Robust Data Partitioning

Alekh Jindal, Anil Shanbhag, Yi Lu

Overview

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 query processing across them.

This chapter summarizes traditional data partitioning techniques, motivates the need for a more *robust* data partitioning over modern ad-hoc query workloads, introduces the concept of hyper partitioning for creating a robust partitioning tree and hyper join to process join queries over such a partitioning tree, and finally discusses repartitioning techniques for adapting the partitioning tree in a robust manner.

Alekh Jindal, e-mail: aljindal@microsoft.com
Anil Shanbhag, e-mail: anils@mit.edu
Yi Lu, e-mail: yil@mit.edu

Traditional Partitioning Approaches

The traditional approach to data partitioning is to split a table on some key, using hash or range partitioning. This helps queries that have selection predicates involving the key go faster, by only accessing the relevant portions of data. Likewise, for queries with joins, queries will benefit when the database is partitioned on attributes involved in the join, due to local co-partitioned join processing in each partition. Because of these performance gains, many techniques have been proposed in the literature.

Workload-based Partitioning

The typical approach is to find a good data partitioning for a given query workload. These approaches assume that the query workload is either provided upfront or collected over time, and try to choose the best partitioning for that workload. Examples include fine-grained partitioning (Curino et al 2010), hybrid of fine- and coarse-grained partitioning (Quamar et al 2013), skew-aware partitioning (Pavlo et al 2012), deep integration of partitioning with the query optimizer (Nehme and Bruno 2011), interdependence of different physical design decisions (Zilio et al 2004), integrating vertical and horizontal partitioning decisions (Agrawal et al 2004), and partitioning a B$^+$-Tree on primary keys (Graefe 2003). Workload-based partitioning need to be reconfigured every time the workload changes.

Multi-dimensional Partitioning

Several partitioning techniques have been proposed for multi-dimensional data, e.g., K-d Trees, R-Trees, and Quad-Trees. These are typically used for spatial data with two dimensions. Other approaches include binary search trees such as splay trees (Sleator and Tarjan 1985) and MAGIC to *decluster* data on multiple attributes (Ghandeharizadeh and DeWitt 1994). Recent approaches layer multi-dimensional index structures over distributed data in large clusters. This includes SpatialHadoop (Eldawy and Mokbel 2015), MD-HBase (Nishimura et al 2011), and epiC (Wang et al 2010), or adapting the multi-dimensional index to the workload in TrajStore (Cudré-Mauroux et al 2010). Commercially, 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.

Big Data Partitioning

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. Apache Hive and SCOPE (Zhou et al 2012). However, such a partitioning is no different than traditional database partitioning since (i) partitioning is a static one time activity, and (ii) the partitioning keys must be known a-priori and provided by users. Recently, (Sun et al 2014) 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 (Aly et al 2015). Apart from single table partitioning, Hadoop++ (Dittrich et al 2010) and CoHadoop (Eltabakh et al 2011) propose to co-partition datasets in HDFS to speed-up join queries. These systems still assume a workload.

Database Cracking

Database cracking (Idreos et al 2007) 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 (Idreos et al 2009). Cracking is designed for in-memory column-stores and it adapts the data to *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, it is very expensive 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.

Robustness

Modern data analytics has newer data partitioning needs. Data science, for instance, often involves looking for anomalies and trends in data. There is 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. 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 it's 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. Furthermore, as the workload is ad-hoc in nature, database cracking cannot be applied as well. Figure 1(a) illustrates the data partitioning dilemma that analysts face with modern workloads.

Analysts 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 (Idreos et al 2007) 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 worse than full scan before gradually improving with successive queries on the new

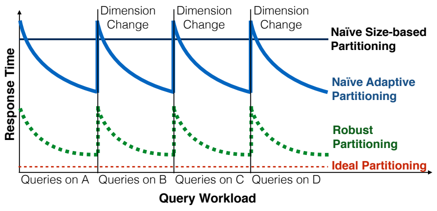


Fig. 1 Need for robust data partitioning.

dimension (referred to as naïve adaptive partitioning). This is really painful for an analyst exploring multiple dimensions: analysts want a data partitioning scheme that is **robust** *to the ad-hoc nature of the modern workloads and provides good performance from the first query itself, adaptively improving from there on*.

Hyper-partitioning

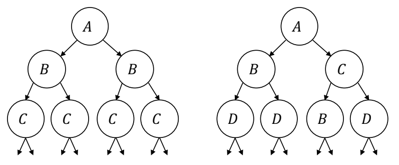
Distributed storage systems, such as HDFS, subdivide a dataset into chunks, called blocks, based on size (usually 128MB). Workload-based partitioning techniques for such systems, including content-based chunking (Bhatotia et al 2011) and feature-based blocking (Sun et al 2014), create blocks such that irrelevant blocks could be quickly skipped for the specific query workload. Hyper-partitioning goes a step further by creating blocks based on a partitioning tree that allows to skip data over almost *all* ad-hoc queries, *without* having any information about the query workload. Such a partitioning also serves as a good starting point for an adaptive query executor to improve upon.

