Amoeba: A Shape changing Storage System for Big Data

Anil Shanbhag*[1] Alekh Jindal⋆[2] Yi Lu*[3] Samuel Madden*[4]

*MIT CSAIL ⋆Microsoft

{[1]anil, [3]yilu, [4]madden}@csail.mit.edu [2]aljindal@microsoft.com

ABSTRACT

Data partitioning significantly improves the query performance in distributed database systems. A large number of techniques have been proposed to efficiently partition a dataset for a given query workload. However, many modern analytic applications involve ad-hoc or exploratory analysis where users do not have a representative query workload upfront. Furthermore, workloads change over time as businesses evolve or as analysts gain better understanding of their data. Static workload-based data partitioning techniques are therefore not suitable for such settings.

In this paper, we describe the demonstration of Amoeba, a distributed storage system which uses adaptive multi-attribute data partitioning to efficiently support ad-hoc as well as recurring queries. Amoeba applies a robust partitioning algorithm such that ad-hoc queries on all attributes have similar performance gains. Thereafter, Amoeba adaptively repartitions the data based on the observed query sequence, i.e., the system improves over time. All along Amoeba offers both *adaptivity* (i.e., adjustments according to workload changes) as well as *robustness* (i.e., avoiding performance spikes due to workload changes). We propose to demonstrate Amoeba on scenarios from an internet-of-things startup that tracks user driving patterns. We invite the audience to interactively fire fast ad-hoc queries, observe multi-dimensional adaptivity, and play with a robust/reactive knob in Amoeba. The web front end displays the layout changes, runtime costs, and compares it to Spark with both default and workload-aware partitioning.

1. INTRODUCTION

Data partitioning is a well-known technique for improving the performance of database applications. For instance, when selecting subsets of the data, having the data prepartitioned on the selection attribute allows *skipping* irrelevant pieces of the data, i.e., without scanning the entire dataset. Joins and aggregations also benefit from data partitioning. Because of these performance gains, the database research community has proposed many techniques to find good data partitioning for a query workload. Such *workload-based* data partitioning techniques assume that the query workload is provided upfront or collected over time [8, 6, 7]. Unfortunately, in many cases a static query workload may not be known a priori.

One reason for this is that modern data analytics are data-centric and tend to involve ad-hoc and exploratory analysis. For example, an analyst may look for anomalies and trends in a user activity log, such as from web servers, network systems, transportation services, or any other sensors. Such analyses are ad-hoc and a representative set of queries is not available upfront. To illustrate, production workload traces from a Boston-based analytics startup reveal that even after seeing 80% of the workload, the remaining 20% of the workload still contains 57% new queries. These workload patterns are hard to collect in advance. Furthermore, collecting the query workload is tedious as analysts would typically like to start using the data as soon as possible, rather than having to provide a workload before obtaining acceptable performance. Providing a workload upfront has the further complexity that it can *overfit* the database to that workload, requiring all other queries to scan unnecessary data partitions to compute answers.

Our solution Amoeba, is designed with three key properties in mind: (1) it requires *no upfront query workload*, while still providing good performance for a wide range of ad-hoc queries; (2) as users pose more queries over certain attributes, it *adaptively repartitions* the data, to gradually perform better on queries over frequent attributes and attribute ranges; and (3) it provides *robust performance* that avoids overfitting to any particular workload.

Distributed storage systems like HDFS store large files as a collection of blocks of fixed size (for HDFS the blocks size is usually 64/128MB). A block acts as the smallest unit of storage and gets replicated across multiple machines. Amoeba exploits this block structure to build and maintain a partitioning tree on top of the table. A partitioning tree is a binary tree which partitions the data into a number of small partitions of roughly the size of a block. Each such partition contain a hypercube of the data. Figure 1 shows an example partitioning tree for a 1GB dataset over 4 attributes with block size 128MB. The data is split into 8 blocks, the same as what would be created by a block-based system, however each block now has additional metadata. For example, block 1's tuple satisfy $A \leq 4$ & $B \leq 5$ & $D \leq 4$. This results in it being possible to answer *any* query by reading a subset of partitions. The partitioning tree is improved over time

Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

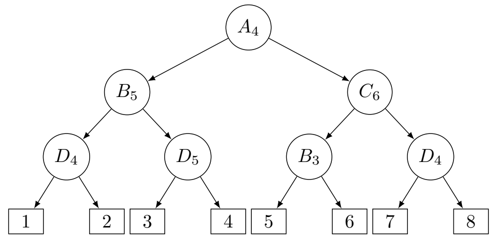


Figure 1: Example partitioning tree with 8 blocks

based on queries submitted by the user.

Amoeba exposes a relational storage manager which can be used to do predicate-based storage access. Amoeba seamlessly integrates into the Spark/HDFS [1] stack as a custom data source [2].

In this demo, we will show Amoeba running on a 10 node cluster and using it to answer queries over a 705 GB dataset containing measurement data from user trips. We will show the benefits of using Amoeba in terms of time to get started, the performance of the first few queries and total runtime reduction under different scenarios that an analyst using this data would encounter. We compare against unmodified Spark and using Spark with the table range partitioned on the most popular attribute.

2. SYSTEM OVERVIEW

The goal of Amoeba is to provide good query performance at all instants of time without assuming an upfront workload being given. The system exposes a relational storage manager, consisting of a collection of tables, with support for predicate-based data access i.e. scan a table with a set of predicates to filter over returns a subset of the data to access. For example, scan table employee with filter: age $\geq$ 30 and 1000 $\leq$ salary $\leq$ 2000. The data stored is partitioned and as a result we end up accessing a subset of the data. The system is self-tuning and as users start submitting queries to the system, it specializes the partitioning to the observed patterns over time.

Query optimizers tend to push predicates down to scan operator and big data systems like Spark SQL [1] allow custom predicate-based scan operators. Amoeba integrates as a predicate-based scan operator and the layout changes due to data repartitionings are invisible to the user.

The system consists of three major components: the upfront data partitioner, the optimizer, and the adaptive repartitioner.

2.1 Upfront Data Partitioner

The upfront data partitioner runs when the data is uploaded into a block-based storage system like HDFS. Since there is no workload information, the goal is to partition the data such that queries on any attribute see improved performance. We do this by creating a robust partitioning tree, which tries to make the number of ways the data is partitioned on all the attributes the same. Figure 1 is an example of a robust partitioning tree for a 1GB dataset over 4 attributes with block size 128MB. Real world datasets tend to be skewed or have correlation among attributes. In order to generate almost equally sized blocks, we collect a sample from the data and use it to choose the appropriate cut points.

At the end of upload process, each dataset resides in its own directory. The data is split into blocks as decided by the partitioning tree. In addition, we create three additional files. The index file stores the partitioning tree used to partition the data. The sample file contains a sample of the dataset. The queries file stores all the queries that touched this dataset. The upfront partitioning is useful for improved query performance but more importantly it serves as a good start point for later adaptivity.

2.2 Optimizer

As users begin to query the dataset, it is beneficial to improve the partitioning based on the observed queries. As in most online algorithms, we assume that past queries are indicative of the queries to come. When a query is submitted to the system, the optimizer needs to 1) explore alternative partitioning trees for the dataset to find the best one and 2) decide if re-partitioning is worthwhile.

We maintain a history of the past queries called the query window. The window maintains a history of queries for the past X hours (we use $X = 4$). The cost of a query q over a partitioning tree T is:

$$\text{Cost}(T, q) = \sum_{b \in \text{lookup}(T,q)} n_b$$

where $lookup(T, q)$ returns the set of relevant buckets for query q in T and n_b is the number of tuples in bucket b.

