AdaptDB: Adaptive Partitioning for Distributed Joins

Yi Lu
MIT CSAIL
yilu@csail.mit.edu

Anil Shanbhag
MIT CSAIL
anils@mit.edu

Alekh Jindal
Microsoft
aljindal@microsoft.com

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

Big data analytics often involves complex join queries over two or more tables. Such join processing is expensive in a distributed setting both because large amounts of data must be read from disk, and because of data shuffling across the network. Many techniques based on data partitioning have been proposed to reduce the amount of data that must be accessed, often focusing on finding the best partitioning scheme for a particular workload, rather than adapting to changes in the workload over time.

In this paper, we present AdaptDB, an adaptive storage manager for analytical database workloads in a distributed setting. It works by partitioning datasets across a cluster and incrementally refining data partitioning as queries are run. AdaptDB introduces a novel *hyper-join* that avoids expensive data shuffling by identifying storage blocks of the joining tables that overlap on the join attribute, and only joining those blocks. Hyper-join performs well when each block in one table overlaps with few blocks in the other table, since that will minimize the number of blocks that have to be accessed. To minimize the number of overlapping blocks for common join queries, AdaptDB users *smooth repartitioning* to repartition small portions of the tables on join attributes as queries run. A prototype of AdaptDB running on top of Spark improves query performance by 2-3x on TPC-H as well as real-world dataset, versus a system that employs scans and shuffle-joins.

1. INTRODUCTION

Data partitioning is a well-known technique for improving the performance of database applications. By splitting data into partitions and only accessing those that are needed to answer a query, databases can avoid reading data that is not relevant to the query being executed, often significantly improving performance. Additionally, when partitions are spread across multiple machines, databases can effectively parallelize large scan operations across them. The traditional approach to partitioning has been to split each table on some key, using hashing or range partitioning. This helps queries that have selection predicates involving the key go faster, but does not affect the performance of queries without the key attribute. Likewise, for queries with joins, queries will benefit when the database is partitioned on attributes involved in the join, i.e., if the tables are *co-partitioned* on the join attribute. If one or both tables are not partitioned in this way, a *shuffle join*, where tables are dynamically repartitioned so that partitions that join with each other are on the same machine, is typically performed instead.

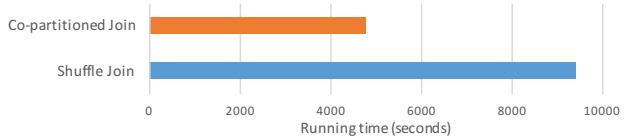


Figure 1: Shuffle vs co-partitioned joins

Partitioning can dramatically improve the performance of database applications, particularly when expensive shuffle-joins can be avoided. To illustrate, Figure 1 shows that co-partitioned joins can be almost 2 times faster than shuffle joins (here we are joining `lineitem` and `orders` tables from TPC-H, a popular decision support benchmark, at scale-factor 1000 in Spark [24] on 10 nodes). Because of these performance gains, many techniques have been proposed to find good data partitionings for a query workload. Such workload-based data partitioning techniques typically assume that the query workload is provided upfront or collected over time [2, 6, 15, 17, 19, 25, 26], and try to choose the best partitioning for that workload.

However, in many cases there may be no single static partitioning that is good for all workloads. For example, data science often involves looking for anomalies and trends in data. There is often no representative workload for this kind of ad-hoc, exploratory analysis, and the set of tables and predicates of interest will often shift over time. To illustrate this point, we obtained a workload trace from the analytics engine of a local startup company, which shows that even after seeing the first 80% of the workload, the remaining 20% of the workload still contains 57% new queries (i.e., only 43% of the queries are similar to previous ones). Besides not necessarily being representative of what analysts want to do, collecting a workload upfront inhibits data scientists from starting to explore the data; instead they must perform a tedious and time consuming data collection task before they can even ask their first query.

In previous work [21], we proposed a system called Amoeba that ameliorates some of these problems using a technique we call *hyper-partitioning*. Specifically, it first performs an initial partitioning of a dataset on as many attributes as possible, such that we create a number of small partitions, each of which contains a hypercube of the data, potentially from a different subset of attributes. This is done without any *a priori* workload. For example, partition 1 of the `lineitem` table from TPC-H might be partitioned first on product id, then on price, finally on quantity and partition 2 might be partitioned on price, then on order id. In this way, the system is

Proceedings of the VLDB Endowment, Vol. 10, No. 5

able to answer any query by reading a subset of partitions. Second, as more and more queries are observed, hyper-partitioning adaptively repartitions the data to gradually perform better on queries over frequent attributes and attribute ranges.

A key limitation of hyper-partitioning is that it does not adapt in response to join queries. Instead, tables are simply adapted based on the range predicates issued over them. Consider TPC-H, where `lineitem` can join with three dimension tables: `orders`, `part` and `supplier`. A dimension table can also join with another dimension table. For example, `orders` can join with `customer` on `custkey`. Nearly all data warehouse workloads involve similar joins between the fact table and multiple dimension tables. Because each table adapts differently in Amoeba, tables end up being partitioned on different attributes and ranges, such that joins often involve moving large portions of the table from one machine to another. As a result, shuffle-join is frequently the only sensible choice for a distributed join algorithm in the Amoeba system. Such shuffle-joins typically dominate query processing time in distributed databases, and so represent a missed opportunity for adaptive partitioning.

In this paper, we describe AdaptDB, which adaptively repartitions datasets to get good performance on frequent join queries. As two tables are joined over and over again, AdaptDB smoothly repartitions these two tables based on the join attribute using a technique we call two-phase partitioning. The key idea is that data is divided into a number of *partitioning trees*, with one tree per frequent join attribute per table. Blocks of data are incrementally moved from one tree to another as join frequencies vary. Each partitioning tree is split into two levels: in the top-most level data is partitioned according to the join attribute, and in the bottom levels it is partitioned according to frequent selection attributes (as in Amoeba). In AdaptDB, these blocks are spread across many different nodes in a cluster (we implemented AdaptDB on HDFS in Spark [24]).

As a result of this new partitioning approach, tables may end up partially partitioned on several different attributes, such that when two tables A and B are joined, a partition in A may join with several partitions in B, each located on HDFS. One option is to simply perform a shuffle join, repartitioning both A and B so that each partition of A joins with just one partition of B. However, this can be suboptimal if each partition of A only joins with a few partitions on B; instead, building a hash table over some partitions of A (or B) and probing it with partitions from B (or A) can result in significantly less network and disk I/O. Interestingly, building hash tables over different partitions of A or B can significantly affect the total cost, as we show in the next example:

EXAMPLE 1. *Suppose table A has 3 partitions and table B has 3 partitions. Suppose A_1 joins with B_1 and B_2, A_2 joins with B_1, B_2 and B_3, and A_3 joins with B_2 and B_3, and each machine $\mathcal{M}_i$ has memory to hold 2 partitions to build hash tables on A. Consider building a hash table over A_1 and A_3 on $\mathcal{M}_1$; we will need to read B_1, B_2 and B_3. We then build another hash table over A_2 on $\mathcal{M}_2$ and again read B_1, B_2 and B_3. In total, we read 6 blocks. As an alternative, building a hash table over A_1 and A_2 on $\mathcal{M}_1$ and another one over A_3 on $\mathcal{M}_2$ requires reading just $B_1, 2 * B_2, 2 * B_3 = 5$ blocks.*

Thus, building hash tables over different subsets of partitions will result in different costs. Unfortunately, as we show, finding the optimal collection of partitions to read is NP-Hard. Instead, we develop a new join algorithm called *hyper-join* that solves this problem heuristically, providing significant performance gains over shuffling in practice. To obtain these gains, partitions must be constructed such that, for a join between tables A and B, each partition of A only joins with a subset of the partitions of B. To do this, we develop several partitioning techniques that adapt partitioning trees to provide this property for commonly occurring joins.

In summary, we make the following major contributions:

- We introduce the *hyper-join* algorithm, which does not require shuffling datasets across the cluster. The challenge of hyper-join is to find optimal splits to minimize the total amount of disk I/O. We formulate this as an optimization problem based on mixed integer programming and give a proof of the hardness of the problem. An approximate algorithm is also proposed, which runs in much less time.
- We introduce several techniques to incrementally generate partitionings that are good for hyper-joins: *two-phase partitioning* and *smooth repartitioning*. AdaptDB's optimizer makes the decision to smoothly repartition part of the data with two-phase partitioning based on the queries in the query window.
- We describe an implementation of AdaptDB on top of HDFS and Spark and report a detailed evaluation of the AdaptDB storage manager on both synthetic and real workloads. We demonstrate that hyper-join can significantly reduce the cost of joins versus shuffle join, and that our incremental repartitioning techniques can yield partitionings that are good for hyper-join. Overall we show that hyper-join can be 2x faster than shuffle join on TPC-H and a real workload, and that it can effectively adapt as the mix of queries changes.

Before describing how these algorithms work, we present the architecture of the AdaptDB system.

