cases, Bob and his colleagues typically: (**1.**) do not know the data access patterns in advance; (**2.**) have different interests and hence cannot agree upon common selection criteria at data upload time; (**3.**) even if they agree which attributes to index at data upload time, they might end up filtering records according to values on different attributes. Therefore, using any traditional indexing technique [44, 22, 3, 19, 23, 112, 79, 35, 64] would be problematic, because they cannot adapt well to unknown or changing query workloads.

Adaptive indexing

When searching for a solution to his problem with static indexing, Bob stumbles across a new approach called *adaptive indexing* [58], where the general idea is to create indexes as a side effect of query processing. This is similar to the idea of *soft indexes* [84], where the system piggybacks the index creation for a given attribute on a single incoming query. However, in contrast to soft indexes, adaptive indexing aims at creating indexes incrementally (i.e., piggybacking on several incoming queries) in order to avoid high upfront index creation times. Thus, Bob is excited about the adaptive indexing idea since this could be the missing piece to solve his remaining concern. However, Bob quickly notices that he cannot simply apply existing adaptive indexing works [38, 58, 59, 50, 61, 53] in MapReduce systems for several reasons:

1. *Global index convergence.* These techniques aim at converging to a global index for an entire attribute, which requires sorting the attribute globally. Therefore, these techniques perform many data movements across the entire dataset. Doing this in MapReduce would hurt fault-tolerance as well as the performance of MapReduce jobs. This is because the system would have to move data across data blocks in sync with all their three physical data block replicas. We do not plan to create global indexes, but focus on creating partial indexes that in total cover the whole dataset. A small back of the envelope calculation shows that the possible gains of a global index are negligible in comparison with the overhead of the MapReduce framework. For instance, if a dataset is uniformly distributed over a cluster and occupies 160 HDFS blocks on each datanode (like the dataset in our experiments in Section 3.9) and we do not have a global index, then we need to perform 160 index accesses on each datanode. Since all datanodes can access their blocks in parallel to each other, we assume that the overhead is determined by the highest overhead per datanode. Overall, our approach requires at most 318 additional random reads in HDFS per datanode in this scenario, which in turn cost roughly 15ms each. In total, this amounts to 4.77s overhead compared to a global index stored in HDFS. However, even empty MapReduce jobs, which do not read any data nor compute a single map function, run for more than 10s.

2. *High I/O-costs.* Even if Bob applied existing adaptive indexing techniques inside data blocks, these techniques would end up in many costly I/O operations to move data on disk. This is because these techniques consider main memory systems and thus do not factor in the I/O-cost for reading/writing data from/to disk. Only one of these works [50] proposes an adaptive merging technique for disk-based systems. However, applying this technique inside a HDFS block would not make sense in MapReduce since HDFS blocks are typically loaded entirely into main memory anyways when processing map tasks. One may think about applying

adaptive merging across HDFS blocks, but this would again hurt fault-tolerance and the performance of MapReduce jobs as described above.

3. *Unclustered index.* These works focus on creating unclustered indexes in the first place, and hence, it is only beneficial for highly selective queries. One of these works [59] introduced lazy tuple reorganization in order to converge to clustered indexes. However, this technique needs several thousand queries to converge, and its application in a disk-based system would again introduce a huge number of expensive I/O operations.

4. *Centralized approach.* Existing adaptive indexing approaches were mainly designed for single-node DBMSs. Therefore, applying these works in a distributed parallel systems, like Hadoop MapReduce, would not fully exploit the existing parallelism to distribute the indexing effort across several computing nodes.

Despite all these open problems, Bob is very enthusiastic to combine the above interesting ideas on indexing into a new system to revolutionize the way his company can use Hadoop. And this is where the story begins.

3.1.2 Research Questions and Challenges

This article addresses the following research questions:

1. *Zero-overhead indexing.* Current indexing approaches in Hadoop involve a significant upfront cost for index creation. How can we make indexing in Hadoop so effective that it is basically invisible for the user? How can we minimize the I/O costs for indexing or eventually reduce them to zero? How can we fully utilize the available CPU resources and parallelism of large clusters for indexing?

2. *Per-replica indexing.* Hadoop uses data replication for failover. How can we exploit this replication to support different sort orders and indexes? Which changes to the HDFS upload pipeline need to be done to make this efficient? What happens to the involved checksum mechanism of HDFS? How can we teach the HDFS namenode to distinguish the different replicas and keep track of the different indexes?

3. *Job execution.* How can we change Hadoop MapReduce to utilize different sort orders and indexes at query time? How can we change Hadoop MapReduce to schedule tasks to replicas having the appropriate index? How can we schedule map tasks to efficiently process indexed and nonindexed data blocks without affecting failover? How much do we need to change existing MapReduce jobs? How will Hadoop MapReduce change from the user's perspective?

4. *Zero-overhead adaptive indexing.* How can we adaptively and automatically create additional useful indexes online at minimal costs per job? How to index big data incrementally in a distributed, disk-based system like Hadoop as byproduct of job execution? How to minimize the impact of indexing on individual job execution times? How to efficiently interleave data processing with indexing? How to distribute the indexing effort efficiently by considering data locality and index placement across computing nodes? How to create several clustered indexes at query time? How to support a different number of replicas per data block?

3.1.3 Contributions

We propose HAIL (*Hadoop Adaptive Indexing Library*), a static and adaptive indexing approach for MapReduce systems. The main goal of HAIL is to minimize both (i) the index creation time when uploading data and (ii) the impact of concurrent index creation on job execution times. In summary, we make the following main contributions to tackle the questions and challenges mentioned above:

1. *Zero-overhead indexing.* We show how to effectively piggyback sorting and index creation on the existing HDFS upload pipeline. This way it is not needed to run a MapReduce job to create those indexes, nor to read the data a second time in any other way for the purpose of indexing. In fact, the HAIL upload pipeline is so effective when compared to HDFS that the additional overhead of sorting and index creation is hardly noticeable in the overall process. Therefore, we offer a win-win situation over Hadoop MapReduce and even over Hadoop++ [35]. We give an overview of HAIL and its benefits in Section 3.2.

2. *Per-replica indexing.* We show how to exploit the default replication of Hadoop to support different sort orders and indexes for each block replica (Section 3.3). For a default replication factor of three, up to three different sort orders and clustered indexes are available for processing MapReduce jobs. Thus, the likelihood to find a suitable index increases and the runtime for a workload improves. Our approach benefits from the fact that Hadoop is only used for appends: there are no updates. Thus, once a block is full, it will never be changed again.

3. *Job execution.* We show how to effectively change the Hadoop MapReduce pipeline to exploit existing indexes (Section 3.4). Our goal is to do this without changing the code of the MapReduce framework. Therefore, we introduce optional annotations for MapReduce jobs that allow users to enrich their queries by explicitly specifying their selections and projections. HAIL takes care of performing MapReduce jobs using normal data block replicas or pseudo data block replicas (or even both).

4. *HAIL scheduling.* We propose a new task scheduling, called *HAIL Scheduling*, to fully exploit statically and adaptively indexed data blocks (Section 3.7). The goal of HAIL Scheduling is twofold: (i) to reduce the scheduling overhead when executing a MapReduce job, and (ii) to balance the indexing effort across computing nodes to limit the impact of adaptive indexing.

5. *Zero-Overhead adaptive indexing.* We show how to effectively piggyback adaptive index creation on the existing MapReduce job execution pipeline (Section 3.5). The idea is to combine adaptive indexing and zero-overhead indexing to solve the problem of missing indexes for evolving or unpredictable workloads. In other words, when HAIL executes a map reduce job with a filter condition on an unindexed attribute, HAIL creates that missing index for a certain fraction of the HDFS blocks in parallel.

6. *Adaptive indexing strategies.* We propose a set of adaptive indexing strategies that makes HAIL aware of the performance and the selectivity of MapReduce jobs (Section 3.6). We present:

   (a) *lazy* adaptive indexing, a technique that allows HAIL to adapt to changes in the users' workloads at a constant indexing overhead.

   (b) *eager* adaptive indexing, a technique that allows HAIL to quickly adapt to changes in the users' workloads with a robust performance.

   (c) We then show how HAIL can decide which data blocks to index based on the selectivities of MapReduce jobs.

7. *Exhaustive validation.* We present an extensive experimental comparison of HAIL with Hadoop and Hadoop++ [35] (Section 3.9 and Section 3.10). We use seven different clusters including physical and virtual EC2 clusters of up to 100 nodes. A series of experiments shows the superiority of HAIL over both Hadoop and Hadoop++. Another series of scalability experiments with different datasets also demonstrates the superiority of using adaptive indexing in HAIL. In particular, our experimental results demonstrate that HAIL: (i) creates clustered indexes at upload time almost for free; (ii) quickly adapts to query workloads with a negligible indexing overhead; and (iii) only for the very first job does HAIL have a small overhead over Hadoop when creating indexes adaptively: all the following jobs are faster in HAIL.

3.2 Overview

In the following, we give an overview of HAIL by contrasting it with normal HDFS and Hadoop MapReduce. Thereby, we introduce the two indexing pipelines of HAIL.

First, *static indexing* allows us to create several clustered indexes at upload time. Second, *HAIL adaptive indexing* creates additional indexes as a byproduct of actual job execution, which enables HAIL to adapt to unexpected workloads. For a more detailed contrast to related work, see Section 3.8.

For now, let's consider again our motivating example: *How can Bob analyze his log file with Hadoop and HAIL?*

3.2.1 HDFS and Hadoop MapReduce

In HDFS and Hadoop MapReduce, Bob starts by uploading his log file to HDFS using the *HDFS client*. HDFS then partitions the file into logical *HDFS blocks* using a constant block size (the HDFS default is 64MB). Each HDFS block is then physically stored three times (assuming the default replication factor). Each physical copy of a block is called a *replica*. Each replica will sit on a different *datanode*. Therefore, at least two datanode failures may be survived by HDFS. Note that HDFS keeps information on the different replicas for an HDFS block in a central *namenode* directory.

After uploading his log file to HDFS, Bob may run an actual MapReduce job. Bob invokes Hadoop MapReduce through a Hadoop MapReduce *JobClient*, which sends his *MapReduce job* to a central node termed *JobTracker*. The MapReduce job consists of several *tasks*. A task is executed on a subset of the input file, typically an HDFS block[3]. The JobTracker assigns each task to a different *TaskTracker*, which typically runs on the same machine as an HDFS datanode. Each datanode will then read its subset of the input file, i.e., a set of HDFS blocks, and feed that data into the *MapReduce processing pipeline* which usually consists of a Map, Shuffle, and a Reduce Phase (see [27, 35, 34] for a detailed description). As soon as all results have been written to HDFS, the JobClient informs Bob that the results are available. Notice that, the execution time of the MapReduce job is heavily influenced by the size of the input dataset, because Hadoop MapReduce reads the input dataset entirely in order to perform any incoming MapReduce job.

3.2.2 HAIL

In HAIL, Bob analyzes his log file as follows. He starts by uploading his log file to HAIL using the *HAIL client*. In contrast to the HDFS client, the HAIL client analyzes the input data for each HDFS block, converts each HDFS block directly to a binary columnar layout, which resembles PAX [4] and sends it to three datanodes. Then, all datanodes sort the data contained in that HDFS block in parallel using a different sort order. The required sort orders can be manually specified by Bob in a configuration file or computed by a physical design algorithm. For each HDFS block, all sorting

[3] Actually it is a *split*. The difference does not matter here. We will get back to this in Section 3.4.2.

and index creation happens in main memory. This is feasible as the HDFS block size is typically between 64MB (default) and 1GB. This easily fits in the main memory of most machines. In addition, in HAIL, each datanode creates a different clustered index for each HDFS block replica and stores it with the sorted data. This process is called the *HAIL static indexing* pipeline.

After uploading his log file to HAIL, Bob runs his MapReduce jobs that can now immediately exploit the indexes that were created by HAIL statically (i.e., at upload time). As before, Bob invokes Hadoop MapReduce through a JobClient, which sends his MapReduce jobs to the JobTracker. However, his MapReduce jobs are slightly modified so that the system can decide to eventually use available indexes on the data block replicas. For example, assume that a data block has three replicas with clustered indexes on visitDate, adRevenue, and sourceIP. In case that Bob has a MapReduce job filtering on visitDate, HAIL uses the replicas having the clustered index on visitDate. If Bob is filtering on sourceIP, HAIL uses the replicas having the clustered index on sourceIP and so on. To provide failover and load balancing, HAIL may fall back to standard Hadoop scanning for some of the blocks. However, even factoring this in, Bob's queries run much faster on average, if indexes on the right attributes exist.

In case that Bob submits jobs that filter on unindexed attributes (e.g., on duration), HAIL again falls back to a standard full scan by choosing any arbitrary replica, just like Hadoop. However, in contrast to Hadoop, HAIL can index HDFS blocks in parallel to job execution. If another job filters again on the duration field, the new job can already benefit from the previously indexed blocks. So, HAIL takes incoming jobs, which have a selection predicate on currently unindexed attributes, as hints for valuable additional clustered indexes. Consequently, the set of available indexes in HAIL evolves with changing workloads. We call this process the *HAIL adaptive indexing* pipeline.

3.2.3 HAIL Benefits

1. HAIL often improves both upload *and* query times. The upload is dramatically faster than Hadoop++ and often faster (or only slightly slower) than with the standard Hadoop even though we (i) convert the input file into binary PAX, (ii) create a series of different sort orders, and (iii) create multiple clustered indexes. From the user-side, this provides a win-win situation: there is no noticeable punishment for upload. For querying, users can only win: if our indexes cannot help, we will fall back to standard Hadoop scanning; if the indexes can help, query runtimes will improve.

   Why do we not have high costs at upload time? We basically exploit the unused CPU ticks that are not used by standard HDFS. As the standard HDFS upload pipeline is I/O-bound, the effort for our sorting and index creation in the HAIL upload pipeline is hardly noticeable. In addition, since we parse data to binary

while uploading, we often benefit from smaller datasets triggering less network and disk I/O.

2. Even if we did not create the right indexes at upload time, HAIL can create indexes adaptively at job execution time without incurring high overhead.

   Why don't we see a high overhead? We do not need to additionally load the block data to main memory, since we piggyback on the reading of the map tasks. Furthermore, HAIL creates indexes incrementally over several job executions using different adaptive indexing strategies.

3. We do not change the failover properties of Hadoop.

   Why is failover not affected? All data stays on the same *logical* HDFS block. We just change the *physical* representation of each replica of an HDFS block. Therefore, from each physical replica, we may recover the logical HDFS block.

4. HAIL works with existing MapReduce jobs incurring only minimal changes to those jobs.

   Why does this work? We allow Bob to annotate his existing jobs with selections and projections. Those annotations are then considered by HAIL to pick the right index. Like that, for Bob, the changes to his MapReduce jobs are minimal.

3.3 HAIL Zero-Overhead Static Indexing

We create static indexes in HAIL while uploading data. One of the main challenges is to support different sort orders and clustered indexes per-replica as well as to build those indexes efficiently without much impact on upload times. Figure 3.1 shows the data flow when Bob uploads a file to HAIL. Let's first explore the details of the static indexing pipeline.

3.3.1 Data Layout

In HDFS, for each block, the client contacts the namenode to obtain the list of datanodes that should store the block replicas. Then, the client sends the original block to the first datanode, which forwards this to the second datanode and so on. In the end, each datanode stores a byte-identical copy of the original block data.
In HAIL, the HAIL client preprocesses the file based on its content to consider end of lines ① in Figure 3.1. We parse the contents into rows by searching for end of line symbols and never split a row between two blocks. This is in contrast to standard HDFS which splits a file into HDFS blocks after a constant number of bytes. For each

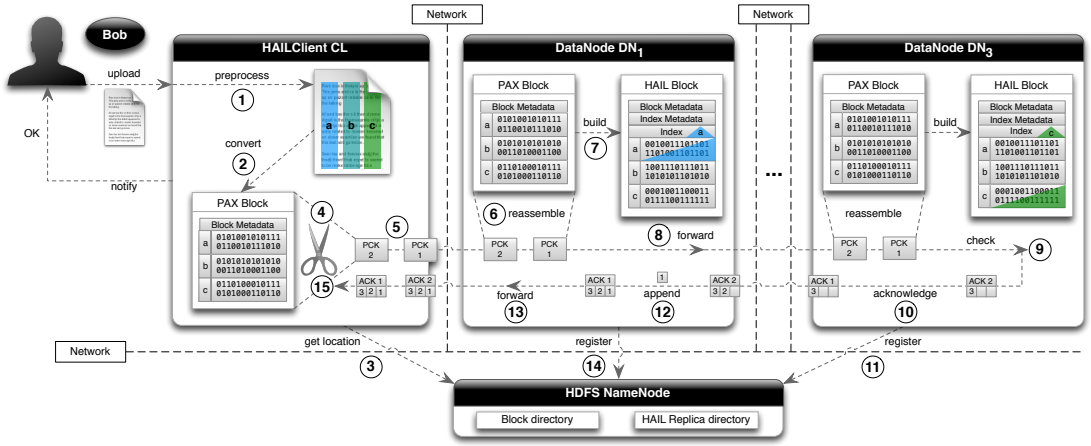


Figure 3.1: *The HAIL static indexing pipeline as part of uploading data to HDFS*

block, the HAIL client parses each row according to the schema specified by the user[4]. If HAIL encounters a row that does not match the given schema (i.e., a bad record), it separates this record into a special part of the data block. HAIL then converts all HDFS blocks to a binary columnar layout that resembles PAX (2). This allows us to index and access individual attributes more efficiently. The HAIL client also collects metadata information from each HDFS block (such as the data schema) and creates a block header (*Block Metadata*) for each HDFS block (2).

We could naively piggyback on this existing HDFS upload pipeline by first storing the original block data as done in Hadoop and then converting it to binary PAX layout in a second step. However, we would have to re-read and then re-write each block, which would trigger one extra write and read *for each replica*, e.g., for an input file of a 100GB we would have to pay 600GB extra I/O on the cluster. This would lead to very long upload times. In contrast, HAIL does not have to pay any of that extra I/O. However, to achieve this dramatic improvement, we have to make nontrivial changes in the standard Hadoop upload pipeline.

3.3.2 Static Indexing in the Upload Pipeline

To understand the implementation of static indexing in the HAIL upload pipeline, we first have to analyze the normal HDFS upload pipeline in more detail.

In HDFS, while uploading a block, the data is further partitioned into *chunks* of constant size 512B. Chunks are collected into *packets*. A packet is a sequence of chunks plus a checksum for each of the chunks. In addition, some metadata is kept. In total, a packet has a size of up to 64KB. Immediately before sending the data over the network, each

[4] Alternatively, HAIL can also suggest an appropriate schema to users through schema analysis.

HDFS block is converted to a sequence of packets. On disk, HDFS keeps, for each replica, a separate file containing checksums for all of its chunks. Hence, for each replica, two files are created on local disk: one file with the actual data and one file with its checksums. These checksums are reused by HDFS whenever data is send over the network. The HDFS client (CL) sends the first packet of the block to the first datanode (DN_1) in the upload pipeline. DN_1 splits the packet into two parts: the first contains the actual chunk data and the second contains the checksums for those chunks. Then, DN_1 flushes the chunk data to a file on local disk. The checksums are flushed to an extra file. In parallel, DN_1 forwards the packet to DN_2 which splits and flushes the data like DN_1 and in turn forwards the packet to DN_3 which splits and flushes the data as well. Yet, only DN_3 verifies the checksum for each chunk. If the recomputed checksums for each chunk of a packet match the received checksums, DN_3 acknowledges the packet back to DN_2, which acknowledges back to DN_1. Finally, DN_1 acknowledges back to CL. Each datanode also appends its ID to the ACK. Like that only one of the datanodes (the last in the chain, here DN_3 as the replication factor is three) has to verify the checksums. DN_2 believes DN_3, DN_1 believes DN_2, and CL believes DN_1. If any CL or DN_i receives ACKs in the wrong order, the upload is considered failed. The idea of sending multiple packets from CL is to hide the roundtrip latencies of the individual packets. Creating this chain of ACKs also has the benefit that CL only receives a single ACK for each packet and not three. Notice that, HDFS provides this checksum mechanism on top of the existing TCP/IP checksum mechanism (which has weaker correctness guarantees than HDFS).

In HAIL, in order to reuse as much of the existing HDFS pipeline and yet to make this efficient, we need to perform the following changes. As before, the HAIL client (CL) gets the list of datanodes to use for this block from the HDFS namenode ③. But rather than sending the original input, CL creates the PAX block, cuts it into packets ④, and sends it to DN_1 ⑤. Whenever a datanode DN_1–DN_3 receives a packet, it does *neither* flush its data *nor* its checksums to disk. Still, DN_1 and DN_2 immediately forward the packet to the next datanode as before⑧. DN_3 will verify the checksum of the chunks for the received PAX block ⑨ and acknowledge the packet back to DN_2 ⑩. This means the semantics of an ACK for a packet of a block are changed from "packet received, validated, and flushed" to "packet received and validated". We do neither flush the chunks nor its checksums to disk as we first have to sort the entire block according to the desired sort key. On each datanode, we assemble the block from all packets in main memory ⑥. This is realistic in practice, since main memories tend to be >10GB for any modern server. Typically, the size of a block is between 64MB (default) and 1GB. This means that for the default size, we could keep about 150 blocks in main memory at the same time.

In parallel to forwarding and reassembling packets, each datanode sorts the data, creates indexes, and forms a *HAIL Block*⑦, (see Section 3.3.4). As part of this process,

each datanode also adds *Index Metadata* information to each HAIL block in order to specify the index it created for this block. Each datanode (e.g., DN_1) typically sorts the data inside a block in a different sort order. It is worth noting that having different sort orders across replicas does not impact fault-tolerance as all data is reorganized *inside* the same block only, i.e., data is *not* reorganized *across* blocks. Hence, all replicas of the same HDFS block logically contain the same records with just a different order and therefore can still act as logical replacements for each other. Additionally, this property helps HAIL to preserve the load balancing capabilities of Hadoop. For example, when a datanode containing the replica with matching sort order for a certain job is overloaded, HAIL might choose to read from a different replica on another datanode, just like normal Hadoop. To avoid overloading datanodes in the first place, HAIL employs a round robin strategy for assigning sort orders to physical replicas on top of the replica placement of HDFS. This means that while HDFS already cares about distributing HDFS block replicas across the cluster, HAIL cares about distributing the sort orders (and hence the indexes) across those replicas.

As soon as a datanode has completed sorting and creating its index, it will recompute checksums for each chunk of a block. Notice that, checksums will differ on each replica, as different sort orders and indexes are used. Hence, each datanode has to compute its own checksums. Then, each datanode flushes the chunks and newly computed checksums to two separate files on local disk as before. For DN_3, once all chunks and checksums have been flushed to disk, DN_3 will acknowledge the last packet of the block back to DN_2 (10). After that, DN_3 will inform the HDFS namenode about its new replica including its HAIL block size, the created indexes, and the sort order (11) (see Section 3.3.3). Datanodes DN_2 and DN_1 append their ID to each ACK (12). Then, they forward each ACK back in the chain (13). DN_2 and DN_1 will forward the last ACK of the block only if all chunks and checksums have been flushed to their disks. After that DN_2 and DN_1 individually inform the HDFS namenode (14). The HAIL client also verifies that all ACKs arrive in order (15).

Notice that, it is important to change the HDFS namenode in order to keep track of the different sort orders. We discuss these changes in Section 3.3.3.

3.3.3 HDFS Namenode Extensions

In HDFS, the central namenode keeps a directory Dir_block of blocks, i.e., a mapping blockID $\mapsto$ Set Of DataNodes. This directory is required by any operation retrieving blocks from HDFS. Hadoop MapReduce exploits Dir_block for scheduling. In Hadoop MapReduce whenever a split needs to be assigned to a worker in the map phase, the scheduler looks up Dir_block in the HDFS namenode to retrieve the list of datanodes having a replica of the contained HDFS block. Then, the Hadoop MapReduce scheduler will try to schedule map tasks on those datanodes if possible. Unfortunately, the HDFS namenode does not differentiate the replicas w.r.t. their physical layouts. HDFS was

simply not designed for this. Thus, from the point of view of the namenode, all replicas are byte-equivalent and have the same size.

In HAIL, we need to allow Hadoop MapReduce to change the scheduling process to schedule map tasks close to replicas having a suitable index — otherwise Hadoop MapReduce would pick indexes randomly. Hence, we have to enrich the HDFS namenode to keep additional information about the available indexes. We do this by keeping an additional directory Dir_rep mapping (blockID, datanode) $\mapsto$ HAILBlockReplicaInfo. An instance of HAILBlockReplicaInfo contains detailed information about the types of available indexes for a replica, i.e., indexing key, index type, size, start offsets, etc. As before, Hadoop MapReduce looks up Dir_block to retrieve the list of datanodes having a replica for a given block. However, in addition, HAIL looks up the main memory Dir_rep to obtain the detailed HAILBlockReplicaInfo for each replica, i.e., one main memory lookup for each replica. HAILBlockReplicaInfo is then exploited by HAIL to change the scheduling strategy of Hadoop (we will discuss this in detail in Section 3.4).

3.3.4 An Index Structure for Zero-Overhead Indexing

In this section, we briefly discuss our choice of an appropriate index structure for indexing at minimal costs in HAIL and give some details on our concrete implementation.

Why Clustered Indexes? An interesting question is why we focus on clustered indexes. For indexing with minimal overhead, we require an index structure that is cheap to *create in main memory*, cheap to *write to disk*, and cheap to *query from disk*. We tried a number of indexes in the beginning of the project — including coarse-granular indexes and unclustered indexes. After some experimentation we, quickly discovered that sorting and index creation in main memory is so fast that techniques like partial or coarse-granular sorting do not pay off for HAIL. Whether you pay three or two seconds for sorting and indexing per block during upload is hardly noticeable in the overall upload process of HDFS. In addition, a major problem with unclustered indexes is that they are only competitive for very selective queries as they may trigger considerable random I/O for nonselective index traversals. In contrast, clustered indexes do not have that problem. Whatever the selectivity, we will read the clustered index and scan the qualifying blocks. Hence, even for very low selectivities, the only overhead over a scan is the initial index node traversal, which is negligible. Moreover, as unclustered indexes are dense by definition, they require considerably more additional space on disk and require more write I/O than a sparse clustered index. Thus, using unclustered indexes would severely affect upload times. Yet, an interesting direction for future work would be to extend HAIL to support additional indexes that might boost performance, such as bitmap indexes and inverted lists.

3.4 HAIL Job Execution

We now focus on general job execution in HAIL. First, we present from Bob's perspective how he can enhance MapReduce jobs to benefit from HAIL static indexing (Section 3.4.1). We will explain how Bob can write his MapReduce jobs (almost) as before and run them exactly as when using Hadoop MapReduce. After that, we analyze from the system's perspective the standard Hadoop MapReduce pipeline and then compare how HAIL executes jobs (Section 3.4.2). We will see that HAIL requires only small changes in the Hadoop MapReduce framework, which makes HAIL easy to integrate into newer Hadoop versions (Section 3.4.3). Figure 3.2 shows the query pipeline when Bob runs a MapReduce job on HAIL. Finally, we briefly discuss the case of selections on unindexed attributes, i.e., when a job requests a static index that was not created, as motivation for HAIL adaptive indexing (Section 3.4.4).

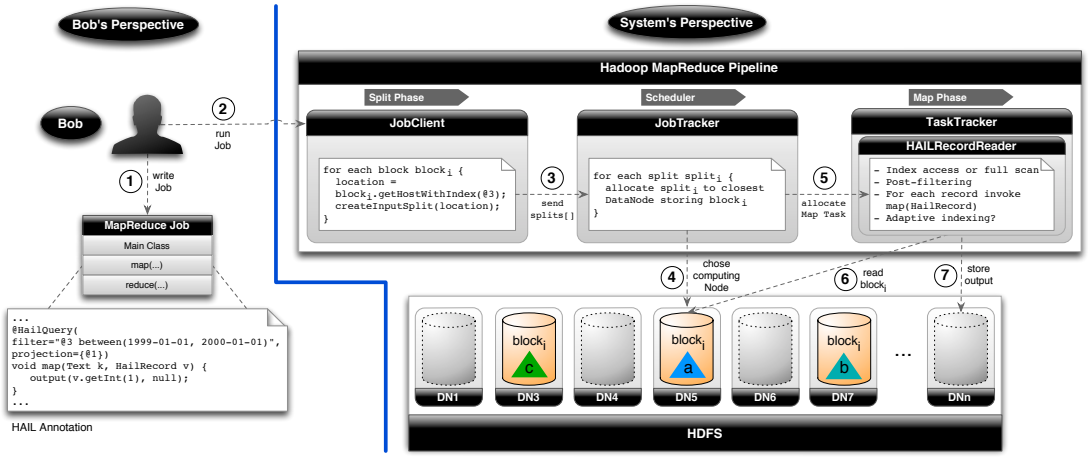


Figure 3.2: *The HAIL query pipeline*

3.4.1 Bob's Perspective

In Hadoop MapReduce, Bob writes a MapReduce job, which includes a job configuration class, a map function, and a reduce function.
In HAIL, the MapReduce job remains the same (see ① and ② in Figure 3.2), but with three tiny changes:

1. Bob specifies the *HailInputFormat* (which uses a *HailRecordReader* internally) in the main class of the MapReduce job. By doing this, Bob enables his MapReduce job to read HAIL Blocks (see Section 3.3.2).

2. Bob annotates his map function to specify the selection predicate and the projected attributes required by his MapReduce job[5]. For example, assume that Bob wants to write a MapReduce job that performs the following SQL query (example from Introduction):

```
SELECT sourceIP
FROM UserVisits
WHERE visitDate BETWEEN '1999-01-01' AND '2000-01-01'
```

   To execute this query in HAIL, Bob adds to his map function a *HailQuery* annotation as follows:

```
@HailQuery(filter="@3 between(1999-01-01,
       2000-01-01)", projection={"@1"})
void map(Text key, Text v) { ... }
```

   Where the literal @3 in the filter value and the literal @1 in the projection value denote the attribute position in the UserVisits records. In this example, the third attribute (i.e., @3) is visitDate and the first attribute (i.e., @1) is sourceIP. By annotating his map function as mentioned above, Bob indicates that he wants to receive in the map function only the projected attribute values of those tuples qualifying the specified selection predicate. In case Bob does not specify filter predicates, HAIL will perform a full scan as the standard Hadoop. At query time, if the HailQuery annotation is set, HAIL checks (using the *Index Metadata* of a data block) whether an index exists on the filter attribute. Using such an index allows us to speed up the job execution. HAIL also uses the *Block Metadata* to determine the schema of a data block. This allows HAIL to read the attributes specified in the filter and projection parameters only.

3. Bob uses a *HailRecord* object as input value in the map function. This allows Bob to directly read the projected attributes without splitting the record into attributes as he would do it in the standard Hadoop MapReduce. In Hadoop, Bob would write the following map function to perform the above SQL query:

   Map Function for Hadoop MapReduce (pseudo-code):

```
void map(Text key, Text v) {
   String[] attr = v.toString().split(",");
   if (DateUtils.isBetween(attr[2],
        "1999-01-01", "2000-01-01"))
      output(attr[0], null);
}
```

[5] Alternatively, HAIL allows Bob to specify the selection predicate and the projected attributes in the job configuration class.

Using HAIL Bob writes the following map function:

Map Function for HAIL:

```
void map(Text key, HailRecord v) {
   output(v.getInt(1), null);
}
```

Notice that, Bob now does not have to filter out the incoming records, because this is automatically handled by HAIL via the HailQuery annotation (as mentioned earlier). This annotation is illustrated in Figure 3.2.

3.4.2 System Perspective

In Hadoop MapReduce, when Bob submits a MapReduce job a *JobClient* instance is created. The main goal of the JobClient is to copy all the resources needed to run the MapReduce job (e.g., metadata and job class files). But also, the JobClient fetches all the block metadata (BlockLocation[]) of the input dataset. Then, the JobClient logically breaks the *input* into smaller pieces called *input splits* (*split phase* in Figure 3.2) as defined in the *InputFormat*. By default, the JobClient computes input splits such that each input split maps to a distinct HDFS block. An input split defines the input of a map task while an HDFS block is a horizontal partition of a dataset stored in HDFS (see Section 3.3.1 for details on how HDFS stores datasets). For scheduling purposes, the JobClient retrieves for each input split all datanode locations having a replica of that HDFS block. This is done by calling getHosts() of each BlockLocation. For instance, in Figure 3.2, datanodes DN3, DN5, and DN7 are the *split locations* for split_{42} since block_{42} is stored on such datanodes.

After this split phase, the JobClient submits the job to the *JobTracker* with the set of input splits to process ③. Among other operations, the JobTracker creates a *map task* for each input split. Then, for each map task, the JobTracker decides on which computing node to schedule the map task, using the split locations ④. This decision is based on data locality and availability [27]. After this, the JobTracker allocates the map task to the *TaskTracker* (which performs map and reduce tasks) running on that computing node ⑤.

Only then, the map task can start processing its input split. The map task uses a *RecordReader* UDF in order to read its input data block_i from the closest datanode ⑥. Interestingly, it is the *local HDFS client* running on the node where the map task is running that decides from which datanode a map task will read its input — and *not* the Hadoop MapReduce scheduler. This is done when the RecordReader asks for the input stream pointing to block_i. It is worth noticing that the HDFS client chooses a datanode from the set of all datanodes storing a replica of block_{42} (via the getHosts() method) rather than from the locations given by the input split. This means that a map task might

eventually end up reading its input data from a remote node even though it is available locally. Once the input stream is opened, the RecordReader breaks $block_{42}$ into records and makes a call to the map function for each record. Assuming that the MapReduce job consists of a map phase only, the map task then writes its output back to HDFS ⑦. See [35, 111, 34] for more details on the MapReduce execution pipeline.

In HAIL, it is crucial to be nonintrusive to the standard Hadoop execution pipeline so that users run MapReduce jobs exactly as before. However, supporting per-replica indexes in an efficient way and without significant changes to the standard execution pipeline is challenging for several reasons. First, the JobClient cannot simply create input splits based only on the default block size as each HDFS block replica has a different size (because of indexes). Second, the JobTracker can no longer schedule map tasks based on data locality and nodes availability only. The JobTracker now has to consider the existing indexes for each HDFS block. Third, the RecordReader has to perform either index access or full scan of HDFS blocks without any interaction with users, e.g., depending on the availability of suitable indexes. Fourth, the HDFS client cannot anymore open an input stream to a given HDFS block based on data locality and nodes availability only: it has to consider index locality and availability as well. HAIL overcomes these issues by mainly providing two UDFs: the HailInputFormat and the HailRecordReader. By using UDFs, we allow HAIL to be easy to integrate into newer versions of Hadoop MapReduce. We discuss these two UDFs in the following.

3.4.3 HailInputFormat and HailRecordReader

HAILInputFormat implements a different splitting strategy than standard InputFormats. This strategy allows HAIL to reduce the number of *map waves* per job, i.e., the maximum number of map tasks per map slot required to complete this job. Thereby, the total scheduling overhead of MapReduce jobs is drastically reduced. We discuss the details of the HAIL splitting strategy in Section 3.7.

HAILRecordReader is responsible for retrieving the records that satisfy the selection predicate of MapReduce jobs (as illustrated in the MapReduce Pipeline of Figure 3.2). Those records are then passed to the map function. For example in Bob's query of Section 3.4.1, we need to find all records having a visitDate between 1999-01-01 and 2000-01-01. To do so, for each data block required by the job, we first try to open an input stream to a block replica having the required index. For this, HAIL instructs the local HDFS Client to use the newly introduced getHostsWithIndex() method of each BlockLocation so as to choose the closest datanode with the desired index. Let us first focus on the case where a suitable, statically created index is available so that HAIL can open an input stream to an indexed replica. Once that input stream has been opened, we use the information about selection predicates and attribute projections from the HailQuery annotation or from the job configuration file. When performing an index scan, we read the index entirely into main memory (typically a few KB) to perform an index

lookup. This also implies reading the qualifying block parts from disk into main memory and post-filtering records (see Section 3.3.4). Then, we reconstruct the projected attributes of qualifying tuples from PAX to row layout. In case that no projection was specified by users, we then reconstruct all attributes. Finally, we make a call to the map function for each qualifying tuple. For bad records (see Section 3.3.1), HAIL passes them directly to the map function, which in turn has to deal with them (just like in standard Hadoop MapReduce). For this, HAIL passes a record to the map function with a flag to indicate a bad record or not.

