Understanding Fundamental Database Operations on Modern Hardware

Stefan Schuh

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. Markus Bläser
Day of Colloquium	18.12.2015

Examination Board:

Chairman	Prof. Dr. Jörg Hoffmann
Adviser and First Reviewer	Prof. Dr. Jens Dittrich
Second Reviewer	Prof. Dr. Sebastian Michel
Academic Assistant	Dr. Johannes Hoffart

To my family

Acknowledgments

First and foremost, I want to thank my supervisor Prof. Dr. Jens Dittrich for his guidance and support. This thesis would not exist without him. I also want to thank Prof. Dr. Sebastian Michel for agreeing to review this thesis. I am also very grateful to all my wonderful colleagues that also became great friends over the last years, especially Stefan Richter, Endre Palatinus, and Felix Martin Schuhknecht.

The research work presented in this thesis was partially supported by the funding from the German Ministry of Education and Science (BMBF). Additionally, the IBM Germany Research & Development laboratory Böblingen supported this thesis with a joint project. Here I want to thank Christian Jacobi, Matthias Pflanz, and Kai Weber for their valuable input over the course of the project.

Abstract

We live in an interconnected digital society, where many companies like e.g. Google, Facebook, and Twitter gather and manage tremendous amounts of data every day. The ongoing rise of mobile computing and the availability of more and more sensor data, e.g. from smart meters, increases the amount of data that is produced every day. Several different architectures have evolved to cope with these vast amount of data over the years. Traditionally, mainframes were used to handle huge amounts of data. However, the mainframe has to renew itself to allow for modern data analytics to be efficient and affordable. Advances in the main memory capacity led to the development of in-memory databases architectures, run on many-core non-uniform memory access (NUMA) machines that can handle terabytes of data on a single machine. As another architecture Google developed MapReduce, a distributed framework for data processing on hundreds or even thousands of commodity machines, to handle data that cannot be stored or processed by a single machine, even if it has a capacity in the range of terabytes.

This thesis consists of three independent parts, as we investigate different fundamental database operations on three different hardware environments mentioned before in three independent projects. In the first project we look at recently published relational equi-join algorithms on modern many-core NUMA servers with large main memory capacities and introduce our own variants of those algorithms. Afterwards, in a second project we investigate how to introduce efficient static and adaptive indexing into the open source Hadoop MapReduce framework, which runs on a cluster of commodity machines. In that project we will also introduce and investigate the Adaptive Index Replacement problem, a variant of the online Index Selection problem. Finally, in the third project we investigate how to bring analytical workloads to the IBM System Z mainframe and introduce a new hardware component that allows us to accelerate filtering and aggregation on large in-memory column stores.

Zusammenfassung

Wir leben in einer vernetzten digitalen Gesellschaft, in der viele Unternehmen, wie zum Beispiel Google, Facebook und Twitter, enorme Datenmengen sammeln und verwalten. Das anhaltende Wachstum des mobilen Computing und die Verfügbarkeit von immer mehr Sensordaten, zum Beispiel von intelligenten Stromzählern, erhöhen die täglich generierte Datenmenge. Um mit der immer stärker wachsenden Datenflut fertig zu werden, haben sich verschiedene Architekturen entwickelt. Die älteste Architektur zur Handhabung großer Daten stellt der Mainframe Computer dar. Allerdings muss sich der Mainframe neu erfinden, um die heutige Datenmenge effizient und vor allem auch zu einem bezahlbaren Preis nicht nur abzuspeichern, sondern auch zu analysieren. Stark erhöhte Hauptspeicher-Kapazitäten haben zur Entwicklung einer neuen Architektur von In-Memory Datenbanken geführt, die typischerweise auf modernen Vielkern-Servern ausgeführt werden. Um mehrere Terabytes an Hauptspeicher in einem einzelnen Server anzubieten wird der Hauptspeicher an mehrere CPUs angeschlossen, was zu nicht-uniformen Speicherzugriffszeiten (NUMA) führt, da es einen Unterschied in der Zugriffszeit zwischen lokalem und entfernten Speicher gibt. Als eine weitere Architektur hat Google MapReduce entwickelt, ein verteiltes System zur Datenverarbeitung auf hunderten oder sogar tausenden Servern.

Diese Thesis besteht aus drei unabhängigen Teilen, in denen wir verschiedene fundamentale Datenbank-Operationen in drei unterschiedlichen Hardwareumgebungen untersuchen. In dem ersten Teil untersuchen wir kürzlich veröffentlichte relationale Equi-Joins auf modernen Vielkern-Systemen mit NUMA und großen Hauptspeicherkapazitäten. Zusätzlich stellen wir eigene Join Algorithmen vor. Im zweiten Teil untersuchen wir, wie wir effizientes statisches und adaptives Indizieren in das Open Source Framework Hadoop MapReduce integrieren können. In diesem Teil untersuchen wir auch das Adaptive Index Replacement Problem, eine Variante des Online Index Selection Problems. Im dritten und letzten Teil untersuchen wir Möglichkeiten den System Z Mainframe von IBM für Datenanalyse interessant zu machen und entwickeln eine neue Hardware-Komponente um Filter- und Aggregationsanfragen auf großen In-Memory Column Stores zu beschleunigen.

Contents

Chapter 1

Introduction

1.1 Overview

We live in an interconnected digital society, where many companies like e.g. Google, Facebook, and Twitter gather and manage tremendous amounts of data every day. The ongoing rise of mobile computing and the availability of more and more sensor data, e.g. from smart meters, increases the amount of data that is produced every day. Several different architectures have evolved to cope with these vast amount of data over the years. Traditionally, mainframes were used to handle huge amounts of data. However, the mainframe has to renew itself to allow for modern data analytics to be efficient and affordable. Advances in the main memory capacity led to the development of in-memory databases architectures, run on many-core non-uniform memory access (NUMA) machines that can handle terabytes of data on a single machine. As another architecture Google developed MapReduce, a distributed framework for data processing on hundreds or even thousands of commodity machines, to handle data that cannot be stored or processed by a single machine, even if it has a capacity in the range of terabytes.

This thesis consists of three independent parts, as we investigate different fundamental database operations on three different hardware environments mentioned before in three independent projects. In the first project we look at recently published relational equi-join algorithms on modern many-core NUMA servers with large main memory capacities and introduce our own variants of those algorithms (Chapter 2). Afterwards, in a second project we investigate how to introduce efficient static and adaptive indexing into the open source Hadoop MapReduce framework, which runs on a cluster of commodity machines (Chapter 3). In that project we will also introduce and investigate the Adaptive Index Replacement problem, a variant of the online Index Selection problem (Chapter 4). Finally, in the third project we investigate how to bring analytical workloads to the IBM

System Z mainframe and introduce a new hardware component that allows us to accelerate filtering and aggregation on large in-memory column stores (Chapter 5).

Additional work on the mainframe can be found in the Appendix.

1.2 Contributions and Publications

In this section I give a short overview of the main contributions contained in the different chapters of this thesis and clearly mark my personal contributions in contrast to contributions from my colleagues and co-authors.

1.2.1 Chapter 2 Multi-core NUMA-aware Main Memory Join Processing

Contributions: In this chapter we look at the performance of several recently published relational equi-join algorithms in the context of a modern many-core NUMA server with a large main-memory capacity.

1. **Black box Comparison.** We start with the core join algorithms from [62, 13, 72] which are already improved versions of join algorithms proposed in [16, 60] or have been shown to outperform join algorithms proposed in [10]. We will start with a black box end-to-end comparison of these four principal join algorithms in Section 2.4.

2. **White box Comparison.** We proceed by performing a white box comparison. We enable all optimizations mentioned in prior work and explore their effects on the runtime performance of the joins, see Section 2.5.1. We then proceed by evaluating the effect of different hash-table implementations in Section 2.5.2, for both NOP and PRB.

3. **Optimizing Join Algorithms.** In order to improve the join performance on NUMA architectures, we optimize the different versions of the PRB join algorithm. First, We will show how to make the partitioning phase NUMA-aware in Section 2.6.1 and also see that we can improve over PRB by 20%. Second, we will show in Section 2.6.2 that a NUMA-aware join task scheduling can also improve the performance by roughly 20%. These improvements are not cumulative as the NUMA-aware partitioning already makes use of all NUMA nodes in the join phase and therefore does not profit from a NUMA-aware scheduling.

4. **Comprehensive Comparison of Joins.** Finally, we will be in the position to perform a comprehensive comparison of all join algorithms, see Section 2.7.

First, we define a common workload for all methods. We also do a reality check and use another meaningful baseline. For primary key columns it is common sense to use automatically generated integer IDs. This leads to a dense key domain of integers. For this distribution a simple array rather than a hash table may be a surprisingly good choice. Though the practicability of this approach may be questioned if the join key domain is sparse, it serves at least as a baseline on how good a join algorithm may get anyway. Additionally, for non-dense distributions a compressed array like the recently suggested [72] may be a good choice to avoid using a full-blown hash table.

Second, we proceed by evaluating the page size of the virtual memory management on all phases of the different joins. This parameter has a surprising effect on runtime.

Third, we look at the scalability of the join methods in terms of the input relation sizes. In this context we propose a strategy for choosing the right number of bits for partitioning the input relations in radix-based joins.

Fourth, we explore the behavior of the join algorithms under moderately skewed and highly skewed data (Section 2.8).

Fifth, we examine the scalability of the different joins in terms of the number of threads and discuss possible reasons for the different scalability characteristics (Section 2.8.1).

Sixth, we evaluate the feasibility of using array joins in moderately dense domains (Section 2.8.2).

Seventh, we evaluate microarchitectural performance aspects (Section 2.8.3).

5. **Effect on real queries.** We will evaluate the effect of the choice of the join algorithm in the context of a real TPC-H query in Section 2.9. We will start with a simple single join query, TPC-H Q19. We will analyze the portion of the query time spent in the actual join. We also provide an additional experiment changing the selectivity of Q19's predicates (see Section 2.9.2). Additionally, see Section 2.9.1 and 2.9.3.

6. **Key Lessons Learned.** Finally, we conclude by identifying the key lessons learned from our experimental study, see Section 2.10.

Personal Contributions: The work presented in this chapter is mainly my personal contribution. Xiao Chen helped me with implementing the experiments and performed many micro-benchmarks, not present in this chapter, that guided us in the study.
Publications: This work has been accepted at SIGMOD 2016.

- [86] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. *SIGMOD*, 2016

1.2.2 Chapter 3 HAIL: Hadoop Adaptive Indexing Library

Contributions: In this chapter 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.

1. **Zero-Overhead indexing.** We show how to effectively piggy-back sorting and index creation on the existing HDFS upload pipeline. This way no additional MapReduce job is required to create those indexes and also no additional read of the data is required at all. In fact, the HAIL upload pipeline is so effective when compared to HDFS that the additional overhead for 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++ [28]. 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). Hence, 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 hence 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 with explicit specifications of 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 users' workloads at a constant indexing overhead.

   (b) *eager* adaptive indexing, a technique that allows HAIL to quickly adapt to changes in 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++ [28] (Section 3.9). 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 HAIL has a small overhead over Hadoop when creating indexes adaptively: all the following jobs are faster in HAIL.

Personal Contributions: Please note that this work was a big team effort of the whole group at that time. I participated in the project from the middle of 2011 until the middle of 2013. As a member of the team developing HAIL I was involved in almost all aspects of the system. I indicate my personal contributions in Table 1.1.

Publications:

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

Contribution	Involvement	Details
Zero-Overhead indexing.	minor	I was involved in the discussions and was often co-piloting when peer-programming or debugging with SR and JQ.
Per-Replica indexing.	minor	see above
Job Execution.	minor	see above
HAIL Scheduling	major	I proposed the idea and implemented this together with JQ.
Zero-Overhead Adaptive indexing.	minor	I was involved in the discussions and was also often co-piloting when peer-programming or debugging with SR and JQ.
Lazy Adaptive Indexing Strategy.	no	Was implemented by SR.
Eager Adaptive Indexing Strategy.	major	I implemented this strategy and also came up with the formal definitions shown in Chapter 3.
Selectivity-based Adaptive Indexing Strategy.	minor	I was only involved in the discussion
Exhaustive validation.	major	I conducted almost all experiments, either alone or together with JQ. The only exception are the experiments run on the EC2 instances, which were conducted by JS.

Table 1.1: Personal contributions to Chapter 3. Contributions by Stefan Richter (SR), Jorge-Arnulfo Quiané-Ruiz (JQ), and Jörg Schad (JS) are mentioned in the Details column.

- [29] 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

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

- [80] 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

- *Patent:* Replicated data storage system and methods WO 2013139379 A1.

Initial results of the HAIL project were published in Stefan Richter's Master Thesis [78]. An extended version containing the HAIL Splitting strategy was published in PVLDB [29]. Afterwards, we introduced adaptive indexing in Hadoop and published the results in a technical report [79]. A combined article including the description of the original HAIL system and the adaptive indexing of LIAH was then published in the VLDB Journal [80]. An extended version of that journal article is presented in Chapter 3 of this Thesis. I decided to include the full article, as it is very hard to understand my personal contributions isolated from the rest of the system and that article is also very important for the later chapter, Chapter 4, as the adaptive index replacement is integrated into HAIL.

1.2.3 Chapter 4 AIR: Adaptive Index Replacement in Hadoop

Contributions: In this chapter we investigate which indexes to create and/or which indexes to replace in the scenario of HAIL with a given capacity constraint.

1. **Adaptive Index Replacement problem.** We formulate the Adaptive Index Replacement problem, a variant of the Index Selection problem, tailored to the adaptive indexing scenario. In contrast to previous work, our model does not strictly divide the cost into the workload execution cost and the reconfiguration cost; since in adaptive indexing the cost to reconfigure the physical structures depend on the currently executed query. Furthermore, we present a cost model for the adaptive indexing scenario in Hadoop.

2. **MILP formulation of OPT.** We present a Mixed Integer Linear Program to find the optimal *offline* solution to the introduced AIR problem.

3. **Buffer Replacement Adoption.** We show how Buffer Replacement algorithms can be used to tackle the AIR problem. Additionally, we show that the straightforward use of those algorithms lead to (poor) performance that might even defeat the purpose of indexing.

4. **Adaptive Index Replacement algorithms.** We present our Adaptive Index Replacement algorithm LEB-K that is based on the LRU-K Buffer Replacement algorithm [70]. We also present a variant of our algorithm suitable for stable workloads called LEB-∞.

5. **Extensive experimental study.** We perform an extensive experimental study, evaluating our algorithms against LRU-K as well as two online Index Selection algorithms, namely SoftIndex [66] and BC [19]. We examine the

overall performance as well as the robustness of all algorithms with respect to evolving workloads and abrupt shifts of focus. We validate our findings using our cluster.

6. **Flaws in related work.** In the evaluation of the Adaptive Index Replacement algorithms we uncover a small flaw in the BC algorithm. If the space requirement would allow for creating many indexes, the BC algorithm leads to poor performance in comparison to other algorithms in the append only scenario of Hadoop.

Personal Contributions: All the above mentioned contributions are my own personal contributions.
Publications:

- [87] Stefan Schuh and Jens Dittrich. AIR: Adaptive Index Replacement in Hadoop. *ICDE Workshops*, pages 22–29, 2015

This chapter is an extended version of a workshop paper [87] published in CLOUDDM 2015, an ICDE workshop.

1.2.4 Chapter 5 Smart Caches (not only) for Analytical Workloads

Contributions: In this chapter we look at the characteristics of the IBM System Z mainframe and investigate possible hardware optimizations that would enable faster analytical computation on data stored in the mainframe.

1. **Computation on Cache Lines.** We evaluate the possible performance gains by introducing computational power in the L1 cache lines (see Section 5.3). As the cost to introduce computational power into the L1 cache is most likely too high and as we also notice that the access of memory not resident in the caches dominate the total cost, we decide to look at the memory bandwidth of System Z.

2. **Bandwidth analysis on System Z.** System Z provides at least four different hardware instructions to move data from one location in the memory to another location in memory. We analyze these move methods (see Section 5.4) and discover that the MVCL, *move character long*, instruction has the highest throughput. This is especially interesting as this instruction does not use the CPU registers but only uses the last level cache to move the data.

3. **Simulation of new hardware.** We simulate the effect of a new hardware component, that allows us to perform filter and aggregation on pages that are moved through the last level cache (see Section 5.6).

4. **Instruction Design.** We introduce several instructions in Section 5.7, that make use of the new hardware component and show how these instructions can be used to accelerate typical analytical queries.

5. **Sample implementation.** We give a sketch of a possible implementation of the new hardware in Section 5.8.

Personal Contributions: All the above mentioned contributions are my own personal contributions. IBM provided me with access to a System Z mainframe and fruitful discussions.
Publications:

- *Patent Application:* Accelerator for analytics workload. USPTO No: 14/543319

Chapter 2

Multi-core NUMA-aware Main Memory Join Processing

2.1 Introduction

Database research is full of traditions. One of our most prominent traditions is to publish new join algorithms every year. And, yes, we mean equi-joins, on relational data; not some fancy approximate similarity join on probabilistic JSON data streams. Isn't this kind of magic that after 40 years of database research, there is still progress in an area that is at the core of almost every query plan? Shouldn't relational equi-join algorithms be a solved problem anyway? So, why should we care?

When taking a deep look at the abundant recent related work on relational equi-joins from 2011 to 2015 [16, 14, 62, 10, 72, 12, 13], it quickly turns out that based on that literature it is surprisingly hard to decide which is the best join algorithm. For instance, it seems clear from [12] that a hash-based approach outperforms sort based approaches, at least on current hardware with the limited SIMD register width available today. However, for the different hash table-based join implementations it is unclear if hardware-conscious partition-based algorithm (the PRB-family of join algorithms) outperforms non-partitioning hardware-oblivious algorithms (the NOP-family of join algorithms) anyway. For instance, in 2011, the experiments in [16] show that NOP outperforms PRB. However, in 2013, the experiments in [12] show exactly the opposite: PRB outperforms NOP. In another work in 2013, [62] again, NOP outperforms PRB. So which algorithm is better? In odd years it is NOP? In even years it is PRB? The answer is: we do not know. The more interesting question is: why do those experiments report contradicting results? There are several reasons:

1. **different implementations** were used, e.g. in [16] a very basic NOP im-

plementation was used where the concurrent chaining hash table was implemented using linked lists and two separate arrays were used for locks and head pointers. In contrast, [13] provided a more cache-efficient chaining hash table implementation which used a single array for both locks and tuples and removed head pointers. NOP from [62] implemented a lock free hash table by using linear probing and Compare-and-Swap instructions. Another example is PRB where [13] reimplemented the PRB algorithm from [16] since the implementation from [16] had too many function calls and pointer dereferencing in critical code paths.

2. **different optimizations** were applied to the different join algorithms, e.g. in [12, 14] software-write combine buffers were used for PRB, in [62, 16] they were not used, for [13] it is unclear wether software-write combine buffers were used for the results presented in the paper. Moreover, [62] used a linear probing hash-table as its hash-table implementation whereas [13, 12, 14] used chained hashing. In addition, [16, 12] did not consider NUMA-aware join processing at all, while in [62, 12] at least the join relations as well as working memory is distributed over all NUMA nodes.

3. **different performance metrics** were used, e.g. different definitions of "join throughput", e.g. in [12] it is defined as the number of join results produced per second, i.e. $Throughput = \frac{|R \bowtie S|}{|\text{total runtime}|}$. Notice, that this definition focusses on the amount of tuples output by the join algorithm. Hence, it is sensitive to the join selectivity. In contrast, in [62] throughput is defined as the ratio of the sum of the relation sizes and the total runtime, i.e. $Throughput = \frac{|R| + |S|}{|\text{total runtime}|}$. This definition focusses on the input of the join algorithms. Hence, this definition is independent of the join selectivity. We will use the latter definition in our study.

4. **different machines**, e.g. the machine used in [16] is a single socket six-core Intel Xeon X5650 Nehalem. In contrast, [13] reports the results for several architectures with the best performance achieved on an eight-core Intel Xeon E5-2680 server. In [62] the authors used four eight-core Intel Xeon X7560 Nehalem CPUs in their experiments. Similarly, [13] used four eight-core Intel CPUs however with the Sandy Bridge architecture.

5. **difference of micro-benchmarks and real queries**, e.g. [16, 14, 13, 62, 10, 72, 12, 60] did not consider total query execution times of real world (or at least TPC-H) queries, that include several attributes that have to be considered for predicate evaluation after the join processing. However, simple techniques like selection push-down may considerably reduce the input sizes to a join algorithm (even though the unfiltered relations are huge). In such

a situation the choice of the join algorithm may become less crucial. In addition, tuple reconstruction has been shown to have a substantial effect on the overall runtime of a query [3]. Whether a join is run with or without tuple reconstruction makes a huge difference in practice. However, none of these effects were evaluated in those works.

Any of these differences alone may have a substantial effect on the runtime of a join algorithm or its interpretation. Accumulating multiple of those differences makes a comparison very hard. In particular, comparing results from different papers becomes close to infeasible.

Therefore, we believe the time is ripe for a clean slate experimental comparison of relational equi-joins. This chapter fills this gap. We will focus on hash-based join algorithms as almost all recent works suggest that these algorithms are the most promising ones. Still, we will use a modern sort-based approach as one baseline [12][1]. We do not further explore sort-based joins as the evidence from recent work suggests that sort-based joins cannot match the performance of other joins. We will evaluate all algorithms in the context of a modern NUMA (non-uniform memory access) architecture.

The algorithms we evaluate in our study are based on algorithms published in four recent papers [72, 12, 62, 13]. Notice that those papers in turn improve upon several other older papers [60, 16, 10]. We will introduce several variants of those algorithms. In total, in our study, we evaluate **thirteen different join algorithms**.

2.2 Related work

We focus on papers that discuss in-memory joins on multicore systems; we are aware that there is also a lot of related work on sorting, hashing or partitioning in memory, which is of course highly related to join processing. That work will be cited in other sections whenever appropriate. In the following we will discuss related work in chronological order.

Kim et. al. [60] revisited the sort vs hash argument in the context of main memory multi-core system by comparing a hash join and a sort-merge join that are optimized for modern multi-core systems. The hash join algorithm they introduced is called parallel radix hash join. Their experimental results showed that the parallel radix hash join outperforms the sort-merge join by a factor of two. They also developed an analytical model for the join performance and predicted that the sort-merge joins will become faster with the following two future hardware trends. First, *Wider SIMD instructions*: the sort-merge join algorithm scaled

[1] We also wanted to use a second sort-based baseline [62]. However, that code was not available.

near-linearly with the width of SIMD instructions in their models, while the hash join algorithm hardly exploits the capability of SIMD instructions. Second, *Limited per-core bandwidth*: the hash join algorithm needs at least two pass partitioning for large data sets due to the limited number of TLB cache entries, which result in at least three trips to main memory, while the sort-merge join only needs two. With limited per-core bandwidth more memory trips would lead to worse performance.

Blanas et. al. [16] extended the categories of main memory multi-core join algorithms by introducing no-partitioning hash join. Unlike partition-based algorithm like parallel radix hash join [60], the no-partitioning hash join is a straightforward parallellised version of canonical (simple) hash join without partitioning the data at all. Blanas implemented the no-partitioning hash join with a lock-based concurrent chaining hash table and compared it with partition-based algorithms using three different partitioning algorithms: shared partitioning, independent partitioning, and the radix partitioning from Kim et. al. [60]. Their experimental results show that the no-partitioning hash join outperforms all partition-based hash joins for almost all data distributions and is only slightly slower than parallel radix hash join with uniform datasets.

A later work by Albutiu et. al. [10] on sort-merge join further extended the design space of main memory join algorithms by giving focus on optimizing for NUMA systems. The Massively Parallel Sort-Merge join (MPSM) proposed in their paper uses a carefully tuned memory access pattern and avoids inter-thread synchronization as much as possible. Their experimental results show that MPSM runs much faster than no-partitioning hash join [16] and radix hash join [67] (the latter implemented in Vectorwise on a 32-core, 4-socket machine). Unfortunately, the authors did not make their code available to us. Hence, we will use the sort-based algorithm from Balkesen et. al. [12] as the sort-based baseline as it is freely available and additionally was also shown to be superior to MPSM.

Balkesen et. al. [13] investigated the parallel radix hash join and no-partitioning hash join algorithms from Blanas et. al. [16] and proposed better variants for both algorithms. They achieved higher throughputs by improving the cache efficiency of the hash table implementations for both algorithms and adopting a better skew handling mechanism for parallel radix hash join. We discuss this algorithm (called PRB in this chapter) in more detail in Section 2.3. In their experimental results, using their new implementations, the parallel radix hash join outperforms the no-partitioning hash join for almost all architectures and workloads — except on Niagara2, which has 8 threads per core, for a workload using a very large probe relation. This result is contradicting the conclusions made by Blanas et. al. [16].

The picture changed again when Lang et. al. [62] published their NUMA-aware no-partitioning hash join. In their results, their no-partitioning hash join method outperforms the parallel radix hash join from Balkesen et. al. [13] by a factor of

more than two on a 4-socket machine with 64 hardware contexts. We discuss the no-partition algorithm (called NOP in this chapter) from Lang et. al. in Section 2.3.

Another work by Balkesen et. al. [12] improved the sort-merge join from Kim et. al. [60] and their own parallel radix join from [13]. Their sort-merge join uses wider SIMD instructions and uses range partitioning to allow for efficient multithreading without heavy synchronization. We discuss the proposed sort-merge join (called MWAY in this chapter) in more detail in Section 2.3. In their experimental results, in contrast to the prediction from Kim et. al. [60], wider SIMD instructions did not yet make sort-merge join superior to parallel radix hash join. In fact, in their experiments, parallel radix hash join still always outperforms sort-merge join.

While all the previous research focused on the runtime of join algorithms, the work from Barber [72] studied the memory-efficiency of hash join methods. They proposed a highly memory efficient linear probing hash table called concise hash table (CHT). We also discuss this algorithm (called CHTJ in this chapter) in more detail in Section 2.3. They compared their method with the no-partitioning hash join and the parallel radix hash join from [13]. Their experimental results show that they can reduce the memory usage by one to three orders of magnitude while maintaining competitive performance.

Finally, there has been work optimizing joins for specialized architectures including GPUs, e.g. [57], hybrid CPU-GPU architectures, e.g. [44], and coprocessors attached through PCI express cards like Intel's Xeon Phi, e.g. [53, 76]. Though these are all interesting works they are way beyond the scope of this chapter. We will focus on modern server CPUs which are still abundant in many places.

2.3 Fundamental Representatives of Main-Memory Join Algorithms

In summary, we can identify three fundamental classes of join algorithms into which the most recently published join algorithms fall. Namely: (1) partition-based hash joins, (2) no-partitioning hash joins, and (3) sort-merge joins.

In the following we will discuss one or two modern variants of each class in more detail. This discussion will serve as the starting point of our study.

Table 2.1 shows this classification assigning the papers discussed in Section 2.2 to their corresponding class. Notice that some of those papers, e.g. [16], presented algorithms from multiple classes.

Join Class	Modern Variants Introduced in Paper
Partition-based Hash Joins	[60],[16],[13],[12]
No-partitioning Hash Joins	[16],[13],[62],[72]
Sort-merge Joins	[60],[10],[12]

Table 2.1: Join algorithms from Section 2.2 and their assignment to classes

2.3.1 Partition-based Hash Joins

Core Idea: *Partition-based Hash Joins partition the input relations into small pairs of partitions (co-partitions) where one of the partitions typically fits into one of the caches. The overall goal of this method is to minimize the number of cache misses when building and probing hash tables.*

PRB is the two-pass parallel radix hash join described in [13]. A problem with partitioning joins is that different partitions will most likely reside on different memory pages. Thus, randomly writing tuples to a large number of partitions may cause excessive TLB misses. In order to fix this problem, **PRB** uses two-pass partitioning to guarantee that the number of partitions does not exceed the number of TLB entries. The first partitioning pass starts by assigning equal-sized regions (chunks) to each thread. The algorithm precomputes the output memory ranges of each target partition by building histograms. Hence, each thread knows where and how much to write for each partition without the need for further synchronization. After histograms have been built, each thread scans the input relation and writes each tuple to its destination region. The first partitioning pass already produces a considerable number of partitions. Therefore, in order to perform the second partitioning pass, entire sub-partitions (rather than chunks of a partition as done in the first partitioning pass) are assigned to worker threads by using a task queue. If required, skew handling may be done to break up larger partitions further by assigning multiple threads to an individual partition. In the join phase, each thread takes one co-partition at a time and runs a textbook hash join algorithm on it using a chained hash table.

2.3.2 No-partitioning Hash Joins

Core Idea: *No-partitioning hash joins concurrently build a single global hash table. Simultaneous multi-threading and out-of-order execution is used to hide cache miss penalties automatically. In contrast to partition-based joins, no knowledge about the hardware cache sizes or number of TLB entries is required for tuning.*

Join	Description	Paper	Code
Fundamental Classes of Join Algorithms (Section 2.3) &Black box comparison (Section 2.4)			
PRB	Basic two-pass parallel radix join without software managed buffer and non-temporal streaming	[13]	Original
NOP	No-partitioning hash join	[62]	Original
CHTJ	Concise hash table join	[72]	Own
MWAY	Multi-way sort merge join	[12]	Original
White box comparison (Section 2.5)			
NOPA	Same as NOP except using an array as the hash table	This	Modified
PRO	One-pass parallel radix join with software managed buffer and non-temporal streaming	[13]	Original
PRL	Same as PRO except using linear probing hashing instead of bucket chaining	This	Modified
PRA	Same as PRO except using arrays as hash tables	This	Modified
Optimizing Parallel Radix Join (Section 2.6)			
CPRL	Chunked parallel radix join with software managed buffer and non-temporal streaming	This	Own
CPRA	Same as CPRL except using arrays as hash tables	This	Own
PROiS	PRO with improved scheduling	This	Modified
PRLiS	Same as PROiS except using linear probing hashing instead of bucket chaining	This	Modified
PRAiS	PRA with improved scheduling	This	Modified

Table 2.2: reference table for the algorithms evaluated in this chapter

NOP is the no-partitioning hash join described in [62]. It uses a lock-free synchronization mechanism for a linear probing hash table using compare-and-swap. The algorithm starts by assigning equal-sized regions (chunks) to each thread. Each thread then inserts its chunk of the build relation into the global hash table. After all threads are done inserting, each thread starts probing its chunk of the probe relation against the global hash table.

For inserts into the global hash table, each thread uses an atomic Compare-

and-Swap (CAS) operation. This is a transactional and conditional operation that is only executed if the slot contains the empty key; in that case the empty key is overwritten by the key to be inserted. Otherwise the operation returns false. As entries are never removed or overwritten in a slot, the thread can copy the payload to the bucket in an additional non-transactional operation. In addition to the lock-free hash table, another optimization of NOP is to interleave hash table allocation among all available NUMA nodes for better memory bandwidth utilization.

CHTJ is the concise hash table join described in [72]. At its core a Concise Hash Table (CHT) consists of four major components. First, an array A of size n storing all n inserted tuples without any additional empty slots. Second, a hash function $hash : Key \mapsto [8 \cdot n]$[2], where Key denotes the domain of the join keys and $[8 \cdot n]$ denotes the set of all integers in the range from 1 to $8 \cdot n$. Third, a bitmap B of size $8 \cdot n$. This bitmap marks if a certain hash bucket is occupied. Fourth, a population count array PC of size $n/4$ with a running sum of the population count of the bitmap, i.e. for for every 32 bits of the bitmap we count the number of elements stored up to that point. A CHT is a static structure that is bulkloaded once and then used for lookups only. Hence, this structure is very suitable for join processing. Notice that Google sparse hash map [36] is very similar to CHT, but additionally allows for inserts and deletes.

In a CHT, a lookup for a key works as follows: we first check if the bit $B(hash(key))$ is set. If that is the case, this implies that array slot $hash(key)$ is occupied. In other words: there *may* be a result. In that case, from the population count array PC we retrieve the population count from position $\lfloor hash(key)/32 \rfloor - 1$ and add it to the number of bits set in B within the range

$$\big[\lfloor hash(key)/32 \rfloor \cdot 32;\ hash(key) - 1\big].$$

In order to speed up this process B and PC are physically interleaved in the same structure.

CHTJ works as follows: first it radix-partitions the build input into a small number of partitions very similar to PRB. Second, one global CHT is allocated where each thread bulkloads its partition to a disjoint region in that CHT. Hence, there is no need for additional synchronization at this point. Fourth, the probe relation is handled similarly to NOP: each thread probes one chunk of the probe relation against the global CHT. Again, as no inserts are performed in the CHT at this point, there is no need for synchronization.

We classify CHTJ as a no-partition hash join even though the algorithm uses partitioning on the build input. However, that partitioning is only used to build the global CHT in parallel. Afterwards the algorithms is equal to NOP as discussed

[2]For simplicity we assume here that n is a power of 2.

above. Moreover, in CHTJ the partitioning is not used to run independent joins on co-groups like in PRB.

2.3.3 Sort-merge Joins

Core Idea: *Sort-merge joins belong to the oldest join methods used in databases. The idea is to first sort both input relations on their join keys, if they are not yet sorted, and to use an efficient merge step afterwards to find all matching tuples. Sort-merge join algorithms can exploit and create so-called interesting orders. Even if the performance of a single join in a complex multi-join query would be suboptimal, the overall performance of the sort-merge join plan could be superior.*
MWAY is the m-way sort merge join described in [12]. It is also very similar to the method described in [60]. **MWAY** partitions the data very similar to **PRB**, however using only a single partitioning phase and creating only few partitions. In addition, software write-combine buffers (see Section 2.5.1) are used. After partitioning, each partition will be merge-sorted independently by a separate thread. The merge-sort is implemented with bitonic sorting- and merge-networks. Both sort- and merge-networks are vectorized using SIMD instructions. In addition, multi-way merging is used to save memory bandwidth.

2.4 Black Box Comparisons

In this section we want to compare the representatives of the fundamental join algorithms that were also compared in prior work. We want to identify some of the reasons for contradicting results in previous works. Table 2.2 lists all evaluated join algorithms and the abbreviations used. The *paper* column refers to papers where the specific algorithm was published and "this" means that the algorithm is proposed in this chapter. The *code* column shows the implementation we used for each algorithm. There are three values for this column: "Original" means we use the implementation from authors of the paper, "Modified" means we modify the original implementation to get the new variant and "Own" means we implemented the algorithm from scratch.

For our experimental evaluations we use a setup that is similar to the one used throughout all mentioned previous join papers. The tuples of the join relations consist of two four byte attributes, namely the join key and the payload. Furthermore, the keys in the smaller relation are dense and unique, like in a primary key column. Throughout this study we use the same 60 core machine, see Section 2.7.1 for details. Unless stated differently, all algorithms use at most 32 threads even though our machine has 60 cores available. The reason is that the code of the MWAY algorithm only works with a power of two number of threads. In Sec-

tion 2.8.1 we will increase the number of threads beyond 32 threads. We measure the throughput of a join as $(|R| + |S|)/t_{join}$, i.e. the total number of input tuples of both relations divided by the algorithm runtime.

