Mosquito: Another One Bites the Data Upload STream

Stefan Richter Jens Dittrich Stefan Schuh Tobias Frey

Information Systems Group
Saarland University
infosys.cs.uni-saarland.de

ABSTRACT

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

1. INTRODUCTION

Hadoop is a popular data processing engine in the context of cloud computing, NoSQL, and Big Data. In the past years, the DB community has taught efficiency to Hadoop MapReduce and its distributed file systems HDFS in several ways. An important family of techniques has investigated on how to use better physical layouts [7], clustered indexes [3, 4], and adaptive indexes [13, 12]. Though these technique can always be implemented in a traditional way by using Hadoop MapReduce jobs to create indexes on top of its file system HDFS — similar to a traditional DBMS using a physical design engine on top of a file system, this approach has a severe drawback: data is read and written several times across the two layers. As HDFS is agnostic about Hadoop MapReduce, considerable time is wasted doing things twice in the two layers that could be combined effectively if the two layers were a single layer. This is prohibitevly expensive in an environment handling Petabytes of data. It makes physical design expensive. And it dramatically increases MapReduce job latencies, be it at data upload or be it at MapReduce job execution. In the context of a distributed system the separation of data storage and data processing into two layers is a pain.

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

An obvious idea to fix this problem is the other extreme of a system design: use many interfaces and layers. Whatever technique you want to teach to Hadoop, implement them in another layer: make sure you implement them against system- or UDF-interfaces [3, 9], i.e. whatever you do, stick to the existing interfaces. However, such systems quickly become inefficient. In addition, the impact of your optimizations is limited by the interfaces that were provided by those systems in the first place, even when using UDFs [3, 9]. Moreover, even though these approaches do not need to touch the source code of the software layer underneath, the limitedness of the system interfaces often forces you to reimplement considerable parts of its functionality. For instance, the recently proposed [6] does not need to change HDFS. However, it needs to reimplement failover, data placement, as well as load balancing. This reduces the role of HDFS to a simple local file system with network access.

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

2. MOSQUITO OVERVIEW

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

2.1 Aggressive Indexing

Mosquito Aggressive Indexing allows users to efficiently create different clustered indexes over terabytes of data as a side-effect of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.
Proceedings of the VLDB Endowment, Vol. 6, No. 12
Copyright 2013 VLDB Endowment 2150-8097/13/10... $ 10.00.

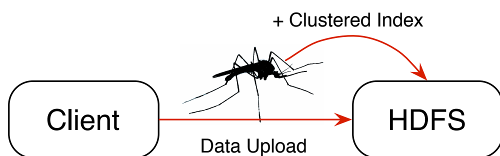


Figure 1: Mosquito aggressive indexing as a side-effect of HDFS data upload.

uploading their dataset to HDFS. Mosquito can support different sort orders (and layouts), one for each physical replica of the data *without* affecting Hadoop's data placement and failover properties. Like this Mosquito can fully emulate HAIL [4].

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

2.2 Adaptive Indexing

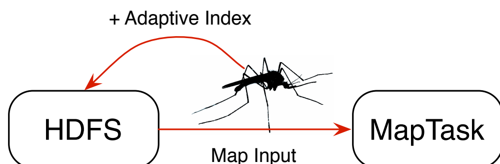


Figure 2: Mosquito adaptive indexing as a side-effect of a MapTask execution.

Mosquito Adaptive Indexing allows users to efficiently create different clustered indexes over terabytes of data as a side-effect of query processing. In contrast to adaptive indexing in main memory [5], for every MapTask we collect a subset of the HDFS blocks and create full indexes on those blocks. In addition, again, this also allows us to keep all replicas in sync and keep HDFS' failover properties. Like this Mosquito can fully emulate LIAH [13].

Our motivation for Mosquito adaptive indexing is a scenario where users want to apply selections (using Mosquito annotations) on attributes that where not indexed at data upload time. For example, this can easily happen when the selection criteria are hard to predict in advance or whenever workloads change over time. Mosquito adaptive indexing sits on top of Hadoop's MapReduce job execution. The core idea is to create missing but promising indexes as byproducts of full scans in the map phase of MapReduce jobs. Similar to aggressive indexing, our goal is again to create additional indexes without significant overhead on individual job runtimes. Mosquito piggybacks on another procedure that is naturally reading data from disk to main memory. This allows Mosquito to completely save the data read cost for adaptive index creation. Second, as map tasks are usually I/O-bound, Mosquito can again exploit unused CPU time for computing clustered indexes in parallel to job execution. Figure 2 illustrates the core concept of the Mosquito adaptive indexing pipeline.

