Efficient OR Hadoop: Why not both?

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

Abstract

In this article[1], we give an overview of research related to Big Data processing in Hadoop going on at the Information Systems Group at Saarland University. We discuss how to make Hadoop efficient. We briefly survey four of our projects in this context: Hadoop++, Trojan Layouts, HAIL, and LIAH. All projects aim to provide efficient physical layouts in Hadoop including vertically partitioned data layouts, clustered indexes, and adaptively created clustered indexes. Most of our techniques come (almost) for free: they create little to no overhead in comparison to standard Hadoop.

1 Introduction

In the past years, the Hadoop ecosystem has become the de facto standard to handle so-called Big Data [3]. Hadoop's main components are Hadoop Distributed File System (HDFS) and Hadoop MapReduce.

HDFS allows users to store petabytes of data on large clusters. HDFS provides high fault-tolerance capabilities in an environment where failures of single disks or whole nodes are not the exception but the rule. Hadoop MapReduce allows users to query the data with the simple yet expressive map()/reduce() paradigm without a need for the user to care about parallelization, scheduling, or failover. In contrast to parallel DBMS, Hadoop MapReduce scales easily to very large clusters of thousands of machines. In addition, the upfront investment for using MapReduce is small: no need to use schemas, no integrity constraints, no data cleaning, no normalization, and NoSQL [12]. Moreover, installing and configuring a MapReduce cluster is relatively easy compared to a parallel DBMS: Almost any user with minimal knowledge of Java is able to write and run Hadoop jobs. All of this explains the popularity of MapReduce among non-database people.

On the flip side, Hadoop MapReduce processes MapReduce jobs by default in a non-optimized, scan-oriented fashion — in fact the entire system design is centered around the idea of executing long running jobs. Furthermore, several classes of tasks cannot be expressed naturally with the map()/reduce() paradigm, e.g., joins, iterative tasks, and updates. And finally, the performance of MapReduce is in many cases far from the one of an optimized parallel DBMS.

One might conclude that there is a deep divide among the two classes of systems — parallel DBMS and Hadoop MapReduce. And in fact: in 2009, the database community triggered a heated discussion with a paper by Pavlo et.al. [14] which unfortunately widened that divide. However, given the recent popularity of Hadoop, one might get the idea that there must be a reason for this popularity. If database systems are so great, why isn't everyone using them?

We believe that the key to this discussion is not about the 'new kid on the block' Hadoop solely learning from mature database technology, but that it is key for databases to also learn from Hadoop. Our research question is:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] A short version of this article appeared in the German Database magazine „Datenbankspektrum" in January 2013.

is there a way to preserve the properties of Hadoop while fixing its performance issues AND without turning it into yet another parallel DBMS? As a consequence, in 2009, we started a series of projects investigating this. These projects are Hadoop++, Trojan Layouts, HAIL, and LIAH. We will briefly sketch these projects in the following.

2 Hadoop++

The very first question we had in mind in 2009 was *can we substantially improve the runtimes of MapReduce jobs without touching the Hadoop source code?*

We investigated this question in [4]. In that work, we analyzed the query processing pipeline of Hadoop. The major observation was that Hadoop implements a hard-coded processing pipeline whose structure is very hard to change. However, Hadoop's processing pipeline also provides at least ten different user-defined functions (UDFs) — `map()` and `reduce()` being just two of them. These different UDFs may be exploited to place arbitrary code inside the Hadoop processing pipeline and turn Hadoop into a versatile distributed runtime. We exploited this to *inject* indexing and co-partitioning algorithms into Hadoop. This idea is somehow similar to injecting a trojan, however: this time for good. Hence, we coined the resulting index structure a *trojan index*. For instance, we change the `group()` and `shuffle()` UDFs that control grouping and shuffling. This allows us to create separate indexes for each HDFS block. We evaluated our indexes using the benchmark proposed in [14]. We could show that the query runtimes of Hadoop++ are by up to a factor 20 faster than Hadoop[2]. These performance improvements are possible, as we spend additional time creating indexes and copartitions before executing any MapReduce job, i.e. we create appropriate indexes matching the expected workload and we partition both tables on their join keys to be able to execute joins locally. The time spent for creating those indexes may be considerable [4].