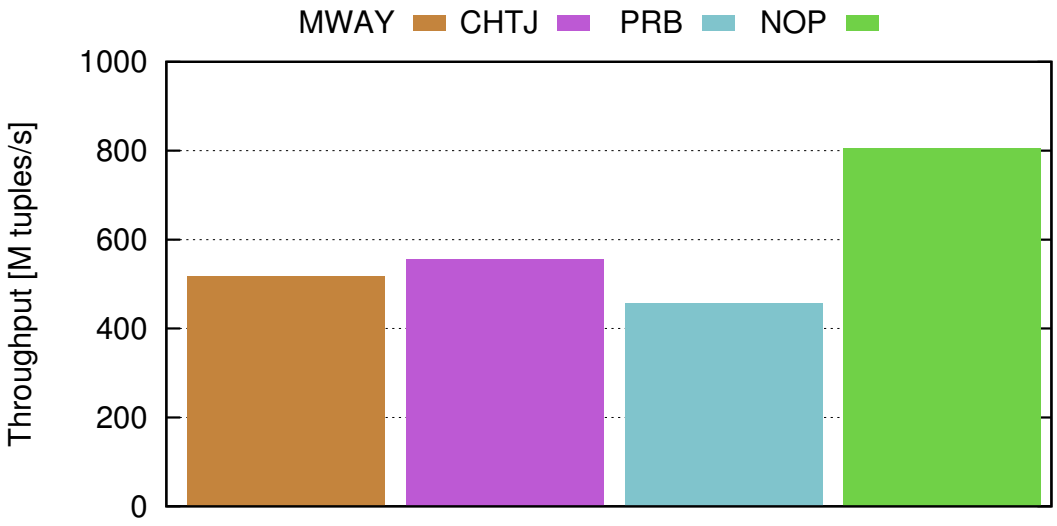


Figure 2.1: Black box comparison of the fundamental join representatives using 32 threads and relation sizes |R| = 128M and |S| = 1280M.

Figure 2.1 shows the performance of this black box comparison in terms of throughput. We use inputs of size 128 million and 1280 million tuples respectively. These results are comparable to the results found in [62] and [72], but do not match the findings in [12] as for instance in that study the performance of PRB was found to be much better than MWAY.

To understand this inconsistency between results from prior work, we will take a closer look at the parallel radix partitioning join in the next section.

2.5 White Box Comparisons

From the prior publications it is not always clear what optimizations were used for the parallel radix join. We therefore take a closer look at different possible optimization for the different methods to make them more comparable.

Let's start by taking a closer look at the code for PRB provided by [2]. We see at least two optimizations that can be enabled.

2.5.1 Optimizing Radix Partitioning

NUMA-Awareness. The first option is the `--basic-numa` flag that equally allocates the partition buffer on all NUMA nodes, as otherwise the buffer will be allocated randomly over different NUMA regions. This option was most likely enabled in all related work as otherwise the performance of the join algorithm decreases considerably on NUMA machines. Turning on this option has another important effect:

Memory Allocation Locality. Before running the actual join, all physical pages will be allocated locally and mapped in the virtual memory table. Hence, we will not trigger page faults and consequent allocations while running a join algorithm. As database system always have a buffer manager anyways, we believe it is a fair assumption that memory buffers were already physically allocated. We already used this option in the previous section and we will also keep this option turned on throughout all following experiments.

Software Write-Combine Buffers. The second option provided by the code of [2] is the `--enable-swwc-part` flag[3]: this flag enables the use of software write-combine buffers (SWWCB) and non-temporal streaming instructions for the parallel partitioning algorithm. SWWCBs, aka software managed buffers, have been known for quite some time [81]. The idea is to allocate a small local buffer for each partition and first put tuples into buffers instead of directly flushing them to the output. This is similar to buffered writes in disk-based partitioning, however, *the size of each buffer is only one cache line*. SWWCB reduce the pressure on the

Algorithm 1: Partitioning with Software Write-Combine Buffers

```
1 for tuple ∈ relation do
2     partition ← hash(tuple.key);
3     pos ← slots[partition] mod TuplePerCacheline;
4     slots[partition]++;
5     buffer[partition].data[pos] = tuple;
6     if pos == TuplePerCacheline - 1 then
7         dest ← slots[partition] - TuplePerCacheline;
8         copy buffer[partition].data to output[dest];
```

TLB as the buffers are very likely to reside in cache and on very few pages while only a full buffer is flushed to main memory. If a buffer can hold N tuples, then the

[3]To be fair with prior work that did not enable this feature in their comparisons. It is marked as experimental and we applied some fixes that removed minor race conditions that did not influence the performance.

number of TLB misses will be reduced by a factor of N. Algorithm 1 illustrates how SWWCB works. Turning on this option has another important effect:

Non-temporal streaming. This is a technique allowing programmers to write half a cache line to DRAM bypassing all caches. It prevents polluting the caches with data that will never be read again. Recently, Schuhknecht et.al [89] performed an in-depth study of the effects of using both software write-combine buffers and non-temporal streaming on the single-threaded radix partitioning algorithm. We follow the guideline provided in that paper.

After enabling all those features we obtain a much more efficient algorithm called **PRO** (Parallel Radix with Optimized partitioning).

Single-pass Partitioning. The original PRO used two-pass radix-partitioning. We ran micro-benchmarks on PRO comparing single-pass and two-pass partitioning and also determined the optimal number of partitions to use for partitioning. Figure 2.2 shows that a single-pass partitioning using 14 bits leads to the highest throughput. We will use this setting in the following for all variants of PRO.

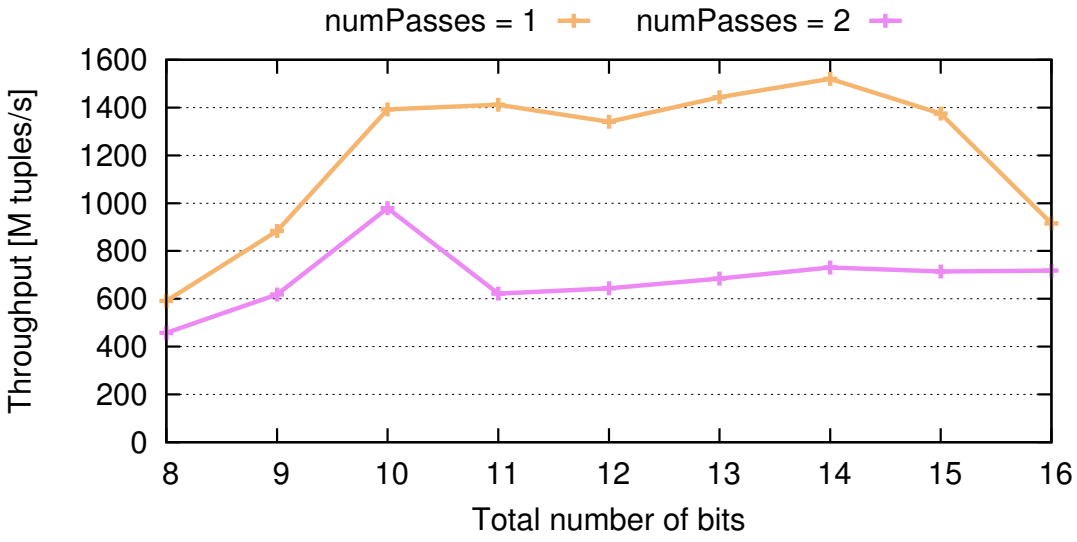


Figure 2.2: Throughput of PRO for different partition sizes and number of radix bits for partitioning (total join including partitioning and join phase); the two-pass algorithm divides the bits evenly over the two passes.

2.5.2 Choice of Hash Method

Linear vs Chained vs CHT. Another dimension that makes the available join algorithms harder to compare is the usage of different hash table implementations

in the algorithms. All presented hash join methods use different hash table implementations. CHTJ of course uses a concise hash table; PRO uses a variant of chained hashing while the NOP algorithm uses linear probing to implement the hash table. We therefore also implemented a version of PRO that uses a linear probing hash table and call that method **PRL**.

Arrays. When using unique and dense domains of join keys we can go one step further and use an even simpler hash table implementation — a simple array. Instead of storing key value pairs in a hash table we can simply use the key as an index in the array and store the value in that position. This assumption on the key domain may sound unrealistic on first sight, however, often joins are performed along 1:n or n:1 foreign key relationships using artificially created IDs (ID Integer PRIMARY KEY AUTOINCREMENT), this situation may occur frequently in practice. It can also be identified easily by the query optimizer through the available statistics. This simple array implementation can also be used in the NOP-family of joins and hence we get two new hash join variants called No Partition Array join (**NOPA**) and the Parallel Radix partition Array join (**PRA**). These joins are of course not as widely applicable as the other hash joins. We will also investigate the usefulness of these methods in the presence of holes in the key domain in Section 2.8.2.

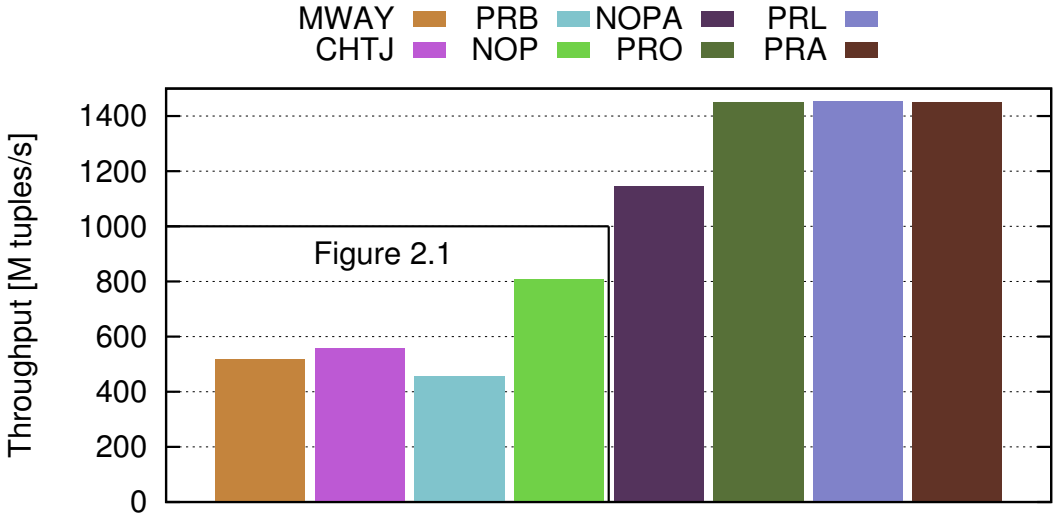


Figure 2.3: Join throughput including improved versions. We observe almost a twofold performance improvement over the blackbox versions shown in Figure 2.1.

Performance Comparison. Considering the optimizations discussed in Sections 2.5.1 and 2.5.2, we take another look at the performance of the join algorithms and show their throughput in Figure 2.3. We can see that PRO clearly

outperforms NOP and now the performance of PRO also resembles the results presented in [12]. However, surprisingly, there is almost no difference in runtime between PRA, PRO, and PRL. Therefore, we might conclude that the choice of the hash method does not have an effect on the runtime; however, later on we will learn that this conclusion would be wrong (see Section 2.6.2).

2.6 Optimizing Parallel Radix Join

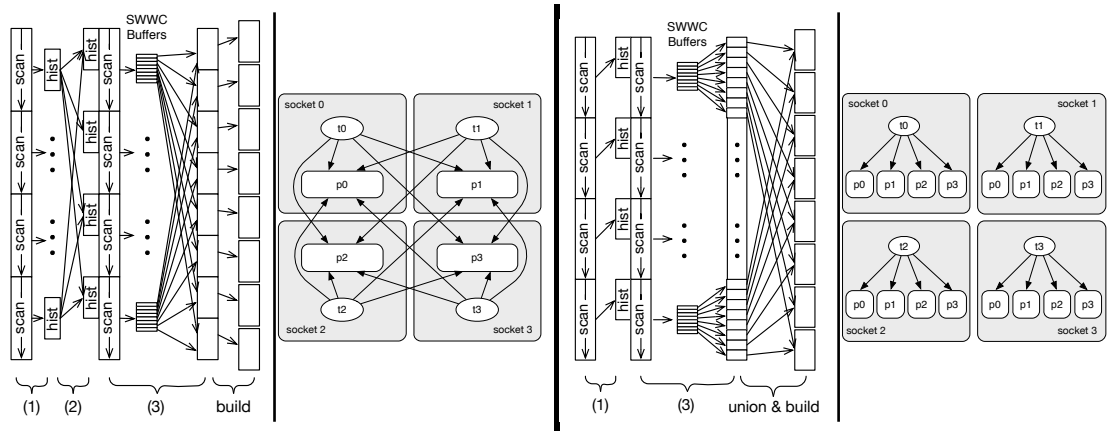
We have observed in Figure 2.3 that the parallel radix partitioning joins PR* are providing the highest throughput so far. We are therefore looking for ways to further improve their performance. In the following, we will look at the partition phase and the join phase separately.

2.6.1 NUMA-aware Partitioning

First we investigate the partitioning phase. The Parallel Radix Partitioning algorithm is illustrated in Figure 2.4(a). It works as follows: (1) every thread sequentially reads a horizontal chunk of the input relation to create a local histogram. (2) a global thread merges the local histograms into a global histogram[4]. The goal is to exploit this global histogram later on as an index to the target partitions. To merge the local histograms, we need a synchronization barrier, as every thread has to complete the local histogram before the final output positions can be computed. (3) each thread reads again its horizontal local chunk of the input relation and partitions the data into its corresponding SWWCB. Each thread keeps as many SWWCB as it has target partitions. Whenever a SWWCB becomes full, it is flushed to the final output position in its target partition using non-temporal streaming. As the final output position of every partition was already determined in phase (2), no further synchronization is necessary. For the probe relation, phases (1)–(3) are executed similarly using the same partitioning function. Once both inputs have been partitioned, each pair of corresponding partitions is joined independently. This is done by building a hash table on the left input and probing the right input against that hash table. Hence, from a high-level perspective this join algorithm is a variant of Grace Hash Join applied to NUMA.

NUMA-partitioning is a task which triggers a considerable number of memory reads and writes, especially when writing out tuples to their destinations. However, in NUMA systems, careless memory access patterns can hurt the performance very badly as remote memory accesses have higher latency and lower bandwidth than local memory accesses. So we devoted our effort on analyzing memory access

[4]Technically, this may also be implemented by letting the threads merge their histogram independently as in phase (3) each thread only requires a subset of the global histogram.



(a) Schematic view on build relation partitioning of PRO

(b) NUMA write pattern for PRO

(c) Schematic view on build relation partitioning of CPRL

(d) NUMA write pattern for CPRL

Figure 2.4: High-level schematic view and NUMA write pattern: PRO vs CPRL

patterns of PRO and we found out that there are in particular random remote memory writes that we can avoid in this algorithm.

Figure 2.4(b) depicts the NUMA access pattern in a simplified case with four sockets, four threads, and four partitions when writing tuples to their target partitions. We see that the partitioning algorithm of PRO introduces many random **remote** writes when writing tuples to their corresponding partitions. Based on this observation, we propose the Chunked Parallel Radix partition (**CPRL**) algorithm that eliminates remote writes when flushing tuples to partitions.

Figure 2.4(c) shows a high-level schematic view on our algorithm CPRL. Notice, that the histogram phase (1) is the same as in PRO. However in contrast to PRO, in CPRL we leave out phase (2), i.e. we do not compute a global histogram. We proceed directly with phase (3), i.e. each thread partitions its data locally within its chunk only based on its local histogram. On a high-level this can be viewed as running a single-threaded histogram-based radix-partitioning inside a chunk. For the probe relation phases (1)–(3) are executed similarly using the same partitioning function. Once both inputs have been partitioned, each pair of corresponding partitions, i.e. each co-partition, is joined independently. In contrast to PRO, at this point we neither have physically contiguous probe nor build partitions available. Hence, we first have to read the different chunks belonging to the build input from its (possibly NUMA-remote) sources. In that process, we directly load that data into a local hash table. Then we also read the different chunks belonging to the probe input from its (possibly NUMA-remote) sources and probe them directly against the hash table. Hence, from a high-level perspective this

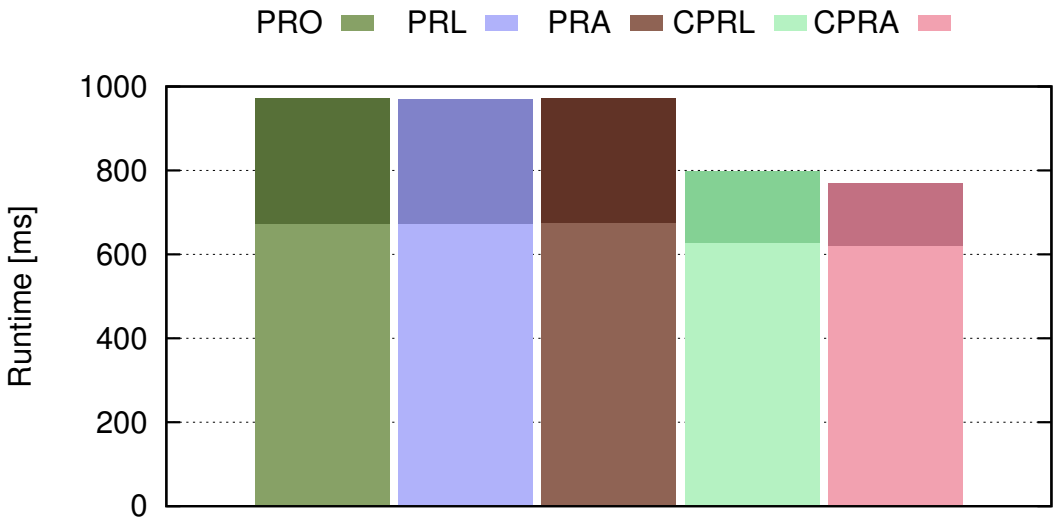


Figure 2.5: Runtime of PR* vs CPR*-algorithms. Relation sizes: |R| = 128M, |S| = 1280M. Lighter colors denote the partition phase and darker colors denote the join phase.

join algorithm is also a variant of Grace Hash Join applied to NUMA. However, in contrast to PRO, we do not require the inputs to each join to be physically contiguous in main memory. Therefore, CPRL trades small random writes to remote memory for large sequential reads from remote memory. Notice that CPRL uses the same linear probing hash table as PRL since it was easier to integrate our own linear hash table implementation. In addition, the linear hash table also provided a slightly better performance compared to the chained hash table implementation by Balkesen et. al. When we use arrays rather than hash tables in the join phase of the Chunked Parallel Radix join, we call it **CPRA**. Again, optimizations like software write-combine buffers and non-temporal store instructions are also used in this algorithm. The write pattern of CPRL is illustrated in Figure 2.4(d).

Let's compare the performance of PRO, PRL, and PRA with our proposed CPRL and CPRA. Figure 2.5 shows the end to end join processing time broken down into partition and join phase. We see that the CPR*-algorithms outperform the PR*-algorithms by ∼20%. We also observe that the partitioning times of the CPR*-algorithms are indeed reduced as expected. However, surprisingly even the join time is reduced. This is counterintuitive to what we expected as we traded remote writes in the partitioning phase with remote reads from all sockets in the join phase. We will investigate the reason for this in the next section.

2.6.2 NUMA-aware Scheduling

All PR*- and CPR*-algorithms build co-partitions which eventually have to be joined independently. How are those individual joins scheduled? What effect does this schedule have on the overall performance of the join algorithms?

After partitioning, in both PR*- and CPR*-algorithms, all co-partitions are put into a LIFO-task queue (which is actually a stack), to be processed by different threads. Recall that the PR*-algorithms partitions an input array into p partitions where for any two partitions $i, j \in [0; p-1], i < j$ it holds that the starting address of partition j is greater than the starting address of partition i. In other words, the partition indices correlate with their virtual addresses. We observed that in all PR*- and CPR*-algorithms, co-partitions are inserted into the queue in ascending sequential indices order, i.e. for p co-partitions the insert order into the queue is $0, \ldots, p-1$. However, recall, that before executing any join, one quarter of each input relation is physically allocated on one of the NUMA-regions. In addition, any additional memory required for partititioning or building hash tables is also equally distributed across NUMA-regions. This memory allocation strategy was already present in the code used by [13]. Assume that the number of threads is considerably smaller than the number of partitions, i.e. $t << p$, typically $p = 16384$ and for our machine $t = 60$. This implies that the first $\lceil 16384/60 \rceil = 274$ partitions reside **on the same** NUMA-region. Hence, all of the first 60 threads removing tasks from the queue will have to read their input data from the same NUMA-region. Moreover, for three quarters of those threads, i.e. 45 threads, this is a remote NUMA-region. Similar bottlenecks can be observed for all other blocks of 274 partitions.

Figure 2.6 shows the bandwidth profile of PRO. We observe that most of the time PRO uses only a single NUMA-region.

We can improve this by carefully reordering the join tasks. We fixed this by changing the task scheduling strategy used by all PR*-algorithms as follows: we insert co-partitions into the task queue in a round-robin manner. Specifically, we first put a partition from the first NUMA region into the queue and then a partition from the second NUMA region and so on. An alternative would be to use a different queue for each NUMA-region. Like that it is very likely that all memory controllers are utilized simultaneously.

Figure 2.6 shows a bandwidth plot of the original PRO, the variant of PRO using improved scheduling, coined **PROiS**, as well as CPRL. In addition, we also introduce variants of PRL and PRA using improved scheduling called **PRLiS** and **PRAiS**. We observe that the improved scheduling of PROiS has a substantial effect on the total bandwidth utilization, i.e. all NUMA nodes are used at the same time. Even though the suboptimal scheduling is also used for CPRL, it does not affect the bandwidth utilization, as every partition has to be read from all

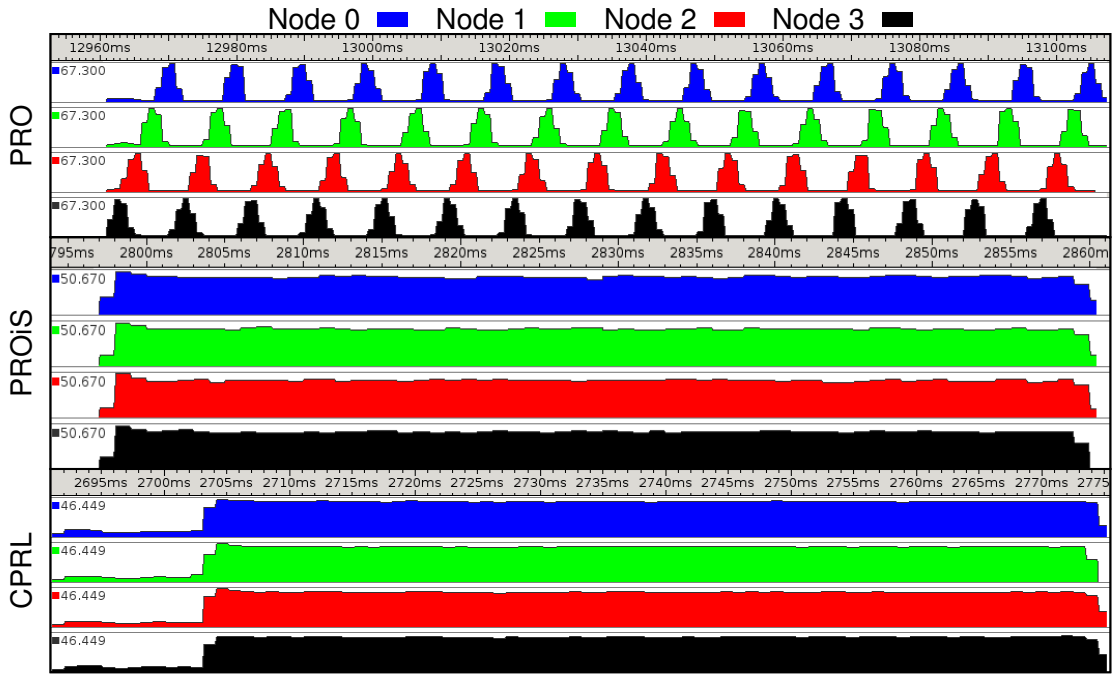


Figure 2.6: Bandwidth profiles for PRO, PROiS, and CPRL obtained with Intel VTunes

NUMA nodes anyhow.

The improved scheduling results in a speedup of the join phase of PRL and PRA by more than a factor of 2, see Figure 2.7. As expected in Section 2.6.1, we can now observe that the join phase of the CPR*-algorithms is in fact slightly more expensive than the one of the PR*iS-algorithms. However, still, in total the CPR*-algorithms are slightly faster than the PR*iS-algorithms. Moreover, in contrast to our results from in Figure 2.3, we can now clearly observe that different hash table implementations have an effect on the runtime of the algorithms.

2.7 Putting it All Together

After our initial black box comparison (Section 2.4), after having analyzed the effects of optimizing radix partitioning and using different hash tables (Section 2.5), and after optimizing the NUMA memory allocation and NUMA access pattern of the various radix algorithms (Section 2.6), we are finally in the position to perform a comprehensive comparison of *all* join algorithms.

In this section we will perform a large-scale experimental study of all thirteen algorithms mentioned above. Recall that Table 2.2 lists all algorithm abbrevi-

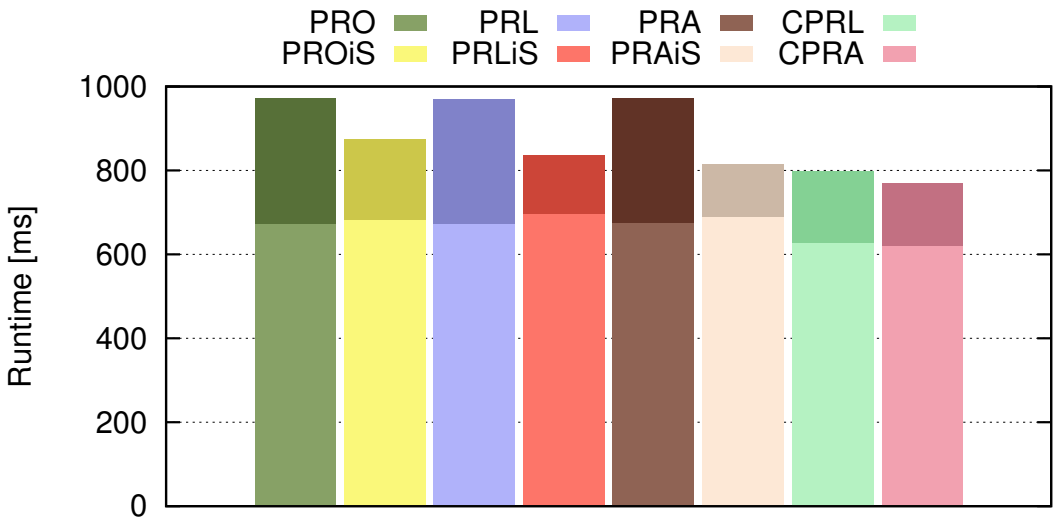


Figure 2.7: Runtime of PR* and CPR*-algorithms vs their variants with improved scheduling (PR*iS-algorithms). Relation sizes: |R| = 128M, |S| = 1280M. Lighter colors denote the partition phase and darker colors denote the join phase.

ations and their short descriptions. In the pdf of this thesis all occurrences of algorithm abbreviations are hyperlinks pointing to their description. We evaluate all algorithms in the same benchmarking framework.

2.7.1 Settings

All our experiments are performed on a server with half a terabyte of main memory and four Intel Xeon E7-4870 v2 CPUs, clocked at 2.30 GHz (published in Q1 2014). This CPU has 30 hardware contexts executed on 15 physical cores that share a 30 MB L3 cache. Each core has one private 32 KB L1 data, one 32 KB instruction cache, and one 256 KB L2 data cache. Notice that the number of TLB entries when using 4 KB page is 256. However, if we use 2 MB pages, we only have 32 TLB-entries! The operating system we used is a 64-bit Debian 7 server with the kernel version of 3.2.0-4. The CPU supports AVX 1.0. Just like the original studies [60, 12, 10, 62, 14], we also assume a column-oriented storage model and adopt the configuration of using a <Key, Payload> pair as a tuple. We use a 4-byte integer key and a 4-byte integer payload to make a fair comparison between all methods, since some available implementations only work for this key size. We assume that the build relation follows a dense primary key distribution and the keys of the probe relation have a foreign key relationship to the keys of the build relation, if not mentioned otherwise. This setting was also used in the mentioned

previous studies.

In the following experiments, we use the implementations of PRO, PRB, NOP from the original authors. PRA, PRL, PRAiS, PROiS, NOPA are implemented based on the authors' implementations. We implemented CPRL, CPRA, and CHTJ ourself from scratch. Since the build relation has dense primary keys, we use the identity hash function modulo the hash table size for all hash joins as it is very effective and efficient in this setting and was also used in the previous studies. Lang et. al. [62] additionally evaluated different hash functions like Murmur, CRC, and multiplicative hashing. We are not evaluating the effect of different hash functions on the join performance in this chapter. All software is implemented in C/C++ and compiled by gcc/g++ version 4.7.2 with optimization level -O3.

2.7.2 Varying Page Sizes

The first dimension we want to explore is the page size of the virtual memory used for the join algorithms. As partitioning is very sensitive to TLB misses, this is a very important aspect. In the previous experiments we used huge page sizes for all allocations, i.e. 2 MB. To study the effect of the page size on the different algorithms we evaluate pages of 4 KB and 2 MB by switching the kernel setting `transparent_hugepage/enabled` between `never` and `always`. We also ensured that all allocations use the default `malloc` or `posix_memalign` methods.

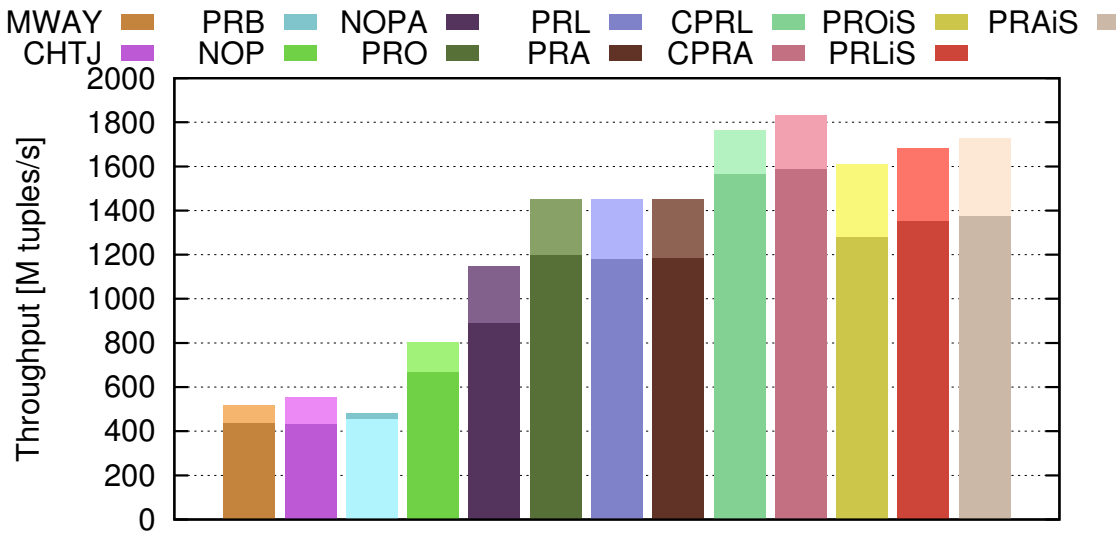


Figure 2.8: Performance of all thirteen join algorithms when using small (4 KB, dark color) and huge pages (2 MB, light color)

Figure 2.8 shows the performance of all thirteen join algorithms when using small (4 KB) and huge pages (2 MB). We observe that all algorithms except PRB improve by using huge pages. PRB is the only algorithm that has a slightly worse performance when using huge pages. This is due to the naive partitioning that does not use any software write-combine buffers for the different partitions. In each of the two radix passes PRB partitions along 7 bits = 128 partitions. The entries for all those pages fit into the TLB-cache when using small pages. However, when using huge pages, we only have 32 TLB-entries available. Hence, many write operations to partitions lead to TLB-misses. This effect is mitigated when using SWWCB as in PRO.

As some of the algorithms are clearly dominated by others, in the following, we do not report for results for PRB, PRO, PRL, and PRA anymore. In addition, for all following experiments we use huge pages.

2.7.3 Scalability in Dataset Size

The next dimension we will explore is the scalability in the size of the input data to the joins. We will explore two workloads: (1) the probe relation is ten times the size of the build relation. The factor ten is motivated by the typical ratio in the TPC-H benchmark and the observation that in a star schema, often used in OLAP applications, the dimension tables are typically much smaller than the fact table. (2) both relations have the same size. This is close to a worst case for hash joins, as the typically more expensive build phase is followed by a rather short probe phase. If the build relation becomes smaller than the probe relation, the optimizer should actually have switched the roles of the relations in the first place. At the same time, this case is close to a best case for sort-based methods, as sorting has super-linear costs and is therefore minimized if both relations have the same size. Previous studies [12, 62] also used similar workloads.

Fine-tuning the partition-based joins. When we started benchmarking the effects of scaling the input datasets, we quickly noticed that the radix-based algorithms are very sensitive to the number of bits used for partitioning. Recall, that in Section 2.5.1 we already explored this effect for *a fixed-size input dataset*. Following the results of our micro-benchmark in Figure 2.2 we assumed that when doubling the data size, we take one additional bit for partitioning and hence end up with the same partition size and obtain good performance. But is that really true? Let's take a second look:

Figure 2.11 shows the average partitioning time per tuple for a varying number of partitions and corresponding data set sizes. On the horizontal axis we use a log scale doubling the number of partitions at every tic. The number of partitions is chosen such that a chained hash table that is built on a single partition fits into L2 cache.

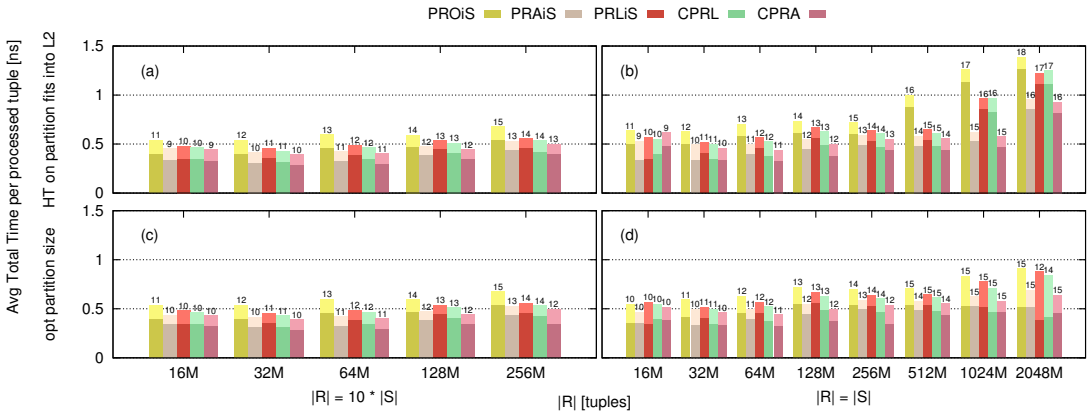


Figure 2.9: Average total time per tuple (partitioning and join) when varying the number of radix-bits used for partitioning. The dark color marks the time for partitioning; the light color marks the time for joining. In (a) and (b) we choose the number of radix bits such that the hash table on a partition fits onto L2. In contrast, in (c) and (d) we depict the number of radix bits leading to the lowest overall runtime. In particular for $|R| = |S|$ (right column) and $|R| \geq 512$ M tuples we see that our assumption, (a) and (b) diverges heavily from the optimal number of bits, (c) and (d). Notice that we can observe in (b) that the partitioning costs increase heavily whereas the join costs stay the same.

We observe that the average partition time per tuple stays almost constant up to including 2^{15} partitions. Starting with 2^{16} partitions the performance deteriorates. This can be explained with the size of the software write-combine buffers. Recall that we use 32 threads. If we create 2^{15} partitions using a single cache line per partition as an SWWCB, all SWWCBs together including other working variables, e.g. histograms, still fit into the shared last level cache (LLC). However, when using 2^{16} partitions, this is no longer the case.

