A Robust Partitioning Scheme for Ad-Hoc Query Workloads

Anil Shanbhag
MIT CSAIL
anil@csail.mit.edu

Alekh Jindal
Microsoft
aljindal@microsoft.com

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Jorge Quiane
QCRI
jquianeruiz@qf.org.qa

Aaron J. Elmore
U.Chicago
aelmore@cs.uchicago.edu

ABSTRACT

Data partitioning is crucial to improving query performance and several workload-based partitioning techniques have been proposed in database literature. However, many modern analytic applications involve ad-hoc or exploratory analysis where users do not have a representative query workload a priori. Static workload-based data partitioning techniques are therefore not suitable for such settings. In this paper, we propose Amoeba, a distributed storage system that uses adaptive multi-attribute data partitioning to efficiently support ad-hoc as well as recurring queries. Amoeba requires zero set-up and tuning effort, allowing analysts to get the benefits of partitioning without requiring an upfront query workload. The key idea is to build and maintain a partitioning tree on top of the dataset. The partitioning tree allows us to answer queries with predicates by reading a subset of the data. The initial partitioning tree is created without requiring an upfront query workload and Amoeba adapts it over time by incrementally modifying subtrees based on user queries using repartitioning. A prototype of Amoeba running on top of Apache Spark improves query performance by up to 7x over full scans and up to 2x over range-based partitioning techniques on TPC-H as well as a real-world workload.

CCS CONCEPTS

• **Information systems → MapReduce-based systems**; **Relational parallel and distributed DBMSs**;

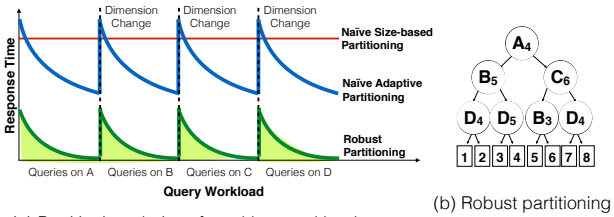
KEYWORDS

adaptive partitioning, data skipping, ad-hoc analytics, exploratory data analysis

ACM Reference Format:
Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. 2017. A Robust Partitioning Scheme for Ad-Hoc Query Workloads. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3131613

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3131613



(a) Partitioning choices for ad-hoc workloads

(b) Robust partitioning tree with 8 blocks

Figure 1: Need for robust data partitioning.

1 INTRODUCTION

Data partitioning is a well-known technique for selecting subsets of data and a myriad of *workload-based* partitioning techniques have been proposed in the database literature [4, 12, 24, 26, 31, 34]. Modern data analytics, however, tends to involve ad-hoc and exploratory analysis where a static query workload is not known a priori. For example, an analyst may look for patterns in a database of multi-dimensional web click events (with user history, demographic information, and platform information as dimensions). The analyst may want to view this data according to any of its dimensions – e.g., they may want to query according to the user's past browsing patterns, by their age or income, or by whether they are using a mobile phone or a laptop. As the specific set of attributes of interest is not necessarily known upfront, workload-based partitioning techniques cannot be applied. We have observed that this is often a problem in practice. For example, we analyzed a production database workload traces from an Internet of Things (IoT) startup company and found that even after seeing the first 80% of the queries, the remaining 20% of the workload still contained 57% new queries.

Figure 1(a) illustrates the data partitioning dilemma that analysts face with these newer workloads. They are either stuck with naïve size based partitioning that offers no data skipping capability and hence very poor performance (full scan). Or, alternatively, they could pick one of the more recent adaptive partitioning techniques (e.g., cracking [20]) that would make the first few queries even slower than full scan, but will gradually improve if successive queries are on the same dimension, i.e., having a selection predicate on the same attribute. In case the query dimension changes, the performance again goes back to worse than full scan before gradually improving with successive queries on the new dimension (we call it naïve adaptive partitioning). This is really painful for an analyst exploring multiple dimensions: she wants a data partitioning scheme that is **robust** *to the ad-hoc nature of her workload*

and provides good performance from the first query itself, adaptively improving from there on.

In this paper, we present a novel data partitioning scheme that creates an upfront (multi-dimensional) partitioning tree to chunk data across all attributes in the schema. It then adaptively modifies the (multi-dimensional) tree based on the user queries. Figure 1(b) shows an example partitioning tree for a 0.5GB dataset over 4 attributes with block size 64MB. Each node splits the data arriving based on the cut-point (e.g., node B_5 splits the data arriving such that the left subtree has tuples with $B \leq 5$ and the right subtree has tuples with $B > 5$). The data is split into 8 blocks and each block has additional partitioning metadata. For example, block 1's tuples satisfy $A \leq 4$ & $B \leq 5$ & $D \leq 4$. The advantage of this partitioning tree is that it is possible to answer a query with a predicate on any attribute by reading a subset of partitions, rather than scanning the whole table. Of course, the benefit per query is less than if the data were completely partitioned on the exact attributes the query used. A second advantage is that it is possible to refine the partitioning over time, in response to patterns observed in queries that are run in the system. We show that this can be done in a lightweight fashion, by merging and repartitioning a few blocks at a time.

We implemented this idea in a new storage system we have built called Amoeba. Amoeba is designed with two 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. Note that our approach is complementary to many other physical storage optimizations, (e.g., column-stores), and these optimizations could still be applied to our partitioning scheme, (e.g., individual columns or column groups could easily be separately partitioned and accessed in our approach).

In summary, we make the following major contributions:

(1.) We describe a set of techniques to partition a dataset over several attributes and propose an algorithm to generate an initial partitioning tree. Our partitioning tree spreads the benefits of data partitioning across all attributes in the schema. It does not require an upfront query workload and also handles data skew and correlations (Section 3).

(2.) We describe an algorithm to adaptively repartition the data based on the observed workload. Our approach repartitions only the accessed portions of the data and uses divide-and-conquer to efficiently pick the best repartitioning strategy, such that the expected benefit of repartitioning outweighs the expected cost. To the best of our knowledge, this is the first work to propose adaptive data partitioning for analytical workloads on distributed systems (Section 4).

(3.) We implement our system as a storage engine for Spark and SparkSQL (Amoeba could equally work with any other distributed database system) (Section 5).

(4.) We present a detailed evaluation of the Amoeba storage system on real and synthetic query workloads to demonstrate three key properties: (i) robustness in terms of improved performance over ad-hoc queries right from the start, (ii) adaptivity to the changes in the query workload, and (iii) use of workload hints to outperform workload-aware static data partitioning. We also evaluate our system on a real query workload from an IoT startup (Section 6).

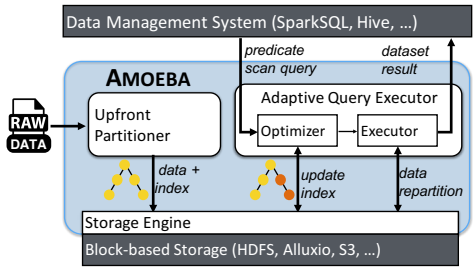


Figure 2: Amoeba Architecture

2 AMOEBA OVERVIEW

Amoeba exposes a relational storage manager, consisting of a collection of tables. A query to Amoeba is of the form <table, (filter predicates)>, for example $< employee, (age > 30, 100 \leq salary \leq 200) >$. As the table is stored based on the table's partitioning tree, Amoeba is able to answer the query by accessing only the relevant data blocks. Figure 2 shows the overall architecture of Amoeba. The three key components are:

(i) Upfront partitioner. The upfront partitioner partitions a dataset into blocks and spreads them throughout a block-based file system. The blocks are created based on a partitioning tree, without requiring a query workload. Note that since Amoeba partitions the data along several dimensions, we could end up de-clustering the data blocks across machines and performing random I/Os for each block. However, this is fine as large block sizes in distributed file systems [18] combined with fast network speeds lead to remote reads being almost as fast as local reads [6, 9]. Essentially, we sacrifice some data locality in order to quickly locate the relevant portions of the data on each machine in a distributed setting.

(ii) Storage Engine. The storage engine builds on top of a block-based storage system to store tables. Each table represents a dataset loaded using the upfront partitioner. The table contains an index file which stores the partitioning tree used to partition the dataset and the partitioned dataset as a collection of data blocks. In addition, we also store a query log containing the most recent queries that accessed the dataset and a sample of the dataset whose use is described later.