The idea of improving the performance of a closed-source system by injecting code afterwards may also be applied to traditional database systems. In [11] we investigated how to change the data layout of a closed-source row-store into using compressed column-oriented layouts yielding up to a factor of 20 performance improvements.

3 Trojan Layouts

When working in the Hadoop++ project, we could observed that Hadoop MapReduce wastes a significant amount of time reading unnecessary data from disk as it stores data in row layout. Obviously, one could simply store all data in a column layout and hope for similar speed-ups as known from traditional column stores. However, in a distributed system there is a major issue with this approach: column data representing the same rows should be stored physically close as to avoid expensive network I/O for tuple reconstruction. Therefore, the question we had at that point in time was: *How could we change the data layout in Hadoop to be better suitable for analytical query processing?*

We investigated this question in [9]. As a first step, we studied the three most popular data layouts in the literature (namely Row Layout, Column Layout, and PAX Layout) and compared then with the optimal layout[3]. Figure 1 shows results of a simulation of the access costs for different layouts in Hadoop. The horizontal axis depicts the number of referenced attributes for a query. The vertical axis depicts the estimated data access costs. For a Column Layout the costs for network transfer have to be factored in and ruin the overall performance. For Row Layout, the number of referenced attributes does not have an effect. Therefore, a popular layout in the context of MapReduce is the hybrid layout PAX [1]: in this approach, all data inside an HDFS block, i.e.,

[2] A companion paper explores the pitfalls when measuring distributed systems like MapReduce in a cloud environment [18]. Other works look at the efficiency of the Hadoop failover algorithms [16, 15].

[3] Where the optimal layout denotes the data layout that has all attributes required by an incoming query in row format.

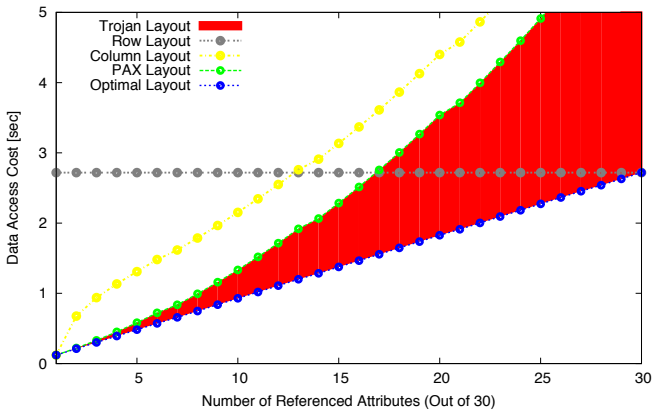


Figure 1: Simulation of data access costs for different data layouts in Hadoop [9].

a horizontal partition of data of 64MB by default, is stored in column layout. This avoids the problems with network I/O for tuple reconstruction and still gives column-like access. However, for some workloads, PAX is not the best layout. The interesting space in Figure 1 is the red area. It depicts data layouts that are enabled by Trojan Layouts and perform better than PAX. Some of those layouts are at the same time better than Row Layout. The latter layouts are the ones we wanted to identify.

In [9] we follow the PAX philosophy in that we keep data belonging to a particular HDFS block on that HDFS block i.e. there is no global reorganization of data *across* HDFS blocks. However, in contrast to PAX, we introduce an important change: Hadoop's Distibuted File Systems (HDFS) stores three copies of an HDFS block for fault-tolerance anyway. All of these copies are byte-identical. We change this to allow the different copies of a *logical* HDFS block to have different *physical*s layouts. As we do not remove any data from the different copies, we fully preserve the fault-tolerance properties of HDFS. At the same time, we are able to optimize the different copies for different types of queries. In [9], we explore this to compute different vertical partitionings for each copy, i.e. we end up with three different vertical partitionings which in turn are then exploited at query time. In summary, Trojan Layouts improves query runtimes both over row layouts and over PAX layouts by up to a factor 5. Recently, in a follow-up work, we investigated the performance trade-offs of vertical partitioning (aka column grouping) in more detail [8]. A major lesson of that study was that the performance of these layouts depend heavily on the database buffer size. In particular, vertically partitioned layouts only pay off for very small input buffers.