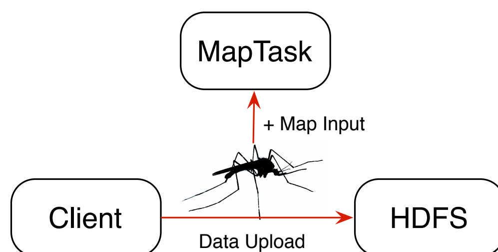


Figure 3: Aggressive Map Execution as a side-effect of HDFS data upload.

2.3 Aggressive Map Execution

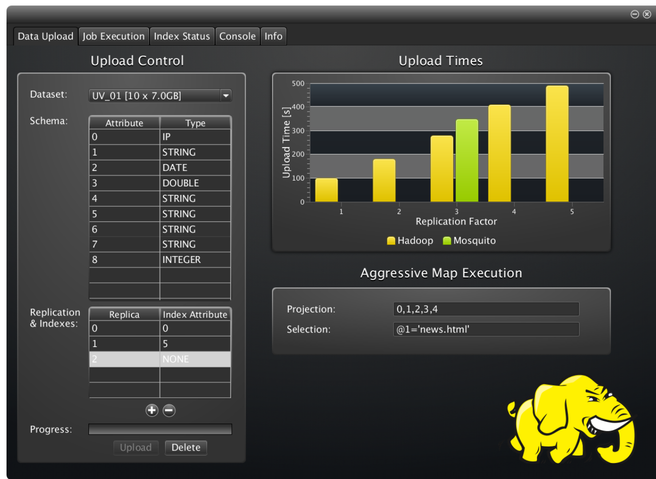
Mosquito Aggressive Map Execution allows users to efficiently run one or several map phases as a side-effect of uploading data into HDFS. This means each data node receiving data to store already executes a MapTask on that data before writing it to disk. This is interesting for cases where the map-functions to execute are already known at data upload time. Like this Mosquito can be run in 'NoHadoop'-mode [11].

Mosquito Aggressive Map Execution allows users to execute MapReduce jobs while uploading their dataset to HDFS. This means that users can immediately start analyzing their data instead of waiting for their initial upload to finish. Our Aggressive Map Execution is illustrated in Figure 3 and works on top of the HDFS upload pipeline as follows: (1) The user uploads her data with the HDFS upload command and additionally provides one (or a set of) MapReduce job(s) to execute on that data. (2) The client splits the data into blocks and these blocks into packets. For each block, the client sends those packets to the first data node for storage. (3) On each data node, those packets are persisted on local disk and forwarded to the next data node if applicable, just like in normal HDFS. However, in parallel, one data node that stores a block replica is chosen by the job scheduler to reassemble this data block from the packets in main memory and spawn a new map tasks for the provided job. Since data block replicas are distributed over the cluster, the scheduler can parallelize the tasks on the cluster similar to normal Hadoop. Whenever a map tasks fails, it is rescheduled after the upload phase was completed. As a result, our system can save the complete read costs of the map tasks while preserving full failover properties. (4) After the upload (and hence the map phase) is completed, Mosquito runs a reduce phase as in normal Hadoop. Notice, that Mosquito could also be used to chain multiple MapReduce jobs, e.g. for iterative computations: in the end of a reduce task, when the output of a job is written back to HDFS, the consecutive map task may already be executed using the technique of Aggressive Map Execution.

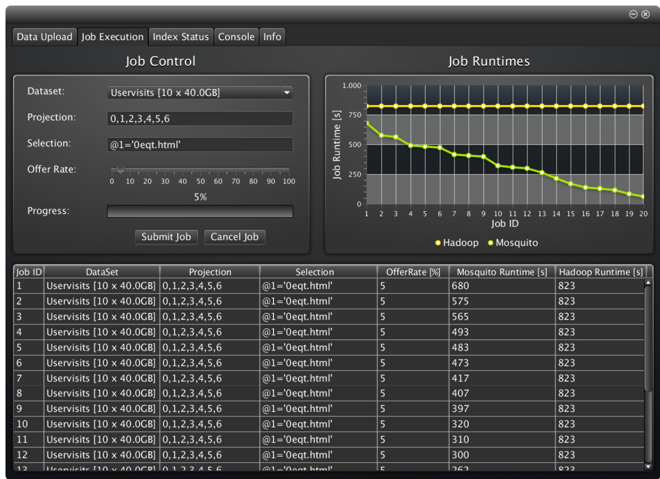
We are currently evaluating this technique in more detail [11].