(iii) Adaptive Query Executor. The adaptive query executor takes queries in the form of a predicated scan and returns back the matching tuples. As Amoeba internally stores the data partitioned by the partitioning tree, it is able to skip many data blocks while answering queries. The query first goes to the optimizer. When the data blocks accessed by the query are not perfectly partitioned, the optimizer considers repartitioning some or all of the accessed data blocks, as they are accessed by the query and using the query predicates as cut-points. We use a cost model to evaluate the expected cost and benefit of repartitioning.

Amoeba integrates into the Spark/HDFS stack as a custom data source that supports predicate pushdown. It can be queried from a relational data processing framework like Spark SQL [2] or Hive [1] that supports custom data sources. The Spark SQL optimizer pushes

predicates down to the scan and calls Amoeba with the predicated scan query. The system returns the matching tuples as a Spark RDD, which is then used by Spark SQL to do subsequent operations like join processing and aggregation. The Amoeba storage system is self-tuning, lightweight (both in terms of the upfront and repartitioning costs), and does not increase the storage space requirements. Repartitioning happens continuously underneath the interface in a way that is invisible to the user.

In the rest of the paper, we describe an efficient predicate-based data access system that does data skipping to improve query performance. An extension to do efficient join processing on top of Amoeba is discussed in a companion paper [22].

3 UPFRONT DATA PARTITIONING

A distributed storage system, such as HDFS, subdivides a dataset into smaller chunks, called blocks, based on size (usually 64MB or 128MB). Prior workload-aware techniques, such as content-based chunking [8] and feature-based blocking [31], create blocks such that irrelevant blocks could be quickly skipped for the specific query workload. We go a step further by creating blocks based on a partitioning tree that allows us to skip data for almost *all* ad-hoc queries, *without* having any information about the query workload. Such a partitioning also serves as a good starting point for the adaptive query executor to improve upon.

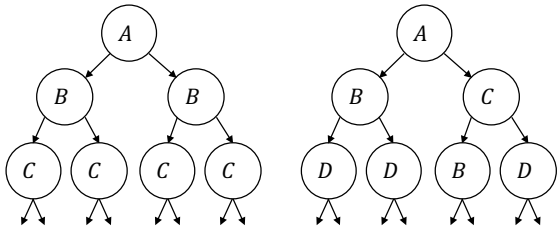
We first present the three key ideas used in building the partitioning tree: heterogeneous branching to accommodate many attributes into a binary partitioning tree, the concept of attribute allocation to capture the average partitioning effort to be spent on each attribute, and median-based splitting to handle skew and correlation in the dataset. Finally, we describe our upfront partitioning algorithm which uses these ideas to come up with a partitioning tree for a given dataset.

3.1 Heterogeneous Binary Partitioning Tree

We represent the partitioning tree as a balanced binary tree, i.e., we successively partition the dataset into two until we reach the maximum partition size[1]. The choice of binary tree is deliberate as it is fairly general (a four-way partitioning can be achieved by two successive two-way partitioning), and it allows fine-grained modifications when adapting the tree to workloads changes later. Each node in the tree is represented as A_p, where A is the attribute being partitioned on and p is the cut-point. All tuples with $A \leq p$ go to the left subtree and rest go to the right subtree. A leaf node in the tree is a **bucket**, having a unique identifier and a file name in the underlying file system. This file contains the tuples that satisfy the predicates of all nodes traversing upwards from the bucket to the root of the tree. Note that an attribute can appear in multiple nodes in the tree. Having multiple occurrences of an attribute in the same branch of the tree increases the number of ways the data is partitioned on that attribute.

Figure 3(a) shows a partitioning tree analogous to the k-d tree [7]. A k-d tree typically partitions the space by considering the attributes in a round robin fashion, until the smallest partition size is reached. Hence, the tree can only accommodate as many attributes as the depth of the tree. For a dataset size D, minimum partition size P, and n-way partitioning over each attribute, the partitioning tree contains $\lfloor log_n \frac{D}{P} \rfloor$ attributes. With $n = 2$, $D = 1$TB, and $P = 64$MB, we can only accommodate 14 attributes in the partitioning tree. However, many real-world schemas have many more attributes.

[1]For HDFS, we take the block size as the maximum partition size.



(a) Partitioning Tree (b) Heterogeneous Branching

Figure 3: Partitioning Techniques.

Therefore, we introduce *heterogeneous branching* in order to accommodate more attributes by partitioning different branches of the partitioning tree on different attributes. In other words, we sacrifice the best performance on a few attributes to achieve *robustness*, i.e., improved performance over more attributes. This is reasonable as without a workload, there is no evident reason to prefer one attribute over another. Figure 3(b) shows a partitioning tree with heterogeneous branching. After partitioning on attribute A, the left side of the tree partitions on B while the right side partitions on C. Thus, we are now able to accommodate 4 attributes, instead of 3. However, attributes B and D are each partitioned on 75% of the data while attribute C is partitioned on 50%. As a result, ad-hoc queries would now gain partially over all the four attributes, which makes the partitioning more effective.

The number of attributes in the partitioning tree, with c as the minimum fraction of the data partitioned by each attribute and r as the number of replicas, is given as $\frac{1}{c} \cdot \lfloor log_n \frac{D}{P} \rfloor$. With $n = 2$, $D = 1$TB, $P = 64$MB and $c = 50\%$, the number of attributes that can be partitioned is 28. Note that the number of attributes that can be partitioned increases with the dataset size. This shows that with larger dataset sizes, upfront partitioning is even more useful for quickly finding the relevant portions of the data.

3.2 Attribute Allocation

Our goal is to allocate attributes to nodes in the heterogeneous binary partitioning tree such that all attributes have similar advantage in terms of data skipping. To do this, we define the allocation of an attribute as the weighted sum of its fanout on each of the nodes it appears in the partitioning tree T, i.e., the allocation of attribute i is given as:

$$Allocation_i(T) = \sum_{n \in nodes(T,i)} \text{DataFraction}_n \cdot \text{Fanout}_n$$

The *Allocation* defined above gives the granularity of data partitioning over an attribute. Higher allocation means more data skipping is possible. For example, in Figure 3(b), attribute B appears on two nodes, one covering 50% of the data while the other covering 25% of the data. Thus, B has an allocation of $(0.5 * 2 + 0.25 * 2) = 1.5$. With no query workload, our goal is to balance the benefit of partitioning across all attributes in the dataset. This means that same

selectivity predicates on any two attributes X and Y should have similar speed-ups, compared to scanning the entire dataset. To achieve this, we distribute the total allocation equally among all attributes. Each attribute gets an allocation of $b^{1/|\mathbb{A}|}$, where $|\mathbb{A}|$ is the number of attributes and b is the number of buckets. For instance, if there are 8 buckets, and 3 attributes, the allocation (average fanout) per attribute is $8^{1/3} = 2$. In the case where the user has prior workload information, he can provide relative weights of the attributes and the attribute allocation will be distributed proportional to these weights. We then place attributes into the tree so as to approximate this ideal allocation.

3.3 Handling Skew and Correlation

Real world datasets are often skewed and have attributes that are correlated (e.g., state and zip code). As a result, if we uniformly partition the domain range, some branches of the partitioning tree could have much more data than others resulting in unbalanced final partitions. We would then lose the benefit of partitioning due to either very small or very large partitions. To illustrate, consider two partitionings to partition a skewed dataset $D_1 = \{1, 1, 1, 2, 2, 2, 3, 4, 5, 6, 7, 8\}$ into four partitions, one based on the domain range and the other on the median value:

$$P_{\text{domain}}(D_1) = [\{1, 1, 1, 2, 2, 2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}]$$
$$P_{\text{median}}(D_1) = [\{1, 1, 1\}, \{2, 2, 2\}, \{3, 4, 5\}, \{6, 7, 8\}]$$

We observe that $P_{\text{domain}}(D_1)$ is clearly unbalanced whereas $P_{\text{median}}(D_1)$ produces balanced partitions.

Amoeba uses this idea to avoid imbalance in the partitioning tree. We do a breadth-first traversal while constructing the tree. At each node, we assign the median of the data arriving at the node on the chosen attribute as the pivot. We split the data on the pivot and use it to assign (attribute,pivot) for the left and right child nodes later. Using median from the data as pivot ensures that child nodes get equal portions of data even when there is skew and correlation. In order to find the median efficiently, we use a random sample of the data and find median at each node using the sample. We refer to Section 5 for more details on implementation.

