Noname manuscript No.
(will be inserted by the editor)

Towards Zero-Overhead Static and Adaptive Indexing in Hadoop

Stefan Richter · Jorge-Arnulfo Quiané-Ruiz · Stefan Schuh · Jens Dittrich

the date of receipt and acceptance should be inserted later

Abstract Hadoop MapReduce has evolved to an important industry standard for massive parallel data processing and has become widely adopted for a variety of use cases. Recent works have shown that indexes can improve the performance of selective MapReduce jobs dramatically. However, one major weakness of existing approaches are high index creation costs. We present HAIL (Hadoop Aggressive Indexing Library), a novel indexing approach for HDFS and Hadoop MapReduce. HAIL creates different clustered indexes over terabytes of data with minimal, often invisible costs and it dramatically improves runtimes of several classes of MapReduce jobs. HAIL features two different indexing pipelines, *static indexing* and *adaptive indexing*. HAIL static indexing efficiently indexes datasets while uploading them to HDFS. Thereby, HAIL leverages the default replication of Hadoop and enhances it with logical replication. This allows HAIL to create multiple clustered indexes for a dataset, e.g. one for each physical replica. Still, in terms of upload time, HAIL matches or even improves over the performance of standard HDFS. Additionally, HAIL adaptive indexing allows for automatic, incremental indexing at job runtime with minimal runtime overhead. For example, HAIL adaptive indexing can completely index a dataset as byproduct of only four MapReduce jobs while incurring an overhead as low as 11% for the very first of those job only. In our experiments, we show that HAIL improves job runtimes by up to 68x over Hadoop. This article is an extended version of the VLDB 2012 paper "Only Aggressive Elephants are Fast Elephants" (PVLDB, 5(11):1591-1602, 2012).

S. Richter, S. Schuh, J. Dittrich
Information Systems Group
Saarland University

J.-A. Quiané-Ruiz
Qatar Computing Research Institute
Qatar Foundation

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 [23]. In particular, log processing emerges as an important type of data analysis commonly done with MapReduce [5,36,18].

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 [43,22,35]. Over the last years, a lot of research works have focused on improving the performance of Hadoop MapReduce [12,26,32,34]. 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 [45, 15,35,33]. Some of these works have improved the performance of selective MapReduce jobs by orders of magnitude. However, all these indexing approaches have three main weaknesses. First, they require a high upfront cost for index creation. This translates to long waiting times for users until they can actually start to run queries. Second, they can only support one physical sort order (and hence one clustered index) per dataset. This becomes a serious problem if the workload demands indexes for several attributes. Third, they require users to have a good knowledge of the workload

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

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.

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 [5,18,38]. 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 [39,33]. 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 [33,12,26,32]) and finds a paper that shows how to create a so-called *trojan index* [15]. 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 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] The professor is aware that for some situations the opposite is true.

(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 [34] 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 [19,10, 2,8,11,45,35,15,33] 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* [28], where the general idea is to create indexes as a side-effect of query processing. This is similar to the idea of *soft indexes* [37], 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 [17,28,29,21,30,24] 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 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 [21] 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 [29] 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.

1.2 Research Questions and Challenges

This article 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?

1.3 Contributions

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

(1.) Zero-Overhead indexing. We show how to effectively 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++ [15]. We give an overview of HAIL and its benefits in Section 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). 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 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). In addition, we propose a new task scheduling, called *HAIL Scheduling*, to fully exploit statically and adaptively indexed data blocks (Section 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.

(4.) Zero-Overhead Adaptive indexing. We show how to effectively piggyback adaptive index creation on the existing MapReduce job execution pipeline (Section 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. We additionally propose a set of adaptive indexing strategies that makes HAIL aware of the performance and the selectivity of MapReduce jobs (Section 6). We present *lazy* and *eager* adaptive indexing, two techniques that allow HAIL to quickly adapt to changes in users' workloads at a low indexing overhead. We then show how HAIL can decide which data blocks to index based on the selectivities of MapReduce jobs.

(5.) Exhaustive validation. We present an extensive experimental comparison of HAIL with Hadoop and Hadoop++ [15] (Section 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.

Notice that, this article presents an extended version of the initial HAIL system [16] with the following significant added value: we enrich HAIL with the adaptive indexing pipeline, that allows HAIL to adapt to changes in query workloads in an automatic, incremental, and dynamic way (all of contribution **Zero-Overhead Adaptive indexing.**); we extend the HAIL task scheduling in order to balance the index effort at job execution time and exploit pseudo data blocks (half of contribution **Job execution.**); we run a large number of new experiments to validate our adaptive indexing techniques as well as the extended HAIL task scheduling (one third of contribution **Exhaustive validation.**).

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 8.

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

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 [13, 15, 14] for a detailed description). As soon as all results have been written to HDFS, the JobClient informs Bob that the results are available. Notice that, the execution time of the MapReduce job is heavily influenced by the size of the input dataset, because Hadoop MapReduce reads the input dataset entirely in order to perform any incoming MapReduce job.

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 [3] 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.

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] Actually it is a *split*. The difference does not matter here. We will get back to this in Section 4.2.