We conclude from this experiment that partitioning data into too many partitions might overshadow the performance gains obtained in the join phase. We therefore micro-benchmarked the performance of all partition-based joins with a varying number of bits used for partitioning and show these results in Figure 2.9. In Figures 2.9(a)&(b) we choose the number of radix bits such that the hash table on a partition fits onto L2. In contrast, in Figures 2.9(c)&(d) we depict the number of radix bits actually leading to the lowest overall runtime. We can see that choosing the number of bits such that the partitions fit into L2 cache is close to the optimal choice as long as the SWWCBs still fit into the shared LLC. However, for larger datasets, we observe in Figure 2.9(b) that the partitioning costs increase sharply. Hence, we conclude that for these input sizes it is better to balance the

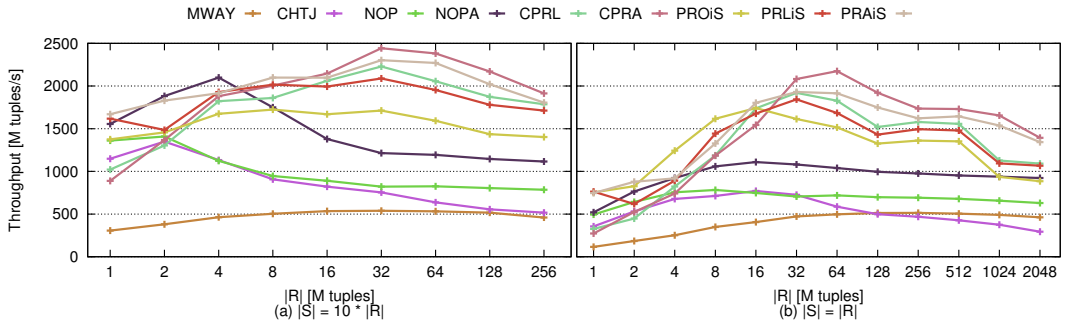


Figure 2.10: Throughput of join algorithms when scaling input dataset sizes

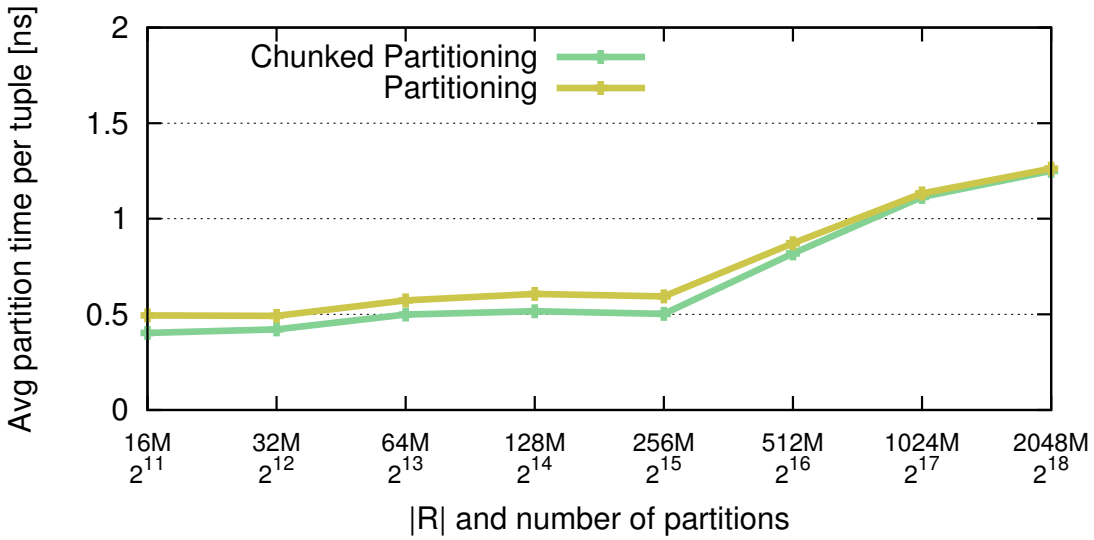


Figure 2.11: Scalability of the partition phase for chunked and non-chunked partitioning

partitioning cost with the join costs. The sweet spot for the number of bits used in partitioning seems to be the minimal number of bits such that the partitions still fit into the shared LLC.

Predicting the optimal number of radix bits. This leads to the following formula for the number of bits for partitioning n_p. Given the size of R as $|R|$, the size of a tuple of R as s_t, the intended load factor of the join hash tables l, the size of a partition buffer as s_b, the size of the L2 cache as $L2$, and the size per thread

of the last level cache as LLC_t [5]:

$$n_p(|R|) = \begin{cases} \log_2\left(\frac{|R|\cdot s_t}{l\cdot L2}\right), & \frac{|R|\cdot s_b\cdot s_t}{L2\cdot l} < LLC_t \\ \log_2\left(\frac{|R|\cdot s_t}{l\cdot LLC_t}\right), & \text{otherwise} \end{cases} \tag{2.1}$$

Figure 2.12 shows the observed runtime for CPRL[6] when varying the number of radix bits from 8 to 18 bits (black points) versus the performance observed when setting the bits according to Equation (2.1) (red line). We can see that the number of bits computed by Equation (2.1) leads in almost all cases to the lowest runtime.

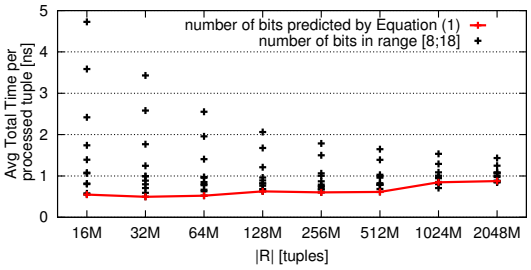


Figure 2.12: Runtime of CPRL when setting the number of partitioning bits according to Equation (2.1)

Back to Figure 2.9, we can also make an additional observation: we see that the different hash table implementations have an effect on the optimal number of bits for partitioning. This is reasonable, as the different hash table implementations differ in their space efficiency. For instance, array joins use a tight array that only keeps the payload, the key however is implicitly represented through the array index. In contrast, a linear probing hash table has to store the key explicitly.

Based on these results, from now on, we will use Equation (2.1) to set the bits of all PR*- and CPR*-algorithms and are in the position to evaluate the join performance when scaling the sizes of the input datasets.

Scalability results. Finally, Figure 2.10 contains the performance results for all join methods when scaling the input data size. For the partitioning joins we observe that for very small input sizes, i.e. up to 4M tuples, the various algorithms show similar performance. However, if we scale the input data to larger sizes, we observe that the PR*- and CPR*-algorithms outperform the NOP*-algorithms, CHTJ, and MWAY. In particular, for the NOP*-algorithms we can see from Figure 2.10 that the throughput is very good up to 4M. However, for larger inputs the throughput decreases. This matches our expectation since no-partitioning join methods need to build a big global hash table. With growing data sizes the global hash table won't fit into the LLC anymore, which in our case is just 30 MB. Hence,

[5] As the LLC is shared between cores, the available share per thread is dependent on the number of concurrently running threads.

[6] Similar results for the other algorithms are not shown due to space constraints.

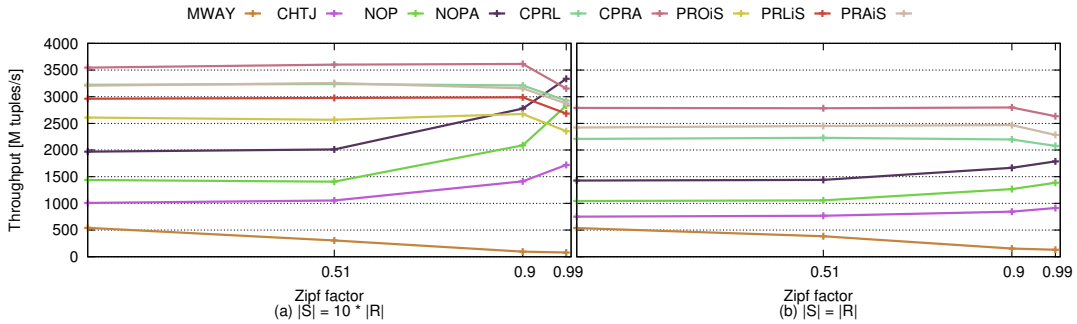


Figure 2.13: Throughput of join algorithms on skewed data. Relation size: |R| = 128M.

the bigger the build relation, the higher the probability for an LLC miss as well a TLB-miss. This effect can be observed very well in Figure 2.10(a): up to an input size for $|R|$ =32 M tuples there is a decrease in performance. Afterwards the performance does not deteriorate (visibly) anymore as almost all hash table accesses trigger LLC and TLB misses anyway (due to lacking spatial proximity in the caches). In other words, the NOP*-algorithms are already bound by this bottleneck.

The *inverse argument* to this holds for the PR*- and CPR*-algorithms. These algorithms perform *more* memory operations than the NOP*-algorithms. The underlying assumption of these algorithms, however, is that memory accesses to individual tuples may be very expensive, i.e. they may lead to a costly LLC miss. This effect is similar to external memory algorithms trying to avoid individual seeks on disk by bundling operations into larger granules. The PR*- and CPR*-algorithms try to access (and implicitly cache) memory according to larger granules, i.e. partitions and/or memory pages. This algorithmic pattern does not have much of an effect if the input data fits into LLC anyways (then, the underlying assumption of the algorithms simply does not hold). However, once the underlying assumption holds, i.e. the input data exceeds the size of the LLC, these algorithms can efficiently avoid costly DRAM accesses to individual tuples.

Notice that among the NOP*-algorithms, CHTJ is very sensitive to the data size as it needs at least two random accesses for every operation on its CHT. MWAY sort-merge join is another very stable algorithm, it even outperforms the CHTJ for large datasets.

2.8 Skewed Data Distributions

Until now we only looked at uniformly distributed data sets. In the next set of experiments we use different skew factors for the probe relation. We used an

algorithm proposed by Gray et. al. in [40] to quickly generate large amounts of skewed join keys. To achieve a more realistic distribution and to avoid that the key occurring most often, i.e. the smallest keys, are all in a single partition, we map the 10 smallest keys to random keys in the full domain. Figure 2.13 shows the performance of all algorithms with different zipf factors θ ranging from zero to 0.99. For every method we choose the number of threads such that the throughput was the highest. This means the no-partition algorithms make use of all hyper threads while the partition based algorithms only use a single thread per core to not pollute the private caches. We can see that lower levels of skew have no real impact on the performance of the algorithms.

When the skew factor is high we observe a shift in the throughput towards methods that do not partition the input. This has two reasons. First, the partition based methods have to handle skewed partition sizes, which is for now only handled automatically by a task queue. This means that the threads responsible for larger partitions are processing less partitions. We do not exploit the possibility to use multiple threads to process the join on the largest partitions in parallel. Second, a high skew factor makes the caches more effective, as the keys that are accessed most often are likely to be cached. For the partition based algorithms this effect is not helping, as the partitioning already makes the caches effective. We can see that the partition based approaches are still competitive with the no partition joins for the presented data size.

2.8.1 Scalability in number of threads

In this section we explore the scalability of the different join methods in terms of multithreading. All previous experiments were run using 32 threads. As the implementation of MWAY only works with a power of two many threads, we cannot report numbers for MWAY with more than 32 threads[7].

We take as a starting point four threads where each thread is assigned to one of the four NUMA regions. From that starting point we increase the number of threads distributing threads evenly across NUMA regions.

Figure 2.14 shows the results when scaling the number of threads from 4 to 120. All methods achieve good performance when using all physical cores, i.e. 60 threads. All partitioned-based approaches perform worse when using hyper-threading. This is understandable, as then even the private caches have to be shared among the hyper-threads. Even for the NOP*-joins the benefit of hyper-threading is minimal. This is also understandable for our fast hash function, as we do not have many computations that could hide the memory latency. A more com-

[7]We also evaluated MWAY using 64 threads, but the results are not competitive and also not fair, as only four out of 60 cores have to work on two threads.

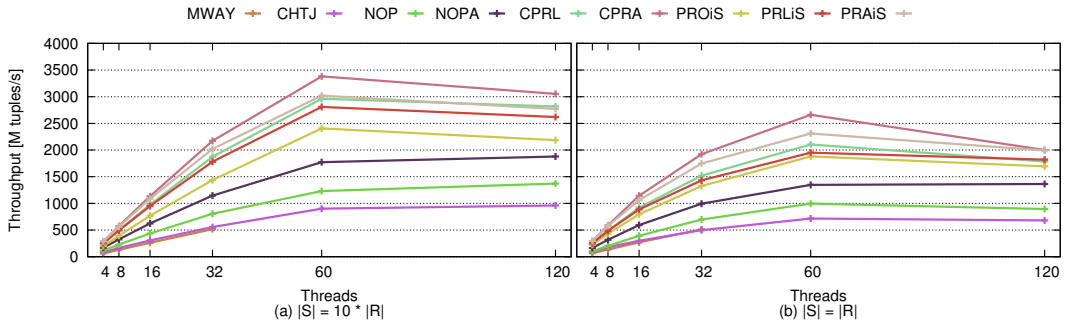


Figure 2.14: Throughput of join algorithms when scaling the number of threads. Size of relation |R| = 128M tuples

putationally intensive hash function could also benefit more from hyper-threads.

			Relative Speedup		
Join	**4 Threads [M/s]**	**60 Threads [M/s]**	**Total**	**Build or Partition Phase**	**Probe or Join Phase**
CHTJ	87.3	945.7	10.8	8.4	10.9
NOP	122.1	1291.2	10.6	9.4	10.7
NOPA	176.2	1859.5	10.6	6.8	11.2
CPRL	264.3	3105.4	11.7	12.1	10.6
CPRA	300.1	3545.1	11.8	12.4	10.1
PROiS	212.0	2522.9	11.9	11.3	13.2
PRLiS	263.7	2944.0	11.2	11.3	10.8
PRAiS	302.3	3168.0	10.5	10.7	9.8

Table 2.3: With |R| = 128M and |S| = 1280M

Tables 2.3 and 2.4 show a summary of the relative speedup for the join algorithms. We calculate the relative speedup as runtime($T > 4$ threads)/runtime(4 threads) where T is the number of threads used. Hence, the perfect speedup for $T = 60$ threads would be 15. Of course no method scales to this theoretically achievable perfect speedup, but CPRA and CPRL come close with a speedup of almost 12.

2.8.2 Holes in the key range

All previous experiments used a dense key range. In this section we want to study the effect of holes in the key range on the performance of the different join algorithms, especially on the array-based methods. We generate the build relation with a domain k times the size of $|R|$ for increasing values of k. Figure 2.15 shows

Join	4 Threads [M/s]	60 Threads [M/s]	Relative Speedup		
			Total	Build or Partition Phase	Probe or Join Phase
CHTJ	96.6	751.6	7.8	8.5	7.7
NOP	109.8	1043.7	9.5	9.7	9.3
NOPA	173.0	1413.1	8.2	7.1	9.8
CPRL	260.2	2207.9	8.5	8.5	8.4
CPRA	304.9	2790.2	9.2	9.2	9.1
PROiS	234.2	1971.2	8.4	7.3	12.9
PRLiS	263.5	2046.0	7.8	7.3	10.4
PRAiS	312.3	2422.4	7.8	7.3	9.6

Table 2.4: With |R| = |S| = 128M

the performance of the different join algorithms for varying domain sizes $k \cdot |R|$. We can see that the performance of the NOPA join is not influenced that much. This is expected, even without any holes in the domain is it very unlikely that neighboring elements in the array are probed in a short enough sequence for the second tuple to still reside in the caches. This slight chance is simply eliminated in the case of very large domains, as neighboring elements are very unlikely to even be present at all. The size of the used array is of course growing linear with the domain size and occupies a larger and larger part of the available memory for the whole time of the join processing. The partition-based array joins on the other hand suffer greatly from large domains, as the array does no longer fit into the caches for larger and larger domains. A possible remedy for the partition-based methods is to use more fine grained partitioning in the case of larger domains, such that the array again fits into the cache. We applied this technique to PRAiS and CPRA and depicted the performance as dashed lines in Figure 2.15. Please note, that the main memory consumption of PRAiS and CPRA is much lower compared to NOPA as we only construct temporary arrays on small partitions, that can be freed after processing the co-partition join. On another note, all our hash table implementations suffer a small performance hit when increasing the domain, as now we can observe some collisions in the hash table when inserting and probing the join keys.

From the results it looks like NOPA stays very competitive for arbitrary large domains, as long as you are willing and able to pay the additional memory overhead. PRAiS and CPRA can also deal with domains that are reasonably dense, especially when adapting the partition strategy to the domain size.

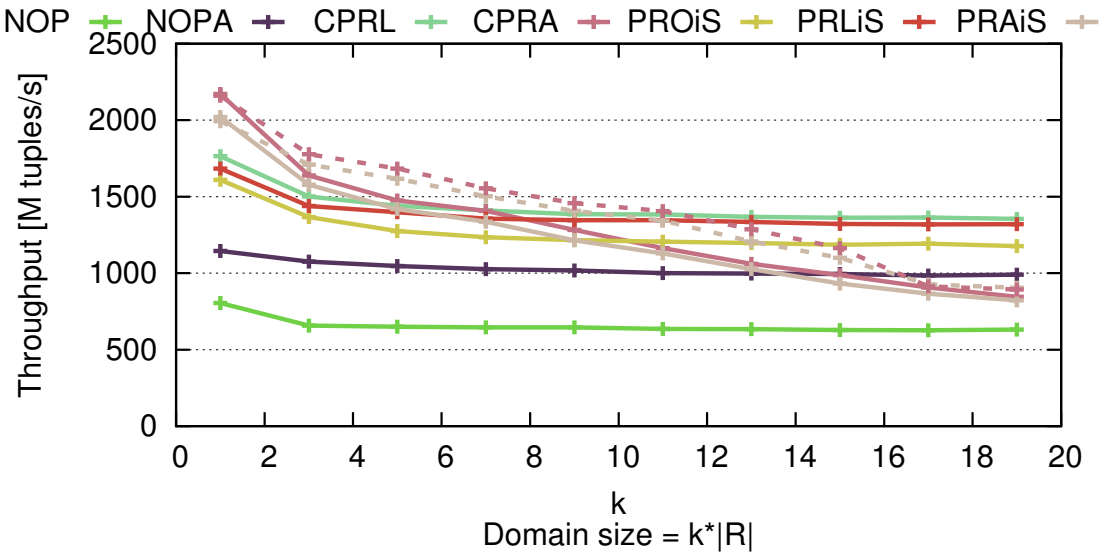


Figure 2.15: Performance of join algorithms with increasing domain size. $|R|$ = 128 M and $|S|$ = 1280 M. The dashed lines for CPRA and PRAiS denote the throughput when adapting the number of partitions to the domain size

2.8.3 Micro-architectural performance aspects

We also measured the performance of all presented join algorithms with respect to the number of cache misses and the instructions per cycle (IPC) metric. Table 2.5 shows all measurements. The number of cache misses are measured in millions while the instructions retired (IR) count is given in billions. We see that the

	Sort or Build or Partition Phase						Probe or Join Phase					
Join	**L2 Misses [M]**	**L3 Misses [M]**	**L2 Hit Rate**	**L3 Hit Rate**	**IR [B]**	**IPC**	**L2 Misses [M]**	**L3 Misses [M]**	**L2 Hit Rate**	**L3 Hit Rate**	**IR [B]**	**IPC**
MWAY	430	388	0.64	0.10	260	1.36	10	10	0.01	0.04	17	1.78
CHTJ	559	353	0.20	0.37	15	0.40	1911	1561	< 0.01	0.18	29	0.25
PRB	558	555	< 0.01	0.01	65	0.33	59	40	0.98	0.33	30	1.46
NOP	394	393	0.38	< 0.01	8	0.36	957	955	0.39	< 0.01	20	0.39
NOPA	409	391	< 0.01	0.04	6	0.27	335	320	< 0.01	0.05	5	0.28
PRO	981	209	0.51	0.79	42	0.87	60	45	0.98	0.26	30	1.34
PRL	791	209	0.45	0.74	41	0.90	86	52	0.93	0.40	24	1.08
PRA	396	110	0.61	0.72	41	1.05	88	52	0.92	0.42	17	0.83
CPRL	730	193	0.47	0.74	43	1.06	72	25	0.94	0.65	29	2.26
CPRA	341	85	0.65	0.75	44	1.24	64	24	0.94	0.62	22	1.94
PROiS	976	209	0.51	0.79	40	0.83	31	13	0.99	0.59	30	2.10
PRLiS	788	209	0.45	0.73	40	0.87	55	23	0.95	0.58	24	2.28
PRAiS	398	110	0.61	0.72	40	1.00	63	31	0.94	0.50	17	1.80

Table 2.5: Performance counter for the join with |R| = 128M and |S| = 1280M and 32 threads.

partition-based joins indeed lead to a dramatic reduction in cache misses and reach a cache hit rate of up to 99% for the join phase. Furthermore, the CHTJ suffers

from roughly two times the number of cache misses compared to NOP, due to the additional bitmap lookup, as expected. We can also observe that the partition-based algorithms need more instructions to perform the join but they also have a much higher IPC rate, that allows them to perform the join faster than the no-partitioning joins. Please note, that the different amount of cache misses for PRA, PRL, PRO in the partition phase stem from the fact that we use 12, 13, or 14 radix bits respectively according to Equation (2.1).

2.9 Effects on Real Queries

Up to now we focused on benchmarking raw performance of multithreaded joins. Like that we followed the micro-benchmarking philosophy of previous work [16, 14, 13, 62, 10, 72, 12, 60]. However, micro-benchmarks always trigger the same (and important) question: how big is the impact observed in micro-benchmarks in a larger context, e.g. when joins are used inside a larger query or inside a system.

The development of a full-fledged multithreaded NUMA-aware query execution engine is beyond the scope of this chapter. However, state-of-the-art main-memory databases use code compilation anyways [68], i.e. at query time they translate incoming SQL to machine code (a standalone program if you wish) and then execute that program kind of independently from the remaining system (if you do not require locking which is the case for us). Therefore we decided to simply emulate a column store in C++. Similar to MonetDB we represent every column as a separate array consisting of <virtual oid,value> pairs, where the virtual oid is given implicit by the position of the value in the array. We choose TPC-H query 19 (Listing 2.1 as that query is the only query that contains a single join followed by an aggregation without any subqueries. This query joins the `Lineitem` table with the `Part` table.

Both tables are stored as a struct of arrays and additionally, we dictionary-compress all string columns. We used float values instead of arbitrary precision numeric values. All foreign and primary key columns are represented as <Key , Payload> pairs with the row ID as the payload, this made it easier to use the join implementations with minimal modifications.

On a high-level we execute this query according to the query execution plan depicted in Figure 2.16. We obtained this plan through textbook query optimization. Notice that the same plan is also used by HyperDB according to the explain functionality of their web interface [8]. In this plan, the selection, that was pushed down to the scan of the `Lineitem` table, has a selectivity of 3.57%. This means that for scale-factor 100 the build relation Part has 20 M tuples and the probe

[8] `http://hyper-db.de/interface.html`

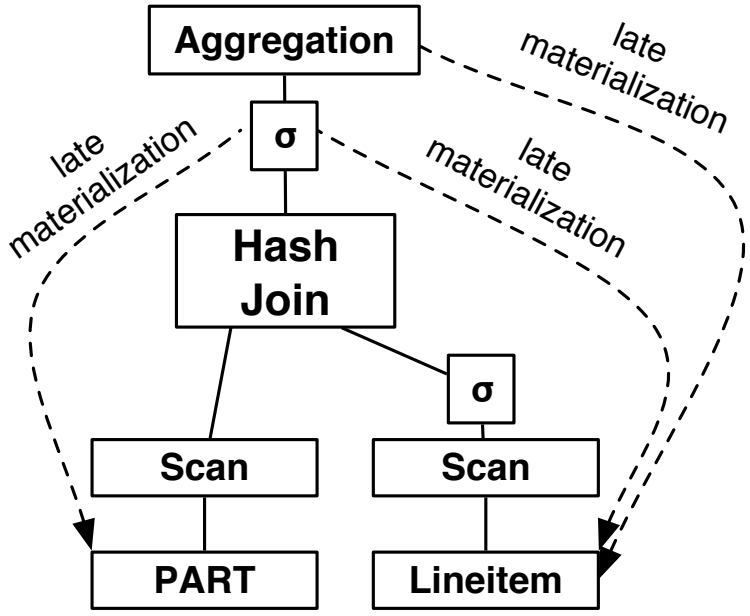


Figure 2.16: Optimized semi-physical query plan for TPC-H Q19 plus materialization strategy in the column store

relation Lineitem has 600 M tuples (600 M· 3.57% = 21.42 M tuples after filtering), i.e. in the join both relations have roughly the same size.

For scale-factor=100 this corresponds to the results of our micro-benchmark of Figure 2.10(b) for $|R| \approx 20$ M tuples. Notice that the plan in Figure 2.16 is actually only *semi-physical* as it does not specify *when* to reconstruct tuples. We used late materialization, i.e. all attributes are only touched when required by an operation[9].

Figure 2.17 shows the runtime of Q19 for TPC-H for scale-factors 100. This figure includes a cost breakdown where the colored bars represent the time spent for the join. In contrast, the black bars represent the time spent in other parts of the query. We obtained the numbers for the colored bars by executing each join just like in the micro-benchmarks above, i.e. each of the four join algorithms receives a build input of 20 M tuples and a pre-filtered (and pre-materialized) probe input of 21.42 M tuples. The difference of the execution time of the query and the join smicro-benchmark yields the black bars[10].

We immediately observe that even for this relatively simple query a major part

[9]This strategy is also used by MonetDB.

[10]This method is not entirely fair as the join- and non-join parts of a query may overlap. However, it gives a good indication on how much of the total query time is actually due to the actual join.

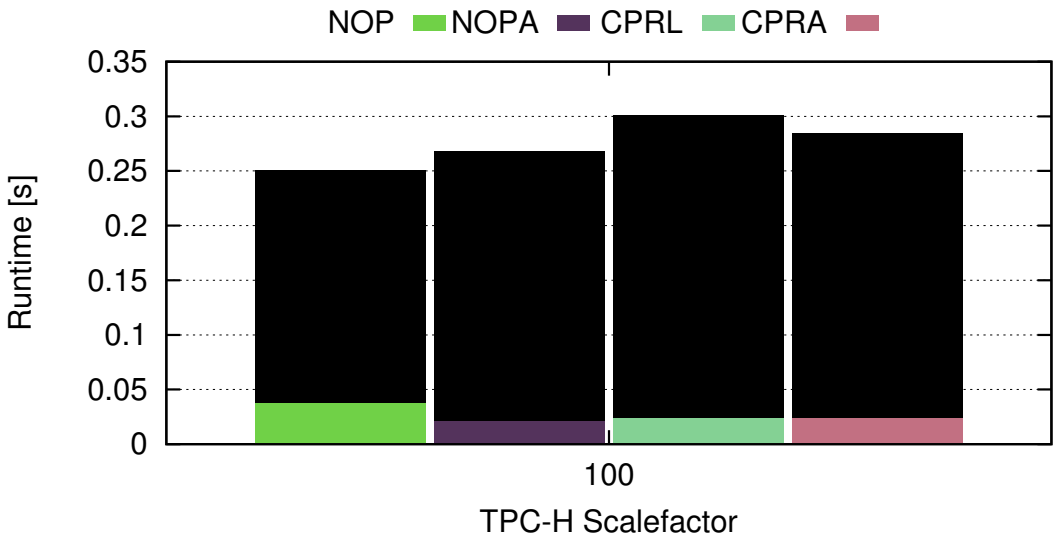


Figure 2.17: Runtime of TPC-H Query 19, colored bars mark the fraction of the time spent in the actual join; the black bars mark the time spent for the rest of the query.

of the runtime is spent in the non-join parts of the query. The time spent in the actual join is only about 10%–15% of the total runtime! In addition, for some methods more time is spend outside the join than for others.

There are several reasons for this: First, the join key column is a primary dense key and the `Part` table is even generated in sorted order according to this key. This means that we have an ideal sequential access pattern for NOPA when building the join array (also compare our discussion in Section 2.5.2 as well as additional experimentation in Section 2.8.2). Second, scanning and filtering 600 M tuples from Lineitem down to 21.42 M tuples simply eats up some time. Third, in contrast to the micro-benchmark experiments, for Q19 we have to access several attributes other than the join key. This happens in multiple places: in order to evaluate the complex predicate after the probe and to aggregate the final result, i.e. we have to perform implicit positional joins (for tuple reconstruction). In particular, In NOPA neither the `Lineitem` table nor the `Part` table have to be partitioned. Therefore, all other attributes *stay aligned with the join array and the probe relation*. This is especially beneficial for all attributes of the probe relation since they are accessed sequentially when evaluating the complex join predicate. In contrast, for the CPR*-algorithms those benefits do not apply. If we access attributes that are not the join key, we have to follow the row ids contained in the narrow join tuples. Those row-ids point to arbitrary locations after partitioning the data. This means that we pollute our cache and TLB with data from other

attributes and lose locality in our accesses. Therefore, it would be beneficial to explore tuple reconstruction strategies for CPR*-joins in more detail.

2.9.1 Details on used Query

Listing 2.1 contains the full SQL code of TPC-H query 19.

Listing 2.1: TPC-H Query 19

```
select
        sum(l_extendedprice* (1 - l_discount)) as revenue
from
        lineitem,
        part
where
(
p_partkey = l_partkey
and p_brand = 'Brand#12'
and p_container in ('SM␣CASE', 'SM␣BOX', 'SM␣PACK', 'SM␣PKG')
and l_quantity >= 1 and l_quantity <= 1 + 10
and p_size between 1 and 5
and l_shipmode in ('AIR', 'AIR␣REG')
and l_shipinstruct = 'DELIVER␣IN␣PERSON'
)
or
(
 p_partkey = l_partkey
and p_brand = 'Brand#23'
and p_container in ('MED␣BAG', 'MED␣BOX', 'MED␣PKG', 'MED␣PACK')
and l_quantity >= 10 and l_quantity <= 10 + 10
and p_size between 1 and 10
and l_shipmode in ('AIR', 'AIR␣REG')
and l_shipinstruct = 'DELIVER␣IN␣PERSON'
)
or
(
p_partkey = l_partkey
and p_brand = 'Brand#34'
and p_container in ('LG␣CASE', 'LG␣BOX', 'LG␣PACK', 'LG␣PKG')
and l_quantity >= 20 and l_quantity <= 20 + 10
and p_size between 1 and 15
and l_shipmode in ('AIR', 'AIR␣REG')
and l_shipinstruct = 'DELIVER␣IN␣PERSON'
)
```

Listing 2.2: Data-structures to represent the Lineitem and Parts tables

```
struct LineitemTable {
```

```
    size_t numTuples;
    float *l_extendedprice;
    float *l_discount;
    tuple_t *l_partkey;
    unsigned int *l_quantity;
    uint8_t *l_shipmode;
    uint8_t *l_shipinstruct;
};

struct PartTable {
    size_t numTuples;
    tuple_t *p_partkey;
    uint8_t *p_brand;
    uint8_t *p_container;
    unsigned int *p_size;
};
```

Our simulated column store represents the TPC-H tables as structs of column pointers as depicted in Listing 2.2. Please note, that we only represent the columns accessed in TPC-H Q19. The type `tuple_t` is a <key,payload> pair with the rowID as the payload. We depict the filter predicate implementation in Listing 2.3.

Listing 2.3: Filter conditions

```
inline bool preJoin(LineitemTable *l, size_t rowID) {
    return (l->l_shipinstruct[rowID] == DELIVER_IN_PERSON &&
            (l->l_shipmode[rowID] ==AIR||l->l_shipmode[rowID] ==
                AIR_REG));
}

inline bool postJoin(LineitemTable *l, PartTable *p, size_t rowIDL,
    size_t rowIDP) {
    uint8_t p_brand = p->p_brand[rowIDP];
    uint8_t p_container = p->p_container[rowIDP];
    auto l_quantity = l->l_quantity[rowIDL];
    auto p_size = p->p_size[rowIDP];
    return (p_brand == BRAND12
            && (p_container == SM_CASE || p_container == SM_BOX ||
                p_container == SM_PACK || p_container == SM_PKG)
            && l_quantity >= 1 && l_quantity <= 1 + 10
            && 1 <= p_size && p_size <= 5) ||
           (p_brand == BRAND23 &&
            (p_container == MED_BAG || p_container == MED_BOX ||
                p_container == MED_PKG || p_container == MED_PACK)
            && l_quantity >= 10 && l_quantity <= 10 + 10
            && 1 <= p_size && p_size <= 10) ||
           (p_brand == BRAND34 &&
            (p_container == LG_CASE || p_container == LG_BOX ||
                p_container == LG_PACK || p_container == LG_PKG)
```

```
            && l_quantity >= 20 && l_quantity <= 20 + 10
            && 1 <= p_size && p_size <= 15);
}
```

These predicates correspond one-to-one to the predicates in the SQL query depicted in Listing 2.1.

Listing 2.4 shows the pseudo code for the Q19 implementation using the NOP join. All threads build a hash table on `p_partkey` concurrently. Afterwards, every thread is responsible for a fixed chunk of tuples of the probe relation and first accesses the necessary attributes in `LineitemTable` to evaluate the preJoin predicate. All passing tuples from the `LineitemTable` are probed against the hash table and as soon as a join partner is found the postJoin predicate is evaluated. If the matched tuples pass this predicate, the `l_extendedprice` and `l_discount` attributes are immediately accessed and added to the final aggregate. With this execution strategy it is not necessary to materialize a join index for further processing. This execution strategy also corresponds to the strategy described for the HyperDB system [68].

Listing 2.4: Q19 Pseudo code for NOP

```
1  query_result_t
2  NOPQ19(LineitemTable *L, PartTable *P, int threadCount) {
3    parallelBuild(P,threadCount);
4    parallel for in chunks numTuples/threadCount
5    for (int i=0;i < L-> numTuples;++i) {
6      if (preJoin(L,i)) {
7        auto tuple=probe(L->l_partkey);
8        if (postJoin(L,P,i,tuple.rowID)) {
9          res+=L->l_extendedprice[i] * (1.0 - L->l_discount[i]);
10       }}}}
```

2.9.2 Varying the Selectivity of the Selection in Q19

We also measure the query performance with a varying selectivity on the probe relation. Figure 2.18 shows that the partition based joins indeed outperform the no partition based joins when the actual probe relation in the join becomes large.

2.9.3 Further cost-breakdown of Q19

We designed an additional experiment for NOP just to find out how much individual components of that query contribute to the overall runtime of that query. The core idea of this experiment is to start with the “naked join”, i.e. the microbenchmark, and then gradually morph the microbenchmark into TPCH-Q19. Like that we see at each step the overhead introduced by that step.