3.4 Upfront Partitioning Algorithm

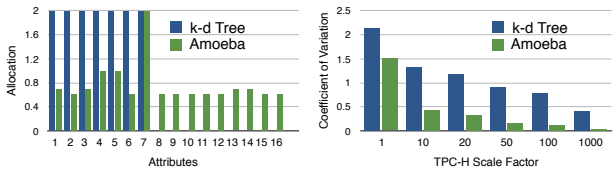
We now describe our upfront partitioning algorithm which is used to generate the partitioning tree. Algorithm 1 shows the pseudocode. The function takes in the dataset size, the maximum partition size, the allocation per attribute, and, a sample of the data, to produce the partitioning tree. It first calculates the depth of the tree to be created from the partition size and dataset size (Line 3). The algorithm initializes the queue with the root node of the tree (Line 4) and starts a breadth-first traversal to assign an attribute to every node. The attribute to be assigned at a given node is given by the function `leastAllocated`, which returns the attribute which has the highest allocation remaining. If two or more attributes have the same highest allocation remaining, we randomly choose among the ones that have occurred the least number of times in the path from the node to the root. `findMedian` returns the median of the attribute assigned to this node. This is done by finding the median in the sampled data which comes to this branch. The algorithm starts with an allocation of 2 for the root node, since we are partitioning the entire dataset into two partitions. Each time we go to the left or the right subtree, we reduce the data we operate on by half. Once an attribute is assigned to a node, we subtract from the overall allocation of the attribute (Line 13). The algorithm creates a leaf-level bucket in case we reach the maximum depth (Line 18).

Algorithm 1: `CreateTree`

```
Input : Int datasetSize, Int maxPartitionSize, Float[] allocation, Tuple[]
        initSample
1  Tree tree;
2  numBuckets ← ⌊datasetSize/maxPartitionSize⌋;
3  treeDepth ← log2(numBuckets);
4  Queue nodeQueue ← {(tree.root, treeDepth, initSample)};
5  while nodeQueue.size > 0 do
6      node,depth,sample ← nodeQueue.pollFirst();
7      if depth > 0 then
8          node.attr ← leastAllocated(allocation);
9          node.value ← findMedian(sample,node.attr);
10         lS, rS ← splitSample(node.attr, node.value);
11         node.left ← CreateNode();
12         node.right ← CreateNode();
13         allocation[node.attr] -= 2.0/2^(maxDepth - depth);
14         depth -=1;
15         nodeQueue.add((node.left, depth, lS));
16         nodeQueue.add((node.right, depth, rS));
17     else
18         node ← newBucket ();
```

3.5 Comparison with K-d Tree

We analyze our partitioning tree in comparison to k-d trees. Specifically, we implement a k-d tree that partitions on attributes in a round robin fashion, one attribute at a time, until the partition size falls below the minimum size. This emulates the standard way of performing data placement in a conventional k-d tree [7]. In contrast, our upfront partitioning algorithm places the attributes such that all attributes have similar partitioning benefits.



(a) Attribute Allocation (SF=1) (b) Variation in Allocation

Figure 4: Comparison with k-d Tree on TPC-H lineitem

Figure 4(a) shows the allocation for each attribute of TPC-H lineitem. We can see that the k-d tree has higher allocation for the first seven attributes, however the remaining nine attributes are not partitioned at all. Amoeba's partitioning tree, on the other hand, distributes the allocation more evenly across all attributes. Figure 4(b) shows the coefficient of variation in allocation over different TPC-H scale factors. Our approach has much lower variation than k-d tree and it drops sharply with data size. Thus, larger datasets get more similar data skipping benefits on all attributes. We also compared the algorithm runtime of our approach with k-d tree. Similar to k-d tree, our approach generates the partitioning tree in a few seconds.

3.6 Heterogeneous Replication

Distributed storage systems replicate data for fault-tolerance, e.g., 3x replication in HDFS. Such replication mechanisms first partition the dataset into blocks and then replicate each block multiple times. Instead, we can first replicate the entire dataset and then partition each replica using a different partitioning tree. Figure 5 shows a dataset with 2 replicas, the first one partitioned on attributes $\{A, C, D\}$ and the second on $\{B, E, F\}$. While the system is still fault-tolerant (because it has the same degree of replication), recovery becomes slower because we need to read several or all replica blocks in case we have a block failure. Essentially, we sacrifice fast recovery time for improved ad-hoc query performance.

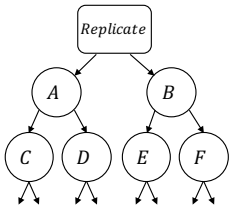


Figure 5: Heterogeneous Replication.

We know from Section 3.1 that the number of attributes accommodated in the tree is $a = \frac{1}{c} \cdot \lfloor log_n \frac{D}{P} \rfloor$. Having r replicas allows us to have $a * r$ number of attributes, with a attributes per replica or increase the c for each attribute. Both these lead to improved query performance due to greater partition pruning. There are interesting open questions here, such as how to partition the attributes across replicas and how to move them later, which we plan to explore as future work.

4 ADAPTIVE REPARTITIONING

Upfront data partitioning allows an analyst to quickly get started with her ad-hoc queries. However, she would want the partitioning to also adapt as her analysis progresses, e.g., drilling down web click data into successively smaller age groups, to provide even better query performance. Amoeba provides an adaptive query executor to achieve this.

When a query is submitted: (1) the *optimizer* explores alternative partitioning trees to find the best one and decides whether repartitioning is worthwhile, and (2) the *plan executor* runs the chosen plan. The optimized plan only accesses data which is to be read by input queries, i.e., we do not access data that is not read by queries during repartitioning. This has two benefits: (i) we never repartition data that is not touched by any query, and (ii) query processing and repartitioning share scans reducing the cost of repartitioning.

In the rest of this section, we describe the optimizer and defer discussion about the plan executor to Section 5.2. In detail, we first discuss our workload monitor and cost model. We then introduce three basic transformations used to transform a given partitioning tree. Next, we describe a bottom-up algorithm to consider all possible alternatives generated from the transformation rules for inserting a single predicate. Last, we discuss how to handle multi-predicate queries. It is worth noting that the entire optimization process is transparent to users, i.e., users do not have to worry about making repartitioning decisions and their queries remain unchanged with the new access methods.

4.1 Workload Monitor and Cost Model

Amoeba maintains a history of the queries seen by the system, called the query window (W). To exclude older queries that are stale and not representative of the queries to be seen, we restrict the window to contain only queries that happened in the past X hours[2]. For each query q in the query sequence, the cost of processing q using partitioning tree T is given as:

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

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 cost of the query window is the sum of the cost of individual queries. For a query being executed, the optimizer might want to transform the partitioning tree to a new partitioning tree T' resulting in a set of buckets $B \subset lookup(T, q)$ being repartitioned. The benefit of this transformation is:

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

and the added cost of repartitioning is given as:

$$\text{RepartitioningCost}(T, q) = c \sum_{b \in B} n_b$$

where c is the write-multiplier i.e., how expensive writes are compared to a read. The value of c is obtained empirically by modeling the increased query runtime due to repartitioning. For our evaluation setup we got $c = 4$. Repartitioning is expensive, however it only happens when the resulting decrease in the cost of the query window (benefit) is greater than the repartitioning cost. This check prevents constant re-paritioning due to a random query sequence and bounds the worst-case impact. To illustrate, consider a tree with one node and a query sequence of the form $\sigma_{A<2}, \sigma_{B<2}, \sigma_{A<2}, \sigma_{B<2}$.... In this case, we do not constantly repartition the data. After doing it once, say on A, the total cost goes down and hence the repartitioning on B would not happen as Benefit < RepartitioningCost.

4.2 Partitioning Tree Transformations

We now describe a set of transformation rules to explore the space of possible plans when repartitioning the data. For now, we restrict ourselves to a query with a single predicate of the form $A \leq p$, denoted as A_p. Later in Section 4.4, we discuss how to handle other predicate forms and multiple predicates.

Our approach is to consider partitioning transformations that are local, i.e., that do not involve rewriting the entire tree. These local transformations are cheaper and amortizes the repartitioning effort over several queries. Amoeba considers the following three basic partitioning transformations:

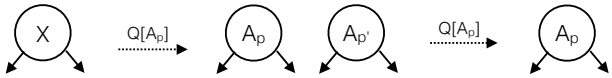
(1) Swap is the primary data transformation in Amoeba. It replaces an existing node in the partitioning with the incoming query predicate A_p. As we repartition only the accessed data, we consider swapping only those nodes whose left *and* right children are fully accessed by the incoming query. Applying swap on an existing node

[2] X is a parameter in adaptive query executor and it determines how quickly the system reacts to workload changes. We used $X = 4$ for our experiments.