2. SYSTEM ARCHITECTURE

AdaptDB is a table-oriented relational storage manager. It provides support for predicate-based data access and efficient data analytics based on joins. The primary goal of AdaptDB is to adapt to changes in the underlying workload using incremental repartitioning to ensure that join performance is good (i.e., does not require shuffle joins), and that sequential scans of entire tables can be avoided.

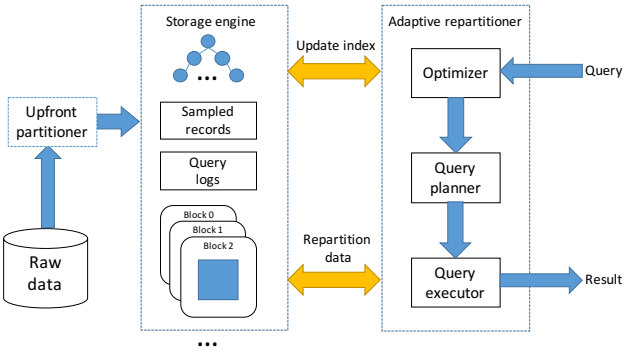


Figure 2: AdaptDB architecture

AdaptDB is a self-tuning storage system that can adapt itself to a specific workload and does not require manual effort or configuration from users. Figure 2 shows the key components of AdaptDB. The *upfront partitioner* creates an initial partitioning of the data using the method of Amoeba [21], described in the next section. This results in a collection of blocks (typically 64 MB or larger, as in HDFS), spread across the nodes of a distributed storage engine. Each of these blocks is partitioned on one or more attributes. This

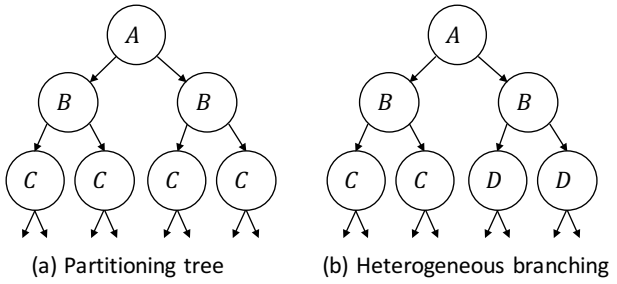


Figure 3: Upfront partitioning in Amoeba

up-front partitioning is done according to a *partitioning tree*, where the root splits the data in two partitions according to a randomly selected attribute, and each of these partitions is itself randomly partitioned (each on different attributes). This recursion repeats until blocks are less than the desired block size (Amoeba uses a sample of data to compute these splits and then loads data into blocks in a single pass). In addition to storing these blocks and samples, we store meta-data that tracks the split points for the data in the tree, and query logs for use in making repartitioning decisions.

They key contribution of AdaptDB is that, for join queries over multiple tables, we can execute joins using *hyper-join*. Hyper-join is preferred over a conventional shuffle join when the partitioning of the two tables is such that the number of blocks that would have to be moved to complete the join would be less than the number of blocks required in a shuffle join. This will be the case when the number of blocks in each table overlap with only a few blocks in the other table. Once it has chosen to use a hyper-join, AdaptDB employs an optimization algorithm to build hash tables optimally.

To provide good performance, AdaptDB repartitions blocks as queries are run. For selection predicates, it employs the adaptation method of Amoeba (also described in the next section). For join predicates, it employs a technique which we call *smooth repartitioning*, where it uses the first few levels of the tree to split on join attributes, and maintains multiple trees, one for each common join attribute on a particular table. As joins are executed, blocks are incrementally repartitioned from one partitioning tree to the other. AdaptDB uses a cost model and query log to make decisions about which blocks to move, and when.

3. BACKGROUND

In this section, we briefly describe the Amoeba storage system [21]. AdaptDB builds on top of Amoeba by adding support for adaptive joins. Amoeba exposes a 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 and return matching records. By partitioning data, Amoeba can often access a subset of blocks of data.

3.1 Upfront Data Partitioning

In Amoeba, a new block is created for every B bytes. Amoeba also considers the attributes of the dataset when creating blocks. As noted in the previous section, a dataset is split into data blocks on the underlying storage system using a partitioning-tree based on attributes. Amoeba recursively divides a dataset on different attributes, until the partition size is smaller than the block size of the storage system.

Amoeba represents the partitioning tree as a balanced binary tree i.e., it recursively partitions the dataset into two parts until it reaches the minimum partition size. Each node in the tree is denoted as A_p where A is the attribute being partitioned on and p is the cut-point. All records with attribute $A \leq p$ go to the left subtree and the rest of records go to the right subtree. The leaf nodes in the tree are data blocks, each with a unique identifier. An attribute can appear in multiple nodes in the tree. Having multiple occurrences of an attribute increases the number of ways the data is partitioned on that attribute.

Figure 3(a) shows such a partitioning tree. Here, it first partitions the dataset over attribute A, and then on attributes B and C recursively. In the end, 8 data blocks are created in total. As a result, queries with predicates on any attribute of A, B, and C can skip up to 50% of the data. The partitioning tree in Figure 3(a) can only accommodate as many attributes as the depth of the tree. Given a dataset with of size D and minimum block size p, the partitioning tree can only contain $\lfloor \log_n \frac{D}{P} \rfloor$ attributes when using n way partitioning. For example, for $n = 2$, $D =$ 1TB, and $P =$ 64MB, the tree can only accommodate 14 attributes. However, many real-world schemas have many attributes. To accommodate more attributes, Amoeba employs heterogeneous branching in the partitioning tree, i.e., it puts different attributes on the same level in the tree as shown in Figure 3(b). This sacrifices optimal performance on a few attributes to achieve improved performance over more attributes. In Figure 3(b), Amoeba is now able to accommodate 4 attributes, instead of 3. However, attributes C and D are each partitioned on 50% of the data. Heterogeneous branching is based on the premise that, in the absence of a workload, there is no reason to prefer one attribute over another.

Amoeba uses a top-down algorithm to initially assign different attributes to different nodes in the tree while trying to ensure the *average* number of ways each attribute is partitioned on is almost the same. The resulting balanced binary partitioning tree is used to partition the data into blocks which are then saved on HDFS. Real world datasets tend to be skewed or have correlation among attributes. In order to generate almost equally sized blocks, the system collects a sample from the data and uses it to choose the appropriate cut points.

3.2 Adaptive Re-partitioning

As users query the dataset, the goal of Amoeba is to adapt the partitioning based on the observed queries. Amoeba maintains a query window denoted by W. Each incoming query q is added into the query window. After each query, Amoeba looks at the current query window and considers alternative partitioning trees that could be generated by repartitioning on one or more blocks. These alternatives are generated by using a set of transformation rules on the current partitioning tree (i.e., merge two existing blocks partitioned on A and repartition them on B.) The system uses a bottom-up algorithm to compute the set of alternatives trees efficiently, using query predicates as hints to generate them. Among the set of alternative trees generated, it switches to the tree T that maximizes the total benefit using a simple cost formula based on the number of blocks read and an estimate of the repartitioning cost.

4. ADAPTIVE DISTRIBUTED JOINS

We now turn our attention to the hyper-join algorithm, which avoids expensive data shuffling whenever possible.

4.1 Hyper-join

Hyper-join is designed to move fewer blocks throughout the cluster than a complete shuffle join when tables are not co-partitioned. In the rest of this section, we first give the problem definition and formulate it as an optimization problem. We then introduce an optimal solution based on mixed integer programming. Finally, we present our approximate algorithm which can run in a much shorter time.

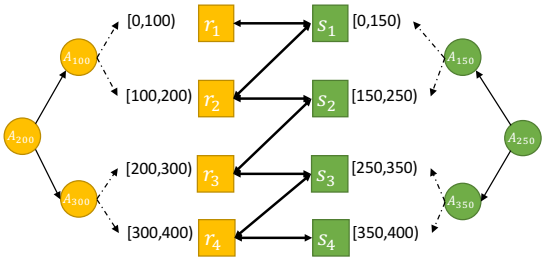


Figure 4: Illustrating hyper-join

4.1.1 Problem definition

Suppose we have two relations R and S, which can join on attribute t. Let $R = \{r_1, r_2, \ldots, r_n\}$ and $S = \{s_1, s_2, \ldots, s_m\}$ be the collection of data blocks obtained from AdaptDB.

$V = \{v_1, v_2, \ldots, v_n\}$ is a collection of m-dimensional vectors, where each vector corresponds to a data block in relation R. The j-th bit of v_i, denoted by v_{ij}, indicates whether block r_i from relation R overlaps with block s_j from relation S on attribute t (these are the blocks that must be joined with each other). Let $\text{Range}_t(x)$ be a function which gives the range (min and max values) of attribute t in data block x and $\mathbb{1}(s)$ be a function which gives 1 when statement s is true. Given two relations R and S, and for each block r_i from R and s_j from S, let $v_{ij} = \mathbb{1}(\text{Range}_t(r_i) \cap \text{Range}_t(s_j) \neq \emptyset)$. A straightforward algorithm to compute V has a time complexity of $O(nm)$. The Range_t values for each block are stored with each block in the partitioning tree.