3.4.4 Missing Static Indexes

Finally, let us now discuss the second case when Bob submits a job which filters on an unindexed attribute (e.g., on duration). Here, the HailRecordReader must completely scan the required attributes of unindexed blocks, apply the selection predicate, and perform tuple reconstruction. Notice that, with static indexing, there is no way for HAIL to overcome the problem of missing indexes efficiently. This means that when the attributes used in the selection predicates of the workload change over time, the only way to adapt the set of available indexes is to upload the data again. However, this has the significant overhead of an additional upload, which goes against the principle of zero-overhead indexing. Thus, HAIL introduces an adaptive indexing technique that offers a much more elegant and efficient solution to this problem. We discuss this technique in the following section.

3.5 HAIL Zero-Overhead Adaptive Indexing

We now discuss the *adaptive indexing pipeline* of HAIL. The core idea is to create missing but promising indexes as byproducts of full scans in the map phase of MapReduce jobs. Similar to the static indexing pipeline, our goal is to come closer towards zero overhead indexing. Therefore, we adopt two important principles from our static indexing pipeline. First, we piggyback again on a procedure that is naturally reading data from disk to main memory. This allows HAIL to completely save the read cost for adaptive index creation. Second, as map tasks are usually I/O-bound, HAIL again exploits unused CPU time when computing clustered indexes in parallel to job execution.

In Section 3.5.1, we start with a general overview of the HAIL adaptive indexing pipeline. In Section 3.5.2, we focus on the internal components for building and storing clustered indexes incrementally. In Section 3.5.3, we present how HAIL accesses the indexes created at job runtime in a way that is transparent to the MapReduce job execution pipeline. Finally, in Section 3.6, we introduce three additional adaptive indexing techniques that make the indexing overhead over MapReduce jobs almost invisible to users.

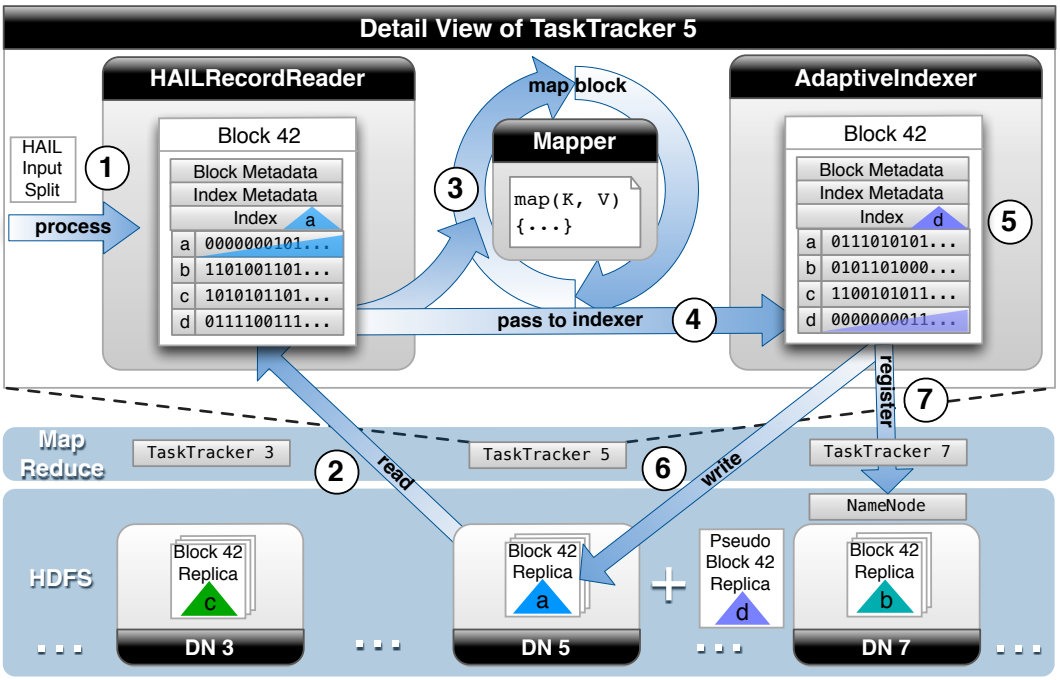


Figure 3.3: *HAIL adaptive indexing pipeline.*

3.5.1 HAIL Adaptive Indexing in the Execution Pipeline

For our motivating example, let's assume Bob continues to analyze his logs and notices some suspicious activities, e.g., many user visits with very short duration, indicating spam bot activities. Therefore, Bob suddenly needs different jobs for his analysis that selects user visits with short durations. However, recall that unfortunately he did not create a static index on attribute duration at upload time which would help for these new jobs. In general, as soon as Bob (or one of his colleagues) sends a new job (say job_d) with a selection predicate on an unindexed attribute (e.g., on attribute duration, which we will denote as d in the following.), HAIL cannot benefit from index scans anymore. However, HAIL takes these jobs as hints on how to adaptively improve the repertoire of indexes for future jobs. HAIL piggybacks the creation of a clustered index over attribute duration on the execution of job_d. Without any loss of generality, we assume that job_d projects all attributes from its input dataset.

Figure 3.3 illustrates the general workflow of the HAIL adaptive indexing pipeline. The figure shows how HAIL processes map tasks of job_d when no suitable index is available (i.e., when performing a full scan) in more detail. As soon as HAIL schedules a map task to a specific TaskTracker[6], e.g. TaskTracker 5, the HAILRecordReader of the

[6] A Hadoop instance responsible to execute map and reduce tasks.

map task first reads the metadata from the HAILInputSplit ①[7]. With this metadata, the HAILRecordReader checks whether a suitable index is available for its input data block (say $block_{42}$). As no index on attribute d is available, the HAILRecordReader simply opens an input stream to the local replica of $block_{42}$ stored on DataNode 5. Then, the HAILRecordReader: (i) loads all values of the attributes required by job_d from disk to main memory ②; (ii) reconstructs records (as our HDFS blocks are in columnar layout); and (iii) feeds the map function with each record ③. Here lies the beauty of HAIL: an HDFS block that is a potential candidate for indexing was completely transferred to main memory as part of the job execution process. In addition to feeding the entire $block_{42}$ to the map function, HAIL can create a clustered index on attribute d to speed up future jobs. For this, the HAILRecordReader passes $block_{42}$ to the *AdaptiveIndexer* as soon as the map function finished processing this data block ④.[8] The AdaptiveIndexer, in turn, sorts the data in $block_{42}$ according to attribute d, aligns other attributes through reordering, and creates a sparse clustered index ⑤. Finally, the AdaptiveIndexer stores this index together with a copy of $block_{42}$ (sorted on attribute d) as a *pseudo data block replica* ⑥. Additionally, the AdaptiveIndexer registers the new created index for $block_{42}$ with the HDFS NameNode ⑦. In fact, the implementation of the adaptive indexing pipeline solves some interesting technical challenges. We discuss the pipeline in more detail in the remainder of this section.

3.5.2 AdaptiveIndexer Architecture

Adaptive indexing is an automatic process that is not explicitly requested by users and therefore should not unexpectedly impose significant performance penalties on users' jobs. Piggybacking adaptive indexing on map tasks allows us to completely save the read I/O-cost. However, the indexing effort is shifted to query time. As a result, any additional time involved in indexing will potentially add to the total runtime of MapReduce jobs. Therefore, the first concern of HAIL is *how to make adaptive index creation efficient?*

To overcome this issue, the idea of HAIL is to run the mapping and indexing processes in parallel. However, interleaving map task execution with indexing bears the risk of race conditions between map tasks and the AdaptiveIndexer on the data block. In other words, the AdaptiveIndexer might potentially reorder data inside a data block, while the map task is still concurrently reading the data block. One might think about copying data blocks before indexing to deal with this issue. Nevertheless, this would entail the additional runtime and memory overhead of copying such memory chunks. For this reason, HAIL does not interleave the mapping and indexing processes on

[7] That was obtained from the HAILInputFormat via getSplits().

[8] Notice that, all map tasks (even from different MapReduce jobs) running on the same node interact with the same AdaptiveIndexer instance. Hence, the AdaptiveIndexer can end up by indexing data blocks from different MapReduce jobs at the same time.

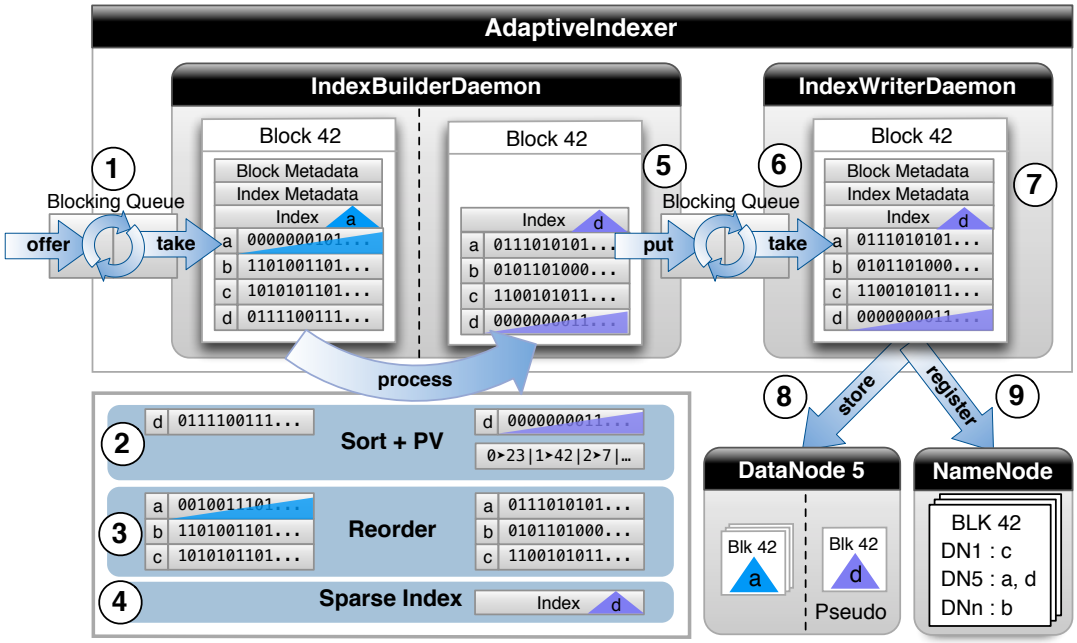


Figure 3.4: *AdaptiveIndexer internals.*

the same data block. Instead, HAIL interleaves the indexing of a given data block (e.g., $block_{42}$) with the mapping phase of the succeeding data block (e.g. $block_{43}$), i.e. , HAIL keeps two HDFS blocks in memory at the same time. For this, HAIL uses a *producer-consumer* pattern: a map task acts as producer by offering a data block to the AdaptiveIndexer, via a bounded blocking queue, as soon as it finishes processing the data block; in turn, the AdaptiveIndexer is constantly consuming data blocks from this queue. As a result, HAIL can perfectly interleave map tasks with indexing, except for the first and last data block to process in each node. It is worth noting that the queue exposed by the AdaptiveIndexer is allowed to reject data blocks in case a certain limit of enqueued data blocks is exceeded. This prevents the AdaptiveIndexer to run out of memory because of overload. Still, future MapReduce jobs with a selection predicate on the same attribute (i.e., on attribute *d*) can at their turn take care of indexing the rejected data blocks. Once the AdaptiveIndexer pulls a data block from its queue, it processes the data block using two internal components: the *IndexBuilder* and the *IndexWriter*. Figure 3.4 illustrates the pipeline of these two internal components, which we discuss in the following.

The IndexBuilder is a daemon thread that is responsible for creating sparse clustered indexes on data blocks in the data queue. With this aim, the IndexBuilder is constantly pulling one data block after another from the data block queue (1). Then, for each data block, the IndexBuilder starts with sorting the attribute column to index (attribute *d* in

our example) ②. Additionally, the IndexBuilder builds a mapping {*old_position* ↦ *new_position*} for all values as a permutation vector. After that, the IndexBuilder uses the permutation vector to reorder all other attributes in the offered data block ③. Once the IndexBuilder finishes sorting the entire data block on attribute *d*, it builds a sparse clustered index on attribute *d* ④. Then, the IndexBuilder passes the newly indexed data block to the IndexWriter ⑤. The IndexBuilder also communicates with the IndexWriter via a blocking queue. This allows HAIL to parallelize indexing with the I/O process for storing newly indexed data blocks.

The IndexWriter is another daemon thread and responsible for persisting indexes created by the IndexBuilder to disk. The IndexWriter continuously pulls newly indexed data blocks from its queue in order to persist them on HDFS ⑥. Once the IndexWriter pulls a newly indexed data block (say $block_{42}$), it creates the block metadata and index metadata for $block_{42}$ ⑦. Notice that, a newly indexed data block is just another replica of the logical data block, but with a different sort order. For instance, in our example of Section 3.5.1, creating an index on attribute *d* for $block_{42}$ leads to having four data block replicas for $block_{42}$: one replica for each of the first four attributes. The IndexWriter creates a *pseudo data block replica* ⑧ and registers the new index with the NameNode ⑨. This allows HAIL to consider the newly created indexes in future jobs. In the following, we discuss pseudo data block replicas in more detail.

3.5.3 Pseudo Data Block Replicas

The IndexWriter could simply write a new indexed data block as another replica. However, HDFS supports data block replication only at the file level, i.e., HDFS replicates all the data blocks of a given dataset the same number of times. This goes against the incremental nature of HAIL. A pseudo data block replica is basically a logical copy of a data block and allows HAIL to keep a different replication factor on a block basis rather than on a file basis. Therefore, we store each pseudo data block replica in a new HDFS file with replication factor one. Hence, the NameNode does not recognize it as a normal data block replica and instead simply sees the pseudo data block replica as another index available for the HDFS block. To avoid shipping across nodes, each IndexWriter aims at storing the pseudo data block replicas locally. The created HDFS files follow a naming convention, which includes the block id and the index attribute, to uniquely identify a pseudo data block replica.

As pseudo data block replicas are stored in different HDFS files than normal data block replicas, three important questions arise:

*H*ow to access pseudo data block replicas in an invisible way for users? HAIL achieves this transparency via the HAILRecordReader. Users continue annotating their map functions (with selection predicates and projections). Then, the HAILRecordReader takes care of automatically switching from normal to pseudo data block replicas. For

this, the HAILRecordReader uses the *HAILInputStream*, a wrapper of the Hadoop FS-InputStream.
*H*ow to manage and limit the storage space consumed by the pseudo data block replicas? This question is related to optimization problems from physical database design, i.e., index selection. Given a certain storage budget, the question is which indexes for an HDFS block to drop, to achieve the highest workload benefit without exceeding the storage constraint? Solving this problem is beyond the scope of this article and is subject to ongoing work. A simple implementation could borrow ideas from buffer replacement strategies to attack the problem, e.g., LRU or replacing the least beneficial indexes.
*H*ow does the amount of relatively small files created for pseudo data block replicas impact HDFS performance? The metadata storage overhead for each file entry with one associated block in the NameNode is about 150 bytes. This means that given 6GB of free heap space on the NameNode and an HDFS block size of 256MB, HAIL can support more than 10PB of data in pseudo block replicas. Additionally, future Hadoop versions will support a federation of NameNodes to increase capacity, availability, and load balancing. This would alleviate the mentioned problem even further. Furthermore, sequential read performance of a file that is stored in pseudo data block replicas matches the performance of normal HDFS files. This is because the involved amount of seeks and DataNode hops for switching between pseudo data block replicas is comparable to reading over block boundaries when scanning normal HDFS files.

3.5.4 HAIL RecordReader Internals

Figure 3.5 illustrates the internal pipeline of the HAILRecordReader when processing a given HAILInputSplit. When a map task starts, the HAILRecordReader first reads the metadata of its HAILInputSplit in order to check if there exists a suitable index to process the input data block ($block_{42}$) ①. If a suitable index is available, the HAIL-RecordReader initializes the HAILInputStream with the selection predicate of job_d as a parameter②. Internally, the HAILInputStream checks if the index resides in a normal or pseudo data block replica③. This allows the HAILInputStream to open an input stream to the right HDFS file. This is because normal and pseudo data block replicas are stored in different HDFS files. While all normal data block replicas belong to the same HDFS file, each pseudo data block replica belongs to a different HDFS file④. In our example, the index on attribute d for $block_{42}$ resides in a pseudo data block replica. Therefore, the HAILInputStream opens an input stream to the HDFS file `/pseudo/blk_42/d`⑤. As a result, the HAILRecordReader does not care from which file it is reading, since normal and pseudo data block replicas have the same format. Therefore, switching between a normal and a pseudo data block replica is not only invisible to users, but also to the HAILRecordReader. The HAILRecordReader just reads the block and index metadata using the HAILInputStream ⑥. After performing an index lookup for the selection predicate of job_d, the HAILRecordReader loads only the projected attributes (a, b, c,

and *d*) from the qualifying tuples (e.g., tuples with rowIDs in 1024 – 2048) ⑦. Finally, the HAILRecordReader forms key/value-pairs and passes only qualifying pairs to the map function ⑧.

In case that no suitable index exists, the HAILRecordReader takes the Hadoop InputStream, which opens an input stream to any normal data block replica, and falls back to full scan (like standard Hadoop MapReduce).

3.6 Adaptive Indexing Strategies

In the previous section, we discussed the core principles of the HAIL adaptive indexing pipeline. Now, we introduce three strategies that allow HAIL to improve the performance of MapReduce jobs. We first present *lazy adaptive indexing* and *eager adaptive*

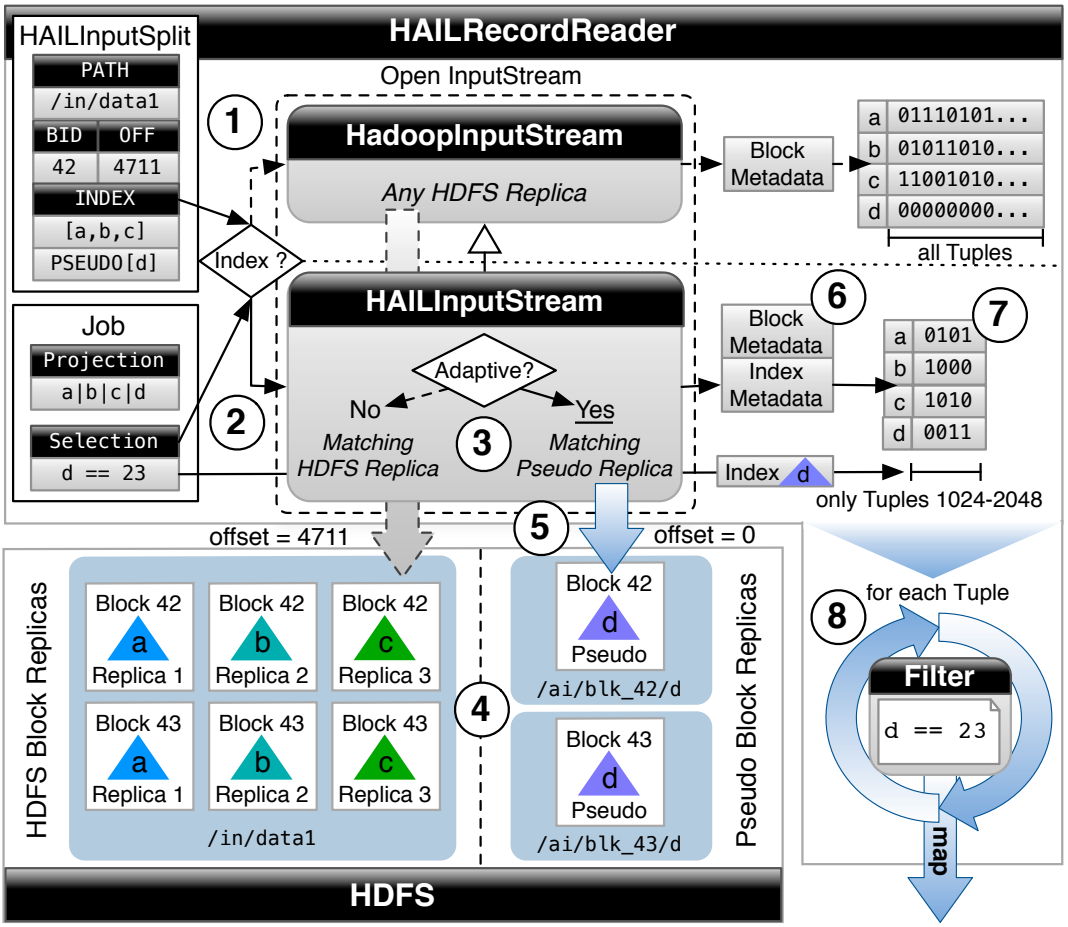


Figure 3.5: *HAILRecordReader internals.*

indexing, two techniques that allow HAIL to control its incremental indexing mechanism with respect to runtime overhead and convergence rate. We then discuss how HAIL can prioritize data blocks for indexing based on their selectivity. Finally, we introduce *selectivity-based indexing*, a technique to decide which blocks to offer to the adaptive indexer based on job selectivity.

3.6.1 Lazy Adaptive Indexing

The blocking queues used by the AdaptiveIndexer allow us to easily protect HAIL against CPU overloading. However, writing pseudo data block replicas can also slow down the parallel read and write processes of MapReduce jobs. In fact, the negative impact of extra I/O operations can be high, as MapReduce jobs are typically I/O-bound. As a result, HAIL as a whole might become slower even if the AdaptiveIndexer can computationally keep up with the job execution. So, the question that arises is *how to write pseudo data block replicas efficiently?*

HAIL solves this problem by making indexing incremental, i.e., HAIL spreads index creation over multiple MapReduce jobs. The goal is to balance index creation cost over multiple MapReduce jobs so that users perceive small (or no) overhead in their jobs. To do so, HAIL uses an *offer rate*, which is a ratio that limits the maximum number of pseudo data block replicas (i.e., number of data blocks to index) to create during a single MapReduce job. For example, using an offer rate of 10%, HAIL indexes in a single MapReduce job at maximum one data block out of ten processed data blocks (i.e., HAIL only indexes 10% of the total data blocks). Notice that, consecutive adaptive indexing jobs with selections on the same attribute already benefit from pseudo data block replicas created during previous jobs. This strategy has two major advantages. First, HAIL can reduce the additional I/O introduced by indexing to a level that is acceptable for the user. Second, the indexing effort done by HAIL for a certain attribute is proportional to the number of times a selection is performed on that attribute. Another advantage of using an offer rate is that users can decide how fast they want to converge to a *complete index*, i.e., all data blocks are indexed. For instance, using an offer rate of 10%, HAIL would require 10 MapReduce jobs with a selection predicate on the same attribute to converge to a *complete index* (i.e., until all HDFS blocks are fully indexed). Like that, on the one hand, the investment in terms of time and space for MapReduce jobs with selection predicates on unfrequent attributes is minimized. On the other hand, MapReduce jobs with selection predicates on frequent attributes quickly converge to a completely indexed copy.

3.6.2 Eager Adaptive Indexing

Lazy adaptive indexing allows HAIL to easily throttle down adaptive indexing efforts to an acceptable (or even invisible) degree for users (see Section 3.6.1). However, let us

make two important observations that could make a constant offer rate not desirable for certain users:

1. Using a constant offer rate, the job runtime of consecutive MapReduce jobs having a filter condition on the same attribute is not constant. Instead, they have an almost linearly decreasing runtime up to the point where all blocks are indexed. This is because the first MapReduce job is the only to perform a full scan over all the data blocks of a given dataset. Consecutive jobs, even when indexing and storing the same amount of blocks, are likely to run faster as they benefit from all indexing work of their predecessors.

2. HAIL actually delays indexing by using an offer rate. The tradeoff here is that using a lower offer rate leads to a lower indexing overhead, but it requires more MapReduce jobs to index all the data blocks in a given dataset. However, some users might want to limit the experienced indexing overhead and still desire to benefit from complete indexing as soon as possible.

Therefore, we propose an *eager adaptive indexing* strategy to deal with this problem. The basic idea of eager adaptive indexing is to dynamically adapt the offer rate for MapReduce jobs according to the indexing work achieved by previous jobs. In other words, eager adaptive indexing tries to exploit the saved runtime and reinvest it as much as possible into further indexing. To do so, HAIL first needs to estimate the runtime gain (in a given MapReduce job) from performing an index scan on the already created pseudo data block replicas. For this, HAIL uses a cost model to estimate the total runtime, T_{job}, of a given MapReduce job (Equation 3.1). Table 3.1 lists the parameters we use in the cost model.

$$T_{job} = T_{is} + t_{fsw} \cdot n_{fsw} + T_{idxOverhead}. \tag{3.1}$$

We define the number of map waves performing a full scan, n_{fsw}, as $\lceil \frac{n_{blocks} - n_{idxBlocks}}{n_{slots}} \rceil$. Intuitively, the total runtime T_{job} of a job consists of three parts. First, the time required by HAIL to process the existing pseudo data block replicas, i.e., all data blocks having a relevant index, T_{is}. Second, the time required by HAIL to process the data blocks without a relevant index, $t_{fsw} \cdot n_{fsw}$. Third, the time overhead caused by adaptive indexing, $T_{idxOverhead}$.[9] This overhead depends on the number of data blocks that are offered to the AdaptiveIndexer and the average time overhead observed for indexing a block. Formally, we define $T_{idxOverhead}$ as follows:

$$T_{idxOverhead} = t_{idxOverhead} \cdot \min\left(\rho \cdot \left\lceil \frac{n_{blocks}}{n_{slots}} \right\rceil, n_{fsw}\right). \tag{3.2}$$

[9] It is worth noting that $T_{idxOverhead}$ denotes only the additional runtime that a MapReduce job has due to adaptive indexing.

Table 3.1: *Cost model parameters.*

Notation	Description
n_{slots}	The number of map tasks that can run in parallel in a given Hadoop cluster
n_{blocks}	The number of data blocks of a given dataset
$n_{idxBlocks}$	The number of blocks with a relevant index
n_{fsw}	The number of map waves performing a full scan
t_{fsw}	The *average* runtime of a map wave performing a full scan (without adaptive indexing overhead)
$t_{idxOverhead}$	The average time overhead of adaptive indexing in a map wave
$T_{idxOverhead}$	The *total* time overhead of adaptive indexing
T_{is}	The total runtime of the map waves performing an index scan
T_{job}	The total runtime of a given job
T_{target}	The targeted total job runtime
ρ	The ratio of data blocks (w.r.t. n_{blocks}) offered to the AdaptiveIndexer

We can use this model to automatically calculate the offer rate ρ in order to keep the adaptive indexing overhead acceptable for users. Formally, from Equations 3.1 and 3.2, we deduct ρ as follows:

$$\rho = \frac{T_{target} - T_{is} - t_{fsw} \cdot n_{fsw}}{t_{idxOverhead} \cdot \lceil \frac{n_{blocks}}{n_{slots}} \rceil}.$$

Therefore, given a target job runtime T_{target}, HAIL can automatically set ρ in order to fully spent its time budget for creating indexes and use the gained runtime in the next jobs either to speed up the jobs or to create even more indexes. Usually, we choose T_{target} to be equal to the runtime of the very first job so that users can observe a stable runtime till almost everything is indexed. However, users can set T_{target} to any time budget in order to adapt the indexing effort to their needs. Notice that, since already

indexed pseudo data block replicas are not offered again to the AdaptiveIndexer, HAIL first processes pseudo data block replicas and measures T_{is}, before deciding what offer rate to use for the unindexed blocks. The times t_{fsw} (from Equation 3.1) and $t_{idxOverhead}$ (from Equation 3.2) can be measured in a calibration job or given by users.

On the one hand, HAIL can now adapt the offer rates to the performance gains obtained from performing index scans over the already indexed data blocks. On the other hand, by gradually increasing the offer rate, eager adaptive indexing prioritizes complete index convergence over early runtime improvements for users. Thus, users no longer experience an incremental and linear speed up in job performance until the index is eventually complete, but instead they experience a sharp improvement when HAIL approaches to a complete index. In summary, besides limiting the overhead of adaptive indexing, the offer rate can also be considered as a tuning knob to trade early runtime improvements with faster indexing.

3.6.3 Selectivity-based Adaptive Indexing

Earlier, we saw that HAIL uses an offer rate to limit the number of data blocks to index in a single MapReduce job. For this, HAIL uses a round robin policy to select the data blocks to pass to the AdaptiveIndexer. This sounds reasonable under the assumption that data is uniformly distributed. However, datasets are typically skewed in practice, and hence, some data blocks might contain more qualifying tuples than others under a given query workload. Consequently, indexing highly selective data blocks before other data blocks promises higher performance benefits.

Therefore, HAIL can also use a selectivity-based data block selection approach for deciding which data blocks to use. The overall goal is to optimize the use of available computing resources. In order to maximize the expected performance improvement for future MapReduce jobs running on partially indexed datasets, we prioritize HDFS blocks with a higher selectivity. The big advantage of this approach is that users can perceive higher improvements in performance for their MapReduce jobs from the very first runs. Additionally, as a side effect of using this approach, HAIL can adapt faster to the selection predicates of MapReduce jobs.

However, *how can HAIL efficiently obtain the selectivities of data blocks?* For this, HAIL exploits the natural process of map tasks to propose data blocks to the AdaptiveIndexer. Recall that a map task passes a data block to the AdaptiveIndexer once the map task finished processing the block. Thus, HAIL can obtain the accurate selectivity of a data block by piggybacking on the map phase, when the data block is filtered according to the provided selection predicate. This allows HAIL to have perfect knowledge about selectivities for free. Given the selectivity of a data block, HAIL can decide if it is worth to index the data block or not. In our current HAIL prototype, a map task proposes a data block to the AdaptiveIndexer if the percentage of qualifying tuples in the data block is at most 80%. However, users can adapt this threshold to their applica-

tions. Notice that, with the statistics on data block selectivities, HAIL can also decide which indexes to drop in case of storage limitations. However, a discussion on an index eviction strategy is out of the scope of this article.

3.7 HAIL Splitting and Scheduling

We now discuss how HAIL creates and schedules map tasks for any incoming MapReduce job.

In contrast to the Hadoop MapReduce InputFormat, the HailInputFormat uses a more elaborate splitting policy, called *HailSplitting*. The overall idea of HailSplitting is to map one input split to several data blocks whenever a MapReduce job performs an index scan over its input. In the beginning, HailSplitting divides all input data blocks into two groups B_i and B_n. Where B_i contains blocks that have at least one replica with a matching index (i.e., having a relevant replica) and B_n contains blocks with no relevant replica. Then, the main goal of the HailSplitting is to combine several data blocks from B_i into one input split. For this, HailSplitting first partitions data blocks from B_i according to the locations of their relevant replica in order to improve data locality. As a result of this process, HailSplitting produces as many partitions of blocks as there are datanodes storing at least one indexed block of the given input. Then, for each partition of data blocks, HailSplitting creates as many input splits as there exists map slots per TaskTracker. Thus, HAIL reduces the number of map tasks and hence reduces the aggregated costs of initializing and finalizing map tasks.

The reader might think that using several blocks per input split may significantly impact failover. However, this is not true since tasks performing an index scan are relatively short running. Therefore, the probability that one node fails in this period of time is very low [94]. Still, in case a node fails in this period of time, HAIL simply reschedules the failed map tasks, which results only in a few seconds overhead anyways. Optionally, HAIL could apply the checkpointing techniques proposed in [94] in order to improve failover. We will study these interesting aspects in a future work. The reader might also think that performance could be negatively impacted in case that data locality is not achieved for several map tasks. However, fetching small parts of blocks through the network (which is the case when using index scan) is negligible [66]. Moreover, one can significantly improve data locality by simply using an adequate scheduling policy (e.g., the Delay Scheduler [113]). If no relevant index exists, HAIL scheduling falls back to standard Hadoop scheduling by optimizing data locality only.

For all data blocks in B_n, HAIL creates one map task per unindexed data block just like standard Hadoop. Then, for each map task, HAIL considers r different computing nodes as possible locations to schedule a map task, where r is the replication factor of the input dataset. However, in contrast to original Hadoop, HAIL prefers to assign map tasks to those nodes that currently store less indexes than the average. Since HAIL stores

pseudo data block replicas local to the map tasks that created them, this scheduling strategy results in a balanced index placement and allows HAIL to better parallelize index access for future MapReduce jobs.

3.8 Related Work

HAIL uses PAX [4] as data layout for HDFS block, i.e., a columnar layout inside the HDFS block. PAX was originally invented for cache-conscious processing, but it has been adapted in the context of MapReduce [24]. In our previous work [66], we showed how to improve over PAX by computing different layouts on the different replicas, but we did not consider indexing. This article fills this gap.

Static Indexing

Indexing is a crucial step in all major DBMSs [44, 22, 3, 19, 23]. The overall idea behind all these approaches is to analyze a query workload and to statically decide which attributes to index based on these observations. Several research works have focused on supporting index access in MapReduce workflows [112, 79, 35, 64]. However, all these offline approaches have three big disadvantages. First, they incur a high upfront indexing cost that several applications cannot afford (such as scientific applications). Second, they only create a single clustered index per dataset, which is not suitable for query workloads having selection predicates on different attributes. Third, they cannot adapt to changes in workloads without the intervention of a DBA.

Online Indexing

Tuning a database at upload time has become harder as query workloads become more dynamic and complex. Thus, different DBMSs started to use online tuning tools to attack the problem of dynamic workloads [103, 17, 18, 84]. The idea is to continuously monitor the performance of the system and create (or drop) indexes as soon as it is considered beneficial. Manimal [20, 63] can be used as an online indexing approach for automatically optimizing MapReduce jobs. The idea of Manimal is to generate a MapReduce job for index creation as soon as an incoming MapReduce job has a selection predicate on an unindexed attribute. Online indexing can then adapt to query workloads. However, online indexing techniques require us to index a dataset completely in one pass. Therefore, online indexing techniques simply transfer the high cost of index creation from upload time to query processing time.

Adaptive Indexing

HAIL is inspired by database cracking [58] which aims at removing the high upfront cost barrier of index creation. The main idea of database cracking is to start organizing a given attribute (i.e., to create an adaptive index on an attribute) when it receives for the first time a query with a selection predicate on that attribute. Thus, future incoming queries having predicates on the same attribute continue refining the adaptive index as long as finer granularity of key ranges is advantageous. Key ranges in an adaptive index are disjoint, where keys in each key range are unsorted. Basically, adaptive indexing performs for each query one step of *quicksort* using the selection predicates as pivot for partitioning attributes. HAIL differs from adaptive indexing in four aspects. First, HAIL creates a clustered index for each data block and hence avoids any data shuffling across data blocks. This allows HAIL to preserve Hadoop fault-tolerance. Second, HAIL considers disk-based systems, and thus, it factors in the cost of reorganizing data inside data blocks. Third, HAIL parallelizes the indexing effort across several computing nodes to minimize the indexing overhead. Fourth, HAIL focuses on creating clustered indexes instead of unclustered indexes. A follow-up work [59] focuses on lazily aligning attributes to converge into a clustered index after a certain number of queries. However, it considers a main memory system and hence does not factor in the I/O-cost for moving data many times on disk. Other works on adaptive indexing in main memory databases have focused on updates [62], concurrency control [49], and robustness [54], but these works are orthogonal to the problem we address in this chapter.

Adaptive Merging

Another related work to HAIL is the adaptive merging [50]. This approach uses standard B-trees to persist intermediate results during an external sort. Then, it only merges those key ranges that are relevant to queries. In other words, adaptive merging incrementally performs external sort steps as a side effect of query processing. However, this approach cannot be applied directly for MapReduce workflows for three reasons. First, like adaptive indexing, this approach creates unclustered indexes. Second, merging data in MapReduce destroys Hadoop fault-tolerance and hurts the performance of MapReduce jobs. This is because adaptive merging would require us to merge data from several data blocks into one. Notice that, merging data inside a data block would not make sense as a data block is typically loaded entirely into main memory by map tasks anyways. Third, it has an expensive initial step to create the first sorted runs. A follow-up work uses adaptive indexing to reduce the cost of the initial step of adaptive merging in main memory [61]. However, it considers main memory systems, and hence, it has the first two problems.