Transformation	Notation	Cost (C)	Benefit (B)
Swap	$P_{\text{swap}}(n, n')$	$\sum_{b \in T_n} c \cdot n_b$	$\sum_{i=0}^{k}[\text{Cost}(T_n, q_i) - \text{Cost}(T_{n'}, q_i)]$
Pushup	$P_{\text{pushup}}(n, n_{\text{left}}, n_{\text{right}})$	$C(P(n_{\text{left}})) + C(P(n_{\text{right}}))$	$B(P(n_{\text{left}})) + B(P(n_{\text{right}}))$
Rotate	$P_{\text{rotate}}(p, p')$	$C(P(n_{\text{left\|right}}))$, for p' on $n_{\text{left\|right}}$	$B(P(n_{\text{left\|right}}))$, for p' on $n_{\text{left\|right}}$
None	$P_{\text{none}}(n)$	$C(P(n_{\text{left}})) + C(P(n_{\text{right}}))$	$B(P(n_{\text{left}})) + B(P(n_{\text{right}}))$

Table 1: The cost and benefit estimates for different partitioning tree transformations.

involves reading both sub-branches, and restructuring all partitions beneath the left subtree to contain data satisfying A_p and the right subtree to contain data that does not satisfy A_p. Swaps can happen between different attributes (Figure 6(a)), in which case both branches are completely rewritten in the new tree. Swaps can also happen between two predicates of the same attribute (Figure 6(b)), in which case the data moves from one branch to the other. For example, in the Figure 6(b), if node $A_{p'}$ is A_{10} and predicate A_p is $A \leq 5$, then data moves from the left branch to the right branch, i.e., the left branch is completely rewritten while the right branch just has new data appended.



(a) Different Attribute Swap (b) Same Attribute Swap

Figure 6: Node swap in the partitioning tree.

Swaps serve the dual purpose of un-partitioning an existing (less accessed) attribute while refining on another (more accessed) attribute. As both the swap attributes as well as their predicates are driven by the incoming queries, they reduce the access times for the incoming query predicates. Finally, note that it is cheaper to apply swaps at lower levels in the partitioning tree because less data is rewritten. Applying them at higher levels of the tree results in a much higher cost.

(2) Pushup has as goal of pushing a predicate as high up the tree as possible. This can be done when both the left and the right child of a node contain the incoming predicate, as a result of a previous swap, as shown in Figure 7. This is a logical partitioning tree transformation, i.e., it only involves rearranging the internal nodes without any modification of the contents of leaf nodes[3].

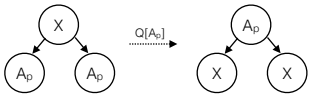


Figure 7: Node pushdown in partitioning tree.

We check for a pushup transformation every time we perform a swap transformation. The idea is to move important predicates (i.e., ones that have recently or frequently appeared in the query sequence) progressively up the partitioning tree, from the leaves right up to the root. This makes important predicates less likely to be swapped immediately, because swapping a node higher in the partitioning tree is much more expensive. Another advantage of pushup is that it causes a churn of the attributes assigned to higher-up nodes in the partitioning tree. When such a dormant node is pushed down, subsequent predicates can swap them in an incremental fashion, affecting fewer branches. Overall, node pushup allows Amoeba to naturally cause less important attributes to be repartitioned more frequently, thereby striking a balance between adaptivity and robustness. Note that if possible, a pushup always happens as there is no cost associated with doing it.

(3) Rotate transformation rearranges two predicates on the same attribute such that more important (recently accessed or frequently appearing in the query sequence) predicate appears higher up in the partitioning tree. Figure 8 shows a rotate transformation involving predicates p and p' on attribute A. The goal here is to churn the partitioning tree such that predicates on less important attributes are more likely to be replaced first. Similar to the pushup transformation, rotate is a logical transformation, i.e., it only rearranges the internal nodes of the partitioning tree and always happens if possible.

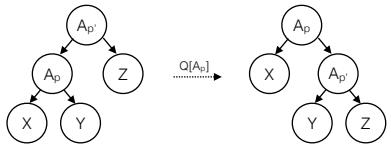


Figure 8: Node rotation in partitioning tree.

Discussion These three partitioning tree transformations can be further combined to capture a fairly general set of repartitioning scenarios. Figure 9 shows such an example. Starting from an initial partitioning tree, we first swap nodes D_4 with incoming predicate A_2 at the lower level. Then, we pushup A_2 one level above and finally rotate with nodes A_5 and C_3. In the process, we end up repartitioning only half the leaves. As we show later in evaluation, in larger trees, repartitioning mostly happens on small fractions of the data modifying a few subtrees locally. Swapping nodes based on incoming query predicates may introduce skew i.e., some leaves having more tuples compared to others. This is not a problem as our cost model ensures that the skew is actually beneficial if it arises. For example, if we have a node $A_{0.5}$ and many queries access $A \leq 0.25$, where A is uniformly distributed $(0, 1)$, it is beneficial to replace $A_{0.5}$ with $A_{0.25}$ in the tree even though it introduces skew. In the next section, we describe how we generate alternate partitioning trees using these transformations.

4.3 Divide-And-Conquer Approach

Given a query with predicate A_p and a partitioning tree T, there are many different combinations of transformations that need to be considered. We propose a divide-and-conquer approach to explore the space of all alternate repartitioning trees generated by applying the transformation rules. Observe that the data access costs over a subtree T_n, rooted at node n, could be broken down into the access costs over its subtrees, i.e.,

$$\text{Cost}(T_n, q_i) = \text{Cost}(T_{n_{left}}, q_i) + \text{Cost}(T_{n_{right}}, q_i)$$

[3]In this case, the swap must have happened in one of the child subtrees.

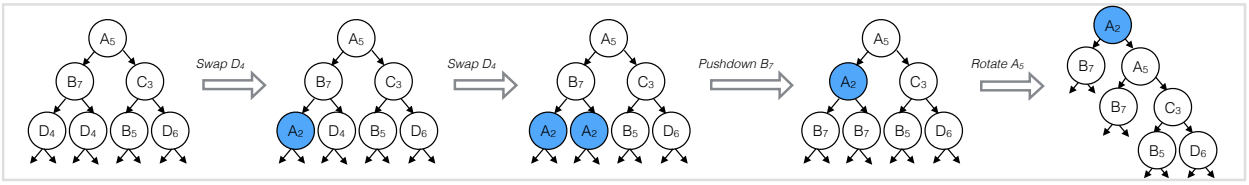


Figure 9: Introducing predicate A_2 into the partitioning tree.

Algorithm 2: `getSubtreePlan`

```
   Input  :Node node, Predicate pred
1  if isLeaf(node) then
2      return P_none(node);
3  else
4      if isLeftAccessed(node) then
5          leftPlan ← getSubtreePlan(node.lChild, pred);
6      if isRightAccessed(node) then
7          rightPlan ← getSubtreePlan(node.rChild, pred)
       /* consider swap                                          */
8      if leftPlan.fullyAccessed and rightPlan.fullyAccessed then
9          currentCost ← ∑_i Cost(node,q_i);
10         whatIfNode ← clone(node);
11         whatIfNode.predicate ← newPred;
12         swapNode(node, whatIfNode);
13         newCost ← ∑_i Cost(whatIfNode,q_i);
14         benefit = currentCost - newCost;
15         if benefit > 0 then
16             updatePlanIfBetter(node, P_swap(node,whatIfNode));
       /* consider pushup                                        */
17     if leftPlan.ptop and rightPlan.ptop then
18         updatePlanIfBetter(node, P_pushup(node, node.lChild, node.rChild));
       /* consider rotate                                        */
19     if node.attribute == predicate.attribute then
20         if leftPlan.ptop then
21             updatePlanIfBetter(node, P_rotate(node, node.lChild));
22         if rightPlan.ptop then
23             updatePlanIfBetter(node, P_rotate(node, node.rChild));
       /* consider doing nothing                                 */
24     updatePlanIfBetter(node, P_none(node));
25     return node.plan;
```

where, $T_{n_{left}}$ and $T_{n_{right}}$ are subtrees rooted respectively at the left and the right child of n. Thus, finding the best partitioning tree can be broken down into recursively finding the best left and right subtrees at each level, and considering parent node transformations only on top of the best child subtrees. For each transformation, we consider the benefit and cost of that transformation and pick the one which has the best benefit-to-cost ratio. Table 1 shows the cost and benefit estimates for the different transformations. For the swap transformation, denoted as $P_{\text{swap}}(n, n')$, we recalculate the query costs. However, pushup and rotate transformations, denoted as $P_{\text{swap}}(n, n')$ and $P_{\text{pushup}}(n, n_{\text{left}}, n_{\text{right}})$ respectively, inherit the costs from children subtrees. We also consider applying none of the transformations at a given node, denoted as $P_{\text{none}}(n)$. This approach helps to significantly reduce the candidate set of modified partitioning trees.