Let $P = \{p_1, p_2, \ldots, p_k\}$ be a partitioning over R, where P is a set of disjoint subsets of the blocks of R and its union is all blocks in R. We constrain each p_i to be able to fit into memory of the node performing the join. We use $\tilde{v}(p_i)$ to denote the union vector of all vectors in p_i, i.e., $\tilde{v}(p_i) = \bigvee_{r_j \in p_i} v_j$, where v_j is the vector for block r_j. Let $\delta(v_i) = \sum_{k=1}^{m} v_{ik}$ indicating the number of bits set in v_i.

Given a partition p_i, we define the cost $C(p_i)$ of joining p_i with all partitions in S as the number of bits set in in $\tilde{v}(p_i)$, i.e., $C(p_i) = \delta(\tilde{v}(p_i))$. This corresponds to the number of blocks that will have to be read to join p_i in a hyper-join. Next, we define the cost function $C(P)$ over a partitioning, which is the sum of $C(p_i)$ over all p_i in P:

$$C(P) = \sum_{p_i \in P} C(p_i)$$

Thus, the problem of computing hyper-join is finding the optimal partitioning P of relation R.

Consider the example in Figure 4, with table $R = \{r_1, r_2, r_3, r_4\}$ and table $S = \{s_1, s_2, s_3, s_4\}$ and we assume $|P| = 2$, i.e., that we have sufficient memory to store $|R|/|P| = 4/2 = 2$ blocks of R in memory at a time. The interval on each partition indicates the minimum and maximum value on the join attribute from all the records. The arrows in the figure indicate the two corresponding partitions overlapping on the join attribute. As we can see from the figure, r_1 needs to join with s_1, r_2 needs to join with s_1, s_2, etc. Therefore, we have $V = \{v_1 = 1000, v_2 = 1100, v_3 = 0110, v_4 = 0011\}$. We can build a hash table over multiple yellow partitions to share some disk access of green partitions. For example, we can build a hash table over the first two yellow blocks (r_1 and r_2) and another one over the last two yellow blocks (r_3 and r_4), so that only 5 green blocks need to be read from disk, assuming only one green block is in memory at a time. In this way, the partition $P = \{p_1 = \{r_1, r_2\}, p_2 = \{r_3, r_4\}\}$, which is optimal. The overall cost $C(P) = 5$, since $\tilde{v}(p_1) = 2$ and $\tilde{v}(p_2) = 3$.

Intuitively, the objective function $C(P)$ is the total number of blocks read from relation S, with some blocks being read multiple times. From the perspective of a real system, we have to constrain the size of p_i, both due to memory limits and to ensure a minimum degree of parallelism (the number of partitions should be larger than a threshold). If memory is sufficient to hold B blocks from relation R, then we need $c = \lceil n/B \rceil$ partitions. We now define the *Minimal Partitioning*.

PROBLEM 1. *Given a set of data blocks from relation R, find a partitioning P over R such that $C(P)$ is minimized, i.e.,*

$$\begin{aligned} \underset{P}{\textit{arg min}} \quad & C(P) \\ \textit{subject to} \quad & |P| = c, \\ & |p_i| \leq B, \forall p_i \in P. \end{aligned}$$

4.1.2 Optimal algorithm

We now describe a mixed integer programming formulation which can generate the minimal partitioning. Since the algorithm takes a long time, it's not practical for real-world deployment. Instead, it provides a baseline with which to compare the faster approximation algorithm that we present in the subsequent section.

Given the maximum number of data blocks B that we can use to build a hash table due to available worker memory, we need to build $c = \lceil n/B \rceil$ hash tables in total. For each data block r_i from relation R and each partition p_k, we indicate the assignment of r_i to partition p_k with a binary decision variable $x_{i,k} \in \{0, 1\}$. Likewise, for each data block s_j from relation S, we create a binary decision variable $y_{j,k} \in \{0, 1\}$ to indicate if the j-th bit of $\tilde{v}(p_k)$ is 1.

The first constraint requires that the size of each partition p_k is under the memory budget B,

$$\forall k, \qquad \sum_{i=1}^{n} x_{i,k} \leq B$$

The second constraint requires that each data block r_i from relation R is assigned to exactly one partition, so for each r_i,

$$\forall i, \qquad \sum_{k=1}^{c} x_{i,k} = 1$$

Given a partitioning P, for each partition p_k, we need to guarantee every overlapping data block from relation S is also in partition p_k. Let J_k be the set of data blocks from relation R which overlaps with data block s_k from relation S.

$$\forall i, \forall k, \forall j \in J_k, \qquad y_{i,k} \geq x_{i,j}$$

We seek the minimal input size of relation S,

$$\min \sum_{j=1}^{m} \sum_{k=1}^{c} y_{j,k}$$

Solving integer linear programming (ILP) of this form is generally exponential in the number of decision variables; hence the running time of this algorithm may be prohibitive.

4.1.3 Approximate partitioning

We now consider a heuristic algorithm to partition R into partitions of size B. The algorithm is given in Figure 5.

The algorithm starts from an empty set of partitions P. It iteratively generates a partition $\mathcal{P}$ by taking at most B data blocks from relation R with smallest $\delta(\tilde{v}(\mathcal{P}))$ and adds $\mathcal{P}$ into P until P contains all blocks from relation R.

$R \leftarrow \{r_1, r_2, \ldots, r_n\}$, $P \leftarrow \emptyset$
while R is not empty:
  generate $\mathcal{P}$ from $\min(B, |R|)$ blocks with smallest $\delta(\tilde{v}(\mathcal{P}))$
  remove all blocks in $\mathcal{P}$ from R and add $\mathcal{P}$ to P
return P

Figure 5: An approximate partitioning algorithm

4.1.4 NP-Hardness

We now prove that problem of taking B data blocks from relation R with smallest $\delta(\tilde{v}(\mathcal{P}))$ is NP-hard by reduction from maximum k-subset intersection [5].

Given a maximum k-Subset intersection instance $I = (C, E)$, where $C = \{c_1, c_2, \ldots, c_n\}$ subsets over a finite set of elements $E = \{e_1, e_2, \ldots, e_m\}$, and a positive integer k. The maximum k-Subset intersection problem finds exactly k subsets $c_{j_1}, c_{j_2}, \ldots, c_{j_k}$ from C whose intersection size $|c_{j_1} \cap c_{j_2} \cap \cdots \cap c_{j_k}|$ is maximum.

We construct an input to maximum k-Subset intersection problem from an instance to our problem as follows. For each bit vector v_i, we construct a flipped bit vector $\bar{v}_i$, i.e., $\forall k, \bar{v}_{i_k} = 1 - v_{i_k}$. Let $C = \{\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_n\}$ be a set which consists of all flipped bit vectors from relation R. Each flipped vector $\bar{v}_i$ from C denotes which blocks from relation S that $\bar{v}_i$ does *not* overlap with. Therefore, we let $E = \{s_1, s_2, \ldots, s_m\}$ consist of all blocks from relation S and $k = B$. Since $\bigwedge \bar{v}_i = \overline{\bigvee v_i}$, minimizing $|\bar{v}_{j_1} \wedge \bar{v}_{j_2} \vee \cdots \vee \bar{v}_{j_k}|$ is equivalent to maximizing $|v_{j_1} \vee v_{j_2} \vee \cdots \vee v_{j_k}|$. Therefore, an optimal solution to our problem solves the maximum k-Subset intersection problem on I.

4.1.5 A bottom-up solution

Since taking B data blocks from relation R with smallest $\delta(\tilde{v}(\mathcal{P}))$ is NP-hard and there is no algorithm for $n^{1-\epsilon}$-approximation for any constant $\epsilon > 0$, we developed a simple bottom up algorithm with practical runtimes for use in AdaptDB. It is shown in Figure 6.

$R \leftarrow \{r_1, r_2, \ldots, r_n\}$, $P \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$
while R is not empty:
  merge $\mathcal{P}$ with data block r_i with smallest $\delta(r_i \vee \tilde{v}(\mathcal{P}))$
  if $|\mathcal{P}| = B$ or r_i is the last one in R:
    add $\mathcal{P}$ to P and $\mathcal{P} \leftarrow \emptyset$
  remove data block r_i from R
return P

Figure 6: A bottom-up solution

The algorithm starts from an empty set of partitions P and an empty partition $\mathcal{P}$. It iteratively adds a data block r_i into $\mathcal{P}$ with smallest $\delta(r_i \vee \tilde{v}(\mathcal{P}))$ until we have B blocks in partition $\mathcal{P}$ or no data block left in relation R. It then adds $\mathcal{P}$ into P until P contains all blocks from relation R. A straightforward implementation of this algorithm has a time complexity of $O(n^2)$ (where n is the number of blocks of R), since we have to compute the minimum cost block (requiring a scan of the non-placed blocks) n times.

4.2 Analysis of Shuffle Join and Hyper-join

We consider a theoretical model here which analyzes the cost of shuffle join and hyper-join in a distributed database. The model focuses on the number of blocks read (I/O cost), as the time to process a join is directly proportional to the number of blocks accessed. Each block incurs approximately the same amount of disk I/O, network access, and CPU (hashing/joining) costs.