Adaptive Loading

Some other works focus on loading data into a database in an incremental [2] or in a lazy [60] manner with the goal of reducing the upfront cost for parsing and storing data inside a database. These approaches allow for reducing the delay until users can execute their first queries dramatically. In the context of Hadoop, [2] proposes to load those parts of a dataset that were parsed as input to MapReduce Jobs into a database at job runtime. Hence, consecutive MapReduce Jobs that require the same data can benefit, e.g., from the binary representation or indexes inside the database store. However, this scenario already involves an additional roundtrip of first writing the data to HDFS, reading it from HDFS to then again store the data inside a database plus some overhead for index creation. In contrast to these works, HAIL aims at reducing the upfront cost of data parsing and index creation already when loading data into HDFS. In other words, while these approaches aim at adaptively uploading raw datasets from HDFS into a database to improve performance, HAIL aims at indexing raw datasets directly in HDFS to improve performance, without additional read/write cycles. NoDB, another recent work, proposes to run queries directly on raw datasets [5]. Additionally, this approach (i) remembers the offsets of individual attribute values, and (ii) caches binary values from the dataset which are both extracted as byproducts of query execution. Those optimizations allow for reducing the tokenizing and parsing costs for consecutive queries that touch previously processed parts of the dataset. However, NoDB considers a single-node scenario using a local file system, while HAIL considers a distributed environment and a distributed file system. As shown in our experiments, writing to HDFS is I/O-bound and parsing the attributes of a dataset entirely can be performed in parallel to storing the data in HDFS. Since data parsing does not cause noticeable runtime overhead in our scenario, incremental loading techniques as presented in [5] are not required for HAIL. Furthermore, NoDB does not consider different sort orders or indexes to improve data access.

To the best of our knowledge, this work is the first work that aims at pushing indexing to the extreme at low index creation cost and to propose an adaptive indexing solution suitable for MapReduce systems.

3.9 Experiments on Static Indexing

Let's get back to Bob again and his initial question: *will HAIL solve his indexing problem efficiently?* To answer this question, we need to run a wave of **experiments for static indexing** in order to answer the following questions as well:

1. What is the performance of HAIL at upload time?

2. What is the impact of static indexing in the upload pipeline?

3. How many indexes can we create in the time the standard HDFS uploads the data?
4. How does hardware performance affect HAIL upload?
5. How well does HAIL scale-out on large clusters?

We answer these questions in Section 3.9.3.

6. What is the performance of HAIL at query time?
7. How much does HAIL benefit from statically created indexes?
8. How does query selectivity affect HAIL?
9. How do failing nodes affect performance?

We answer these questions in Section 3.9.4.

10. How does HailSplitting improve end-to-end job runtimes?

We answer this question in Section 3.9.5.

3.9.1 Hardware and Systems

Hardware

For our experiments on static indexing, we use six different clusters in total. One is a physical 10-node cluster (*Cluster-A*). Each node has one 2.66GHz Quad Core Xeon processor running 64-bit platform Linux openSuse 11.1 OS, 4x4GB of main memory, 6x750GB SATA HD, and three Gigabit network cards. Our physical cluster has the advantage that the amount of runtime variance is limited [101]. Yet, to fully understand the scale-up properties of HAIL, we use three different EC2 clusters, each having 10 nodes. For each of these three clusters, we use different node types (see Section 3.9.3). Finally, to understand how well HAIL scales-out, we consider two more EC2 clusters: one with 50 nodes and one with 100 nodes (see Section 3.9.3).

Systems

We compared the following systems: (1) Hadoop, (2) Hadoop++ as described in [35], and (3) HAIL as described previously. For HAIL, we disable the HAIL splitting in Section 3.9.4 in order to measure the benefits of using this policy in Section 3.9.5. All three systems are based on Hadoop 0.20.203 and are compiled and run using Java 7. All systems were configured to use the default HDFS block size of 64MB if not mentioned otherwise.

3.9.2 Datasets and Queries

Datasets

For our benchmarks, we use two different datasets. First, we use the UserVisits table as described in [91]. This dataset nicely matches Bob's Use-Case. We generated 20GB of UserVisits data per node using the data generator proposed by [91]. Second, we additionally use a Synthetic dataset consisting of 19 integer attributes in order to understand the effects of selectivity. Notice that, this Synthetic dataset is similar to scientific datasets, where all or most of the attributes are integer or float attributes (e.g., the SDSS dataset). For this dataset, we generated 13GB per node.

Queries

For the UserVisits dataset, we consider the following queries as Bob's workload:

- **Bob-Q1** (selectivity: 3.1 x 10^{-2})

```
SELECT sourceIP FROM UserVisits WHERE visitDate
BETWEEN '1999-01-01' AND '2000-01-01'
```

- **Bob-Q2** (selectivity: 3.2 x 10^{-8})

```
SELECT searchWord, duration, adRevenue
FROM UserVisits WHERE sourceIP='172.101.11.46'
```

- **Bob-Q3** (selectivity: 6 x 10^{-9})

```
SELECT searchWord, duration, adRevenue
FROM UserVisits WHERE sourceIP='172.101.11.46'
AND visitDate='1992-12-22'
```

- **Bob-Q4** (selectivity: 1.7 x 10^{-2})

```
SELECT searchWord, duration, adRevenue
FROM UserVisits WHERE adRevenue>=1 AND adRevenue<=10
```

Additionally, we use a variation of query Bob-Q4 to see how well HAIL performs on queries with low selectivities:

- **Bob-Q5** (selectivity: 2.04 x 10^{-1})

```
SELECT searchWord, duration, adRevenue
FROM UserVisits WHERE adRevenue>=1 AND adRevenue<=100
```

For the Synthetic dataset, we use the queries in Table 3.2. Notice that, for Synthetic all queries use the *same* attribute for filtering. Hence, for this dataset HAIL cannot benefit from its different indexes: it creates three different indexes, yet only one of them will be used by these queries.

Table 3.2: *Synthetic queries.*

Query	#Projected Attributes	Selectivity
Syn-Q1a	19	0.10
Syn-Q1b	9	0.10
Syn-Q1c	1	0.10
Syn-Q2a	19	0.01
Syn-Q2b	9	0.01
Syn-Q2c	1	0.01

For all queries and experiments, we report the average runtime of three trials.

3.9.3 Data Loading with Static Indexing

We strongly believe that upload time is a crucial aspect for to adopt a parallel data-intensive system. This is because most users (such as Bob or scientists) want to start analyzing their data early. In fact, low startup costs are one of the big advantages of standard Hadoop over RDBMSs. Thus, we exhaustively study the upload performance of HAIL.

Varying the Number of Indexes

We first measure the impact in performance when creating indexes statically. For this, we scale the number of indexes to create when uploading the UserVisits and the Synthetic datasets. For HAIL, we vary the number of indexes from 0 to 3 and for Hadoop++ from 0 to 1 (this is because Hadoop++ cannot create more than one index). For Hadoop, we only report numbers with 0 indexes as it cannot create any index.

Figure 3.6a shows the results for the UserVisits dataset. We observe that HAIL has a negligible upload overhead of ~2% over standard Hadoop. Then, when HAIL creates one index per replica the overhead still remains very low (at most ~14%). On the other hand, we observe that HAIL improves over Hadoop++ by a factor of 5.1 when creating no index and by a factor of 7.3 when creating one index. This is because Hadoop++ has to run two expensive MapReduce jobs for creating one index. For HAIL, we observe that for two and three indexes, the upload costs increase only slightly.

Figure 3.6b illustrates the results for the Synthetic dataset. We observe that HAIL significantly outperforms Hadoop++ again by a factor of 5.2 when creating no index

and by a factor of 8.2 when creating one index. On the other hand, we now observe that HAIL outperforms Hadoop by a factor of 1.6 even when creating three indexes. This is because the Synthetic dataset is well suited for binary representation, i.e., in contrast to the UserVisits dataset, HAIL can significantly reduce the initial dataset size. This allows HAIL to outperform Hadoop even when creating one, two, or three indexes.

For the remaining upload experiments, we discard Hadoop++ as we clearly saw in this section that it does not upload datasets efficiently. Therefore, we focus on HAIL using Hadoop as baseline.

Varying the Replication Factor

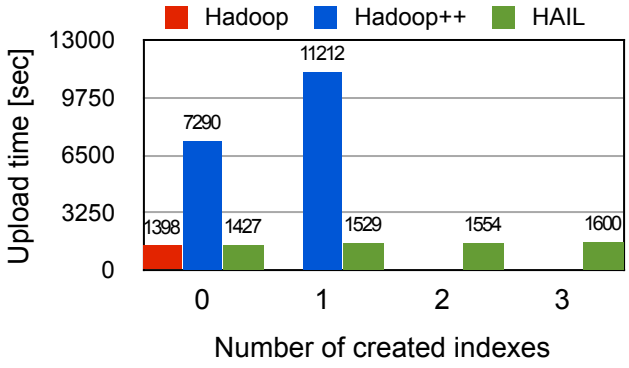
We now analyze how well HAIL performs when increasing the number of replicas. In particular, we aim at finding out how many indexes HAIL can create for a given dataset in the same time standard Hadoop needs to upload the same dataset with the default replication factor of three and creating no indexes. To do this, we upload the Synthetic dataset with different replication factors. In this experiment, HAIL creates as many clustered indexes as block replicas. In other words, when HAIL uploads the Synthetic dataset with a replication factor of five, it creates five different clustered index for each block.

Figure 3.6c shows the results for this experiment. The dotted line marks the time Hadoop takes to upload with the default replication factor of three. We see that HAIL significantly outperforms Hadoop for any replication factor and up to a factor of 2.5. More interestingly, we observe that HAIL stores six replicas (and hence it creates six different clustered indexes) in a little less than the same time Hadoop uploads the same dataset with only three replicas without creating any index. Still, when increasing the replication factor even further for HAIL, we see that HAIL has only a minor overhead over Hadoop with three replicas only. These results also show that choosing the replication factor mainly depends on the available disk space. Even in this respect, HAIL improves over Hadoop. For example, while Hadoop needs 390GB to upload the Synthetic dataset with 3 block replicas, HAIL needs only 420GB to upload the same dataset with 6 block replicas! HAIL enables users to stress indexing to the extreme to speed up their query workloads.

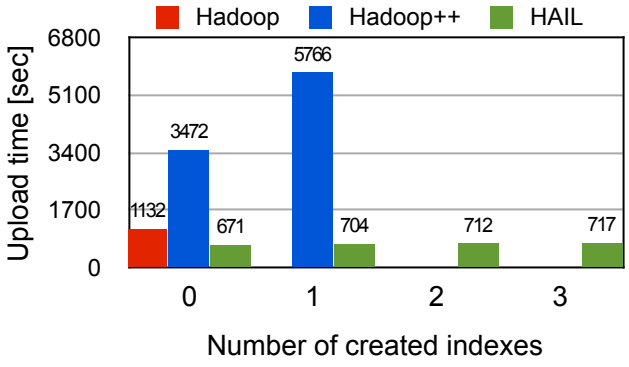
Cluster Scale-Up for Static Indexing

In this section, we study how different hardware affects HAIL upload times. For this, we create three 10-nodes EC2 clusters: the first uses *large* (*m1.large*) nodes[10], the second *extra large* (*m1.xlarge*) nodes, and the third *cluster quadruple* (*cc1.4xlarge*) nodes. We upload the UserVisits and the Synthetic datasets on each of these clusters.

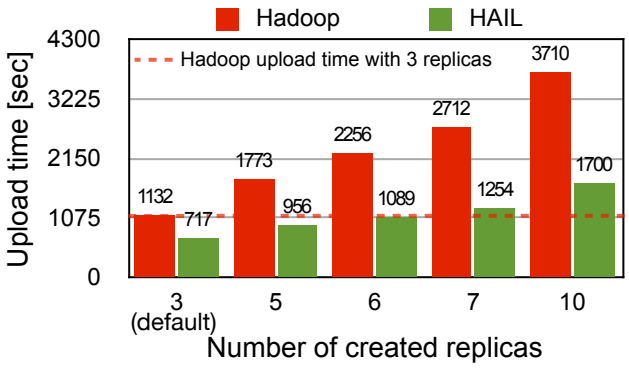
[10]For this cluster type, we allocate an additional large node to run the namenode and jobtracker.



(a) Upload time for UserVisits



(b) Upload time for Synthetic



(c) Varying replication for Synthetic

Figure 3.6: *Upload times when varying the number of created indexes (a)&(b) and the number of data block replicas (c)*

Table 3.3: *Scale-up results*

(a) *Upload times for* `UserVisits` *when scaling-up [sec]*

Cluster Node Type	Hadoop	HAIL	System Speedup
Large	1844	3418	0.54
Extra Large	1296	2039	0.64
Cluster Quadruple	1284	1742	0.74
Scale-Up Speedup	1.4	2.0	
Physical	1398	1600	0.87

(b) *Upload times for* `Synthetic` *when scaling-up [sec]*

Cluster Node Type	Hadoop	HAIL	System Speedup
Large	1176	1023	1.15
Extra Large	788	640	1.23
Cluster Quadruple	827	600	1.38
Scale-Up Speedup	1.4	1.7	
Physical	1132	717	1.58

We report the results of these experiments in Table 3.3a (for UserVisits) and in Table 3.3b (for Synthetic), where we display the *System Speedup* of HAIL over Hadoop as well as the *Scale-Up Speedup* for Hadoop and HAIL. Additionally, we show again the results for our local cluster as baseline. As expected, we observe that both Hadoop and HAIL benefit from using better hardware. In addition, we also observe that HAIL always benefits from scaling-up computing nodes, especially, using a better CPU makes parsing to binary faster. As a result, HAIL decreases (in the 3.3a) or increases (Table 3.3b) the performance gap with respect to Hadoop when scaling-up (System Speedup).

We see that Hadoop significantly improves its performance when scaling from Large (1844 s) to Extra Large (1296 s) instances. This is thanks to the better I/O subsystem of the Extra Large instance types. When scaling from Extra Large to Cluster Quadruple instances, we see no real improvement, since the I/O subsystem stays the same and only the CPU power increases. In contrast, HAIL benefits from additional and/or better CPU cores when scaling-up. Finally, we observe that the speedup of HAIL over Hadoop is even better when using physical nodes.

Cluster Scale-Out for Static Indexing

At this point, the reader might have already started wondering how well HAIL performs for larger clusters. To answer this question, we allocate one 50-nodes EC2 cluster and

one 100-nodes EC2 cluster. We use *cluster quadruple* (*cc1.4xlarge*) nodes for both clusters, because with this node type we experienced the lowest performance variability. In both clusters, we allocated two additional nodes: one to serve as Namenode and the other to serve as JobTracker. While varying the number of nodes per cluster, we keep the amount of data per node constant.

Figure 3.7 shows these results. We observe that HAIL achieves roughly the same upload times for the Synthetic dataset. For the UserVisits dataset, we see that HAIL improves its upload times for larger clusters. In particular, for 100 nodes, we see that HAIL matches the Hadoop upload times for the UserVisits dataset and outperforms Hadoop by a factor up to ~1.4 for the Synthetic dataset. More interesting, we observe that, in contrast to Hadoop, HAIL does not suffer from high performance variability [101]. Overall, these results show the efficiency of HAIL when scaling-out.

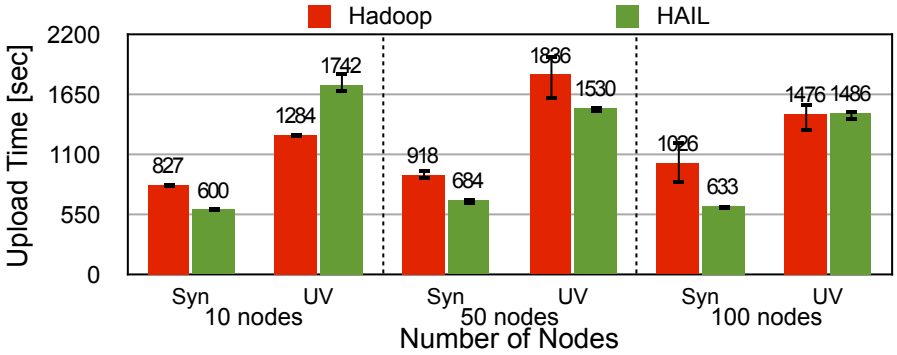


Figure 3.7: *Scale-out results*

3.9.4 Job Execution using Static Indexes

We now analyze the performance of HAIL when running MapReduce jobs. Our main goal for all these experiments is to understand how well HAIL can perform compared to the standard Hadoop MapReduce and Hadoop++ systems. With this in mind, we measure two different execution times. First, we measure the *end-to-end* job runtimes, which is the time a given job takes to run completely. Second, we measure the *record reader* runtimes, which is dominated by the time a given map task spends reading its input data. Recall that for these experiments, we disable the HailSplitting policy (presented in Section 3.7) in order to better evaluate the benefits of having several clustered indexes per dataset. We study the benefits of HailSplitting in Section 3.9.5.

Bob's Query Workload

For these experiments, Hadoop does not create any index; since Hadoop++ can only create a single clustered index, it creates one clustered index on sourceIP for all three

replicas, as two very selective queries will benefit from this; HAIL creates one clustered index for each replica: one on visitDate, one on sourceIP, and one on adRevenue.

Figure 3.8a shows the average end-to-end runtimes for Bob's queries. We observe that HAIL outperforms both Hadoop and Hadoop++ in all queries. For Bob-Q2 and Bob-Q3, Hadoop++ has similar results as HAIL since both systems have an index on sourceIP. However, HAIL still outperforms Hadoop++. This is because HAIL does not have to read any block header to compute input splits while Hadoop++ does. Consequently, HAIL starts processing the input dataset earlier, and hence, it finishes before.

Figure 3.8b shows the RecordReader times[11]. Once more again, we observe that HAIL outperforms both Hadoop and Hadoop++. HAIL is up to a factor 46 faster than Hadoop and up to a factor 38 faster than Hadoop++. This is because Hadoop++ is only competitive if it happens to hit the right index. As HAIL has additional clustered indexes (one for each replica), the likelihood to hit an index increases. Then, query runtimes for Bob-Q1, Bob-Q4, and Bob-Q5 are sharply improved over Hadoop *and* Hadoop++.

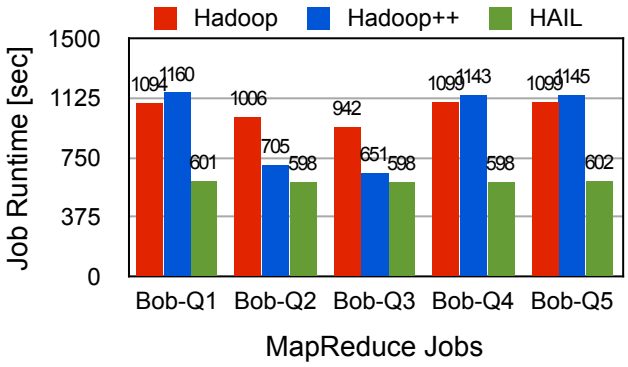
Yet, if HAIL allows map tasks to read their input data by more than one order of magnitude faster than Hadoop and Hadoop++, *why do MapReduce jobs not benefit from this?* To understand this, we estimate the overhead of the Hadoop MapReduce framework. We do this by considering an ideal execution time, i.e., the time needed to read all the required input data and execute the map functions over such data. We estimate the ideal execution time $T_{\text{ideal}} = \#MapTasks/\#ParallelMapTasks \times \text{Avg}(T_{\text{RecordReader}})$. Here, $\#ParallelMapTasks$ is the maximum number of map tasks that can be performed at the same time by all computing nodes. We define the overhead as $T_{\text{overhead}} = T_{\text{end-to-end}} - T_{\text{ideal}}$. We show the results in Figure 3.8c. We see that the Hadoop framework overhead is in fact dominating the total job runtime. This has many reasons. A major reason is that Hadoop was not built to execute very short tasks. To schedule a single task, Hadoop spends several seconds even though the actual task just runs in a few ms (as it is the case for HAIL). Therefore, reducing the number of map tasks of a job could greatly decrease the end-to-end job runtime. We tackle this problem in Section 3.9.5.

Synthetic Query Workload

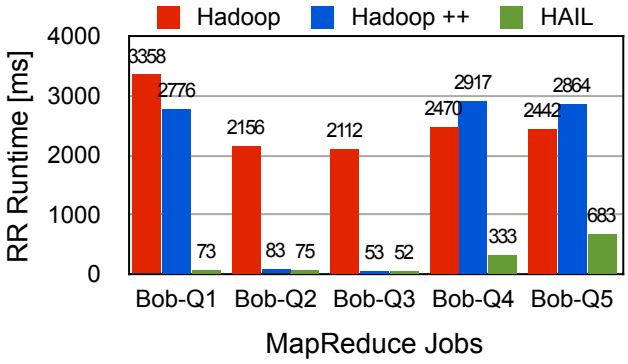
Our goal in this section is to study how query selectivities affect HAIL's performance. Recall that, for this experiment, HAIL *cannot* benefit from its different indexes: all queries filter on the same attribute. We use this setup to isolate the effects of selectivity.

We present the end-to-end job runtimes in Figure 3.9a and the record reader times in Figure 3.9b. We observe in Figure 3.9a that HAIL outperforms both Hadoop and Hadoop++. We see again that even if Hadoop++ has an index on the selected attribute, Hadoop++ runs slower than HAIL. This is because HAIL has a slightly different splitting phase than Hadoop++. Looking at the results in Figure 3.9b, the reader might think

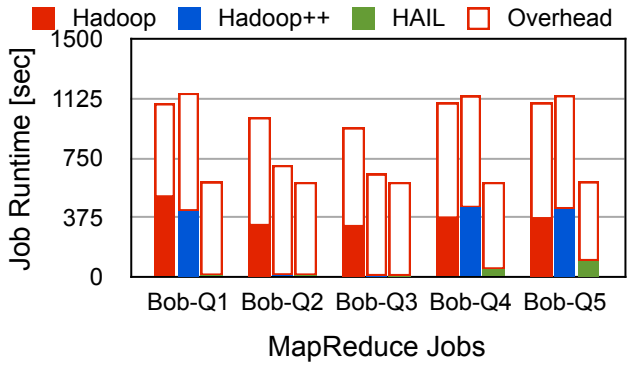
[11] This is the time a map task takes to read and process its input.



(a) *End-to-end job runtimes*

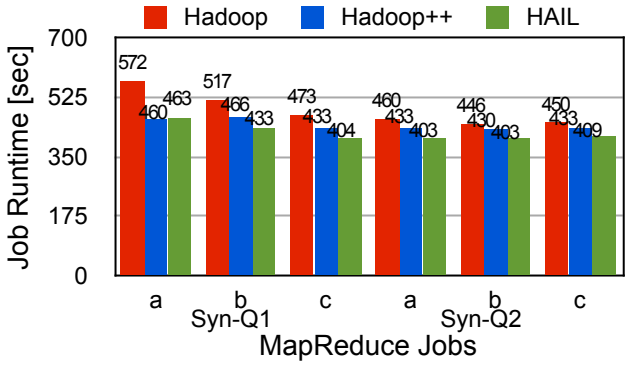


(b) *Average record reader runtimes*

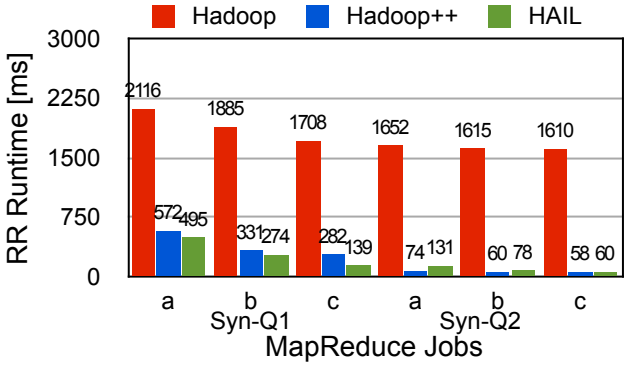


(c) *Hadoop scheduling overhead*

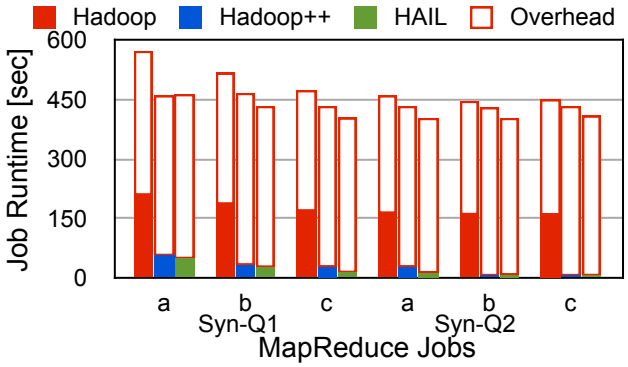
Figure 3.8: *Job runtimes, record reader times, and Hadoop MapReduce framework overhead for Bob's query workload filtering on multiple attributes*



(a) End-to-end job runtimes



(b) Average record reader runtimes



(c) Hadoop scheduling overhead

Figure 3.9: *Job runtimes, record reader times, and Hadoop scheduling overhead overhead for* `Synthetic` *query workload filtering on a single attribute*

that HAIL is better than Hadoop++ because of the PAX layout used by HAIL. However, we clearly see in the results for query Syn-Q1a that this is not true[12]. We observe that even in this case, HAIL is better than Hadoop++. The reason is that the index size in HAIL (2KB) is much smaller than the index size in Hadoop++ (304KB), which allows HAIL to read the index slightly faster. On the other hand, we see that Hadoop++ slightly outperforms HAIL for all three Syn-Q2 queries. This is because these queries are more selective and then, the random I/O-cost due to tuple reconstruction starts to dominate the record reader times.

Surprisingly, we observe that query selectivity does not affect end-to-end job runtimes (see Figure 3.9a) even if query selectivity has a clear impact on the RecordReader times (see Figure 3.9b). As explained in Section 3.9.4, this is due to the overhead of the Hadoop MapReduce framework. We clearly see this overhead in Figure 3.9c. In Section 3.9.5, we will investigate this in more detail.

Fault-Tolerance

In very large-scale clusters (especially on the Cloud), node failures are no more an exception but rather the rule. A big advantage of Hadoop MapReduce is that it can gracefully recover from these failures. Therefore, it is crucial to preserve this key property to reliably run MapReduce jobs with minimal performance impact under failures. In this section, we study the effects of node failures in HAIL and compare it with standard Hadoop MapReduce.

We perform these experiments as follows: (i) we set the expiry interval to detect that a TaskTracker or a datanode failed to 30 seconds, (ii) we chose a node randomly and kill all Java processes on that node after 50% of work progress, and (iii) we measure the slowdown as in [35], $slowdown = \frac{(T_f - T_b)}{T_b} \cdot 100$, where T_b is the job runtime without node failures and T_f is the job runtime with a node failure. We use two configurations for HAIL. First, we configure HAIL to create indexes on three different attributes, one for each replica. Second, we use a variant of HAIL, coined HAIL-1Idx, where we create an index on the same attribute for all three replicas. We do so to measure the performance impact of HAIL falling back to full scan for some blocks after the node failure. This happens for any map task reading its input from the killed node. In the case of HAIL-1Idx, all map tasks will still perform an index scan as all blocks have the same index.

Figure 3.10 shows the fault-tolerance results for Hadoop and HAIL. Overall, we observe that HAIL preserves the failover property of Hadoop by having almost the same slowdown. However, it is worth noting that HAIL can even improve over Hadoop. This is because HAIL can still perform an index scan when having the same index on all replicas (HAIL-1Idx). We clearly see this when HAIL creates the same index on all

[12]Recall that this query projects all attributes, which is indeed more beneficial for Hadoop++ as it uses a row layout.

replicas (HAIL-1Idx). In this case, HAIL has a lower slowdown since failed map tasks can still perform an index scan even after failure. As a result, HAIL runs almost as fast as when no failure occurs.

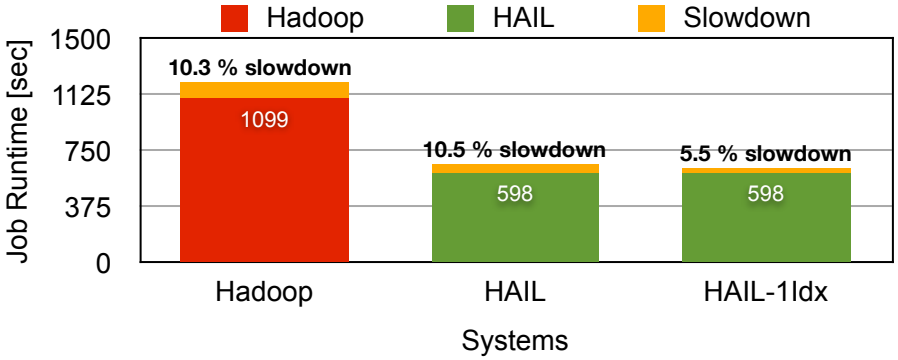


Figure 3.10: *Fault-tolerance results*

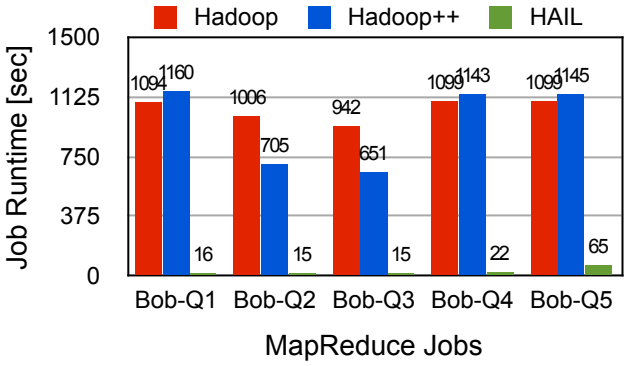
3.9.5 Impact of the HAIL Splitting Policy

We observed in Figures 3.8c and 3.9c that the Hadoop MapReduce framework incurs a high overhead in the end-to-end job runtimes. To evaluate the efficiency of HAIL to deal with this problem, we now enable the HailSplitting policy (described in Section 3.7) and run again the Bob and Synthetic queries on HAIL.

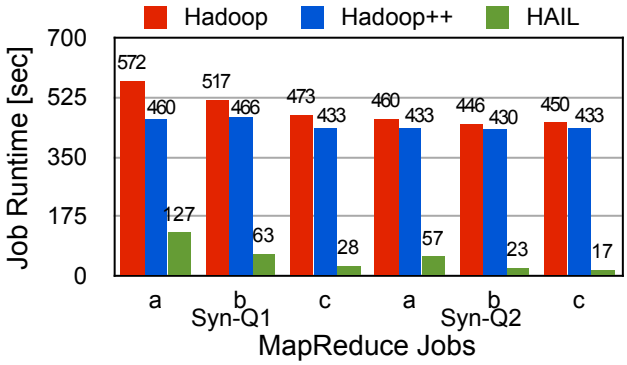
Figure 3.11 illustrates these results. We clearly observe that HAIL significantly outperforms both Hadoop and Hadoop++. We see in Figure 3.11a that HAIL outperforms Hadoop up to a factor of 68 and Hadoop++ up to a factor of 73 for Bob's workload. This is mainly because the HailSplitting policy significantly reduces the number of map tasks from 3,200 (which is the number of map tasks for Hadoop and Hadoop++) to only 20. As a result of HAIL Splitting policy, the scheduling overhead does not impact the end-to-end workload runtimes in HAIL (see Section 3.9.4). For the Synthetic workload (Figure 3.11b), we observe that HAIL outperforms Hadoop up to a factor of 26 and Hadoop++ up to a factor of 25. Overall, we observe in Figure 3.11c that using HAIL Bob can run all his five queries 39x faster than Hadoop and 36x faster than Hadoop++. We also observe that HAIL runs all six Synthetic queries 9x faster than Hadoop and 8x faster than Hadoop++.

3.10 Experiments on Adaptive Indexing

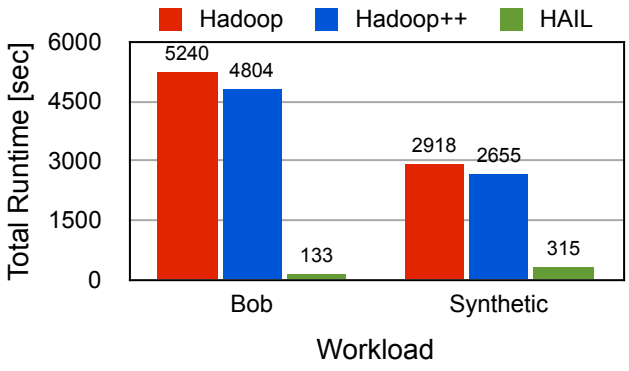
In the experiments of Section 3.9, we focused on the performance of HAIL with static indexing only, i.e., we deactivated HAIL adaptive indexing. For the following experi-



(a) Bob queries



(b) `Synthetic` queries



(c) Total Workload

Figure 3.11: *End-to-end job runtimes for Bob and Synthetic queries using the HailSplitting policy*

ments, we now focus on the evaluation of the HAIL adaptive indexing pipeline. Therefore, we examine the following scenario: *what happens if Bob did not create the right indexes upfront? Can Bob adapt his indexes to a new workload that he did not predict at upload time?* For this, we need to evaluate the efficiency of HAIL to adapt to query workloads and compare it with Hadoop and a version of HAIL, that only uses static indexing. We present a second wave of experiments for adaptive indexing to answer the following main questions:

1. What is the overhead of running the adaptive indexing techniques in HAIL?
2. How fast can HAIL adapt to changes in the query workload?
3. How well does each of the adaptive indexing strategies of HAIL allow MapReduce jobs to improve their runtime?

3.10.1 Hardware and Systems

Hardware

For our experiments on adaptive indexing, we use *Cluster-A* as described in Section 3.9.1 and an additional 4-node cluster (*Cluster-B*) in order to measure the influence of more efficient processors. In Cluster-B, each node has one 3.46 GHz Hexa Core Xeon X5690 processors; 20GB of main memory; one 278GB SATA hard disk (for the OS) and one 837GB SATA hard disk (for HDFS); two one Gigabit network cards.

Systems

Since the results from Section 3.9 clearly showed the high superiority of HAIL over Hadoop++, we decide to discard Hadoop++ and keep only Hadoop and HAIL with no adaptive indexing activated as baselines. For HAIL using the adaptive indexing techniques, we consider four different variants according to the offer rate ρ: HAIL (ρ = 0.1), HAIL (ρ = 0.25), HAIL (ρ = 0.5), and HAIL (ρ = 1). Notice that, HAIL with no adaptive indexing is the same as HAIL (ρ = 0). Still, as in previous sections, we assume that HAIL creates one index on sourceIP, one on visitDate, and one on adRevenue, for the UserVisits dataset. For the Synthetic dataset, we assume that HAIL does not create any index at upload time. Notice that, given the high Hadoop scheduling overhead, we observed in previous experiments, we increase the data block size to 256MB to decrease such overhead for Hadoop.

3.10.2 Datasets and Queries

Making use of the lessons learned from the first wave of experiments on static indexing, we slightly change our datasets and queries in order to stress and better evaluate HAIL

under bigger datasets and different query selectivities. We describe these changes in the following.

Datasets

We again use the web log dataset (UserVisits) but scaled it to 40GB per node, i.e., 400GB for Cluster-A and 160GB for Cluster-B. Additionally, the Synthetic dataset has now six attributes and a total size of $50GB$ per node, i.e., 500GB for Cluster-A and 200GB for Cluster-B. We generate the values for the first attribute in the range [1..10] and with an exponential repetition for each value, i.e., 10^{i-1} where $i \in [1..10]$. We generate the other five attributes at random. Then, we shuffle all tuples across the entire dataset to have the same distribution across data blocks.

Queries

For the UserVisits dataset we consider eleven queries, formulated as MapReduce jobs (JobUV1 – JobUV11), with a selection predicate on attribute *searchWord* and with a full projection (i.e., projecting all 9 attributes). The first four jobs JobUV1 – JobUV4 have a selectivity of 0.4% (1.24 million output records), and the remaining seven jobs (JobUV5 – JobUV11) have a selectivity of 0.2% (0.62 million output records). For the Synthetic dataset, we consider other eleven jobs (JobSyn1 – JobSyn11) with a full projection, but with a selection predicate on the first attribute. These jobs have a selectively of 0.2% (2.2 million output records). All jobs for both datasets select disjoint ranges to avoid caching effects. We report the average performance over three runs.