However, more interesting research questions remained: *Would it be possible to also store different clustered indexes for each replica rather than different layouts? And what happens if the query workload is not known at data upload time? Is there a way to adaptively create those clustered indexes?*

We will answer these questions in Sections 4 and 5.

4 HAIL

When we observed the big benefits of having different data representations per data block replica in the Trojan Layouts project, we immediately felt the need of applying the same idea for creating a different clustered *index* per data block replica. This triggered two interesting challenges:

How could we instrument the different copies of an HDFS block to use different clustered indexes? And how could we teach Hadoop to create those indexes without paying a high price for expensive index creation jobs as observed for Hadoop++?

For this we started the HAIL (Hadoop Aggressive Indexing Library) project with the goal of answering these questions [5]. HAIL is an enhancement of HDFS and Hadoop MapReduce that keeps the existing physical

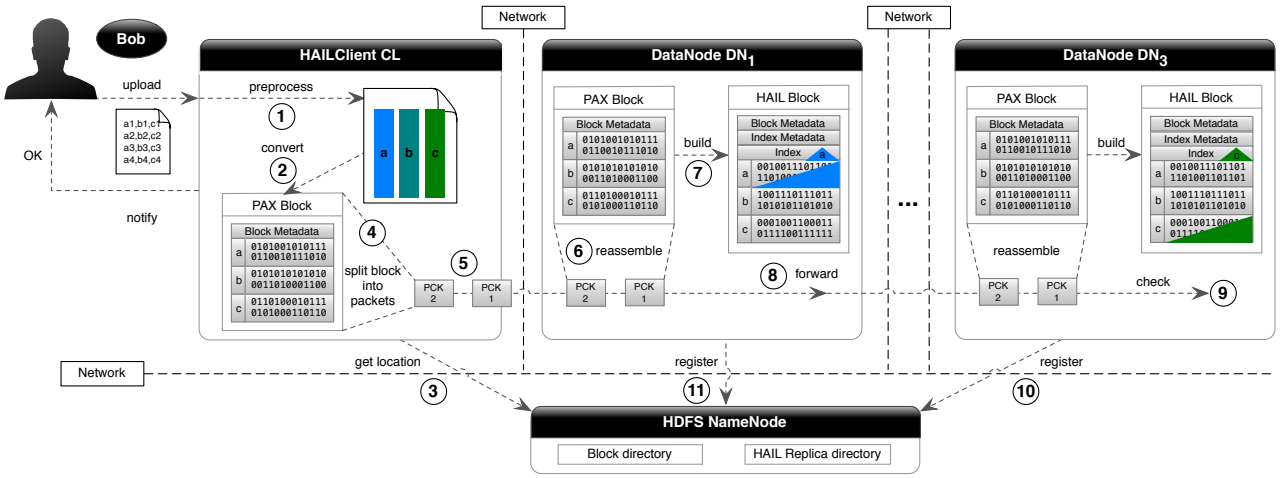


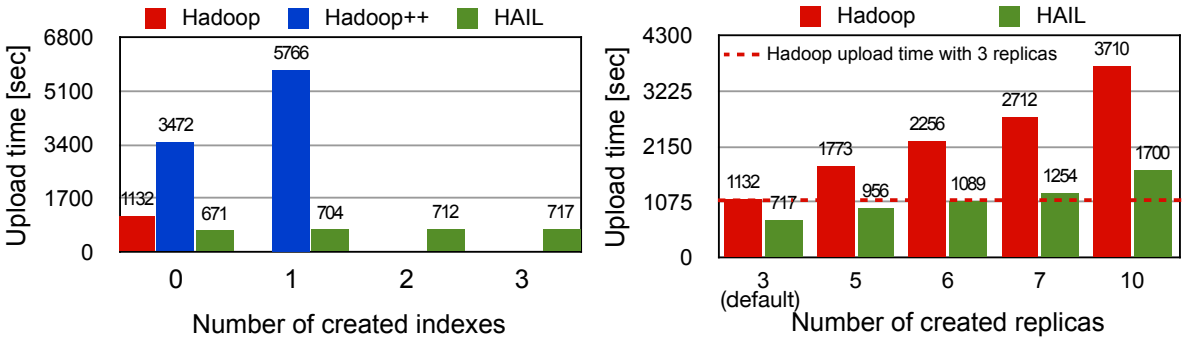
Figure 2: Overview of the HAIL upload pipeline [5]

replicas of an HDFS block in different sort orders and with different clustered indexes. Hence, for a default replication factor of three, three different sort orders and indexes are available for MapReduce job processing. Thus, the likelihood to find a suitable index increases and hence the runtime for a workload improves. 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. Why don't we have high costs at upload time? We basically exploit the unused CPU ticks which 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, as we already parse data to binary while uploading, we often benefit from a smaller binary representation triggering less network and disk I/O.

Let's assume we have a world population table stored in HDFS containing records of type [city, country, population]. If we now want to analyze the population of China, Hadoop MapReduce has to scan the whole world population table and filter for people living in China. While this might be relatively efficient for a country like China, we would still waste our time with reading data that is not needed, and this becomes even more wasteful for a small country such as Luxembourg. If we are now interested in data for a specific city, the traditional Hadoop approach feels like finding a needle in a haystack. This is a typical situation that could be solved with an index, e.g., like the trojan index from Hadoop++. However, creating Trojan indexes is a very costly operation that needs many queries that select on the indexed attribute to amortize [4]. Additionally, the Trojan index will only help when selecting one particular attribute. But what happens if our workload consists of queries selecting on many different attributes like age or name?