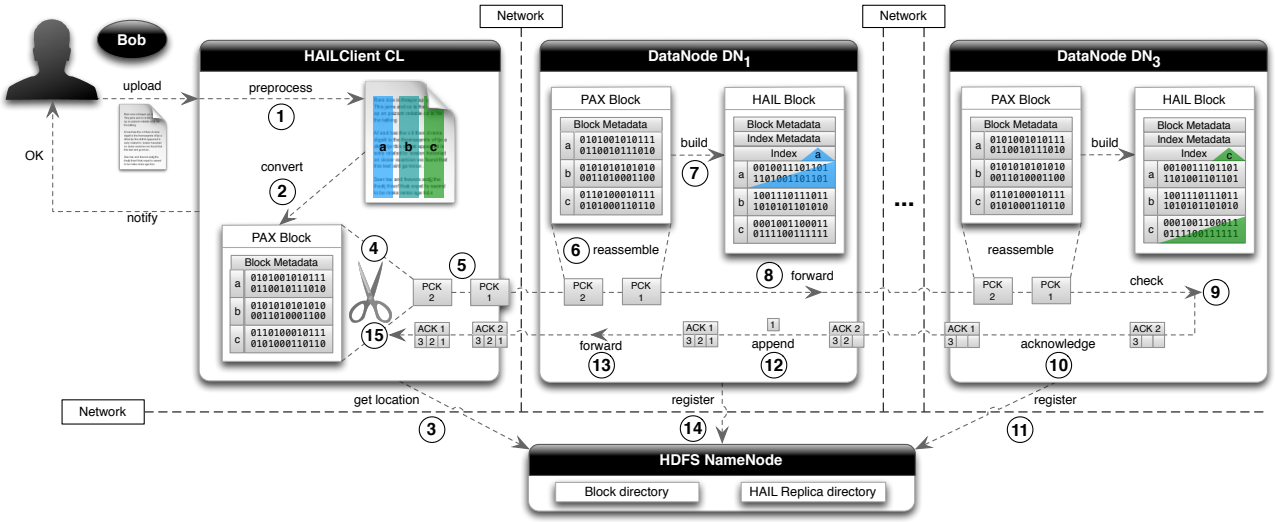


Fig. 1 The HAIL static indexing pipeline as part of uploading data to HDFS

3 HAIL Zero-Overhead Static Indexing

We create static indexes in HAIL while uploading data. One of the main challenges is to support different sort orders and clustered indexes per replica as well as to build those indexes efficiently without much impact on upload times. Figure 1 shows the data flow when Bob uploads a file to HAIL. Let's first explore the details of the static indexing pipeline.

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 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.

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

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

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

packet match the received checksums, DN_3 acknowledges the packet back to DN_2, which acknowledges back to DN_1. Finally, DN_1 acknowledges back to CL. Each datanode also appends its ID to the ACK. Like that only one of the datanodes (the last in the chain, here DN_3 as the replication factor is three) has to verify the checksums. DN_2 believes DN_3, DN_1 believes DN_2, and CL believes DN_1. If any CL or DN_i receives ACKs in the wrong order, the upload is considered failed. The idea of sending multiple packets from CL is to hide the roundtrip latencies of the individual packets. Creating this chain of ACKs also has the benefit that CL only receives a single ACK for each packet and not three. Notice, that HDFS provides this checksum mechanism on top of the existing TCP/IP checksum mechanism (which has weaker correctness guarantees than HDFS).

In HAIL, in order to reuse as much of the existing HDFS pipeline and yet to make this efficient, we need to perform the following changes. As before, the HAIL client (CL) gets the list of datanodes to use for this block from the HDFS namenode ③. But rather than sending the original input, CL creates the PAX block, cuts it into packets ④, and sends it to DN_1 ⑤. Whenever a datanode DN_1–DN_3 receives a packet, it does *neither* flush its data *nor* its checksums to disk. Still, DN_1 and DN_2 immediately forward the packet to the next datanode as 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 3.4). 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). 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 HDFS Namenode Extensions

In HDFS, the central namenode keeps a directory Dir_block of blocks, i.e., a mapping blockID ↦ 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) ↦

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 4).

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 as 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.

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 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 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 4.3). Figure 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 4.4).

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 2), but with three tiny changes:

(1) Bob specifies the *HailInputFormat* (which uses a *HailRecordReader* internally) in the main class of the MapReduce job. By doing this, Bob enables his MapReduce job to read HAIL Blocks (see Section 3.2).