Algorithm 2 shows our divide-and-conquer approach to find the best repartitioning plan. The algorithm recursively finds the best plan for each subtree until we reach the leaf. If leaf, we return a do-nothing plan (Lines 1–2). If not, we first check if the left subtree is accessed, if yes we recursively call `getSubtreePlan` to find the best plan for the left subtree (Lines 4–5). Similarly, for the right subtree (Lines 6–7). The output plan contains the transformation applied at the node (*ptop* to indicate if A_p is the root after the plan is applied, $fullyAccessed$ to indicate if the entire subtree is accessed, and *cost* and $benefit$ calculated using Table 1), and pointers to the left and right child plan.

Once we have the best plans for the left and right subtrees, we first consider swap rule (Lines 8–16). Swapping happens only if both the subtrees are fully accessed. Otherwise, we will need to access additional data in order to create new partitioning. We perform a *what-if* analysis to analyze the plan produced by the swap transformation. This is done by replacing the current node with a hypothetical node having the incoming query predicate. We then recalculate the new bucket counts at the leaf level of this new tree using the sample. We now estimate the total query cost with the hypothetical node present. In case the what-if node reduces the query costs, i.e., it has benefits, we update the transformation plan of the current node. The update method checks whether the benefit-cost ratio of the new plan is greater than that of the best plan so-far. If so, we update the best plan.

Similarly, we apply the pushup and rotate transformation if possible (Lines 17–23). Both transformations are logical; hence, we simply check whether the pushdown results in better benefit-cost ratio. Finally, we check whether no transformation is needed, i.e., we simply inherit the transformations of the child nodes (Line 24). The algorithm returns the best plan (Line 25). The algorithm has a runtime complexity of $O(QNlogN)$ where N is the number of nodes in the tree and Q is the number of queries in the query window.

4.4 Handling Multiple Predicates

So far, we assumed that a predicate is always of the form $A \leq p$. It gets inserted in the tree as A_p and on insertion, only the leaf nodes on the left side of the node are accessed. $A > p$ is also inserted as A_p with the right side of the node being accessed. For $A \geq p$ and $A < p$, let p' be $p - \delta$ where δ is the smallest change for p's data type. We insert $A_{p'}$ into the tree. $A = p$ is treated as combination of $A \leq p$ and $A > p'$.

Let us now consider queries with multiple predicates. Consider a simple query with two predicates A_p and A_{p2}. The brute force approach is to consider choosing a set of accessed non-terminal nodes to be replaced by A_p and then for every such choice, choose among the set of remaining nodes, nodes to be replaced by A_{p2}. Thus, the number of choices grows exponentially with the number of predicates. Amoeba uses a greedy approach to work around this exponential complexity. For each predicate in the query, we try to insert the predicate into the tree. We find the best plan for that predicate by calling $getSubtreePlan(root, p_i)$ for the i^{th} predicate.

Algorithm 3: `getBestPlan`

```
Input  : Tree tree, Predicate[] predicates
while predicates ≠ ϕ do
    prevPlan ← tree.plan;
    foreach p in predicates do
        Plan newPlan ← getSubtreePlan(tree.root, p);
        updatePlan(tree.root, newPlan, p);
    if tree.plan ≠ prevPlan then
        remove from predicates the newly inserted predicate;
    else
        break;
if tree.plan.benefit > tree.plan.cost then
    return tree.plan;
else
    return null;
```

We take the best among the best plans obtained for different predicates and remove the corresponding predicate from the predicate set. We then try to insert the remaining predicates into the best plan obtained so far. The algorithm stops when either all predicates have been inserted or when the tree stops changing. Doing this adds a multiplicative complexity of $O(|P|^2)$ where P is the set of query predicates.

The best plan obtained is applied on the current partitioning tree and a new one is generated only if the plan's benefit is greater than the cost. Here, plan's benefit is the total reduction in the cost of the query window. If benefit is less than cost, we ignore it; no repartitioning happens, and we just filter out the partitions based on the current partitioning tree. We do this check at the end and not after every predicate insertion as a single predicate insertion may not generate enough benefit, however subsequent insertions may generate enough to pay for the cost of repartitioning. Algorithm 3 shows the pseudocode.

5 AMOEBA USING SPARK AND HDFS

We implement Amoeba on top of HDFS, with Spark used for query processing. Still, one can easily implement our ideas on any block-based distributed system. Also, as we show with SparkSQL, integrating heterogeneous partitioning into an existing system is straightforward as most changes are below the storage layer. The Amoeba storage system has around 8,000 lines of code and comprises of two modules: (i) a data loader module; and (ii) a query processor module.

5.1 Data Loader

This module bulk loads raw input files into HDFS, partitioned across all attributes. It first builds an initial partitioning tree and then creates the data blocks accordingly.

Tree Construction. When constructing the partitioning tree on a cluster of machines, we collect the sample in parallel using block sampling. Later, we combine the samples (via HDFS) and pass the sampled records and attribute allocations to the *tree builder* (Algorithm 3) on a single machine to produce a single partitioning tree across the entire dataset. The index is serialized and stored as a file on HDFS. By default, we use uniform allocation for the attributes. However, the user may use prior workload knowledge to set a custom allocation and get better performance (see Section 6.4).

Data Blocking. The second phase takes the partitioning tree and the input files as input and creates the data blocks. During this phase, in parallel on each machine, we scan the input files. For each tuple, we use the partitioning tree to find the leaf node that it lands in. We use a *buffered stream* to collect the tuples belonging to each partition (leaf node) separately and buffer them before flushing to the partition in the underlying file system (e.g. HDFS). As multiple machines might write to the same partition, we employ partition-level distributed exclusive locks (via Zookeeper [3]) to synchronize writes. Our current implementation creates a different HDFS file for each partition in the dataset. With the current main memory and CPU capacities, having a file per partition (with partitions in the order of ten thousand) does not lead to any observable slowdown [14]. However, future work could also integrate Amoeba deeply within HDFS to generate a single file with each block holding one partition.

Bulk Updates. Updates can change the median values of the existing partitioning tree and require us to reshuffle the data entirely. To avoid this, Amoeba creates a new partitioning tree for each new bulk update. The new partitioning tree uses the same attribute allocation; as it has the same schema, however, we recompute the medians based on the data distribution in the update. At query time, the Amoeba query executor looks up all partitioning trees on a table in order to compute the relevant data blocks. Streaming appends could be handled using techniques from column store where a small "write optimized" buffer stores the incoming records, which are periodically inserted as a bulk update [10, 30]. This creates a date-partitioned set of datasets, each of which is internally partitioned using its partitioning tree.

5.2 Query Processor

Amoeba can be a standalone storage system or a data source for systems like SparkSQL. Consider a simple query:

```
spark.read.amoeba('employee')
  .where('age > 30').count()
```

Internally SparkSQL does the predicate pushdown and passes to Amoeba a predicated scan query $< employee, (age > 30) >$. There are two main parts involved in running this query: (i) creating an execution plan which may involve repartitioning some or all the of the data that is being accessed by the query, and (ii) actually executing the plan.

Optimizer. The optimizer is responsible for generating an execution plan for the given query. It reads the current tree file from HDFS and uses Algorithm 3 to check if it is feasible to improve the current partitioning tree. Note that while creating the plan, we also end up filtering out partitions that do not match any of the query predicates. For example, if there is a node A_5 in the tree and one of the predicates in the query is $A \leq 4$, then we don't have to scan any of the partitions in right subtree of the node. If the plan changes the tree, we write out the new partitioning tree to HDFS. From the plan, we now get two set of buckets: (i) buckets that will just be scanned, and (ii) buckets that will be repartitioned to generate a new set of buckets.

Plan Executor. Amoeba uses Spark for executing queries. We construct a Spark job from the plan returned by the optimizer. We split each of the two sets of buckets into smaller sets called tasks. A task contains a set of buckets such that the sum of the sizes of the buckets is not more than 4GB. Each task reads the blocks from HDFS in bulk and iterates over the tuples in main-memory. Tasks created from set 1 run with a scan iterator which simply reads a tuple at a time from the buckets and returns the tuple if it matches the predicates in the query. Tasks created from set 2 run with a distributed repartitioning iterator. The iterator reads the tree from HDFS. For each tuple, the iterator looks up in the new partitioning tree to find its new partition id in addition to checking if it matches the query predicates. It then re-clusters the data in main-memory according to the new partitioning tree. Once the buffers are filled, the repartitioner flushes the new partitions into physical files on HDFS.