Figure 2 shows the HAIL upload pipeline. When uploading a data file to HDFS using the HAIL client, we first analyze the schema of the input (1) and convert the textual data into PAX layout (2). This allows us to save bandwidth, because the binary format is often more space efficient than the textual representation. Like in normal Hadoop, HAIL first asks the Namenode for the locations of all Datanodes that should store a replica of the current block (3). Then, HAIL divides each block into packets (4) and sends them to the first Datanode (the node that was chosen by the Namenode to store the first replica) (5) The first Datanode then reassembles the blocks from the packets (6), sorts the tuples on the index attribute and creates the actual clustered index (7). In parallel, the first Datanode immediately forwards each incoming packet to the next Datanode that stores replica 2. This procedure is repeated for all Datanodes that store a replica until the packets reach the last Datanode. This allows us to create different indexes in parallel on all Datanodes. After reaching the last Datanode, all packets are validated against their checksums (9) Finally, if the blocks could be verified, all Datanodes register their created indexes with the Namenode (10 and 11). With this approach, HAIL even allows us to create more

than three indexes at reasonable costs. Figure 3 shows a comparison of upload times for Hadoop, Hadoop++, and HAIL on our ten-node cluster with a dataset of 130GB. This dataset resembles a typical scientific dataset. A more detailed description of the experiments and the used datasets can be found in our HAIL paper [5].



(a) Upload time for `Synthetic`

(b) Varying replication for `Synthetic`

Figure 3: Upload times when (a) varying the number of created indexex, and (b) the number of data block replicas [5]

In Figure 3(a), we vary the number of indexes from 0 to 3 for HAIL and for Hadoop++ from 0 to 1 (this is because Hadoop++ cannot create more than one index). Notice that we only report numbers for 0 indexes for standard Hadoop as it cannot create any indexes. For all measurements we use three replicas. We observe that HAIL significantly outperforms Hadoop++ by a factor of 5.2 when creating no index and by a factor of 8.2 when creating one index. We observe that HAIL also outperforms Hadoop by a factor of 1.6 even when creating three indexes. This is because HAIL's binary representation of the dataset has a reduced size which allows HAIL to outperform Hadoop even when creating one, two or three indexes.