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

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

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

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

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

(3) Bob uses a *HailRecord* object as input value in the map function. This allows Bob to directly read the projected attributes without splitting the record into attributes as he

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

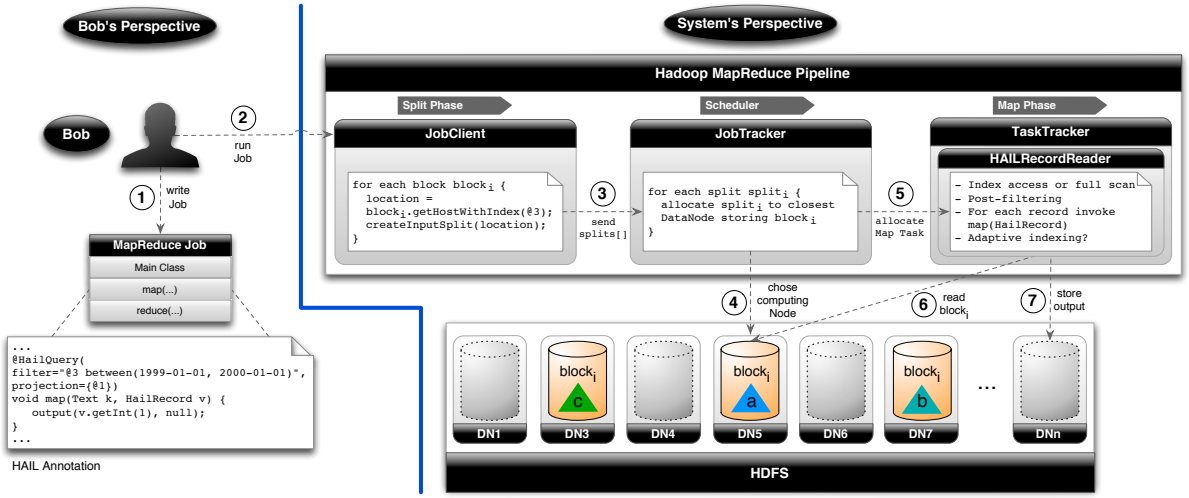


Fig. 2 The HAIL query pipeline

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 2.

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 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 3.1 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 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 [13]. 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 HDFS⑦. See [15,44, 14] 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 sev-

eral reasons. First, the JobClient cannot simply create input splits based only on the default block size as each HDFS block replica has a different size (because of indexes). Second, the JobTracker can no longer schedule map tasks based on data-locality and nodes availability only. The JobTracker now has to consider the existing indexes for each HDFS block. Third, the RecordReader has to perform either index access or full scan of HDFS blocks without any interaction with users, e.g. depending on the availability of suitable indexes. Fourth, the HDFS client cannot anymore open an input stream to a given HDFS block based on data-locality and nodes availability only: it has to consider index locality and availability as well. HAIL overcomes these issues by mainly providing two UDFs: the HailInputFormat and the HailRecordReader. 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.

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 7.

HAILRecordReader is responsible for retrieving the records that satisfy the selection predicate of MapReduce jobs (as illustrated in the MapReduce Pipeline of Figure 2). Those records are then passed to the map function. For example in Bob's query of Section 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 3.4). 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 3.1), HAIL passes them directly to the map function, which in turn has to deal with them (just like in standard Hadoop MapReduce). For this, HAIL passes a record to the map function with a flag to indicate a bad record or not.

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.

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 5.1, we start with a general overview of the HAIL adaptive indexing pipeline. In Section 5.2, we focus on the internal components for building and storing clustered indexes incrementally. In Section 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 6, we introduce three additional adaptive indexing techniques that make the indexing overhead over MapReduce jobs almost invisible to users.

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

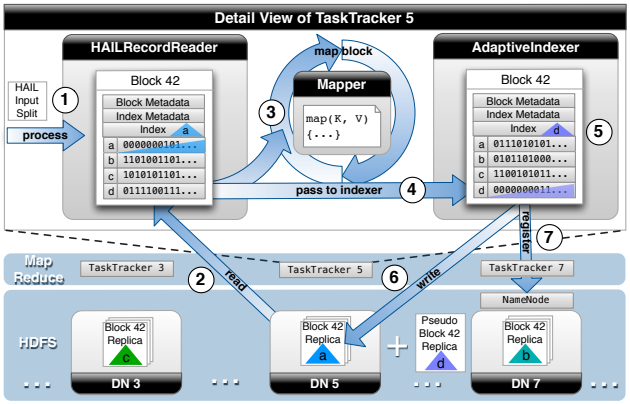


Fig. 3 HAIL adaptive indexing pipeline.

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 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.

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 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

[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.

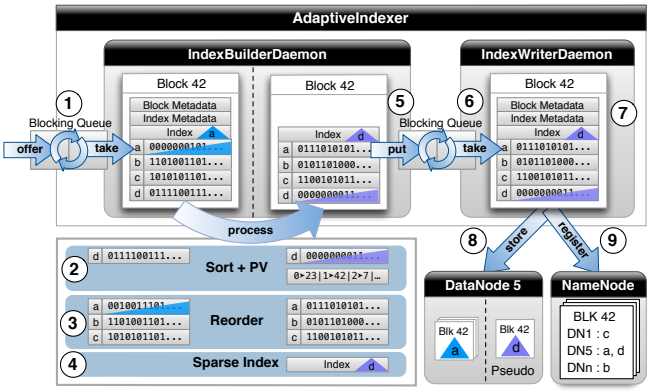


Fig. 4 AdaptiveIndexer internals.

internal components: the *IndexBuilder* and the *IndexWriter*. Figure 4 illustrates the pipeline of these two internal components, which we discuss in the following.