3. DEMONSTRATION AND USE CASES

Mosquito offers interfaces to plug user defined operations on top of ongoing data movement in Hadoop clusters. As a result, the Mosquito framework acts as a flexible platform that greatly simplifies the realization of many optimization techniques for Hadoop's data storage and job execution pipeline, such as indexing, layout transformation or ad-hoc job execution. In the following, we will describe our demo setup (Section 3.1) and three use cases that demonstrate possible Mosquito applications (Section 3.2).

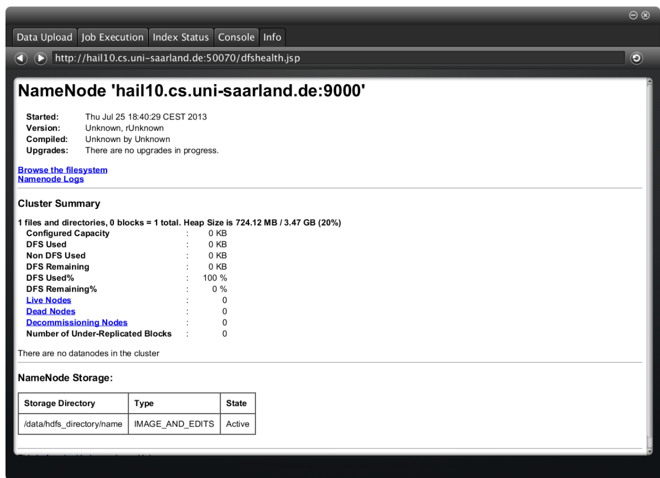


(a) Dataset upload



(b) Job execution

(c) Indexing status



(d) Browse cluster information

Figure 4: Graphical User Interface of Mosquito

3.1 Demo Setup

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

3.2 Use Cases

Mosquito emulating HAIL. In our first use case, we leverage Mosquito to emulate HAIL [4]. The goal of this demo scenario is to (i) illustrate how Mosquito can easily create several clustered indexes in parallel to uploading a dataset to HDFS and (ii) to show how such indexes can dramatically decrease the runtimes of selective MapReduce jobs. This scenario represents a typical analytical use case, where the user wants to create one or more indexes on a large dataset, e.g. a weblog, and afterwards exploits the indexes to speed up queries. We invite the audience to specify the clustered indexes to create and to compare the upload times for the weblog dataset of Mosquito with the ones of standard HDFS. Then, the audience can edit and enhance MapReduce jobs with Mosquito annotations and run the jobs on the previously uploaded dataset. Finally, we report the runtime improvements of Mosquito in comparison to normal Hadoop MapReduce.

Mosquito emulating LIAH. In our second use case, we configure Mosquito to emulate adaptive indexing as presented in LIAH [13]. In this scenario, we show how Mosquito can be used to realize pluggable adaptive indexing capabilities on top of Hadoop MapReduce. In more detail, we show how Mosquito exploits running map tasks to incrementally build missing indexes with minimal or no runtime overhead per job. This approach proved useful in applications where the query workload is unknown at data upload or changes over time. Our Mosquito GUI, as shown in Figure 4, allows the audience to edit and schedule sequences of annotated MapReduce jobs. Additionally, the audience can configure runtime parameters, such as the offer rate[1]. We plot the runtimes for the job sequence executed on Mosquito and standard Hadoop. Thereby, the audience can observe the gradual runtime improvement of adaptive indexing. Furthermore, an index map visualizes the progress of index creation

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

while executing the job sequence.