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. Those results are presented in Figure 3(b). The dotted line marks the time Hadoop takes to upload with the default replication factor of three. We see that HAIL significantly outperforms Hadoop for any replication factor by up to a factor of 2.5. More interestingly, we observe that HAIL stores six replicas (and hence it creates six different clustered indexes) in a little less than the same time Hadoop uploads the same dataset with only three replicas without creating any index. Still, when increasing the replication factor even further for HAIL, we see that HAIL has only a minor overhead over Hadoop with three replicas only.

A more detailed description of the HAIL upload pipeline that discusses some interesting implementation challenges like adapting Hadoop's packeting and checksumming, Namenode extension, index structure, and fault tolerance can be found in our paper [5].

From these result, we can see a huge improvement for indexing overhead when compared to Hadoop++ and conclude that HAIL provides efficient indexing of many attributes with no or almost invisible overhead. But how can we now use the HAIL indexes and what are the corresponding improvements in terms of query performance? There are at least three options:

1. We can analyze the user-provided map()-function using static code analysis. Then we rewrite the map()-function automatically against our indexes. This approach is fully user transparent. This type of code analysis has already been successfully done in [2] and could be extended to exploit HAIL indexes as well.

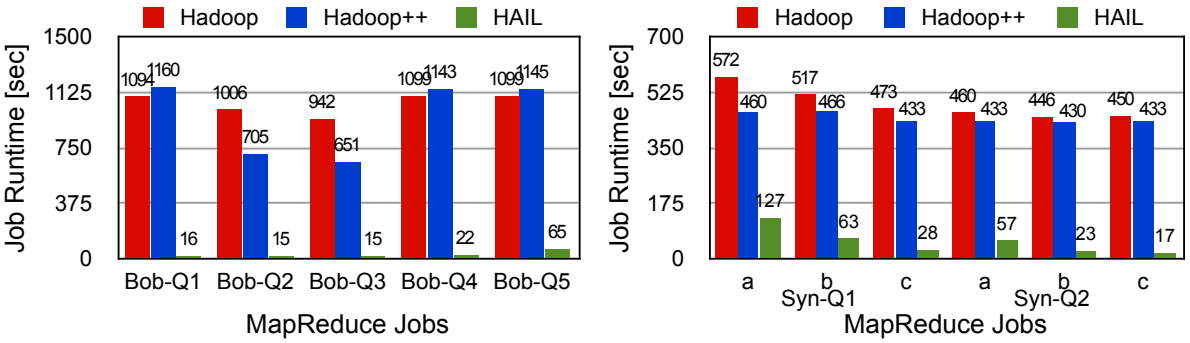
2. We allow users to annotate the map-functions slightly. This approach is not fully user transparent yet minimally invasive. A simple example would be to find the names of all people living in Luxembourg. If

we assume that 'name' is the first attribute and 'country' is the second attribute in the world-population dataset, we simply annotate the map function in Java with

```
@HailQuery(filter="@2='Luxembourg'",
           projection={@1}).
```

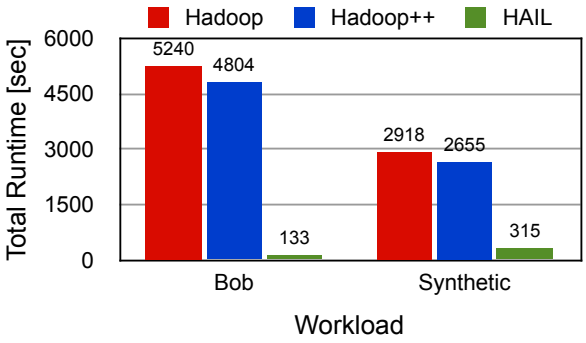
This has the effect that the dataset is pre-filtered and only the attribute name from tuples where country equals to Luxembourg are passed to the map function.