The **IndexBuilder** is a daemon thread that is responsible for creating sparse clustered indexes on data blocks in the data queue. With this aim, the IndexBuilder is constantly pulling one data block after another from the data block queue ①. Then, for each data block, the IndexBuilder starts with sorting the attribute column to index (attribute d in our example) ②. Additionally, the IndexBuilder builds a mapping $\{old\_position \mapsto new\_position\}$ 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 sort order. For instance, in our example of Section 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.

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 article and is subject to ongoing work. A simple implementation could borrow ideas from buffer replacement strategies to attack the problem, e.g. LRU or replacing the least beneficial indexes.

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.

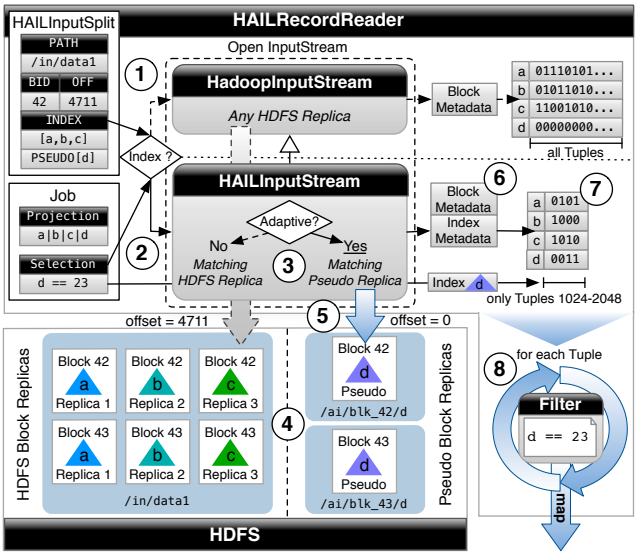


Fig. 5 HAILRecordReader internals.

5.4 HAIL RecordReader Internals

Figure 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 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 /pseudo/blk_42/d⑤. 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).

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.

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 pred-

icates on unfrequent attributes is minimized. On the other hand, MapReduce jobs with selection predicates on frequent attributes quickly converge to a completely indexed copy.

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 6.1). However, let us make two important observations that could make a constant offer rate not desirable for certain users:

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

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

Therefore, we propose an *eager adaptive indexing* strategy to deal with this problem. The basic idea of eager adaptive indexing is to dynamically adapt the offer rate for MapReduce jobs according to the indexing work achieved by previous jobs. In other words, eager adaptive indexing tries to exploit the saved runtime and reinvest it as much as possible into further indexing. To do so, HAIL first needs to estimate the runtime gain (in a given MapReduce job) from performing an index scan on the already created pseudo data block replicas. For this, HAIL uses a cost model to estimate the total runtime, T_{job}, of a given MapReduce job (Equation 1). Table 1 lists the parameters we use in the cost model.

$$T_{job} = T_{is} + t_{fsw} \cdot n_{fsw} + T_{idxOverhead}. \quad (1)$$

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

$$T_{idxOverhead} = t_{idxOverhead} \cdot \min\left(\rho \cdot \left\lceil \frac{n_{blocks}}{n_{slots}} \right\rceil, n_{fsw}\right). \quad (2)$$

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 1 and 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 1) and $t_{idxOverhead}$ (from Equation 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.

Table 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

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

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 article.

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 [40]. 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 [40] 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 [34]. Moreover, one can significantly improve data locality by simply using an adequate scheduling policy (e.g. the Delay Scheduler [46]). 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.

8 Related Work

HAIL uses PAX [3] 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 [12]. In our previous work [34], we showed how to improve over PAX by computing different layouts on the different replicas, but we did not consider indexing. This article fills this gap.

Static Indexing. Indexing is a crucial step in all major DBMSs [19,10,2,8,11]. 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 [45,35,15,33]. 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 [42,6,7, 37]. The idea is to continuously monitor the performance of the system and create (or drop) indexes as soon as it is considered beneficial. Manimal [9,32] 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 [28] 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 [29] 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 [31], concurrency control [20], and robustness [25], but these works are orthogonal to the problem we address in this paper.

Adaptive Merging. Another related work to HAIL is the adaptive merging [21]. 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 [30]. 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 [1] or in a lazy [27] 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, [1] 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 [4]. 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 [4] 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.

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 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 9.4). How does HailSplitting improve end-to-end job runtimes? (We answer this question in Section 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 9.6)

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 [41]. 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 9.3.3). 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 9.3.4).

Systems. We compared the following systems: (1) Hadoop, (2) Hadoop++ as described in [15], and (3) HAIL as described in this article. For HAIL, we disable the HAIL splitting in Section 9.4 in order to measure the benefits of using this policy in Section 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.