Amoeba uses partition-level distributed exclusive locks (via Zookeeper) to ensure consistency. First, multiple queries can execute concurrently and might end up repartitioning the same buckets. To prevent this, before a plan involving repartitioning executes, it acquires exclusive locks on all the buckets it wants to repartition. If it fails to get the locks before timeout, the plan just scans both sets of buckets. This scheme allows multiple queries to repartition data simultaneously if they are modifying different subtrees. After repartitioning, the index file is updated and the old buckets are garbage collected at a later point when no there no queries using them. Second, when the optimizer decides to repartition a large subtree, the repartitioning work may end up being distributed across several tasks. To ensure writes are done atomically, workers use locks on the new partitions being created to coordinate while flushing data to them. As a result, each partition resides in a single file across the cluster.

Tasks are executed independently by the Spark job manager across all machines and the result is exposed to users as a Spark RDD. Users can use these RDDs to do more analysis using the standard Spark APIs, e.g., run an aggregation.

6 EXPERIMENTS

We evaluate the system on the following four metrics: *Upfront overhead*, i.e., the initial partitioning overhead; *First query runtime*, i.e., the initial query performance; *Break-even point* i.e., number of queries to recover the cost of upfront partitioning, and; *Total gain*, i.e., the ratio of the total runtime of all queries using the system.

We divide the experiments into five sections: (i) we examine the benefits and overheads of upfront data partitioning, (ii) we study the repartitioning efficiency, (iii) we show how to improve query performance by using query workload information, (iv) we validate Amoeba using a real world workload from an IoT startup and (v) we present detailed micro benchmarks.

6.1 Experimental Setup.

Our testbed consists of a cluster of 10 nodes. Each node has 16 2.07 GHz Xeon cores, running on Ubuntu 12.04, 128 GB main-memory, and 11 TB of disk storage. The Amoeba storage system runs on top of Hadoop 2.6.0 and uses Zookeeper 3.4.6 for synchronization. We run queries using Spark 1.6.1, with Spark programs running on Java 7. All experiments are run with cold caches. We use the following two datasets in our experiments:

TPC-H is a ad-hoc decision benchmark and reflective of ad-hoc workloads generated from templates in real-world application. We use the TPC-H benchmark with scale factor of 200. To focus on the effects of reduced table scan, we denormalize all the tables against the *lineitem* table, which results in a single table of roughly 1.2 billion rows and **1.4TB** in size. We evaluate all TPC-H query templates that have selection predicates on *lineitem* table, namely $(q_3, q_5, q_6, q_8, q_{12}, q_{14}, q_{19})$[4]. The FROM clauses in these templates were changed to use the denormalized table. To remove the runtime difference across queries due to different aggregates being computed, we replace the select clause with $COUNT(*)$. This choice of workload is common to other papers that evaluate partitioning techniques in distributed databases [31].

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 103 queries, run between 04/19/2015 and 04/21/2015, on the trip data. The queries can be modelled as 5 sets of query patterns with a sequential shift from one query pattern to another. There are instants where two neighboring query patterns are active at the same time. Each query pattern looks at a time window and filters on a set of attributes. Some of the predicates in a query pattern are static (for example: predicate on `company_id`) while others change (e.g. predicate on `trip_start`, `phone_model`, etc.).

6.2 Upfront Partitioning Performance

We start studying the impact of doing upfront partitioning. We analyze two aspects: the added overhead as a result of doing upfront partitioning (i.e., *upfront overhead*) and the benefit of doing upfront data partitioning (i.e., *time to first query*).

Data Loading Overhead. As mentioned earlier, loading data into the Amoeba storage manager is a three-step process. The loader does block-sampling in parallel across all the nodes to generate a sample of the data. Then, on a single machine the partitioning tree algorithm is used to generate a partitioning tree. Finally, the data is partitioned and stored in HDFS 3-way replicated. To look at the overhead involved, we compare the data upload time in Amoeba against the standard data upload time in HDFS using the command line interface (CLI). In this experiment, we use the denormalized TPC-H data. The data is uniformly distributed across all the machines to begin with, each machine has $1/10^{th}$ of the data. The data is loaded in parallel from all the machines in both methods.

[4]Our focus in this work is data skipping and therefore we only consider selection predicates. In [22], we show how this approach applies to join predicates.

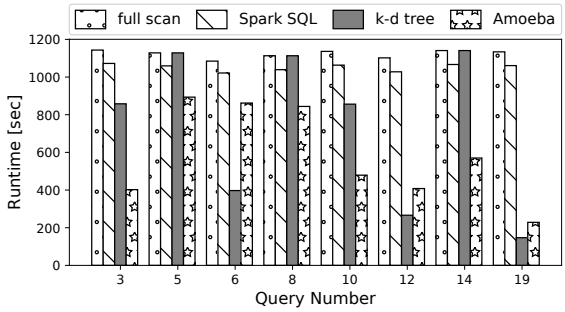


Figure 10: Upfront query runtimes of the TPC-H queries

For a fair comparison, Amoeba preserves the same format (row, uncompressed) and layout (text) as in standard HDFS, i.e., it only differs in how the data is partitioned.

The total upload time for Amoeba is 1.38 times that of HDFS CLI. The breakdown of the total time shows that the sample generation step takes 122s. The partitioning tree creation step takes 14s. Writing out the partitions takes 2890s which is only slightly higher than the total upload time using the HDFS CLI which takes 2190s. This is because the step is IO-bound and the amount of data being uploaded by both methods into HDFS is the same. The difference is due to the Zookeeper synchronization overhead and the cost of doing multiple appends to create the partitions. Finally, the overhead is similar to other workload-specific data preparation, such as indexing and co-partitioning [13, 27]. Amoeba can also read data directly from HDFS. We observed an additional 5% slowdown when data is read from HDFS.

Ad-hoc Query Processing. Given that Amoeba distributes the partitioning effort over all attributes, ad-hoc queries are expected to show a benefit right from the beginning. To illustrate this, we ran the 8 TPC-H queries selected one at a time independently. We compare the query runtime of each query on Amoeba against the following alternatives: `full scan`, which uses Amoeba but it does not prune the partitions, instead it reads all partitions and applies the query predicates; `Spark SQL`, which uses Spark SQL to run the entire query and hence it ends up doing a full scan, and; `k-d tree`, which partitions attributes in a round robin fashion, one attribute at a time, until the partition size falls below the maximum partition size. This emulates the standard way of performing data placement in a conventional k-d tree [7].

Figure 10 shows the results. Comparing `full scan` and `Spark SQL`, we observe that the latter does slightly better. This is because (i) the raw dataset is partitioned into 200 files while Amoeba stores the dataset as a collection of 8,192 files, and (ii) `full scan` ends up reading the index file too. Still, together this leads to only around 7% overhead. So, even though we are reading more partitions and an extra index file, our implementation is still competitive. Now, comparing `full scan` and Amoeba shows that the upfront partitioning indeed helps, and all the queries have improved query performance. Overall, we get 44.8% improvement over full scan due to the upfront partitioning.

Both k-d tree and Amoeba's partitioning tree have 13 levels, which results into 8,192 blocks. K-d tree partitions one attribute at a time in a round-robin fashion which results in only 13 attributes of the 45 attributes in the denormalized TPC-H

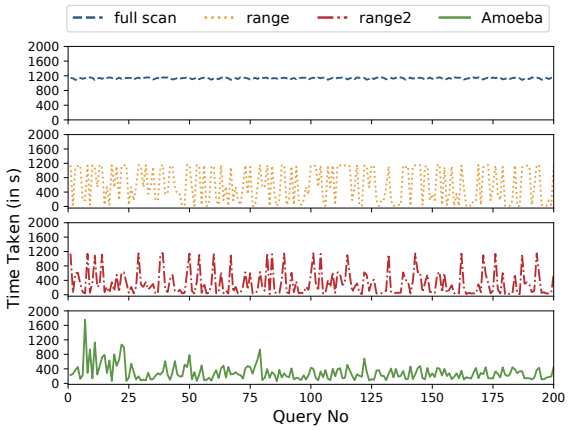


Figure 11: Running the queries on the TPC-H dataset

schema being accommodated in the k-d tree. It ends up doing full scans in case the attributes queried were not used in the k-d tree. In this instantiation, k-d tree has 4 attributes which filter blocks: ($c\_mktsegment, l\_quantity, l\_returnflag, l\_shipmode$) and the other 9 attributes don't occur in any query. As a result, (q_5, q_8, q_{14}) end up being full scans while (q_6, q_{12}, q_{19}) end up being better than Amoeba as `k-d tree` is partitioned on the attributes needed by those queries. Amoeba uses heterogeneous branching in the partitioning tree, which results in partitioning on many attributes partially. As TPC-H queries have multiple predicates, each predicate ends up independently filtering out a fraction of the blocks. Thus, every query shows improved performance in Amoeba with lesser variance in runtime across queries. On average, Amoeba does 20% better than using the k-d tree based partitioning.

