CARTILAGE: Adding Flexibility to the Hadoop Skeleton

Alekh Jindal
CSAIL, MIT
alekh@csail.mit.edu

Jorge Quiané-Ruiz
QCRI, Qatar Foundation
jquianeruiz@qf.org.qa

Samuel Madden
CSAIL, MIT
madden@csail.mit.edu

ABSTRACT

Modern enterprises have to deal with a variety of analytical queries over very large datasets, also commonly referred to as big data analytics. In this respect, Hadoop — an open-source implementation of MapReduce paradigm — has gained much popularity in the recent years. Hadoop scales to thousand of nodes and terabytes of data. However, Hadoop suffers from poor performance, especially in I/O performance, over a variety of analytical tasks. Several works have proposed alternate data storage for Hadoop in order to improve the query performance. However, many of these works end up making deep changes in Hadoop or HDFS. As a result, they are (i) difficult to adopt by several users, and (ii) not compatible with future Hadoop releases. Furthermore, these works still do not exploit full flexibility of data storage and hence there is still a lot more potential to improve query performance.

In this paper, we present CARTILAGE, a comprehensive data storage framework built on top of HDFS. CARTILAGE allows users full control over their data storage, including data partitioning, data replication, data layouts, and data placement. As a result, users can implement several efficient query plans in Hadoop and improve the performance of their analytical queries significantly. Furthermore, CARTILAGE can be layered on top of an existing HDFS installation. This means that Hadoop, as well as other query engines, can readily make use of CARTILAGE. We describe several use-cases of CARTILAGE and propose to demonstrate the flexibility and efficiency of CARTILAGE through a set of novel scenarios.

1. INTRODUCTION

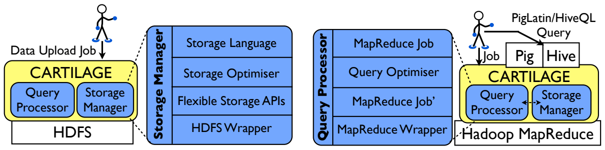
Hadoop is a parallel data processing framework that has gained much popularity in the recent years as it allows non-expert users to run their analytical queries over big data and clusters. However, today, it is well known that the simplicity of Hadoop comes at the price of poor performance [14]. One of the reasons is because Hadoop has poor I/O performance. By default, Hadoop stores all data in an identical byte-copy of input datasets, which typically are in row layouts, and has a scan-oriented data processing (i.e. large datasets are scanned entirely). Therefore, in the last two years, several works have proposed alternate data layouts, such as CFile [12], Cheetah-layout [3], RCFile [8], CIF [7], and Trojan Layouts [9], and alternate data access methods, such as HadoopDB [1], Trojan Indexes [5], Full text indexes [11], and HAIL [6]. However, all these approaches have two main problems:

(1.) These approaches are limited to data layouts and indexes. But, there is still lot more flexibility that can be harnessed, such as compression, co-location, co-grouping, and heterogenous data partitioning (i.e., different partitioning key for different replicas). In particular, we should be able to consider more than one of these aspects at the same time. Furthermore, default data storage features in Hadoop such as data replication, data partitioning, and data distribution gives rise to newer challenges as well as research opportunities, e.g., multiple sort orders at the same time.

(2.) Many of these approaches of alternate data layouts and access paths require deep changes in Hadoop or HDFS. However, with the growing mass of Hadoop users, deep changes in the Hadoop framework are bad. This is because now users must replace standard Hadoop or HDFS with the deeply changed one. This is just not an option for many users. Furthermore, with deep changes we make our software incompatible with future Hadoop versions. As a result, it is difficult to support the software stack built on top of Hadoop, e.g. Pig, Hive, with the deeply changed Hadoop or HDFS.

This calls for an approach that provides full flexibility of data storage in Hadoop, while making minimal or no changes to the Hadoop infrastructure. Our idea is to have a flexible storage layer on top of HDFS, which (i) on the one hand exploits default features of HDFS (such as data replication, data partitioning, and data distribution) and (ii) on the other hand exposes full data storage flexibility (using UDFs) to Hadoop users. As a result, users must be able to plug this new flexible data storage layer into existing (and most recent) Hadoop versions and use any application on top of it. Notice that, this is in contrast to high-level storage layers, e.g. WWHow! layer [10] and RodentStore [4], because we do not come up with the physical design decisions, e.g. using what, where, and how in WWHow! layer and using storage algebra in RodentStore. Rather, we allow for storing data in HDFS with a *given* set of physical design decisions. In summary, the main goal of having this new full flexibility in data storage is to boost query performance while keeping users' applications compatible with newer versions of Hadoop.