9.2 Datasets and Queries

Datasets. For our benchmarks we use two different datasets. First, we use the UserVisits table as described in [39]. This dataset nicely matches Bob's Use Case. We generated 20GB of UserVisits data per node using the data generator proposed by [39]. 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 x 10^{-2})

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

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

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

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

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

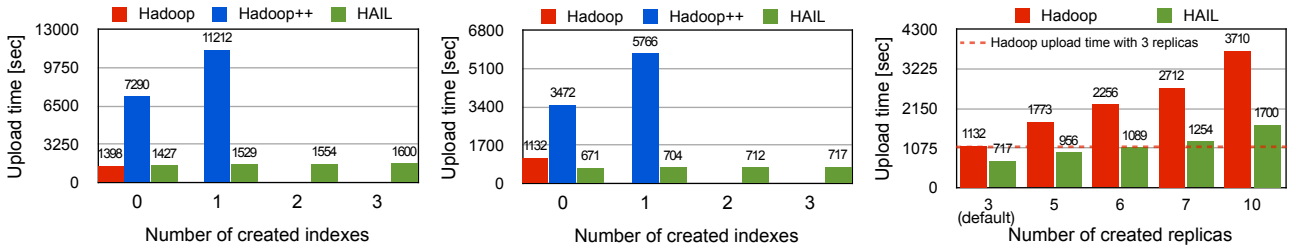
Bob-Q4 (selectivity: 1.7 x 10^{-2})

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

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

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

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



(a) Upload time for UserVisits (b) Upload time for Synthetic (c) Varying replication for Synthetic

Fig. 6 Upload times when varying the number of created indexes (a)&(b) and the number of data block replicas (c)

For the Synthetic dataset, we use the queries in Table 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 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.

9.3 Data Loading

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

9.3.1 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 6(a) shows the results for the UserVisits dataset. We observe that HAIL has a negligible upload overhead of ~2% over standard Hadoop. Then, when HAIL creates one index per replica the overhead still remains very low (at most ~14%). On the other hand, we observe that HAIL improves over Hadoop++ by a factor of 5.1 when creating no index and by a factor of 7.3 when creating one index. This is because Hadoop++ has to run two expensive MapReduce jobs for creating one index. For HAIL, we observe that for two and three indexes the upload costs increase only slightly.

Figure 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.

9.3.2 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 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 and up to a factor of 2.5. More interestingly, we observe that HAIL stores six replicas (and hence it creates six different clustered indexes) in a little less than the same time Hadoop uploads the same dataset with only three replicas without creating any index. Still, when increasing the replication factor even further for HAIL, we see that HAIL has only a minor overhead over Hadoop with three replicas only. These results also show that choosing the replication fac-

Table 3 Scale-up results

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

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

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

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

tor 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.

9.3.3 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(a) (for UserVisits) and in Table 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 decreases (in Table 3(a)) or increases (Table 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.

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

Finally, we observe that the system speedup of HAIL over Hadoop is even better when using physical nodes.

9.3.4 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.

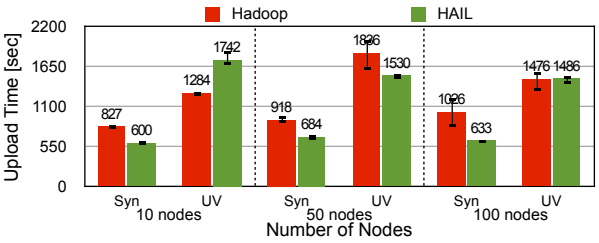
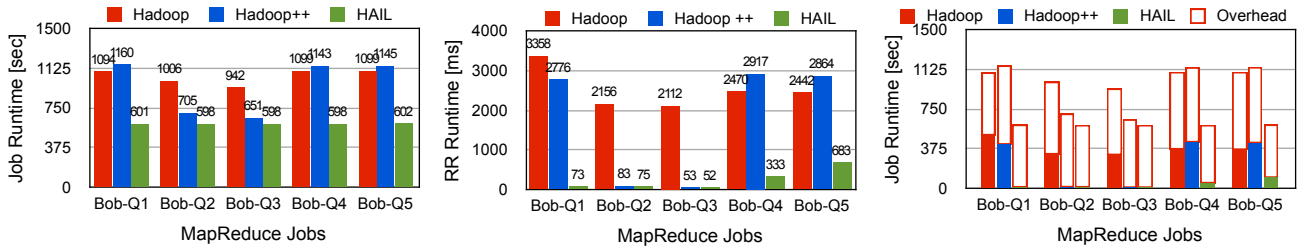


Fig. 7 Scale-out results

Figure 7 shows these results. We observe that HAIL achieves roughly the same upload times for the Synthetic dataset. For the UserVisits dataset, we see that HAIL improves its upload times for larger clusters. In particular, for 100 nodes, we see that HAIL matches the Hadoop upload times for the UserVisits dataset and outperforms Hadoop by a factor up to ~1.4 for the Synthetic dataset. More interesting, we observe that, in contrast to Hadoop, HAIL does not suffer from high performance variability [41]. Overall, these results show the efficiency of HAIL when scaling-out.

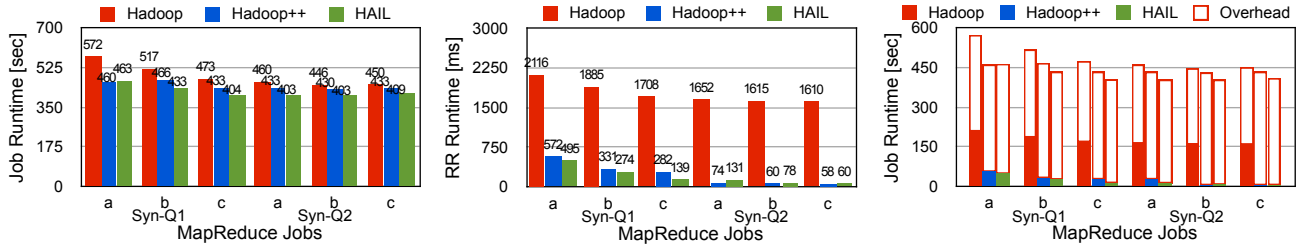
9.4 MapReduce Job Execution

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 7) in order to better evaluate the benefits of having several clustered indexes per dataset. We study the benefits of HailSplitting in Section 9.5.