3.10.3 Adaptive Indexing Overhead for a Single Job

Adaptive indexing in HAIl always happens in the context of job execution since HAIL piggybacks adaptive indexing on MapReduce jobs. Therefore, the very first question that the reader might ask is *what is the additional runtime incurred by HAIL on MapReduce jobs?* We answer this question in this section. In particular, we want to measure the maximum overhead of adaptive indexing on job execution for a single job. The maximum overhead can be observed with the first job that starts indexing on a completely unindexed attribute. For this, we run job JobUV1 for UserVisits and job JobSyn1 for Synthetic and we assume that there is no block with a applicable index for jobs JobUV1 and JobSyn1. Figure 3.12 shows the job runtime for five variants of HAIL for the UserVisits dataset. In Cluster-A, we observe that HAIL has almost no overhead (only 1%) over HAIL ($\rho = 0$) when using an offer rate of 10% (i.e., $\rho = 0.1$). Notice that, HAIL ($\rho = 0$) has no matching index available and hence behaves like normal Hadoop with just the binary PAX layout to speed up the job execution. We can also see that the

new layout gives us an improvement of at most a factor of two in our experiments. Interestingly, we observe that HAIL is still faster than Hadoop with $\rho = 0.1$ and $\rho = 0.25$. Indeed, the overhead incurred by HAIL increases along with the offer rate used by HAIL. However, we observe that HAIL increases the execution time of JobUV1 by less than factor of two w.r.t. both Hadoop and HAIL without any indexing, even though all data blocks are indexed in a single MapReduce job. We especially observe that the overhead incurred by HAIL scales linearly with the ratio of indexed data blocks (i.e., with ρ), except when scaling from $\rho = 0.1$ to $\rho = 0.25$. This is because HAIL starts to be CPU bound only when offering more than 20% of the data blocks (i.e., from $\rho = 0.25$). This changes when running JobUV1 in Cluster-B. In these results, we clearly observe that the overhead incurred by HAIL scales linearly with ρ. We especially observe that HAIL benefits from using newer CPUs and have better performance than Hadoop for most offer rates. HAIL has only 4% overhead over Hadoop when having $\rho = 1$. Additionally, we can see that the adaptive indexing in HAIL incurs low overhead: from 10% (with $\rho = 0.1$) to 43% (with $\rho = 1$).

Figure 3.13 shows the job runtimes for Synthetic. Overall, we observe that the overhead incurred by HAIL continues to scale linearly with the offer rate. In particular, we observe that HAIL has no overhead over Hadoop in both clusters, except for HAIL ($\rho = 1$) in Cluster-A (where HAIL incurs a negligible overhead of ~3%). It is worth noting that when using newer CPUs (Cluster-B) adaptive indexing in HAIL has very low overhead as well: from 9% to only 23%.

From these results, we can conclude that HAIL can efficiently create indexes at job runtime while limiting the overhead of writing pseudo data blocks. We observe the efficiency of the lazy adaptive indexing mechanism of HAIL to adapt to users' requirements via different offer rates.

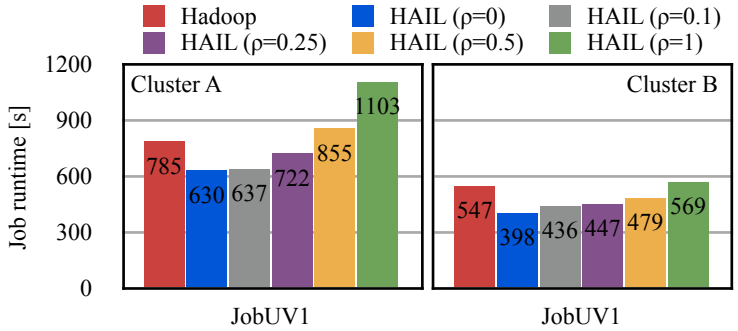


Figure 3.12: *HAIL Performance when running the first MapReduce job over UserVisits.*

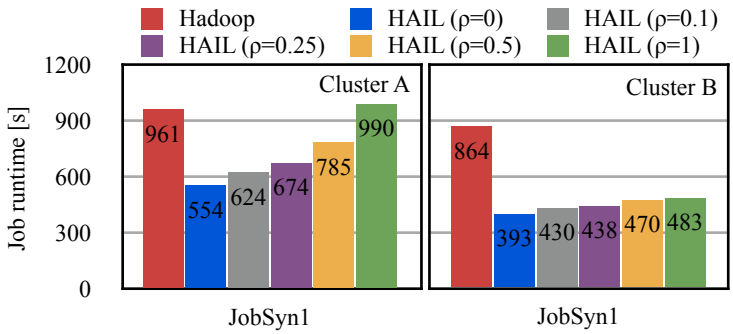


Figure 3.13: *HAIL Performance when running the first MapReduce job over Synthetic.*

3.10.4 Adaptive Indexing Performance for a Sequence of Jobs

We saw in the previous section that HAIL adaptive indexing techniques can scale linearly for a single job with the help of the offer rate. But, *which are the implications for a sequence of MapReduce jobs?* To answer this question, we run the sequence of eleven MapReduce jobs for each dataset.

Figures 3.14 and 3.15 show the job runtimes for the UserVisit and Synthetic datasets, respectively. Overall, we clearly see in both computing clusters that HAIL improves the performance of MapReduce jobs linearly with the number of indexed data blocks. In particular, we observe that the higher the offer rate, the faster HAIL converges to a complete index. However, the higher the offer rate, the higher the adaptive indexing overhead for the initial job (JobUV1 and JobSyn1). Thus, users are faced with a natural tradeoff between indexing overhead and the required number of jobs to index all blocks. But, it is worth noting that users can use low offer rates (e.g. $\rho = 0.1$) and still quickly converge to a complete index (e.g., after 10 job executions for $\rho = 0.1$). In particular, we observe that after executing only a few jobs, HAIL already outperforms Hadoop significantly. For example, let us consider the sequence of jobs on Synthetic using $\rho = 0.25$ on Cluster-B. Remember that for this offer rate, the overhead for the first job compared to HAIL without any indexing is relatively small (11%) while HAIL is still able to outperform Hadoop. With the second job, HAIL is slightly faster than the full scan, and the fourth job improves over full scan in HAIL by more than a factor of two and over Hadoop by more than a factor of five[13]. As soon as HAIL converges to a complete index, HAIL significantly outperforms full scan job execution in HAIL by up to a factor of 23 and Hadoop by up to a factor of 52. For the UserVisits dataset, HAIL outperforms unindexed HAIL by up to a factor of 24 and Hadoop by up to a factor of 32. Notice that, performing a full scan over Synthetic in HAIL is faster than in Hadoop, because HAIL reduces the size of this dataset when converting it to binary representation.

[13] Although HAIL is still indexing further blocks.

In summary, the results show that HAIL can efficiently adapt to query workloads with a very low overhead only for the very first job: the following jobs always benefit from the indexes created in previous jobs. Interestingly, an important result is that HAIL can converge to a complete index after running only a few jobs.

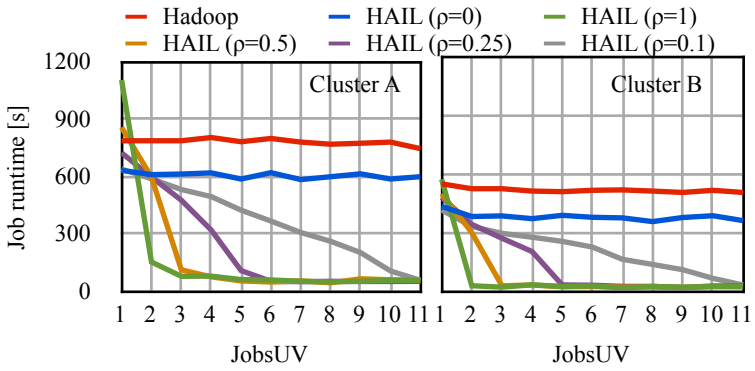


Figure 3.14: *HAIL performance when running a sequence of MapReduce jobs over UserVisits.*

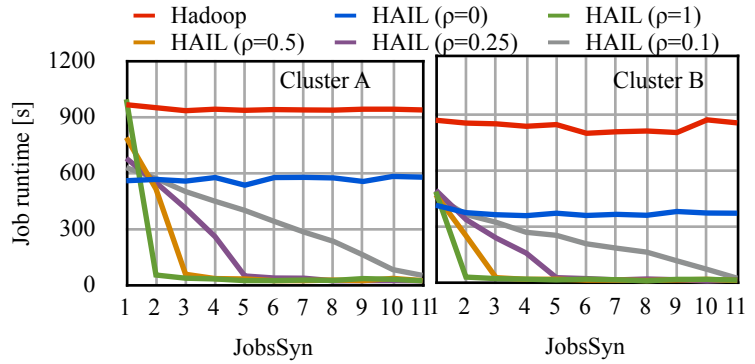


Figure 3.15: *HAIL performance when running a sequence of MapReduce jobs over Synthetic.*

3.10.5 Eager Adaptive Indexing for a Sequence of Jobs

We saw in the previous section that HAIL improves the performance of MapReduce jobs linearly with the number of indexed data blocks. Now, the question that might arise in the reader's mind is *can HAIL efficiently exploit the saved runtimes for further adaptive indexing?* To answer this question, we enable the eager adaptive indexing strategy in HAIL and run again all UserVisits jobs using an initial offer rate of 10%. In these experiments, we use Cluster-A and consider HAIL (without eager adaptive indexing enabled) with offer rates of 10% and 100% as baselines.

Figure 3.16 shows the result of this experiment. As expected, we observe that HAIL (eager) has the same performance as HAIL ($\rho = 0.1$) for JobUV1. However, in contrast to HAIL ($\rho = 0.1$), HAIL (eager) keeps its performance constant for JobUV2. This is because HAIL (eager) automatically increases ρ from 0.1 to 0.17 in order to exploit saved runtimes. For JobUV3, HAIL (eager) still keeps its performance constant by increasing ρ from 0.17 to 0.33. Now, even though HAIL (eager) increases ρ from 0.33 to 1 for JobUV4, HAIL (eager) now improves the job runtime as only 40% of the data blocks remain unindexed. As a result of adapting its offer rate, HAIL (eager) converges to a complete index only after 4 jobs while incurring almost no overhead over HAIL. From JobUV5, HAIL (eager) ensures the same performance as HAIL ($\rho = 1$) since all data blocks are already indexed, while HAIL ($\rho = 0.1$) takes 6 more jobs to converge to a complete index, i.e., to index all data blocks.

These results show that HAIL can converge even faster to a complete index, while still keeping a negligible indexing overhead for MapReduce jobs. Overall, these results demonstrate the high efficiency of HAIL (eager) to adapt its offer rate according to the number of already indexed data blocks.

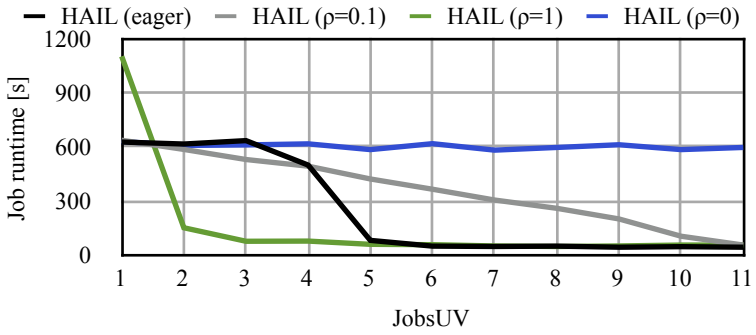


Figure 3.16: *Eager adaptive indexing vs. $\rho = 0.1$ and $\rho = 1$*

3.11 Conclusion

We presented HAIL (Hadoop Adaptive Indexing Library), a twofold approach towards zero-overhead indexing in Hadoop MapReduce. HAIL introduced two indexing pipelines that address two major problems of traditional indexing techniques. First, HAIL static indexing solves the problem of long indexing times, which had to be invested on previous indexing approaches in Hadoop. This was a severe drawback of Hadoop++ [35], which required expensive MapReduce jobs in the first place to create indexes. Second, HAIL adaptive indexing allows us to automatically adapt the set of available indexes to previously unknown or changing workloads at runtime with only minimal costs.

In more detail, HAIL static indexing allows users to efficiently build clustered indexes while uploading data to HDFS. Thereby, our novel concept of logical replication enables the system to create different sort orders (and hence clustered indexes) for each physical replica of a data set without additional storage overhead. This means that in a standard system setup, HAIL can create three different indexes (almost) for free as byproduct of uploading the data to HDFS. We have shown that HAIL static indexing also works well for a larger number of replicas. For example, in our experiments HAIL created six different clustered indexes in the same time HDFS took to just upload three byte-identical copies without any index.

With HAIL static indexing, we can already provide several matching indexes for a variety of queries. Still, our static indexing approach has similar limitations as other traditional techniques when it comes to unknown or changing workloads. The problem is that users have to decide upfront on which attributes to index and it is usually costly to revisit this choice in case of missing indexes. We solve this problem with HAIL adaptive indexing. Using this approach, our system can create missing but valuable indexes automatically and incrementally at job execution time. In contrast to previous work, our adaptive indexing technique again focuses on indexing at minimal expense.

We have experimentally compared HAIL with Hadoop as well as Hadoop++ using different datasets and different clusters. The results demonstrated the high superiority of HAIL. For HAIL static indexing, our experiments showed that we typically create a win-win situation: e.g., users can upload datasets up to 1.6x faster than Hadoop (despite the additional indexing effort!) and run jobs up to 68x faster than Hadoop.

Our second set of experiments demonstrated the high efficiency of HAIL adaptive indexing to create clustered indexes at job runtime and adapt to users' workloads. In terms of indexing effort, HAIL adaptive indexing has a very low overhead compared to HAIL full scan (which is already 2x faster than Hadoop full scan). For example, we observed 1% runtime overhead for the UserVisits dataset when using an offer rate of 10% and only for the very first job. The following jobs already run faster than the full scan in HAIL, e.g. ~2 times faster from the fourth job, with an offer rate of 25%. The results also show that, even for low offer rates, our approach quickly converges to a complete index after running only a few number of MapReduce jobs (e.g., after 10 jobs with an offer rate of 10%). In terms of job runtimes, HAIL adaptive indexing improves performance dramatically. For a sequence of previously unseen jobs on unindexed attributes, runtime improved by up to a factor of 24 over HAIL without adaptive indexing and a factor of 52 over Hadoop.

Chapter 4

A Comparison of Adaptive Radix Trees and Hash Tables

4.1 Introduction

In the last decade the amount of main memory in commodity servers has constantly increased — nowadays, servers with terabytes of main memory are widely available at affordable prices. This memory capacity makes it possible to store most databases completely in main memory, and has triggered a considerable amount of research and development in the area. As a result, new high performance index structures for main memory databases are emerging to challenge hash tables — which have been widely used for decades due to their good performance. A recent and promising structure in this domain is the adaptive radix tree **ARTful** [76], which we call just **ART** from now on. This recent data structure was reported to be significantly faster than existing data structures like FAST [71] and the cache-conscious B^+-tree CSB^+ [95]. Moreover, it was also reported that only a hash table is competitive to ART. Thus, ART was reported to be as good as a hash table while also supporting range queries. Nonetheless, three important details were not considered during the experimental comparison of ART with other data structures that we would like to point out:

1. To the best of our knowledge, the first adaptive radix tree in the literature was the **Judy Array** [12], which we simply call **Judy** from now on. A comparison between ART and Judy was not offered by the original study [76], but given the strong similarities between the two structures, we think that there ought to be a comparison between the two.

2. The hash table used by the authors of ART for the experimental comparison was a chained hash table. This kind of hashing became popular for being, perhaps, the very first iteration of hashing, appearing back in the 50s. Nevertheless, it is still

popular for being the default method in standard libraries of popular programming languages like `C++` and `Java`. However, nowadays chained hashing could be considered suboptimal in performance and space because of the (potentially) high overhead due to pointers, and other hashing schemes are preferred where performance is sought — like quadratic probing [25, 74]. Moreover, rather recently, Cuckoo hashing [90] has seen a considerable amount of research [86], and it has been reported to be competitive [90] in practice to, for example, quadratic probing. Thus, we believe that the experimental comparison between ART and hashing was not complete. This brings us to our last point, hash functions.

3. Choosing a hash function should be considered as important as choosing a hashing scheme (table), since it highly determines the performance of the data structure. Over decades there has been a considerable amount of research focusing *only* on hash functions — sometimes on their theoretical guarantees, some other times on their performance in practice. The authors of ART chose Murmur [7] as a hash function — presumably due to the robustness (ability of shuffling data) shown in practice, although nothing is known about its theoretical guarantees, to the best of our knowledge. In our own experiments we noticed that Murmur hashing is indeed rather robust, but for many applications, or at least the ones considered by the authors of ART, that much robustness could be seen as an overkill. Thus, it is interesting to see how much an easier (but still good) hash function changes the picture.

4.1.1 Contributions

The main goal of this chapter is to *extend* the experimental comparison offered by the authors of ART by providing a thorough experimental evaluation of ART against Judy, two variants of quadratic probing, and three variants of Cuckoo hashing. We provide different variants of the same hashing scheme because some variants are tuned for performance, while other are tuned for space efficiency. However, it is *not* our intention to compare ART against structures already considered (covered) in the original ART paper [76] *again*. Consequently, just as in the micro-benchmarks presented in [76], we only focus on keys from an integer domain. In this regard, we would like to point out that the story could change if keys were arbitrary strings of variable size. However, a thorough study on indexing strings in main memory deserves a study on its own, and is thus out of scope of this work.

For each considered hash table we test two different hash functions, Murmur hashing [7], for reference, completeness, and compatibility with the original study [76], and the well-known multiplicative hashing [25, 74, 30] — which is perhaps the easiest-to-compute hash function with still good theoretical guarantees. Our experiments strongly indicate that neither ART nor Judy are competitive in terms of performance

to well-engineered hash tables, and in the case of ART, sometimes not even in terms of space. For example, for one billion indexed keys, *one* non-covering variant of Cuckoo hashing is *at least* 4.8× faster for insertions than ART, *at least* 2.8× faster for lookups, and it sometimes requires just half the space of ART, see Figures 4.2, 4.3, and 4.4. We also hope to convey more awareness as of how important it is to consider newer hashing approaches (hashing schemes *and* hash functions) when throughput performance and/or memory efficiency are crucial.

The remainder of the chapter is organized as follows. In Section 4.2 we give a general description of adaptive radix trees — highlighting key similarities and differences between ART and Judy. In Section 4.3 we give a detailed description of the hashing schemes and hash functions used in our study. In 4.4 we present our experiments. Finally, in Section 4.5 we close the chapter with our conclusions. Our presentation is given in a self-contained manner.

4.2 Radix Trees

In this section we give a general description of the (adaptive) radix trees included in our study. In general, a radix tree [74] (also called prefix tree, or trie) is a data structure to represent ordered associative arrays. In contrast to many other commonly used tree data structures such as binary search trees or standard B-Trees, nodes in radix trees do not cover complete keys. Instead, nodes in a radix tree represent partial keys, and only the full path from the root to a leaf describes the complete key corresponding to a value. Furthermore, operations on radix trees do not perform comparisons on the keys in the nodes but rather, operations like looking up for a key work as follows: (1) Starting from the root, and for each inner node, a partial key is extracted on each level. (2) This partial key determines the branch that leads to the next child node. (3) The process repeats until a leaf or an empty branch is reached. In the first case, the key is found in the tree, in the second case, it is not.

In a radix tree, the length of the partial keys determines the fan-out of the nodes because *for each* node there is *exactly one* branch *for each* possible partial key. For example, let us assume a radix tree that maps 32-bit integer keys to values of the same type. If we chose each level to represent a partial key of one byte, this results in a 4-level radix tree having a fan-out of 256 branches per node. Notice that, for all levels, all keys under a certain branch have a common prefix and unpopulated branches can be omitted. For efficiency, nodes in a radix tree are traditionally implemented as arrays of pointers to child nodes. When interpreting the partial key as an index to the array of child pointers, finding the right branch on a node is as efficient as one array access. However, this representation can easily lead to excessive memory consumption and bad cache utilization for data distributions that lead to many sparsely populated branches, such as uniform random distribution. In the context of our example, each node would

contain an array of 256 pointers, even if only a single child node exists. This easily leads to a high memory overhead. This is the reason why radix trees have usually been considered as a data structure that is only suitable for certain use cases, e.g., textual data, and not for general purposes. For example, radix trees are often used for prefix search on skewed data, e.g. in dictionaries. Still, radix trees have many interesting and useful properties: (1) Shape depends *only* on the key space and length of partial keys, but not on the contained keys or their insertion order. (2) Do not require rebalancing operations. (3) Establish an order on the keys and allow for efficient prefix lookups. (4) Allow for prefix compression on keys.

The aforementioned memory overheads that traditional radix trees potentially suffer from leads to the natural question of whether the situation can be somehow alleviated. To the best of our knowledge, the **Judy Array** [12] is the first variant of a radix tree that adaptively varies its node representation depending on the key distribution and/or cardinality of the contained data. Judy realizes adaptivity by introducing several compression techniques. These techniques prevent excessive memory footprints on sparsely populated trees, and improve cache utilization. According to the inventors, Judy offers performance similar to hash maps, supports efficient range queries like a (comparison-based) tree structures, and prefix queries like traditional radix trees. All this while also providing better memory efficiency than *all* aforementioned data structures.

Very recently, in 2013, the **ARTful** index [76] was introduced as a new index structure for main memory database systems. ART is also an adaptive radix tree, and has similar purposes as Judy — high performance at low memory cost. However, unlike Judy, ART was not designed as an associative array, but rather ART is tailored towards the use case of an index structure for a database system — on top of a main memory storage. In the following we will discuss both, Judy arrays and ART, highlighting their similarities and differences.

4.2.1 Judy Array

Judy array can be characterized as a variant of a 256-way radix tree. There are three different types of Judy arrays: (1) **Judy1**: A bit array that maps integer keys to true or false and hence can be used as a set. (2) **JudyL**: An array that maps integer keys to integer values (or pointers) and hence can be used as an integer to integer map. (3) **JudySL**: An array that maps string keys of arbitrary length to integer values (or pointers) and hence can be used as a map from byte sequences to integers.

For a meaningful comparison with the other data structures considered by us, we will only focus on **JudyL** for the remainder of this work, and thus consider Judy and JudyL as synonyms from now on. In the following, we give a brief overview of the most important design decisions that affect the performance and memory footprint of JudyL.

The authors of Judy observed that cache misses have a tremendous impact on the performance of any data structure, up to the point where cache miss costs dominate the

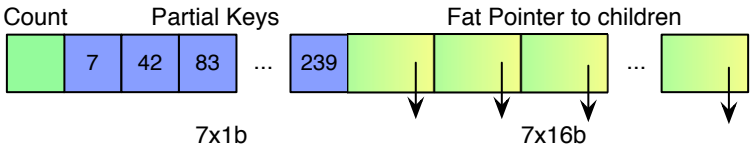
runtime. Hence, to offer high performance across different data distributions, one major design concern of Judy was to avoid cache-line fills (which can result in cache misses) at almost any cost. Observe that the maximum number of cache-line fills in a radix tree is determined by the number of tree levels. Moreover, the maximum number of tree levels is determined by the maximum key length divided by the partial key size. For every tree level in a standard radix tree, we need to access exactly one cache line that contains the pointer to the child node under the index of that partial key.

Judy addresses memory overheads of traditional radix trees under sparse data distributions and simultaneously avoids cache-line fills through a combination of more than 20 different compression techniques. We can roughly divide these techniques into two categories: *horizontal compression* and *vertical compression*. We only give a brief overview of the most important ideas in Judy. A full description of the ideas can be found in [12].

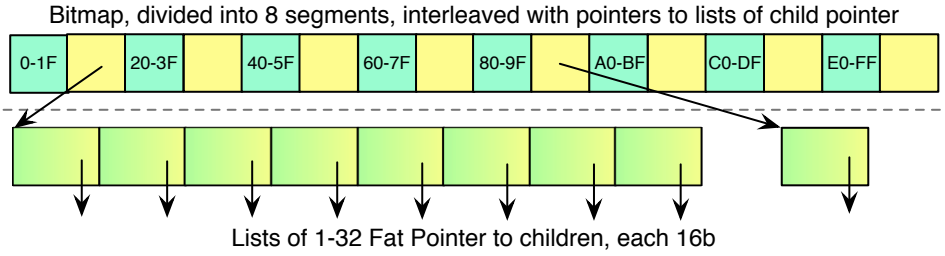
Horizontal compression

Here the problem of many large, but sparsely populated nodes, is addressed. The solution offered by Judy is to adapt node sizes dynamically and individually with respect to the actual population of the subtree underneath each node. Hence, Judy can compress unused branches out of nodes. For example, Judy may use smaller node types that have e.g., only seven children. However, in contrast to uncompressed (traditional) radix nodes with 256 branches, the slots in compressed nodes are not directly addressable through the index represented by the current partial key. Consequently, compressed nodes need different access methods, such as comparisons, which can potentially lead to multiple additional cache-line fills. Judy minimizes such effects through clever design of the compressed nodes. There are two basic types of horizontally compressed nodes in Judy: linear nodes and bitmap nodes, which we briefly explain: (1) A **linear node** is a space efficient implementation for nodes with very small number of children. In Judy, the size of linear nodes is limited to one cache line.[1] Linear nodes start with a sorted list that contains only the partial keys for branches to existing child nodes. This list is then followed by a list of the corresponding pointers to child nodes in the same order, see Figure 4.1a. To find the child node under a partial key, we search the partial key in the list of partial keys and follow the corresponding child pointer if the partial key is contained in the list. Hence, linear nodes are similar to the nodes in a B-tree w.r.t. structure and function. (2) A **bitmap node** is a compressed node that uses a bitmap of 256 bits to mark the present child nodes. This bitmap is divided into eight 32-bit segments, interleaved with pointers to the corresponding lists of child pointers, see Figure 4.1b. Hence, bitmap nodes are the only structure in Judy that involve up to two cache-line fills. To

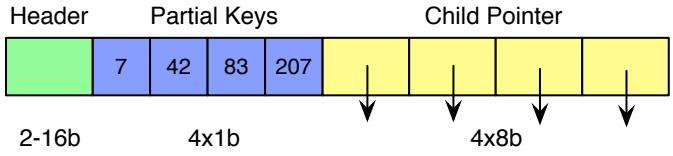
[1] Judy's 10-year-old design assumes cache-line size of 16 machine words, which is not the case for modern main-stream architectures.



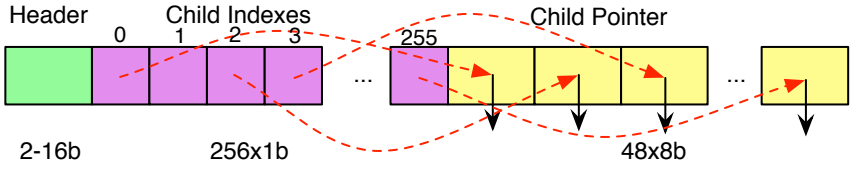
(a) JudyL Linear Node



(b) JudyL Bitmap Node



(c) ART Node4



(d) ART Node48

Figure 4.1: *Comparison of node types (64-bit).*

lookup the child under a partial key, we first detect if the bit for the partial key is set. In that case, we count the leading set bits in the partial bitmap to determine the index of the child pointer in the pointer list. Bitmap nodes are converted to uncompressed nodes (256 pointers) as soon as the population reaches a point where the additional memory usage amortizes.[2]

To differentiate between node types, Judy must keep some meta information about every node. In contrast to most other data structures, Judy does not put meta information in the header of each node, because this can potentially lead to one additional cache-line fill per access. Instead, Judy use what the authors of Judy call *Judy pointers*. These pointers are fat pointers of two machine words size (i.e., 128bit on 64bit architectures) that combine the address of a node with the corresponding meta data, such as: node type, population count, and key prefix. Judy pointers avoid additional cache-line fills by densely packing pointers with the meta information about the object they point to.

Vertical compression

In Judy arrays vertical compression is mainly achieved by skipping levels in the tree when an inner node has only one child. In such cases, the key prefix corresponding to the missing nodes is stored as decoding information in the Judy pointer. This kind of vertical compression is commonly known in the literature as **path compression**. Yet another technique for vertical compression is **immediate indexing**. With immediate indexing, Judy can store values immediately inside of Judy pointers instead of introducing a whole path to a leaf when there is no need to further distinguish between keys.

4.2.2 ART

This newer data structure shares many ideas and design principles with Judy. In fact, ART is also a 256-radix tree that uses (1) different node types for horizontal compression, and (2) vertical compression also via path compression and immediate indexing — called lazy expansion in the ART paper. However, there are two major differences between ART and Judy: (1) There exist four different node types in ART in contrast to three types in Judy. These node types in ART are labeled with respect to the maximum amount of children they can have: `Node4`, `Node16`, `Node48`, and the uncompressed `Node256`. Those nodes are also organized slightly different than the nodes in Judy. For example, the meta information of each node is stored in a header instead of a fat pointer (Judy pointer). Furthermore, ART nodes take into account the latest changes and features in hardware design, such as SIMD instructions to speedup searching in the linearly-organized `Node16`. It is worth pointing out that we cannot find any consideration of that kind of instructions in the decade-old design of Judy. (2) ART was

[2]The concrete conversion policies between nodes types are out of the scope of this work.

designed as an index structure for a database, whereas Judy was designed as a general purpose associative array. As a consequence, Judy owns its keys and values and covers them both inside the structure. In contrast to that, ART does not necessarily cover full keys or values (e.g., when applying vertical compression) but rather stores a pointer (as value) to the primary storage structure provided by the database — thus ART is primarily used as a non-covering index. At lookup time, we use a given key to lookup for the corresponding pointer to the database store containing the complete ⟨key, value⟩ pair.

Finally, and for completeness, let us give a more detailed comparison of the different node types between Judy and ART. `Node4` and `Node16` of ART are very much comparable to a linear node in Judy except for their sizes, see Figures 4.1a and 4.1c. `Node16` is just like a `Node4` but with 16 entries. Uncompressed `Node256` of ART is the same as the uncompressed node in Judy, and thus also as in plain radix trees. `Node48` of ART consists of a 256-byte array (which allows direct addressing by a partial key) follow by an array of 48 child pointers Up to 48 locations of the 256-byte array can be occupied, and each occupied entry stores the index in the child pointer array holding the corresponding pointer for the partial key, see Figure 4.1d. `Node48` of ART and the bitmap node of Judy fill in the gap between small and large nodes.

4.3 Hash Tables

In this section we elaborate on the hashing schemes and the hash functions we use in our study. In short, the hashing schemes are (1) the well-known quadratic probing [74, 25], and (2) Cuckoo hashing [90]. As for hash functions we use 64-bit versions of (1) Murmur hashing [7], which is the hash function used for the original study [76], and (2) the well-known, and somewhat part of the hashing folklore, multiplicative hashing [25, 74, 30]. In the rest of this section we consider each of these parts in turn.

4.3.1 Quadratic probing

Quadratic probing is one of the best-known open-addressing schemes for hashing. In open-addressing, every hashed element is contained in the hash table itself, i.e. every table entry contains either an element or a special key denoting that the corresponding location is unoccupied. The hash function in quadratic probing has the following form:

$$h(x, i) = (h'(x) + c_1 \cdot i + c_2 \cdot i^2)$$

where i represents the i-th probed location, h' is an auxiliary hash function, and $c_1 \geq 0$, $c_2 > 0$ are auxiliary constants.

What makes quadratic probing attractive and popular is: (1) It is easy to implement. In its simplest iteration, the hash table consists of a single array only. (2) In the particular case that the size of the hash table is a power of two, it can be proven that quadratic

probing will examine *every* single location of the table in the worst case [25]. That is, as long as there are available slots in the hash table, this particular version of quadratic probing will *always* find them, at the expense of an increasing number of probes.

Quadratic probing is, however, not bulletproof. It is known that it could suffer from *secondary clustering*. This means that if two different keys collide in the very first probe, they will also collide in all sub-sequent probes. Thus, choosing a good hash function is of primary concern.

The implementations of quadratic probing used in this study are the ones provided by Google dense and sparse hashes [48]. These C++ implementations are well-engineered for general purposes[3], and are readily available. Furthermore, they are designed to be used as direct replacements of `std::unordered_map`[4]. This reduces integration in existing code to the minimal effort. These Google hashes come in two variants, dense and sparse. The former is optimized for (raw) performance, potentially sacrificing space, while the latter is optimized for space while potentially sacrificing performance. In this study we consider both variants, and, for simplicity, we will refer to Google dense and sparse hashes simply as **GHFast** (for performance) and **GHMem** (for memory efficiency) respectively.

4.3.2 Cuckoo hashing

Cuckoo hashing is a relatively new open-addressing scheme [90], and somewhat still not well-known. The original (and simplest) version of Cuckoo hashing works as follows: There are two hash tables T_0, T_1, each one having its own hash function h_0, h_1. Every inserted element x is stored at either $T_0[h_0(x)]$ or $T_1[h_1(x)]$ but *never* in both. When inserting an element x, location $T_0[h_0(x)]$ is first probed, if the location is empty, x is store there, otherwise, x kicks out the element y already found at that location, x is stored there, but now y is out of the table and has to be inserted, so location $T_1[h_1(y)]$ is probed. If this location is free, y is stored there, otherwise y kicks out the element therein, and we repeat: in iteration $i \geq 0$, location $T_j[h_j(\cdot)]$ is probed, where $j = i \mod 2$. In the end we hope that *every* element finds its own "nest" in the hash table. However, it may happen that this process enters a loop, and thus a place for each element is never found. This is dealt with by performing only a fixed amount of iterations. Once this limit is achieved, a complete rehash is performed by choosing two new hash functions. How this rehash is done is a design decision: it is not necessary to allocate new tables, one can reuse the already allocated space by deleting and reinserting every element already found in the table. However, if the set of elements to be contained in the table increases over time, then perhaps increasing the size of the table when the rehash happens is a better

[3] This does not necessarily imply optimal performance in certain domains. That is, it is plausible that specialized implementations could be faster.

[4] Whose implementation happens to be hashing with chaining just as the ones used in the original ART paper.

policy for future operations. It has been empirically observed [90, 46] that in order to work, and obtain good performance, the load factor of Cuckoo hashing should stay slightly below 50%. That is, it requires at least twice as much space as the cardinality of the set to be indexed. Nevertheless, it has also been observed [46] that this situation can be alleviated by generalizing Cuckoo hashing to use more tables $T_0, T_1, T_2 \dots T_k$, each having its own hash function h_k, $k > 1$. For example, for $k = 4$ the load factor (empirically) increases to 96%, at the expense of performance. Thus, as for the Google hashes mentioned before, we can consider two versions of Cuckoo hashing, one tuned for performance, when $k = 2$, and the other tuned for space-efficiency, when $k = 4$.

Finally, we include in this study yet another variant of Cuckoo hashing. This variant allows more than one element per location in the hash table [33], as opposed to the original Cuckoo hashing where *every* location of the hash table holds *exactly* one element. In this other variant, we use only two tables T_0, T_1, just as the original Cuckoo hashing, but every location of the hash table is a bucket of size equal to the cache-line size, 64 bytes for our machines. This variant works essentially as the original Cuckoo hashing, when inserting an element x, it checks whether there is a free slot in the corresponding bucket, if yes, then x is inserted, otherwise a random element y of that bucket is kicked out, x is left in its place, and we start the Cuckoo cycles. We decided to include this variant of Cuckoo hashing because when a location of the hash table is accessed, this location is accessed through a cache line, so by aligning these buckets to cache lines boundaries we hope to have better data locality for lookups, at the expense of making more comparisons to find the given element in the bucket. This comparisons happen, nevertheless, only among elements that are already on cache (close to the processor).

For simplicity, we will refer to standard Cuckoo hashing using two and four tables as **CHFast** (for performance) and **CHMem** (for memory efficiency) — highlighting similarities of each of these hashes with Google's GHFast and GHMem, respectively, mentioned before. The last variant of Cuckoo hashing described above, using 64-byte buckets, will be simply referred to as **CHBucket**.

Let us now explain the reasons behind our decision to include Cuckoo hashing in our study. (1) For lookups, traditional Cuckoo hashing requires at most two tables accesses, which is in general optimal among hashing schemes using linear space. In particular, it is independent of the current load factor of the hash table — unlike other open-addressing schemes, like quadratic probing. (2) It has been reported to be competitive with other good hashing schemes, like quadratic probing or double hashing [90], and (3) It is easy to implement.

Like quadratic probing, Cuckoo hashing is not bulletproof either. It has been observed [90] that Cuckoo hashing is sensitive to what hash functions are used [32]. With good (and robust) hash functions, the performance of Cuckoo hashing is good, but with hash functions that are not as robust, performance deteriorates. We will see this effect in our experiments.

4.3.3 Hash functions

Having explained the hashing schemes used in our study, we now turn our attention to the hash functions used. We pointed out before that both used hashing schemes are highly dependent on the hash functions used. For our study we have decided to include two different hash functions: (1) MurmurHash64A [7] and the well-known multiplicative hashing [74]. The first one has been reported to be efficient and robust[5] [7], but more importantly, it is included here because it is the hash function that was used in the original ART paper [76], and we wanted to make our study equivalent.