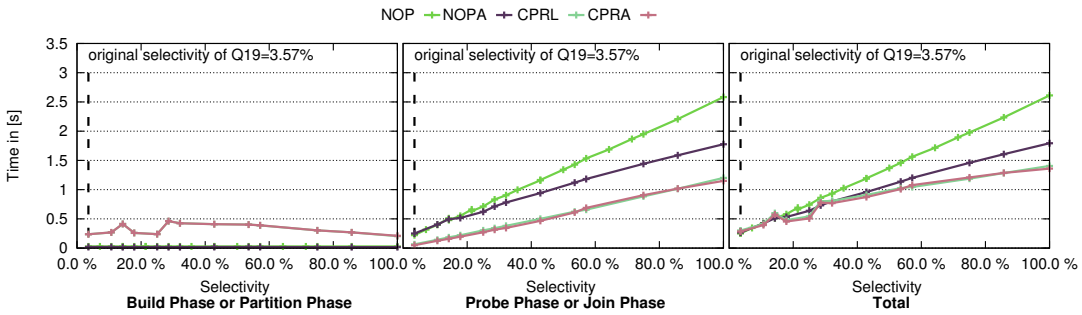


Figure 2.18: Runtime of TPC-H Query 19 (sf=100) when varying the selectivity of the pushed-down selection predicate

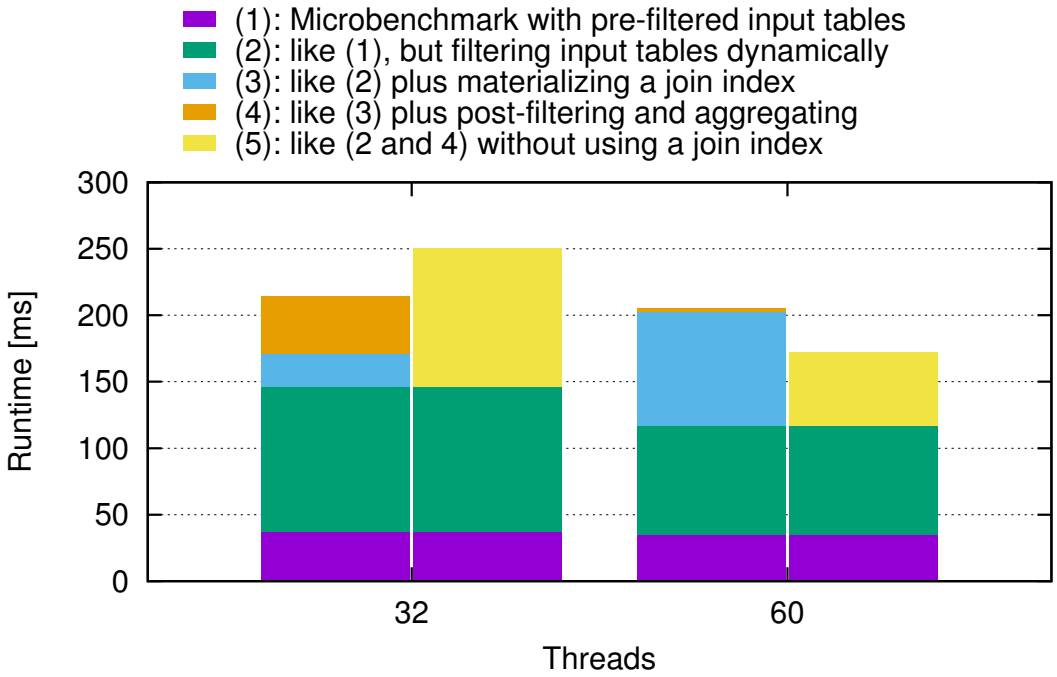


Figure 2.19: Additional cost-breakdown morphing a microbenchmark stepwise into Q19.

From the results in Figure 2.19, we can already learn many things:

(1.) Tuple reconstruction is **not** the main culprit for the overheads. In fact, for this query filtering the input rows eats up most of the additional time for both 32 and 60 threads.

(2.) Even writing out join results first into a join index and then doing all additional work like post-filtering, tuple reconstruction (in the same order as before!),

and aggregating is faster than running all of this in a pipeline! But only for 32 threads! For 60 threads this does not hold anymore and the results turn upside down: here the overheads for creating and using a join index do not pay off anymore.

(3.) for 32 threads there is room for a tuple reconstruction algorithm, i.e. at most ∼20% performance improvement seem possible (for a tuple reconstruction running in zero time).

2.10 Lessons Learned

(1.) Don't use CPR* algorithms on small inputs. For input relations with less than 8 million tuples we do not observe a benefit of partitioning local chunks instead of the global relation. On the contrary, we even observe a performance degradation. This is mainly due to two things: (1) the overhead of creating all the threads does not pay off for the small input data. This is also true for all other presented algorithms. (2) the size of a chunk becomes smaller than a page. This leads to a random allocation of the pages to different NUMA nodes. Several threads will have to read from and write to remote memory, even though the main advantage of the CPR* algorithms over the PR* algorithms should be to avoid remote writes. For very small inputs the NOP* algorithms become very interesting, especially if the build relation starts to fit into the LLCs.

(2.) Clearly specify all options used in experiments. This sounds like common sense for any experimental study. Still, we list it here again as a gentle reminder since we ran into this problem when we were trying to interpret results from different papers. As the implementations provided by authors typically have multiple optimization options, it is sometimes hard to understand which optimizations were actually used in a paper. Rather over-specify than under-specify your experiments.

(3.) If in doubt, use a partition-based algorithm for large scale joins. In this chapter, we have studied the performance of three variants of no-partitioning algorithm and eight variants of partition-based algorithms with different workloads by varying ratio between build and probe relations, data size, number of threads and skewness. **All** Partition-based algorithms outperform **all** no-partitioning algorithms in almost all except for only two cases. The first case is when the size of input relations scales to 32 GB where the worst partition-based algorithm is a little bit slower than the best no-partitioning algorithm. The second case is when the probe relation is highly skewed such that partition-base algorithms suffer from unbalanced loads between threads while no-partitioning algorithms have less cache misses. No-partitioning algorithms start outperforming partition-based algorithms

only for a Zipf factor > 0.9.

(4.) Use huge pages. As we discussed in Section 2.7.2, all algorithms except PRB benefit from using huge pages. Using huge pages means less pages are needed for a certain amount of data, thus reduces the pressure on TLB system.

(5.) Use Software-write combine buffer. Software-write combine buffers are a very effective technique to reduce the number of TLB misses. Hence, using SWWCBs makes it possible to use single pass algorithm which significantly accelerates partition-based algorithms.

(6.) Use the right number of partition bits for partition-based algorithms. Partition-based algorithms are very sensitive to choosing the right number of radix bits. For different data sizes, one has to choose different number of partition bits to get the optimal performance. As shown in Figure 2.9, choosing suboptimal number of bits can lead to performance degradation by up to a factor of 2.5.

(7.) Use a simple algorithm when possible. In our study, simple ideas turned out to be surprisingly efficient and effective. For instance, array joins are very efficient in the case of dense primary key distributions. They outperform other non-array variants under all workloads by up to 44%. Chunking is another simple idea we used to create faster join algorithms. Chunking eliminates remote memory writes in partition phase and improves the join performance by up to 26%.

(8.) Be sure to make your algorithm NUMA-aware. Back at the time when PRO and PRB [13] were published, the authors ran experiments on several single socket machines. These algorithms were designed for multicore systems but not yet for NUMA systems. Directly running these algorithms on NUMA systems will yield suboptimal performance. We evaluated chunking to eliminate writes to remote memory and explored NUMA-aware scheduling to avoid bandwidth saturation on a single memory controller. These two optimizations improve the performance over non-NUMA-aware algorithms by up to 26% and 20%, respectively.

(9.) Be aware that join runtime≠query time. Our experiments with a TPC-H query clearly indicated that the join time may actually only be a 10%–15% share of the total runtime of the query. We identified multiple reasons that lead to interesting avenues for future work. However, already at this point it again emphasizes that micro-benchmarks alone may be misleading in case we want to understand performance in a bigger context, e.g. an entire query.

2.11 Conclusions

In this chapter, we evaluated thirteen main-memory join algorithms in a common setting. We resolved some contradicting results and showed that hardware-conscious partition-based approaches typically outperform hardware-oblivious no-partition based joins on modern multi-core NUMA architectures. At least this is the case if the probe relation is not highly skewed; for very skewed data the unpartitioned hash table can match or even outperform the partition-based approaches. We also presented new partition-based approaches, called CPRL and CPRA that often outperform the PR*-algorithms from prior work. Overall CPRL and CPRA achieve a remarkable join throughput of up to 3.4 billion input tuples per second.

So should we finally consider relational joins a solved problem? The bad news is, we will probably never be able to label it 100% solved as there will always be some fancy new SIMD instruction or whatever "new" hardware that may impact the relative performance differences. The good news, with this study we believe we made major steps forward in understanding the performance of state-of-the-art join algorithms as of 2015. However, as stated above, almost all previous works, e.g. [16, 14, 13, 62, 10, 72, 12, 60], evaluated micro-benchmarks only. We departed from that and evaluated the runtime of the most promising join algorithms when used in a simple TPC-H query. This initial experimentation already reveals that only a fraction of the query runtime may be spent in the actual join, a majority may be spent in other parts of the query including scanning, filtering, and tuple reconstruction. Hence, as future work we would like to evaluate the cross product of different join algorithms and the large space of tuple reconstruction algorithms, in particular for the very promising CPR*-family of join algorithms.

Chapter 3

HAIL: Hadoop Adaptive Indexing Library

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 [42]. In particular, log processing emerges as an important type of data analysis commonly done with MapReduce [17, 65, 32].

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 [91, 41, 64]. Over the last years, a lot of research works have focused on improving the performance of Hadoop MapReduce [24, 45, 52, 56]. 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 [94, 28, 64, 54]. 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

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

a serious problem if the workload 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 [17, 32, 69]. 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 [74, 54]. 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 [54, 24, 45, 52]) and finds a paper that shows how to create a so-called *trojan index* [28]. 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:

1. *Expensive index creation.* The time to create the trojan index on sourceIP (or

[2]The professor is aware that for some situations the opposite is true.

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 [56] 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. Afterwards, 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 [33, 22, 6, 20, 23, 94, 64, 28, 54] 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* [47], where the general idea is to create indexes as a side-effect of query processing. This is similar to the idea of *soft indexes* [66], 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 [30, 47, 48, 38, 50, 43] 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 to 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, that 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 [38] 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 [48] introduced lazy tuple reorganisation 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 chapter addresses the following research questions:
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?
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?
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 non-indexed 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?
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?

3.2 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 Hadoop and HDFS

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 [26, 28, 27] 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

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

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, that resembles PAX [8] 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 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

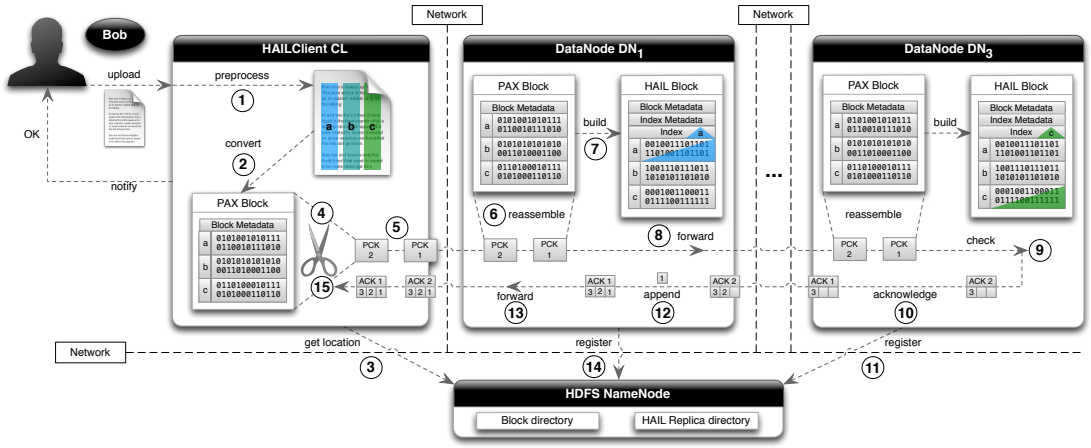


Figure 3.1: The HAIL static indexing pipeline as part of uploading data to HDFS

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 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.

Alternatively we could naively piggy-back on the existing HDFS upload pipeline

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

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 non-trivial 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 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, 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 storing 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 before ⑧. 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 non-selective 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).

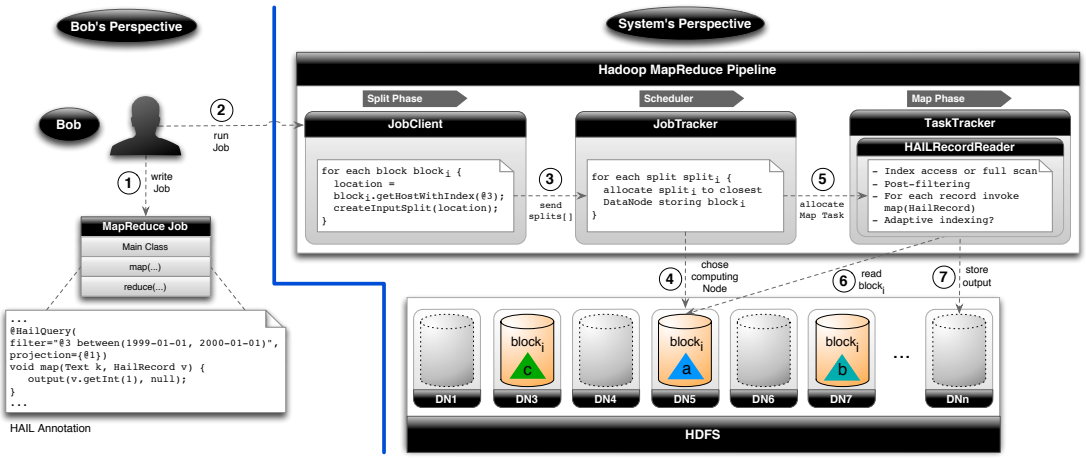


Figure 3.2: The HAIL query pipeline

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), instead of the default *InputFormat*, 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})
```

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