(a) End-to-end job runtimes (b) Average record reader runtimes (c) Hadoop scheduling overhead

Fig. 8 Job runtimes, record reader times, and Hadoop MapReduce framework overhead for Bob's query workload filtering on multiple attributes



(a) End-to-end job runtimes (b) Average record reader runtimes (c) Hadoop scheduling overhead

Fig. 9 Job runtimes, record reader times, and Hadoop scheduling overhead overhead for `Synthetic` query workload filtering on a single attribute

9.4.1 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 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 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 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 9.5.

9.4.2 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 9(a) and the record reader times in Figure 9(b). We observe in Figure 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 in Figure 9(b),

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

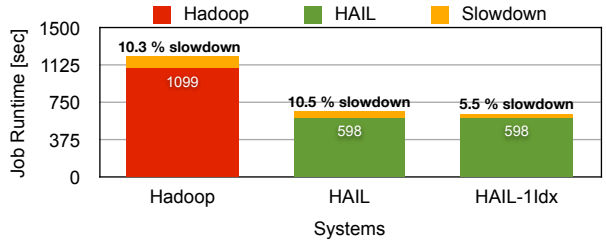


Fig. 10 Fault-tolerance results

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 9(a)) even if query selectivity has a clear impact on the RecordReader times (see Figure 9(b)). As explained in Section 9.4.1, this is due to the overhead of the Hadoop MapReduce framework. We clearly see this overhead in Figure 9(c). In Section 9.5, we will investigate this in more detail.

9.4.3 Fault-Tolerance

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

We perform these experiments as follows: (i) we set the expiry interval to detect that a TaskTracker or a datanode failed to 30 seconds, (ii) we chose a node randomly and kill all Java processes on that node after 50% of work progress, and (iii) we measure the slowdown as in [15], $slowdown = \frac{(T_f - T_b)}{T_b} \cdot 100$, where T_b is the job runtime without node failures and T_f is the job runtime with a node failure. We use two configurations for HAIL. First, we configure HAIL to create indexes on three different attributes, one for each replica. Second, we use a variant of HAIL, coined HAIL-1Idx, where we create an index on the same attribute for all three replicas. We do so to measure the performance impact of HAIL falling back to full scan for some blocks after the node failure. This happens for any map task reading its input from the killed node. 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 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.

9.5 Impact of the HAIL Splitting Policy

We observed in Figures 8(c) and 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 7) and run again the Bob and Synthetic queries on HAIL.

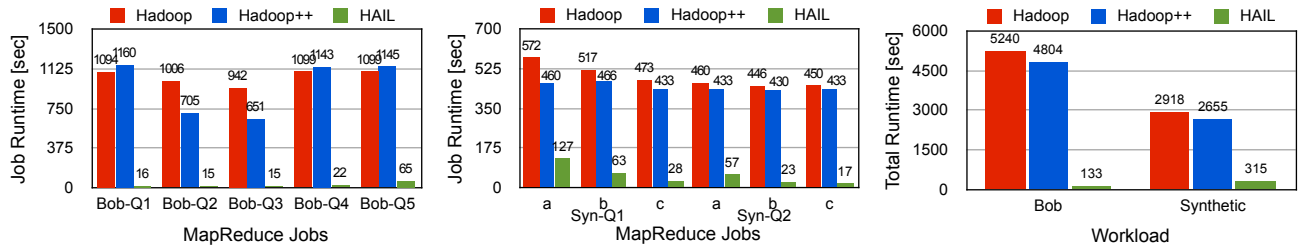
Figure 11 illustrates these results. We clearly observe that HAIL significantly outperforms both Hadoop and Hadoop++. We see in Figure 11(a) that HAIL outperforms Hadoop up to a factor of 68 and Hadoop++ up to a factor of 73 for Bob's workload. This is mainly because the HailSplitting policy significantly 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 9.4.1). For the Synthetic workload (Figure 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 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++.

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

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



(a) Bob queries (b) `Synthetic` queries (c) Total Workload

Fig. 11 End-to-end job runtimes for Bob and Synthetic queries using the HailSplitting policy

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., 10^{i-1} where $i \in [1..10]$. We generate the other five attributes at random. Then, we shuffle all tuples across the entire dataset to have the same distribution across data blocks.

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 selectivity of 0.2% (2.2 million output records). All jobs for both datasets select disjoint ranges to avoid caching effects.