Thus, Amoeba not only provides improved *time to first query* with low *upfront overhead*, but also provides improvements across several attributes.

6.3 Adaptive Query Executor Performance

To show the effectiveness of the adaptive query executor, we ran a workload of 200 queries generated by random initialization of the 8 query templates listed in Section 6.1 on the TPC-H dataset. We compare the end-to-end query runtimes using Amoeba against the following alternatives: `full scan`, which uses Amoeba but without partition pruning; `range`, which is a workload-oblivious partitioning on *o_orderdate*, creating one partition per date. The partitioning tree thus created has all nodes with *o_orderdate* attribute and results in roughly 2300 partitions, and; `range2`, which is the workload-based multi-dimensional partitioning scheme of Sun et al. [31], created based on the frequently queried columns in the workload. The scheme range partitions the data on *o_orderdate* (64 partitions), *r_name* (4 partitions), *c_mktsegment* (4 partitions), and *quantity* (8 partitions). This results in 8,192 partitions, the same as the number of partitions generated by the upfront partitioner in Amoeba.

Figure 11 shows the query runtimes using the different approaches. In `full scan`, all the queries take approximately the same amount of time. In `range`, several queries end up performing full scans as 3 of the query templates (q_6, q_{12} and q_{14}) do not filter on *o_orderdate*. In `range2`, only queries from query template q_{12}

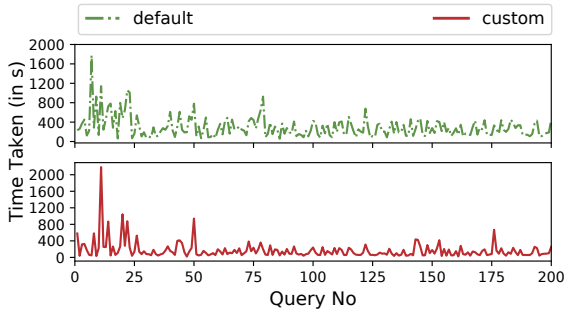


Figure 12: Query runtimes using the custom initialization

end up being full scans and all other queries show improved query performance. Amoeba starts out with no workload information. As a result, index re-organizations are initially more disruptive. However over time the re-organizations become less costly. In the last 20 queries, the query executor repartitions parts of the data 6 times but there are no large spikes, as only a small fraction of the data accessed is being re-organized. The repartitioning overhead makes only 2 queries take more time than doing a full scan. In terms of the total workload runtime, `range` improves performance by 1.88x, `range2` improves it by 3.48x and Amoeba improves it by 3.84x compared to `full scan`. More importantly, Amoeba started with no workload information while `range2` had prior information about the frequently accessed columns.

Thus, Amoeba's partitioning tree improves as the dataset is queried and eventually outperforms the workload-aware scheme `range2` due to its adaptive partitioning approach.

6.4 Better Initialization

As shown in the previous subsection, Amoeba outperforms `range2`, but only by 0.36x. This is because `range2` is already partitioned on the frequently accessed attributes. In this section, we give the upfront partitioner as much information as `range2` to begin with and compare against `range2` and default Amoeba. Specifically, instead of starting with uniform allocation for all attributes, we set the following custom allocation: *(o_orderdate, r_name, c_mktsegment, quantity) = (12,4,4,6)*. This is identical to the allocations calculated on `range2`.

Figure 12 shows the query runtimes for the same set of queries used in the previous section. In the figure, `custom` is Amoeba using the custom allocation to begin with and `default` is the same as Amoeba in Figure 11. Amoeba with this custom allocation improves the end-to-end workload runtime by 6.67x compared to full scan versus 3.48x achieved using `range2` and 3.84x achieved using uniform allocation. As the partitioning tree created by upfront partitioner already contains the frequently accessed attributes, fewer repartitionings are triggered, leading to much better performance. Note that the spike in the beginning with `custom` is larger than with `default`. In `default`, the data is partially partitioned on all the attributes while in `custom` it contains only 4 attributes. In this instance, the optimizer decides to do partial partitionings on *l_recieptdate* and *l_shipmode*, both of which aren't in the tree resulting in close to a full repartitioning. New attribute insertions are less expensive if the data is already partially partitioned on the attribute. In the end, after running the workload, the tree that started with

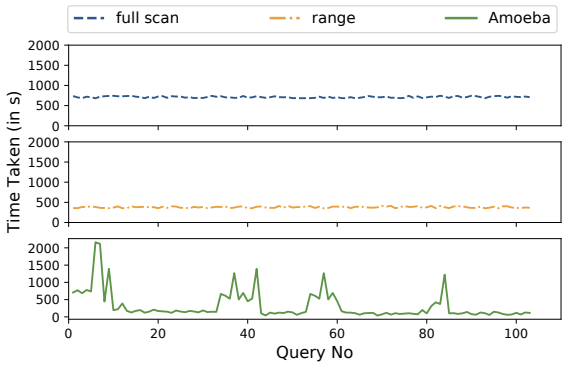


Figure 13: Query runtimes on the IoT dataset

4 different attributes ends up having all the 15 attributes being queried across the 8 query templates.

The optimal performance is achieved when a partitioning layout can filter out all the tuples that don't match the query predicates. Note that this upper bound may not be achievable due to the constrain on having a fixed number of blocks. To check how close `default` and `custom` are to the optimal, we measure the average of the ratio of number of tuples skipped due to the partitioning tree to the number of tuples that don't match the query predicates across the 8 query templates. After running the workload, the ratio for `default` is 0.80 while the ratio for `custom` is 0.90. Amoeba indeed comes pretty close to the optimal partitioning.

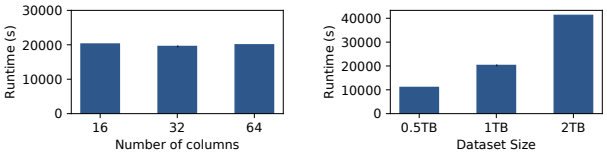
Thus, we see that Amoeba can exploit upfront workload information and match (or even outperform) workload-based partitioning. Amoeba can capture both static workload-based and online adaptive partitionings.

6.5 Amoeba on a Real Workload

We ran the exact queries from the workload associated with the IoT dataset (run in the same order as in the trace) on our synthetic version of the dataset on our testbed. We compare the performance of queries on Amoeba against: `full scan` and `range`, which has the data partitioned by upload_time (the best performing single-attribute partitioning).

Figure 13 shows the per-query runtime of the different approaches. The entire workload took 5.71 hours to run on the Amoeba storage system, compared to 20.36 hours with `full scan` (3.6x improvement) and 10.86 hours with `range` (1.9x improvement). `range`' s performance being static is an artifact of the synthetic data generation which distributes the data uniformly across all dates which may not be the case in the real dataset. In Amoeba, the first query does only slightly better than full scan, due to large number of attributes. However, Amoeba quickly adapts the partitioning to actual attributes and predicates encountered in the workload. Note that out of 148 columns in the dataset, only 18 attributes are used in the predicates. Amoeba repartitions the data 17 times in total and the peak repartitioning cost is 2, 126s, which is three times the scan cost.

In the query workload, there is a sequential shift from one query pattern to another with at most 2 patterns active at any given time. This is in contrast to the TPC-H workload, which had 8 query patterns (templates) interleaved. The shift in query pattern is reflected



(a) Dataset size 1TB fixed (b) Number of columns 16 fixed

Figure 14: Load time variation

as spikes in the Amoeba subplot. The reductions in runtime following a spike are significant as Amoeba can exploit the presence of static predicates in the query patterns. Finally, as noted in the introduction, there is no starting partitioning that works well for this data set, as even after seeing 80% of the workload, the remaining 20% still contained 57% new queries.