In this demo, we present CARTILAGE, a fully flexible data storage framework built on top of HDFS. The beauty of CARTILAGE is that it does not require deep changes in Hadoop or HDFS. Rather, CARTILAGE builds on existing HDFS capabilities and exposes full control of data storage to users via UDFs. In the following, we give an overview of CARTILAGE in Section 2. We show four use-cases of CARTILAGE in Section 3 and describe our demonstration scenarios in Section 4.



(a) Data Upload (b) Job Execution

Figure 1: The CARTILAGE Architecture

2. CARTILAGE

In this section, we illustrate the features of Cartilage via an example of uploading a dataset (Section 2.1) and running a MapReduce job on top of it (Section 2.2). Overall, the core idea of CARTILAGE is to provide users full flexibility to store their datasets, while still using the Hadoop/HDFS framework as the underlying data storage/processing system. CARTILAGE does not require any change to the Hadoop framework. To do so, CARTILAGE wraps around existing HDFS and provides APIs with full storage control to users. This is a challenging task since HDFS has several storage constraints. For example, by default, HDFS maps every input data file to one *HDFS file* and internally partitions the file into several data blocks, based on only the size[1]. In the following, let us see how CARTILAGE frees the user from such storage constraints.

2.1 Data Upload

We illustrate the data upload architecture of the CARTILAGE framework in Figure 1(a). We see that CARTILAGE layers on top of any existing HDFS installation. Users upload datasets into HDFS using CARTILAGE. For this, users run a `DataUpload` job, wherein they provide their data storage preferences as configuration parameters. For example, a user can upload a dataset with a replication factor of 2 and can define the logical and physical partitioning for one of the replicas as follows:

```
// setting the replication factor
conf.setReplicationFactor(2);
/* Configuring the First Replica (i.e., Replica 0) */
// using a range partitioner for the logical partitioning
conf.setLogicalPart(RangePart.class, 0);
// setting the partitioning attribute
conf.setRangePartAttIdx("0", 0);
// setting the range values
conf.setRangePartLowKeys("1,101", 0);
conf.setRangePartHighKeys("100,200", 0);
// using a size-based partitioning for the physical partitioning
conf.setPhysicalPart(FileSizePart.class, 0);
// setting the physical size for the physical partitions
conf.setMaxPhysicalPartSize(maxFilesize, 0);
```

It is worth noticing that a user can use a different partitioning function for other data replicas. In this way, a user can replicate its dataset heterogeneously. Thus, in contrast to existing distributed file systems, CARTILAGE can partition each data block replica using a different partitioning function from each other.

Once CARTILAGE receives the `DataUpload` job, it automatically maps the input data to several HDFS files according to the provided configuration parameters. This means that for a given input file (or dataset), CARTILAGE might create one or more HDFS files. Essentially, CARTILAGE translates users' datasets into HDFS files, based on the storage preferences provided by users in the `DataUpload` job. As a result of this flexible mapping between user dataset and HDFS files (*physical file independence*), CARTILAGE offers users full flexibility to store their datasets. For example, CARTILAGE can support users to create arbitrary data layouts (row, column, PAX, etc.), sort orders, indexes (clustered, unclustered, covered, partial, etc), partitioning strategies (range, hash, etc.), compression schemes, data placement strategies, and replication policies (adjusting replication factor or even performing partial replication). Furthermore, users can make each of these storage decisions at different storage granularities such as file, data node, or block. In the extreme case, however, CARTILAGE users can still fall back to standard HDFS storage characteristics.

The reader might think that by layering CARTILAGE on top of HDFS, we significantly increase its data upload time. However, this is not the case. To illustrate this, Figure 2(a) shows the upload times of CARTILAGE (with the logical and physical partitioning described above) and HDFS over TPC-H `lineitem` table[2]. We vary the data size per node along the X-axis. From the figure, we see that the upload time of CARTILAGE is very close to the upload time of HDFS. The minor gap is due to the additional data parsing done by CARTILAGE. Thus, CARTILAGE is comparable to Hadoop when uploading the datasets.