9.6.1 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 12 shows the job runtime for five variants of HAIL for the UserVisits dataset. In Cluster-A, we observe that HAIL has almost no overhead (only 1%) over HAIL ($\rho = 0$) when using an offer rate of 10% (i.e., $\rho = 0.1$). Notice that HAIL ($\rho = 0$) has no matching index available and hence behaves like normal Hadoop with just the binary PAX layout to speed up the job execution. We can also see that the new layout gives us an improvement of at most a factor of two in our experiments. Interestingly, we observe that HAIL is still faster than Hadoop with $\rho = 0.1$ and $\rho = 0.25$. Indeed, the overhead incurred by HAIL increases along with the offer rate used by HAIL. However, we observe that HAIL increases the execution time of JobUV1 by less than factor of two w.r.t. both Hadoop and HAIL without any indexing, even though all data blocks are indexed in a single MapReduce job. We especially observe that the overhead incurred by HAIL scales linearly with the ratio of indexed data blocks (i.e., with ρ), except when scaling from $\rho = 0.1$ to $\rho = 0.25$. This is because HAIL starts to be CPU bound only when offering more than 20% of the data blocks (i.e., from $\rho = 0.25$). This changes when running JobUV1 in Cluster-B. In these results, we clearly observe that the overhead incurred by HAIL scales linearly with ρ. We especially observe that HAIL benefits from using newer CPUs and have better performance than Hadoop for most offer rates. HAIL has only 4% overhead over Hadoop when having $\rho = 1$. Additionally, we can see that the adaptive indexing in HAIL incurs low overhead: from 10% (with $\rho = 0.1$) to 43% (with $\rho = 1$).

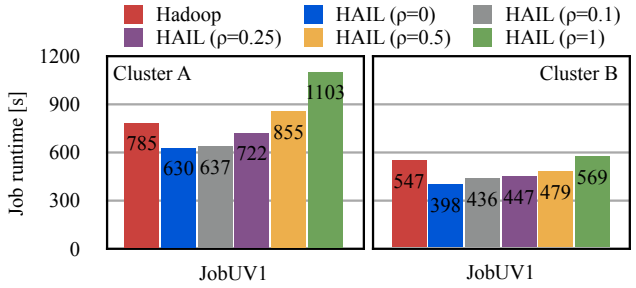


Fig. 12 HAIL Performance when running the first MapReduce job over UserVisits.

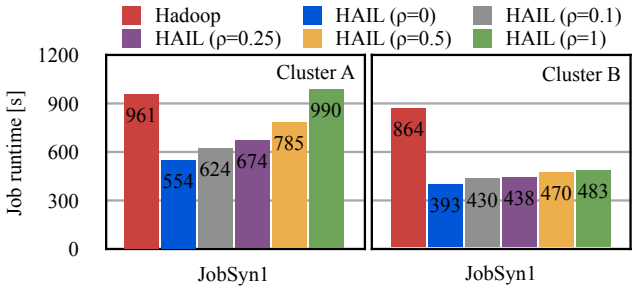


Fig. 13 HAIL Performance when running the first MapReduce job over Synthetic.

Figure 13 shows the job runtimes for Synthetic. Overall, we observe that the overhead incurred by HAIL continues to scale linearly with the offer rate. In particular, we observe that HAIL has no overhead over Hadoop in both clusters, except for HAIL ($\rho = 1$) in Cluster-A (where HAIL incurs a negligible overhead of ~3%). It is worth noting that when using newer CPUs (Cluster-B) adaptive indexing in HAIL has very low overhead as well: from 9% to only 23%.

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

9.6.2 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 14 and 15 show the job runtimes for the UserVisit and Synthetic datasets, respectively. Overall, we clearly see in both computing clusters that HAIL improves the performance of MapReduce jobs linearly with the number of indexed data blocks. In particular, we observe that the higher the offer rate, the faster HAIL converges to a complete index. However, the higher the offer rate, the higher the adaptive indexing overhead for the initial job (JobUV1 and JobSyn1). Thus, users are faced with a natural tradeoff between indexing overhead and the required number of jobs

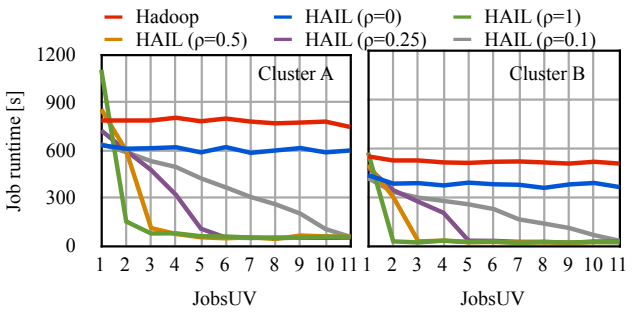


Fig. 14 HAIL performance when running a sequence of MapReduce jobs over UserVisits.

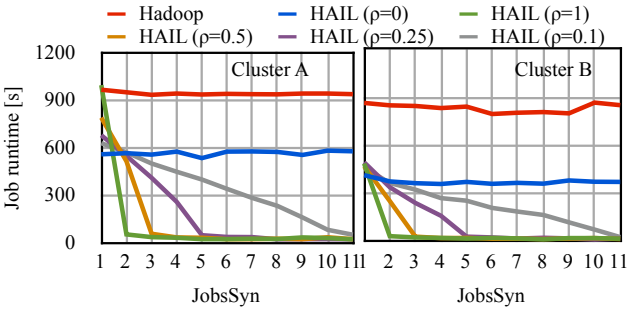


Fig. 15 HAIL performance when running a sequence of MapReduce jobs over Synthetic.