One concern might be that local I/O is cheaper than remote (network I/O). However, recent improvements in datacenter network design have resulted in designs that provide full cross-section bandwidth of 1 Gbit/sec or more between all pairs of nodes [4], such that network throughput is no longer a bottleneck. Recent research has shown that accessing a remote disk in a computing cluster is only about 8% lower throughput than reading from a local disk [3].

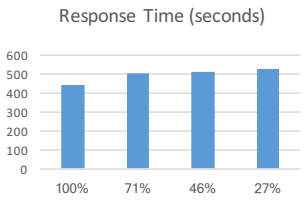


Figure 7: Varying data locality

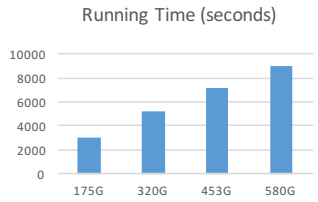


Figure 8: Varying dataset size

To verify this in a real system, we ran a micro-benchmark on Hadoop, in which we measured the runtime of a map-only job (performing simple aggregation in the combiner) while varying the locality of data blocks on HDFS. Figure 7 shows the results from a 4-node cluster with full duplex 1 Gbit/sec network. Note that even with locality as low as 27%, the job is just 18% slower than with 100% data locality. We leverage these new hardware trends and the fact that the cost of remote disk access is essentially the same as local disk access in our cost analysis below.

We now analyze the cost of shuffle join and hyper-join. Suppose we have a query q over two relations R and S.

Shuffle Join. There are two phases in shuffle join. In the first phase, map tasks are created to read data blocks from HDFS. For each record in the dataset, it uses a partitioning function to find a corresponding partition it belongs to and write it to a file on local disk. In the second phase, each machine is responsible for some key space of the partitioning function and reads partitions either locally or from remote machines before joining two tables. In summary, each record is read from disk, partitioned by a partition function and written to disk, and read from disk again to compute join result. We use *Cost-SJ* to denote the cost of shuffle join over two tables.

$$\text{Cost-SJ}(q) = \sum_{b \in \text{lookup}(T_R, q)} C_{\text{SJ}} \cdot |b| + \sum_{b \in \text{lookup}(T_S, q)} C_{\text{SJ}} \cdot |b| \quad (1)$$

Where T_R and T_S are the partitioning trees for relation R and S, and the function $\text{lookup}(T, q)$ gives the set of relevant data blocks for query q in T. The value of C_{SJ} is obtained empirically by modeling the disk access and the cost of data shuffling, which is set to 3 in our evaluation. To verify Cost-SJ(q) is linear with the number of data blocks read from disk, we ran a micro-benchmark in Spark, in which we measured the runtime of joining table `lineitem` and `orders` from TPC-H while varying the size of tables. Figure 8 shows the results from 175G to 580G; note that the running time increases linearly with the size of dataset.

Hyper-join. In hyper-join, data blocks from one table are read through HDFS to build a hash table. It then probes the hash table with all overlapping data blocks from the other table. Without loss of generality, a hash table is built on relation R in the analysis. We use *Cost-HyJ* to denote the cost of hyper-join over two tables and hash tables are built on table R.

$$\text{Cost-HyJ}(q) = \sum_{b \in \text{lookup}(T_R, q)} |b| + \sum_{b \in \text{lookup}(T_S, q)} C_{\text{HyJ}} \cdot |b| \quad (2)$$

Here C_{HyJ} is a measure of the number of times (on average) a data block from relation S needs to be read from disk – this depends on the quality of the partitioning in the two tables. For a completely co-partitioned table, C_{HyJ} will be 1, as each block in relation R joins with exactly one block in relation S. In Section 7.4, we show that our algorithms can achieve an C_{HyJ} of around 2 on real query workloads using a memory size of 4GB on a 1 TB dataset.

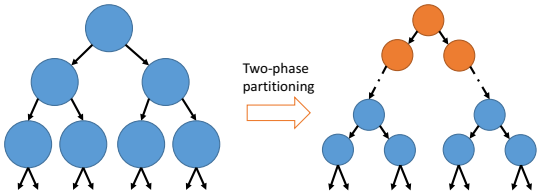


Figure 9: Illustrating two-phase partitioning

Depending on the values of Cost-SJ(q) and Cost-HyJ(q), hyper-join can be substantially more efficient in terms of the communication cost than shuffle join. However, in order for this to be true, data sets must be partitioned on the join attribute. It's difficult to achieve this by adaptively repartitioning a tree based solely on selection predicates, as it's unlikely to have the join attribute in very many nodes in the tree, and it's highly possible that every partition will overlap with a large number of partitions. To tackle this challenge, AdaptDB employs a technique we call *smooth repartitioning with two-phase partitioning* to push the join attribute into the partitioning tree. We describe this in the next section.

4.3 Joins Over Multiple Relations

We have so far restricted our discussion on hyper-join to two relations but our techniques extend to multiple inputs as well. Consider TPC-H query 3. If the join order is (`lineitem` ⋈ `orders`) ⋈ `customer` and the intermediate result of the first two tables is denoted by `tempLO`, then the relation `customer` needs to join with `tempLO` on `custkey`. If `custkey` is the join attribute in the `customer` partitioning tree, AdaptDB only needs to shuffle `tempLO` based on `custkey`, and can then use hyper-join instead of an expensive shuffle join, in which both `tempLO` and `customer` need to be shuffled.

When there are more relations to join, shuffle join over two intermediate relations from hyper-join could be more efficient. Consider TPC-H query 8. If the join order is ((`lineitem` ⋈ `part`) ⋈ `orders`) ⋈ `customer`, then the intermediate result with relation `lineitem` needs to be shuffled twice. Instead, we can change the join order to (`lineitem` ⋈ `part`) ⋈ (`orders` ⋈ `customer`) and use hyper-join twice and a shuffle join over the intermediate results.

5. PARTITIONING FOR HYPER-JOIN

In this section, we describe how we build and maintain partitioning trees to support the hyper-join algorithm running in AdaptDB. Specifically, we introduce the idea of *two-phase partitioning*, where partitioning trees have join predicates injected at their root, and smooth repartitioning where we maintain multiple partitioning trees and migrate blocks between them. AdaptDB's optimizer automatically applies these techniques as appropriate to achieve better performance without the need for manual tuning.

5.1 Two-phase Partitioning

A key limitation of the adaptive repartitioning technique used in Amoeba is that it does not adapt in response to join queries. Instead, each table adapts independently and tables end up being partitioned on different attributes and ranges, such that hyper-join would not provide a performance advantage over shuffle joins. Hence a key goal is to adapt partitioning trees in a way that facilitates joins while maintaining the performance advantages of partitioning for selection queries.

Each AdaptDB tree is designed to support a single join attribute. We choose to build a new partitioning tree when a new popular join attribute is seen (this is described in more detail in the next section), or if requested to do so by the user (because he or she believes a particular join will be common).

AdaptDB uses two-phase partitioning to inject the join attribute into the tree, as depicted in Figure 9. In the first phase, splitting is done join attributes, and in the second it is done on selection attributes.

In Figure 9, the join partitions are depicted in orange, and partitioning nodes in support of selections are shown in blue. Median values of the join attribute are used to split the dataset into two subsets during the first phase. This median-based partitioning is further applied in lower-levels of the tree, splitting each partition on its median (we do this efficiently by sorting all values of the attribute in the sample at the root, and recursively computing medians for each subtree over this sorted list). During the second phase, the join partitions are further partitioned using the adaptive partitioning technique of Amoeba. There is a trade-off between the number of levels reserved for the join attribute and the number of levels reserved for selection attributes. We evaluate the number of levels that should be reserved for the join attribute in our evaluation.

Consider the left partitioning tree in Figure 4 as an example. There are two levels in the tree which are reserved for the join attribute, which, assuming data is uniformly distributed in the range $[0, 400]$, leads to four disjoint partitions with range $[0, 100)$, $[100, 200)$, $[200, 300)$, and $[300, 400)$. The same procedure is also applied to the right partitioning tree, which creates four disjoint partitions with range $[0, 150)$, $[150, 250)$, $[250, 350)$, and $[350, 400)$.

As an alternative to partitioning the top levels of the tree on the median value, we could have used hashing or range-based partitioning. The disadvantage of hash-based co-partitioning is that it cannot answer queries with range queries on the join attribute. Such joins may occur, for example, in social network applications, where we may wish to join on some geographic or temporal range, e.g., users who have been in a lat/lon region within some time frame. Likewise, data skew is prevalent in social networks, and using simple range-based co-partitioning can lead to imbalanced data blocks; medians help avoid this skew.

5.2 Smooth Repartitioning

A partitioning tree created through two-phase partitioning is only optimized for a single join attribute. However, a table with multiple foreign keys may join with multiple tables. For example, in TPC-H, queries join `lineitem` and `orders` on order_key and `lineitem` and `supplier` join on supplier_key. We observe multiple instances of this kind of multi-join pattern in the real workloads we use to evaluate AdaptDB.

Our goal is that, when AdaptDB observes new incoming queries containing a new join attribute, it should shift to the new join attribute. However, repartitioning all of the data immediately would introduce a potentially very long delay, and, when the workload is periodic, could lead to oscillatory behavior where it switches from one partitioning to another. In addition, during the shift to the new join attribute, AdaptDB should provide good performance for both types of queries instead of always using shuffle join.