3. The third approach is to modify the applications sitting on top of HDFS or Hadoop MapReduce. As HAIL is a replacement for HDFS, user transparency may be achieved by modifying any software layer on top. For instance, Hive [6] and Pig [13] output machine-generated MapReduce programs; Impala [7] operates directly on flat HDFS files. For these systems, it would be straight-forward to change their MapReduce program generation to exploit HAIL indexes — similar to changing a DB-optimizer to create physical plans using index access paths.



(a) Bob queries

(b) `Synthetic` queries



(c) Total workload

Figure 4: End-to-end job runtimes for two different workloads [5]

Figure 4 illustrates the query performance of HAIL compared to Hadoop and Hadoop++. We clearly observe that HAIL significantly outperforms both Hadoop and Hadoop++. We see in Figure 4(a) that HAIL outperforms Hadoop up to a factor of 68 and Hadoop++ up to a factor of 73 for a log analysis workload (Bob queries). For a Synthetic workload (Figure 4(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 4(c) that using HAIL we can run all 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++.

When developing HAIL we learned that the high scheduling overhead of MapReduce tasks is a severe problem when improving the performance of block accesses. All improvements can be eaten up by this overhead. HAIL reduces this overhead significantly using a novel splitting policy at query time (HAIL scheduling). At its core, HAIL scheduling assigns multiple index accesses to a single map task. Thus, we avoid the Hadoop MapReduce overheads for scheduling multiple map waves (see [5] for details and also failover experiments). Overall, using HAIL scheduling we achieve the performance seen in Figure 4.

5 LIAH

One of the disadvantages of HAIL is that the performance of an incoming MapReduce job can only be improved if HAIL creates indexes on the "right" attributes during upload. The set of "right" indexes may be computed by a what-if-analysis. However, what-if-analyzes requires the workload to be known *before* upload — this might not always be the case. Therefore, another challenge remains: *What happens if we do not know the workload beforehand? How can we teach HAIL to adapt its indexes to the workload?*

For this we propose LIAH (Lazy Indexing and Adaptivity in Hadoop) [17] to improve the total runtime of those tasks dramatically.

LIAH effectively piggybacks adaptive index creation on the existing MapReduce job execution pipeline. In particular, we show how to parallelise indexing with ongoing computations of map tasks and disk I/O. All this without any additional data copy in main memory and minimal synchronisation. A particularity of LIAH is that we always index a data block entirely, i.e. in a single pass. As a result, LIAH allows map tasks of future jobs to perform index accessess. In addition, it also frees future map tasks from costly extra I/O-operations for refining indexes. This is in contrast to existing work on adaptive indexing which requires a large number of queries in order to fully refine the indexes. In addition, these works are not designed to preserve the failover properties of HDFS. Hence we had to come up with new ways to enable adaptivity in a distributed system like Hadoop.

LIAH may be used either on top of an unmodifed Hadoop installation or on top of HAIL. This means, it does not matter whether we already have indexes on the data. Similarly to existing adaptive indexing approaches, we analyze the incoming map()-function for its access patterns — using the same analysis techniques used at query time for HAIL and already explained above (code analysis, annotations or application information). Based on these access patterns, we decide which indexes to create. As an ongoing map()-function not finding a suitable index has to scan all data anyways, we simply piggy back on the I/O performed for this scan. However, we do not create indexes for each block in the input, but rather index only one out of ρ blocks. Here, ρ is the *offer rate* determining the fraction of blocks being indexed for a given MapReduce-job, e.g. for $\rho = 1/3$ we require three MapReduce-jobs to adaptively create indexes on all blocks. Notice that $\rho = 1/3$ does *not* imply that the input file is fully read three times. This is because as the first MapReduce-job indexes $1/3$ as a side-effect, the second MapReduce-job may already perform index access on the indexed blocks, i.e. for $1/3$ of the blocks we perform index accesses, for $2/3$ we perform scans.