Thus, we see that Amoeba is useful for real-world ad-hoc query workloads, where we need to quickly access different subsets of data for further analysis.

6.6 Micro Benchmarks

Optimizer Overhead: Every query first goes to the optimizer which evaluates if the index can be improved. The optimizer runtime varies from $< 1s$ to $10s$ depending on the number of choices to be evaluated. Most of the time is spent in the what-if analysis of the swap transformation which involves finding median of an attribute in the sampled data. The optimizer runtime is small compared to the query runtime, which takes several minutes. So, we did not attempt to significantly optimize it. However, since each predicate is evaluated independently, they can also be evaluated in parallel should optimization time be problematic.

Load Time Scaling: To evaluate the scaling of upfront partitioner, we consider two settings: 1) varying columns 2) varying data size. For varying columns, we construct a synthetic dataset of 160 billion random integers and shape it into 16, 32 and, 64 columns. The size of the dataset is constant at 1.02 TB on disk with only the number of columns is varying. Figure 14(a) shows the results. The time is almost a constant as load time is proportional to data size and types and, independent of the number of columns. For varying data sizes, we keep the number of columns at 16 and vary the data sizes as 500GB, 1TB and, 2TB. Figure 14(b) shows the results. The time increases proportional to the data size. The depth of the tree increases by 1 each time we double the data size, however we didn't notice any impact on performance.

The Worst Case: Amoeba uses an online algorithm. Every time re-partitioning happens, the system spends effort with the hope that future queries would benefit from it. To examine the worst case, we consider two scenarios. First, using the TPC-H data we constructed a workload of 100 queries where each query has a range filter on a random attribute with 10% selectivity. In this case, Amoeba avoids doing any re-partitioning. This is because, in every tree transformation, some nodes get replaced based on the query predicates and re-partitioning happens only if the benefit generated is greater than cost of re-partitioning. When attributes are chosen randomly, replacing any existing node will increase the cost for the query using that attribute, making it unlikely to be replaced. In this case, Amoeba still does 33% better than Spark SQL due to the upfront partitioning. Second, for the same TPC-H data, we create one query on a random attribute with 10% selectivity and run it 5 times, then repeated the same process 3 times resulting in 20 queries run on 4 different attributes. In each of the 4 cases, we end up re-partitioning the data and never using it more than once, which is the worst case. This results in overall 20% slowdown compared to doing full scans.

7 RELATED WORK

Several database partitioning techniques have been proposed in the past including fine-grained partitioning [12], hybrid of fine- and coarse-grained [28], skew-aware partitioning [26], deep integration with the optimizer [24], interdependence of difference design decisions [34], and integrating vertical and horizontal partitioning decisions [4]. MAGIC aims at supporting *declustering* data on multiple attributes [17]. All these techniques, however, are workload-driven, and require that a workload is either provided upfront or monitored and collected over time. Oracle and MySQL support *sub-partitioning* to create nested partitions on multiple attributes. IBM DB2 supports multi-dimensional clustering tables to cluster data along multiple dimensions and build block-based indices on them [23]. These are still static and need to be reconfigured every time the workload changes.

Big data storage systems, such as HDFS, partition datasets based on size. Developers can later create attribute-based partitioning using a variety of data processing tools, e.g. Hive [1] and SCOPE [33]. However, such a partitioning is no different than traditional database partitioning as (i) partitioning is a static one-time activity, and (ii) the partitioning keys must be known a-priori and provided by users. Recently, [31] proposed to create data blocks in HDFS based on the features extracted from each input tuple. Again, the features are selected based on a workload and the goal is to cluster tuples with similar features in the same data block. AQWA looks at adaptive data partitioning for spatial data (2 dimensions). Their techniques do not scale to higher dimensions [5]. Apart from single table partitioning, Hadoop++ [13] and CoHadoop [16] propose to co-partition datasets in HDFS to speed-up join queries. These systems still assume a workload.

Partitioning has also been considered in the context of indexing, e.g., partitioning a B^+-Tree on primary keys [19]. For multi-dimensional data, K-d Trees, R-Trees, and Quad-Trees have been proposed. These index structures are typically used for spatial data with 2 dimensions. Several other binary search trees have been proposed in the literature, such as splay tree [29]. Recent approaches layer multidimensional index structures over distributed data in large clusters. This includes SpatialHadoop [15], MD-HBase [25], and epiC [32], or adapting the multidimensional index to the workload in TrajStore [11]. However, all of these multidimensional indexing approaches typically focus on data locality and 2-dimensional spatial data, whereas our focus is on a balanced sub-division of multi-dimensional data.

Like Amoeba, cracking [20] is a technique to adapt the layout of data and indexes as queries arrive. Partial sideways cracking extends this idea to generate adaptive indexes on multiple columns [21]. Unlike Amoeba, cracking is a technique designed for in-memory column-stores and so is done as a part of *every* query in the system.

It does not naturally apply to a distributed setting for two main reasons. First, the cost of repartitioning in a distributed setting is higher than in a main memory system. So, we cannot afford to repartition data on every access as cracking does. Second, cracking splits the data on every new predicate it encounters, which can result in a large number of blocks. However, in a distributed setting, the number of data blocks that can be created is limited because blocks must be a certain size to amortize latencies of disk and network access. As a result, adding a split for a new predicate involves merging existing partitions and re-splitting them to keep the number of blocks constant.

8 CONCLUSION

In this paper, we presented Amoeba, a distributed storage system based on an *adaptive* and yet *robust* data partitioning scheme. Amoeba allows analysts to get started right away and reap the benefits of data partitioning without having to come up with a query workload. The key idea is to build and maintain a carefully crafted multi-dimension partitioning tree. We described an upfront partitioning algorithm to spread the benefits of partitioning over all attributes in a dataset. Subsequently, the partitioning adapts incrementally based on the predicates from the user queries. We showed a divide and conquer approach to transform the tree based on a cost model and described how Amoeba can be used with existing relational data processing frameworks like Spark SQL. Our results on both real and synthetic workloads show that Amoeba provides improved up-front query performance significantly, and, can match and even outperform workload-based range-partitioning schemes.

REFERENCES

[1] [n. d.]. Apache Hive. https://hive.apache.org. ([n. d.]).
[2] [n. d.]. Apache Spark. https://spark.apache.org. ([n. d.]).
[3] [n. d.]. Apache Zookeeper. https://zookeeper.apache.org/. ([n. d.]).
[4] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD*.
[5] Ahmed M Aly, Ahmed R Mahmood, Mohamed S Hassan, Walid G Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: adaptive query workload aware partitioning of big spatial data. *PVLDB* (2015).
[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2011. Disk-Locality in Datacenter Computing Considered Irrelevant.. In *HotOS*.
[7] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* (1975).
[8] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for Incremental Computations. In *SoCC*.
[9] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *PVLDB* (2016).
[10] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*.
[11] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. 2010. TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets. In *ICDE*.
[12] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB* (2010).
[13] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB* (2010).
[14] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. 2012. Only Aggressive Elephants are Fast Elephants. *PVLDB* (2012).
[15] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE*.
[16] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB* (2011).
[17] S. Ghandeharizadeh and D. J. DeWitt. 1994. MAGIC: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines. *IEEE Trans. Parallel Distrib. Syst.* (1994).
[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. *SIGOPS Oper. Syst. Rev.* (2003).
[19] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees. *CIDR*.
[20] Stratos Idreos, Martin Kersten, and Stefan Manegold. 2007. Database Cracking. In *CIDR*.
[21] Stratos Idreos, Martin Kersten, and Stefan Manegold. 2009. Self-organizing Tuple Reconstruction In Column-stores. In *SIGMOD*.
[22] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: adaptive partitioning for distributed joins. *PVLDB* (2017).
[23] mdc [n. d.]. IBM DB2 Multidimensional Clustering Tables, http://ibm.co/2971o1P.
[24] Rimma Nehme and Nicolas Bruno. 2011. Automated Partitioning Design in Parallel Database Systems. In *SIGMOD*.
[25] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *MDM*.
[26] Andrew Pavlo, Carlo Curino, and Stan Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD*.
[27] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *SIGMOD*.
[28] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *EDBT*.
[29] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting Binary Search Trees. *J. ACM* (1985).
[30] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2005. C-store: a column-oriented DBMS. *PVLDB* (2005).
[31] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *SIGMOD*.
[32] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. 2010. Indexing Multi-dimensional Data in a Cloud System. In *SIGMOD*.
[33] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *PVLDB* (2012).
[34] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. *PVLDB* (2004).