To tackle these challenges, AdaptDB introduces smooth repartitioning, which smoothly adapts to the new join partitioning attribute, providing reasonably good performance for both types of queries during the transition. For the sake of simplicity, we restrict our discussion to shifting from one join attribute to another attribute in this section, but our techniques generalize naturally to multiple join trees.

AdaptDB keeps all queries in a recent query window. When AdaptDB observes a query with a new join attribute, it creates a

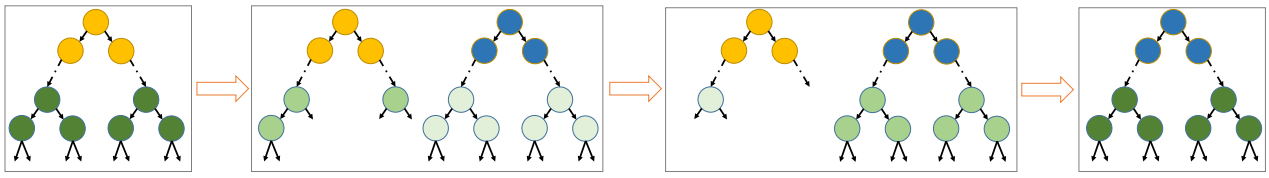


Figure 10: Illustrating smooth repartitioning

$W \leftarrow$ Query window, $q \leftarrow$ New incoming query
$T \leftarrow$ Old partitioning tree, $T' \leftarrow$ New partitioning tree
$W \leftarrow W \cup \{q\}$
if q's join attribute t is as the same as T''s:
  $n \leftarrow |\{q | q \in W \wedge q\text{'s join attribute} = t\}|$
  $p \leftarrow \frac{n}{|W|} - \frac{|T|}{|T|+|T'|}$
  if $p > 0$, repartition p percent of the data from T to T'

Figure 11: The smooth repartitioning algorithm

new partitioning (initially empty) tree. The new tree's join attribute comes from the new query and the its predicates are used to build the lower levels of the tree. AdaptDB also repartitions $1/|W|$ of the dataset from the old tree to the new tree, where $|W|$ is the length of the query window. This is accomplished by randomly choosing $1/|W|$ of the blocks in the old tree, and inserting them into the new tree (because files are only appended in HDFS, it is possible to do this without affecting the correctness of any concurrent queries). To avoid doing repartitioning work when rare queries arrive, AdaptDB can be configured to wait to create a new partitioning tree until the query window contains some minimum frequency f_{min} of queries for a new join attribute; in this case once the tree is created, $f_{min}/|W|$ of the blocks will be moved.

As AdaptDB starts seeing more and more queries with the new join attribute, it repartitions more data into the new partitioning tree using the following algorithm. It first calculates the percentage of two types of query in the query window and how much data each partitioning tree has. If the incoming query's join attribute is the same as the newly created partitioning tree and the fraction of data in the new partitioning tree is less than the fraction of its type in the query window, AdaptDB moves data from the old partitioning tree to the new one, again by randomly selecting blocks and moving them. Pseudo-code of the algorithm is shown in Figure 11, where $|T|$ denotes the size of data under the partitioning tree T.

Consider the example in Figure 10. The algorithm starts from a partitioning tree optimized for join attribute A. When a query with new join attribute B comes into AdaptDB, AdaptDB creates a new partitioning tree for B with two-phase partitioning and repartitions $1/|W|$ of the dataset from the old partitioning tree. The color of nodes from the lower levels of the partitioning trees indicate the size of data. The darker the color is, the larger the size of data is. After the new tree is created, AdaptDB maintains two partitioning trees with different join attributes during smooth repartitioning. As more and more queries with join attribute B appear in the query window, AdaptDB repartitions more data from the old partitioning tree to the new one. AdaptDB iterates the above procedures until the query window only includes queries with join attribute B. After the dataset finishes repartitioning, the old partitioning tree for join attribute A is removed and AdaptDB only maintains the partitioning tree for join attribute B, which is depicted by the last sub-figure in Figure 10. (Of course, in many applications there will not be a complete shift from one join to another, in which case multiple trees will be preserved.)

5.3 Key Benefits

The key benefits of two-phase partitioning and smooth repartitioning are as follows:

Avoiding shuffle join. It's not uncommon for multiple tables to be involved in data analytics. 18 out of 22 queries from TPC-H need to join multiple tables, such as `lineitem`, `orders`, `customer` and `supplier`. However, it's prohibitively expensive to shuffle a dataset across a cluster of machines, especially, when there is no selective predicate on a large dataset. For example, consider TPC-H query 3, the selectivity of predicate *l_shipdate > date '[DATE]'* is from 75% to 90%. As another example, TPC-H queries 5 and 8 do not have any predicate on `lineitem` at all. We would like to fine-grained partition a dataset based on the join attribute. After partitioning, each partition only needs to join with a few partitions from the other dataset, so hyper-join will be very effective. Without using partitioning on the join attribute, it may not be possible to get any performance benefit from partitioning. For example, in the case of TPC-H queries 5 and 8, no matter how many queries arrive, it's still not possible to reduce the number of records to read by simply partitioning with selection predicates. In contrast, partitioning on join attributes and employing hyper-join can provide a significant speedup.

Smooth shift to other join attributes. When queries with a new join attribute arrive in AdaptDB, it shifts the partitioning of the dataset from the old join attribute to the new one. Meanwhile, it can use a combination of hyper-join and shuffle to execute a query. Consider the case of a mix of TPC-H queries 12 and 14. Query 12 joins `lineitem` and `orders` on `order_key`, and query 14 joins `lineitem` and `part` on `part_key`. As more and more queries from query 14 arrive, the partitioning will shift from `order_key` to `part_key` smoothly. Even when maintaining two partitioning trees for `lineitem` (one on `order_key` and the one on `part_key`), AdaptDB can use a combination of shuffle join and hyper-join to execute queries from both queries, performing much better than using full shuffle joins.

5.4 Putting it All Together

AdaptDB manages partitioning trees using the smooth-repartitioning approach, continuously migrating blocks as join queries arrive to ensure a proper balancing of data across partitioning trees. To execute joins, AdaptDB uses a simple cost model based on equations 1 and 2. First, it estimates C_{HyJ} for the two tables being joined. It does this by using the hyper-join algorithm (Section 4.1.5) to compute the schedule of blocks to read, and counts the total number of block reads that would result if the schedule were run. Then, using the two equations, it decides whether to actually run hyper-join or shuffle join based on the sizes of the two tables and the estimated C_{HyJ} value.

6. IMPLEMENTATION

In this section, we describe AdaptDB's implementation[1]. AdaptDB runs on HDFS and Spark [24]. We choose HDFS as it's a popular open source distributed file system, but our ideas can be implemented on any other distributed file system. Likewise, AdaptDB's query executor can also be built on any other data-parallel computing system, e.g., Hadoop [1] or Husky [23].

The AdaptDB storage manager consists of two modules: (1) the upfront partitioner, which generates an upfront partitioning from raw data and writes initial partitions. (2) the adaptive repartitioner, which adaptively repartitions the underlying data according to query workload. When a query is submitted to AdaptDB, it first goes to the optimizer, then to the query planner and finally to the query executor for query execution.

Optimizer. This component is responsible for adjusting the partitioning tree(s) for each table in AdaptDB. It decides how much data should be repartitioned. If there are multiple partitioning trees for a table, some blocks may be repartitioned based on the percentages of each type of query in the query window. If there is only one partitioning tree but the optimizer decides it can be refined, some blocks also need to be repartitioned.

The optimizer yields two disjoint sets of data blocks: (1) Type 1 blocks that will only be scanned, and (2) Type 2 blocks that will be scanned and repartitioned to generate new data blocks using the new partitioning tree. Either one of these sets above may be empty.

Query Planner. This component decides how to join two tables in AdaptDB. There are three cases: (1) both tables have only one partitioning tree and each table is partitioned on the join attribute; in this case, hyper-join can be used instead of shuffle join; (2) one table has one partitioning tree on the join attribute and the other table has multiple partitioning trees, which could happen during smooth repartitioning – in this case, AdaptDB uses a combination of hyper-join and shuffle join for query execution; (3) both tables have multiple partitioning trees, or none of the partitioning trees are on the join attribute – in this case, AdaptDB will generally fall back to shuffle join for query execution (although its possible hyper-join could still be beneficial if the up-front partitioning happens to work out.)

Query Executor. AdaptDB executes queries in Spark. A Spark job is constructed from the two sets of data blocks returned by the optimizer. We create file splits from these data blocks. The size of each file split is less than a user-supplied threshold; we use a 4GB split size in our experiments.

A Spark task is created on each file split; the task reads the data blocks from HDFS in bulk and iterates over the records in memory. Tasks created for Type 1 blocks run with a scan iterator which simply reads all records and filter out ones that cannot pass the predicates in the query. Tasks created for Type 2 blocks run with a repartitioning iterator. Besides reading and filtering records as in the scan iterator, the repartitioning iterator also looks up each record in the new partitioning tree to find its new partition id and repartitions the record accordingly. The repartitioning iterator maintains a buffered writer. Once a buffer is full, the repartitioner flushes the records in the buffer into HDFS. Several repartitioners across the cluster may write to the same file. As a result, repartitioners need to coordinate while flushing the new partitions. We use ZooKeeper for distributed coordination.