The second hash function, multiplicative hashing, is *very* well known [25, 74, 30], and it is given here:

$$h_z(x) = (x \cdot z \mod 2^w) \text{ div } 2^{w-d}$$

where x is a w-bit integer in $\{0, \ldots, 2^w - 1\}$, z is an odd w-bit integer in $\{1, \ldots, 2^w - 1\}$, the hash table is of size 2^d, and the div operator is defined as: $a \text{ div } b = \lfloor a/b \rfloor$. What makes this hash function highly interesting is: (1) It can be implemented extremely efficiently by observing that the multiplication $x \cdot z$ is per se already done modulo 2^w, and the operator div is equivalent to a right bit shift by $w - d$ positions. (2) It has also theoretical guarantees. It has been proven [30] that if $x, y \in \{0, \ldots, 2^w - 1\}$, with $x \neq y$, and if $z \in \{1, \ldots, 2^w - 1\}$ chosen uniformly at random, then the collision probability is:

$$Pr[h_z(x) = h_z(y)] \leq \frac{2}{2^d} = \frac{1}{2^{d-1}}$$

This probability is twice as large as the ideal probability that, for a hash function, *every* location of the hash table is equally likely. This also means that the family of hash functions $H_{w,d} = \{h_z \mid 0 < z < 2^w \text{ and } z \text{ odd}\}$ is the perfect candidate for simple and somewhat robust hash functions.

As we will see in our experiments, MurmurHash64A is indeed more robust than multiplicative hashing, but this robustness comes at a very high performance degradation. In our opinion multiplicative hashing showed to be robust enough in all our scenarios. From now on, and for simplicity, we will refer to MurmurHash64A simply as **Murmur** and to multiplicative hashing just as **Simple**.

4.4 Main Experiments

In this section we experimentally confront the adaptive radix tree ART [76] with all other structures previously mentioned: (1) Judy [12], which is another kind of adaptive radix tree highly space-efficient — discussed in Section 4.2 and (2) Quadratic probing [48] and Cuckoo hashing [90] — discussed in Section 4.3. The experiments are mainly divided into three parts.

[5] Although, to the best of our knowledge, no theoretical guarantee of this has been shown.

In 4.4.3 we first show experiments comparing ART only against Cuckoo hashing under the following metrics: **insertion throughput**, **point query throughput**, and **memory footprint**. The reason why we only compare ART against Cuckoo hashing is the following: ART, as presented and implemented in [76] was designed as a non-covering indexing data structure for databases. That is, as mentioned in Section 4.2.2, ART will index a set of ⟨key, value⟩ pairs already stored and provided by a database. Thus, ART will, in general, neither cover the key nor the value[6], but it will rather use the key to place a pointer to the location in the database where the corresponding pair is stored. Thus, when looking up for a given key, ART will find the corresponding pointer (if previously inserted) and then follow it to the database store to retrieve the corresponding ⟨key, value⟩ pair. The semantics of the freely available implementations of Judy arrays [12] and Google hashes [48] are that of a map container (associative array), i.e., self-contained general-purpose indexing data structures (covering both the key and the value). We could have compared ART against these implementations as well but we think the comparison is slightly unfair, since inserting a pointer in those implementations will still cover the key, and thus the data structure will per se require more space. This is where our own implementation of Cuckoo hashing enters the picture. For the experiments presented in 4.4.3, Cuckoo hashing uses the key to insert a pointer to the database store, exactly just as ART — making an apple-to-apple comparison. For these experiments we assume that we *only* know upfront the number n of elements to be indexed. This is a valid assumption since we are interested in indexing a set of elements already found in a database. With this in mind, the hash tables are prepared to be able to contain at least n elements. Observe that the ability of pre-allocate towards certain size is (trivially) inherent to hash tables. In contrast, trees require knowledge not only about their potential sizes, but also the actual values and dedicated (bulk-loading) algorithms. This kind of workload (4.4.3) can be considered static, like in an OLAP scenario.

In 4.4.4 we test the structures considered in 4.4.3 under TPC-C-like dynamic workloads by mixing insertions, deletions, and point queries. This way we simulate an OLTP scenario. The metric here is only **operation throughput**. In this experiment the data structures assume nothing about the amount of elements to be inserted or deleted, and thus we will be able to observe how the structures perform under dynamic workloads. In particular, we will observe how the hash tables handle growth (rehashing) over time.

In 4.4.5 we consider ART as a standalone data structure, i.e., a data structure used to store (cover) ⟨key, value⟩ pairs, and we compare it this time against Judy array, Google hashes, and Cuckoo hashing under the same metrics as before. As ART was not originally designed for this purpose, we can go about two different ways: (1) We endow ART with its own store and we use the original implementation of ART, or (2) We endow ART with explicit leaf nodes to store the ⟨key, value⟩ pairs. We actually implemented

[6]The only exception to this happens when the key equals the value, effectively making ART a set container.

both solutions but we decided to keep for this study only the first one. The reason for this is that for the second option we observed mild slowdowns for insertions and mild speedups for lookups, but space consumption increases significantly as the size of the set to be contained also increases. The reason for this is that a leaf node requires more information (the header) than simply storing only ⟨key, value⟩ pairs in a pre-allocated array. For these experiments, the hash tables and the store of ART are prepared to be able to store at least n elements, where n is the number of elements to be indexed.

4.4.1 Experimental setup

All experiments are **single-threaded**. The implementations of ART, Judy arrays, and Google hashes are the ones freely available [77, 12, 48]. No algorithmic detail of those data structures was touched except that we implemented the missing range-query support in ART. All implementations of Cuckoo hashing are our own. All experiments are in **main memory** using a **single core** (one NUMA region) of a dual-socket machine having two hexacore Intel Xeon Processors X5690 running at 3.47 GHz. The L1 and L2 cache sizes are 64 and 256KB respectively per core. The L3 cache is shared and has a size of 12MB. The machine has a total of 192GB of RAM running at 1066 MHz. The OS is Linux (3.4.63, 64-bit) with a default page size of 4KB. All programs are implemented in C/C++ and compiled with the Intel `icc-14` with optimization `-O3`.

4.4.2 Specifics of our workloads

In our experiments we include two variants of ART, let us call them unoptimized and optimized. The difference between the two of them is that the latter applies path compression (one of the techniques for vertical compression mentioned in Section 4.2.2) to the nodes. By making (some) paths in the tree shorter, there is hope that this will decrease space and also speedup lookups. However, path compression clearly incurs into more overheads at insertion time, since at that time it has to be checked whether there is opportunity for compression and then it must be performed. In our experiments we denote the version of ART with path compression by **ART-PC**, and the one without it simply by **ART**. From now on, when we make remarks about ART, those remarks apply to *both* variants of ART, unless we say otherwise and point out the variant of ART we are referring to.

In the original ART paper [76] *all* micro-benchmarks are performed on 32-bit integer keys because some of the structures therein tested are 32-bit only. The authors also pointed out that for such short keys, path compression increases space instead of reducing it, and thus they left path compression out of their study. In our study we have no architectural restrictions since *all* herein tested structures support 32- and 64-bit integer keys. Due to the lack of space, and in order to see the effect of path compression, we have decided to (only) present **64-bit** integer keys.

We perform the experiments of 4.4.3 and 4.4.5 on two different key distributions on three different dataset sizes — for a total of six datasets. The two key distributions considered are the ones also considered in the original paper [76] and these are: (1) **Sparse** distribution, where *each* indexed key is *unique* and chosen uniformly at random from $[1, 2^{64})$, and (2) **Dense** distribution, where *every* key in $1, \ldots, n$ is indexed[7] (n is the total number of elements to be indexed by the data structures). As for datasets, for each of the aforementioned distributions we considered three different sizes: **16**, **256**, and **1000 million**. Two out of these three datasets (16M, 256M) were also considered in the original ART paper, along with a size of 65K. We would like to point out that 65K pairs of 16 bytes each is rather small and fits comfortably in the L3 cache of a modern machine. For such a small size whether an index structure is needed is debatable. Thus, we decided to move towards "big" datasets, and include the one billion size instead. Finally, the shown performance is the average of three independent measurements. For the sparse distribution of keys each measurement has a different input set.

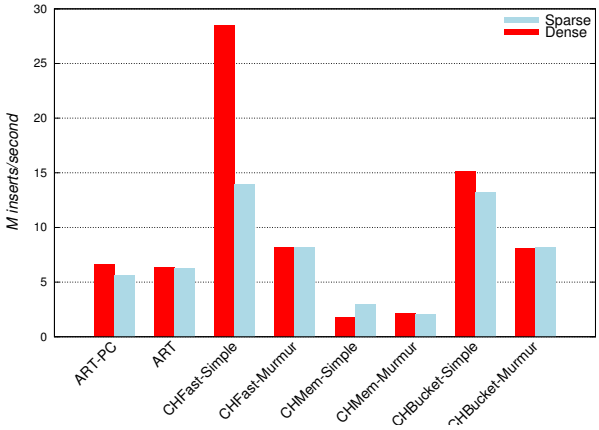
4.4.3 Non-covering evaluation

In this very first set of experiments we test ART against Cuckoo hashing under the workload explained in 4.4.2. Lookups are point queries and each one of them looks up for an existing key. After having inserted all keys, the set of keys used for insertions is permuted uniformly at random, and then the keys are looked up in this random order. This guarantees that insertions and lookups are independent from each other. Insertion and lookup performance can be seen in Figures 4.2 and 4.3 respectively, where each is presented in millions of operations per second. In Figure 4.4 we present the effective memory footprint of each structure in megabytes. This size accounts *only* for the size of the data structure, i.e. everything except the store.

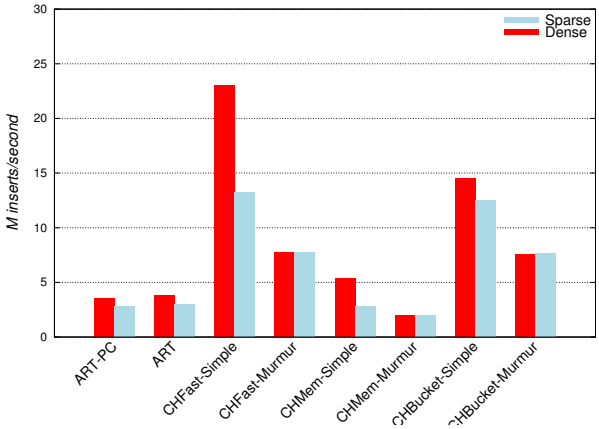
Before analyzing the results of our experiments, let us state beforehand our conclusion. The adaptive radix tree ART was originally reported [76] to have better performance than other well-engineered tree structures (of both kinds, comparison-based and radix trees). It was also reported that only hashes were competitive to ART. Our own experience indicates that well-engineered performance-based hash tables are not only competitive to ART, but actually significantly better. For example, CHFast-Simple is *at least* 2× faster for insertions and lookups than ART throughout the experiments. Moreover, this difference gets only worse for ART as the size of the set to be indexed increases. For one billion CHFast-Simple is *at least* 4.8× faster than ART for insertions and *at least* 2.8× faster for lookups, and CHBucket-Simple is *at least* 4× faster than ART for insertions, and *at least* 2× faster for lookups.

Having stated our conclusion, let us now dig more into the data obtained by the experiments. First of all (1) we can observe that using a simple, but still good hash

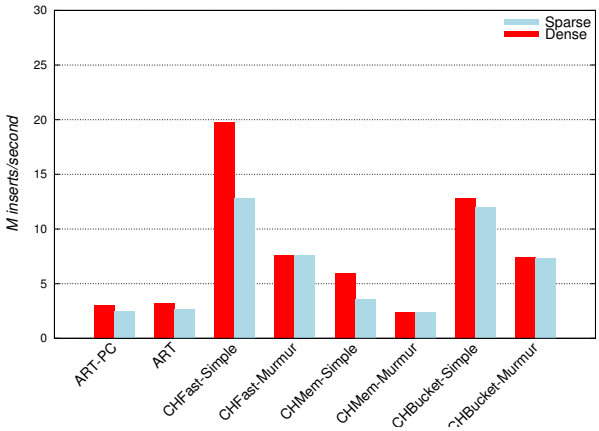
[7] Dense keys are randomly shuffled before insertion.



(a) *16M*

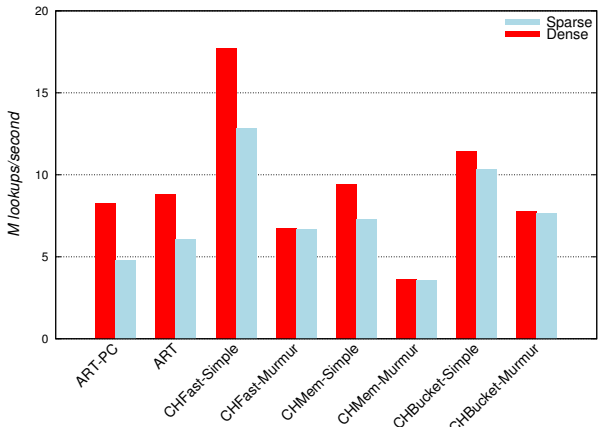


(b) *256M*

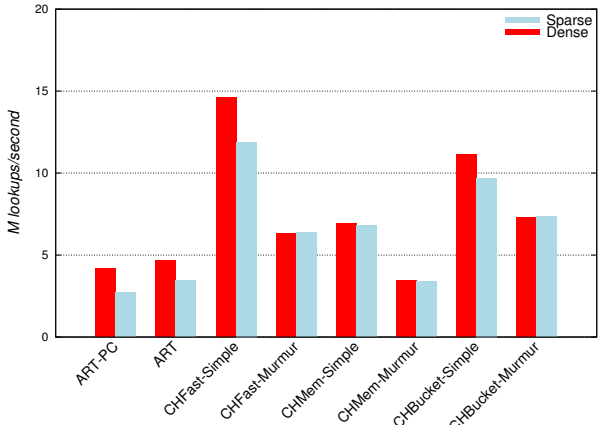


(c) *1000M*

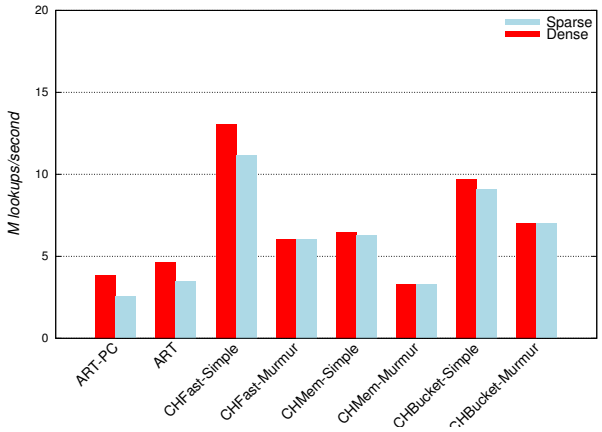
Figure 4.2: *Insertion throughput (non-covering). Higher is better.*



(a) *16M*

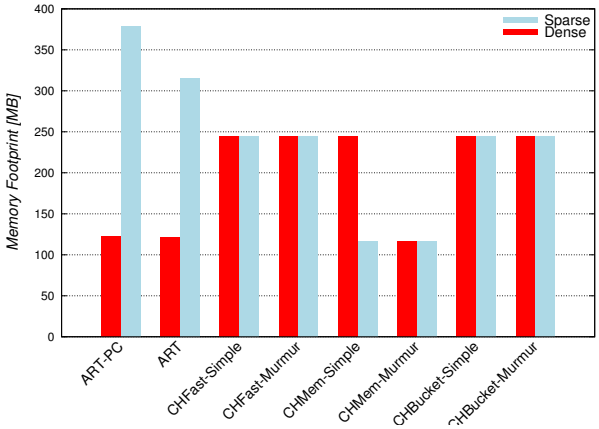


(b) *256M*

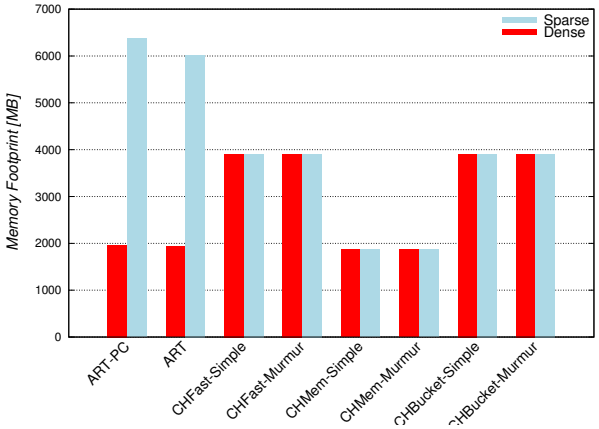


(c) *1000M*

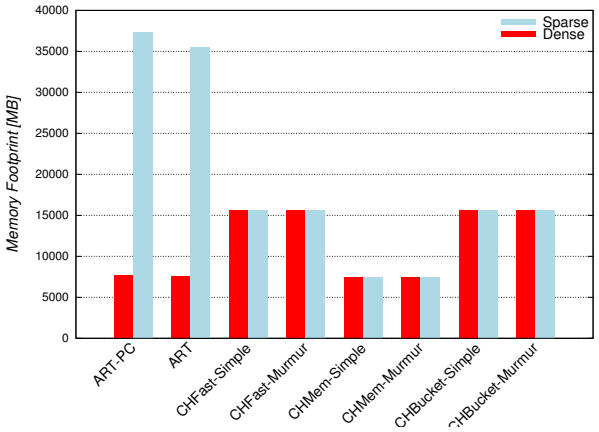
Figure 4.3: *Lookup throughput (non-covering). Higher is better.*



***(a)** 16M*



***(b)** 256M*



***(c)** 1000M*

***Figure 4.4:** Memory footprint in MB (non-covering). Lower is better.*

function, has in practice an enormous advantage over robust but complicated hash functions. CHFast-Simple is throughout the experiments roughly 1.7× faster for insertions than CHFast-Murmur, and also roughly 1.93× faster for lookups. This difference in performance is intuitively clear, but quantifying and seeing the effect makes an impression stronger than initially expected. (2) With respect to memory consumption, see Figure 4.4, multiplicative hashing seems to be robust enough for the two used distributions of keys (dense and sparse). In all but one tested case, see Figure 4.4a, multiplicative hashing uses as much space as Murmur hashing — which has been used in the past for its robustness. The discrepancy in the robustness of both hash functions suggests that a dense distribution pushes multiplicative hashing to its limits, and this has been pointed out before [32]. In our opinion, however, multiplicative hashing remains as a strong candidate to be used in practice. Also, and perhaps more important, it is interesting to see that the memory consumption of either version of ART is competitive only under the dense distribution of keys, although not better than that of CHMem. This is where the adaptivity of ART plays a significant role, in contrast to the sparse distribution, where ART seems very wasteful w.r.t. memory consumption. (3) 64-bit integer keys are (again) still too short to notice the positive effect of path compression in ART — both versions of ART have essentially the same performance, but the version without path compression is in general more space-efficient. The same effect was also reported in the original ART paper [76]. (4) With respect to performance (insertions and lookups) we can see that the performance of all structures degrades as the size of the index increases. This is due to caching effects (data and TLB misses) and it is expected, as it was also observed in the original ART paper [76]. When analyzing lookup performance, we go into more detail on these caching effects. We can also observe that as the size of the index increases, the space-efficient variant of Cuckoo hashing, CHMem-Simple, gains territory to ART. Thus, a strong argument in favor of CHMem-Simple is that it has similar performance to ART but it is in general more space-efficient.

Let us now try to understand the performance of the data structures better. Due to the lack of space we will only analyze lookups on two out of three datasets, and comparing the variant of ART without path compression against the two fastest hash tables (CHFast-Simple and CHBucket-Simple).

Lookup performance

Tables 4.1 and 4.2 show a basic cost breakdown *per lookup* for 16M and 256M respectively. From these tables we can deduce that the limiting factor in the (lookup) performance of ART is a combination of long latency instructions plus the complexity of the lookup procedure. For the first term (long latency instructions) we can observe that the sum of L3 Hits + L3 Misses is considerably larger than the corresponding sum of CHFast-Simple and CHBucket-Simple. The L3-cache-hit term is essentially nonexistent for the hashes, which is clear, and the L3-cache-miss term of ART rapidly exceeds

Table 4.1: *Cost breakdown per lookup for 16M.*

	ART		CHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	405.3	590.3	200.8	277.6	339.1	373.4
Instructions	149.3	151.1	34.60	40.13	91.94	97.78
Misp. Branches	0.027	0.972	0.126	0.662	1.645	1.857
L3 Hits	2.539	3.104	0.083	0.118	0.145	0.156
L3 Misses	2.414	3.831	2.397	3.716	3.189	3.460

Table 4.2: *Cost breakdown per lookup for 256M.*

	ART		CHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	785.0	1119	248.4	297.5	353.9	399.9
Instructions	164.9	162.9	36.02	39.68	90.75	97.00
Misp. Branches	0.045	0.686	0.303	0.638	1.608	1.825
L3 Hits	2.435	3.235	0.081	0.075	0.090	0.107
L3 Misses	4.297	6.671	2.863	3.747	3.170	3.472

that of the hashes as the index size increases. This makes perfect sense since ART decomposes a key into bytes and then uses each byte to traverse the tree. In the extreme (worst) case, this traversal incurs into at least as many cache misses as the length of the key (8 for full 64-bit integers). On the other hand, CHFast and CHBucket incur into at most two (hash) table accesses, and each access loads records from the store. Thus, CHFast incurs into *at most* four cache misses, but CHBucket could still potentially incur into more. We will go into more detail on this when analyzing CHBucket. Still, for ART we can also observe that the instruction count per lookup is the highest among the three structures. This lookup procedure works as follows: at every step, ART obtains the next byte to search for. Afterwards, by the adaptivity of ART, *every* lookup has to test whether the node we are currently at is one of four kinds. Depending on this there are four possible outcomes, in which the lightest to handle is `Node256` and the most expensive in terms of instructions is `Node16`, where search is implemented using SIMD instructions. `Node4` and `Node48` are lighter in terms of instructions than `Node16` but more expensive than `Node256`. This lookup procedure is clearly more complicated than the computation of at most two Simple hash functions (multiplicative hashing).

Let us now discuss the limiting factors in the (lookup) performance of CHFast and CHBucket. Since the amount of L3 cache hits is negligible, and the computation of Simple hash functions is also rather efficient, we can conclude that the (lookup) perfor-

mance is essentially governed by the L3 cache misses, and actually, for CHFast that is the only factor, since the lookup procedure does nothing else than hash computations (two at most) and data access. The lookup procedure of CHBucket is slightly more complicated since each location in the hash table is a bucket that contains up to eight pointers ($8 \cdot 8$ bytes = 64 bytes) to the database store. The lookup procedure first performs a hash computation for the given key k (using the first hash function). Once the corresponding bucket has been fetched, it computes a small fingerprint of k (relying only on the least significant byte) and *every* slot of the bucket is then tested against this fingerprint. A record from the store is then loaded *only* when there is a match with the fingerprint, so there could be false-positives. The fingerprint is used to avoid loading from the store all elements in a bucket. If the fingerprint matches, the corresponding record is fetched from the store and then the keys are compared to see whether the record should be returned or not. In the former case, the lookup procedure finishes, and in the later we keep looking for the right record in the same bucket. If the bucket has been exhausted, then a similar round of computation is performed using the second hash function. This procedure clearly incurs into more computations than that of CHFast, and it also seems to increase branch misprediction — there is at least one more mispredicted branch per lookup (on average) than in ART and CHFast. We tested an (almost) branchless version of this lookup procedure but the performance was slightly slower, so we decided to keep and present the branching version. This concludes our analysis of the lookup performance.

There is one more detail that we would like to point out: We can see from Figures 4.2 and 4.3 that CHBucket has a performance that lies between the performance-oriented CHFast and the space-efficient CHMem, but its space requirement is equivalent to that of CHFast. Thus, a natural question at this point is: does CHBucket make sense at all? We would like to argue in favor of CHBucket. Let us perform the following experiment: we insert 1 billion dense keys on CHFast-Simple, CHFast-Murmur, and CHBucket-Simple without preparing the hash tables to hold that many keys, i.e., we allow the hash tables to grow from the rather small capacity of $2 \cdot 2^6 = 128$ locations all the way to $2 \cdot 2^{30} = 2,147,483,648$ — growth is set to happen in powers of two. We observed that CHFast-Simple grows (rehashes) at an average load factor of 39%, CHFast-Murmur grows at an average load factor of 51%, and CHBucket-Simple is always explicitly kept at a load factor of 75%, and it always rehashes exactly at that load factor[8]. The load factor of 75% was set for performance purposes — as the load factor of CHBucket approaches 100%, its performance drops rapidly. Also, by rehashing at a load factor of 75%, we save roughly 25% of hash function computations when rehashing in comparison of rehashing at a load factor near 100%. Thus, rehashing also becomes computationally cheaper for CHBucket, and follow up insertions and lookups will benefit from the new available space (less collisions). But now, what is the real

[8]Without the manual 75% load factor, CHBucket-Simple rehashes on average at a load factor of 97%.

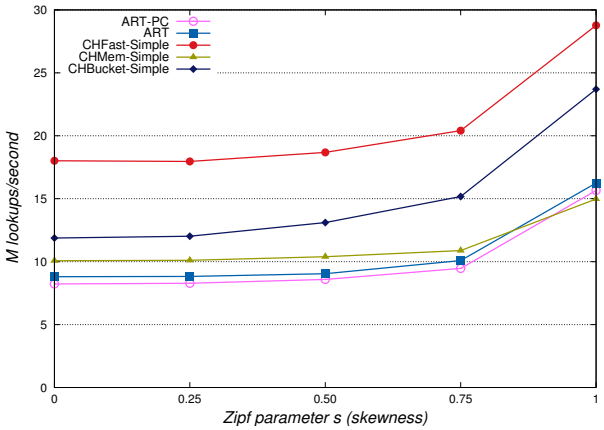


Figure 4.5: *Skewed (Zipf-distributed) lookups. Higher is better.*

argument in favor of CHBucket? The answer is in the robustness of CHFast-Simple. If CHFast-Simple was as robust as CHFast-Murmur, the former would always rehash around a 50% load factor, just as the latter, but that is not the case. This negative effect has been already studied [32], and engineering can alleviate it, but the effect will not disappear. Practitioners should be aware of this. On the other hand, CHBucket-Simple seems as robust as CHBucket-Murmur, and it could actually be considered as its replacement. Thus, by tuning the rehashing policy we can keep CHBucket-Simple at an excellent performance. In particular, it is considerably better than ART for somewhat large datasets.

We would like to close this section by presenting one more small experiment. In Figure 4.5 the effect of looking up for keys that are skewed can be observed. The lookup keys follow a Zipf distribution [74]. This experiment tries to simulate the fact that, in practice, some elements tend to be more important than others, and thus they are queried more often. Now, if certain elements are queried more often others, then they also tend to reside more often in cache, speeding up lookups. In this experiment *each* structure contains 16M dense keys.

We can see that all structures profit from skewed queries, although the relative performance of the structures stays essentially the same — CHFast-Simple and CHBucket-Simple set themselves strongly apart from ART.

4.4.4 Mixed workloads

The experiment to test mixed workloads is composed as follows: We perform one billion operations in which we vary the amount of lookups (point queries) and updates (insertions and deletions). Insertions and deletions are performed in a ratio 4:1 respectively. The distribution used for the keys is the dense distribution. Lookups, insertions and

deletions are all independent from one another. In the beginning, *every* data structure contains 16M dense keys and thus, as we perform more updates, the effect of growing the hash tables will become more apparent as they have to grow multiple times. This growth comes of course with a serious performance penalty. The results of this experiments can be seen in Figure 4.6.

We can see how the performance of *all* structures decreases rapidly as more and more updates are performed. In the particular case of the hash tables, more updates mean more growing, and thus more rehashing, which are very expensive operations. Yet, we can see that CHFast-Simple remains in terms of performance indisputably above all other structures. We would also like to point out that, although the gap between ART (ART-PC) and CHBucket-Simple narrows towards the right end (only updates), the latter still performs around one million of operations per second more than the former, which is around 20% speedup. This can hardly be ignored.

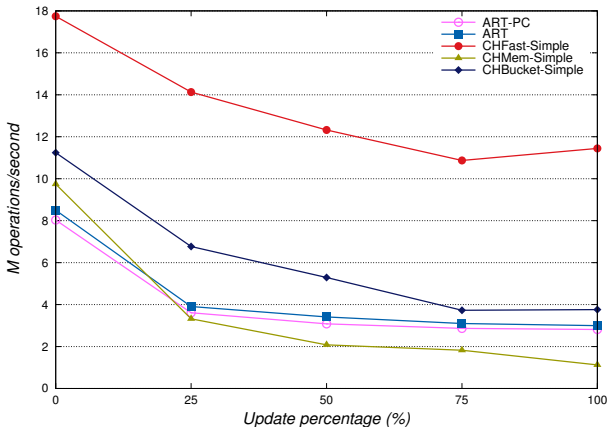


Figure 4.6: *Mixed workload of insertions, deletions, and point queries. Insertion-to-deletion ratio is 4:1. Higher is better.*

4.4.5 Covering evaluation

In this section we confront experimentally *all* data structures considered in this study: Judy, ART, Google hashes, and Cuckoo hashing. Additionally, as B$^+$-trees are omnipresent in databases, we include measurements for a B$^+$-tree [13] from now on. As B$^+$-trees have been already broadly studied in the literature, we will not discuss them here any further — we just provide them as a baseline reference and to put the other structures in perspective. Unlike the experiments presented in 4.4.3, in this section we consider each data structure as a standalone data structure, i.e., covering ⟨key, value⟩ pairs. As we mentioned before, ART was designed to be a non-covering index, unable to cover keys and values. We also mentioned that, in order to compare ART against

Table 4.3: *Cost breakdown per lookup for 16M.*

	JudyL		GHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	623.6	931.9	94.32	140.2	116.9	141.2
Instructions	216.6	215.8	46.98	53.84	32.69	36.55
Misp. Branches	0.041	1.466	0.006	0.572	1.135	1.382
L3 Hits	3.527	4.016	0.016	0.043	0.077	0.083
L3 Misses	1.460	3.737	1.104	1.793	2.006	2.466

Table 4.4: *Cost breakdown per lookup for 256M.*

	JudyL		GHFast-Simple		CHBucket-Simple	
Distribution	Dense	Sparse	Dense	Sparse	Dense	Sparse
Cycles	1212	1339.	94.72	143.2	126.9	146.4
Instructions	244.0	271.7	45.69	52.61	32.86	35.74
Misp. Branches	0.011	0.412	0.006	0.553	1.121	1.282
L3 Hits	4.103	3.116	0.025	0.058	0.084	0.085
L3 Misses	2.838	6.151	1.086	1.814	2.114	2.451

other data structures in this section, we endowed ART with its own store, which we now consider as a fundamental part of the data structure. For this store we chose the simplest and most efficient implementation, an array where each entry holds a ⟨key, value⟩ pair. This array, just as the hash tables, is pre-allocated and has enough space to hold at least n ⟨key, value⟩ pairs, for n = 16M, 256M, 1000M. We do all this to minimize the performance overhead contributed by the store of ART to the measurements. This way, we simulate an ideal table storage. Therefore, we want to point out that, when it comes to ART, there is essentially no difference in the experiments presented in Section 4.4.3 and this section. The only actual difference is that in Section 4.4.3 the store of ART is left out of the computation of space requirements since it is provided by the database. Here, nevertheless, this is not the case anymore.

As before, lookups are point queries and we query only existing keys. The insertion order is random, and once all pairs have been inserted, they are looked up for in a different random order — making insertions and lookups independent from each other. Insertion and lookup performance can be seen in Figures 4.7 and 4.8 respectively, and it is presented in millions of operations per seconds. The space requirement, in megabytes, of each data structure can be seen in Figure 4.9. Also, Tables 4.3 and 4.4 present a basic cost breakdown *per lookup* for 16M and 256M respectively. Due to the lack of space, and the similitude of the performance counters between GHFast-Simple and

CHFast-Simple, we present this cost breakdown only for JudyL, GHFast-Simple, and CHBucket-Simple. A comparison against ART can be done using the corresponding entries of Tables 4.1 and 4.2 on page 89.

Lookup performance

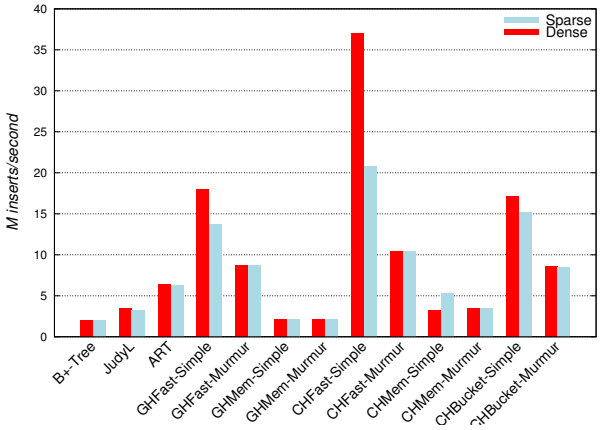
By just looking at the plots, Figures 4.7 and 4.8, we can see that there is clearly no comparison between JudyL and ART with GFastHash-Simple and CFastHash-Simple. The latter seem to be in their own league. Moreover, we can see that this comparison gets only worse for JudyL and ART as the size of the dataset increases. For one billion entries the space-efficient CHMem-Simple is now *at least* 2.3× as fast as ART, and Judy, while requiring significantly less space than ART. In this regard, space consumption, JudyL is extremely competitive. For the dense distribution of keys no other structure requires less space than JudyL, and under the sparse distribution JudyL is comparable with the space-efficient hashes. However, all optimizations (compression techniques) performed by JudyL, in order to save space, come at very expensive price. JudyL is by far the structure with the highest number of instructions per lookup.

We can observe that the amount of long latency operations (L3 Hits + L3 Misses) of ART and JudyL are very similar. Thus, we can conclude that the other limiting factor of JudyL is algorithmic, which, in the particular case of JudyL, it is also translated into code complexity — JudyL's source code is extremely complex and obfuscated.

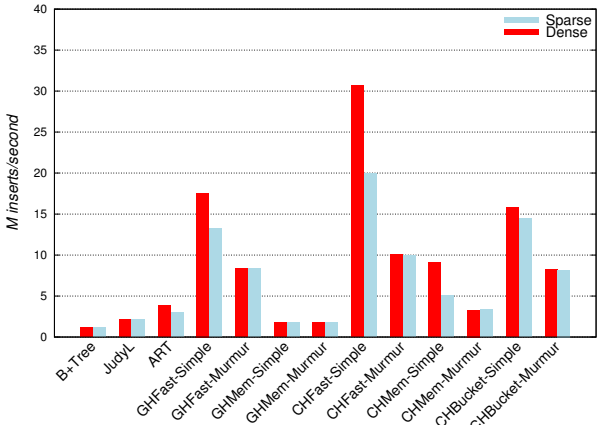
With respect to the factors limiting the (lookup) performance of the hash tables, we can again observe that the amount of L3 cache hits is negligible, the instruction counts is very small, and thus, what is limiting the hash tables is essentially the amount of L3 cache misses. We can additionally observe that CHBucket-Simple incurs into more than one branch misprediction per lookup — these mispredictions are happening when looking for the right element inside a bucket. However, these mispredictions cannot affect the performance of CHBucket-Simple as they potentially do in its non-covering version (Tables 4.1 and 4.2), since this time these mispredictions cannot trigger long latency operations due to speculative loads (usually resulting into L3 cache misses).