Many more indexing options exist including *Eager Indexing*, i.e. throttle the adaptive indexing effort to a user-defined SLA; and *Lazy Projection*, i.e. create clustered indexes only for a subset of the attributes, then recluster the missing attributes lazily on demand. We encourage the reader to see [17] for details.

Here we just want to briefly show some performance results of our techniques. Figure 5 shows the job runtimes for Hadoop, HAIL (without precreated indexes) and LIAH (using different offer rates ρ). Overall, we clearly see in both computing clusters that LIAH improves the performance of MapReduce jobs linearly with the number of indexed data blocks.

Figure 5 shows the job runtimes for the `UserVisit` dataset for two different clusters. Overall, we clearly see in both computing clusters that LIAH 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 LIAH converges to a complete index. However, the higher the offer rate, the higher the adaptive indexing overhead for the initial job

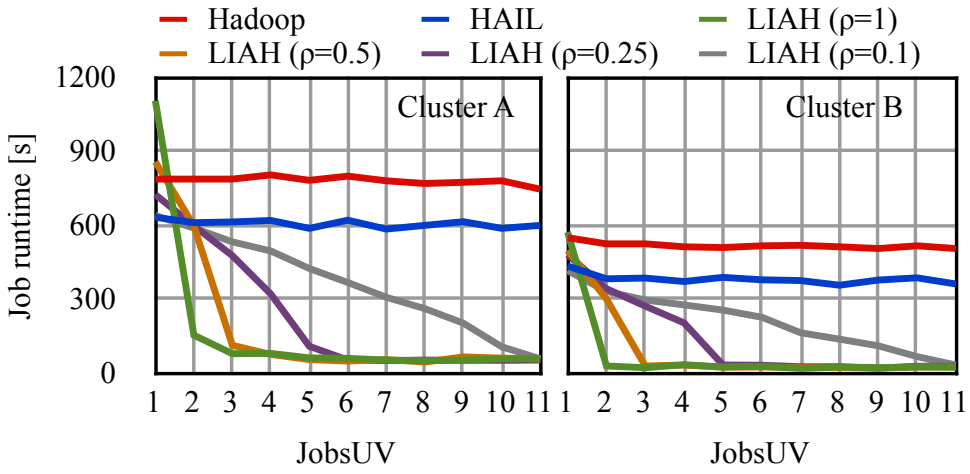


Figure 5: LIAH performance when running a sequence of MapReduce jobs over `UserVisits` [17]

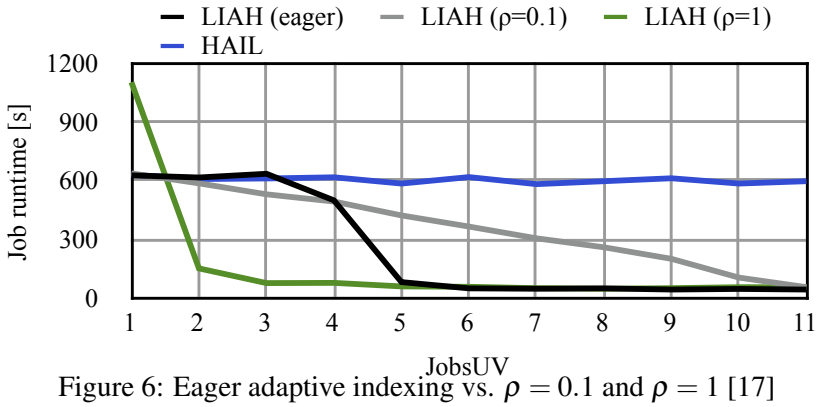


Figure 6: Eager adaptive indexing vs. $\rho = 0.1$ and $\rho = 1$ [17]

(`JobUV1`). Thus, users are faced with a natural tradeoff between indexing overhead and the required number of jobs to index all blocks. But, it is worth noting that users can use low offer rates (e.g. $\rho = 0.1$) and still quickly converge to a complete index (e.g. after 10 job executions for $\rho = 0.1$). In particular, we observe that after executing only a few jobs LIAH already outperforms Hadoop and HAIL (without appropriate indexes) significantly. As soon as LIAH converges to a completely indexed dataset, LIAH significantly outperforms HAIL by up to a factor of 24 and Hadoop by up to a factor of 32.