Tasks are scheduled by the Spark scheduler and executed across the cluster in parallel. The result exposed to users is a Spark RDD. Users can conduct more complex analysis on top of the returned RDDs using the standard Spark APIs, e.g., run an aggregation.

[1]The source code is available at: `https://github.com/mitdbg/AdaptDB`

7. EVALUATION

In this section, we analyze the performance of AdaptDB focusing on the following key questions:

- How much performance gain can AdaptDB's hyper-join algorithm achieve over a traditional shuffle join algorithm?
- Does AdaptDB eventually converge when a particular workload is seen more often?
- How sensitive is AdaptDB to different parameters?
- Is AdaptDB's ILP formulation for choosing which blocks to join necessary for achieving good performance under a space constraint? How does the heuristic block selection algorithm perform?
- What is AdaptDB's performance on real workloads, in addition to TPC-H?

7.1 Experimental Setup

We ran our experiments on a cluster of 10 machines, each with 256 GB RAM and four 2.13 GHz Intel(R) Xeon(R) E7-4830 CPUs running 64-bit Ubuntu 12.04 with Linux kernel 3.2.0-23. The AdaptDB storage system runs on top of Hadoop 2.6.0 and uses ZooKeeper 3.4.6 for synchronization. We ran queries in Spark 1.6.0 with Java 7. All experiments were conducted with cold caches.

TPC-H . The TPC-H benchmark is a decision support benchmark. We ran the benchmark with scale factor 1000 (1TB) on AdaptDB. There are 22 query templates in TPC-H. We chose eight query templates (q_3, q_5, q_6, q_8, q_{10}, q_{12}, q_{14}, q_{19}) from the TPC-H workload. The reason that the other 14 query templates were not chosen is twofold. First, five of the query templates (q_2, q_{11}, q_{13}, q_{16} and q_{22}) do not involve the `lineitem` table, which is the largest table in TPC-H. Second, nine query templates (q_1, q_4, q_7, q_9, q_{15}, q_{17}, q_{18}, q_{20} and q_{21}) do not have selective filters, so will not benefit from any partitioning technique. This choice of workload is common to other papers that evaluate partitioning techniques in distributed databases [22].

CMT. The CMT data consists of anonymized logs of user trips obtained from a MA-based startup that specializes in processing mobile sensor data for telematics. The data consists of a single large fact table with 115 columns and several dimension tables with 33 columns in total. Each entry in the dataset has attributes of a trip from users, such as user ID, average velocity, trip start time and end time. Due to privacy concerns, we generated a synthetic version of the data according to the statistics collected by the company. The total size of the data is 205GB. A production query trace collected from 04/19/2015 to 04/21/2015 was also obtained from the company (103 queries were issued by data scientists when performing exploratory analysis on the data). Each query sub-selects different portions of the data based on different predicates, e.g., user ID and trip time range.

Unless otherwise stated, we set the length of AdaptDB's query window to 10, the split size of Spark [24] to 4GB, and used half of the levels of the partitioning tree for join attributes.

7.2 Effect of Hyper-join Algorithm

We first look at how much benefit we can get from hyper-join over shuffle join. For this experiment, we ran seven sets of queries (q_3, q_5, q_8, q_{10}, q_{12}, q_{14}, q_{19}) from TPC-H using both shuffle join and hyper-join and show the results in Figure 12. We do not report the performance of TPC-H query 6, since there is no join involved in it. For each query template, we ran the smooth partitioning algorithm for several iterations until just one tree with the join attribute

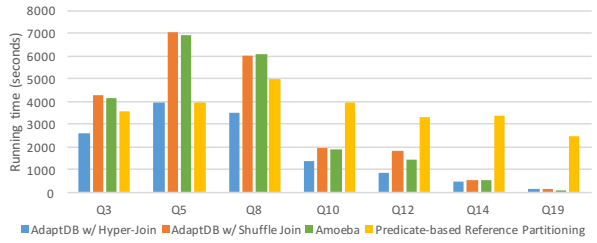


Figure 12: Execution time for queries on TPC-H

existed for the target query. We then report the average runtime of 10 runs of each query, using both hyper-join and shuffle join. Both types of joins benefit from the Amoeba adaptive parititoning algorithm in this case.

From Figure 12, we observe that hyper-join is more efficient than shuffle join whether or not there are selective predicates in a query. For example, there are selective predicates on `shipinstruct`, `quantity` and `shipmode` in TPC-H query 19. In this case, AdaptDB has a 33% performance gain. In cases with no selective predicates, e.g. TPC-H query 5, AdaptDB has a 76% performance gain due to the more efficient hyper-join algorithm. Overall, we can observe that the hyper-join is consistently faster than shuffle join in all seven query templates. AdaptDB achieves a 1.60x performance gain on average over shuffle join (maximum 2.16x).

To better understand the significance of our adaptive join techniques, we compared AdaptDB with Amoeba [21]. Amoeba does not include join attributes in the partitioning tree and uses shuffle joins. From Figure 12, we observe that AdaptDB with hyper-join is always much faster than Amoeba. AdaptDB with shuffle join has almost the same performance as Amoeba in q_3, q_5, q_8, q_{10} and q_{14}. Amoeba has better performance than AdaptDB with shuffle join in q_{12} and q_{19}, since these two queries have more selective predicates and fewer nodes in AdaptDB are used for selection predicates. In effect, AdaptDB trades some of the levels in a partitioning tree for more efficient hyper-join algorithm, which, at least for these TPC-H queries, is a good tradeoff.

We also compared against a static data partitioning technique, specifically predicate-based reference partitioning (or PREF [25] for short). To provide a fair comparison, we partitioned the TPC-H dataset by the PREF partitioner provided by its authors and ran queries using the Spark-based executor as AdaptDB. We tried different numbers of partitions in PREF. Fewer partitions result in less data redundancy but also has lead to lower parallelism in query execution. We report the performance of PREF with 200 partitions across 10 machines (we tried a number of different settings for the number of partitions and found 200 to be optimal). Figure 12 shows AdaptDB with hyper-join always outperforms PREF, often significantly. This is due to the fact that, in order to avoid shuffle joins, PREF replicates data, which often results in significantly more I/O than AdaptDB. We do see that, compared to AdaptDB's with *shuffle join*, PREF outperforms AdaptDB in q_3, q_5 and q_8, because these queries do not have selective predicates. In q_{10}, q_{12}, q_{14} and q_{19}, AdaptDB is much faster than PREF no matter what join algorithm AdaptDB uses since the predicates here are selective.

In summary, these experiments show that when partitioning is good, hyper-join performs well, usually better than competing techniques. Next we show that the adaptive partitioning can generate good partitionings over time.

7.3 Performance of Adaptive Repartitioner

We now study how AdaptDB adaptively repartitions the dataset over different workload patterns on TPC-H. Initially, each table is randomly partitioned by the upfront partitioner. We constructed queries with different predicate values from each query template and ran query templates in the order: q_3, q_5, q_6, q_8, q_{10}, q_{12}, q_{14}, q_{19}.

We consider two types of workloads:

- *switching* workload: We run 20 queries for each query template and switch from one query template to another immediately. For example, we start from q_3 and switch to q_5 after 20 queries. We next switch to q_6 again after 40 queries again. In total, there are 160 queries in this workload.
- *shifting* workload: We gradually shift from one query template to another one. For example, we start from query template q_3. As more queries are run, the workload shifts to query template q_5 smoothly and the transition finishes in 20 queries. Specifically, the probability of running query q_5 (q_3) is increased (decreased) by 1/20th after each query. We next shift the workload from query template q_5 to query template q_6, and so on. In total, there are 140 queries in this workload.

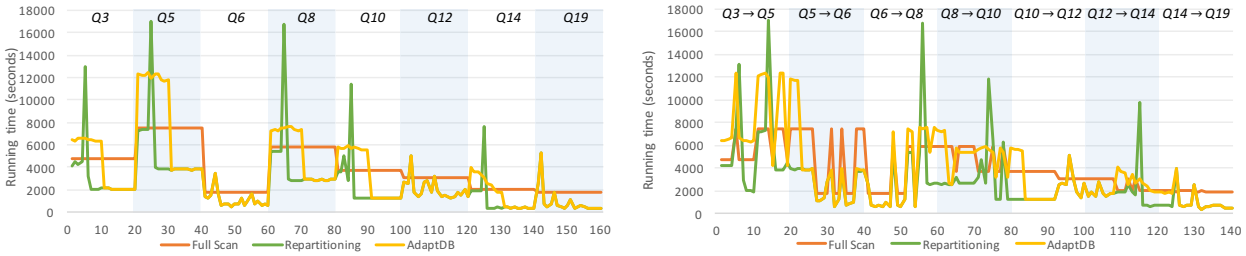
We compare AdaptDB with two different baselines: (1) Full Scan, where no partitioning tree is used, and full scans and shuffle joins are run and (2) Repartitioning, where smooth repartitioning is disabled, and AdaptDB does a complete repartitioning of the data when half of the queries in the query window have a new join attribute. hyper-join is used in this baseline whenever possible (e.g., after repartitioning).