Mosquito running NoHadoop[2]**.** Our third use case for the Mosquito framework is NoHadoop [11], an approach for ad-hoc job execution on top of data uploads to HDFS. With NoHadoop, users no longer have to wait for their data being uploaded to HDFS before running their MapReduce jobs. Instead, they can immediately start running MapReduce jobs while their data is s being uploaded to HDFS. Consequently, NoHadoop eliminates the upload-to-job time, which is the fundamental measure for the delay before Hadoop can actually start to execute jobs on new data. We encourage the audience to benchmark Mosquito NoHadoop against normal Hadoop for one or more jobs. Overall, we show the abilities of Mosquito NoHadoop to reduce upload-to-job time as well as total runtimes of typical MapReduce workflows dramatically.

4. RELATED WORK

One core principle of Mosquito is to plug in additional operations to existing data streams to eliminate redundant data access. This idea is remotely related to shared scans [14, 10]. However, the focus of shared scans is multi-query optimization, i.e. sharing input data or results among *queries* with common sub-expressions rather than sharing data streams *across layers*. Mosquito offers a much more general interface to piggyback data streams of data-producing processes (such as data uploads or task execution) in Hadoop to arbitrary data-consuming processes (like indexing, arbitrary tasks or layout optimizations). Thus, we see Mosquito as a framework to integrate various optimizations easily and efficiently into Hadoop. In Mosquito, shared scans are only one possible special case.

The recently proposed Cartilage [6] is a framework that aims to provide a more flexible storage layer on top of HDFS along the same lines as [8]. According to the authors of [6], the long term goal is to give users full control over many aspects of data storage, including partitioning, replication, layout, placement, and sort orders. However, as explained in the Introduction, Cartilage needs to reimplement fundamental HDFS features on a higher layer including replication, failover, namenode, etc. Thus, even though Cartilage does not change the source code of HDFS, this approach needs to duplicate considerable parts of HDFS's functionality and restricts its role to a mere local file system with remote access.

The recently proposed Invisible Loading [1] gradually loads data to a DBMS as a side-effect of a map task. Thus, [1] is another proposal for a two-layer approach. On first sight this looks similar to our Aggressive Indexing use-case of Mosquito (see Section 2.1). However, there is a major difference: in Invisible Loading the data is moved *across* systems from HDFS to a distributed DBMS. This means the data is not only kept three times in HDFS, but possibly also several times in the DBMS. In addition, the implications for failover are unclear in that approach.

NoDB [2] provides query processing on top of raw text data. As query processing on raw text is inefficient, that system creates positional maps to positions in raw text and/or creates a data value cache to avoid text access at query time alltogether. All of this is done as a byproduct of query processing. Again, similar to Invisible Loading, this is another special case of populating another system layer, i.e. PostgreSQL on top of a local file system, as a side-effect of query processing. Hence, another appropriate name for the NoDB-approach, which is implemented in the open source DBMS PostgreSQL, might be 'AdaptiveETL'. In contrast to NoDB which focusses on this data loading use case, we provide a more general framework for piggybacking on data streams. Moreover, our main goal is to improve Hadoop MapReduce and HDFS rather than single instance DBMSs. However, as Hadoop is actually often used for ETL-style jobs, the two philosophies might be combined in the long run.

Acknowledgments. Research partially supported by BMBF.

[2]Name inspired by http://youtu.be/fXc-QDJBXpw.

5. 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] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB in Action: Adaptive Query Processing on Raw Data. *PVLDB*, 5(12):1942–1945, 2012.

[3] 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.

[4] 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.

[5] 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.

[6] A. Jindal, J. Quiané-Ruiz, and S. Madden. CARTILAGE: Adding Flexibility to the Hadoop Skeleton. *SIGMOD Demo*, 2013.

[7] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. *SOCC*, 2011.

[8] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. WWHow! Freeing Data Storage from Cages. *CIDR*, 2013.

[9] A. Jindal, F. M. Schuhknecht, J. Dittrich, K. Khachatryan, and A. Bunte. How Achaeans Would Construct Columns in Troy. *CIDR*, 2013.

[10] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB*, 3(1):494–505, 2010.

[11] S. Richter and J. Dittrich. NoHadoop: Three Extra Strong Indexes and Four Map Phases to Go, Please! in preparation.

[12] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards Zero-Overhead Static and Adaptive Indexing in Hadoop. *VLDB Journal*, 2013.

[13] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards Zero-Overhead Adaptive Indexing in Hadoop. arXiv:1212.3480 [cs.db], TR 12/2012.

[14] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, pages 723–734, 2007.