```
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. For example, using standard Hadoop MapReduce 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);
}
```

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 [26]. 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 the HDFS ⑦. See [28, 92, 27] for more details on the MapReduce execution pipeline.

In HAIL, it is crucial to be non-intrusive 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 open an input stream to a given HDFS block based on data-locality and nodes availability only anymore: it has to consider index locality and availability as well. HAIL overcomes these issues by mainly providing two UDFs: the HailInputFormat and the HailRecordReader. Notice, that 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 if a record is bad or not.

3.4.4 Problem: 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 again 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 data 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.

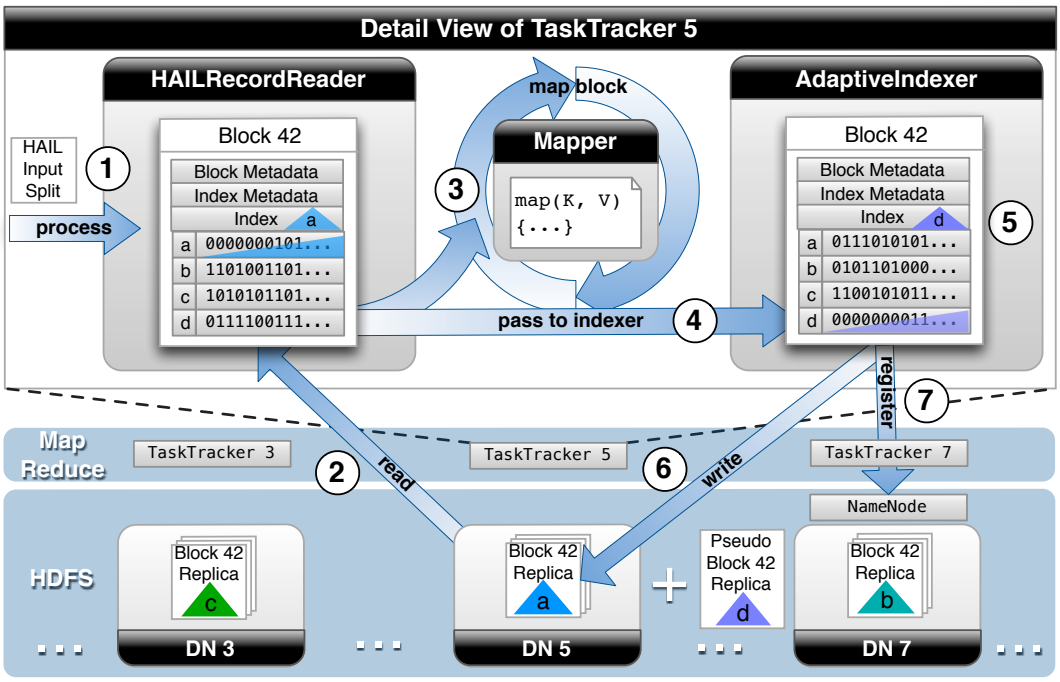


Figure 3.3: HAIL adaptive indexing pipeline.

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 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 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

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

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

[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.

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 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, as depicted in 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 $\{old\_position \mapsto new\_position\}$ 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 parallelise 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

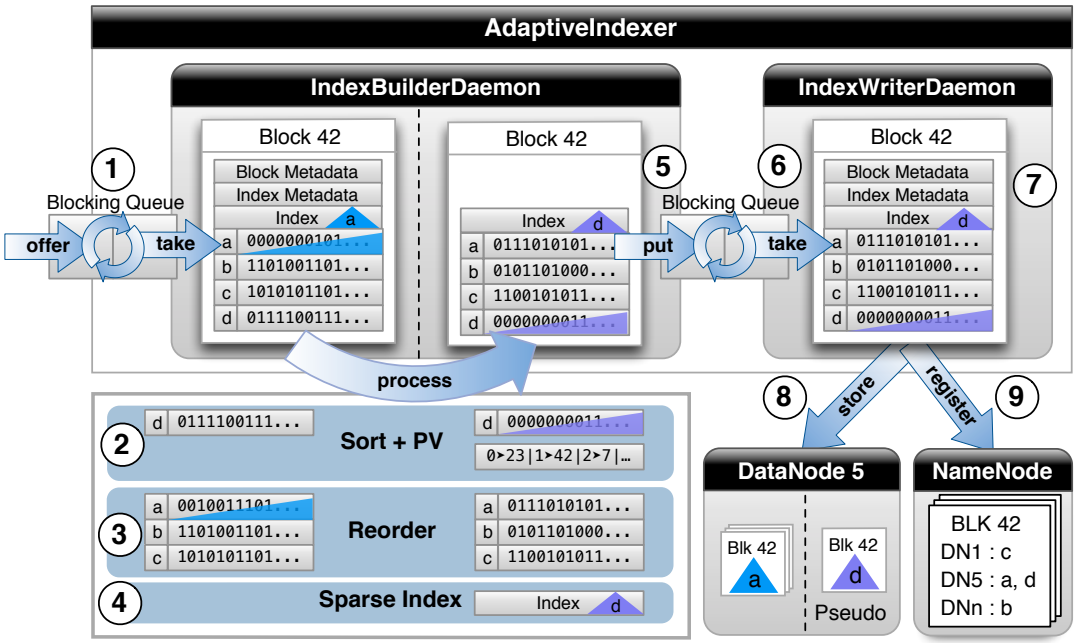


Figure 3.4: AdaptiveIndexer internals.

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 recognise 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:
How 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 FSInputStream.
How 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 chapter and is subject of the following chapter, Chapter 4.
How 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 HAILRecordReader initialises the HAILInputStream with the selection predicate of job_d as a parameter ②. Internally, the HAILInputStream checks if the index resides in a normal or pseudo data block replica ③. 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

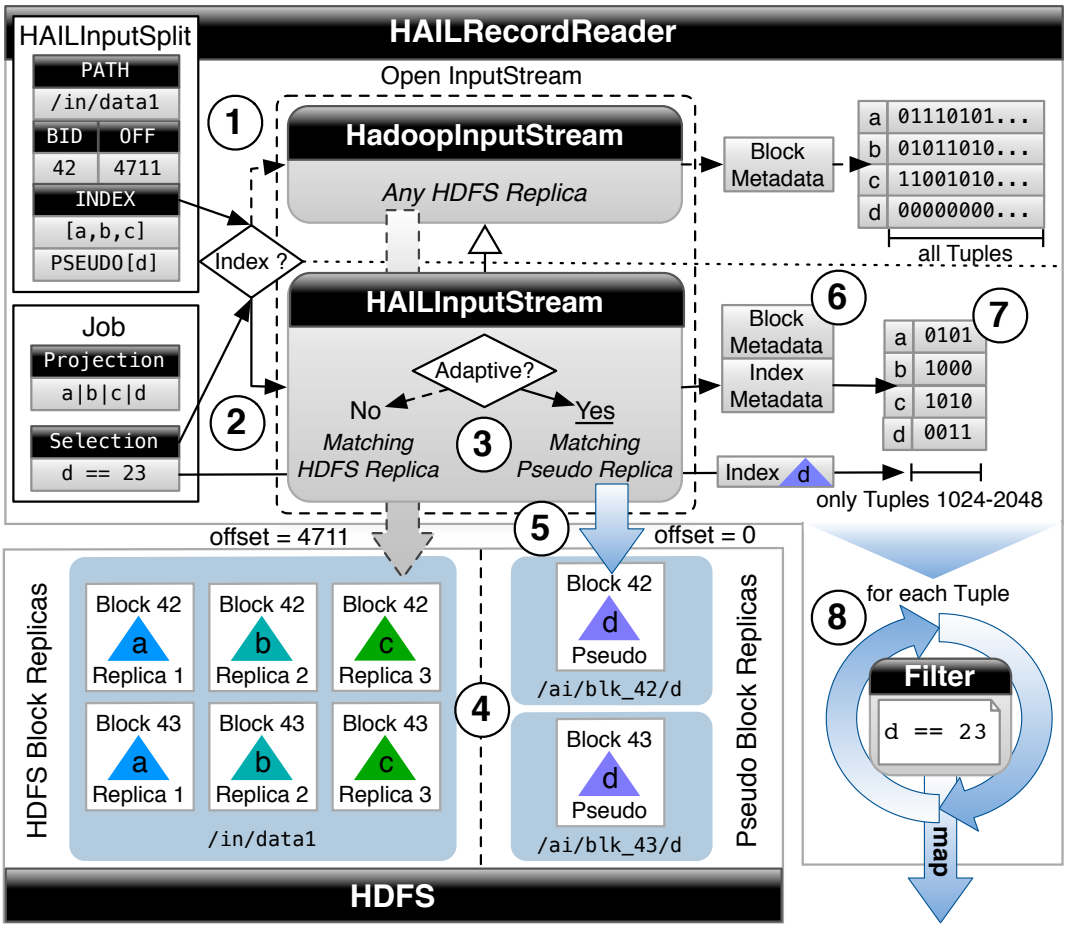


Figure 3.5: HAILRecordReader internals.

replica belongs to a different HDFS file ④. 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 path /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 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 prioritise 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 one 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

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

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.

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 prioritises 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 contains 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 applications. 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 chapter.

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 [77]. 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 [77] 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 [56]. Moreover, one can significantly improve data locality by simply using an adequate scheduling policy (e.g. the Delay Scheduler [95]). 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 [8] 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 [56], we showed how to improve over PAX by computing different layouts on the different replicas, but we did not consider indexing. This chapter fills this gap.

Static Indexing. Indexing is a crucial step in all major DBMSs [33, 22, 6, 20, 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 [94, 64, 28, 54]. 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 [83, 18, 19, 66]. The idea is to continuously monitor the performance of the system and create (or drop) indexes as soon as it is considered beneficial. Manimal [21, 52] 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 [47] which aims at removing the high upfront cost barrier of index creation. The main idea of database cracking is to start organising 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 reorganising data inside data blocks. Third, HAIL parallelises the indexing effort across several computing nodes to minimise the indexing overhead. Fourth, HAIL focuses on creating clustered indexes instead of unclustered indexes. A follow-up work [48] 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 [49], concurrency control [37], and robustness [43], but these works are orthogonal to the problem we address in this chapter.

Adaptive Merging. Another related work to HAIL is the adaptive merging [38]. 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 [50]. 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 [4] or in a lazy [46] 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, [4] 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 [9]. 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 [9] 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

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 first wave of experiments in order to answer the following questions as well:

1. What is the performance of HAIL at upload time? What is the impact of static indexing in the upload pipeline? How many indexes can we create in the time the standard HDFS uploads the data? How does hardware performance affect HAIL upload? How well does HAIL scale-out on large clusters? (We answer these questions in Section 3.9.3).

2. What is the performance of HAIL at query time? How much does HAIL benefit from statically created indexes? How does query selectivity affect HAIL? How do failing nodes affect performance? (We answer these questions in Section 3.9.4). How does HailSplitting improve end-to-end job runtimes? (We answer this question in Section 3.9.5).

But, *what happens if Bob did not create the right indexes upfront? How 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 to answer the following main questions:

3. What is the overhead of running the adaptive indexing techniques in HAIL? How fast can HAIL adapt to changes in the query workload? How much can MapReduce jobs benefit from the adaptivity of HAIL? How well does each of the adaptive indexing technique of HAIL allow MapReduce jobs to improve their runtime? (We answer these questions in Section 3.9.6)

3.9.1 Hardware and Systems

Hardware. We use six different clusters. One is a physical 10-node cluster. 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 [82]. 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 [28], and (3) HAIL as described in this chapter. 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 [74]. This dataset nicely matches Bob's Use Case. We generated 20GB of UserVisits data per node using the data generator proposed by [74]. 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/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: 3.1×10^{-2})

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

Bob-Q2 (selectivity: 3.2×10^{-8})

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

Bob-Q3 (selectivity: 6×10^{-9})

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

Bob-Q4 (selectivity: 1.7×10^{-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: 2.04×10^{-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.

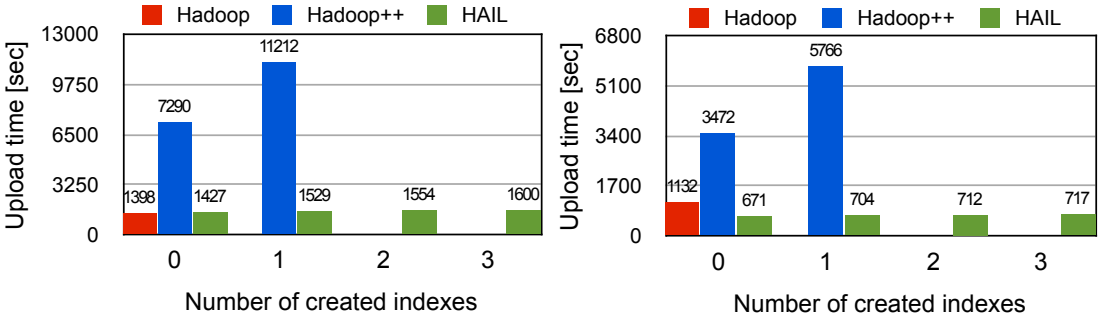
3.9.3 Data Loading

We strongly believe that upload time is a crucial aspect for adopting 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

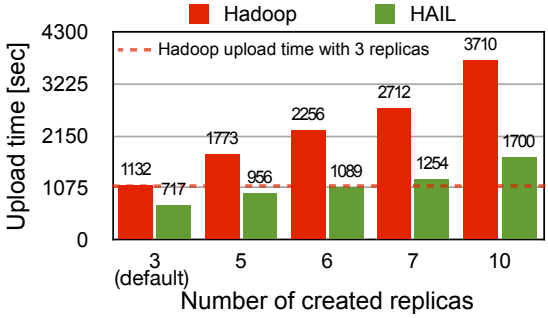
We first measure the impact in performance when creating indexes statically. For this, we scale the number of indexes to create when uploading the UserVisits and the Synthetic datasets. For HAIL, we vary the number of indexes from 0 to 3 and for Hadoop++ from 0 to 1 (this is because Hadoop++ cannot create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

Figure 3.6(a) shows the results for the UserVisits dataset. We observe that HAIL has a negligible upload overhead of $\sim$2% over standard Hadoop. Then, when HAIL creates one index per replica the overhead still remains very low (at most $\sim$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



(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)

for creating one index. For HAIL, we observe that for two and three indexes the upload costs increase only slightly.

Figure 3.6(b) 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 3.6(c) 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 by 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

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.

We report the results of these experiments in Table 3.3(a) (for UserVisits) and in Table 3.3(b) (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

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

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

(a) Upload times for `UserVisits` 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

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

Table 3.3: Scale-up results

decreases (in Table 3.3(a)) or increases (Table 3.3(b)) 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 system speedup of HAIL over Hadoop is even better when using physical nodes.

Cluster Scale-Out

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

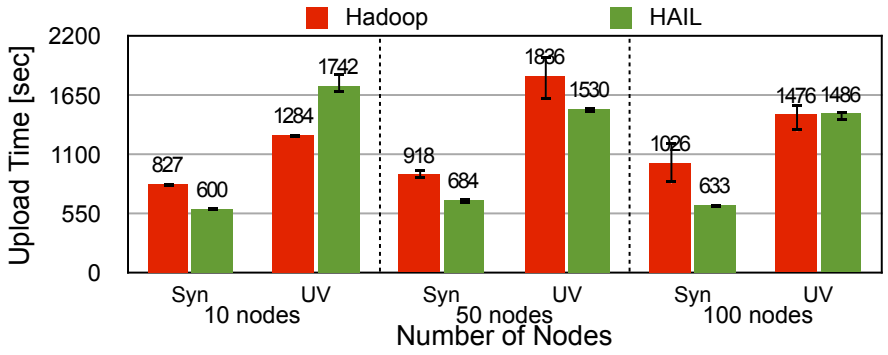


Figure 3.7: Scale-out results

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 [82]. Overall, these results show the efficiency of HAIL when scaling-out.

3.9.4 MapReduce Job Execution

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 3.8(a) 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 3.8(b) 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++.

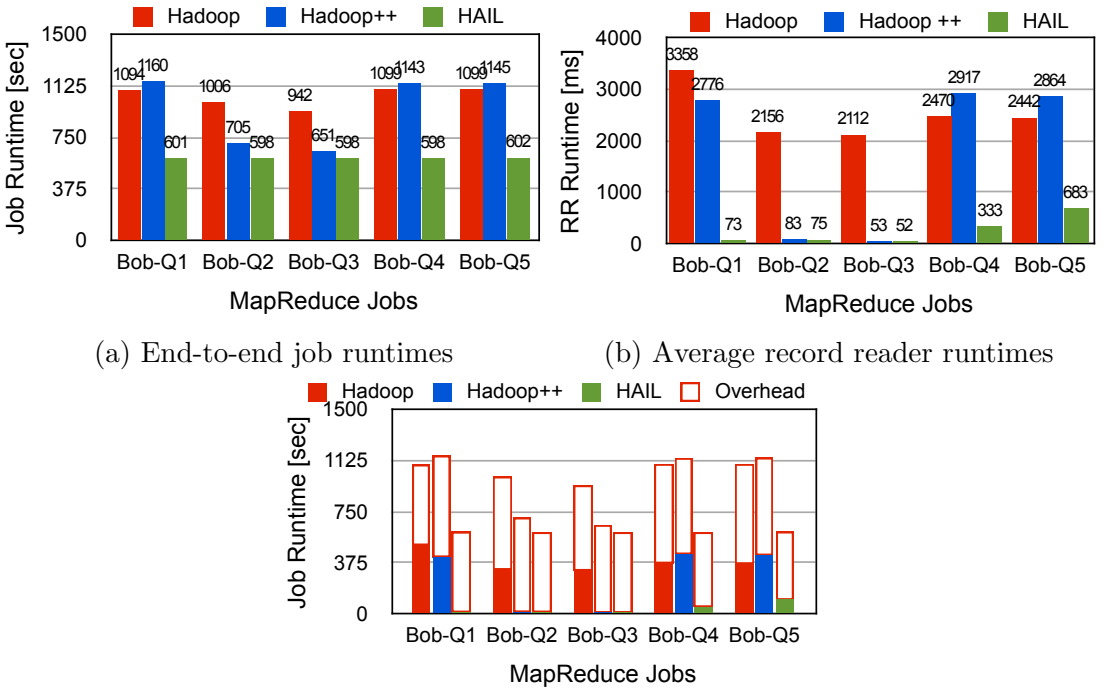
Yet, if HAIL allows map tasks to read their input data by more than one order of magnitude faster than Hadoop and Hadoop++, *why do MapReduce jobs not benefit from this?* To understand this we estimate the overhead of the Hadoop MapReduce framework. We do this by considering an ideal execution time, 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 3.8(c). 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 the performance of HAIL. 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 3.9(a) and the record reader times in Figure 3.9(b). We observe in Figure 3.9(a) 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

[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

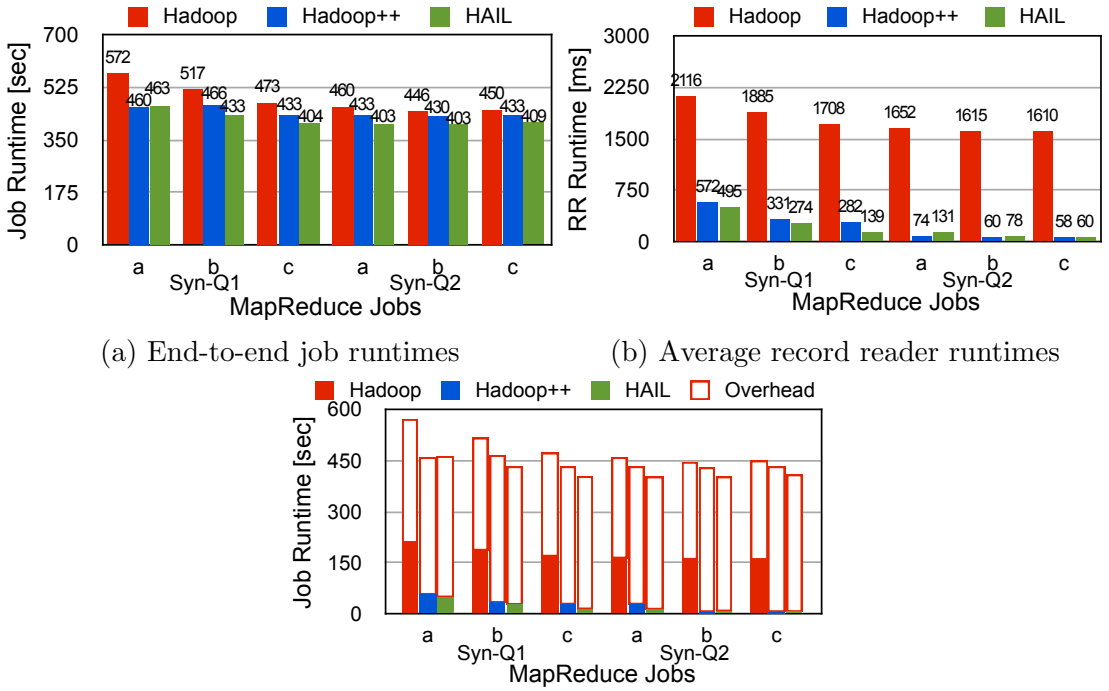
(c) Hadoop scheduling overhead

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

in Figure 3.9(b), the reader might think 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 3.9(a)) even if query selectivity has a clear impact on the RecordReader times (see Figure 3.9(b)). 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 3.9(c). In Section 3.9.5, we will investigate this in more detail.

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



(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

Fault-Tolerance

In very large-scale clusters (especially in 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 [28], $slowdown = \frac{(T_f - T_b)}{T_b} \cdot 100$, 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

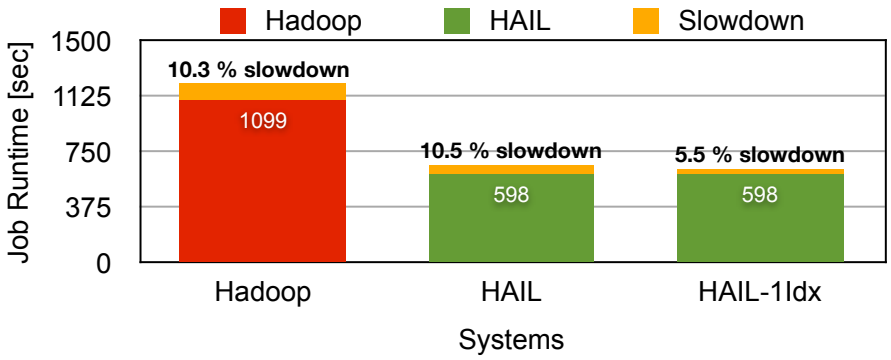
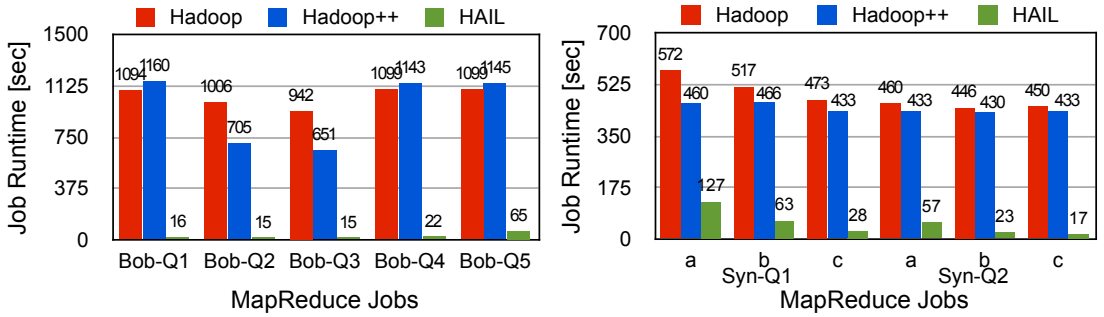
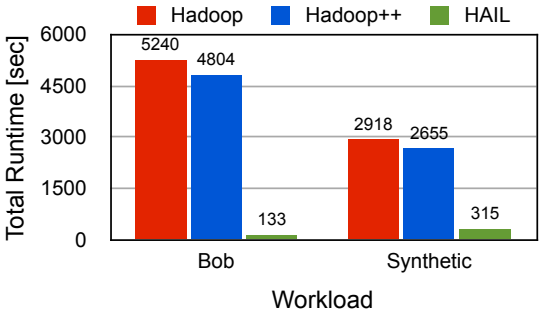


Figure 3.10: Fault-tolerance results



(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

reading its input from the killed node. Notice that, 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 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.

3.9.5 Impact of the HAIL Splitting Policy

We observed in Figures 3.8(c) and 3.9(c) 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 3.11(a) 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 3.11(b)), 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 3.11(c) 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++.

3.9.6 HAIL Adaptive Indexing

In the previous experiments we focused on the performance of HAIL with static indexing only, i.e., we deactivated HAIL adaptive indexing. For the following experiments we now focus on the evaluation of the HAIL adaptive indexing pipeline.

In addition to the 10-node cluster (*Cluster-A*) we used in previous experiments, we use 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.

Since the results from previous experiments 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 ($\rho = 0.1$), HAIL ($\rho = 0.25$), HAIL ($\rho = 0.5$), and HAIL ($\rho = 1$). Notice that HAIL with no adaptive indexing is the same as HAIL ($\rho = 0$). 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.

Moreover, making use of the lessons learned from the first wave of experiments, 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., the values $i \in [1..10]$ are repeated 10^{i-1}times. 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.

MapReduce Jobs. For the UserVisits dataset, we consider eleven 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.

Performance for the First Job

Since HAIL piggybacks adaptive indexing on MapReduce jobs, 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. For this, we run job JobUV1 for UserVisits and job JobSyn1 for Synthetic. For these experiments, we assume that there is no block with a relevant 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

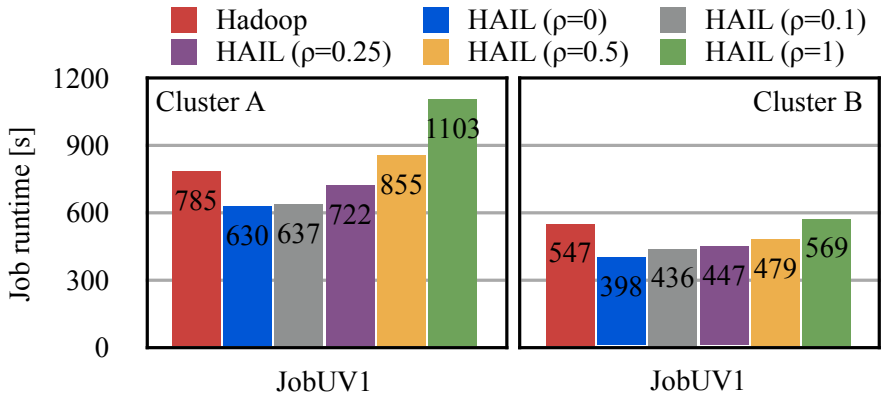


Figure 3.12: HAIL Performance when running the first MapReduce job over UserVisits.

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 $\sim$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'

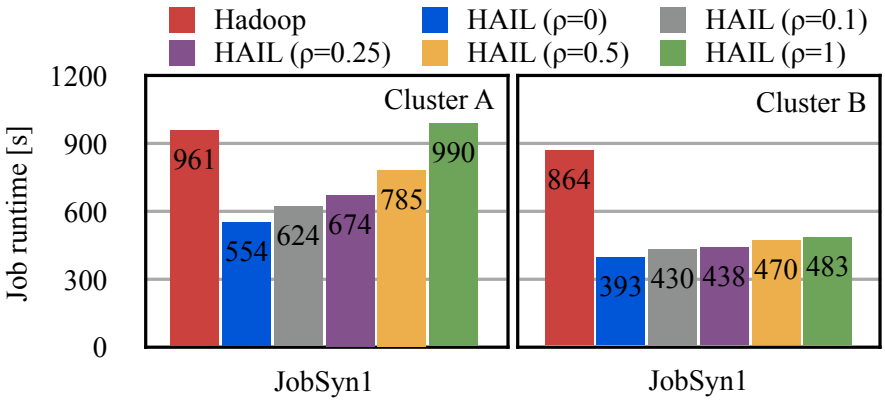


Figure 3.13: HAIL Performance when running the first MapReduce job over Synthetic.

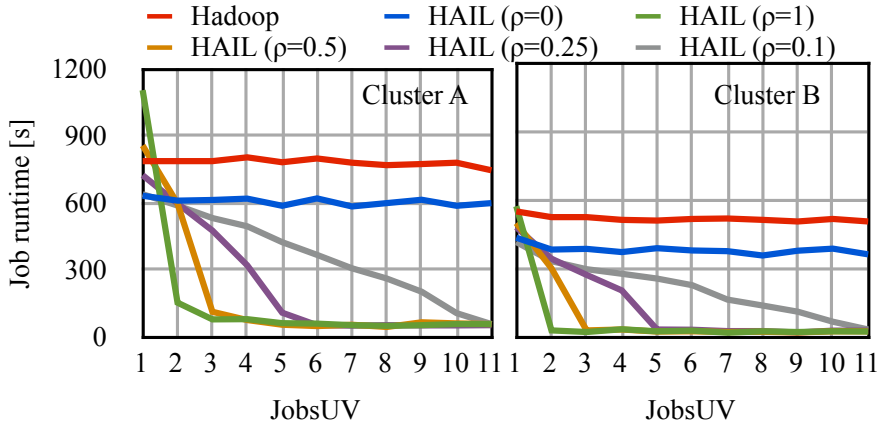


Figure 3.14: HAIL performance when running a sequence of MapReduce jobs over UserVisits.

requirements via different offer rates.

Performance for a Sequence of Jobs

We saw in the previous section that HAIL adaptive indexing techniques can scale linearly 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

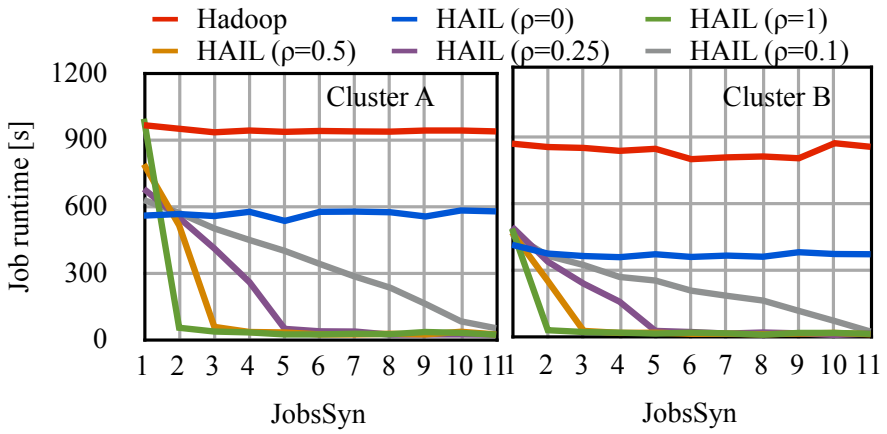


Figure 3.15: HAIL performance when running a sequence of MapReduce jobs over Synthetic.

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.

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.

[13]Although HAIL is still indexing further blocks.

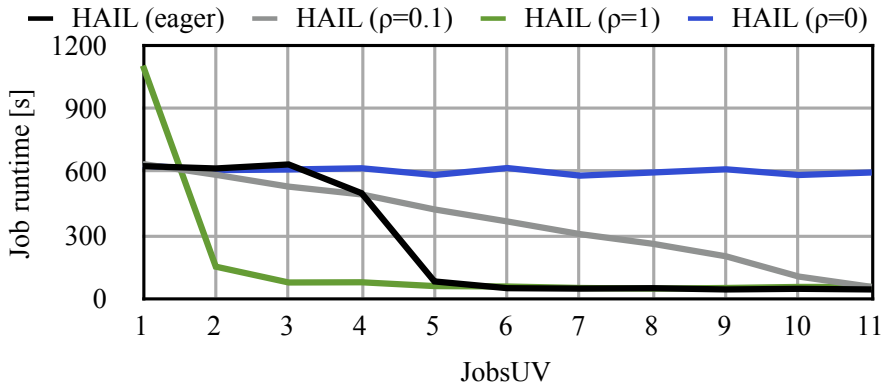


Figure 3.16: Eager adaptive indexing vs. $\rho = 0.1$ and $\rho = 1$

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 show 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.

3.10 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++ [28], 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. E.g. 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 a number of 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 their 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. $\sim$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

AIR: Adaptive Index Replacement in Hadoop

4.1 Introduction

Adaptive indexing has received quite some attention in the community and is a very interesting approach to provide reasonably good performance when facing ever changing or evolving workload patterns, without requiring human intervention. Several publications looked at adaptive indexing in main memory databases [38, 43, 47, 59]. In our recent studies [88, 11] we present an overview over the field and further explore directions in the adaptive indexing field in the context of main memory and multi core architectures. We also introduced an adaptive indexing algorithm into Hadoop MapReduce [80]. None of those works consider a space constraint on how many indexes can be created. Even though hard disk space can be considered cheap nowadays, we believe that it can be a limiting factor in the context of big data; it is therefore important to efficiently use the available resources. In the past years there has been extensive research on physical database design advisors [25, 96, 7] that include space constraints, but usually need a representative query sequence to provide meaningful advise on the physical design of the database. More recent work looked at online index tuning [83, 84, 19], an online approach to the Index Selection problem. That research does not consider an adaptive indexing setting, where indexes are created as byproducts of query execution.

In this chapter we investigate the Adaptive Index Replacement problem, i.e. our formulation of the Index Selection problem in the adaptive indexing scenario.

The chapter is structured as follows. In Section 4.2 we define the Adaptive Index Replacement (AIR) problem. Section 4.3 discusses related work in the field of adaptive indexing, as well as indexing in Hadoop MapReduce, and Buffer Re-

placement algorithms. We then in Section 4.4 describe our cost model and a Mixed Integer Linear Programming (MILP) formulation for the offline AIR problem. Section 4.5 introduces our proposed algorithm, the LeastExpectetBenefit-K, to solve the online AIR problem. Section 4.6 presents the evaluation of our algorithm using simulations and an experimental validation. Finally, Section 4.7 concludes this chapter.

4.2 Adaptive Index Replacement

In this section we introduce the notation needed throughout the chapter and define the Adaptive Index Replacement problem.

In a system that uses adaptive indexing we have the following situation: As queries that would benefit from indexes on certain attributes arrive, new indexes are created and stored. This leads to a growing space consumption for the queried datasets. However, if the total space, or the space per dataset, is limited, it is often not possible to simply create indexes on all queried attributes. As we reach the storage limit, an algorithm has to decide for every query if some older indexes should be dropped, or if no new index should be created.

When using adaptive indexing in Hadoop we create indexes not on the dataset as a whole but on a more fine granular level. This comes naturally as the Hadoop Distributed Filesystem (HDFS) stores the data in so called HDFS blocks. Those HDFS blocks are horizontal partitions of the stored dataset and we do not reorder records across those HDFS blocks in HAIL [80]. We instead use the replication mechanism of HDFS to create different clustered indexes on the block level in the different replicas of a given block. Therefore, we decide at the level of HDFS blocks, rather than at dataset files, to create, drop, or simply keep an index. An index in HAIL is just another replica of a logical HDFS block, that is sorted with respect to an attribute and stored together with a very small coarse-granular lookup directory in HDFS. This allows us to handle appends to the dataset efficiently as no old indexes need to be maintained. Let $\mathbb{B}$ be the set of all stored logical blocks and A the set of all attributes. We define a query q on Hadoop as follows:

Definition 1 *A query $q = (B, a, sel)$ is defined by a triple consisting of the set of accessed blocks $B \subseteq \mathbb{B}$, the attribute that might be used to perform an index access, $a \in A$, and a mapping of the selectivity per block $sel : B \to [0, 1]$.*

The cost of executing a query q highly depends on the current configuration c of the system, which is defined as:

Definition 2 *A configuration $c \subseteq \mathbb{B} \times A$ represents the set of available indexes in the system. An index on block b with respect to the attribute a is available iff*

$(b, a) \in c$. *Let* $\kappa \in \mathbb{N}$, *the configuration* c *is called* valid *w.r.t. the capacity* κ *if* $\kappa \geq |c|$. *The set of valid configuration is denoted as* $C_\kappa = \{\ c \mid \kappa \geq |c|\}$.

Observe, that we could also use a multi set definition of configurations, i.e. to allow for the replication of certain indexes in order to better balance the cluster load in the presence of multiple concurrent jobs. We are focusing on a a single user scenario and therefore use simple set semantics.

We denote the cost to execute a query q given the current configuration c as $QueryCost(q, c)$. The cost to create a new configuration c' is dependent not only on the current configuration c but also on the executed query q, as new indexes are created while executing q. We therefore denote this cost by $IndexCost(q, c, c')$. This is different to previous works [19, 83, 85], as those assume the cost to reorganize a configuration to be independent of the current query q. In Section 4.4 we will give precise definitions of these cost functions in the Hadoop scenario.

We now define the *offline* version of the AIR problem:

Definition 3 *Given a start configuration* c_0, *a query sequence* $Q = q_0, \ldots, q_{n-1}$, *and a capacity* $\kappa \in \mathbb{N}$, *find the sequence* $C = c_0, \ldots, c_{n-1}$, $c_t \in C_\kappa$, *that minimizes the total cost* $TC(Q, C)$, *that is defined as:*

$$TC(Q, C) = \sum_{t=0}^{n-1} QueryCost(q_t, c_t) + \sum_{t=0}^{n-2} IndexCost(q_t, c_t, c_{t+1}).$$

Observe that the optimal solution $OPT(c_0, Q, \kappa)$, that minimizes TC, provides a lower bound for the cost of the solution of any online algorithm for the AIR problem. The *online* version of the AIR problem can be defined as:

Definition 4 *For every* t, *given configurations* $c_0, \ldots, c_t$ *and the partial query sequence* $q_0, \ldots, q_t$, *and a capacity* $\kappa \in \mathbb{N}$, *provide* c_{t+1} *such that the total cost* $TC(Q, C)$ *becomes minimal.*

This definition implies that online algorithms could use the whole sequence of already seen queries to decide what index should be created or refined. Nevertheless real world solutions typically only take some of the previous queries and the current configuration into account. The reason for this is twofold: First to avoid storing all the information about previous queries, and second to adapt to a changing workload pattern.

Definition 5 *We call an AIR algorithm* k*-aware if its decisions are influenced by* k *queries.*

Database cracking algorithms, as well as HAIL, base their decision solely on the current query and the current configuration, they are both 1-aware. This is suboptimal if previous queries can provide hints on future queries.

In contrast to database cracking, an adaptive indexing technique for in memory databases, it is easy to create an index on any attribute in HAIL[1] independent from the currently queried attribute. We will therefore focus on adaptive indexing in Hadoop as proposed in HAIL [80].

4.3 Related Work

Adaptive indexing infrastructure. Our Hadoop Adaptive Indexing Library (HAIL) [29, 80] extends the Hadoop distributed filesystem (HDFS) and allows MapReduce jobs to access data blocks using clustered indexes. These clustered indexes are created per HDFS block, either upfront while uploading [29] data into HDFS or adaptively while executing [80] MapReduce jobs. HAIL creates indexes while uploading without requiring additional storage space by exploiting the existing replication of HDFS and storing every physical block replica in a different sort order. A sorted physical block replica together with a very small lookup directory is a coarse granular index in HAIL; whenever we talk about stored indexes, we refer to these data structures. HAIL still needs additional space for every adaptively created index. As HDFS is an append-only system, we cannot modify existing indexes and need to create new physical block replicas with new sort orders. The original works [80, 29] do not contain an algorithm to decide which indexes to keep, if the storage capacity is reached. This work fills the gap.
Index Selection algorithms. The Index Selection problem has been extensively studied in DBMSs. Commercial systems typically include physical design advisors that propose indexes and other physical structures, like materialized views, in order to speed up workload execution. These tools have to be invoked with a representative query workload to decide on a suitable set of indexes. As index creation is considered a rather costly operation, many physical design advisors usually use a one shot solution; they provide a suitable set of indexes that should be created before executing the workload.

Agrawal et. al. choose a different approach and modeled the query workload as a sequence [7] and showed how this model helps to decide if indexes should be created or dropped during the execution of the workload. That work still assumes that a representative workload is provided offline. In their model, indexes should be dropped if they incur high index maintenance costs and again created when many read queries are expected. In this work we also model our workload as a

[1]Database cracking can only piggyback on comparisons with the requested predicate. HAIL piggybacks on the data load of job execution and can create any index on the loaded block.

sequence, but since we have no updates to the indexed data in the scenarios we are looking at, we observe no index maintenance costs at all.

CoPhy [25] is an index advisor for large workloads. It uses a Binary Integer Programming formulation of the Index Selection problem. The used BIP formulation does not entail index creation costs since CoPhy follows an offline approach, that does not allow for changing the indexes at runtime. We use a Mixed Integer Linear Program (MILP) formulation to find the solution of the offline Adaptive Index Replacement problem that is similar to the proposed BIP formulation in [25], but our formulation allows for changing the index configuration at runtime. We use the MILP formulation to compute a bound for the best possible performance an online AIR algorithm could achieve, while the BIP is a core part of CoPhy.

Online index selection algorithms can be classified as either *retrospective* [19, 85] or *predictive* [66, 83]. Retrospective algorithms only create new indexes after it is clear that an index would have been beneficial. This means that only after enough queries have been observed, so that the total benefit of the index is higher than its creation cost, the index is created. In contrast, predictive algorithms try to predict the future workload and create indexes that will be beneficial for that predicted workload. Predictive online index selection algorithms can on the one hand predict something completely wrong and spend time in creating new indexes that are never used. On the other hand they can react much faster to workload changes and therefore reach higher performance gains than retrospective online index selection algorithms. Therefore, we can say that retrospective online index selection algorithms are more conservative than predictive algorithms. We will see that our approach can be classified as a predictive algorithm.

Nicolas Bruno and Surajit Chaudhuri introduced a retrospective online indexing approach [19]. Their approach keeps track of the accumulated benefit of every index I at any time t, denoted by $\Delta_t(I)$. As index maintenance can also incur costs, $\Delta_t(I)$ can become negative. To detect if an index has become beneficial, they introduce $\Delta_{t,min}(I)$ and $\Delta_{t,max}(I)$ as watermarks. This means that $\Delta_{t,min}(I)$ stores the smallest accumulated benefit since the index I was last dropped and $\Delta_{t,max}(I)$ stores the largest accumulated benefit since the index I was last created. Those watermarks are used to decide whether an index should have been created or dropped in the past. If $\Delta_t(I) - \Delta_{t,min}(I)$ is larger than the index creation cost, the index should have been created. If $\Delta_{t,max}(I) - \Delta_t(I)$ is larger than the index creation cost, the index should have been dropped. As we incur no index maintenance cost in our adaptive indexing scenario we notice that $\Delta_{t,max}(I) = \Delta_t(I)$. Therefore the algorithm would never drop an index because of index maintenance. When it comes to the point where the algorithm has to decide what index should be dropped due to space limitations, the algorithm relies on the size and observed index maintenance costs of the materialized indexes. Since both properties are

the same for all indexes in our scenario we need another criteria to break the tie between the materialized indexes. In order to use the proposed algorithm as a baseline we chose to use the benefit as tie-breaker and we will drop the index with the lowest benefit.

Another retrospective and more recent algorithm to solve the Online Index Tuning problem is the WFIT [85] algorithm introduced by Karl Schnaitter and Neoklis Polyzotis. The WFIT algorithm is very interesting as it can also incorporate feedback from the DBA in its optimization decisions. Unfortunately, WFIT does not contain space constraints, but lets the DBA decide which and how many indexes are materialized. We can therefore not compare our method against the WFIT algorithm, as the decision which index to drop is the single most interesting question in the Hadoop scenario.

Karl Schnaitter et. al. introduced a system called COLT [83] (Continues On-Line Tuning). This system analyzes the current workload and optimizes the physical design of the database continuously. The query sequence is analyzed in epochs, e.g. a batch of 10 queries, and the system optimizes after each epoch the total runtime, by choosing the best set of single column indexes that fit in a given space constraint. A very similar system was introduced by Martin Lühring et. al. [66]. We call that system SoftIndex system, as it creates so called soft indexes. Those soft indexes are materialized concurrently to query execution by using index build scans. In contrast to adaptive indexing approaches like database cracking or HAIL, COLT and the SoftIndex system use the information gathered in several complete epochs to decide which indexes to create and not just the current query. We reimplemented the SoftIndex algorithm and use it as a baseline.

Buffer Replacement algorithm. O'Neil et. al. introduced the LeastRecently Used-K (LRU-K) [70] Buffer Replacement (BR) algorithm as a generalization of the LRU algorithm. The LRU-K algorithm is an online algorithm to solve the BR problem. In the BR problem we have a buffer that can hold up to m pages. Whenever a page is requested that is not in the buffer, it has to be fetched into the buffer. If the buffer is full and a new page has to be fetched, we have to select a page from the buffer for eviction. The goal of a BR algorithm is to minimize the number of buffer misses, e.g., the number of pages fetched into the buffer. The LRU-K algorithm evicts the page with the highest LRU-K-age, where the LRU-K-age of a page is the age of (or time since) the K-th latest access to the page[2]. The rational behind this algorithm is that we can estimate the expected arrival interval between two accesses of a page given the LRU-K-age, and evict the page that is expected not to be accessed for the longest time. O'Neil et. al. have proven the LRU-K algorithm to be optimal with respect to the Independent Reference

[2]If a page has not been accessed K times yet the LRU-K-age is ∞. If more than one page has an LRU-K-age of ∞, LRU-(K-1)-age is used to break the tie and so on.

Model [71]. Here optimality was defined such that no other algorithm that bases its decision on the last K accesses per page, and that has $m-1$ buffer slots, is expected to achieve less buffer misses than LRU-K with m buffer slots. We will use the LRU-K-age together with query selectivities and a indexing threshold in our LEB-K algorithm to decide what index to replace or to create.

4.4 Cost Model

In this section we introduce our cost model and show how to obtain the optimal solution to the offline Adaptive Index Replacement problem. For simplicity we discuss the cost model as if only one block existed. We therefore only consider what attribute the query performs a selection on. The actual cost can then be computed as the sum of all individual costs per block that are accessed by the query. Our cost model considers the I/O-cost to read or write to disk and ignores CPU consumption to create the index. The results in our HAIL paper [80] showed that this is applicable for disk-bound operations, like simple analytics on large data sets.

We first focus on the *QueryCost*. The I/O cost to answer the current query depends on the arriving query q and the current configuration c. When a query q arrives and no suitable index on the query attribute $q.a$ is available in the current configuration c, i.e. no block replica has the right sort-order, we have to read the whole block and pay the $FullScanCost = 1$. If an index is available, we only read the data needed by q from disk and pay the $IndexScanCost(q.sel) = q.sel$. This cost model does not contain the cost to access the index or perform the index lookup but focuses on the cost to sequentially read the needed data from disk. We can make this simplification in the HAIL scenario, as the index itself is coarse granular and therefore really small, in the size of only a few kilobyte, when using an HDFS block-size of 256 MB. The cost model also omits the cost to write out the result of the query, as this cost is independent of the current configuration. We now define the *QueryCost* as follows:

$$QueryCost(q, c) = \begin{cases} 1 & q.a \notin c, \\ q.sel & q.a \in c. \end{cases}$$

As already explained in Section 4.2, the index creation cost $IndexCost$ depends on the current query q as well as the current configuration c and the next configuration c'. In our scenario the cost to create an index are the additional disk-I/O costs to write the new block replicas. This is reasonable in HAIL, as we overlap the in-memory sorting with the reading of the next block. Only the additional write costs are observable in the end-to-end runtime. We therefore have to pay the cost to write a block for every index, that was created, denoted by $|c' \setminus c|$. If

a full scan was performed, the cost to read a block from disk is already accounted for by the *QueryCost*. In case the query could perform an index scan and we still decide to create a new index, we have to read the whole data from disk instead. In our cost model we achieve this by adding the cost of reading the rest of the data, that was not accounted for in the *QueryCost*. This leads to the following indexing cost:

$$IndexCost(q, c, c') = |c' \setminus c| + \begin{cases} 1 - q.sel & q.a \in c \wedge c' \not\subseteq c, \\ 0 & q.a \notin c \vee c' \subseteq c. \end{cases}$$

As a side note, we observed that many systems, including HAIL, perform an index scan whenever a suitable clustered index is available. The following example query sequence shows that this is not necessarily optimal. Assume three different query types q_1, q_2, and q_3, accessing attributes $q_1.a = 1$, $q_2.a = 2$, and $q_3.a = 3$ respectively, with selectivities $q_1.sel = 0.5$, $q_2.sel = 0.1$, and $q_3.sel = 0.1$ and the following query sequence:

$$Q = q_1, q_2, q_1, q_2, q_1, q_3, q_3, q_1, q_1.$$

Assume further that up to two indexes can be kept on the system. For the first five queries it is optimal to quickly build indexes on attributes 1 and 2, while for the last four it is desirable to have indexes on attributes 1 and 3. An optimal algorithm creates the following configuration sequence:

$$C_{OPT} = \emptyset, \{1,2\}, \{1,2\}, \{1,2\}, \{1\}, \{1,3\}, \{1,3\}, \{1\}, \{1\}.$$

This configuration sequence has the following total cost:

$$\begin{aligned} TC(Q, C_{OPT}) = 1 + 2 + 0.1 + 0.5 + 0.1 + 1 \\ + 1 + 0.1 + 0.1 + 0.5 + 0.5 = 6.9 \end{aligned}$$

A pseudo-optimal algorithm that has to use an index, if it is available, would produce the following sequence:

$$C_{P-OPT} = \emptyset, \{1,2\}, \{1,2\}, \{1,2\}, \{1\}, \{1\}, \{1\}, \{1\}, \{1\}.$$

with the following total costs:

$$\begin{aligned} TC(Q, C_{P-OPT}) = 1 + 2 + 0.1 + 0.5 + 0.1 + 0.5 \\ + 1 + 1 + 0.5 + 0.5 = 7.2 \end{aligned}$$

Even though most data management systems, including HAIL, would use the existing index on attribute 1 for the fifth query $Q_5 = q_1$, it is better to fully scan

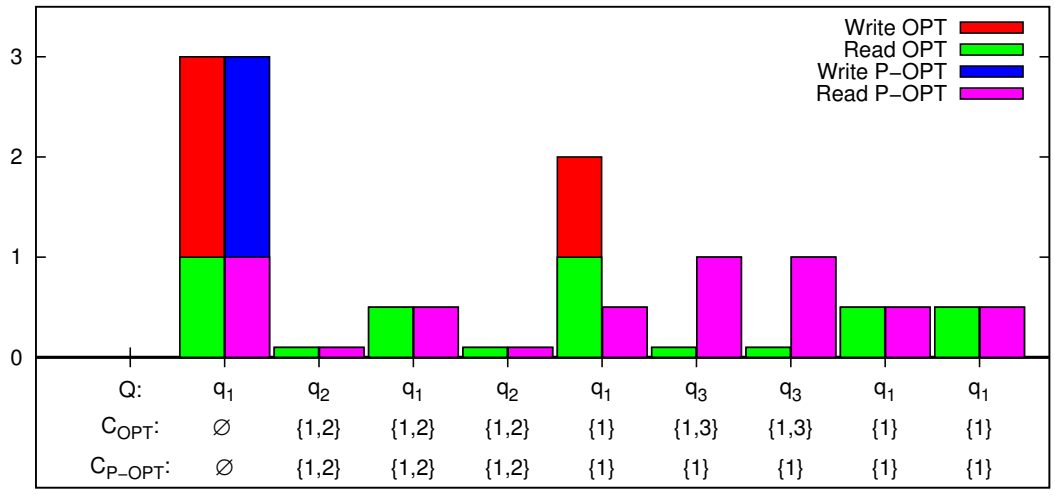


Figure 4.1: Visualization of the cost to execute the example query sequence Q.

the block and create an index on attribute 3 for the upcoming queries. Figure 4.1 visualizes the cost to execute the query sequence Q. Instead of not using the index on attribute 1 for Q_5, one could either perform a full scan on $Q_4 = q_3$ or $Q_6 = q_2$ but both would incur even higher costs.

Given the cost functions from above we can now formulate the offline Adaptive Index Replacement problem as a *Mixed Integer Linear Program* (MILP), depicted in Listing 1. With this MILP formulation we can use an off-the-shelf solver to compute the optimal offline solution. We will use this optimal solution as a baseline in the experimental evaluation.

4.5 LeastExpectedBenefit Algorithms

In this section we introduce our *LeastExpectedBenefit* (LEB) algorithms to solve the Adaptive Index Replacement problem, with respect to the provided cost model. The core idea of the different LEB algorithms is to replace the index with the lowest expected benefit. This expected benefit of an index can be estimated using different statistics. When using a statistic that only takes the last K queries for every attribute into account we talk about the LEB-K algorithm and if we take all queries into account we call it the LEB-∞ algorithm.

Algorithm 2 shows the general LEB algorithm. Depending on the `statistics` field the algorithm actually implements LEB-K, LEB-∞ or even LRU-K. In the following we discuss how we can estimate the expected benefit of an index, based on the last K queries for every block-attribute pair. We can use

We use the following variables:
- $c_{t,i} \in \{0,1\}$: Index i is available in the configuration c_t.
- $w_{t,i} \in [\,0,1\,]$: Write cost to create index i at access t.
- $r_t \in [\,0,1\,]$: Read cost to read data at access t.

Our objective function is the total write and read cost.

$$minimize \sum_{i=1}^{|A|} \sum_{t=0}^{n-1} w_{t,i} + \sum_{t=0}^{n-1} r_t$$

subject to the following constraints:
At most κ indexes are available at any time.

$$\forall t : \sum_{i=1}^{|A|} c_{t,i} \leq \kappa$$

If no index is available, we have to pay the full scan cost.

$$\forall t : r_t + c_{t,q_t.a} \geq 1$$

If an index is available at time t that was not available at time $t-1$, we have to pay the creation cost.

$$\forall t > 1, i : c_{t-1,i} + w_{t-1,i} \geq c_{t,i}$$

We can piggyback index creation only on full scans.

$$\forall t, i : r_t \geq w_{t,i}$$

The scan cost is bound by the selectivity of the query and the full scan cost.

$$\forall t : q_t.sel \leq r_t \leq 1.$$

Listing 1: The MILP formulation to find the optimal sequence of index configurations.

the LRU-K-age to estimate the arrival rate of queries accessing the block-attribute pairs by $\frac{K}{\text{LRU-}K\text{-age}(b,a)}$. These arrival rates in turn can be used to estimate the probability for the next query to hit a given block-attribute pair $P(b,a)$, by normalizing the sum of all arrival rates to one. We define the benefit of an index that was created on a given block, with respect to a certain index attribute, as the amount of I/O-cost that was avoided thanks to the index compared to the full scan costs. If for example a query q accesses block $b42$ selecting on attribute $q.a = 1$ with a selectivity of $q.sel(b42) = 0.1$, then the benefit of an index on $b42$ with respect to attribute 1 would be $\texttt{Benefit}(b42, 1) = 1 - 0.1 = 0.9$. From the last K queries to every block-attribute pair we compute the average benefit and use that to compute the expected benefit of the index on that block, by multiplying the average benefit

Algorithm 2: General LEB Algorithm