Finally, notice that, CARTILAGE framework is fully extensible, i.e. users can define their own UDFs for arbitrary data storage characteristics and plug them into the CARTILAGE framework. For instance, a user can use her own `MyLogPart` logical partitioning function for the second replica (i.e, Replica 1) as follows:

```
conf.setLogicalPart(MyLogPart.class, 1);
```

2.2 Job Execution

We illustrate the MapReduce job execution architecture in Figure 1(b). Users can access their datasets through CARTILAGE running standard MapReduce jobs. The jobs are generated by either users themselves or other query engines, such as Pig or Hive. CARTILAGE exposes the metadata regarding the data storage decisions being made to users and query engines. Moreover, CARTILAGE also allows users to push down some of the operators, such as selection and projection. For example, if a user has applied range partitioning on her data, she can push down the selection predicate by specifying the following configuration parameters in her MapReduce job:

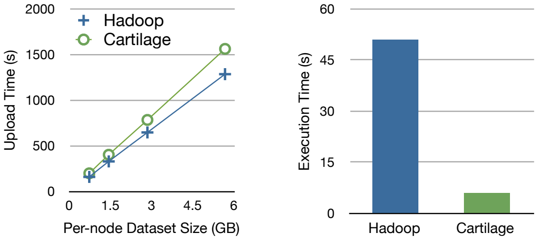
```
// set the selected attribute, i.e., attribute 0
QueryConf.setSelAttIdx(job.getConfiguration(), 0);
// set selection low and high keys
QueryConf.setSelLowKey(job.getConfiguration(), 10000);
QueryConf.setSelHighKey(job.getConfiguration(), 20000);
```

Once CARTILAGE receives the MapReduce job, it automatically uses the provided selection predicate in order to filter out the HDFS files that are not in the range of values defined by the user. CARTILAGE also exposes different data access paths, e.g., data path to access co-partitioned data, in the form of UDFs. These UDFs can be plugged-in when configuring MapReduce jobs, similar to the `DataUpload` job. Again, we do not modify the source code of Hadoop. Rather, we make use of UDFs so that CARTILAGE can be easily interfaced with existing Hadoop installations.

Figure 2(b) shows the MapReduce execution times for CARTILAGE and Hadoop when running a MapReduce job for a *select-groupby* query. The results show that CARTILAGE outperforms Hadoop by a factor of 8.5. Notice that, both Hadoop and CARTILAGE perform a full scan over their input data. However, CARTILAGE performs much better than Hadoop since it pushes down the

[1] Incoop [2] proposes a modified HDFS that creates data blocks based on content, i.e. content based chunking. However, this is the only flexibility that Incoop offers. It still inherits other storage constraints from HDFS. Furthermore, as discussed before, third party HDFS modifications are not viable in practice.

[2] We ran these experiments on a single MacBook Pro with OS X 10.8.1, 2.66 GHz processor, and 4GB memory.



(a) Upload Time (b) Query Time

Figure 2: Cartilage Vs Hadoop Performance

selection predicate to the split phase (i.e., to the phase where input splits are created for a given MapReduce job). As a result, Cartilage is able to process only the qualifying HDFS files rather than the entire dataset.

Once we have the flexibility to decide how to store a given dataset, we need to come up with one or more storage decisions and their granularity in an automatic and invisible way to users. The big challenge then is to automatically come up with all storage decisions using an optimizer. Alternatively, we can give the flexibility to users to configure some of the storage decisions and let the optimizer fill in the rest. In the worst case, we could fall back (at any time) to the default storage settings in standard Hadoop.

3. USE CASES

Cartilage allows users the flexibility to store their data sets in a manner which fits the needs of their applications. As a result, Cartilage covers a number of data storage use cases, in which users can extract better query performance by exploiting the data storage flexibility of Cartilage. In this section, we discuss four such use cases to illustrate the major advantages of using Cartilage as the data storage framework.

3.1 Heterogenous Data Partitioning

Modern data management systems store data in a way that fits the query workloads best. However, it is not always possible to partition the data so that it fits all the queries in the query workload. This is especially a problem for big data analytics, which contains a set of analytical tasks that aggregate data on different attributes. Likewise, big data analytical tasks might join a data set on different attributes. For example, SPARQL queries usually performs joins different pairs of subject, predicate, and object in the same RDF data set. Cartilage allows users to partition different replicas of their data differently, i.e. users can define different partitioning functions for each data replica. This means that, users can have their data partitioned over *three* different attributes at the same time by default, since HDFS keeps three data replicas by default. Such heterogenous data partitioning allows users to improve performance of more queries and fit their workloads better.

3.2 Flexible Data Replication

Distributed file systems, typically, replicate their data in order to achieve fault-tolerance over node failures and data corruption. However, since the goal of replication is only fault-tolerance, most distributed file systems replicate data at the file level, i.e. they make copies of the entire file. For instance, HDFS partitions the input data into blocks and replicates each data block the same number of times. However, this static system wide redundancy is not helpful in many cases, e.g. some parts of the data are queries more often than others. Cartilage relaxes this static file-level replication assumption and allows users to replicate their data sets at different levels, e.g. block level, column level, or even at a row level (partial replication) in order to improve the query performance. This means that a user can now choose, for instance, to replicate more those data blocks that are queried more often and de-replicate those data blocks that are queried infrequently. As a result of this flexible replication, users can achieve much better query performance.