Figure 13(a) shows the *switching* workload. Here we see the benefit of AdaptDB over both repartitioning and full scan. Repartitioning incurs very long partitioning times in queries 5, 25, and 65, whereas AdaptDB spreads out the repartitioning over a longer period during which performance is only moderately degraded. In both cases, after repartitioning completes, the repartitioned system is much faster than full scans with shuffle joins.

Note that the aggregate benefit of repartitioning is dependent on the amount of time each query is active – if a query is run more than 10 times, the benefit will be larger. Also, note that the rate of adaptation of AdaptDB is dependent on the window size; faster and more disruptive partitioning can be achieved by using a smaller window, and slower and less disruptive partitioning can be achieved by using a larger window.

Figure 13(b) shows the *shifting* workload, where a mix of queries are active at each point. The overall performance trends are harder to see because at each point one of two active queries is chosen, and the performance of one query can be quite different from the other (most of the spikes in the AdaptDB and Full Scan line are due to this effect, not the overhead of repartitioning). Here we can see that repartitioning approach performs major reorganizations at queries 6, 14, 56 and 72. As in the switching workloads, AdaptDB takes longer to adapt but has much less pronounced spikes. Again, the overall benefit of partitioning depends on the time each query template is active for, but generally reaches a 2x or greater improvement over full scan with shuffle join. In addition, as in the switching workload, the rate of adaptation of AdaptDB depends on the window size; we could further dampen the overhead of repartitioning by using a bigger window.

We observed a performance degradation on query 5, queries 11-14, queries 17-18 and queries 20-22 when AdaptDB repartitions the data. The same phenomenon also happens from query 21 to query 30 in the switching workload. We found the performance of Spark degrades when writing large amounts of data into HDFS as there is no predicate the 5th query template from TPC-H. We think this is a problem with the Spark executor; we have reported to the issue and hope it will be fixed in a future Spark release.



(a) Switching workload

(b) Shifting workload

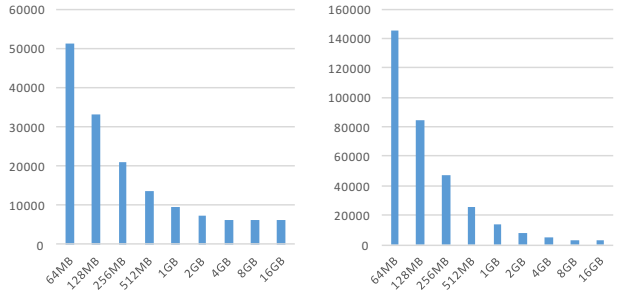
Figure 13: Execution time for changing workload on TPC-H

Overall, we can observe as AdaptDB runs more queries of a particular type, the query runtime approaches the ideal runtime. This shows AdaptDB has the ability to adapt to changes in the workload.

7.4 Parameter Sensitivity

We now study how sensitive AdaptDB is to the size of the memory buffer for hyper-join, the size of query window as well as the number of levels in the partitioning tree for the join attribute.

Effect of varying size of memory buffer. In order to quantify the effect of memory buffer size, we joined the `lineitem` and `orders` tables without selection predicates. As we introduced in Section 4.1, more disk accesses are shared and less data is read when we have more memory for buffers.



(a) Running time (seconds)

(b) # of data blocks

Figure 14: Effect of varying size of memory buffer

Here, both tables are partitioned using two-phase partitioning. In other words, `l_orderkey` and `o_orderkey` appear in the upper level of the trees. We build hash tables over the `lineitem` table and probe them with the `orders` table. We vary the size of the memory buffer from 64 MB to 16 GB and report the query's running time and the number of data blocks read from the `orders` table.

Figure 14(a) shows that better performance is obtained as we increase the size of the memory buffer up to 4 GB. Increasing the buffer size beyond 4 GB does not help since the amount of data read from disk is no longer significantly reduced, as shown in Figure 14(b).

Effect of varying the query window. We now study how the size of the query window affects AdaptDB's adaptive repartitioner. As discussed in Section 3.2, AdaptDB uses the queries in the query window to decide when and how frequently to adapt the tree.

We used a different *shifting* workload over TPC-H queries q_{14} and q_{19}. We chose q_{14} and q_{19} instead of other query templates for two reasons. First, both join the `lineitem` and `part` tables, so AdaptDB does not need to use adaptive repartitioning to adapt to a new join attribute when shifting queries, which is not the focus of this experiment. Second, both queries have selective selection predicates on the `lineitem` table, so we can see how AdaptDB's adaptive repartitioner works more clearly.

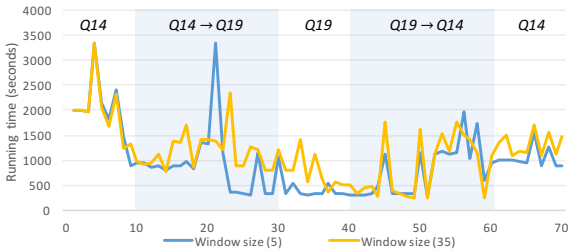


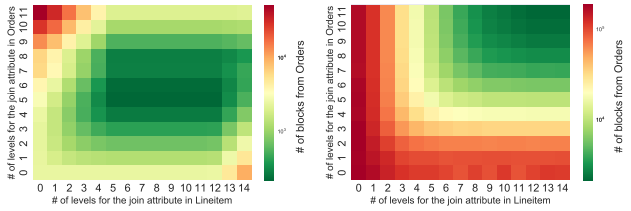
Figure 15: Execution time on varying length of query window

There are 70 queries in this workload. We first run 10 queries from query template q_{14}. In the following 20 queries, We then gradually shift the query from q_{14} to q_{19}. The probability of running query q_{19} (q_{14}) is increased (decreased) by 1/20th after each query. We then run 10 queries from query template q_{19}. In the following 20 queries, we then gradually shift the query from q_{19} back to q_{14} as we did before. Finally, we run 10 queries from query template q_{14}.

We run the above workload using two different window sizes: 5 and 35. 5 is a small window size but still sufficient for AdaptDB's optimizer to estimate the benefit over repartitioning, while 35 is a big window size which is the half of the number of queries in the workload. From Figure 15, we can observe that AdaptDB adapts the partitioning more quickly if the window size is smaller. For example, the blue line is always the first to converge to the best performance in the figure. The blue line also exhibits larger spikes, e.g., at query 21, whereas the yellow line spreads the repartitioning cost out over more time. Besides being more volatile, faster convergence can lead to overfitting the recent workload, adapting significantly even when only a few queries of a particular type arrive.

Effect of varying the number of levels for the join attribute in partitioning trees. We now study how the number of levels for the join attribute in partitioning trees affects the performance of AdaptDB's hyper-join algorithm. We handcrafted a query based on q_{10} from TPC-H where table `customer` is discarded. We chose q_{10} instead of other query templates because there are selective predicates on both tables we are interested in. We varied the number of levels for the join attribute from 0 to 14 in the `lineitem` partitioning tree, since there are at most 2^{14} data blocks. Likewise, we adjusted the number of levels from 0 to 11 in the partitioning tree of `orders`. The size of memory buffer is set to 4GB. Without

loss of generality, we build hash tables over the `lineitem` table and probe them with the `orders` table. We report the number of data blocks scanned from `orders` in Figure 16 as we probe the hash tables using AdaptDB's hyper-join algorithm.



(a) TPC-H 10 w/o `Customer` (b) `Lineitem` ⋈ `Orders`

Figure 16: Effect of varying # of levels for join attributes

From Figure 16(a), we observed that when we set the number of levels for the join attribute to 0 in both tables, a large number of data blocks were scanned when probing the hash tables even though some data blocks are filtered by the partitioning trees. When every level is reserved for the join attribute in `orders` but no levels for the join attribute in `lineitem`, the performance is even worse because no data block from `orders` is filtered by the query's predicates. From the figure, we can see that the number of data blocks achieves its minimum when around half of levels are reserved for join attributes in both tables, which is why we chose this as the default in our experiments.

For the sake of completeness, we also report the result when there are no predicates on either table in Figure 16(b). We can see that the more levels we reserve for join attributes, the fewer data blocks need to be scanned when probing the hash tables since both tables are partitioned on the join attribute. In real scenarios, this is would not like to occur, since it is unusual to join large tables without predicates. It does suggest, however, that a future exploration of adapting the number of join levels in the tree could be worthwhile for some non-selective workloads.