Range queries

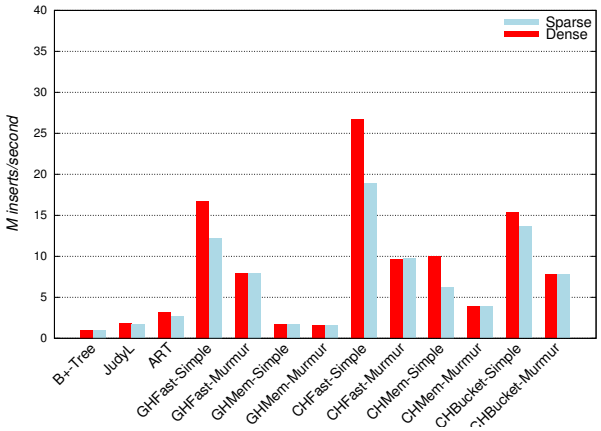
So far, all experiments have considered only point queries. We now take a brief look at range queries. Clearly, range queries are the weak spot of hash tables since elements in a hash table are in general not stored in a particular order. However, in the *very particular* case that keys come from a small discrete universe, as in the case of the dense distribution, we could answer a range query $[a, b]$ by looking up in a hash table for *every* possible value between a and b, the whole range. Depending on the selectivity of the query, this method avoids looking up the whole hash table. For our experiment we fire up three sets of 1000 range queries, every set with a different selectivity: 10%, 1%, 0.1%



***(a)** 16M*

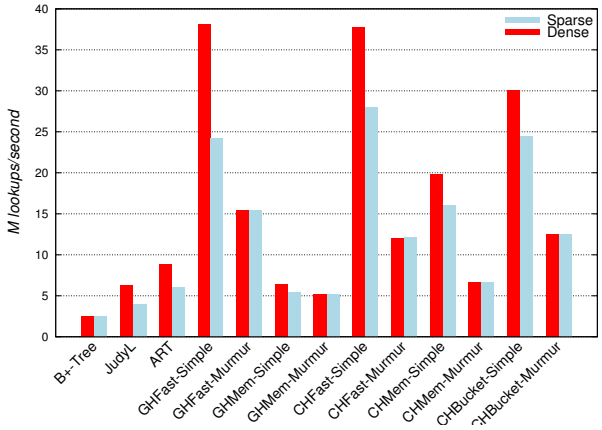


***(b)** 256M*

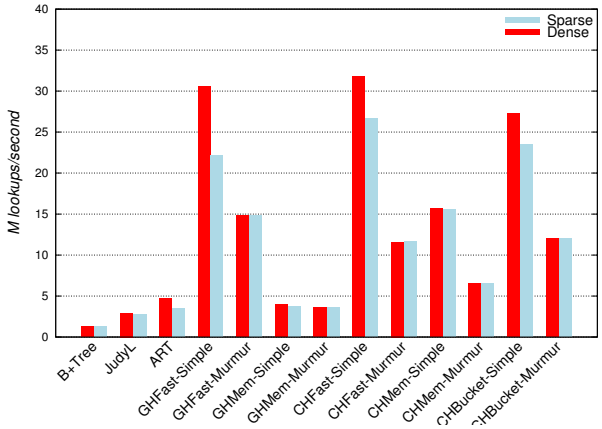


***(c)** 1000M*

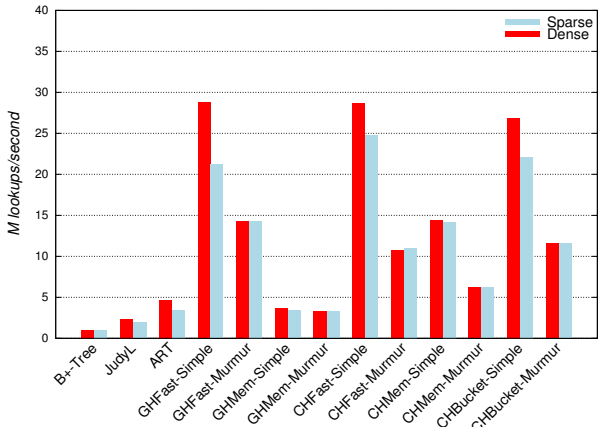
***Figure 4.7:** Insertion throughput (covering). Higher is better.*



(a) *16M*

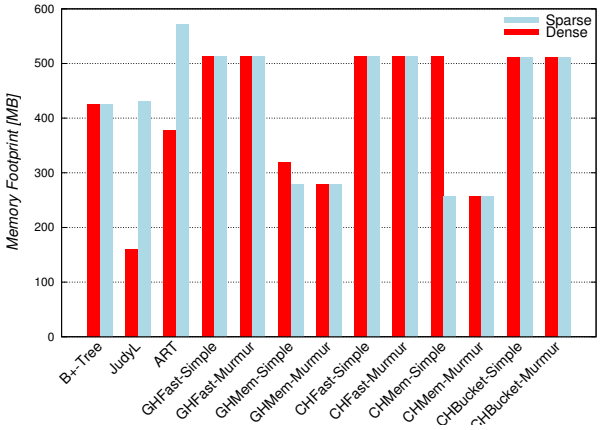


(b) *256M*

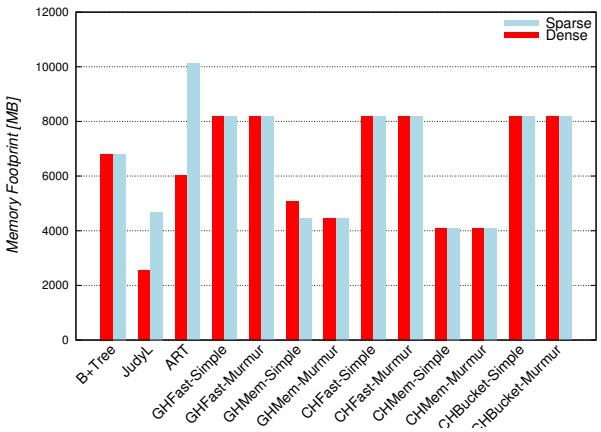


(c) *1000M*

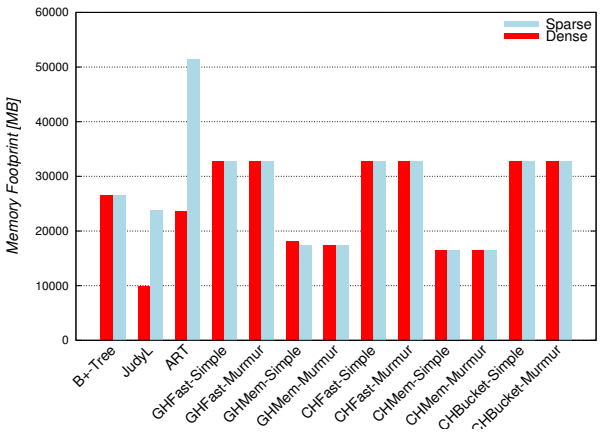
Figure 4.8: *Lookup throughput (covering). Higher is better.*



(a) *16M*



(b) *256M*



(c) *1000M*

Figure 4.9: *Memory footprint in MB (covering). Lower is better.*

respectively, on structures containing exactly 16M keys. For the sparse distribution we refrain ourselves from answering the queries using hash tables because it hardly makes sense. The results can be seen in Figure 4.10 below. All structures are covering versions, as the ones used in Section 4.4.5. As we mentioned before, we implemented range-query support in ART, and our implementation is based on tree-traversal.

It is hard to see in the plot, but the difference in throughput between adjacent selectivities is a factor of 10. It is also very surprising to see that the hashes still perform quite good under the dense distribution. Again, the use cases for which hash tables can be used in this manner are *very* limited, but not impossible to find.

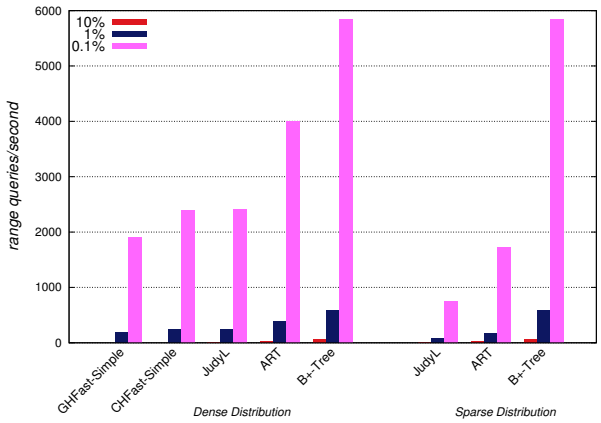


Figure 4.10: *Range queries over 16M dense and sparse keys. Covering versions of hash tables are only shown for dense keys. Higher is better.*

4.5 Conclusions

In the original ART paper [76], the authors thoroughly tested ART, and their experiments supported the claim that only a hash table was competitive (performance-wise) to ART. In our experiments we extended the original experiments by considering hashing schemes other than chained hashing. Our experiments clearly indicate that the picture changes when we carefully choose both, the hashing scheme *and* the hash function. *Our* conclusion is that a *carefully chosen* hash table is not only competitive with ART, but actually significantly better. For example, for an OLAP scenario, and for one billion indexed keys, one non-covering variant of Cuckoo hashing is *at least* 4.8× faster for insertions, *at least* 2.8× faster for lookups, and it sometimes requires just half the space of ART, see Figures 4.2, 4.3, and 4.4. For an OLTP scenario, the same variant is up to 3.8× faster than ART, see Figure 4.6. We also tested ART against another (older) adaptive radix tree (Judy). In our experiments, ART ended up having almost 2× better

performance over Judy, but at the same time, it tends to also use twice as much space. This is an important trade-off to keep in mind.

Towards the very end we presented a small experiment to test performance under range queries. Here, ART was clearly outperforming Judy and all hash tables. However, ART is still slower than a B^+-tree by up to a factor of 3. Furthermore, we also observe that in the *very limited* case of a dense distribution coming from a small discrete universe, hash tables perform surprisingly good (comparable to Judy), and deciding whether hash tables could be use for range queries this way takes no time to the query optimizer.

Chapter 5

A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing

5.1 Introduction

In recent years there has been a considerable amount of research on tree-structured main memory indexes, e.g. [72, 102, 8, 15, 69, 85, 71, 76]. However, it is hard to find recent database literature thoroughly examining the effects of different hash tables in query processing. This is unfortunate for at least two reasons: First, hashing has plenty of applications in modern database systems, including join processing, grouping, and accelerating point queries. In those applications, hash tables serve as a building block. Second, there is strong evidence that hash tables are much faster than even the most recent and best tree-structured indexes. For instance, in our recent experimental analysis [6] we carefully compared the performance of modern tree-structured indexes for main memory databases like ARTful [76] with a selection of different hash tables[1]. A central lesson learned from our work [6] was that a carefully and well-chosen hash table is still *considerably* faster (up to factor 4-5x) for point queries than any of the aforementioned tree-structured indexes. However, our previous work also triggered some nagging research questions: (1) When exactly should we choose which hash table? (2) What are the most efficient hashing methods that should be considered for query processing? (3) What other dimensions affect the choice of "the right" hash table? and finally (4) What is the performance impact of those factors. While investigating answers to these questions we stumbled over interesting results that greatly enriched our

[1] We use the term *hash table* throughout the paper to indicate that *both* the hashing scheme (say linear probing) and the hash function (say Murmur) are chosen.

knowledge, and that could greatly help practitioners, and potentially also the optimizer, to take well-informed decisions as of when to use what hash table.

5.1.1 Our Contributions

We carefully study *single-threaded* hashing for 64-bit integer keys and values in a five-dimensional requirements space:

1. **Data distribution.** Three different data distributions: dense, sparse, and a grid-like distribution (think of IP addresses).
2. **Load factor.** Six different load factors between 25- and 90%.
3. **Hash table size.** We consider a variety of sizes for the hash tables to observe performance when they are rather small (they fit in cache), and when they are of medium and large sizes (outside cache but still addressable by TLB using huge pages or not respectively).
4. **Read/write-ratio.** We consider whether the hash tables are to be used under a static workload (OLAP-like) or a dynamic workload (OLTP-like). For both we simulate an indexing workload — which in turn captures the essence of other important operations such as joins or aggregates.
5. **Un/successful lookup ratio.** We study the performance of the hash tables when the amount of lookups (probes) varies from *all successful* to *all unsuccessful*.

Each point in that design space may potentially suggest a different hash table. We show that a right/wrong decision in picking the right combination ⟨hashing scheme, hash function⟩ may lead to an order of magnitude difference in performance. To substantiate this claim, we carefully analyze two additional dimensions:

6. **Hashing scheme.** We consider linear probing, quadratic probing, Robin Hood hashing as described in [21] but carefully engineered, Cuckoo hashing [90], and two different variants of chained hashing.
7. **Hash function.** We integrate each hashing scheme with four different hash functions: Multiply-shift [30], Multiply-add-shift [29], Tabulation hashing [114, 110, 93], and Murmur hashing [7], which is widely used in practice. This gives 24 different combinations (hash tables).

Therefore, **we study in total a set of seven different dimensions** that are key parameters to the overall performance of a hash table. We shed light on these seven dimensions focusing on one of the most important use-cases in query processing: indexing. This in turn resembles very closely other important operations such as joins and aggregates — like `SUM`, `MIN`, etc. Additionally, we also offer a glimpse about the effect of different table layout and the use of SIMD instructions. Our main goal is to produce enough

results that can guide practitioners, and potentially the optimizer, towards choosing the most appropriate hash table for their use case at hand. **To the best of our knowledge, no work in the literature has considered such a thorough set of experiments on hash tables**.

Our study clearly indicates that picking the right configuration may have considerable impact on standard query processing tasks such as main memory indexing as well as join processing, which heavily rely on hashing. Hence, hashing should be considered as a white box method in query processing and query optimization.

We decided to focus on studying hash tables in a single-threaded context to isolate the impact of the aforementioned dimensions. We believe that a thorough evaluation of concurrency in hash tables is a research topic in its own and *beyond the scope of this study*. However, our observations still play an important role for hash maps in multi-threaded algorithms. For partitioning-based parallelism — which has recently been considered in the context of (partition-based hash) joins [10, 11, 75] — single-threaded performance is still a key parameter: each partition can be considered an isolated unit of work that is only accessed by exactly one thread at a time, and therefore concurrency control inside the hash tables is not needed. Furthermore, all hash tables we present in the chapter can be extended for thread safety through well-known techniques such as striped locking or compare-and-swap. Here, the dimensions we discuss still impact the performance of the underlying hash table.

This chapter is organized as follows: In Sections 5.2 and 5.3 we briefly describe each of the five considered hashing schemes and the four considered hash functions respectively. In Section 5.4 we describe our methodology, setup, measurements, and the three data distributions used. We also discuss why we have narrowed down our result set — we present in this chapter what we consider the most relevant results. In Sections 5.5, 5.6, and 5.7 we present *all* our experiments along with their corresponding discussion.

5.2 Hashing Schemes

In this chapter, we study the performance of five different hashing schemes: (1) chained hashing, (2) linear probing, (3) quadratic probing, (4) Robin Hood hashing on linear probing, and (5) Cuckoo hashing — the last four belong to the so-called open-addressing schemes, in which *every* slot of the hash table stores exactly one element, or stores special values denoting whether the corresponding slot is free. For open-addressing schemes we assume that the tables have l slots (l is called *capacity* of the table). Let $0 \leq n \leq l$ be the number of occupied slots (we call n the *size* of the table) and consider the ratio $\alpha = \frac{n}{l}$ as the *load factor* of the table. For chained hashing, the concept of load factor makes in general little sense since it can store more than one element in the same slot using a linked list, and thus we could obtain $\alpha > 1$. Hence, whenever we

discuss chained hashing for a load factor α, we mean that the presented chained hash tables are memory-wise comparable to open-addressing hash tables at load factor α — in particular, the hash tables contain the same number n of elements, but their directory size can differ. We elaborate on this in Section 5.4.5.

Finally, one fundamental question in open-addressing is whether to organize the table as array-of-structs (AoS) or as a struct-of-arrays (SoA). In AoS, the table is stored in one (or more in case of Cuckoo hashing) arrays of key-value pairs, similar to a row layout. In contrast to that, SoA representation keeps keys and corresponding values separated in two corresponding, aligned arrays - similar to column layout. We found in a micro-benchmark that AoS is superior to SoA in most relevant cases for our setup and hence apply this organization in all open-addressing schemes in this chapter. For more details on this micro-benchmark see Section 5.7. We now proceed to briefly describe each considered hashing scheme in turn.

5.2.1 Chained Hashing

Standard chained hashing is a very simple approach for collision handling, where each slot of table T (the directory) is a pointer to a linked list of entries. On inserts, entries are appended to the list that corresponds to their key k under hash function h, i.e., $T[h(k)]$. In case of lookups, the linked list under $T[h(k)]$ is searched for the entry with key k. Chained hashing is a simple and robust method that is widely used in practice, e.g., in the current implementations of `std::unordered_map` in `C++` STL or `java.util.HashMap` in `Java`. However, compared to open-addressing methods, chained hashing has typically sub-optimal performance for integer keys w.r.t. runtime and memory footprint. Two main reasons for this are: (1) the pointers used by the linked lists lead to a high memory overhead and (2) using linked lists leads to additional cache misses (even for slots with one element and no collisions). This situation brings different opportunities for optimizing a traditional chained hash table. For example, we can reduce cache misses by making the directory wide enough (say 24-byte entries for key-value-pointer triplets) so that we can *always* store one element directly in the directory and avoid following the corresponding pointer. Collisions are then stored in the corresponding linked list. In this version we potentially achieved the latency of open-addressing schemes (if collisions are rare) at the cost of space. Throughout this chapter we denote the two versions of chained hashing we mentioned by **ChainedH8**, and **ChainedH24** respectively.

In the very first set of experiments we studied the performance of ChainedH8, and ChainedH24 under a variety of factors, as to better understand the trade-offs they offer. One key observation that we would like to point out at this point is: We observed that entry allocation in the linked lists is a key factor for insert performance in all our variants of chained hashing. For example, a naive approach with dynamic allocation, i.e., using one `malloc` call per insertion, and one `free` call per delete, lead to a significant

overhead. For most use cases, an alternative allocation strategy provides a considerable performance benefit. That is, for both chained hashing methods in our indexing experiments, Sections 5.5 and 5.6, we use a *slab allocator*. The idea is to bulk-allocate many (or up to all) entries in one large array and store all map entries consecutively in this arrays. This strategy is very efficient in all scenarios where the size of the hash table is either known in advance or only growing. We observed an improvement over traditional allocation in both: memory footprint (due to less fragmentation and less `malloc` metadata) as well as raw performance (by up to one order of magnitude!).

5.2.2 Linear Probing

Linear probing (**LP**) is the simplest scheme for collision handling in open-addressing. The hash function is of the following form: $h(k, i) = (h'(k)+i) \mod l$, where i represents the i-th probed location and $h'(k)$ is an auxiliary hash function. It works as follows: First, try to insert each key-value pair $p = \langle k, v \rangle$ with key k at the optimal slot $T[h(k, 0)]$ in an open-addressing hash table T. In case $h(k, 0)$ is already occupied by another entry with different key, we (circularly) probe the consecutive slots $h(k, 1)$ to $h(k, l-1)$. We store p in the first free slot $T[h(k, i)]$, for some $0 < i < l$, we encounter[2]. We define the *displacement* d of p as i, and the sum of displacements over all entries as the *total displacement* of T. Observe that the total displacement is a measure of performance in linear probing since a high value implies long probe sequences entries during lookups.

The rather simple strategy of LP has two advantages: (1) Low code complexity which allows for fast execution and (2) Excellent cache efficiency due to the sequential linear scan. However, on high load factors > 60%, LP noticeably suffers from *primary clustering*, i.e., a tendency to create long sequences of filled slots and hence high total displacement. We will address those areas of occupied slots that are adjacent w.r.t. probe sequences as clusters. Further, we can also observe that unsuccessful lookups worsen the performance of LP since they require a complete scan of all slots up to the first empty slot. Linear probing also requires dedicated handling of deletes, i.e., we cannot simply remove entries from the hash table because this could disconnect a cluster and produce incorrect results under lookups. One option to handle deletes in LP are the so called *tombstones*, i.e., a special value (different from the empty slot) that marks deleted entries so that lookups continue scanning after seeing one tombstone — yielding correct results. Using tombstones makes deletes very fast. However, tombstones can have a negative impact on performance, as they potentially connect otherwise unconnected clusters, thus building larger clusters. Inserts can replace a tombstone that is found during a probe after confirming that the key to insert is not already contained. Another strategy to handle deletes is *partial cluster rehash*: we delete the entry from the slot and rehash all following entries in the same cluster. For our experiments we decided to

[2]Observe that as long as the table is not full, an empty slot is found.

implement an optimized version of tombstones which will only place tombstones when required to keep a cluster connected (i.e. only if the next slot from the deleted entry is occupied). Placing tombstones is very fast (faster in general than rehashing after every deletion), and the only negative point about tombstones are lookups after a considerable amount of deletions — in such a case we could shrink the hash table and perform a rehash anyway.

One of our main motivations to study linear hashing in this chapter is not only that it belongs to the classical hashing schemes, which dates to the 50's [74], but also the recent developments regarding its analysis. Knuth was the first [73] to give a formal analysis of the operations of linear probing (insertion, deletions, lookups) and he showed that all these operation can be performed in $O(1)$ using *truly* random hash functions[3]. However, very recently [93] it was shown that linear probing with tabulation hashing (see Section 5.3.3) as a function matches asymptotically the bounds of Knuth in expected running time $O(\frac{1}{\varepsilon^2})$, where the hash table has capacity $l = (1 + \varepsilon)n$. That is, from a theoretical point of view, there is no reason to use any other hashing table. We will see in our experiments, however, that the story is slightly different in practice.

5.2.3 Quadratic Probing

Quadratic probing (**QP**) is another popular approach for collision handling in open-addressing. The hash function in QP is of the following form: $h(k, i) = (h'(k)+c_1 \cdot i+c_2 \cdot i^2) \mod l$, where i represents the i-th probed location, h' is an auxiliary hash function, and $c_1 \geq 0$, $c_2 > 0$ are auxiliary constants.

In case that the capacity of the table l is a power of two and $c_1 = c_2 = 1/2$, it can be proven that quadratic probing will consider *every* single slot of the table one time in the worst case [25]. That is, as long as there are empty slots in the hash table, this particular version of quadratic probing will *always* find them eventually. Compared to linear probing, quadratic probing has a reduced tendency for primary clustering and comparably low code complexity. However, QP still suffers from so-called *secondary clustering*: if two different keys collide in the very first probe, they will also collide in all sub-sequent probes. For deletions, we can apply the same strategies as in LP. Our definition of displacement for LP carries over to QP as the number of probes $0 < i < l$ until an empty slot is found.

5.2.4 Robin Hood Hashing on LP

Robin Hood hashing [21] is an interesting extension that can be applied to many collision handling schemes, e.g., linear probing [109]. For the remainder of this chapter, we

[3]Which map *every* key in a given universe of keys independently and uniformly onto the hash table.

will only talk about Robin Hood hashing on top of LP and simply refer to this combination as Robin Hood hashing (**RH**). Furthermore, we introduce a new tuned approach to Robin Hood hashing that improves on the worst-case scenario of LP (unsuccessful lookups on high load factors) at a small cost on inserts, and very high rates of successful lookups (close to 100%, best-case scenario).

According to Viola [109], RH is based on the observation that hash collisions can be resolved in favor of any of the keys involved. Viola proposes to exploit this additional degree of freedom to modify the insertion algorithm of LP as follows: On a probe sequence to insert a new entry e_{new}, whenever we encounter an existing entry e_{old} with displacement $d(e_{new}) > d(e_{old})$[4], we exchange e_{old} by e_{new} and continue the search for an empty slot with e_{old}. As a result, the variance in displacement between all entries is minimized. While this approach does not change the total displacement compared to LP, we can exploit the established ordering in other ways. In this sense, the name Robin Hood was motivated by the observation that the algorithm takes from the"rich" elements (with smaller displacement) and gives to the "poor" (with higher displacement). Thus distributing the "wealth" (proximity to optimal slot) more fairly across all elements without changing the average "wealth" per element.

It is known that RH can reduce the variance in displacement significantly over LP. Viola [109] suggests to exploit this property to improve on unsuccessful lookups in several ways. For example, we could already start searching for elements at the slot with expected (average) displacement from their perfect slot and probe bidirectional from there. In practice, this is not very efficient due to high branch misprediction rates and/or unfriendly access pattern. Another approach introduces an early abort criterium for unsuccessful lookups. If we keep track of the maximum displacement d_{max} among all entries in the hash table, a probe sequence can already stop after d_{max} iterations. However, in practice we observed that d_{max} is often still too high[5] to obtain significant improvements over LP. We can improve on this method by introducing a different abort criterion, which compares the probe iteration i with the displacement of currently probed entry $d(e_i)$ in every step and stops as soon as $d(e_i) < i$. However, comparing against $d(e_i)$ on each iteration requires us to either store displacement information or re-calculate the hash value. We found all those approaches to be prohibitively expensive w.r.t. runtime and inferior to the plain LP in most scenarios. Instead, our approach applies early abortion by hash computation only on every m-th probe, where a good choice of m is slightly bigger than the average displacement in the table. As computing the average displacement under updates can be expensive, a good sweet spot for most load factors is to check once at the end of each cache-line. We found this to give a good tradeoff between an overhead for successful probes and the ability to stop unsuccessful probes early. Hence, this is the configuration we use for RH in our experiments. Furthermore,

[4] If $d(e_{new}) = d(e_{old})$ we can compare the actual keys as tie breaker to establish a full ordering.

[5] For high load factor α, d_{max} can often be an order of magnitude higher than the average displacement.

our approach to RH applies partial rehash for deletions which turned out to be superior to tombstones for this table. Notice that, tombstones in RH would, for correctness, require to store information that allow us to reconstruct the displacement of the deleted entry.

5.2.5 Cuckoo Hashing

Cuckoo hashing [90] (**CuckooH**) is a another open-addressing scheme that, in its original (and simplest) version, works as follows: There are two hash tables T_0, T_1, each one having its own hash function h_0 and h_1 respectively. Every inserted element p is stored at either $T_0[h_0(p)]$ or $T_1[h_1(p)]$ but *never* in both. When inserting an element p, location $T_0[h_0(p)]$ is first probed, if the location is empty, p is stored there, otherwise, p kicks out the element q already found at that location, p is stored there, and q is tried to be inserted at location $T_1[h_1(q)]$. If this location is free, q is stored there, otherwise q kicks out the element therein, and we repeat: in iteration $i \geq 0$, location $T_j[h_j(\cdot)]$ is probed, where $j = i \mod 2$. In the end we hope that *every* element finds its own "nest" in the hash table. However, it may happen that this process enters a loop, and thus a place for each element is never found. This is dealt with by performing only a fixed amount of iterations, once this limit is achieved, a rehash of the complete set is performed by choosing two new hash functions. The advantages of CuckooH are (1) For lookups, traditional CuckooH requires at most two tables accesses, which is in general optimal among hashing schemes using linear space. In particular, the load factor has only a small impact on the lookup performance of the hash table. (2) CuckooH has been reported [90] to be competitive with other good hashing schemes, like linear and quadratic probing, and (3) CuckooH is easy to implement. However, it has been empirically observed [90, 46] that the load factor of traditional CuckooH with 2 tables should stay slightly below 50% in order to work. More precisely, below 50% load factor creation succeeds with high probability, but it starts failing from 50% on [39, 86]. This problem can be alleviated by generalizing CuckooH to use more tables $T_0, T_1, T_2 \ldots T_k$, each having its own hash function h_k, $k > 1$. For example, for $k = 4$ the load factor (empirically) increases to 96% [46]. All this at the expense of performance, since now lookups require at most four table lookups. Furthermore, Cuckoo hashing is very sensitive to what hash functions are used [90, 32, 93] and requires robust hash functions. In our experiments we only consider Cuckoo hashing on four tables (called **CuckooH4**) since we want to study the performance of hash tables under many different load factors, that go up to 90%, and CuckooH4 is the only version of traditional Cuckoo hashing that offers this flexibility.

5.3 Hash Functions

In our study we want to investigate the impact of different hash functions in combination with various hashing schemes (Section 5.2) under different key distributions. Our set of hash functions covers a spectrum of different theoretical guarantees that also admit very efficient implementations (low code complexity) and thus are also used in practice. We also consider one hash function that is, in our opinion, the most representative member of a class of engineered hash functions[6] that *do not* necessarily have theoretical guarantees, but that show good empirical performance, and thus are widely used in practice. We believe that our chosen set of hash functions is very representative and offers practitioners a good set of **hash functions for integers** (64-bit in this study) to choose from. The set of hash functions we considered is: (1) Multiply-shift [30], (2) Multiply-add-shift [29], (3) Tabulation hashing [93], and (4) Murmur hashing [7]. Formally, (1) is the weakest and (3) is the strongest w.r.t. randomization. The definition and properties of these hash functions are as follows:

5.3.1 Multiply-shift

Multiply-shift (**Mult**) is *very* well known [30], and it is given here:

$$h_z(x) = (x \cdot z \mod 2^w) \texttt{ div } 2^{w-d}$$

where x is a w-bit integer in $\{0, \ldots, 2^w - 1\}$, z is an odd w-bit integer in $\{1, \ldots, 2^w - 1\}$, the hash table is of size 2^d, and the `div` operator is defined as: $a \texttt{ div } b = \lfloor a/b \rfloor$. What makes this hash function highly interesting is: (1) It can be implemented extremely efficiently by observing that the multiplication $x \cdot z$ is natively done modulo 2^w in current architectures for native types like 32- and 64-bit integers, and the operator div is equivalent to a right bit shift by $w - d$ positions. (2) It has been proven [30] that if $x, y \in \{0, \ldots, 2^w - 1\}$, with $x \neq y$, and if $z \in \{1, \ldots, 2^w - 1\}$ chosen uniformly at random, then the collision probability is $\frac{1}{2^{d-1}}$.

This also means that the family of hash functions $H_{w,d} = \{h_z \mid$
$0 < z < 2^w$ and z odd$\}$ is the ideal candidate for simple and rather robust hash functions. Multiply-shift is a universal hash function.

5.3.2 Multiply-add-shift

Multiply-add-shift (**MultAdd**) is also a very well known hash function [29]. It's definition is very similar to the previous one:

$$h_{a,b}(x) = ((x \cdot a + b) \mod 2^{2w}) \texttt{ div } 2^{2w-d}$$

[6] Like FNV, CRC, DJB, CityHash for example.

where again x is a w-bit integer, a, b are two $2w$-bit integers, and 2^d is the size of the hash table. For $w = 32$ this hash function can be implemented natively under 64-bit architectures, but $w = 64$ requires 128-bit arithmetic which is still not widely supported natively. It can nevertheless still be implemented (keeping its formal properties) using only 64-bit arithmetic [106]. When a, b are randomly chosen from $\{0, \ldots, 2^{2^w}\}$, it can be proven that collision probability is $\frac{1}{2^d}$, and thus is stronger than Multiply-shift — although it also incurs into heavier computations. Multiply-add-shift is a 2-independent hash function.

5.3.3 Tabulation hashing

Tabulation hashing (**Tab**) is the strongest hash function among all the ones that we consider and also probably the least known. It became more popular in recent years since it can be proven [93] that tabulation and linear probing achieve $O(1)$ for insertions, deletions, and lookups. This produces a hash table that is, in asymptotic terms, unbeatable. Its definition is as follows (we assume 64-bit keys for simplicity): Split the 64-bit keys into c characters, say eight chars $c_1, \ldots, c_8$. For every position $1 \leq i \leq 8$ initialize a table T_i with 256 entries (for chars) with *truly* 64-bit random codes. The hash function for key $x = c_1 \cdots c_8$ is then:

$$h(x) = \bigoplus_{i=1}^{8} T_i[c_i]$$

where $\bigoplus$ denotes the bitwise `XOR`. So a hash code is composed by the `XOR` of the corresponding entries in tables T_i of the characters of x. If all tables are filled with truly random data, then it is known that tabulation is 3-independent (but not stronger), which means that for *any* three distinct keys x_1, x_2, x_3 from our universe of keys, and three (not necessarily distinct) hash codes $y_1, y_2, y_3 \in \{0, \ldots, l\}$ then

$$Pr[h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge h(x_3) = y_3] \leq \frac{1}{l^3}$$

which means that under tabulation hashing, the hash code $h(x_i)$ is uniformly distributed onto the hash table for *every* key in our universe, and that for *any* three distinct keys x_1, x_2, x_3, the corresponding hash codes are three independent random variables.

Now, the interesting part of tabulation hashing is that it requires only bitwise operations, which are very fast, and lookups in tables $T_1, \ldots, T_8$. These tables are as heavy as $256 \cdot 8 \cdot 8\text{B} = 16\text{KB}$. Which mean that they *all* fit comfortably in the L1 cache of processors, which is 32 or 64KB in modern computing servers. That is, lookups in those tables incur in potentially low latency operations, and thus the evaluation of single hash codes is potentially very fast.

5.3.4 Murmur hashing

Murmur hashing (**Murmur**) is one of the most common hash functions used in practice due to its good behavior. It is relatively fast to compute and it seems to produce quite good hash codes. We are not aware of any formal analysis on this, so we use Murmur hashing *essentially* as is. As we limit ourselves in this study to 64-bit keys, we use Murmur3's 64-bit finalizer [7] as shown in the code below.

```
uint64_t murmur3_64_finalizer(uint64_t key) {
    key ^= key >> 33;
    key *= 0xff51afd7ed558ccd;
    key ^= key >> 33;
    key *= 0xc4ceb9fe1a85ec53;
    key ^= key >> 33;
    return key;
}
```

5.4 Methodology

Throughout the chapter we want to understand how well a hash table can work as a plain index for a set of n ⟨key, value⟩ pairs of 64-bit integers. The keys obey three different data distributions, described later on in Section 5.4.3. This scenario, albeit generic, resembles very closely other interesting uses of hash tables such as in join processing or in aggregate operations like `AVERAGE`, `SUM`, `MIN`, `MAX`, and `COUNT`. In fact, we performed experiments simulating these operations, and the results were comparable those from the WORM workload.

We study the relation between (raw) performance and load factors by performing insertions and lookups (successful and unsuccessful) on hash tables at different load factors. For this we consider a write-once-read-many (**WORM**) workload, and a mixed read-write (**RW**) workload. These two kinds of workload simulate elementary operational requirements of OLAP and OLTP scenarios, respectively, for index structures.

5.4.1 Setup

All experiments are **single-threaded** and all implementations are our own. All hash tables have map semantics, i.e., they cover both key and value. All experiments are in **main memory**. For the experiments in Sections 5.5 and 5.6 we use a single core (one NUMA region) of a dual-socket machine having two hexacore Intel Xeon Processors X5690 running at 3.47 GHz. The machine has a total of 192GB of RAM running at 1066 MHz. The OS is a 64-bit Linux (3.2.0) with **page size** of 2MB (using transparent huge pages). All algorithms are implemented in `C++` and compiled with `gcc-4.7.2`

with optimization `-O3`. Prefetching, hyper-threading and turbo-boost are disabled via BIOS to isolate the real characteristics of the considered hash tables.

Since our server does not support AVX-2 instructions, we ran the layout and SIMD evaluation, Section 5.7, on a MacbookPro with Intel Core i7-4980HQ running at 2.80GHz (Haswell) with 16GB DDR3 RAM at 1600 MHz running Mac OS X 10.10.2 in single-user mode. Here, the page size is 4KB and pre-fetching is activated since we could not deactivate it as cleanly as for our linux server. All binaries are compiled with `clang-600.0.56` with optimization `-O3`.

5.4.2 Measurement and Analysis

For all indexing experiments of Sections 5.5 and 5.6 we report the average of three independent runs (three different random seeds for the generation and shuffling of data). In each run, we measure performance through for-loops that invokes the insert or lookup methods of the hash tables, using one method invocation per key.

We performed an analysis of variance on all results and we found that, in general, the results are overall very stable and uniform. Whenever variance was noticeable, we reran the corresponding experiment with the same setting to rule out machine problems. As variance was very insignificant, we decided that there is no added benefit in showing it in the plots.

5.4.3 Data distributions

Every indexed key is **64** bits. We consider three different kinds of data distributions: **Dense**, **Sparse**, and **Grid**. In the dense distribution we index *every* key in $[1 : n] := \{1, 2, \ldots, n\}$. In the sparse distribution, $n \ll 2^{64}$ keys are generated uniformly at random from $[1 : 2^{64} - 1]$. In the grid distribution *every* byte of *every* key is in the range $[1 : 14]$. That is, the universe under the grid distribution consists of $14^8 = 1,475,789,056$ different keys, and we use only the first n keys (in the sorted order). Thus, the grid distribution is also a different kind of dense distribution. Elements are randomly shuffled before insertion, and the set of lookup keys is also randomly shuffled.

5.4.4 Narrowing down our result set

Our overall set of experiments contained the combinations of many different dimensions, and thus the amount of raw information obtained exceeds legibility easily and makes the presentation of the study very difficult. For example, there are in total 24 different hash tables (hashing scheme + hash function). Thus, if we wanted to present all of them, *every* plot would contain 24 different curves, which is too much information for a single plot. Thus, we decided to present in this chapter only the most representative

set of results. Therefore, although we originally considered four different hash functions Mult, MultAdd, Tab, and Murmur, see Section 5.3, the following observations were uniform across *all* experiments: (1) **Mult is the fastest hash function when integrated with all hashing schemes, i.e., producing the highest throughputs and also of good quality (robustness), and thus it definitely deserves to be presented**. (2) MultAdd, when integrated with hashing schemes, has a robustness that falls between Mult and Murmur — more robust than Mult but less than Murmur. In terms of speed it was slower (in throughput) than Murmur. Thus we decided not to present MultAdd here and present Murmur instead. (3) Tabulation was indeed the strongest, most robust hash function of all when integrated with all hashing schemes. However, it is also the slowest, i.e., producing the lowest throughput. By studying the results provided by Mult and Murmur, we think that the trade-off offered by tabulation (robustness instead of speed) is less attractive in practice. Hence we do not present results for tabulation here.