```
Statistics statistics;
double threshold;
public Conf getUpdatedConf(Query q, Conf oldConf) {
 statistics.addNewQuery(q);
 Conf newConf= new Conf(oldConf);
 Index b = q.getSuitableIndex();
 if (!newConf.contains(b)) {
  if (newConf.size() >= capacity) {
   Index leastBene =
      statistics.findLeastBeneficialIndex(oldConf);
   if (shouldReplace(b, leastBene)) {
    newConf.replaceIndex(leastBene,b);
   }
  } else
   newConf.addIndex(b);
 }
 return newConf;
}
public boolean shouldReplace(Index newIndex,
   Index existingIndex) {
 double benefit;
 benefit = benefit(newIndex)-benefit(existingIndex);
 return benefit > threshold;
}
```

with the arrival probability of a query hitting that block-attribute combination.

$$E(\texttt{Benefit}(b,a)) = avg(\texttt{Benefit}(b,a)) \cdot P(b,a)$$

Finally we use the expected benefit of the index with the lowest expected benefit and the expected benefit of the possible new index to decide if the new index should be created and the least beneficial index should be dropped.

To incorporate the index creation cost we only create a new index, if its expected benefit is higher than the expected benefit of an existing index by a certain threshold. If we choose a very low threshold, we will create an index early in the hope that the query distribution does not change for a long time. If we choose for example a threshold of 0.2, we will only create an index, if we expect the index to have amortized not only its creation cost but also the expected benefit of the dropped index in five queries.

We base our family of LEB algorithms on the family of LRU algorithms, because the Buffer Replacement (BR) problem is very similar to the AIR problem. Nevertheless, there are some important differences that we will shortly discuss. First we will see that it is rather straightforward to take any BR algorithm and create an AIR algorithm out of it. As already explained we access pages in the BR problem, whereas we access indexes in the AIR problem. An index can be identified by the attribute together with the block that is sorted with respect to the attribute. Therefore, we can simply use the block-attribute pairs of every query

analogous to buffer page identifiers and ignore the provided selectivity. If the BR algorithm is optimal, the resulting expected number of "index misses" is minimal.

In the following we will discuss the differences between the BR problem and the AIR problem. Not all indexes are equal, some are hit by queries with high selectivity some are not. We therefore want to incorporate the benefit of an index in the decision which index to keep. Another important point that needs to be considered are index creation costs. In contrast to the Buffer Replacement scenario, where pages have to be buffered as soon as they are accessed, we do not have to create an index to access the data. An AIR algorithm therefore has to decide if a new index should be created at all. Some BR algorithms can be relaxed, such that they not always create the missing index when used as an AIR algorithm. For LRU-K this means, that we can choose to create the index based on its LRU-K-age and only if the LRU-K-age of the missing index is lower than an existing index, we create a new index. Based on the LRU-K algorithm we introduce the *LeastExpectedBenefit-K* (LEB-K) algorithm. We choose the LRU-K as the base algorithm, because it is very easy to steer the sensitivity of the algorithm to workload changes by choosing different values of K and also because it is a commonly used and efficient BR algorithm.

4.6 Evaluation

To evaluate our LEB algorithms we simulate the read and write costs for different skewed workloads and experimentally validate our cost model on our cluster. This Section is structured as follows: In Section 4.6.1 we describe the general setup and the used workload patterns. Then in Section 4.6.2 we describe the algorithms we want to evaluate. In Section 4.6.3 we discuss our findings with respect to the overall performance of the different algorithms, while we discuss our robustness experiments in Section 4.6.4. Finally Section 4.6.5 shows our experimental validation of our cost model.

4.6.1 Dataset and Query Distribution

The query pattern in our experiments is called **Evolving Pattern** and changes the focus of the queries after a fixed period of queries. It consists of 1000 queries and every query is drawn from a pool of 20 queries with a Zipfian distribution (i.e the first query is most likely to be drawn). We simulate a shift of focus, by randomly permuting the set of available queries after every hundredth query (i.e. if query 0 was drawn most often, afterwards maybe query 9 is drawn most often). Additionally, we also evaluated all algorithms using the **Stable Pattern**, which does not permute the the queries. Both patterns are visualized in Figure 4.2.

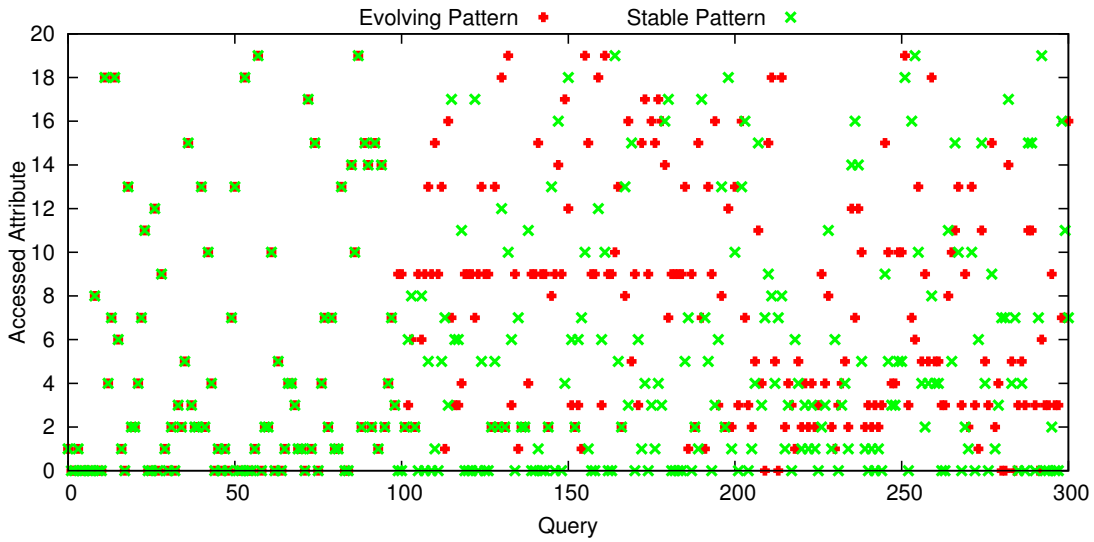


Figure 4.2: Attribute distribution of the first 300 queries.

We vary the distribution of the selectivities of the different queries to measure the effect on the performance of the algorithms. The first distribution is called **Power Distribution**. It is a skewed distribution and follows a power law. To be precise, the queries on attribute i have a selectivity of $0.5^i \cdot 0.99$. For the **Uniform Distribution** we assign every attribute a selectivity chosen uniformly at random between 0 and 0.5. We use the same selectivity for every query on a certain attribute. Figure 4.3 visualizes the different selectivity distributions.

4.6.2 Evaluated Algorithms

To obtain the optimal solution for the Adaptive Index Replacement problem we implemented OPT by solving the MILP representation from Listing 1 with IBM's ILOG CPLEX [1] solver. We use this lower bound on the cost to measure the effectiveness of all tested online AIR algorithms.

We implemented the following ten algorithms and compared their performance with respect to our cost model.

(1.) BC is our re-implementation of the algorithm presented by Bruno and Chaudhuri in [19]. We extended the algorithm to use the accumulated benefit as tie-breaker when deciding what index to drop.

(2.) HAIL uses a very simple algorithm that can be used by our system without requiring any algorithmic decision. HAIL simply creates indexes until the storage capacity is reached and does not throw away any of the created indexes.

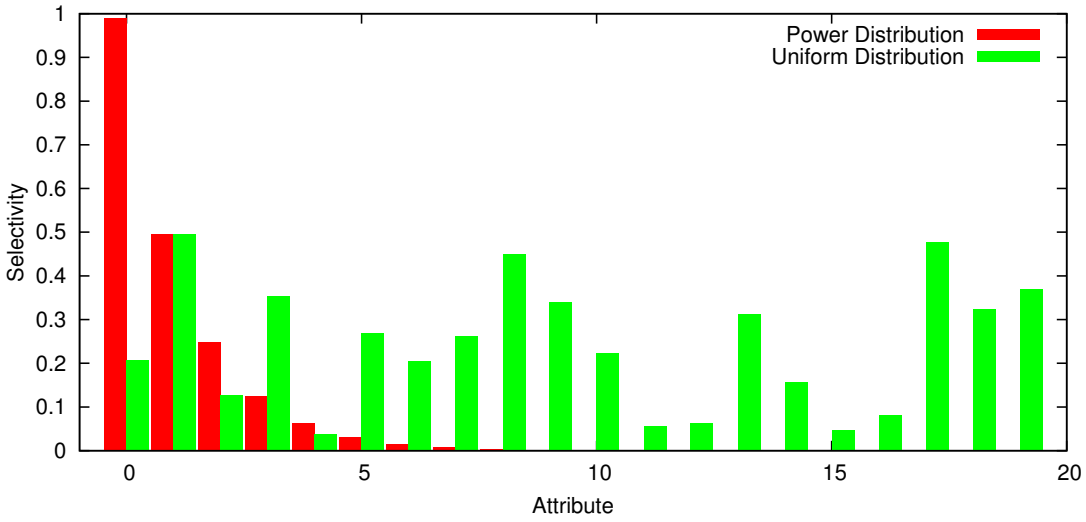


Figure 4.3: Selectivity distributions over the 20 attributes.

(3.) The **LRU-1** algorithm creates a new index for every incoming queries, that does not hit an index. If the capacity is reached, it will drop the index that was least recently used.

(4.) The **LRU-2** algorithm initially creates an index whenever a new attribute is queried. After the capacity is reached, it will keep the indexes that were accessed twice most recently. This means that new indexes are not always created, but only if the LRU-2-Age of the current query is smaller than the LRU-2-Age of one of the existing indexes.

(5.) The **LEB-∞** algorithm estimates the expected benefit of all indexes by counting for each attribute how often it was accessed and summing up the selectivities of all queries that selected on the attribute. The algorithm then keeps the most beneficial indexes. This algorithm corresponds to a benefit aware Least Frequently Used (LFU) algorithm.

(6.) The **LEB-2** algorithm is our implementation of the algorithm described in Section 4.5, we use a threshold of 0.2. With this threshold the algorithm will only create indexes if the index amortizes its creation cost in at most five queries.

(7.) The **Random** algorithm starts by creating an index whenever a new attribute is queried. This continues until the capacity is reached. Afterwards the algorithm decides uniformly at random which index to drop or if an index for the new attribute should be created in the first place.

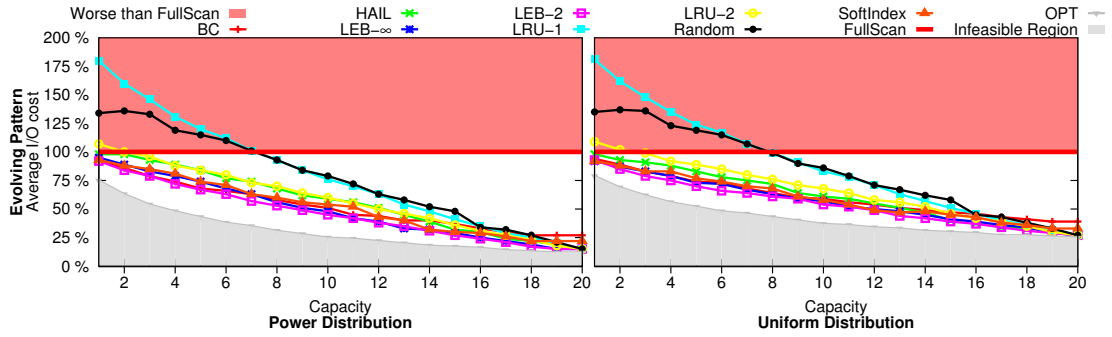
(8.) **SoftIndex** is our re-implementation of soft index management algorithm presented in [66] with a threshold of 1.8. The threshold was experimentally determined

to give the best overall performance.

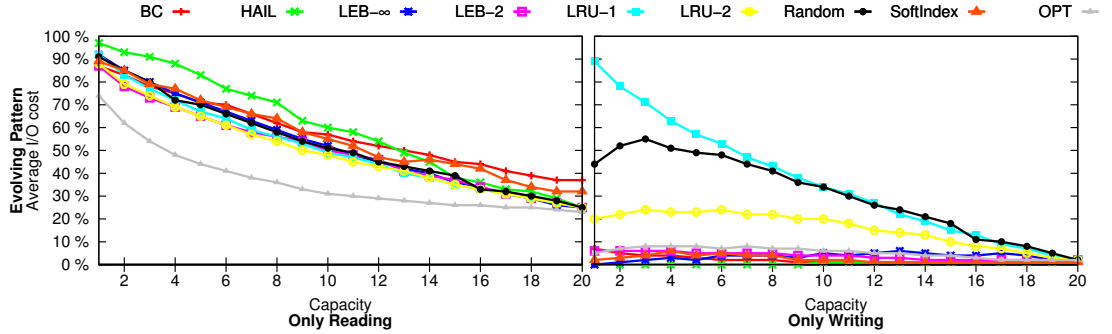
(9.) The **FullScan** algorithm does not create any index and instead always reads all blocks fully. No actual algorithm should perform worse than FullScan.

(10.) **OPT** is a theoretical lower bound of the total cost, obtained from the solution of the previously presented MILP. Recall, that this algorithm in contrast to all other algorithms, *knows the entire workload in advance*.

4.6.3 Performance Results



(a) Average read and write I/O cost per block after executing the whole workload



(b) Cost breakdown into the average read I/O and write I/O cost per block after executing the whole workload using the Uniform distribution

Figure 4.4: Simulated I/O cost for all presented AIR strategies with varying capacities.

Figure 4.4(a) shows the total cost for both the Uniform Distribution as well as the Power Distribution. On the y-axis we see the average I/O cost per data block in percent. This means that if every block is always read fully from disk, the average I/O cost is 100 %. If every block is read fully and written out again this corresponds to the maximum of 200 %. The x-axis shows how many full copies of the dataset can be stored. Since the dataset has 20 attributes, it is clear that we

can create an index for every attribute, if we can store 20 copies of the data. The interesting area starts if the available space is greater or equal to three, as Hadoop creates already three replicas by default.

The first thing to note is that both LRU algorithms perform rather poorly. LRU-2 performs much better than LRU-1 but it fails to beat the other AIR algorithms. The BC algorithm performs rather well if the available space is scarce, but it does not reach the same level as the other algorithms when we can store almost all indexes. This is caused by the way BC tries to prevent oscillating index creation. Whenever an existing index is used by an incoming query, BC lowers the accumulated benefit of all indexes that are not yet created. This leads to the case that several indexes are never created, even though the space would allow for creating more indexes.

We see that LEB-2 is slightly better than the LEB-∞ algorithm. But why does the LEB-2 algorithm only slightly improve over the LEB-∞ algorithm and does not outperform it clearly, even though the access pattern changes drastically after every hundredth query? To explain this result we look at the breakdown of the I/O cost depicted in Figure 4.4(b). Here we see, that the average read cost per query of all algorithms is rather similar, with the LEB-2 and LRU-2 being a little better for the Evolving Pattern. What we also notice is that the LRU algorithms are very competitive in terms of read performance. Since the LRU algorithms are very good at adapting to new query distributions, they manage to provide useful indexes most of the time. This adaptivity comes at a cost as we see when looking at the average write cost per query. This makes BR algorithms unsuitable for the AIR problem. LRU-1 always has to create an index for every "index miss", as this attribute has the smallest LRU-1-age. LRU-2 performs a little better with respect to write performance, as it does not always create a new index for every "index miss".

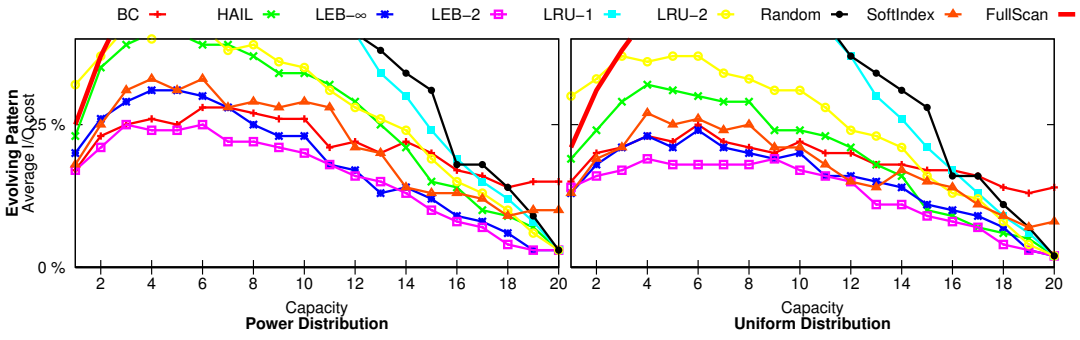


Figure 4.5: Additional I/O cost compared to OPT in absolut numbers (% of block read and written additionally)

We notice that also the LEB-2 has higher write costs compared to the LEB-

∞ algorithm. LEB-2 adapts faster to the evolving workload as LEB-∞, but since adapting means to change the set of available sort-orders more often, it has to write more data. In contrast, LEB-∞ adapts only slowly to a new query distribution and changes only slowly the set of available indexes. The additional write cost to adapt faster diminishes the gain of the better indexes that are available with the LEB-2 algorithm.

Figure 4.5 provides a zoom-in on the I/O overhead compared to the OPT algorithm. Here the absolute overhead compared to OPT is depicted, i.e, the average I/O an algorithm has to pay additionally for every query compared to the OPT strategy. We see that the LEB-2 algorithm is our favorite for the evolving workloads and both selectivity distributions.

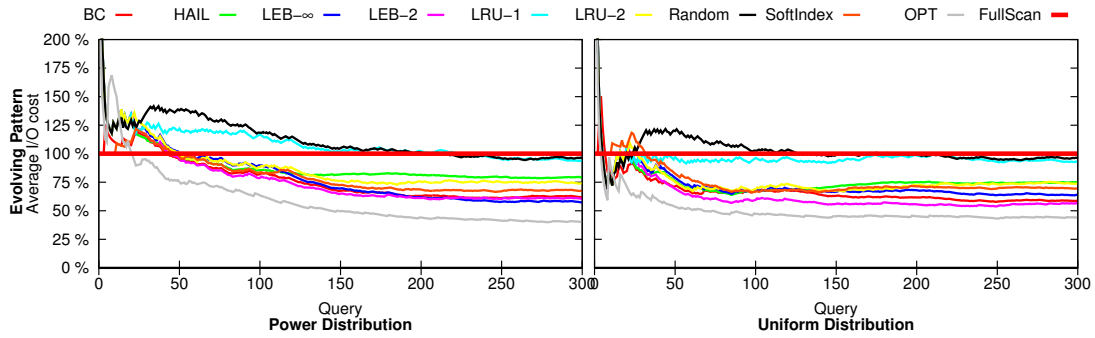


Figure 4.6: Running-average I/O cost per query for the different patterns and distributions

Until here we only looked at the overall performance of the different algorithms. Now we want to look at the development of the performance while executing the workload. Figure 4.6 depicts the running average I/O cost for all implemented algorithms, query patterns, and selectivity distributions with a capacity constraint of eight indexes. We see that the conservative algorithms benefit in the beginning, as they do not incur the initial indexing costs. Please note that our HAIL implementation can limit the initial spike by setting a so called offer rate, that allows only for a certain percentage of the blocks to be indexed.

In the next figure we will visualize the evolving indexes for four of the presented AIR strategies, namely OPT, BC, LRU-2, and LEB-2. Figure 4.7 depicts the accumulated benefit, or overhead respectively, for the different attributes. The black boxes depict the existing indexes and the small crosses depict the accessed attributes. The green color depicts the accumulated benefit over full scan, if we hit the index, while the red color depicts the accumulated overhead over an index scan, if we miss the index. We see that the OPT and LRU-2 strategies often change the set of available indexes. In contrast BC and LEB-2 keep their indexes for a relatively long period of time. We also notice that the green in the OPT strategy is

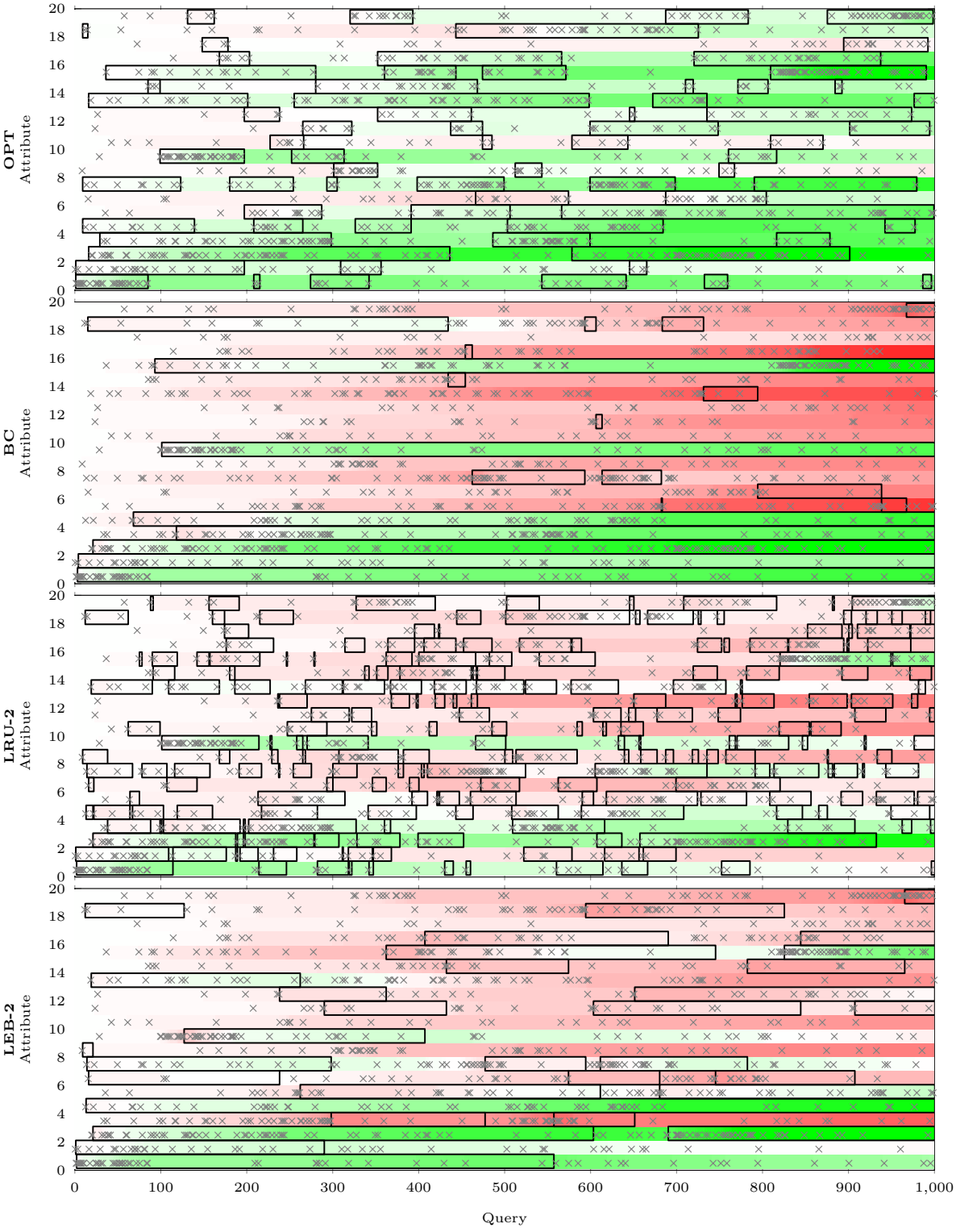


Figure 4.7: Visualization of AIR strategies over an evolving workload, uniform distribution, capacity of 8

not necessarily darker than the green in the other strategies, but there are almost no traces of red visible. Keep in mind that OPT is only shown for reference — it is the only algorithm knowing the future. BC accumulates high benefits on a few attributes, however, it keeps most of the indexes over long periods of time, and only one of the indexes is frequently adapted to the workload. In contrast, LEB-2 adapts all of its indexes more frequently and achieves overall a better performance.

4.6.4 Robustness Results

In this Section we investigate the robustness of the different AIR algorithms. We

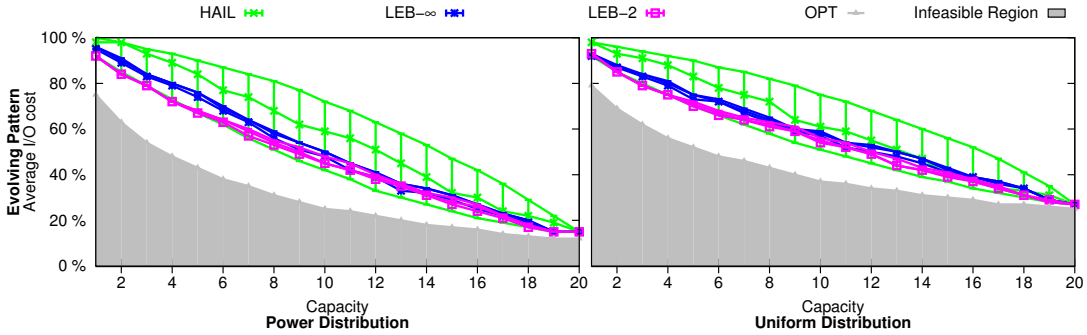


Figure 4.8: Robustness: Average I/O cost on variations of the query sequence

noticed that the simple HAIL algorithm is performing rather competitive for the randomly chosen query sequences. To show that the simple algorithm is highly dependent on the very first queries, we choose to compute the I/O cost for two additional query sequences. In both additional sequences we prepend twenty queries on the twenty attributes to the previously described random query sequence. These twenty queries are ordered with respect to the total benefit an index on that attribute would have for the whole query sequence. One sequence starts with the twenty queries in ascending benefit order, while the other sequence starts with the twenty queries in descending benefit order. Figure 4.8 (a) shows the result for the HAIL algorithm as well as both LEB algorithms. We see that the performance of the simple algorithm has a very high variance while both LEB algorithms are not influenced as much by the first queries. Please notice that none of the described query sequences are yet the worst case sequence for the HAIL algorithm. An adversary for the HAIL algorithm would first access a few attributes and force HAIL to create indexes. As soon as no more indexes can be build, the initially accessed attributes are no longer accessed. This makes the initially created indexes useless for all following queries.

We can also learn from Figure 4.8 (a) that evolving workloads need evolving indexes. The lower green line shows the result of creating the best indexes in the

beginning. We see that for the evolving pattern the gap between that lower green line and the performance of OPT is still rather big.

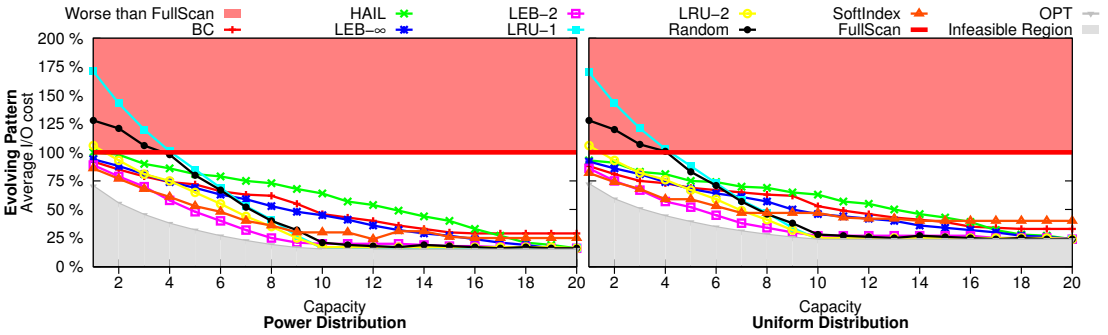


Figure 4.9: Robustness: Average I/O cost per query when the queries abruptly shift their focus after 500 queries

Why do we not use worst case sequences when analyzing the performance of the Index Replacement algorithms? It is textbook knowledge that all deterministic Buffer Replacement algorithms have the same number of page misses in the worst case. Similarly, if we cannot create indexes for all attributes, an adversary can always force our algorithm to fully scan all blocks. If we can create some indexes for all blocks, the perfect algorithm against an adversary is to balance the benefit for all attributes. This means that we do not adapt to any query the adversary provides, but create whatever index is necessary to balance the benefits between all attributes. In the case that we cannot provide a benefit to all attributes at the same time we should not create any index and always rely on full scan.

To further investigate the robustness of our LEB algorithms we perform another set of simulations with a different workload pattern. In this new patterns we divide the workload into two halves, in the fist half we only query attributes 0 to 9 and in the second half we only query attributes 10 to 19. Inside both halves we again use the evolving pattern. In Figure 4.9 (b) we see that OPT as well as LRU-1, 2, Random, and LEB-2 reach the optimal performance as soon as there is enough space to store ten attributes. The BC algorithm does not reach the optimal configuration of storing all indexes even if a capacity of 20 is given. That problem again stems from the anti oscillation measures, the accesses on the first attributes create a rather negative benefit for the remaining attributes and it takes a while until the other attributes are considered for indexing. A similar effect can be observed for the LEB-∞ algorithm, as the expected benefit only shifts slowly toward the attributes that are requested in the second half.

4.6.5 Experimental Results

Finally, to validate our findings we perform an experiment on our cluster and measure the actual runtime of a query sequence using different replacement algorithms.
Hardware: Our cluster consists of nine nodes, each running the 64-bit version of openSuSE 12.2 as OS. Each node has an eight-core Xeon E5-2407 with 2.2 GHz, 47 GB of RAM, 2x2 TB SATA HD, and one gigabit Network adapter.
Dataset: We generated a synthetic dataset on each node consisting of 10 GB of data, so 90 GB in total. The generated dataset allows us to easily generate a workload with the selectivity distribution described in the beginning of Section 4.6. E.g., if the selectivity on an attribute should be 0.5 we generate only the values 0 and 1 for that attribute and distribute them uniformly at random over all tuples.
Workload: We use the first hundred queries of the presented query pattern as our workload.

We upload the dataset to our Hadoop cluster using a replication factor of one and do not create any indexes while uploading. We allow for up to eight adaptively created indexes to exist in the cluster.

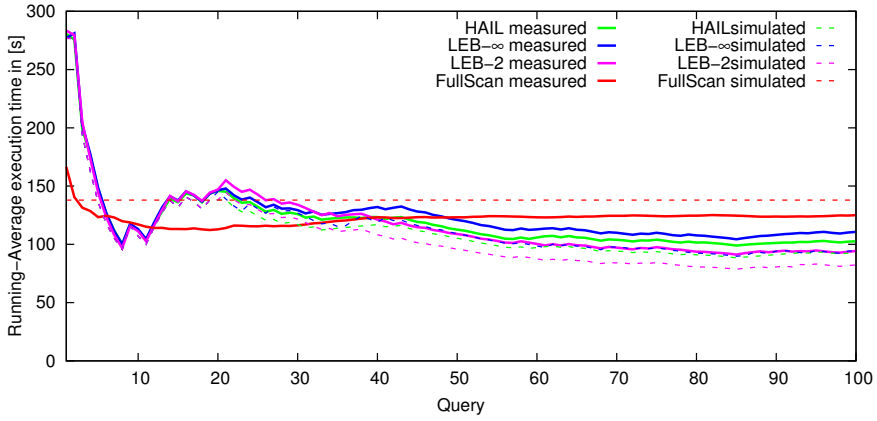


Figure 4.10: Running average of the job runtime for all hundred queries

Figure 4.10 shows the running average of the end-to-end runtime for the whole query sequence. The figure also shows the estimated runtime next to the actual measured runtime. To estimate the runtime based on our cost model we first multiplied the computed I/O cost with the per node dataset size. This yields the average I/O cost per node. We measured the HDFS transfer rate using the TestDFSIO tool that is included in Hadoop; the result is a transfer rate of 74 MB/s. The actually measured runtime nicely fits the predicted runtime. This shows that the I/O costs are indeed a good indicator for the actual performance and that our simulations correspond to the actual performance on our cluster.

4.7 Conclusion and Future Work

We introduced the AIR problem and provided formal definition of the optimal solution to the problem. We have seen that BR algorithms can be used to tackle the AIR problem. However, without any modifications such algorithms work rather poorly, as they do not incorporate the selectivity of the queries and ignore the index creation cost. We discussed and implemented an algorithm based on the LRU-K algorithm and saw in the evaluation that our LEB-K algorithm provides strictly better results than the LRU-K algorithm. We also learned that our algorithms perform as well, if not better, than both presented online Index Selection algorithms from the literature. We also uncovered some flaws in the BC algorithm that can make it unsuitable for the adaptive indexing scenario in Hadoop. The LEB-2 algorithm provided better performance than all other algorithms. We have seen that the I/O based cost model correctly predicts the actual runtime on our cluster. In future work we want to investigate the AIR problem in the context of main memory adaptive indexing. We are also further investigating how to adapt the threshold for the LEB-K algorithm to the current workload.

Chapter 5

Smart Caches (not only) for Analytical Workloads

In this chapter we will introduce a new hardware component that will allow us to perform scan intensive, in contrast to compute intensive, queries faster. Such queries are common in online analytical processing, especially if the analytical queries are performed on the live data instead of predefined aggregates and materialized views. This new hardware component was developed in the context of a joint project with the IBM Germany Research & Development laboratory in Böblingen. In this project we investigate how to bring modern data-analytical workloads to the IBM System Z mainframe machines.

5.1 IBM System Z Mainframe

The IBM System Z Mainframe, zEnterprise196 at the time of the project, is a very potent machine with up to 96 cores running with a clock frequency of 5.2 GHz and a total main-memory capacity of 3 TB. It is noteworthy that even though these processors provide a very high clock rate, several features that are present in modern server CPUs, e.g. from Intel, were not present in these mainframe cores at the time of the project. For instance, out-of-order execution was just introduced into the cores present in the zEnterprise196 mainframe and Simultaneous Multithreading (SMT) or Same Instruction Multiple Data (SIMD) instructions were only recently introduced with the z13 architecture (2015).

Nevertheless, many organizations and companies use the System Z mainframe architecture from IBM as a highly reliable and secure data store. Typically, all business critical data is stored on the mainframe and, for some companies, has to solely reside on the mainframe, for security reasons. Many of those companies store and process their business critical transactions in databases that are tuned

for online transactional processing (OLTP). Those database systems can handle insert or update-intensive short running transactions very well. Such workloads are therefore well supported and often executed on IBM System Z mainframe machines.

However, for decision makers it is also important to analyze the business data. To protect the high throughput of the OLTP system, the state of the art solution is to bulk transfer the data from the OLTP system to a so called data warehouse [90, 63]. This is done to keep long running analytical queries from degrading the high throughput of the OLTP system. On the downside companies have to perform these bulk transfers, maintain a second system, and work on stale data from the time of the last bulk transfer. This also forces the data to be available in an environment that is potentially less secure than the mainframe architecture. Today, it is often required to have real time analytics on the live data. To enable such real time analytics, in-memory databases are becoming more and more popular. For instance ,HyPerDB [58] is an in-memory database that allows for OLTP and online analytical processing (OLAP) on the same data. Such databases are developed for modern server CPUs but not necessarily for the mainframe. We want to develop new hardware components for the mainframe that make it more attractive for analytical processing on in-memory databases.

5.2 Motivation for Smart Caches

Todays memory architectures have several levels of caches to mitigate the widening gap between memory access times and CPU cycle times. The difference in size and speed between the different cache levels already reached several orders of magnitude and led to the introduction of more and more cache levels. System Z has four cache levels, where the first two are core local while the third and forth are shared between four cores. Prefetching and sophisticated cache replacement strategies are used to utilize the available cache space to the fullest. In scan once patterns, that are common for analytical queries, caches are not helping at all. An important observation is that all data needs to be brought into CPU registers to perform even the simplest computation; This also often involves copying the data into all cache levels. The question now is: How can we perform scan intensive operations, that perform almost no computation, more efficiently? And how can we make use of the otherwise useless caches in such queries?

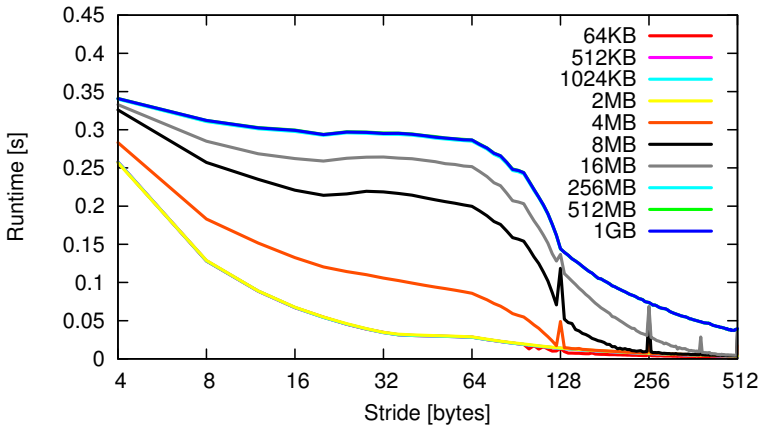
5.3 Computation on Cache Lines

Even though memory is byte-addressable, we cannot simply load a single byte into a register from memory but instead we have to first load the whole cache line into the L1 cache. In the Z architecture a cache line consists of 256 bytes. An interesting approach to speed-up scan intensive queries is to perform computations on the whole cache line at once. Such computations can be vector operations between two cache lines, or aggregation functions, like computing the maximum or the sum of all elements in a single cache line.

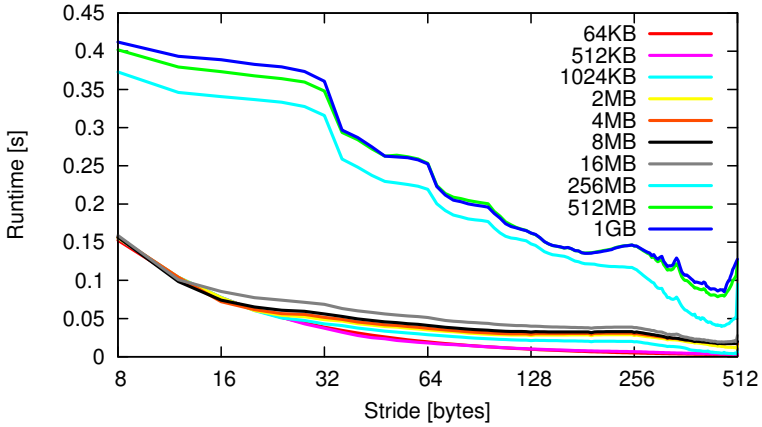
We performed an experiment to measure the possible gains of such an approach. For the experiments we assumed that we want to sum up all longs of a large array. However, instead of summing up all longs we gradually introduce larger and larger strides, i.e. we only sum up every i-th long with increasing i. With this we simulate that summing up all longs in a whole cache line boils down to accessing and summing up a single or very few longs in the cache line. We performed this experiment on an Intel CPU as well as the Z and observed some interesting differences.

Figure 5.1(a) shows the observed behavior on Intel. The different colors represent different sizes of the data array. Smaller arrays are summed up several times, such that the total number of elements brought into CPU is the same for each array. This normalizes the runtime for different sized arrays and we can compare the runtimes. When looking at large array sizes every accessed cache line needs to be brought from memory into the L1 cache. In this case we can observe that it has almost no impact on the runtime whether we access the loaded cache line only once or several times. Smaller arrays are, at least to a large fraction, resident in the cache hierarchy and we can observe that it makes indeed a difference if we have to access a cache line once or several times. This indicates that the bottleneck for our simple aggregation on a large array is not the computation power in the core but the memory bandwidth from the main memory to the core.

The graph in Figure 5.1(b) shows the curves for the z196 CPU. This graph indicates that fetching only every fifth element of a cache line would already improve the total runtime significantly. More detailed examination showed that this speed-up is not due to less computational overhead, but to parallel cache line prefetching, that only occurred when the stride between the accesses is high enough, we observed parallel prefetching as soon as the distance between two accesses is larger than 32 bytes. This indicates that either manual prefetching instructions should be inserted, or the prefetching of the System Z should be more aggressive to perform optimal in those scenarios, where a large region of memory is accessed sequential. In both architectures the possible performance gains, achieved by introducing computation logic on cache lines in the L1 cache, are very limited for data that does not fit into the caches. Unfortunately, analysts are often interested in summaries



(a) Runtime on Intel with 64 byte cache lines



(b) Runtime on Z with 256 byte cache lines

Figure 5.1: Strided access experiment on Z and Intel CPU

of large tables that are unlikely to reside in caches. We therefore examine the achievable memory throughput on System Z in the next section in more detail.

5.4 Memory Throughput

We observed in the previous section that the limiting factor when aggregating large arrays is the available memory bandwidth. Additionally, on System Z we have several instructions to choose from when moving content in memory. Table 5.1 lists the available move instructions. It is noteworthy that the implementation of the *move character* (MVC) instruction performs the move operation in the L1 cache while the *move character long (extended)* (MVCL(E)) instructions are

Instruction	Description
LG — STG	Load 64 bit from memory to a register and store that register back into memory
LGM — STGM	Load multiple registers from memory and stores them back
MVC	Moves 256 Byte in memory without using the CPU registers
MVCL(E)	Moves a memory page (4 KByte) in the memory

Table 5.1: Instructions to move memory content on System Z

performed using only the L4 cache as a buffer – without bringing the data higher in the cache hierarchy.

To determine the bandwidth of the different move instructions we measured the time to move 512 MB in memory using the different instruction. The result can be seen in Figure 5.2. Additionally to the move instructions we measured the

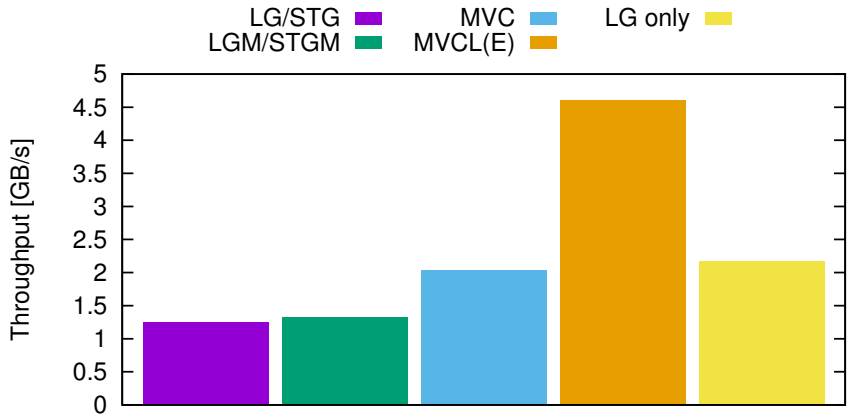


Figure 5.2: Throughput for moving data in memory using different instructions.

time to just load all array elements into a register to sum them up, without storing the content back, the throughput is depicted by the yellow bar. We clearly see that the MVCL instruction has a much higher throughput than all other move instructions.

We are also interested in the multithreaded bandwidth of the move instructions, especially since the mainframe offers many cores and we could already observe the benefits of parallel prefetching in the previous section. Figure 5.3 shows the runtime of the MVC and MVCL instructions when using a varying number of

threads. We performed the same experiment with different number of threads. The throughput is depicted in Figure 5.3. Please note that only four physical cores with a shared L4 cache were used in the experiment. When using several threads to move the memory content we observe that the MVCL based move clearly outperforms the MVC based move. However, the cache line move (MVC) scales better compared to the page move (MVCL), namely by a factor of almost three when using four threads instead of one compared to a factor of almost two for the page move. This behavior can be explained with the fact that there exists

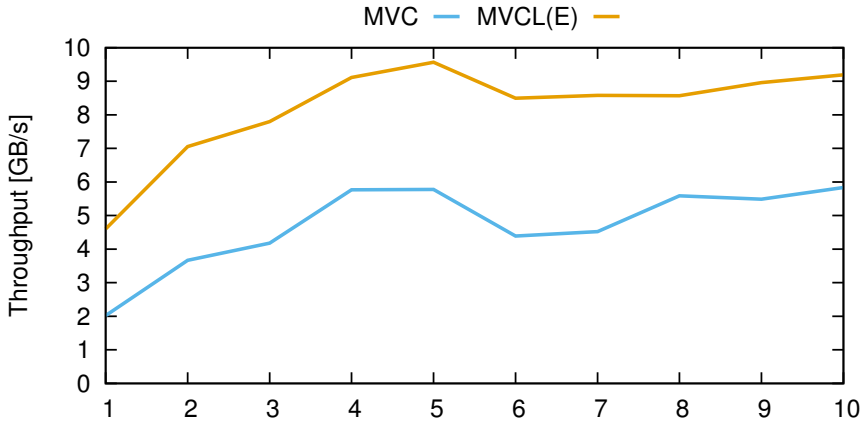


Figure 5.3: Multithreaded move experiment

only one L4 cache and page move hardware for four cores while each core has its own L1 cache and move character engine. Future generations of the System Z could also incorporate more page move engines to improve the parallel executions of page move operations.

5.5 Related Work

The widening gap between memory access latency and CPU cycle times has already been observed in the literature for many years. Researchers from Berkley proposed a solution called intelligent RAM, IRAM for short [73]. Intelligent RAM combines DRAM with processing power, this increases the bandwidth between memory and processing elements by putting the processing elements very close to the memory. Such memory components are typically very small and expensive. One reason for the high price of such memory is the testing complexity, that increases tremendously by introducing processing elements in the memory. Todays business applications, especially in-memory databases, need huge amounts of memory to store the ever-growing amount of data. Therefore, having all data

stored on intelligent RAM is not yet feasible and maybe never will be feasible.

Another approach, taken by e.g. Netezza, is to introduce *field programmable gate arrays* (FPGAs) into the data path between disk and memory. With those FPGAs it is possible to perform computations on the data before it is brought into the memory hierarchy. Such computations can increase the payload that is brought into memory in case of e.g. filtering, aggregation or compression. This approach has two mayor drawbacks. One drawback is the energy efficiency of those FPGA elements [61]. The other drawback only arises with the upcoming in-memory databases — there simply is no need to load **any** data from **disk** to answer queries.

5.6 Computation at the L4 Cache

The high throughput of the MVCL instruction led us to further investigations of the hardware implementation of this instruction. The move of a whole memory page is performed without bringing all cache lines into the L1, L2 or even the L3 cache but instead 16 cache lines of the L4 are used as a buffer for the page that needs to be fetched and stored back. This leads to the idea of performing computations on this memory page while it is being moved from one page address in memory to the other, without bringing any data into CPU registers.

To evaluate such a hardware we performed several simulations. The setup is as follows: We use two arrays A and B, each filled with 512 MB of data, stored in memory. These arrays correspond to columns in a column-store database and we perform filtering and aggregation on those columns. As a baseline we perform filtering and aggregation in the CPU only. Additionally we simulate the existence of filter and aggregation hardware. We simulate the existence of such a hardware component as follows: First, we create and store a bit mask of the qualifying tuples upfront in memory. Afterwards, when the hardware should be used to filter or aggregate a page, we simply move the page to a scratch area and use the upfront created bit mask as if the move created that bit mask. The time measurement of the simulated hardware filter and aggregation operations include the time to perform page moves but not the time to create the bit mask in memory.

Listing 5.1 contains the C++ code that was used to count the qualifying entries in the table. The first very simple query we look at just counts the number of qualifying tuples with a simple equality predicate, `SELECT COUNT(*) WHERE` A=0. Figure 5.4 shows the simulated runtime when varying the selectivity of the filter condition. We see that the move page instruction and therefore our expected filter runtime is only half the time compared with the standard CPU filtering runtime. When we compare the green line with the red line we also note that the cost to count the set bits in the bit mask is rather small compared with the cost to create

Listing 5.1: Code for counting qualifying tuples in CPU