3.3 Flexible Data Placement

By default, HDFS distributes data blocks evenly across data nodes in order to balance the data load. While this works well for simple scan-oriented tasks, it might not work for several complex analytical tasks. For example, users may want to co-locate two data sets on a given join key, in order to speed up the query performance of join tasks. Users might also want to store data blocks in a given range of values (in case of range partitioning) on different data nodes, in order to increase parallelisation. Similarly, a user might want to store input data sets for CPU-intensive tasks on data nodes that have better CPUs. Cartilage offers such data placement flexibilities to the user. Essentially, with Cartilage, users can trade load balancing for improved query performance. Furthermore, Cartilage allows users to automate such data placement decisions for better query performance, just as default HDFS automatically changes block locations for better load balancing.

3.4 Heterogenous Data Layouts

Several research efforts, in the past two years, have tried to overcome the performances limitation of row-oriented data processing, the default data processing in Hadoop MapReduce. As a result, several data layouts has been proposed by many researchers [12, 3, 8, 7]. However, it is well known that no single layout can fit all query workloads perfectly. Therefore, users cannot stick to a single layout all the time. Recently, [9] proposed to store each data replica with a different data layout. But, this approach requires deep change in HDFS. Cartilage allows users to store each data block replica in a different layout without any change in the underlying HDFS framework. Furthermore, Cartilage allows users to change the data layout of any data block replica at any time. In fact, with Cartilage, users can even define their custom data layouts to suit their applications.

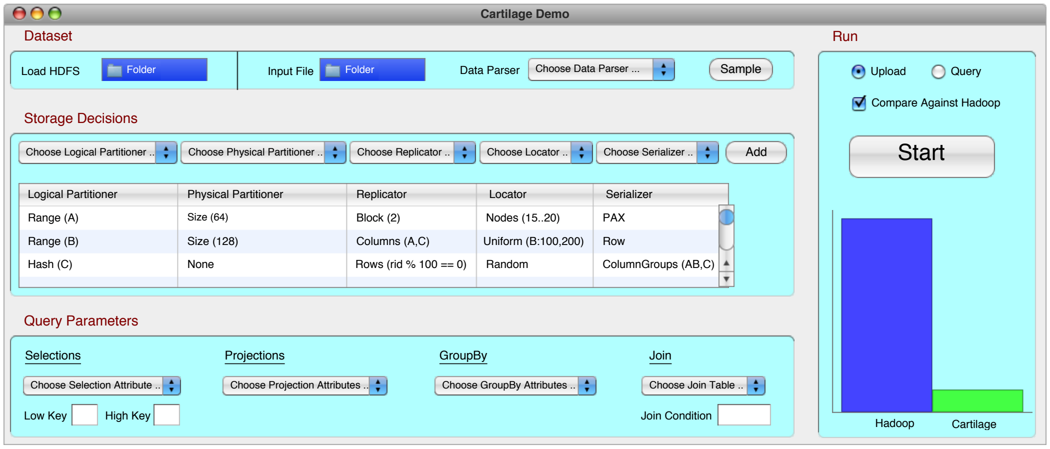
4. DEMONSTRATION

Our aim in this demo is to show how Cartilage can be leveraged by users to improve the performance of MapReduce jobs in different scenarios. We present four demonstration scenarios, along the same lines as the four use cases discussed in Section 3.

Demo Setup. We compare the performance of MapReduce jobs using both Cartilage and standard HDFS, in order to better illustrate the advantages of Cartilage. We plan to use our local 10-node cluster at QCRI. For each demo scenario, we run one MapReduce job using Cartilage and one MapReduce job using standard HDFS. For this, we spit our cluster into two 5-node clusters. We demonstrate our Cartilage storage framework over three datasets and benchmark: (i) RDF dataset in LUBM benchmark [13]; (ii) decision support dataset in TPC-H benchmark [16]; (iii) scientific dataset in SDSS benchmark [15]. We provide a friendly GUI (see Figure 3), wherein the audience can play with the storage flexibility features of Cartilage and validate the following four demonstration scenarios.