In the end, we observed the importance of reducing operations during hash code computations as much as possible. The main reason for this observation is that hash computations contribute to the critical path of data dependencies for insert and lookup algorithms, as the accessed memory addresses depend on the results of hash function. Long dependency chains can heavily reduce the efficiency of pipelining in modern CPUs. Among the hash functions we consider, Mult is by far the lightest to compute — it requires only *one* multiplication and *one* right bit shift. MultAdd for 64-bit keys without 128-bit arithmetic [106] (natively unsupported on our server) requires two multiplications, six additions, plus a number of logical `AND`s and right bit shifts, which is more expensive than Murmur's 64-bit finalizer which requires *only* two multiplications and a number of `XOR`s and right bit shifts. As for tabulation, the eight table lookups per key ended up dominating its execution time. Assuming all tables remain in L1 cache, the latency of each table lookup is around 5-10 clock cycles. One addition requires one clock cycle and one multiplication at most five clock cycles (on Intel architectures). Thus, it is *very* interesting to observe and understand that, when hash code computation is part of hot loops during a workload (as in our experiments), we should really be concerned about how many clock cycles each computation costs — we could observe the effect of *even* one more instruction per hash code computation. We want to point out as well that **the situation of MultAdd changes if we use native 128-bit arithmetic, or if we use 32-bits keys with native 64-bit arithmetic (one multiplication, one addition, and one right bit shift). In that case we could use MultAdd instead of Murmur for the benefit of proven theoretical properties.**

5.4.5 On load factors for chained hashing

As we mentioned before, the load factor makes almost no sense for chained hashing since it can exceed one. Thus, throughout this chapter we refrain ourselves from using the formal definition of load factor together with chained hashing. We will instead study

chained hashing under memory budgets. That is, whenever we compare chained hashing against open-addressing schemes at a given load factor $\alpha = \frac{n}{l}$, what we do is that we modify the size of the directory of the chained hash table so that its overall memory consumption does not exceeds 110% of what open-addressing schemes require. In such a comparison, *all* hash tables will contain the *exact* same number n of elements. Thus, *all* hash tables compute the *exact* same number of hashes. In this regard, whether or not a chained hash table stays within memory constraints depends on the number of chained entries. Both variants of chained hashing considered by us cannot place more than a fraction of $16/24 < 0.67$ of the total of elements that an open-addressing scheme could place under the same memory constraint. If we take the extra 10% we grant to chained hash tables into account, this fraction grows to roughly 0.73. However, in practice this threshold is smaller (< 0.7) due to how collisions distribute over the table. This already strongly limits the usability of chained hashing under memory constraints and also brings up the following interesting situation. If chained hashing has to work under memory constraints, we can also try an open-addressing scheme for the exact same task under the same amount of memory. This potentially means lower load factors (< 0.5) for the latter. Depending on the hash function used, collisions might thus be rare, and the performance might become similar to a direct-addressing scheme — which is ideal. This might render chained hashing irrelevant.

5.5 Write-once-read-many (WORM)

In WORM we are interested in build and probe times (read-only structure) under six different load factors $25\%, 35\%, 45\%, 50\%, 70\%, 90\%$. These load factors are w.r.t. open addressing schemes on three different **pre-allocated capacities**[7]: 2^{16} (small — 1MB), 2^{27} (medium — 2GB) and 2^{30} (large — 16GB). This gives a total of up to **54 different configurations** (three data distributions, six load factors, and three capacities) for *each* of the **24 hash tables**. Due to the lack of space, and by our discussion offered on the load factors of chained hashing, we present here only the subsets of the large capacity presented in Figure 5.1.

The main reason for presenting only the large capacity is that "big" datasets are nowadays of primary concern and most observations can be transferred to smaller datasets. Also, we divided the hash tables this way because, by our explanation before, at low load factors collisions will be rare and performance of open-addressing schemes will be dominated by the simplicity of the used hash table — i.e., low code complexity. Thus we decided to compare the two variants of chained hashing against the simplest open-addressing scheme (linear probing)[8]. At a load factor of 50%, collision resolution of different open-addressing schemes start becoming apparent and thus from that

[7] WORM is a static workload. This means that the hash tables *never* rehash during the workload.

[8] For insignificant amounts of collisions, the performance of LP, RH, and QP is essentially equivalent.

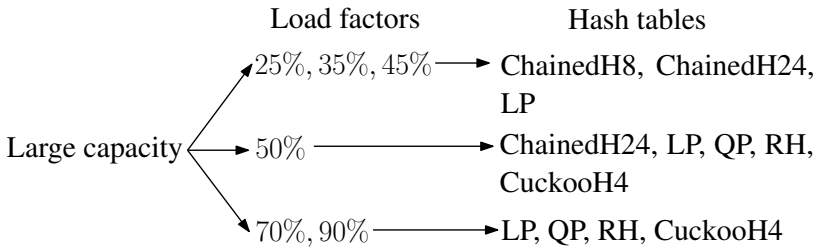


Figure 5.1: *Subset of results for WORM presented in this chapter.*

point on we include all open-addressing schemes considered by us. For chained hashing we consider only the best performing variant. For higher load factors ($\geq$ 70%), however, both variants of chained hashing could not place enough elements in the allocated memory. Thus we removed them altogether and study only open-addressing schemes.

5.5.1 Low load factors: 25%, 35%, 45%

In our very first set of experiments we are interested in understanding (1) the fundamental difference between chained hashing and open-addressing and (2) the trade-offs offer by the two different variants of chained hashing, see Section 5.2.1. The results can be seen in Figure 5.2.

Discussion

We start by discussing the memory footprints of all structures, see Figure 5.3. For linear probing, the footprint is constant (16GB), independent of the load factor, and easily determined only be the size of the directory, i.e., 2^{30} slots of 16B each. In ChainedH8, the footprint is calculated as size of directory, i.e. 2^{30} or 2^{29} slots, times the pointer size — 8B. In addition to that come 24B for each entry in the table. The footprint of ChainedH24 is computed as directory size, 2^{29}, times 24B, plus 24B for each collision. From this data we can obtain the amount of collisions for ChainedH24. For example, at load factor 35%, ChainedH24 requires 12GB for the directory, and all that goes beyond that is due to collisions. Thus, for the sparse distribution for example, ChainedH24 deals with $\approx$ 28% rate of collisions. But under the dense distribution, it deals only with $\approx$ 3% collision rate using Mult as hash function.

For performance results, let us focus on multiplicative hashing (Mult). Here, we can see a clear and stable ranking among the methods. For inserts, ChainedH24 performs better than ChainedH8. This is expected as the inlining of ChainedH24 helps to avoid caches misses for all occupied slots. Linear probing is, however, the top performer. This is because low load factors allow for many in-place insertions to the perfect slot.

In terms of lookup performance, we can also find a clear ranking among the chained hashing variants. Here, ChainedH24 performs best again. The superior performance of

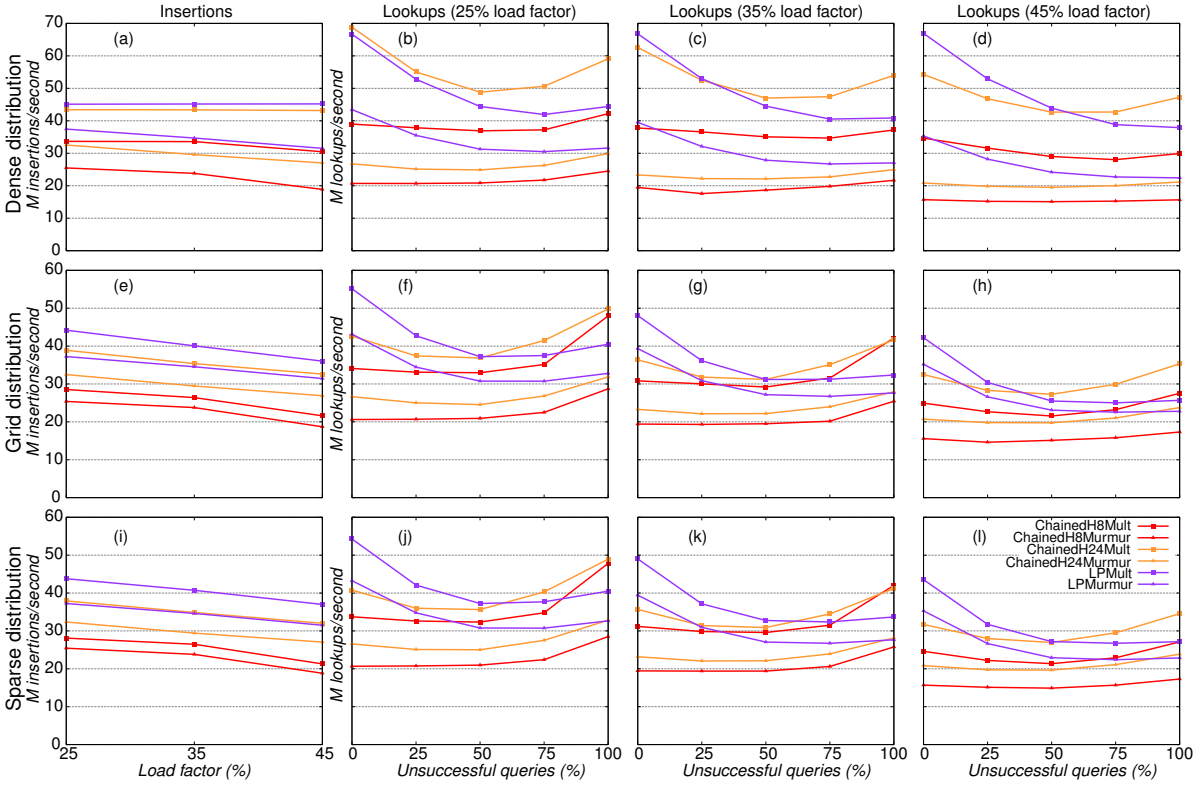


Figure 5.2: *Insertion and lookup throughputs, comparing two different variants of chained hashing with linear probing, under three different distributions at load factors* $25\%, 35\%, 45\%$ *from* 2^{30} *for linear probing. Higher is better.*

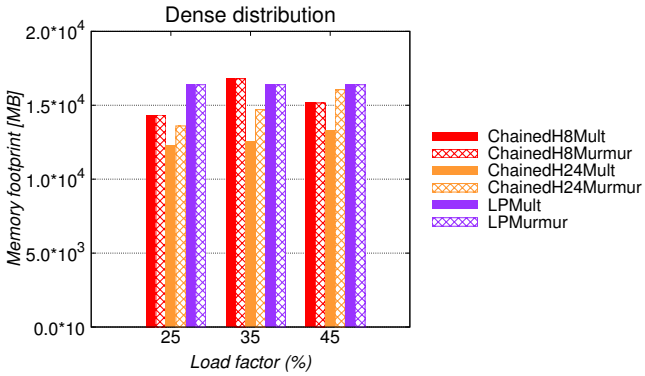


Figure 5.3: *Memory usage under the dense distribution of the hash tables presented in Figure 5.2. This distribution produces the largest differences in memory usage among hash tables. For the sparse and grid distributions, memory of ChainedH24Mult matches that of ChainedH24Murmur, and the rest remain the same. Lower is better.*

ChainedH24 is again easily explainable by the lower amount of pointer-chasing in the structure. We can also observe that between LP and ChainedH24, in all cases, one of the two structures performs best — but each in a different case and the order is typically determined by the ratio of unsuccessful queries. For all successful lookups, LP outperforms, in all but one case, all variants of chained hashing. The only exception is under the dense distribution at 25% load factor, Figure 5.2(b). There, both methods are essentially equivalent because the amount of collisions is essentially zero. The difference we observe is due to variance in code complexity, different penalties for branch misprediction, and different directory sizes — smaller directories lead to better cache behavior. Otherwise, in general, LP improves significantly over ChainedH24 if most queries are successful. In turn, ChainedH24 improves over LP, also by a significant amount in general, if most lookups are unsuccessful. We typically find the crossover point at around 50% unsuccessful lookups. We have found that this is also the point where the branch misprediction rate reaches for the lookup algorithms its maximum for all methods. **We observed, that branch misprediction has negative effects on the instruction pipeline and this penalty becomes clearly visible in our plots, as we vary the rate of unsuccessful lookups**. Interestingly, in some cases we can even observe ChainedH8 performing slightly better than LP for 100% unsuccessful lookups, when branch prediction improves again. This is explainable by a second effect, because even when collisions are rare, primary clusters can build up in linear probing (think of a continuous sequence of perfectly placed elements). For *every* unsuccessful query, LP has to scan until it finds an empty slot, which can entail long probe sequences. Long probe sequences obviously impact performance through additional cache misses, and furthermore, high variance among probe sequence lengths is also a secondary source of costly branch mispredictions. Thus, as the amount of unsuccessful queries increases, LP becomes considerably slower. In comparison to that, chained hashing answers unsuccessful queries right away if it detects an empty slot, or it will follow the linked list until the end. However, linked lists are very short on average. The highest observed collision rate is $\approx 34\%$ (sparse distribution at 45% load factor). This means that, at most, roughly one-third of the elements are outside the directory. Under the probabilistic properties of Mult, it can be argued that the linked list in chained hashing are in expectation of length at most 2, and thus chained hashing follows on average at most two pointers. **We can conclude that, at low load factors ($< 50\%$), LPMult is the way to go if most queries are successful ($\geq 50\%$), and ChainedH24 must be considered otherwise**.

5.5.2 High load factors: 50%, 70%, 90%

In our second set of experiments we study the performance of hash tables when space efficiency is required, and thus we are not able to use hash tables at low load factors. That is, we stress the hash tables to occupy up to 90% of the space assigned to them (chained hashing is allowed up to 10% more). We decided to use Cuckoo hashing on four tables, rather than on two or three tables, because this version of Cuckoo hashing is known to achieve load factors as high as 96.7% [31, 46] with high probability. In contrast, Cuckoo hashing on two and three tables have stable load factors of < 50 and up to $\approx 88\%$ respectively [86]. This means that if in practice we want to consider *very* high load factors ($\geq 90\%$), then Cuckoo hashing on four tables is the best candidate. An overview of the absolute best performers w.r.t. the other two capacities (small and medium) is given later as a table in Figure 5.5.

Discussion

Let us first start with a general discussion about the impact of distributions and hash functions on both, insert and lookup performance across all tables. Our first important observation is that Multiply-shift (Mult) performs essentially *always* better than Murmur hashing in this experiment. We can conclude from this that, overall, the improved quality of Murmur over Mult does not justify the higher computational effort. Mult seems already good enough to drive our five considered hash tables: ChainedH24, LP, QP, RP, and CuckooH4 up to the significantly high load factor of 90% — observe that *no* hash table is the absolute best using Murmur, see all plots of Figure 5.4. Another interesting observation is that, while we can see a significant variance in throughput under Mult across different data distributions — compare for example the throughputs of dense and sparse distributions under Mult — this variance is minimal under Murmur. This indicates that Murmur provides a very good randomization of the input data, basically transforming all input distribution into a distribution that is very close to uniform, and hence the distribution seems not to have much effect under Murmur [9]. However, sensitivity of a hash function to certain data distributions is not necessarily bad. For example, under the dense distribution[10] Mult is known [74] to produce an approximate arithmetic progression as hash codes, which reduces collisions. For a comparison, just observe that the dense distribution achieves higher throughputs than the sparse distribution that is usually considered as an unbiased reference of speed. We have observed that the picture does not easily change, even in the presence of a certain degree of gaps in the sequence of dense keys. Overall this makes Mult a strong candidate for dense keys, which appear *very* often in practice, e.g., for generated primary keys. In contrast to that, Mult is slightly slower on the grid distribution compared to the sparse distribu-

[9] We observed the same for Tab.

[10] Actually for generalized dense distributions following an arithmetic progression $k, k+d, k+2d, \ldots$

tion. We could observe that Mult produces indeed more collisions than the expected amount on uniformly distributed keys. However, this larger amount of collisions does not get highly reflected in the observed performance. Thus, **we consider Mult as the best candidate to be used in practice when quality results on high throughputs is desired, but at the cost of a high variance across data distributions**.

Let us now focus on the difference between the hash tables. We can see immediately that all open-addressing schemes, except CuckooH4, are better than ChainedH24 in almost all cases for up to 50% unsuccessful lookups, see Figure 5.4 (a, b, e, f, i, j). Only for the degenerated case of 100% unsuccessful lookups, ChainedH24 is the overall winner — for the same reasons as for low load factors. ChainedH24 is removed from the comparison for load factors > 50% because it exceeds the memory limit.

Between open-addressing schemes, things are more interesting. On insertions (left-most column of Figure 5.4), we can observe a rather clear ranking among methods that holds across all distributions and load factors. CuckooH4 is showing a very stable insert performance that is only slightly affected by increasing load factors. However, this performance is rather low. We can explain this result by the expensive reorganization that happens during Cuckoo cycles, and can often incur into several cache misses (whenever an element is moved between the tables) for a single insert. Unsurprisingly, LP, QP, and RH show rather similar insert performance characteristics because their insertion algorithm is very similar. Starting with high performance at 50% load factors, this performance drops significantly as the load factor increases. However, even under a high load factor, linearly and quadratically probing a hash table seems to be very effective. Among the three methods, we observe that RH is in general slightly slower than LP and QP. This is because RH performs small reorganizations on already inserted elements. However, these reorganizations often stay within one cache line, and thus the decrease in performance stays typically within less than 10%. With respect to QP and LP, the following are the most relevant observations. QP and LP have very similar insertion throughput for low load factors (up to 50). For higher load factors, when the difference in collision handling plays a role: (1) LPMult is considerable faster than QPMult under the dense distribution of keys (45M insertions/second versus 35M insertions/second — Figure 5.4(a)), and (2) QP (Mult/Murmur) is faster than LP (Mult/Murmur) otherwise. This is explainable: for (1) it suffices to observe that a dense distribution is the best case for LPMult – since Mult produces an approximate arithmetic progression (very few collisions). The best way to lay out an (approximate) arithmetic progression, in order to have better data locality, is to do so linearly, just as LP does. We could also observe that when primary clusters start appearing, they appear well distributed across the whole table, and they have similar sizes. Thus no cluster is arbitrarily long, which is good for LP. On the other hand, QP touches a new cache line in *every* probe subsequent to the third, and touching a new cache line results usually in a cache miss. Data locality is thus not optimal. For (2) the argument complements (1). Data is distributed more randomly, by

the hash function, across the table. This causes an increment in collisions w.r.t. the combination ⟨dense distribution + Mult⟩. For high load factors this increment in collisions means considerable long primary clusters that LP has to deal with. In this case, QP is a better strategy to handle collisions since it scatters collisions more sparsely across the table, and chances to find empty slots fast, over the whole sequence of insertions, are better than in LP with considerable long primary clusters.

For lookups we can find a similar situation as for inserts. LP, QP, and RH perform better than CuckooH4 in many situations, i.e., up to relatively high load factors. However, the performance of the former three significantly decreases with (1) higher load factors and (2) more unsuccessful lookups. We could observe that from a load factor of 80% on, CuckooH4 clearly surpasses the other methods. In general, LP, QP, and RH are better in dealing with higher collision rates than Cuckoo hashing, which is known to be negatively affected by "weak" hash functions [90] such as Mult. However, these "weak" hash functions affect *only* during the construction of the hash table, since once the hash table is constructed, then lookups in Cuckoo hashing are performed in constant time (four cache misses at most for CuckooH4). As such, Cuckoo hashing is also less affected by unsuccessful lookups than LP, QP, and RH. However, it seems that we can benefit from CuckooH4 only on very high load factors $\geq 80\%$.

As expected, the more complex re-organization that RH performs on the keys during insertions, see Section 5.2.4, can be seen to pay off under unsuccessful lookups — RH is much less affected by them than LP and QP. In RH, unsuccessful lookups can stop as soon as the displacement of the search key is exceeded by another key we encounter during the probe. Hence, RH does not necessarily require a complete scan of all adjacent keys in the same cluster, and can stop probing after less iterations than LP or QP. Clearly, this advantage of RH over LP and QP increases with higher load factors and higher rates of unsuccessful lookups — significantly improving on the worst-case of the methods. However, in the best of cases, i.e., when all lookups are successful, RH is slightly slower than the competitors. This is also expected as RH does not improve on the average displacement or amount of loaded cache lines w.r.t. LP (clusters contain *only* different permutations of the elements therein contained under RH and LP). When all lookups are successful, the (small) performance penalty of RH is due to its slightly more complex code. **We can conclude that RH provides a very interesting trade-off: for a small penalty (often within 1-5%) in peak performance on the best of cases (all lookups successful), RH significantly improves on the worst-case over LP in general, up to more than a factor 4**. Under the dense distribution — Figure 5.4 (a – c) — RH and LP have similar performance up to 70% load factor, but for 90% load factor, RH is significantly faster than LP (up to 40%) from 25% unsuccessful lookups on.

Across the whole set of experiments, RH is always among the top performers, and even the best method for most cases. This observation holds for all data set sizes we tested. In this regard, Figure 5.5 gives an overview and summarizes the abso-

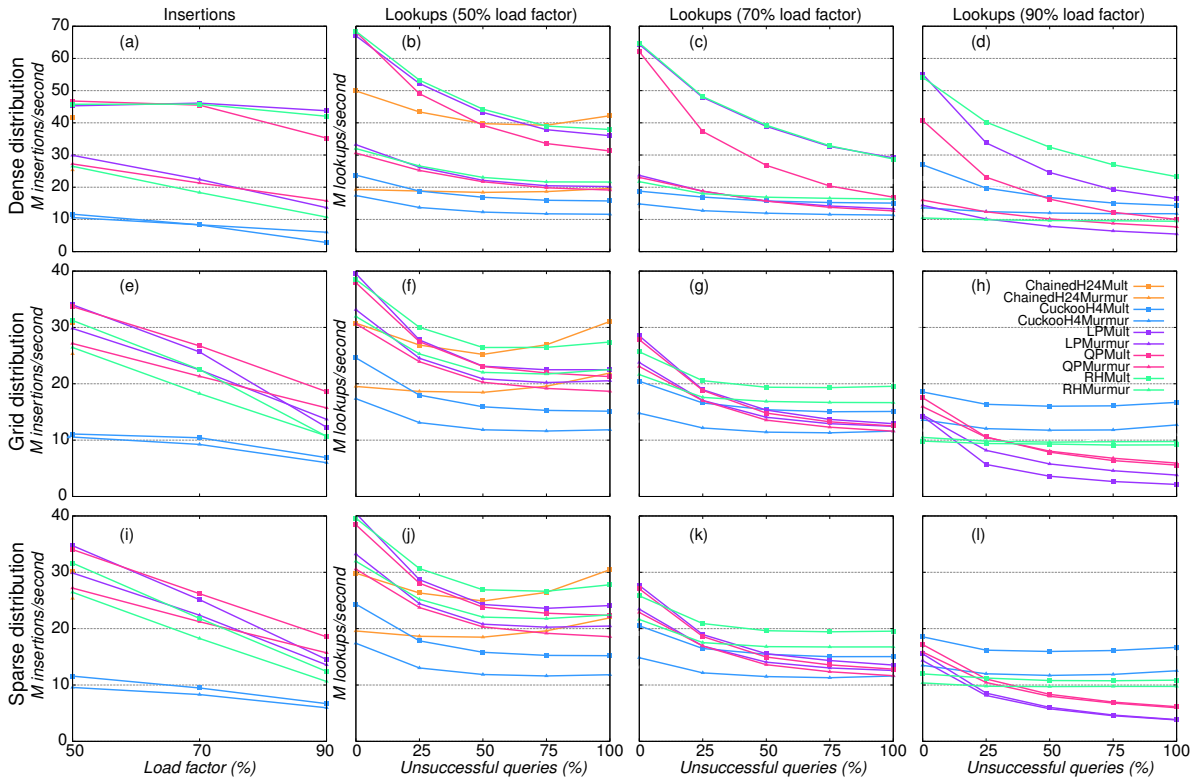


Figure 5.4: *Insertion and lookup throughputs, open-addressing variants and chained hashing, under three different distributions at load factors* $50\%, 70\%, 90\%$ *from* 2^{30}*. Higher is better. Memory consumption for all open-addressing schemes is* 16*GB, and* 16.4*GB for ChainedH24.*

lute best methods we tested in this experiment under all capacities (small, medium, and large). Methods are color-coded as in the curves in the plots. Observe that patterns are nicely recognizable. For lookups in general, RH seems to be an excellent all-rounder unless the hash table is expected to be very full, or the amount of unsuccessful queries is rather large. In such cases, CuckooH4 and ChainedH24 would be better options, respectively, if their slow insertion times are acceptable. With respect to insertions, it is natural not to see RH appearing more often, and certainly CuckooH4 and ChainedH24 not at all, due to their complicated insertion procedures. For insertions, QP seems to be the best option in general. Even when LP or RH are sometimes better, the difference is rather small, less than 10%.

Dist.	Load factor	Capacity	Insertions	Unsuccessful queries (%) 0%	25%	50%	75%	100%
Dense	50%	S	290	345	126	91	114	188
		M	56	76	56	44	46	51
		L	46	68	53	44	39	42
	70%	S	289	346	113	71	62	63
		M	56	75	48	35	30	29
		L	46	64	48	39	32	28
	90%	S	169	183	80	63	61	64
		M	51	60	37	27	22	21
		L	43	55	40	32	26	23
Grid	50%	S	194	212	95	72	85	124
		M	38	43	33	29	28	32
		L	34	39	30	26	26	31
	70%	S	113	119	67	57	56	58
		M	28	30	24	22	21	21
		L	25	28	20	19	19	19
	90%	S	72	90	52	47	49	52
		M	20	22	19	19	20	20
		L	18	18	16	16	16	16
Sparse	50%	S	104	109	68	63	73	103
		M	39	44	33	29	29	33
		L	34	40	30	26	26	30
	70%	S	77	74	52	50	51	53
		M	28	30	22	21	20	20
		L	26	27	20	19	19	19
	90%	S	57	59	51	53	58	66
		M	19	22	19	19	20	21
		L	18	18	16	15	16	16

CuckooH4Mult LPMult QPMult RHMult ChainedH24

Figure 5.5: *Absolute best performers for the WORM workload (Section 5.5.2) across distributions, different load factors, and different capacities: Small (S), Medium (M) and Large (L). Throughput of the corresponding hash table is shown inside its cell in millions of operations per second.*

5.6 Read-write workload (RW)

In RW we are interested in analyzing how growing (rehashing) over a long sequence of operations affects overall throughput and memory consumption. The set of operations we consider is the following: insertions, deletions (all successful), and lookups (successful and unsuccessful). In RW we let the hash tables grow over a set of 1000 million operations that appear in random order. Each hash table initially contains 16

millions keys[11]. We set the insertion-to-deletion ratio (updates) to 4:1 (20% deletions), and the successful-to-unsuccessful-lookup ratio to 3:1 (25% unsuccessful queries). For this kind of workload we present here only the results concerning the sparse distribution of keys. We consider three different thresholds for rehashing: at 50%, 70%, and 90%. Rehashing at 50% allows us to always have enough empty slots, and thus also less collisions. However, this also means a potential loss in space since the workload might stop short after growing, and thus up to 75% of the hash table could be empty. On the other hand, rehashing at 90% deals with a large amount of collisions as the table gets full, but then we potentially waste less space. In addition to that, high load factors will incur into slow lookup times before a rehash. Observe again that by the natural load factors of Cuckoo hashing on two and three tables, Cuckoo hashing on four tables is the best candidate again for controlling at what load factor the hash table must rehash. For chained hashing, similar to the situation in WORM, we present here only the case where rehashing is performed at 50% load factor. This is the only case in which we can keep memory consumption of ChainedH24 comparable to what the open-addressing schemes require. The results of these experiments are shown in Figure 5.6.

Discussion

With respect to the performance in the WORM scenario on high load factors — Section 5.5.2 — the outcome of the RW comparison offers few surprises. One of these surprises is to see that ChainedH24 offers better performance than CuckooH4 (50% load factor only), and sometimes even by a large margin. However, both lag clearly behind the other (open-addressing) schemes. As RW workload is write-heavy, what we see in the plots is mostly the cost of table rehashing — except for data points at 0% updates. In that case, what we see are only lookups with 25% of unsuccessful queries, see Figure 5.4(j) for a comparison. For CuckooH4 the gap narrows as the load factor increases, see Figure 5.6(c), but is not enough to become really competitive with the best performers — which are at least twice as fast as the updates become more frequent. **As a conclusion, although memory requirements of ChainedH24 and CuckooH4 are competitive with that of the other schemes in a dynamic setting, both — chained and Cuckoo hashing — should be avoided for write-heavy workloads**.

We can also see that Mult wins again over Murmur on all hash tables — Figure 5.6 (a – c). Which is to be expected since the hash tables rehash many times and thus hash function computations are fundamental. Also, we always find LP, QP, and RH as the fastest methods, and often with very similar performance. Growing at 50% load factor — Figure 5.6(a) — the difference in throughput of all three methods is mostly within the boundary of variance. In case of high update percentage (> 50%), we can observe a small performance penalty for RH in comparison to LP and QP, which is due to the

[11] In the beginning (no updates), the hash tables have a load factor of roughly 47%.

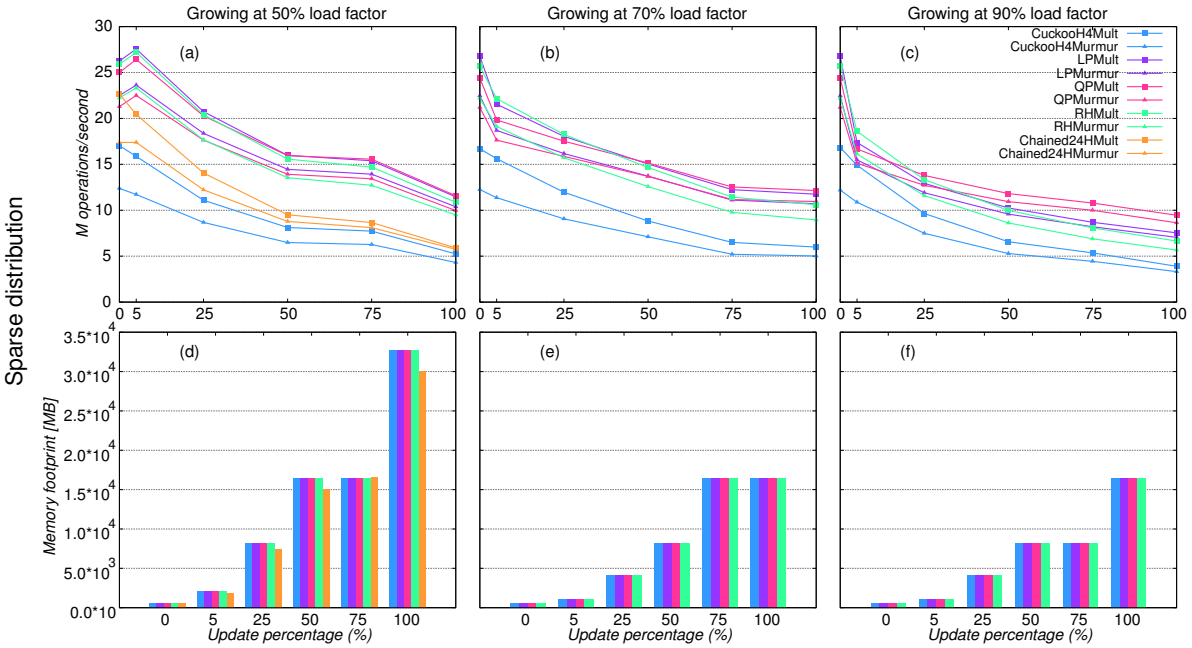


Figure 5.6: *1000M operations of RW workload under different load factors and update-to-lookup ratios. For updates, the insertion-to-deletion ratio is 4:1. For lookups, the successful-to-unsuccessful-lookup is ratio 3:1. The key distribution is sparse. Higher is better in performance, lower is better for memory.*

slightly slower insert performance that we already observed in the WORM benchmark, see Figure 5.4(i). This is expected because at 50% load factor, there are few collisions, and more sophisticated strategies for handling collisions cannot benefit as much. At 70% and 90% load factors — Figures 5.6(b) and 5.6(c) — all three methods are getting slower, and we can also observe a clearer difference between them because different strategies have an impact now. Interestingly, with increasing load factor and update ratios, QP is showing the best performance, with LP being second and RH in third place. This is consistent with our observation in the WORM experiment that QP is best for inserts on high load factors and RH is typically the slowest. **As a conclusion, in a write-heavy workload, quadratic probing looks as the best option in general**.

5.7 On table layout: AoS or SoA

One fundamental question in open-addressing is whether to organize the table as an *array-of-structs* (AoS) or as a *struct-of-arrays* (SoA). In AoS, the table is internally represented as array of key-value pairs whereas SoA keeps keys and values separated in two corresponding, aligned arrays. Both variants offer different performance characteristics and tradeoffs. These tradeoffs are somewhat similar to the difference between row and column layout for storing database tables. In general, we can expect to touch less cache-lines for AoS when the total displacement of the table is rather low, ideally just one cache line. In contrast to that, SoA already needs to touch at least two cache lines for each successful probe (one for the key and one for the value) in the best case. However, for high displacement (and hence longer probe sequences) SoA layout offers the benefit that we can just search through keys only, thus scanning up to only half the amount of data compared to AoS, where keys and values are interleaved. Another advantage of SoA over AoS is that a separation of keys from values makes vectorization with SIMD easy, essentially allowing us to load and compare four densely packed keys at a time on 256-bit SIMD registers as offered on current AVX-2 platforms. In contrast to that, comparing four keys in AoS with SIMD requires to first extract only the keys from the key-value pairs into the SIMD register, e.g., by using gather-scatter vector addressing which we found to be not very efficient on current processors. Independent of this, AoS also needs to touch up to two times more cache lines for long probe sequences compared to SoA when many keys are scanned.

In the following, we present a micro-benchmark to illustrate the effect of different layout and SIMD for inserts and lookups in linear probing. Since our computing server does not support AVX-2 instructions, we ran this micro-benchmark on a new MacBook Pro as described in Section 5.4. We implemented key comparisons with SIMD instructions for lookup and inserts on top of our existing linear probing hash tables by manually introducing intrinsics to our code. For example, in AoS, we load four keys at a time to a SIMD register from an cache-line-aligned index, using the `_mm256_load_si256` command. Then we perform a vectorized comparison on the four keys using `_mm256_cmpeq_epi64` and, in case of one successful comparison, obtain the first matching index with `_mm256_movemask_pd`.

We compare LPMult in AoS layout against LPMult in SoA layout with and without SIMD on a sparse data set. Similar to the indexing experiment of Section 5.5.2, we measure the throughput for insertions and lookups for load factors $50, 70, 90\%$. Due to the limited memory available on the laptop, we use the medium table capacity of 2^{27} slots — 2GB. This still allows us to study the performance outside of caches, where we expect layout effects to matter most, because touching different cache lines typically triggers expensive cache misses. Figure 5.7 shows the results of the experiment.

Discussion

Let us start by discussing the impact of layout *without* using SIMD instructions, methods LPAoSMult and LPSoAMult in Figure 5.7. For inserts (Figure 5.7(a)), AoS performs up to 50% better than SoA, on the lowest load factor (50%). This gap is slowly closing with higher load factors, leaving AoS only 10% faster than SoA on load factor 90%. This result can be explained as follows. When collisions are rare (as on load factor 50), SoA touches two times more cache lines than AoS — it has to place key and value in different locations. In contrast to that, SoA can fit up to two times more keys in one cache line than AoS, which improves throughput for longer probes sequences when searching empty slots under high load factors. However, when beginning inserting into an empty hash table, we can often place the entry into its hash bucket without any further probing. Only over time we will require more and more probes. Thus, in the beginning, there is a high number of insertions where the advantage of AoS has higher impact. This is also the reason why the gap in insertion throughput between AoS and SoA significantly narrows as the load factor increases.