7.5 Approximation Algorithm vs ILP

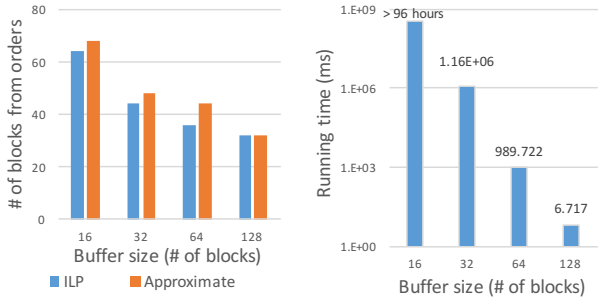
When using the hyper-join algorithm, AdaptDB builds hash tables over the first table and probes them with the second one. The main challenge for our approximate grouping algorithm is to generate a grouping scheme which minimizes the amount of data read from the second table. For this experiment, we run TPC-H scale-factor 10, since the mixed integer programming solver does not scale to TPC-H scale-factor 1000. The `lineitem` and `orders` tables are joined as they are the largest tables in TPC-H. We set the number of blocks in `lineitem` to 128 and the number of blocks of `orders` to 32. In this way, each block size is around 64MB.

We implemented the ILP-based grouping algorithm in AdaptDB using the GLPK solver[2]. We build hash tables on `lineitem` and probe it with `orders`. We report the number of blocks read from `orders` in Figure 17(a); the approximate algorithm performs reasonably well but runs much faster, as shown in Figure 17(b). Here, when we set the buffer size to 32 blocks, the ILP solver needs around 20 minutes. If we set the buffer size to 16 blocks, it cannot find the optimal solution in 96 hours. In contrast, our approximate algorithm always give a reasonably good solution in a millisecond.

7.6 AdaptDB on a Real Workload

We next study the performance of AdaptDB on a real dataset, which is obtained from Cambridge Mobile Telematics (CMT), a

[2] `http://www.gnu.org/software/glpk/`



(a) Optimizer's performance (b) Optimizer's runtime

Figure 17: Varying buffer size on TPC-H with scale 10

Boston-area startup company. We ran the queries on a synthetic version of the dataset, but used an actual query trace. The dataset consists of three tables (a list of trips recorded, a table of historical processed results for each trip, and a table of the most recent processed result for each trip). Most queries in the workload either lookup a trip, or a combination of metadata about the trip and its historical processing, although a few look up the most recent processed result as well.

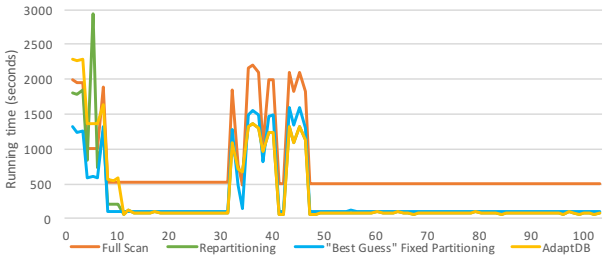


Figure 18: Execution time on CMT dataset

There are 103 queries in the workload. Figure 18 shows the runtime of each single query. AdaptDB spent 9 hours and 51 minutes running all the queries compared to 20 hours and 47 minutes when using full scan with shuffle joins. AdaptDB with full repartitioning (and hyper-join) spent 9 hours and 11 minutes running all queries. Even though the overall running time of AdaptDB with full repartitioning is 40 mins faster, the latency of the fifth query is greatly increased. The spike at query 5 corresponds to the full repartitioning, which took 2945 seconds, which greatly increases the latency of the 5th query. Using the adaptive repartitioner, AdaptDB can finish adapting the dataset according to the join attribute in the first 10 queries with an overhead in each query of around 400 seconds (note that the green and yellow lines totally overlap between queries 10 and 100). The spikes between at queries 30 and 50 correspond to a batch of queries that fetch a large fraction of data from the database; many of the other queries are fetching on a small subset of records.

We also compared the performance of AdaptDB's adaptive repartitioning with a hand-tuned fixed partitioning on the CMT workload. Specifically, we selected attributes appearing in the 103 queries to build a partitioning tree for each table by hand. As we can see from Figure 18, as AdaptDB runs more queries from the CMT workload and adapts the underlying partitions to the ones that fit the workload, the query runtime of AdaptDB approaches the runtime of the this hand-tuned fixed partitioning, occasionally doing slightly better as the mix of queries in the workload changes.

This experiment shows that AdaptDB can effectively improve the performance of real query workloads by a significant margin.

8. RELATED WORK

It is well known that join performance can be improved significantly by co-partitioning the tables being joined. This allows tables to be joined as map-only tasks without any intermediate shuffling. Hadoop++ [7] and CoHadoop [10] proposed to co-partition datasets in HDFS to speed up join performance by avoiding shuffle joins. This approach works for well for two tables and fails if multiple tables need to joined to a single table. Reference partitioning [8] allows to co-partition multiple tables that are linked via foreign keys and predicate-based reference partitioning [25] generalizes this further to allow co-partitioning tables linked with different join keys by allowing tuples to replicate in different partitions. However, these approaches still require prior knowledge of the workload. Also, when tables are not properly co-partitioned, they fallback to shuffle joins, which are expensive. AdaptDB is able to instead use partial partitioning and hyper-joins to get good performance on multiple join attributes at the same time.

Adaptive online joins [9] are proposed in an online or streaming setting where static partitioning schemes are infeasible. New tuples coming to AdaptDB can be appended to the corresponding data blocks based on the partitioning trees and AdaptDB can adapt the underlying partitioning scheme based on workloads in an online manner. Flow-Join [20] is designed to detect load imbalance during data shuffling, which can be incorporated into AdaptDB when shuffle join is used. In hyper-join, the query planner can generate a balanced query plan when the philosophy of flat storage model [16] is adopted.

Database cracking [12, 13] is designed to adaptively index data without requiring an upfront query workload. Crack joins [11] extends the idea of cracking to joins. Crack join does dynamic physical reorganization of the join column based on the input queries. Cracking has been used extensively in single node in-memory column-stores. However, cracking cannot be applied directly to distributed data stores as the cost of re-partitioning is very high. Unlike cracking, where every query triggers re-organization, AdaptDB does careful planning for each round of re-partitioning to amortize its cost. Also, AdaptDB adapts the layout of the primary copy of the data while cracking maintains secondary data structures, which can be expensive to maintain in a distributed setting.

Scheduling the data blocks for computing joins on a single machine is also investigated by many researchers [14, 18]. Merrett et al. [14] designed an algorithm to minimize the page access when the size of buffer pool is limited to 2 pages. Pramanik et al. [18] proposed a solution to give an upper bound of page access on condition that each page is accessed once. Both of these works are focused on scheduling the reads on a single node; in contrast, in hyper-join, our objective is to figure out groups of partitions that should be read together on multiple nodes. Besides the single node vs multi-node objective, the hyper-join problem is different from the problem in [18] because in their work the groups are known in advance and the goal is to order the reads of pages to minimize I/Os. Our problem boils down to solving minimum k-subset union problem, which is not the problem solved in this prior work. In this sense, the two algorithms above are orthogonal approaches for computing joins on a single machine and AdaptDB benefits from them when memory on each machine is limited.

9. CONCLUSION

In this paper, we presented AdaptDB, a system for adaptively parititoning data to provide good performance for distributed joins. Specifically, we showed that our new *hyper-join* algorithm can avoid data shuffling by identifying blocks of data in the joined tables that overlap on the join attribute, and then joining just those blocks. We showed that optimally solving the problem of ordering hyper-join blocks is NP-hard, and developed an approximate algorithm that runs in a millisecond or less for reasonably sized datasets. In addition, we described how AdaptDB maintains several partitioning trees, and employs *smooth repartitioning* to move blocks from one tree to the other without immediately repartitioning the whole data set. Our results on both real and synthetic workloads show that AdaptDB provides improved query performance over shuffle joins, and effectively adapts to changes in workloads over time.

10. REFERENCES

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD Conference*, pages 359–370, 2004.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS*, 2011.

[4] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[5] R. Clifford and A. Popa. Maximum subset intersection. *Information Processing Letters*, 111(7):323–325, 2011.

[6] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[7] J. Dittrich, J. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[8] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. In *SIGMOD Conference*, pages 1111–1122, 2008.

[9] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and adaptive online joins. *PVLDB*, 7(6):441–452, 2014.

[10] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[11] S. Idreos. Database cracking: Towards auto-tuning database kernels. *CWI and University of Amsterdam*, 2010.

[12] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[13] M. L. Kersten and S. Manegold. Cracking the database store. In *CIDR*, pages 213–224, 2005.

[14] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *VLDB*, pages 488–498, 1981.

[15] R. V. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD Conference*, pages 1137–1148, 2011.

[16] E. B. Nightingale, J. Elson, J. Fan, O. S. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *OSDI*, pages 1–15, 2012.

[17] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD Conference*, pages 61–72, 2012.

[18] S. Pramanik and D. Ittner. Use of graph-theoretic models for optimal relational database accesses to perform join. *ACM Trans. Database Syst.*, 10(1):57–74, 1985.

[19] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*, pages 430–441, 2013.

[20] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, pages 1194–1205, 2016.

[21] A. Shanbhag, A. Jindal, Y. Lu, and S. Madden. Amoeba: A shape changing storage system for big data. *PVLDB*, 9(13):1569–1572, 2016.

[22] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD Conference*, pages 1115–1126, 2014.

[23] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.

[24] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[25] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD Conference*, pages 17–30, 2015.

[26] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.