```
int queryForCount(EntryType query, EntryType* table, int tableSize)
{
    int count = 0;
    for (int i = 0; i < tableSize; i++) {
        count += (table[i] == query);
    }
    return count;
}
```

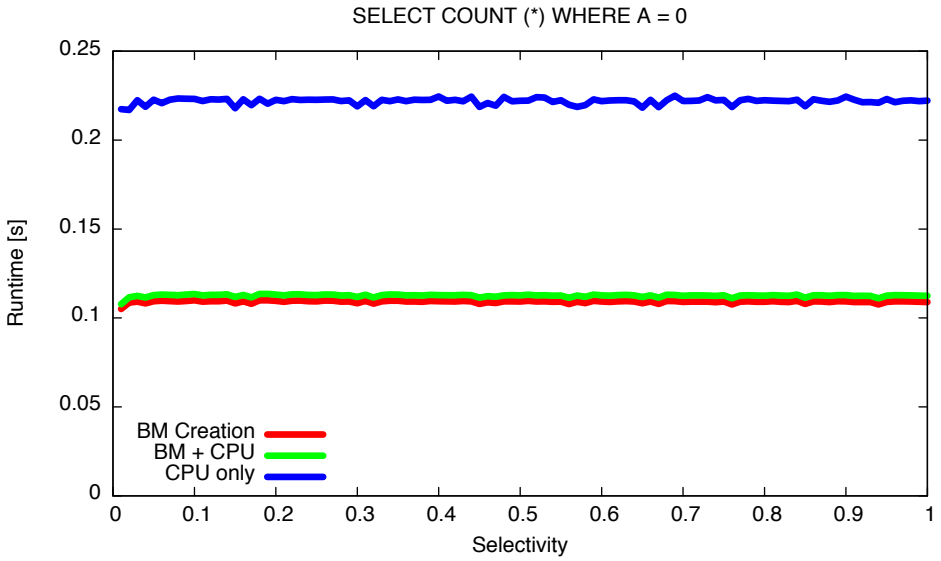


Figure 5.4: Simulated Filter Performance

the bit mask. Nevertheless, this additional cost can be avoided by counting the number of qualifying values already in the filter engine.

When we now look at a slightly more complex query, namely the query `SELECT SUM`(B) `WHERE` A=0, we can perform this query in at least three different ways. First, we can perform the aggregation in the CPU by reading elements from the array A until we find a qualifying element and then read the corresponding element of B and add it to the running sum. We see the code to perform the aggregation fully in the CPU in Listing 5.2. Second, we can first compute a bit mask by streaming A through the new page move engine and then either use this bit mask in the CPU to decide what elements to load from B. Third, after computing the bit mask by streaming A through the new page move engine, we also stream the array B together with the bit mask through the page move engine, which now performs an aggregation under mask operation.

Listing 5.2: Code for aggregating all qualifying tuples in CPU

```
EntryType sum_B_where_A_0
    (EntryType* colB, EntryType* colA, int tableSize)
{
    EntryType sum = 0;
    for (int i = 0; i < tableSize; i++) {
        if (colA[i] == 0)
            sum += colB[i];
    }
    return sum;
}
```

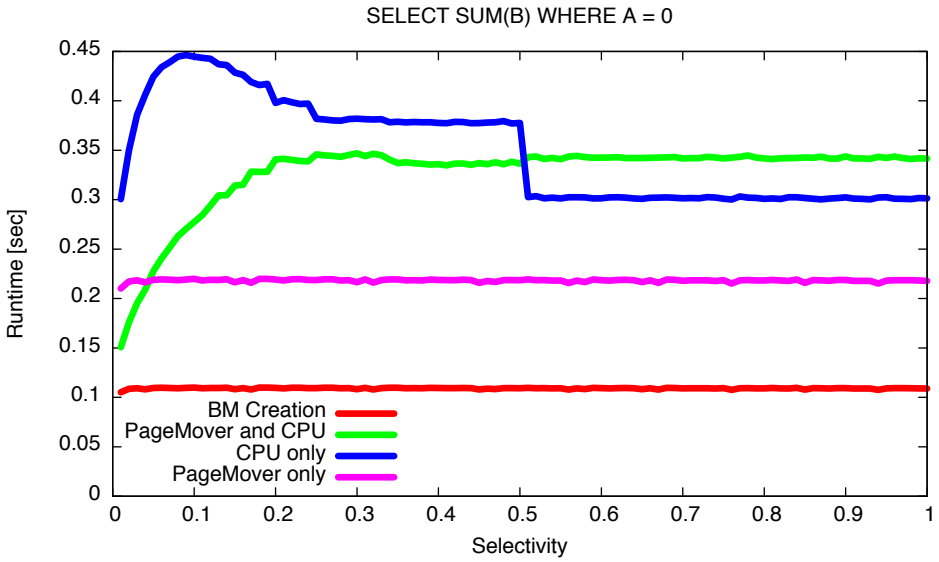


Figure 5.5: Simulated Aggregation and Filter Performance

The runtime of those three approaches can be seen in Figure 5.5. The blue line corresponds to the first option, using only the CPU, the green line to the second, using the created bit mask as filter when performing the sumation in the CPU, and the violet line correspond to the third option, creating the bit mask and performing the aggregation in the smart cache. When the selectivity is very high, i.e almost no elements qualify, it pays off to load only the needed elements into the CPU instead of performing the aggregation in the Page Move Engine. The break even point is around a selectivity of 5%.

5.7 Instruction Design

In this Section we will introduce three new instructions and illustrate their design with several use cases. Those instructions use an extended Page Mover engine, that allows to perform simple computations, like filtering, aggregation, and vector computation on the elements contained in the moved page. As a running example we will use the `lineitem` table from the TPC-H benchmark. The interesting columns of this table and their types are listed in Table 5.2.

Table 5.2: `lineitem` schema

Name	Type
⋮	⋮
`price`	64-bit double
`discount`	64-bit double
`quantity`	32-bit integer
`shippingdate`	64-bit unsigned integer

We assume the `lineitem` table to be stored in column layout in main memory. This means that a column is stored sequentially as a vector of values in memory. The address of the ith element of a column C can be computed given a start address $ad(C_0)$ and a scalar type $t(C)$ by the following formula:

$$ad(C_i) = ad(C_0) + i \cdot sizeof(t(C))$$

Those addresses are logical addresses and the physical addresses might jump after translation at page boundaries. The database system should ensure that columns are page aligned in memory, this means that a column starts at the beginning of a page and that a page always contains an integral number of entries.

We introduce the following instructions with the help of several use cases in the following subsections:

1. The *CompareBetween* instruction allows us to filter all values in a page according to a range predicate. All qualifying values are also immediately aggregated in a register.

2. The *Aggregate* instruction allows us to aggregate all values in a page, optionally using a bit map for masking qualifying values.

3. The *VecOp* instruction allows us to perform a vector operation with values stored on one page with values stored on another page. Additionally, the results of the vector operation are aggregated into a running aggregate that

can be retrieved from a specified register. Optionally, the aggregation uses only the qualifying vector entries specified by a bit mask.

5.7.1 Example Use Case for CompareBetween

Assume a data analyst is interested in the revenue of sales that lie in a certain price range. To obtain that information he formulates the following SQL query.

Q1:
```
SELECT SUM(price) AS rev_mid FROM lineitem
WHERE price BETWEEN 100.00 AND 2,000.00
```

Without our Smart PageMover all `price` values need to be loaded into a register before they can be compared against the values 100.00 and 2,000.00. Whenever a `price` value qualifies it is added onto a running sum, that is finally returned in a register. With the Smart PageMover the `price` column can be streamed

Table 5.3: Inputs to *CompareBetween* to answer Q1

Parameter	Value	Comments
ad	*ad*(`price`)	the start address of the column; the address is updated automatically after each invocation by adding the size of one memory page
size	|`price`|	the number of elements to be compared; the value is updated automatically after each invocation by subtracting 512
type	64-bit double	the type of each element on the memory page
agg_op	sum	operation used for aggregation
agg	0.00	this value is combined with the aggregated value of the current pages and contains the result after all pages have been processed
low	100.00	lower bound of qualifying elements
upp	2,000.00	upper bound of qualifying elements
inc_{low}	true	flag indicating if the lower bound should be included
inc_{up}	false	lag indicating if the upper bound should be included
mask_ad	0	bit mask address; no mask is specified in this case
mask	false	no bit mask is read
invmask	false	bit mask is not inverted

through the PageMover using the *CompareBetween* instruction in a tight loop. A memory page of 4 KB can hold $\frac{4 \cdot 2^{10}}{sizeof(t(\texttt{price}))} = 512$ elements of the price column. While all those elements stream through the Smart PageMover they are compared

against 100.00 and 2,000.00. The qualifying elements are immediately added up and the result is returned in a register. Meanwhile, the state of the bit mask is also updated. Since we are not interested in the bit mask for this use case we will specify how the bit mask is updated later. Table 5.3 contains all arguments passed to the *CompareBetween* instruction to answer Q1.

We use the next use case to explain how the bit mask is updated and to introduce the new *Aggregate* instruction.

5.7.2 Example Use Case for Aggregate

Let's assume a data analyst wants to compute the total revenue *rev* by summing up all `price` values.

Q2:
```
SELECT SUM(price) AS rev FROM lineitem
```

To calculate this sum without the Smart PageMover the whole `price` column needs to be brought into CPU registers and finally added up. With the Smart PageMover hardware the *Aggregate* instruction can be used in a tight loop to aggregate over one memory page at a time. Table 5.4 shows the parameter used to answer Q2.

Table 5.4: Inputs to *Aggregate* to answer Q2

Parameter	Value	Comments
ad	*ad*(`price`)	see Table 5.3
size	|`price`|	see Table 5.3
type	64-bit double	the type of each element on the memory page
agg_op	sum	operation used for aggregation
agg	0.00	see Table 5.3
mask_ad	0	no bit mask is specified
mask	false	no bit mask is used to filter
invmask	false	bit mask is not inverted

In Q1 the data analyst was interested in the revenue of all sales in a certain price range, but what happens if he is interested in the revenue of a particular year? Let's say he wants to calculate the revenue of 1994 and uses the following SQL query.

Q3:
```
SELECT SUM(price) AS rev1994 FROM lineitem
WHERE shippingdate BETWEEN '1994-01-01' AND '1995-01-01'
```

Without any special index structures or sort orders we need to scan the `shippingdate` column and test in the CPU for each entry if it qualifies. Whenever a qualifying entry is found we need to fetch the corresponding price into CPU registers to perform the aggregation. In contrast, our Smart PageMover allows us to use the *CompareBetween* instruction to create a bit mask and the *Aggregate* instruction to conditionally aggregate another column.

The start address of a bit mask can be specified by the *mask_ad* parameter, an address of 0 indicates that no bit map mask will be created or used. Otherwise, the address is interpreted as the start address of bit map in memory and that bit map is used to filter the input elements, iff the *mask* bit is set. If we would set the *mask* bit to true we would be refining an existing bit map. However, here we want to create a new bit map and therefore set the bit to false. *CompareBetween* modifies the bit map according to the outcome of the comparisons between elements of the column and the lower bound *low* and upper bound *upp*. The arguments to the *CompareBetween* instruction to answer Q3 are listed in Table 5.5.

Table 5.5: Inputs to *CompareBetween* to answer Q3

Parameter	Value	Comments
ad	*ad*(`shippingdate`)	see Table 5.3
size	|`shippingdate`|	see Table 5.3
type	64-bit double	the type of each element on the memory page
agg_op	nop	no aggregation needed
agg	0	does not matter
low	'1994-01-01'	lower bound of qualifying elements
upp	'1995-01-01'	upper bound of qualifying elements
inc_{low}	true	include the lower bound
inc_{up}	false	exclude the upper bound
mask_ad	0x2000000	this address is automatically incremented
mask	false	we do not use the mask, but simply overwrite it
invmask	false	we do not invert the bit mask before writing

To make the caching of a bit mask effective, it is recommended to aggregate on all pages of columns that correspond to the cached bit mask, before creating the next bit mask and processing the next pages of the columns. The input to the *Aggregate* instruction can be found in Table 5.6. An alternative instruction design allows for several aggregation operations in the same instruction. This would allow us to formulate, for example, the average calculation as a separate count and sum.

Table 5.6: Inputs to *Aggregate* to answer Q3

Parameter	Value	Comments
ad	*ad*(price)	see Table 5.3
size	|price|	see Table 5.3
type	64-bit double	the type of each element on the memory page
agg_op	sum	operation used for aggregation
agg	0.00	see Table 5.3
mask_ad	0x2000000	bit mask starts at address 0x2000000
mask	true	mask is used to filter
invmask	false	mask is not inverted

5.7.3 Example Use Case for Vector Operations

Vector operations were already part of earlier ESA390 systems. We do not put much focus on those operations. The main difference to the previous vector operations in the ESA390 is that we don't work inside of vector registers in the CPU but perform computations while two or more pages are streamed through the Smart PageMover , hence we do not bring the data into the CPU. Accessing more than two pages simultaneously enables conditional aggregation on the result of the vector operation or writing the result out without destroying one of the inputs.

For example, assume we want to find out how much money the company gave away by granting discounts.

Q4: `SELECT SUM(price * discount) AS lost_rev FROM lineitem`

To answer this query we use the *VecOp* instruction in a tight loop. Its parameters can be found in Table 5.7.

Table 5.7: Inputs to *VecOp* to answer Q4

Parameter	Value	Comments
ad_1	*ad*(`price`)	see Table 5.3
ad_2	*ad*(`discount`)	start address of the second vector; see above
size	|`price`|	see Table 5.3
type	64-bit fix point	the type of each element on the memory page
vec_op	multiply	vector operation used
agg_op	sum	operation used for aggregation
agg	0.00	see Table 5.3
mask_ad	0	no mask is specified
mask	N/A	not used
invmask	N/A	not used

5.8 Patent Application: Accelerator for Analytical Workloads

The following figures show possible implementations of the operations, that were introduced in the previous section.

The Filter Engine, shown in Figure 5.6, produces a bit mask that is stored in memory. The Filter Engine consists of a Parallel Compare circuit, which performs $2 \cdot k$ parallel comparisons between the k input elements from the L4 cache and the lower bound as well as upper bound provided by the CPU instruction. The outcome of those comparisons is given by k bits, which are fed into the MaskBuffer, to create a bit mask, and additionally into an Incrementer to keep track of the number of elements that passed the filter.

In the figure we see an instance of the Filter Engine with $k = 4$ and a scalar width of 64 bit. Figure 5.7 shows the control sequence for the Filter Engine.

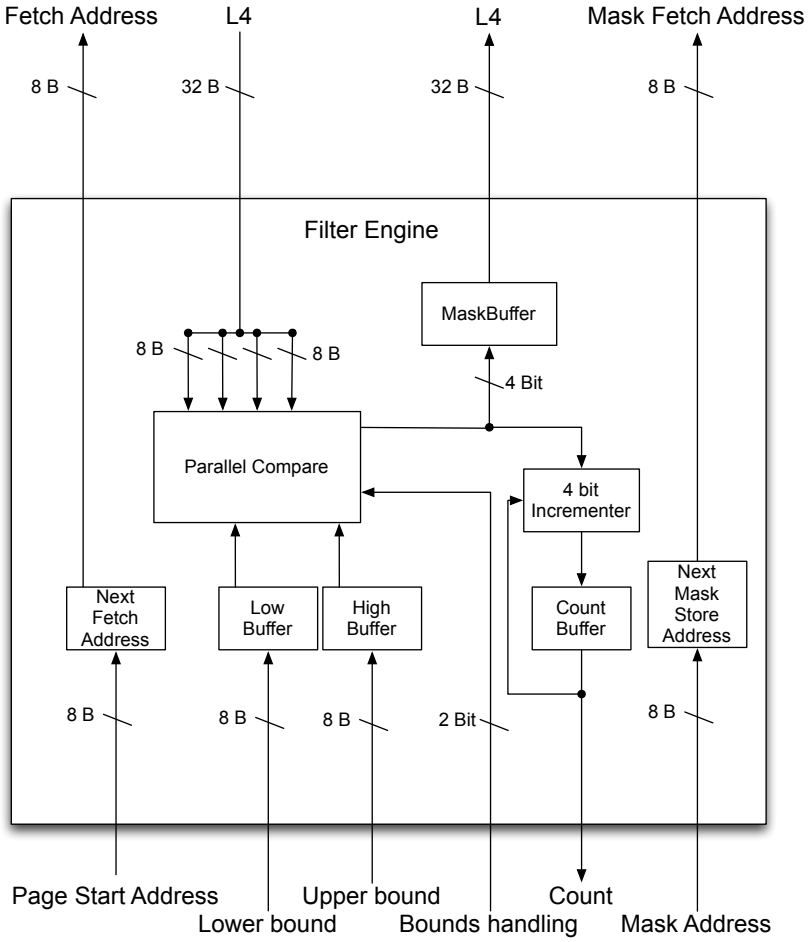


Figure 5.6: Filter Engine

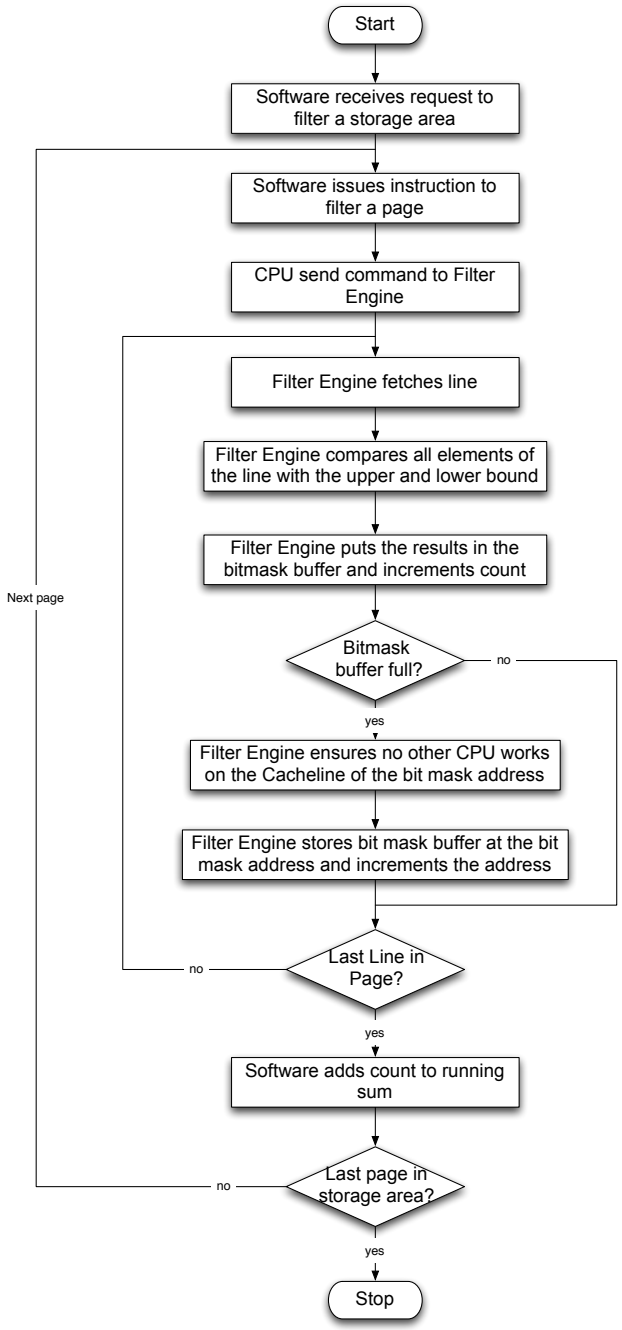


Figure 5.7: Filter Engine Process Flow

A bit mask, that was for example created by the Filter Engine, can be used by the *Aggregation under Mask Engine*, see Figure 5.8, to aggregate all qualifying entries of a memory page. This is achieved by loading the bit mask and chunks of

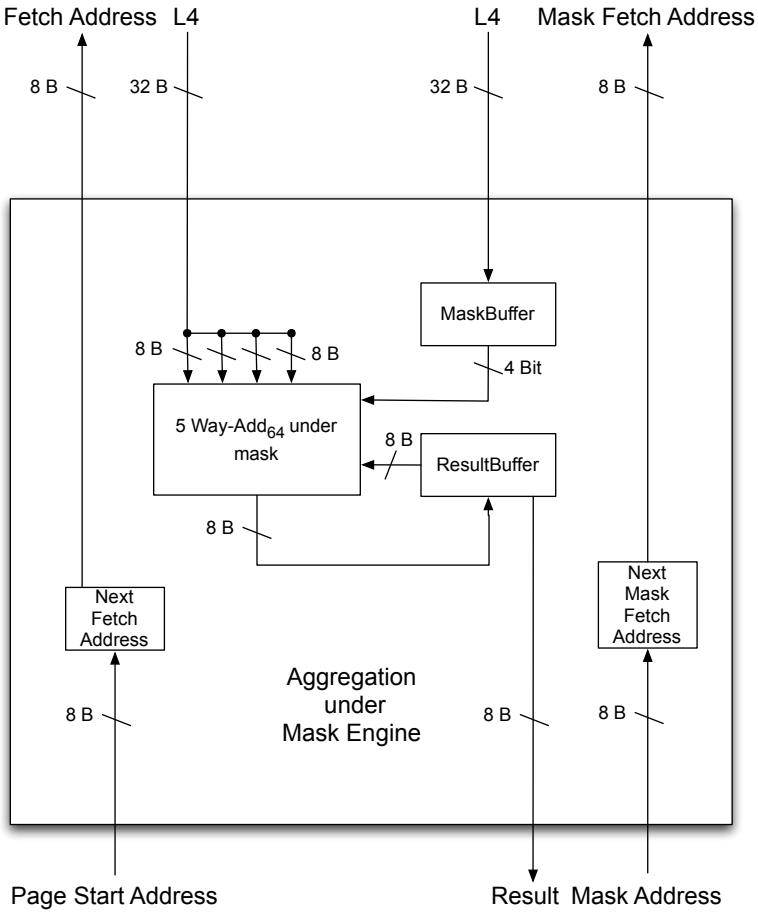


Figure 5.8: Aggregation under Mask Engine

the memory page into the engine. Afterwards aggregation operations, like finding the maximum or summing up all qualifying values, are performed with respect to the loaded bit mask. Figure 5.9 visualizes the process flow of the Aggregation under Mask Engine.

It is interesting to note that the two engines could be operated interleaved instead of sequential. This would allow to cache the bit mask in the engine and could thereby reduce memory traffic for storing and loading the bit mask.

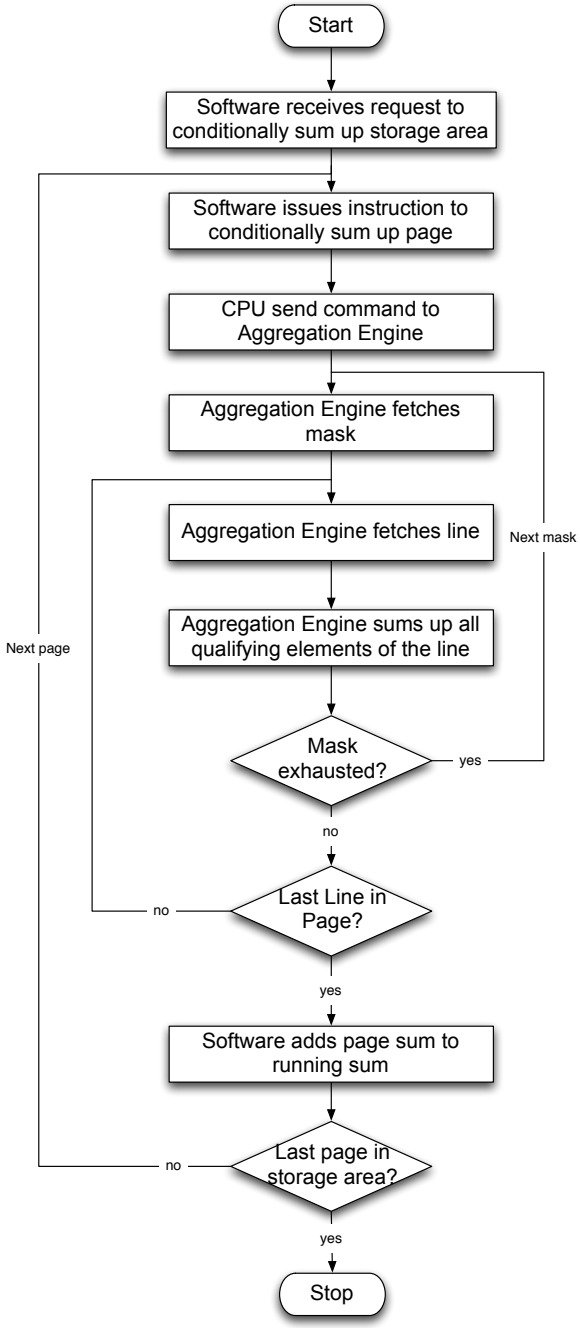


Figure 5.9: Aggregation under Mask Engine Process Flow

5.9 Conclusion

In this chapter we developed a new hardware approach to improve the runtimes of many analytical queries. The idea to bring computational power closer to the data, by introducing it into caches, is a promising compromise to intelligent RAM. Not only filter and aggregation operations could be performed at the cache level, but also vector operations and compression techniques can further increase the payload that is brought into the upper levels of the cache hierarchy.

This idea also led to a patent application and will be hopefully used in next generation mainframes to enable efficient in-memory databases and analytics as the future workload for the IBM System Z.

Appendix A

Additional Results of the "New Workloads for the IBM Mainframe System Z" Project

A.1 Algorithms

The survey paper Top Ten Algorithms in Data Mining [93] served as a starting point to identify the most important problems and algorithms in data mining workloads. Table A.1 lists those ten algorithms.

Algorithm	Type
C4.5	Classification
k-means	Clustering
Support Vector Machines	Classification
Apriori	Frequent Itemset Mining
Expectation-Maximation	Clustering
PageRank	Ranking
AdaBoost	Classification
k-nearest neighboor	Classification
CART	Classification

Table A.1: Top Ten Algorithms in Data Mining

The performance of classification algorithms is not only measured with respect to runtime but also the quality of the resulting model. The quality of a classification model is not easy to measure. Therefore we decided to first look at improved algorithms for clustering and frequent item set mining.

Additionally we took a look at sorting in the context of in-memory databases. Since sorting is, especially for string data, one of the more expensive operations performed in an in-memory database. For some algorithms sorting the data is a necessary preprocessing step to achieve high performance.

A.1.1 Clustering

Two Clustering algorithm are present in the top 10 algorithms in data mining [93]. This indicates the importance of clustering in data mining. Clustering algorithms are used to cluster a set of data points into different compartments. Data points are collections of features, where features can be different quantities such as weight, height, income etc. The previous mentioned features are all continuous, but you can also have features like gender or product category that are discrete. A Compartment can be described by a centre point C_i and consists of all data points that are *closer* to the center point C_i than to any other center point C_j. Such compartments could then be used to classify future data points.

In most clustering algorithms the distance between two data points or between data points and compartment centers need to be determined. In case of the IBM clustering, called BlueClusterer, this involves many logarithm computation, for the continuous features of the data points.

We analyzed the performance of the BlueClusterer algorithm on a z10 machine using gprof [39]. We used the Abalone dataset from the UCI Machine Learning Repository [34] as an input to the clustering algorithm. The resulting run time break down can be seen in Figure A.1.

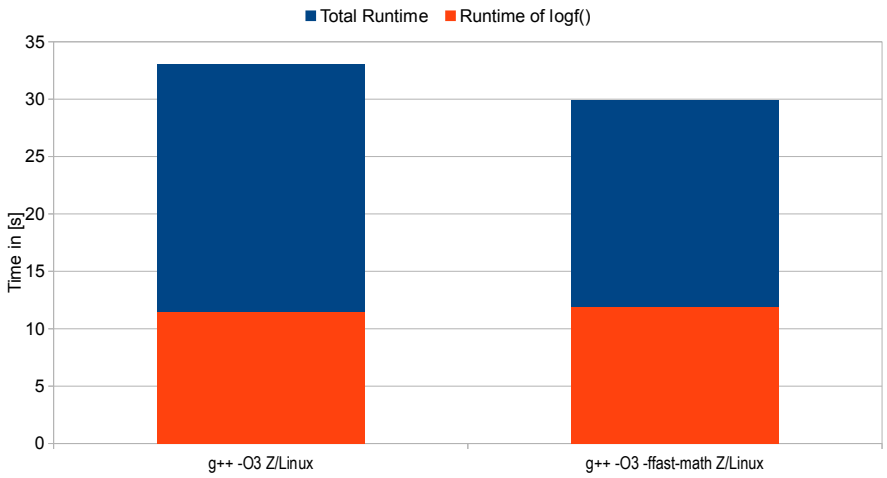


Figure A.1: Clustering runtime breakdown on Z/Linux

We can clearly see, that a large portion of the runtime is spent in the logarithm

calculation. On Intel and Power platforms exists a hardware `log` instruction, so we decided to perform the same analysis on an Intel CPU. Figure A.2 shows the breakdown.

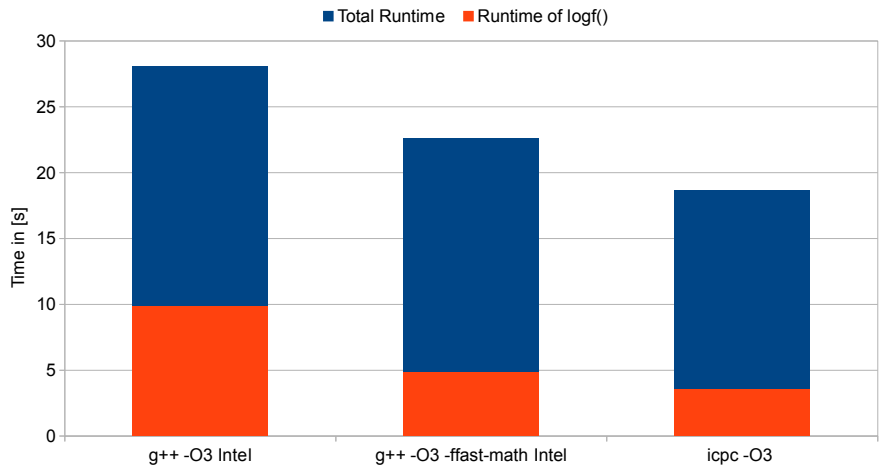


Figure A.2: Clustering runtime breakdown on Intel

The compiler option `-ffast-math` leads to the inlining of the `log` instruction in the source code and the removal of all unnecessary checks before executing the `log` instruction. Interestingly when using the Intel compiler icpc no hardware `log` instruction is issued, but a software routine using SIMD (Same Instruction Multiple Data elements) instructions is performed which performs even better then the hardware instruction.

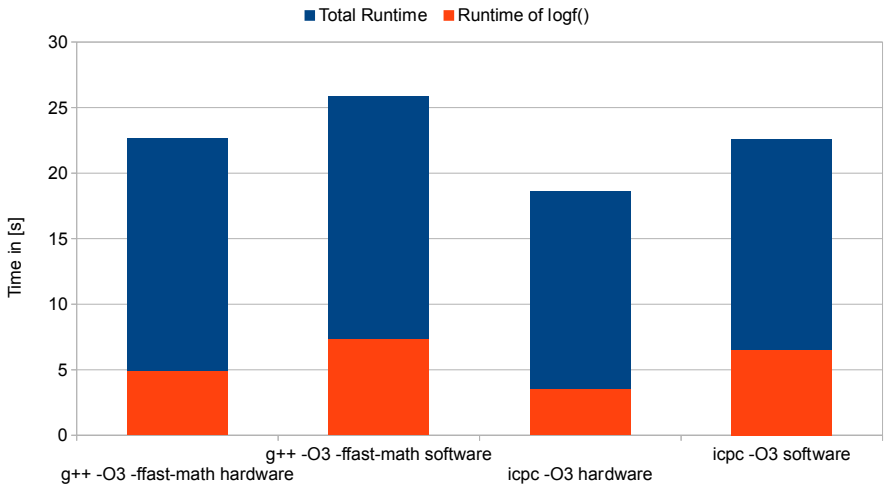


Figure A.3: Clustering Runtime Software vs Hardware

To measure the effect of having a special log instruction in the architecture we forced the compilers on the Intel machine to use the software logarithm routine from the glibc. The result can be seen in Figure A.3.

Summary

We observe, that the runtime of the logarithm calculation is reduced by 33% in case of the `log` instruction and by 45% in case of the SIMD routine of the intel math library. We conclude that such speed-up could also be expected for the Z architecture if either hardware logarithm instruction or even better SIMD instruction would be introduced and we suggest to introduce SIMD instructions in the system Z rather then developing special logarithm instructions.

A.1.2 Frequent Item-set Mining

A good introduction to the topic of frequent item set mining and clustering analysis can be found in the book Data Mining Concepts and Techniques by Jiawei Han and Micheline Kamber [55]. Frequent item-set mining was introduced by Agrawal [5] as the computational expensive step in association rules mining. Association rules mining, if applied to shopping transactions, tries to find patterns of the form: If a customer bought *milk* and *sugar* he probably will also buy *eggs*. Such rules can be used in many decision making processes, e.g. to decide product placement in stores or advertisement.

The frequent item-set mining can be formulated as follows: Given a list of item baskets B, basically the list off all shopping transactions, and a threshold θ, find all sets of items that were bought together more then θ times.

The first algorithm to solve that problem was introduced by Agrawal and is called APRIORI. It is based on the a priori knowledge that no set of items can be frequent, if not all subsets are also frequent. The algorithms builds candidate sets by combining frequent item-sets to larger item-sets and then prunes away all candidate sets, that are not frequent. We implemented an APRIORI algorithm that was described in [5] as a baseline. Additionally we implemented a cache-oblivious FP-Growth algorithm based on [35] and improved further on it by eliminating unnecessary pointers in the trie data structure. Algorithm 3 presents the pseudo

code of the FP-Growth algorithm.

Algorithm 3: FPGrowth($Tree, \alpha$)