4.1 Arbitrary Selections over Linked Data

We consider an RDF data processing scenario using the LUBM [13] benchmark. We consider this scenario to demonstrate the efficiency of Cartilage with *data heterogenous partitioning*. Our main goal is to show how SPARQL queries can significantly decrease their execution times when storing RDF datasets with



Logical Partitioner	Physical Partitioner	Replicator	Locator	Serializer
Range (A)	Size (64)	Block (2)	Nodes (15..20)	PAX
Range (B)	Size (128)	Columns (A,C)	Uniform (B:100,200)	Row
Hash (C)	None	Rows (rid % 100 == 0)	Random	ColumnGroups (AB,C)

● Upload ○ Query

☑ Compare Against Hadoop

Figure 3: Graphical User Interface to Upload and Query Data from CARTILAGE.

CARTILAGE. For this, the audience is invited to use the CARTILAGE GUI (See Figure 3) in order to upload an RDF dataset using three different data partitioning functions. The CARTILAGE GUI also allows the audience to run a set of queries over the uploaded RDF dataset to see the benefits in terms of execution time.

4.2 Storage Space Optimization

In this scenario, we consider the previously uploaded RDF dataset. We run a series of SPARQL queries having a selection predicate on the same attribute and within the same range of values. After this, we invite the audience to use CARTILAGE to increase the replication factor for the data blocks containing only RDF triples in the relevant range of values, i.e. the range which qualifies in the RDF queries. Then, we invite the audience to run again the same series of SPARQL queries. This allows the audience to observe the benefits of replicating a dataset for query performance purposes. Still, even with improved query performance, CARTILAGE requires almost the same storage space as before, which is not the case for HDFS. We also invite the audience to decrease the data replication factor for the infrequently requested data blocks. With this, the audience can observe that CARTILAGE can free storage space without negatively impacting the overall query performance.

4.3 Performance Efficient Data Placement

In this scenario, we show how CARTILAGE can boost data analytics by having full control on the data block placement. To show this, we consider two demo cases. First, we consider a case where users applied flexible data replication to de-replicate some data blocks. The audience is then invited to upload the `lineitem` dataset. Here, we show that CARTILAGE stores the new dataset on data nodes where the local average replication factor of data blocks is below the global average. Second, we consider a case where a query workload frequently requests a given range of values in attribute `orderkey` of the `lineitem` dataset. In this case, we show how CARTILAGE can speed up the query workload by evenly distributing the qualifying data ranges across all data nodes.

4.4 Hybrid Query Plans

We show how CARTILAGE can upload a dataset with a different data layout per replica (e.g. PAX, ROW, and RCFile-like). For this, we consider the `PhotoObj` dataset as defined in [15]. In this scenario, we invite the audience to upload the `PhotoObj` dataset using three different data layouts. Then, we invite the audience to run three different analytical MapReduce jobs over CARTILAGE and standard HDFS. Especially, we show how CARTILAGE enables new possibilities for query processing. We run a MapReduce job that implements the following simplified SQL query from SDSS [15]: `SELECT * FROM PhotoObj WHERE g BETWEEN 0 AND 20`. To perform this job over CARTILAGE, we use an hybrid query plan. We read attribute `g` from the dataset replica in PAX-layout and performs the selection predicate on it. Then, for the qualifying values, we read the corresponding tuples from the dataset replica in ROW-layout to return the results. CARTILAGE makes such hybrid query plans possible in Hadoop.

In all above scenarios, the audience is able to interact with CARTILAGE using the GUI shown in Figure 3. The audience can compare the performance of MapReduce jobs over CARTILAGE with the performance of MapReduce jobs over HDFS. Notice that, for the live demo, we consider small datasets in order to make the demos more interactive when uploading datasets. However, we also consider larger datasets loaded in advance.

5. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1), 2009.

[2] P. Bhatotia et al. Incoop: Mapreduce for Incremental Computations. In *SOCC*, 2011.

[3] S. Chen. Cheetah: A High Performance, Custom Data Warehouse on Top of MapReduce. *PVLDB*, 3(2), 2010.

[4] P. Cudre-Mauroux, E. Wu, and S. Madden. The Case for RodentStore, an Adaptive, Declarative Storage System. In *CIDR*, 2009.

[5] 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), 2010.

[6] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only Aggressive Elephants are Fast Elephants. *PVLDB*, 5(11), 2012.

[7] A. Floratou et al. Column-Oriented Storage Techniques for MapReduce. *PVLDB*, 4(7), 2011.

[8] Y. He et al. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *ICDE*, 2011.

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

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

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

[12] Y. Lin et al. Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In *SIGMOD*, 2011.

[13] LUBM benchmark, `http://swat.cse.lehigh.edu/projects/lubm/`.

[14] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[15] Sloan Digital Sky Survey, http://www.sdss.org.

[16] TPC-H, http://www.tpc.org/tpch/.