The alternative partitioning trees are generated by using a set of transformation rules on the current partitioning tree. The query predicates are used as hints to generate them. Among the set of alternative trees generated, we select the tree T' that maximizes the total benefit:

$$\text{Benefit}(T') = \sum_{q \in Q} Cost(T, q) - \sum_{q \in Q} Cost(T', q)$$

For the best found tree, with B being the set of blocks that need to be repartitioned to achieve it, we compute the repartitioning cost as:

$$\text{RepartitioningCost}(T', q) = \sum_{b \in B} c \cdot n_b$$

The re-partitioning happens only when the difference in cost of the queries before and after can pay for the repartitioning cost. The important parameter to note here is c. c represents the write-multiplier i.e., how expensive writes are compared to a read. Changing c alters the properties of the system: on one end setting $c = \infty$ makes it imitate a system with no re-partitioning, at the other end setting $c = 0$ makes it re-partition the data every time it sees benefit.

2.3 Adaptive Repartitioner

The optimizer returns a plan containing the set of blocks to be accessed and a new partitioning tree if it is feasible. From this, we can split the blocks into two sets: set one contains all blocks that would be scanned and filter predicates applied on them and set two contains all blocks that would be scanned, re-partitioned based on the new tree and filter predicate applied on them. Two things to note here, first the set of blocks to re-partition is always a subset of blocks accessed i.e. we never read extra data. Second, the re-partitioning piggybacks off the query execution saving the scan cost. The final tuples after all the predicates have been applied are returned for further processing.

3. PERFORMANCE

We conducted extensive evaluation using TPC-H and a real-world analytical workload. The experiments were run on a 10 node cluster where each node has 16 2.07 GHz Xeon cores, 256 GB RAM and 10 TB of storage. Here we present a highlight from the real-world analytical workload.

3.1 *IoT* Dataset

We obtained data from a Boston-based company that captures analytics regarding a user's driving trips. Each tuple in the dataset represents a trip taken by the user, with the start time, end time, and a number of statistics associated with the journey regarding various attributes of the driver's speed and driving style. The data consists of a single large fact table with 148 columns. To protect user's privacy, we used statistics provided by the company regarding data distributions to generate a synthetic version of the data according to the actual schema. The total size of the data is 705GB. We also obtained a trace of ad-hoc analytics queries from the company (these queries were generated by data analysts performing one-off exploratory queries on the data). This trace consists of 105 queries, run between 04/19/2015 and 04/21/2015, on the trip data. The queries sub-select different portions of the data, e.g., based on trip time range, distances, and customer ids, before producing aggregates and other statistics.

3.2 Evaluation

We compared the performance of AMOEBA versus two baselines: 1) using Spark without any modifications, 2) using Spark with the data already partitioned on *upload_time*, which is most frequently accessed attribute (accessed in 78% of the queries). The data was split equally across the machines and loaded in parallel using the upfront data partitioner. The queries collected were then run in order.

Upfront Overhead: Time taken to upload the data directly into HDFS is 38 minutes. The time taken to upload via the upfront data partitioner is 62 minutes. The upfront data partitioner does two full scans, one full parse and writes the data 3-way replicated into HDFS compared to the direct method which just does one full scan and writes it 3-way into HDFS. As a result, we get a 1.6x overhead.

Total query runtime: Figure 2 shows the total query runtime for running the 105 queries using the different approaches. To remind the reader, c is the write multiplier (described in Section 2.2). c can be calibrated by measuring the runtime increase due to re-partitioning. For our setup, c is 4, i.e.: writing out data (while re-partitioning) is four times more expensive than just scanning the data. We observe that by range partitioning on *upload_time*, the total query runtime drops by 1.9x. Amoeba initially does worse off as it does not have knowledge of the workload, however in the end it does 1.8x better than Spark with data partitioned and 3.4x better than unmodified Spark. The $c = 2$ setting is more reactive in the sense it adapts to the query workload faster. It does slightly worse than $c = 4$ setting as it ends up doing introducing changes to the tree very soon, as the workload is ad-hoc some patterns are one-off and re-partitioning done to improve them ends up being wasted effort. Finally, we observed that the total runtime of the last 25 of the 105 queries on AMOEBA is $19x$ lower than full scan and $11x$ lower than using the range-partitioned data.