```
1:  if Tree is single path then
2:     output 2^Tree + α with support = sup(α)
3:  else
4:     for all item α_i ∈ Tree do
5:        output pattern β = α + α_i with support = sup(α_i)
6:        construct β's conditional Tree Tree_β
7:        if Tree_β ≠ ∅ then
8:           FPGrowth(Tree_β, β)
9:        end if
10:    end for
11: end if
```

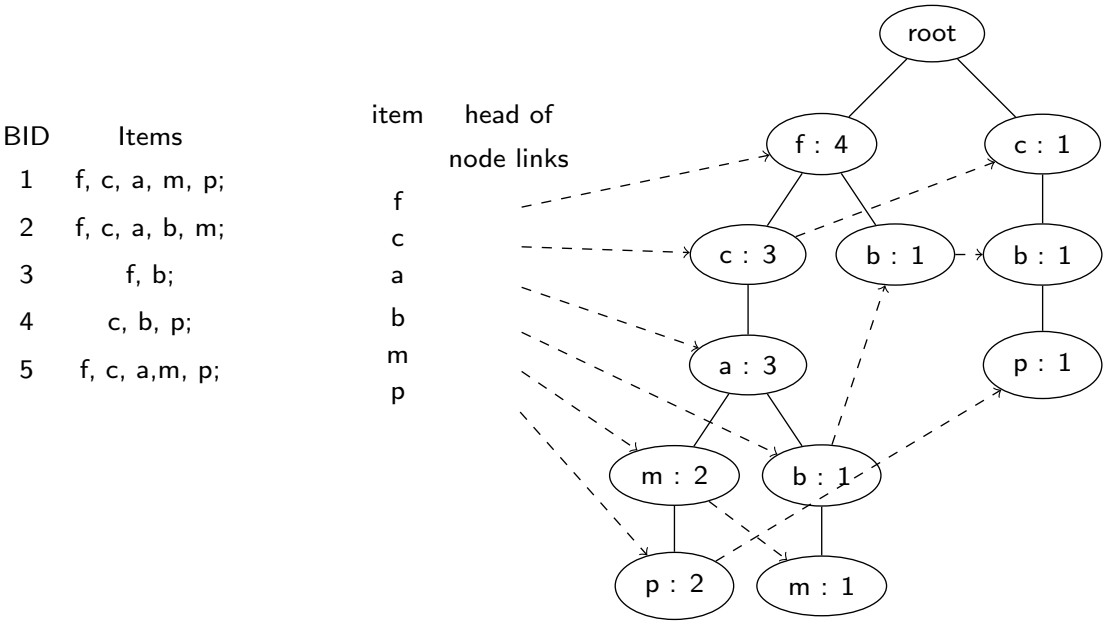


Figure A.4: Example FP-Tree

We see in Figure A.4 an example of an FP-Tree. The FP-Tree data structure can be used to mine frequent item-sets very efficiently. We introduced a dense representation of the data structure to further compress the data structure and to make cache line prefetching more efficient.

Figure A.5 shows the dense representation for our example.

Additionally we implemented an alternative to the node links. In the traditional FP-Tree for every item all nodes that contain the item are explicitly linked, this leads to a considerable space consumption for the link pointers. However, the algorithm only needs to find all nodes containing the item that is deepest in the tree and move upwards from there. This can be done by only storing pointers to the leaf nodes and sweeping through the tree.

Due to time constraints and as we were missing a representative large data-set as well as a state of the art baseline implementation to report performance results we stopped investigating the frequent itemset mining algorithms. In interesting starting point for further investigation is the Frequent Itemset Mining Implementations Repository[1] of the FIMI 03 and FIMI 04 workshops.

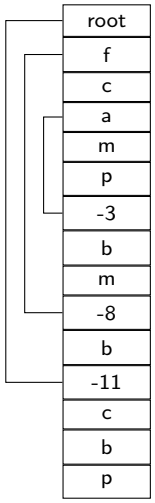


Figure A.5: Dense array representation of the example tree.

A.1.3 Sorting

Sorting is an important step in some database operations like the *sort merge join* or the *order by* operator. Index creation almost always sorts data, except when creating hash indexes. Analysts that are interested in top-k results or different quantiles of the data also often simply sort the data set, or at least parts of it.

Sorting is a very well studied problem with well understood theoretical bounds. To further improve sorting on modern and new hardware, we tried to make better use of already existing hardware, e.g. caches and conditional loads and stores, and also give ideas on new hardware that would support faster sorting algorithms.

Small Hardware Sorts

One approach to speed up sorting operations in memory is to introduce small hardware sorts, that allow quick-sort or any partitioning sort to stop as soon as the partition size is smaller then the hardware sort width.

To measure the effectiveness of such an approach we performed a simulation. In this simulation we created a random sequence of 10^7 elements and sorted this sequence using the quick-sort implementation used in the sun JDK 1.6. Whenever a partition had less then k elements we stopped the recursion and assumed a hardware sort would be performed on this partition. Figure A.6 shows the runtime

[1] `http://fimi.ua.ac.be/`

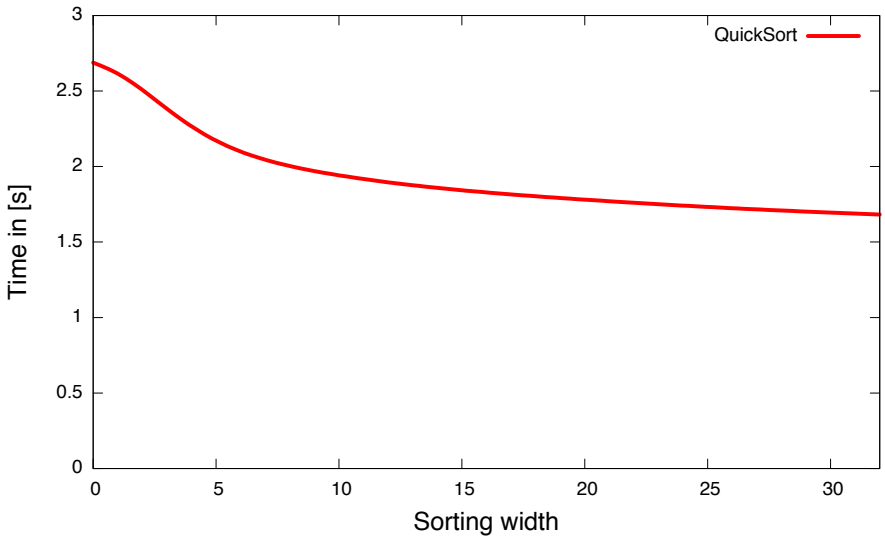


Figure A.6: k wide hardware sort in Quick Sort

for different values of k. The possible gains are in the range of 10% to 30% for fixed size data types. If such gains are enough to justify investing into the development of new sorting hardware was not explored.

In-memory String Sort with Tournament Sort and UPT,CFC

Almost all business databases fit into the main memory of todays clusters and mainframes [75]. Therefore we analyze the in-memory sorting performance of two important string sorting algorithms.

Tournament Sort : External sort algorithm based on a so called *loser tree*, that can be found in The Art of Computer Programming Volume 3 Searching and Sorting by Donald E. Knuth [31]

Multi-key Quick-Sort: Quick-sort adaption for string sorting by Bentley and Sedgewick [15], extended to use 64 bit comparisons

Additionally we improved the Multi-key Quick-Sort algorithm to better utilize caches.

Multi-key Copy Quick-Sort: Multi-key quick-sort with improved cache locality by copying parts of the string

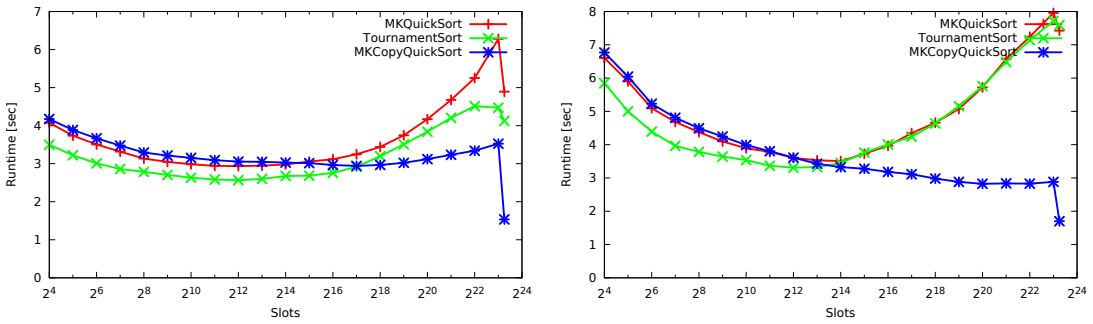
Since the Z architecture already offers some hardware assist instructions, see [51], to perform the tournament sort algorithm we compared it against multi key quick-sort. The third algorithm is an improved version of the multi-key quick-sort,

coined *multi-key copy quick-sort*. In the multi-key copy quick-sort we copy the currently compared part of the strings into an array and use this array to perform the comparisons. Whenever a swap operation is performed, not only the string pointer, but also the copied keys are swapped. This avoids costly cache misses when chasing the string pointers to perform comparisons.

Our sorting benchmark is based on the input generator of the Sortbenchmak website `http://www.sortbenchmark.org`. The input consists of ten million records of size one hundred bytes. A record in turn consists of ten random bytes, called the key, and a sequence number.

For all of the tested algorithms our benchmark starts by loading all records in main memory, this is not part of the time measurement. The next step is the *run generation* step. Each algorithm has an array of size *slots* to generate runs. In case of the tournament sort the run length is expected to be $2 \cdot slots$ and for the two quick-sort variants the run length is always *slots*. Afterwards a merge phase is executed in which all runs are merged into a single run. If only a single run exists no merge phase is needed.

We compare the performance of the three sorting algorithms on a zGryphon+ with the performance on an Intel Xeon CPU. We can see in Figure A.7 that

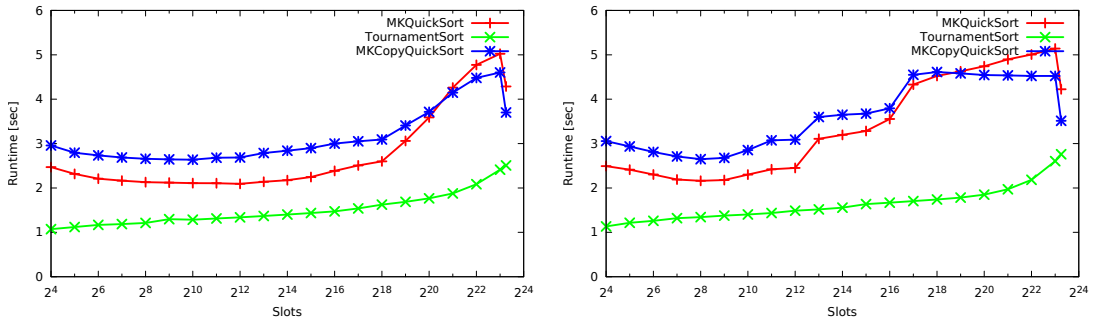


(a) Total runtime on Z

(b) Total runtime on Intel

Figure A.7: Total sort times, including run generation and merging, on random data

tournament sort is the fastest, as long as we have to merge. In case of the normal multi-key qick-sort we can observe that merging helps even in memory and not only in an external sort. However, if we create a single sorted run we can see that our multi key copy quick-sort has a superior runtime. This stems from the better cache locality due to the partitioning. The curves look similar for System Z and Intel, but the tournament sort has no real advantage over quick-sort on Intel. The gap between the multi-key copy quick-sort and the other sorting approaches is also wider on Intel.



(a) Total runtime with sorted input on Z (b) Total runtime with sorted input on Intel

Figure A.8: Total sort times, including run generation and merging, on sorted data

To see the effect of sortedness we also performed the sorting algorithms on already sorted data, the results can be seen in Figure A.8(a) and following. On sorted input the advantage of tournament sort is evident. If the data is fully sorted no merge phase is needed for the tournament sort. Even partially sorted input sequences drastically reduce the number of produced runs. We can clearly observe the smaller caches of the Intel CPU in Figure A.8(b).

What we can observe in our experiments is, that our improved multi-key quicksort is the fastest for unsorted string but the tournament sort can exploit presortedness.

Improvement of Tournament Sorting on the Mainframe

Since tournament sort is also used in IBM products, it is beneficial to improve the hardware assist instruction, to gain better performance without rewriting any client code.

Let us first have a closer look on how tournament sort works and how it is supported by the IBM System Z mainframe. In tournament sorting we sort a set of keys by creating runs and merging those runs. A run can be created by using a so called loser tree. This loser tree is a binary tree. Let us assume 2^n keys fight in a tournament to find out who is the smallest. All keys are compared in the first round with their neighbor. The 2^{n-1} losers are stored in the leaves while the winners proceed to the next round. This is repeated till the last two keys are compared with each other. The loser is stored in the root of the tree while the absolute winner is stored separately. If we now want to insert a new key in the tree and additionally find the next larger key then we just need to look at all keys that lost to the previous winner. In a loser tree those keys all lie on the so called *winner path*, i.e. the path starting with the loser of the first comparison going up till the root. Figure A.9 shows a loser tree and highlights the winner path, notice

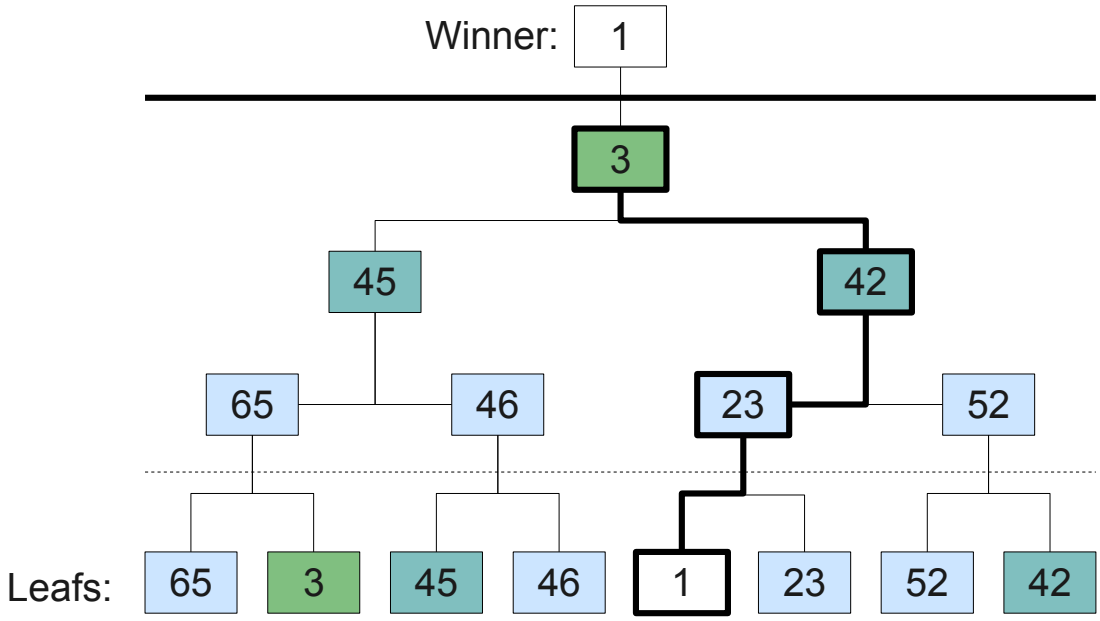


Figure A.9: Loser tree for the sequence : $[65, 3, 45, 46, 1, 23, 52, 42]$

that the leafs are only virtual, that means they are not stored in memory.

We are sorting strings instead of numerical values that fit into registers. This can be solved by using codewords that fit into a register and encode a position and parts of the string key. These codewords can often be used to perform comparisons without touching the stored string. Sometimes, if two codewords are equal, the two strings need to be touched, but in total every string will be loaded at most once into CPU registers. For more details take a look in the description of the CFC (compare and form codeword) instruction in the Principles of Operation or in the COMAD paper [51].

We focus now on the update tree (UPT) instruction. The update tree instruction updates a loser tree in memory by comparing code words on the winner path and swapping them if needed. This check if a swap is needed causes many branch mispredictions if the input sequence is not sorted. To avoid those expensive mispredictions conditional loads and stores (LOC/STOC) can be used. Instead of performing speculative execution and paying the price of rolling back the instructions a data stall is introduced.

The experimental evaluation of this idea can be seen in Figure A.10. The runtime of the tournament sort can be reduced by roughly 25%. We observed that the main gain was in the update tree instruction and almost no gain could be observed by introducing the conditional loads and stores in the compare and form codeword instruction.

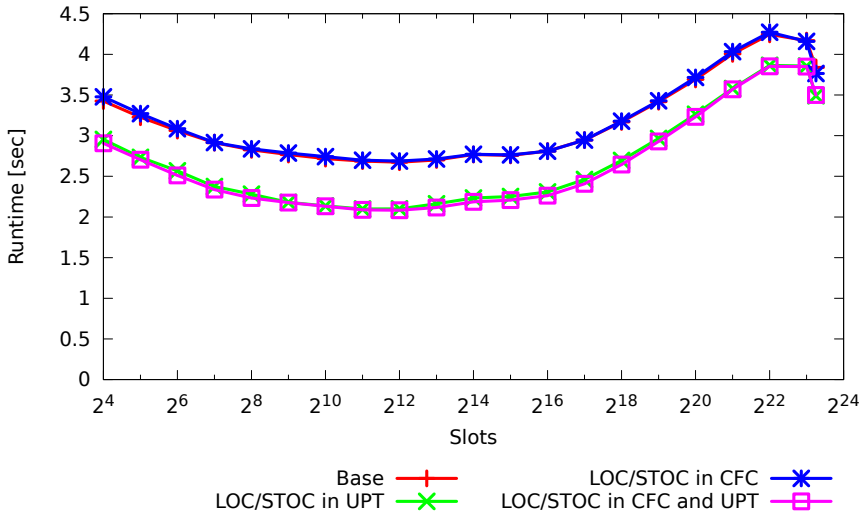


Figure A.10: Runtime of Tournament Sort with LOC/STOC

In another experiment we could also observe that on a sorted sequence the update tree instruction will perform worse with conditional loads and stores then with branches. This is obvious, since for sorted inputs there are no branch mispredictions. In cases of low branch misprediction it is better to use branches, so it would be interesting to only switch to conditional loads and stores if a high branch misprediction rate was detected.

Another approach to speed up the update of the loser tree is to load the elements of the winner path in a predefined storage area. Afterwards the swapping could be done very efficiently by a sequencer hardware similar to the sequencer used in the move character (MVC) instruction. The resulting winner path needs then to be stored back into the tree. We simulated hardware assistance for this approach and the results can be seen in Figure A.11.

The simulated sequencer approach could not beat the normal software update tree instruction in our simulations. For this we assume two main reasons. First, the winner path is not adjacent in memory and needs to be fetched from different places. This could be changed by clever prefetching, since the location of the winner path is deterministic. Second, the software implementation does not always load the whole winner path if an early stopping criterion is met. But even this could be of minor importance, since the prefetched elements of the winner path are still valid for the next update tree instruction, if it is used to sort strings. A real implementation of the sequencer idea could therefore still improve on the current UPT implementation.

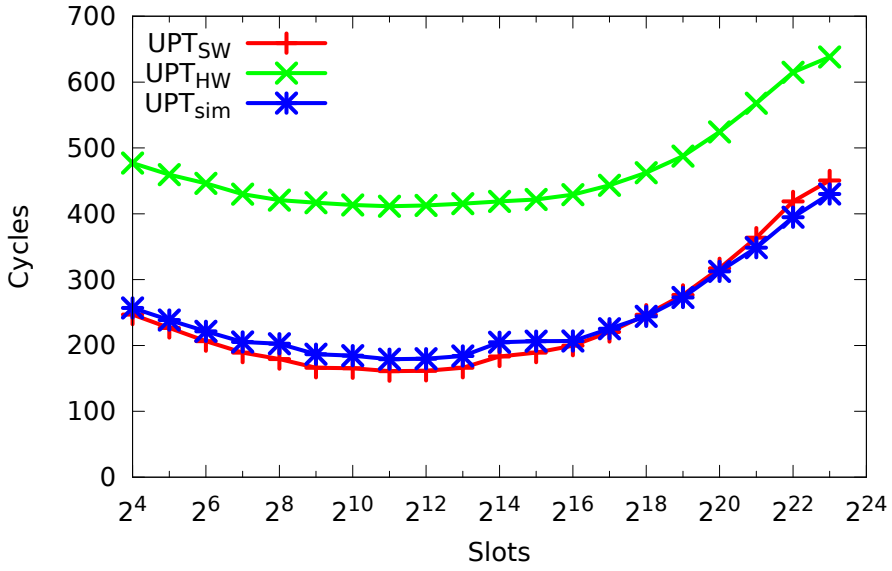


Figure A.11: UPT Sequencer Simulation

A.1.4 Summary

In this chapter we looked at some of the core problems and algorithms for data mining. Namely clustering and frequent item set mining. We identified the logarithm as a bottleneck for a class of clustering algorithms and proposed to implement either hardware logarithm instruction or, even better, to provide SIMD instructions and a SIMD logarithm implementation. For the frequent item set mining problem we implemented a cache oblivious version of the FP-Growth algorithm, that contains no specific optimizations for System Z.

Afterwards we investigated sorting, especially string sorting. For in-memory string sorting we proposed a new sort algorithm called Multi-key Copy Quick-Sort, an improved version of Multi-key Quick-Sort. Additionally we propose some improvements for the hardware assist instruction update tree (UPT) and estimate the possible performance gains.

List of Figures

List of Tables

Bibliography

[1] IBM ILOG CPLEX Optimization Studio. `http://www-03.ibm.com/software/products/us/en/ibmilogcpleoptistud/`.

[2] `https://www.systems.ethz.ch/node/334`.

[3] D.J. Abadi, D.S. Myers, D.J. DeWitt, and S.R. Madden. Materialization Strategies in a Column-Oriented DBMS. *ICDE*, pages 466–475, 2007.

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

[5] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, June 1993.

[6] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. *VLDB*, pages 1110–1121, 2004.

[7] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic Physical Design Tuning: Workload as a Sequence. *SIGMOD*, pages 683–694, 2006.

[8] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. *VLDB*, pages 169–180, 2001.

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

[10] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *PVLDB*, 5(10):1064–1075, 2012.

[11] Victor Alvarez, Felix Martin Schuhknecht, Jens Dittrich, and Stefan Richter. Main Memory Adaptive Indexing for Multi-core Systems. *DaMoN*, pages 3:1–3:10, 2014.

[12] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Ozsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.

[13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. *ICDE*, pages 362–373, 2013.

[14] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Ozsu. Main-Memory Hash Joins on Modern Processor Architectures. *TKDE*, 27(7):1754–1766, 2015.

[15] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. *SODA*, pages 360–369, 1997.

[16] Spyros Blanas, Yinan Li, and Jignesh M Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. *SIGMOD*, pages 37–48, 2011.

[17] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. *SIGMOD*, pages 975–986, 2010.

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

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

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

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

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

[23] Surajit Chaudhuri and Vivek R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. *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] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *PVLDB*, 4(6):362–372, 2011.

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

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

[28] 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.

[29] 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.

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

[31] Donald E. Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison Wesley, second edition, 1998.

[32] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

[33] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical Database Design for Relational Databases. *ACM TODS*, 13(1):91–128, 1988.

[34] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[35] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony D. Nguyen, Yen-Kuang Chen, and Pradeep Dubey. Cache-Conscious Frequent Pattern Mining on modern and emerging Processors. *VLDB Journal*, 16(1):77–96, 2007.

[36] Google Inc. Google Sparse and Dense Hashes. `https://code.google.com/p/sparsehash/`.

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

[38] Goetz Graefe and Harumi A. Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. *EDBT*, pages 371–381, 2010.

[39] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[40] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J Weinberger. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD*, pages 243–252, 1994.

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

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

[43] 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.

[44] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache Query Co-processing on Coupled CPU-GPU Architectures. *PVLDB*, 8(4):329–340, 2014.

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

[46] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? *CIDR*, pages 57–68, 2011.

[47] Stratos Idreos, Martin Kersten, and Stefan Manegold. Database Cracking. *CIDR*, pages 68–78, 2007.

[48] Stratos Idreos, Martin Kersten, and Stefan Manegold. Self-organizing Tuple Reconstruction In Column-stores. *SIGMOD*, pages 297–308, 2009.

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

[50] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.

[51] Balakrishna R Iyer. Hardware Assisted Sorting in IBM's DB2 DBMS. *COMAD*, 2005.

[52] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.

[53] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *PVLDB*, 8(6):642–653, 2015.

[54] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.

[55] Jiawei Han and Micheline Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, second edition, 2006.

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

[57] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. *DaMoN*, pages 55–62, 2012.

[58] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. *ICDE*, pages 195–206, 2011.

[59] Martin Kersten and Stefan Manegold. Cracking the Database Store. *CIDR*, pages 213–224, 2005.

[60] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[61] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *FPGA*, pages 21–30, 2006.

[62] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. Massively Parallel NUMA-aware Hash Joins. *IMDM*, pages 3–14, 2013.

[63] Thomas Legler, Wolfgang Lehner, and Andrew Ross. Data Mining with the SAP Netweaver BI Accelerator. *VLDB*, pages 1059–1068, 2006.

[64] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. Full-Text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics. *MapReduce Workshop*, 2011.

[65] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-Situ MapReduce for Log Processing. *USENIX*, 2011.

[66] Martin Lühring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. Autonomous Management of Soft Indexes. *ICDE Workshops*, pages 450–458, 2007.

[67] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Main-Memory Join on Modern Hardware. *TKDE*, 14(4):709–730, 2002.

[68] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

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

[70] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *SIGMOD*, pages 297–306, 1993.

[71] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the LRU-K page replacement algorithm. *JACM*, pages 92–112, 1999.

[72] R Barber G Lohman I Pandis, V Raman R Sidle, G Attaluri N Chainani S Lightstone, and D Sharpe. Memory-Efficient Hash Joins. *PVLDB*, 8(4):353–364, 2014.

[73] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17:34–44, 1997.

[74] 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. *SIGMOD*, pages 165–178, 2009.

[75] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. *SIGMOD*, pages 1–2, 2009.

[76] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. *SIGMOD*, pages 1493–1508, 2015.

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

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

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

[80] 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.

[81] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. *SIGMOD*, pages 351–362, 2010.

[82] 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.

[83] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. COLT: Continuous On-Line Tuning. *SIGMOD*, pages 793–795, 2006.

[84] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. On-Line Index Selection for Shifting Workloads. *ICDE Workshops*, pages 459–468, 2007.

[85] Karl Schnaitter and Neoklis Polyzotis. Semi-Automatic Index Tuning: Keeping DBAs in the Loop. *PVLDB*, 5(5):478–489, 2012.

[86] Stefan Schuh, Xiao Chen, and Jens Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. *SIGMOD*, 2016.

[87] Stefan Schuh and Jens Dittrich. AIR: Adaptive Index Replacement in Hadoop. *ICDE Workshops*, pages 22–29, 2015.

[88] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[89] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *PVLDB*, 8(9):934–937, 2015.

[90] Knut Stolze, Felix Beier, Kai-Uwe Sattler, Sebastian Sprenger, Carlos Caballero Grolimund, and Marco Czech. Architecture of a Highly Scalable Data Warehouse Appliance Integrated to Mainframe Database Systems. *BTW*, pages 628–639, 2011.

[91] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. *SIGMOD*, pages 1013–1020, 2010.

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

[93] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus F. M. Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowl. Inf. Syst.*, 14(1):1–37, 2008.

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

[95] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. *EuroSys*, pages 265–278, 2010.

[96] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: integrated automatic physical database design. *VLDB*, pages 1087–1097, 2004.