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.

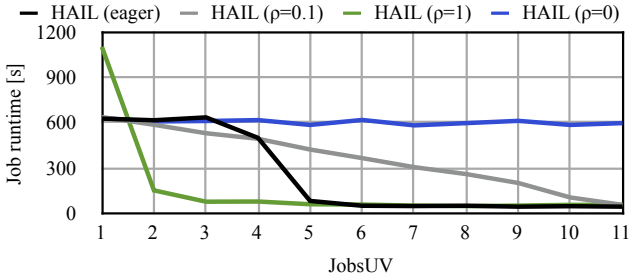


Fig. 16 Eager adaptive indexing vs. $\rho = 0.1$ and $\rho = 1$

9.6.3 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 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.

10 Conclusion

We presented HAIL (Hadoop Aggressive 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++ [15], 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. ~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.

Acknowledgments. Research supported by the Cluster of Excellence on "Multimodal Computing and Interaction" and the Bundesministerium für Bildung und Forschung.

References

1. A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT*, pages 1–10, 2013.
2. S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. *VLDB*, pages 1110–1121, 2004.
3. A. Ailamaki et al. Weaving Relations for Cache Performance. *VLDB*, pages 169–180, 2001.
4. I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD Conference*, pages 241–252, 2012.
5. S. Blanas et al. A Comparison of Join Algorithms for Log Processing in MapReduce. *SIGMOD*, pages 975–986, 2010.
6. N. Bruno and S. Chaudhuri. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB*, pages 499–510, 2006.
7. N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, pages 826–835, 2007.
8. N. Bruno and S. Chaudhuri. Physical Design Refinement: The Merge-Reduce Approach. *ACM TODS*, 32(4), 2007.
9. M. J. Cafarella and C. Ré. Manimal: Relational Optimization for Data-Intensive Programs. *WebDB*, 2010.
10. S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, pages 146–155, 1997.
11. S. Chaudhuri and V. R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, pages 3–14, 2007.
12. S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(1-2):1459–1468, 2010.
13. J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *CACM*, 53(1):72–77, 2010.
14. J. Dittrich and J.-A. Quiané-Ruiz. Efficient Parallel Data Processing in MapReduce Workflows. *PVLDB*, 5, 2012.
15. J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1):518–529, 2010.
16. J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11):1591–1602, 2012.
17. J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: Adaptive Indexing for Context-Aware Information Filters. In *SIGMOD*, pages 215–226, 2005.
18. M. Y. Eltabakh et al. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.
19. S. J. Finkelstein et al. Physical Database Design for Relational Databases. *ACM TODS*, 13(1):91–128, 1988.
20. G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency Control for Adaptive Indexing. *PVLDB*, 5(7):656–667, 2012.
21. G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, pages 371–381, 2010.
22. http://engineering.twitter.com/2010/04/hadoop-at-twitter.html.
23. Hadoop Users, http://wiki.apache.org/hadoop/PoweredBy.
24. F. Halim et al. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.
25. F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.
26. H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4(11):1111–1122, 2011.
27. S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are myResults? In *CIDR*, pages 57–68, 2011.
28. S. Idreos et al. Database Cracking. In *CIDR*, pages 68–78, 2007.
29. S. Idreos et al. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, pages 297–308, 2009.
30. S. Idreos et al. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.
31. S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD Conference*, pages 413–424, 2007.
32. E. Jahani et al. Automatic Optimization for MapReduce Programs. *PVLDB*, 4(6):385–396, 2011.
33. D. Jiang et al. The Performance of MapReduce: An In-depth Study. *PVLDB*, 3(1):472–483, 2010.
34. A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. *SOCC*, 2011.
35. J. Lin et al. Full-Text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics. *MapReduce Workshop*, 2011.
36. D. Logothetis et al. In-Situ MapReduce for Log Processing. *USENIX*, 2011.
37. M. Lühring et al. Autonomous Management of Soft Indexes. In *ICDE Workshop on Self-Managing Database Systems*, pages 450–458, 2007.
38. C. Olston. Keynote: Programming and Debugging Large-Scale Data Processing Workflows. *SOCC*, 2011.
39. A. Pavlo et al. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD*, pages 165–178, 2009.
40. J.-A. Quiané-Ruiz, C. Pinkel, J. Schad, and J. Dittrich. RAFTing MapReduce: Fast recovery on the RAFT. *ICDE*, pages 589–600, 2011.
41. J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1):460–471, 2010.
42. K. Schnaitter et al. COLT: Continuous On-line Tuning. In *SIGMOD*, pages 793–795, 2006.
43. A. Thusoo et al. Data Warehousing and Analytics Infrastructure at Facebook. *SIGMOD*, pages 1013–1020, 2010.
44. T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2011.
45. H.-C. Yang and D. S. Parker. Traverse: Simplified Indexing on Large Map-Reduce-Merge Clusters. In *DASFAA*, pages 308–322, 2009.
46. M. Zaharia et al. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. *EuroSys*, pages 265–278, 2010.