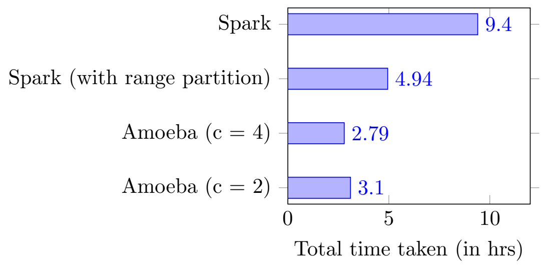


Figure 2: Total runtime of the different approaches

4. DEMONSTRATION

The goal of the demonstration is to help the attendees understand the performance characteristics and the nature of gains got by AMOEBA. We consider the following four characteristics to evaluate our adaptive data system:

(1.) *Upfront overhead*, i.e., the initial partitioning overhead.
(2.) *Time to first query*, i.e., how do the initial query performance(s) look like.
(3.) *Break-even point* i.e., how many queries it takes to recover the cost of upfront partitioning.
(4.) *Total gain*, i.e., the ratio of the total runtime of all queries with the total runtime of a baseline.

Below, we first describe our setup, and then describe the demonstration scenarios for Amoeba.

4.1 Setup Details

Our demo setup consists of an open source implementation of AMOEBA built on top of Spark and HDFS using the same setup as in Section 3. To demonstrate real world ad-hoc analysis scenarios, we will pre-load the IoT dataset of driving events (described in Section 3.1) as a denormalized fact table.

In order to effectively demonstrate the system, we developed a ajax-based web front end that allows attendees to run a set of queries on the system, analyze the changes made to the partition tree over time and compare the performance gain against two baselines: (i) using unmodified Spark to read data and apply predicates on them (incurs full scan), and (ii) Spark with data range-partitioned by upload time which is the most popular attribute. Since the data is preloaded for demonstration purposes, the web front end also shows the upload time in different approaches. Figure 3 shows a screenshot of the console.

Behind the scenes, once the user submits the queries, the system runs the queries one-by-one on the optimizer. Note that the optimizer takes in a query, the past queries, a sample of the data and the current partitioning tree to produce possibly a different partitioning tree. The optimizer also produces estimated runtimes of the queries which is then plotted on a line graph with similar lines drawn for the two baselines. The actual execution time of the queries are much larger than permitted by the time constraints of the demonstration. To show that the optimizer predicted runtimes are close to actual runtimes, we also include the predicted and actual runtimes for the workload executed in Section 3.

4.2 Demonstration Scenario

Our demonstrations places attendees in the shoes of an analyst at an Internet of Things (IoT) startup that tracks user driving patters. She would like to consider using AMOEBA as the storage manager for her data. We describe the demonstration scenarios below that showcase the different aspects of the system and invite the attendees to play with them.

4.2.1 *Fast Ad-hoc Queries*

The analyst gets a daily dump of data and wants to quickly start her analysis. Furthermore, she wants to quickly try subsets of the data along different dimensions, e.g. trip length, start latitude/longitude, etc., before going into more details. Essentially, the analyst wants to have improved query performance right from the first query itself even when the queries are ad-hoc in nature. Amoeba allows the analyst to load the data with a small overhead and then fire ad-hoc queries over any attribute, with better performance than the full scans.

The demonstration consists of two stages. First we invite the audience to provide different data sizes and ad-hoc queries and see the expected time it will take for the analyst to get started (i.e., the response times of these ad-hoc queries). The audience can then fire these ad-hoc queries on the pre-loaded data. Example queries include getting average speed, time spent over 40mph, breaks applied per kilometer, trip distance, etc. We show the improvements compared to the baselines.