For lookups (Figures 5.7(b — d)) we noticed overall that AoS is faster than SoA on short probe sequences, i.e., especially for low load factors and low rates of unsuccessful queries. On the lowest load factor (50%, Figure 5.7(b)), we can see that in the best case (all queries successful) AoS typically encounters half the number of cache misses compared to SoA, because keys and values are adjacent. This is reflected in a 63% higher throughput. With increasing unsuccessful lookup rate, the performance of SoA approaches AoS and the crossover point lies around 75% unsuccessful lookups. For 100% unsuccessful lookups, AoS improves over SoA by 15%. For load factor 70% (Figure 5.7(c)), AoS is again superior to SoA for low rates of unsuccessful queries, but the crossover point at which SoA starts being beneficial shifted to 25% unsuccessful queries instead of 75% of the 50% load factor. Interestingly, we can observe that for load factor 90% (Figure 5.7(d)), the advantage of SoA over AoS layout is unexpectedly low — with the highest difference observed being around 30% instead of close to a factor 2 as we could expect. Our analysis obtained a combination of three different factors

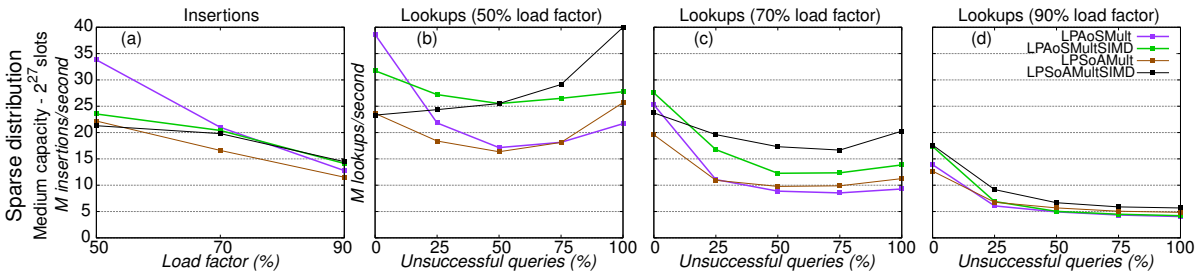


Figure 5.7: *Effect of layout and SIMD in performance of LPMult at load factors* $50, 70, 90\%$ *w.r.t.* 2^{27} *under a sparse distribution of keys. Higher is better.*

that explain this result. First, even in the extreme case of 100% unsuccessful lookups, the difference in touched caches lines is not a factor 2. The combination ⟨sparse distribution, Mult⟩ simulates the ideal case that every key is uniformly distributed over the hash table. Thus, we know [74] that the average number of probes in an unsuccessful search in linear probing is roughly $\frac{1}{2}\left(1 + \left(\frac{1}{(1-\alpha)^2}\right)\right)$, where α is the load factor of the table. Thus, for 90% load factor the average probe length is roughly 50.5 (we could verify this experimentally as well). Now, in AoS we can pack four key-value pairs into a cache line, and twice as much (eight) for SoA. This means that the average number of loaded cache lines in AoS and SoA is roughly $\frac{50.5}{4}$ and $\frac{50.5}{8}$ respectively. However, *in practice* this behaves like $\lceil\frac{50.5}{4}\rceil = 13$ and $\lceil\frac{50.5}{8}\rceil = 7$ respectively — since whole cache lines are loaded. Which means that AoS loads *only* roughly 1.85× more caches lines as SoA — which we were also able to verify experimentally. In addition to that, a second factor are nonuniform costs of visiting cache lines. We observed that the first probe in a sequence is typically more expensive than the subsequent linear probes because the first probe is likely to trigger a TLB miss and a page walk, which amortizes over visiting a larger amount of adjacent slots. The third factor is that, independent from the number of visited cache lines, the number of hash computations, loop iterations, and key comparisons are identical for SoA and AoS. Those parts of the probing algorithm involve data dependencies that build up a long critical path in the pipeline. Long chains of data dependencies prevent modern processors from hiding memory latencies and make out-of-order execution less effective. In addition to that, the concrete length of probe sequences also suffers from high variance for both layouts, which negatively affects branch prediction so that the misprediction penalty overshadows parts of the layout effects. **In conclusion, the ideal advantages of SoA over AoS are less strong in practice due to the way hardware works**.

We now proceed to discuss the impact of SIMD instructions in both layouts. In general, SIMD allows us to compare up to four 8-byte keys (or half a cache line) in parallel, with one instruction. However, this parallelism typically comes at a small price because loading keys into SIMD registers and generating a memory address from the result of SIMD comparison (e.g., by performing count-trailing-zeros on a bit mask) potentially introduce a small overhead in terms of instructions. In case of writes that depend on address calculation based on the result of SIMD operations, we could even observe expensive pipeline stalls. Hence, in certain cases, SIMD can actually make execution slower, e.g., see Figure 5.7(a). For lower load factors, using SIMD for insertions can decrease performance significantly for both AoS and SoA layout, by up to 64% in the extreme case. However, there is a crossover point between SIMD and non-SIMD insertions around 75% load factor. We found that in such cases, SIMD is up to 12% faster than non-SIMD.

For lookups, we can observe that SIMD improves performance in almost all cases. We notice, that in general, the improvement of SIMD is higher for SoA than for AoS.

As mentioned before, SoA layout simplifies loading keys to a SIMD register, whereas AoS requires us to gather the interleaved keys in a register. We observed that on the Haswell architecture, gathering is still a rather expensive operation and this difference gives SoA an edge over AoS for SIMD. As a result, we find SoA-SIMD superior to plain SoA in all cases for lookups, with improvement of up to up to 81% (Figure 5.7(b)). We observed that AoS-SIMD can be up to 17% harmful for low load factors, but beneficial for high load factors.

In general, we could observe in this experiment that AoS is significantly superior to SoA for insertions — even up to very high load factors. **Our overall conclusion is that AoS outperforms SoA by a larger margin than the other way around. Inside caches (not shown), both methods are comparable in terms of lookup performance, with AoS performing slightly better. When using SIMD, SoA has an edge over AoS — at least on current hardware — because keys are already densely packed**.

5.8 Conclusions and future work

All the knowledge we gathered leads us to propose a decision graph, Figure 5.8, that we hope can help practitioners to decide more easily what hash table to use in practice under different circumstances. Obviously, no experiment can be complete enough to fully capture the true nature of *all* hash tables in *every* situation. Our suggestions are, nevertheless, educated as a result of our large set of experiments, and we are confident that they represent very well the behavior of the hash tables. We also hope that our study makes practitioners more aware about trade-offs and consequences of not carefully choosing a hash table.

We stated our conclusions in an inline fashion throughout the chapter, but we would still like to summarize some important points here in a *very* condensed manner:

1. Consider chained hashing *only* when memory is not strongly constraint and the amount of unsuccessful queries is known to be significantly larger (> 50% of all queries). Otherwise consider using an open-addressing scheme (potentially at a high load factor).

2. Linear probing and variants are surprisingly good in conjunction with relatively cheap but decent hash functions like Mult, and this combination should strongly be considered in practice. We have shown the most relevant trade-offs.

3. Every single instruction can matter in long data dependency chains because of their effects on pipelining in modern processors. This becomes clearly visible when considering the performance differences between hash functions, because their computation contributes to the critcal path length of hash table algorithms. From this perspective, we suggest to use the most lightweight hash function that

fulfills the particular quality needs of the use case at hand. Furthermore, we plan to to investigate the exact circumstances and implications of the observed effects on pipelining in a future work.

4. Mult, see Section 5.3.1, as a hash function is extremely efficient and seems to be robust enough for typical database use-cases. In particular Mult on a dense distribution (often found in databases) is an excellent choice since Mult produces an approximate arithmetic progression, thus severely reducing collisions (w.r.t. random distribution for example).

Finally, let us conclude that memory layout has a significant impact on table performance and it is interesting to study the middle ground between SoA and AoS — taking into account hardware granularities such as cache lines or pages. For example, for AoS, it might be beneficial to separate corresponding keys and value but keep them separated in the two halves of the same cache line. This could yield the benefits of AoS in terms of minimal cache misses and the benefits of SoA w.r.t. the use of SIMD instructions. Furthermore, for SoA, we could store keys and their respective values separated in the two halves of the same memory page — thus avoiding additional TLB misses for large table sizes.

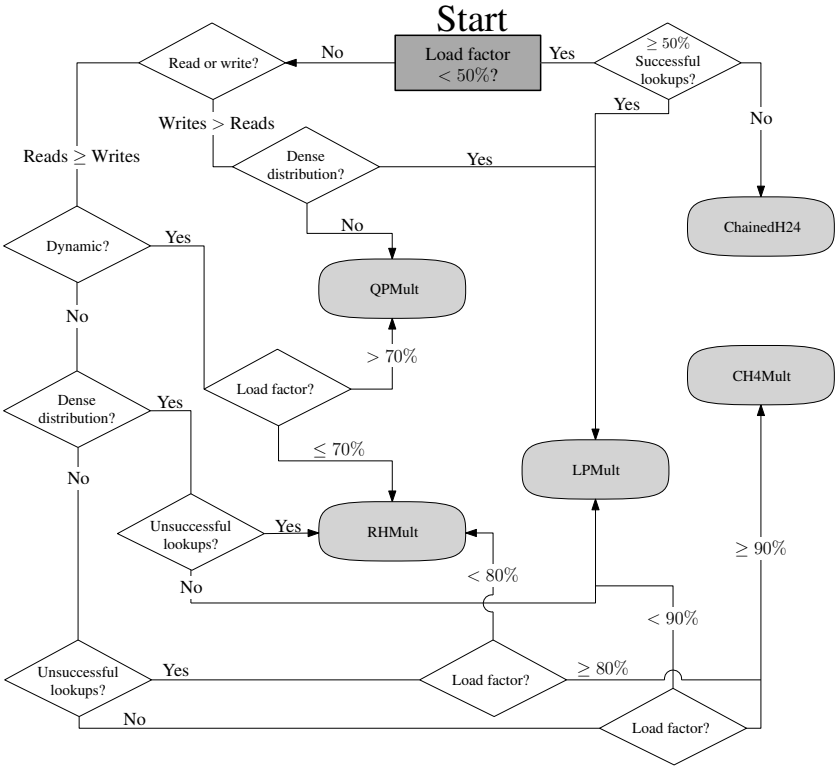


Figure 5.8: *Suggested decision graph for practitioners.*

Appendix A

Mosquito: Another One Bites the Data Upload STream

A.1 Abstract

Mosquito is a lightweight and adaptive physical design framework for Hadoop. Mosquito connects to existing data pipelines in Hadoop MapReduce and/or HDFS, observes the data, and creates better physical designs, i.e. indexes, as a byproduct. Our approach is minimally invasive, yet it allows users and developers to easily improve the runtime of Hadoop. We present three important use cases: first, how to create indexes as a byproduct of data uploads into HDFS; second, how to create indexes as a byproduct of map tasks; and third, how to execute map tasks as a byproduct of HDFS data uploads. These use cases may even be combined.

A.2 Introduction

Hadoop is a popular data processing engine in the context of cloud computing, NoSQL, and Big Data. In the past years, the DB community has taught efficiency to Hadoop MapReduce and its distributed file systems HDFS in several ways. An important family of techniques has investigated on how to use better physical layouts [66], clustered indexes [35, 36], and adaptive indexes [99]. Though these technique can always be implemented in a traditional way by using Hadoop MapReduce jobs to create indexes on top of its file system HDFS — similar to a traditional DBMS using a physical design engine on top of a file system, this approach has a severe drawback: data is read and written several times across the two layers. As HDFS is agnostic about Hadoop MapReduce, considerable time is wasted doing things twice in the two layers that could be combined effectively if the two layers were a single layer. This is prohibitevly expensive in an environment handling Petabytes of data. It makes physical design expensive.

And it dramatically increases MapReduce job latencies, be it at data upload or be it at MapReduce job execution. In the context of a distributed system the separation of data storage and data processing into two layers is a pain.

The reader might recognize this line of thought: it is a vanilla software engineering argument from our DB courses: for highest efficiency it is a great decision to get rid of all interfaces and layers and pack all code into a single monolithic block. However, such a system becomes unmaintainable quickly. Especially if the system is developed as an open source project by a large community — like Hadoop. What happens if fundamental things change in Hadoop's code base? Who makes sure that the techniques we taught to Hadoop will still work with the next release?

An obvious idea to fix this problem is the other extreme of a system design: use many interfaces and layers. Whatever technique you want to teach to Hadoop, implement them in another layer: make sure you implement them against system- or UDF-interfaces [35, 67], i.e. whatever you do, stick to the existing interfaces. However, such systems quickly become inefficient. In addition, the impact of your optimizations is limited by the interfaces that were provided by those systems in the first place, even when using UDFs [35, 67]. Moreover, even though these approaches do not need to touch the source code of the software layer underneath, the limitedness of the system interfaces often forces you to reimplement considerable parts of its functionality. For instance, the recently proposed [65] does not need to change HDFS. However, it needs to reimplement failover, data placement, as well as load balancing. This reduces the role of HDFS to a simple local file system with network access.

To fix this, in this demo we introduce a novel approach coined *Mosquito*. Our system sits in-between the two extremes in the system design space. We allow Mosquito to connect to data pipelines and streams available on lower layers, be it HDFS or Hadoop MapReduce. Yes, like this we break the layering of these systems at small, yet clearly defined points. Yet, with this approach we are able to reduce the maintenance effort of Mosquito to a minimum, but at the same time we are able to perform crosslayer optimizations. These optimizations lead to order of magnitude runtime improvements.

A.3 Mosquito Overview

Mosquito is a software framework allowing developers to easily connect to data streams in Hadoop. Currently Mosquito supports three major scenarios: (1) Aggressive Indexing, i.e. HDFS blocks may be indexed as a side-effect of uploading data into HDFS. All physical replicas of a logical HDFS block may be kept in different sort orders; (2) Adaptive Indexing, i.e. HDFS blocks get indexed at query time as a side-effect of query processing. For every incoming MapReduce job a fraction of the HDFS blocks pertaining to a file are indexed; and (3) Aggressive Map Execution, i.e. the map phase of

a MapReduce job may be executed as a side-effect of uploading data into HDFS already. The three scenarios may even be combined.

A.3.1 Aggressive Indexing

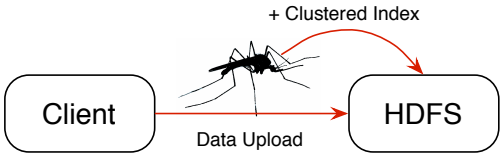


Figure A.1: *Mosquito aggressive indexing as a side-effect of HDFS data upload.*

Mosquito Aggressive Indexing allows users to efficiently create different clustered indexes over terabytes of data as a side-effect of uploading their dataset to HDFS. Mosquito can support different sort orders (and layouts), one for each physical replica of the data *without* affecting Hadoop's data placement and failover properties. Like this Mosquito can fully emulate HAIL Static Indexing [36].

Overall, we will demonstrate that Mosquito indexes can dramatically improve the runtimes of several classes of MapReduce jobs while index creation is basically invisible to the user in terms of upload time overhead. Figure A.1 sketches the idea of a Mosquito biting into the data upload pipeline: whenever a user uploads a new dataset through the HDFS client, the data is partitioned into HDFS blocks and those blocks are shipped and replicated to HDFS data nodes for storage. Mosquito intercepts block storage. While HDFS data blocks are loaded into main memory Mosquito creates user defined indexes, typically one for each block replica, before actually storing the reordered HDFS blocks on the data nodes.

A.3.2 Adaptive Indexing

Mosquito Adaptive Indexing allows users to efficiently create different clustered indexes over terabytes of data as a side-effect of query processing. In contrast to adaptive indexing in main memory [61], for every MapTask we collect a subset of the HDFS blocks and create full indexes on those blocks. In addition, again, this also allows us to keep all replicas in sync and keep HDFS' failover properties. Like this Mosquito can fully emulate HAIL Adaptive Indexing [99].

Our motivation for Mosquito adaptive indexing is a scenario where users want to apply selections (using Mosquito annotations) on attributes that where not indexed at data upload time. For example, this can easily happen when the selection criteria are hard to predict in advance or whenever workloads change over time. Mosquito adaptive

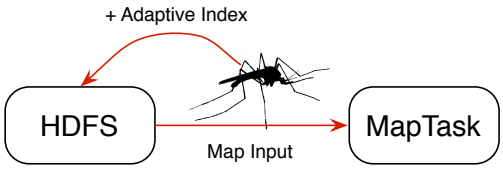


Figure A.2: *Mosquito adaptive indexing as a side-effect of a MapTask execution.*

indexing sits on top of Hadoop's MapReduce job execution. The core idea is to create missing but promising indexes as byproducts of full scans in the map phase of MapReduce jobs. Similar to aggressive indexing, our goal is again to create additional indexes without significant overhead on individual job runtimes. Mosquito piggybacks on another procedure that is naturally reading data from disk to main memory. This allows Mosquito to completely save the data read cost for adaptive index creation. Second, as map tasks are usually I/O-bound, Mosquito can again exploit unused CPU time for computing clustered indexes in parallel to job execution. Figure A.2 illustrates the core concept of the Mosquito adaptive indexing pipeline.

A.3.3 Aggressive Map Execution

Mosquito Aggressive Map Execution allows users to efficiently run one or several map phases as a side-effect of uploading data into HDFS. This means each data node receiving data to store already executes a MapTask on that data before writing it to disk.This is interesting for cases where the map-functions to execute are already known at data upload time. Like this Mosquito can be run in 'NoHadoop'-mode: Mosquito Aggressive Map Execution allows users to execute MapReduce jobs while uploading their dataset to HDFS. This means that users can immediately start analyzing their data instead of waiting for their initial upload to finish. Our Aggressive Map Execution is illustrated in Figure A.3 and works on top of the HDFS upload pipeline as follows: (1) The user uploads her data with the HDFS upload command and additionally provides one (or a set of) MapReduce job(s) to execute on that data. (2) The client splits the data into blocks and these blocks into packets. For each block, the client sends those packets to the first data node for storage. (3) On each data node, those packets are persisted on local disk and forwarded to the next data node if applicable, just like in normal HDFS. However, in parallel, one data node that stores a block replica is chosen by the job scheduler to reassemble this data block from the packets in main memory and spawn a new map tasks for the provided job. Since data block replicas are distributed over the cluster, the scheduler can parallelize the tasks on the cluster similar to normal Hadoop. Whenever a map tasks fails, it is rescheduled after the upload phase was completed. As a result, our system can save the complete read costs of the map tasks while preserving full failover properties. (4) After the upload (and hence the map phase) is completed, Mosquito runs

a reduce phase as in normal Hadoop. Notice that, Mosquito could also be used to chain multiple MapReduce jobs, e.g. for iterative computations: in the end of a reduce task, when the output of a job is written back to HDFS, the consecutive map task may already be executed using the technique of Aggressive Map Execution.

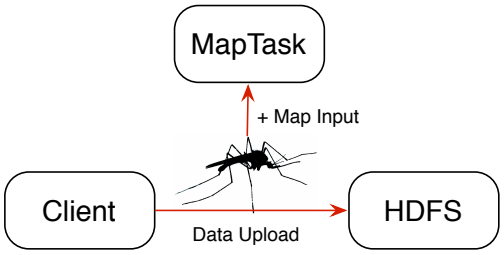


Figure A.3: *Aggressive Map Execution as a side-effect of HDFS data upload.*

A.4 Demonstration and Use Cases

Mosquito offers interfaces to plug user defined operations on top of ongoing data movement in Hadoop clusters. As a result, the Mosquito framework acts as a flexible platform that greatly simplifies the realization of many optimization techniques for Hadoop's data storage and job execution pipeline, such as indexing, layout transformation or ad-hoc job execution. In the following, we will describe our demo setup (Section A.4.1) and three use cases that demonstrate possible Mosquito applications (Section A.4.2).

A.4.1 Demo Setup

In our demo, we compare the performance of our Mosquito applications to standard Hadoop in order to better understand the benefits of using Mosquito. We use our local 10-node cluster at Saarland University. Each cluster node has two Intel Xeon E5-2407 2.20 GHz processor, 48GB of main memory and 2TB HDD. For different demo scenarios we visualize the performance with respect to job runtimes and data upload times.

A.4.2 Use Cases

Mosquito emulating HAIL Static Indexing

In our first use case, we leverage Mosquito to emulate HAIL [36]. The goal of this demo scenario is to (i) illustrate how Mosquito can easily create several clustered indexes in parallel to uploading a dataset to HDFS and (ii) to show how such indexes can dramatically decrease the runtimes of selective MapReduce jobs. This scenario represents a

typical analytical use case, where the user wants to create one or more indexes on a large dataset, e.g. a weblog, and afterwards exploits the indexes to speed up queries. We invite the audience to specify the clustered indexes to create and to compare the upload times for the weblog dataset of Mosquito with the ones of standard HDFS. Then, the audience can edit and enhance MapReduce jobs with Mosquito annotations and run the jobs on the previously uploaded dataset. Finally, we report the runtime improvements of Mosquito in comparison to normal Hadoop MapReduce.

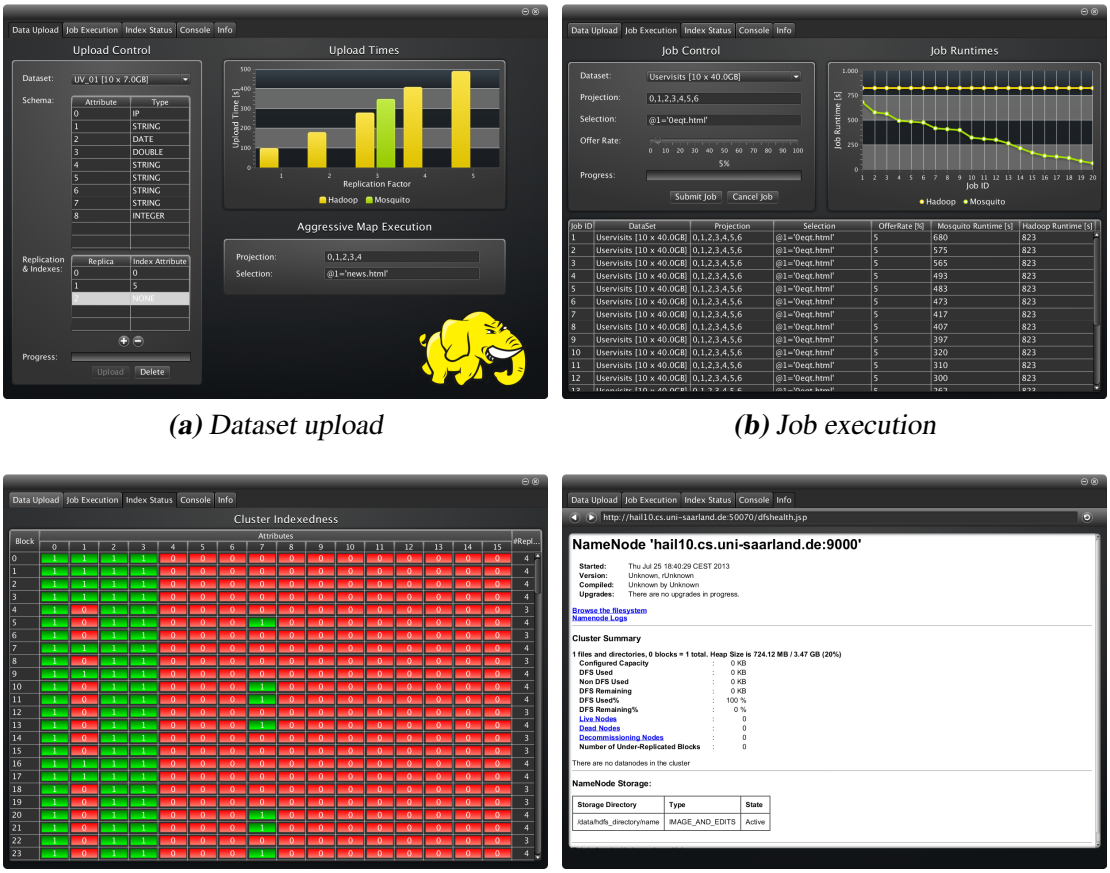
Mosquito emulating HAIL Adaptive Indexing

In our second use case, we configure Mosquito to emulate adaptive indexing as presented in LIAH [99]. In this scenario, we show how Mosquito can be used to realize pluggable adaptive indexing capabilities on top of Hadoop MapReduce. In more detail, we show how Mosquito exploits running map tasks to incrementally build missing indexes with minimal or no runtime overhead per job. This approach proved useful in applications where the query workload is unknown at data upload or changes over time. Our Mosquito GUI, as shown in Figure A.4, allows the audience to edit and schedule sequences of annotated MapReduce jobs. Additionally, the audience can configure runtime parameters, such as the offer rate[1]. We plot the runtimes for the job sequence executed on Mosquito and standard Hadoop. Thereby, the audience can observe the gradual runtime improvement of adaptive indexing. Furthermore, an index map visualizes the progress of index creation while executing the job sequence.

Mosquito running NoHadoop

Our third use case for the Mosquito framework is 'NoHadoop'-mode, an approach for ad-hoc job execution on top of data uploads to HDFS. With NoHadoop, users no longer have to wait for their data being uploaded to HDFS before running their MapReduce jobs. Instead, they can immediately start running MapReduce jobs while their data is s being uploaded to HDFS. Consequently, NoHadoop eliminates the upload-to-job time, which is the fundamental measure for the delay before Hadoop can actually start to execute jobs on new data. We encourage the audience to benchmark Mosquito NoHadoop against normal Hadoop for one or more jobs. Overall, we show the abilities of Mosquito NoHadoop to reduce upload-to-job time as well as total runtimes of typical MapReduce workflows dramatically.

[1]The offer rate defines the maximum percentage of data blocks from the input dataset that can be indexing in parallel to a single MapReduce job.



(a) Dataset upload *(b)* Job execution

(c) Indexing status *(d)* Browse cluster information

Figure A.4: *Graphical User Interface of Mosquito*

List of Figures

List of Tables

Bibliography

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[2] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT*, pages 1–10, 2013.

[3] Sanjay Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. *VLDB*, pages 1110–1121, 2004.

[4] Anastassia Ailamaki et al. Weaving Relations for Cache Performance. *VLDB*, pages 169–180, 2001.

[5] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD Conference*, pages 241–252, 2012.

[6] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. A Comparison of Adaptive Radix Trees and Hash Tables. In *31st IEEE ICDE*, April 2015.

[7] Austin Appleby. MurmurHash3 64-bit finalizer. Version 19/02/15. `https://code.google.com/p/smhasher/wiki/MurmurHash3`.

[8] Nikolas Askitis and Ranjan Sinha. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal*, 19(5):633–660, 2010.

[9] Ching Avery. Giraph: Large-scale graph processing infrastructure on Hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[10] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Ozsu. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE TKDE*, 2014.

[11] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-Efficient Hash Joins. *VLDB*, 8(4), 2014.

[12] Doug Baskins. Judy Arrays. `http://judy.sourceforge.net/` Version 31/07/14.

[13] Timo Bingmann. STX B+-tree implementation. `http://panthema.net/2007/stx-btree/` Version 31/07/14.

[14] Spyros Blanas et al. A Comparison of Join Algorithms for Log Processing in MapReduce. *SIGMOD*, pages 975–986, 2010.

[15] Matthias Boehm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. Efficient In-Memory Indexing with Generalized Prefix Trees. In *BTW*, volume 180, pages 227–246, 2011.

[16] Petter Bae Brandtzaeg. Big Data - for better or worse. `http://www.sintef.no/en/corporate-news/big-data--for-better-or-worse/`.

[17] Nicolas Bruno and Surajit Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB*, pages 499–510, 2006.

[18] Nicolas Bruno and Surajit Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, pages 826–835, 2007.

[19] Nicolas Bruno and Surajit Chaudhuri. Physical Design Refinement: The Merge-Reduce Approach. *ACM TODS*, 32(4), 2007.

[20] Michael J. Cafarella and Christopher Ré. Manimal: Relational Optimization for Data-Intensive Programs. *WebDB*, 2010.

[21] Pedro Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.

[22] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, pages 146–155, 1997.

[23] Surajit Chaudhuri and Vivek R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, pages 3–14, 2007.

[24] Songting Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(1-2):1459–1468, 2010.

[25] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53(1):72–77, 2010.

[28] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[29] Martin Dietzfelbinger. Universal Hashing and k-Wise Independent Random Variables via Integer Arithmetic Without Primes. In *STACS*, pages 569–580, 1996.

[30] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms*, 25(1):19 – 51, 1997.

[31] Martin Dietzfelbinger and Rasmus Pagh. Succinct Data Structures for Retrieval and Approximate Membership. In *ICALP 2008*, pages 385–396, 2008.

[32] Martin Dietzfelbinger and Ulf Schellbach. On Risks of Using Cuckoo Hashing with Simple Universal Hash Classes. In *SODA*, pages 795–804, 2009.

[33] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1–2):47–68, 2007.

[34] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient Parallel Data Processing in MapReduce Workflows. *PVLDB*, 5, 2012.

[35] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):518–529, 2010.

[36] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.

[37] Jens Dittrich, Stefan Richter, and Stefan Schuh. Efficient OR Hadoop: Why Not Both? *Datenbank-Spektrum*, 13(1):17–22, 2013.

[38] Jens-Peter Dittrich, Peter M. Fischer, and Donald Kossmann. AGILE: Adaptive Indexing for Context-Aware Information Filters. In *SIGMOD*, pages 215–226, 2005.

[39] Michael Drmota and Reinhard Kutzelnigg. A Precise Analysis of Cuckoo Hashing. *ACM Trans. Algorithms*, 8(2):11:1–11:36, April 2012.

[40] Ahmed Eldawy and Mohamed F. Mokbel. SpatialHadoop: A MapReduce Framework for Spatial Data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1352–1363, 2015.

[41] Mohamed Y. Eltabakh et al. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

[42] Mohamed Y Eltabakh, Fatma Özcan, Yannis Sismanis, Peter J Haas, Hamid Pirahesh, and Jan Vondrak. Eagle-Eyed Elephant: Split-Oriented Indexing in Hadoop. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 89–100. ACM, 2013.

[43] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA Database - Data Management for Modern Business Applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[44] Sheldon J. Finkelstein et al. Physical Database Design for Relational Databases. *ACM TODS*, 13(1):91–128, 1988.

[45] Avrilia Floratou et al. Column-Oriented Storage Techniques for MapReduce. *PVLDB*, 4(7):419–429, 2011.

[46] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. *Theory of Comp. Sys.*, 38(2):229–248, 2005.

[47] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[48] Google Inc. Google sparse and dense hashes. `https://code.google.com/p/sparsehash/` Version 31/07/14.

[49] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi A. Kuno, and Stefan Manegold. Concurrency Control for Adaptive Indexing. *PVLDB*, 5(7):656–667, 2012.

[50] Goetz Graefe and Harumi A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, pages 371–381, 2010.

[51] http://engineering.twitter.com/2010/04/hadoop-at-twitter.html.

[52] Hadoop Users, http://wiki.apache.org/hadoop/PoweredBy.

[53] Felix Halim et al. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.

[54] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.

[55] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1199–1208, 2011.

[56] Herodotos Herodotou and Shivnath Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11):1111–1122, 2011.

[57] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[58] Stratos Idreos et al. Database Cracking. In *CIDR*, pages 68–78, 2007.

[59] Stratos Idreos et al. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, pages 297–308, 2009.

[60] Stratos Idreos et al. Here are my Data Files. Here are my Queries. Where are my Results? *CIDR*, pages 57–68, 2011.

[61] Stratos Idreos et al. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.

[62] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a Cracked Database. In *SIGMOD Conference*, pages 413–424, 2007.

[63] Eaman Jahani et al. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.

[64] Dawei Jiang et al. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.

[65] Alekh Jindal, Jorge Quiané-Ruiz, and Samuel Madden. CARTILAGE: Adding Flexibility to the Hadoop Skeleton. *SIGMOD Demo*, 2013.

[66] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. *SOCC*, 2011.

[67] Alekh Jindal, Felix Martin Schuhknecht, Jens Dittrich, Karen Khachatryan, and Alexander Bunte. How Achaeans Would Construct Columns in Troy. *CIDR*, 2013.

[68] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[69] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Koonchun Lai, Thomas Legler, Benoit Schlegel, and Wolfgang Lehner. Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 476–487. IEEE, 2014.

[70] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206, 2011.

[71] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *ACM SIGMOD*, pages 339–350, 2010.

[72] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. KISS-Tree: Smart Latch-Free In-Memory Indexing on Modern Architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 16–23. ACM, 2012.

[73] Don Knuth. Notes On "Open" Addressing. *Unpublished Memorandum*, 1963.

[74] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley, 1998.

[75] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-aware Hash Joins. In *Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013.*, pages 1–12, 2013.

[76] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *29th IEEE ICDE*, pages 38–49, April 2013.

[77] Viktor Leis. ART implementations. `http://www-db.in.tum.de/~leis/` Version 31/07/14.

[78] Haojun Liao, Jizhong Han, and Jinyun Fang. Multi-dimensional Index on Hadoop Distributed File System. In *Fifth International Conference on Networking, Architecture, and Storage, NAS 2010, Macau, China, July 15-17, 2010*, pages 240–249, 2010.

[79] Jimmy Lin et al. Full-Text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics. *MapReduce Workshop*, 2011.

[80] Jimmy Lin and Alek Kolcz. Large-Scale Machine Learning at Twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 793–804. ACM, 2012.

[81] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 961–972. ACM, 2011.

[82] Dionysios Logothetis et al. In-Situ MapReduce for Log Processing. *USENIX*, 2011.

[83] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems. *PVLDB*, 7(14):1797–1808, 2014.

[84] Martin Lühring et al. Autonomous Management of Soft Indexes. In *ICDE Workshop on Self-Managing Database Systems*, pages 450–458, 2007.

[85] Lukas M Maas, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. BUZZARD: A NUMA-Aware In-Memory Indexing System. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1285–1286. ACM, 2013.

[86] Michael Mitzenmacher. *Some Open Questions Related to Cuckoo Hashing*, volume 5757, pages 1–10. LNCS, 2009.

[87] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant Loading for Main Memory Databases. *Proceedings of the VLDB Endowment*, 6(14):1702–1713, 2013.

[88] Chris Olston. Keynote: Programming and Debugging Large-Scale Data Processing Workflows. *SOCC*, 2011.

[89] Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman. *Mahout in Action*. Manning Shelter Island, 2011.

[90] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[91] Andrew Pavlo et al. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD*, pages 165–178, 2009.

[92] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[93] Mihai Pătraşcu and Mikkel Thorup. The Power of Simple Tabulation Hashing. *J. ACM*, 59(3):14:1–14:50, June 2012.

[94] Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich. RAFTing MapReduce: Fast recovery on the RAFT. *ICDE*, pages 589–600, 2011.

[95] Jun Rao and Kenneth A. Ross. Making B+-Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.

[96] Stefan Richter. HAIL: Hadoop Aggressive Indexing Library. Master's thesis, Saarland University, Germany, 2012.

[97] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *PVLDB*, 9(3):96–107, 2015.

[98] Stefan Richter, Jens Dittrich, Stefan Schuh, and Tobias Frey. Mosquito: Another one bites the Data Upload STream. *PVLDB*, 6(12):1274–1277, 2013.

[99] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Adaptive Indexing in Hadoop. *CoRR*, abs/1212.3480, 2012.

[100] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Static and Adaptive Indexing in Hadoop. *VLDB Journal*, 23(3):469–494, 2013.

[101] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1):460–471, 2010.

[102] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. k-Ary Search on Modern Processors. In *DaMoN Workshop*, pages 52–60. ACM, 2009.

[103] Karl Schnaitter et al. COLT: Continuous On-line Tuning. In *SIGMOD*, pages 793–795, 2006.

[104] Stefan Schuh and Jens Dittrich. AIR: Adaptive Index Replacement in Hadoop. In *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*, pages 22–29, 2015.

[105] Michael Stonebraker, Daniel Abadi, David J DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Communications of the ACM*, 53(1):64–71, 2010.

[106] Mikkel Thorup. String Hashing for Linear Probing. In *20th ACM-SIAM SODA*, pages 655–664, 2009.

[107] Ashish Thusoo et al. Data Warehousing and Analytics Infrastructure at Facebook. *SIGMOD*, pages 1013–1020, 2010.

[108] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[109] Alfredo Viola. *Analysis of Hashing Algorithms and a New Mathematical Transform*. University of Waterloo, 1995.

[110] Mark N. Wegman and J.Lawrence Carter. New Hash Functions and Their Use in Authentication and Set Equality. *J. of Comp. and Sys. Sciences*, 22(3):265–279, 1981.

[111] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2011.

[112] Hung-Chih Yang and D. Stott Parker. Traverse: Simplified Indexing on Large Map-Reduce-Merge Clusters. In *DASFAA*, pages 308–322, 2009.

[113] Matei Zaharia et al. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. *EuroSys*, pages 265–278, 2010.

[114] Albert L. Zobrist. A New Hashing Method with Applications for Game Playing. *Computer Sciences Department, University of Wisconsin*, Technical Report #88, 1970.