Figure 6 show the result for eager adaptive indexing. Here we adapt the offer rate ρ such that the execution time of LIAH is less than or equal to that of standard HAIL. As expected, we observe that LIAH (eager) has the same performance as LIAH ($\rho = 0.1$) for `JobUV1`. However, in contrast to LIAH ($\rho = 0.1$), LIAH (eager) keeps its performance constant for `JobUV2`. This is because LIAH (eager) automatically increases ρ from 0.1 to 0.17 in order to exploit saved runtimes. For `JobUV3`, LIAH (eager) still keeps its performance constant by increasing ρ from 0.17 to 0.33. Now, even though LIAH (eager) increases ρ from 0.33 to 1 for `JobUV4`, LIAH (eager) now improves the job runtime as only 40% of the data blocks remain unindexed. As a result of adapting its offer rate, LIAH (eager) converges to a complete index only after 4 jobs while incurring almost no overhead over HAIL. From `JobUV5`, LIAH (eager) ensures the same performance as LIAH ($\rho = 1$) since all data blocks are indexed, while LIAH ($\rho = 0.1$) takes 6 more jobs to converge to a complete index. See [17] for more experimental results.

6 Lessons Learned and Conclusion

We learned that it is possible to introduce indexing in Hadoop without changing its source-code (Hadoop++). We also saw that substantial performance improvements are possible when HDFS is changed to support multiple physical layouts (Trojan Layouts). An interesting challenge was to instrument HDFS to provide efficient index creation and query processing at the same time (HAIL). We also explored how to teach HAIL to create indexes adaptivley (LIAH). Future work aims at generalizing the different projects into a common storage optimizer as discussed in [10].

Yes, parallel DBMS and Hadoop MapReduce are very different systems — at first sight. In comparison, Hadoop is a young system compared to parallel DBMS and can still be improved in many different ways. The Hadoop ecosystem provides an opportunity for the database community to broaden the impact of our research. It is also an opportunity to revisit design decisions taken in the past and take different routes than the ones we took before. In this spirit, we believe that it will be important to teach efficiency to Hadoop without turning it into yet another parallel DBMS.

Acknowledgments. Research partially supported by BMBF. We would like to thank all authors and team members of the Hadoop++, Cloud Variance, RAFT, Trojan Layouts, HAIL, and LIAH projects for their support.

References

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

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

[3] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient Big Data Processing in Hadoop MapReduce. *PVLDB*, 5(12):2014–2015, 2012.

[4] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1-2):515–529, 2010.

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

[6] Hive, http://hive.apache.org.

[7] Cloudera Impala, https://github.com/cloudera/impala.

[8] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. A Comparison of Knives for Bread Slicing. *PVLDB*, 6(to appear), 2013.

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

[10] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. WWHow! Freeing Data Storage from Cages. *CIDR*, 2013.

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

[12] say No! No! and No! (=NoSQL Parody), `http://youtu.be/fXc-QDJBXpw`.

[13] Pig, http://pig.apache.org.

[14] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. *SIGMOD*, pages 165–178, 2009.

[15] Jorge-Arnulfo Quiané-Ruiz, Christoph Pinkel, Jörg Schad, and Jens Dittrich. RAFT at Work: Speeding-Up MapReduce Applications under Task and Node Failures. In *SIGMOD*, pages 1225–1228, 2011.

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

[17] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards Zero-Overhead Adaptive Indexing in Hadoop. `http://arxiv.org/pdf/1212.3480v1.pdf` [cs.db], TR 12/2012.

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