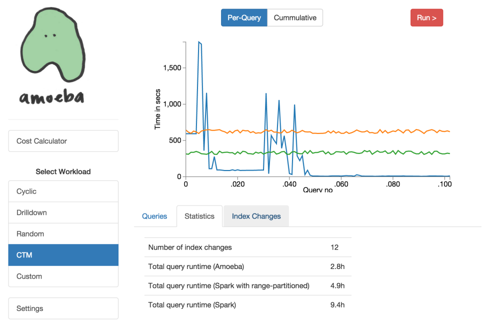
4.2.2 *Multi-dimensional Adaptivity*

While the analyst typically starts with ad-hoc queries, she often runs a long sequence of queries (on one or more attributes/dimensions) in her analysis. While the improved ad-hoc query performance is good to start with, she wants the system to adapt to her query patterns, i.e., recognize the fact that the queries are not ad-hoc anymore, and improve even further. The analyst wants the system to get faster and faster as she proceeds in her analysis. Amoeba monitors the query workloads and automatically repartitions the data as it gets used over time. Essentially, the system refines the partitioning along more frequently used dimensions/attributes, while coarsening the partitioning along less frequently used ones.

This demonstration scenario starts with the audience providing a query sequence pattern to the system. We will provide some pattern templates, such as random query sequences, cyclic query sequences, and drill down query sequences. Amoeba shows the expected costs of running a given query sequence. The audience can observe the individual query time, the partitioning overhead, and convergence towards the ideal runtime. The audience can play with the system to change the query sequence pattern and observe how each of the above metrics change. They can also see the changes to the underlying partitioning tree.

4.2.3 *The Robust/Reactive knob*

Amoeba takes care of adaptively repartitioning the data, however, the analyst still wants to control how robust or reactive the system is to workload changes. She may, for instance, know that it is going to be a short query session and hence spending too much effort in adaptivity will not pay-off. Or, she may already have a specific workload activity in mind and want the system to adapt very quickly. Amoeba allows users to control this by changing the repartitioning cost factor c (see Section 2.2) . On one extreme, the system could repartition the data for every incoming query,



Number of index changes	12
Total query runtime (Amoeba)	2.8h
Total query runtime (Spark with range-partitioned)	4.9h
Total query runtime (Spark)	9.4h

Figure 3: Sample workload run via the web console

while on the other extreme it may never repartition. Thus, Amoeba covers the entire spectrum.

We invite the audience to tune the robustness knob and see how Amoeba reacts to workload changes. A more robust setting will trigger less data repartitioning, while a more reactive setting will trigger more repartitioning. The audience can launch the query sequence with different settings.

5. RELATED WORK

The problem of generating partitioning layout given a query workload has been studied extensively. [8, 6] suggests a physical layout based on the workload. Recently, [7] proposed to create data blocks based on the features extracted from each input tuple. The features are selected based on the workload and the goal is to cluster tuples with similar features into the same data block. All these methods require an upfront workload.

For in-memory column stores, cracking [4] has been used to generate adaptive index on a column based on incoming queries. Partial sideways cracking [5] extends it to generate adaptive index on multiple columns. Cracking happens on each query and maintains additional structures to create the index. The reason cracking cannot be applied to distributed data store is that the cost of re-partitioning is very high. Each round of re-partitioning needs to be carefully planned to amortize the cost associated with it.

AQWA [3] looks at adaptive data partitioning for spatial data (2 dimensions). Their techniques do not scale to higher dimensions.

6. REFERENCES

[1] Apache Spark. `http://spark.apache.org/`.
[2] Spark SQL Data Source API. `http://bit.ly/1HWrGdZ`.
[3] A. M. Aly et al. Aqwa: adaptive query workload aware partitioning of big spatial data. *VLDB*, 2015.
[4] S. Idreos et al. Database cracking. In *CIDR*, 2007.
[5] S. Idreos et al. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.
[6] R. Nehme and N. Bruno. Automated Partitioning Design in Parallel Database Systems. *SIGMOD*, 2011.
[7] L. Sun et al. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, 2014.
[8] D. C. Zilio et al. Db2 design advisor: Integrated automatic physical database design. *VLDB*, 2004.