Since hyper-partitioning partitions the data along several dimensions, it could end up de-clustering the data blocks across machines and performing random I/Os for each block. However, this is still fine; large block sizes in distributed file systems (Ghemawat et al 2003) combined with fast network speeds lead to remote reads being almost as fast as local reads (Ananthanarayanan et al 2011; Binnig et al 2016). Essentially, hyper-partitioning sacrifices some data locality in order to quickly locate the relevant portions of the data on each machine in a distributed setting.

The rest of this section first introduces the notions of robust partitioning tree and attribute allocations in that tree, and then describes how to construct and query such a tree.

Robust Partitioning Tree

The hyper-partitioning partitioning tree, or simply the *robust tree*, is represented as a balanced binary tree, i.e., the dataset is successively partitioned into two until it reaches the maximum partition size. For HDFS, hyper-partitioning takes the block size as the maximum partition size. The choice of binary tree is deliberate as it is more general (a four-way partitioning can be achieved by two successive two-way partitioning) as well as fine-granular when adapting the tree to workload 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.



(a) k-d Tree (b) Robust Tree

Fig. 2 Multi-dimensional Partitioning Tree.

Traditional binary partitioning trees, such as k-d tree (Bentley 1975), partition 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. Figure 2 shows a k-d tree where the three levels of the tree divide the dataset on attributes A, B, and C, respectively. In general, 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 = $ 1TB, and $P = $ 64MB, only 14 attributes can be accommodated in the partitioning tree. However, many real-world schemas have way more attributes.

In contrast to k-d tree, the robust tree performs *heterogeneous branching* in order to accommodate more attributes by partitioning different branches of the partitioning tree on different attributes. In other words, robust tree sacrifices 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 2(b) shows a robust partitioning tree. After partitioning on attribute A, the left side of the tree partitions on B while the right side partitions on C. Thus, the tree is 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%. Ad-hoc queries would now gain partially over all four attributes, which makes the partitioning more effective.

The number of attributes in the robust 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 $n = 2$, $D =$ 1TB, $P =$ 64MB 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, hyper-partitioning is even more useful for quickly finding the relevant portions of the data.

Robust tree can further leverage the data replication in distributed storage systems, e.g., 3x replication in HDFS. Such replication mechanisms first partition the dataset into blocks and then replicate each block multiple times. Instead, first the entire dataset is replicated and then each replica is partitioned using a different partitioning tree. While the system is still fault-tolerant (because it has the same degree of replication), recovery becomes slower because it needs to read several or all replica blocks in case of a block failure. Essentially, fast recovery time is sacrificed for improved ad-hoc query performance. Such a scheme can either increase the number of attributes in the partitioning tree, or increase the data fraction covered per attribute. Both of these lead to improved query performance due to greater partition pruning.

Attribute Allocation

The goal of robust tree is to allocate attributes to nodes in the tree such that all attributes have similar advantage in terms of data skipping or parallel processing. Therefore, the allocation of an attribute is defined 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:

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

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 2(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, the 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, the total allocation is distributed 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 case of prior workload information, users can provide relative weights of the attributes and the attribute allocation will be distributed proportional to these weights. The intuition is then to compute the maximum per-attribute allocation, and then place attributes into the tree so as to approximate this ideal allocation.

Hyper Partitioning Algorithm

Algorithm 1 shows the pseudocode to generate the robust partitioning tree. It first calculates the depth of the tree to be created (Line 3), then 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 `LeastAlloc`, which returns the attribute which has the highest allocation remaining. If two or more attributes have the same highest allocation remaining, the algorithm randomly chooses among the ones that have occurred the least number of times in the path from the node to the root. `Med` returns the median of the attribute assigned to this node 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 it partitions the entire dataset into two. Each time it goes to the left or the right subtree, it reduces the data it operates on by half. Once an attribute is assigned to a node, it subtracts from the overall allocation of the attribute (Line 13). The algorithm creates a leaf-level bucket in case it reaches the maximum depth (Line 18).

Algorithm 1: `CreateRobustTree`

```
Input : Int D, Int maxPartitionSize,
        Float[] alloc, Tuple[] initSample
1  Tree tree;
2  numBuckets ← $\lfloor D/maxPartitionSize \rfloor$;
3  treeDepth ← $log_2$ (numBuckets);
4  Queue queue ← {(tree.root, treeDepth, initSample)};
5  while queue.size > 0 do
6      node,depth,sample ← queue.first();
7      if depth = 0 then
8          node ← NewBucket ();
9          Continue;
10     node.attr ← LeastAlloc(alloc);
11     node.val ← Med(sample,node.attr);
12     lS, rS ← SplitSample(node.attr, node.val);
13     node.left ← CreateNode();
14     node.right ← CreateNode();
15     alloc[node.attr] -= $2/2^{\text{maxDepth - depth}}$;
16     depth -=1;
17     queue.add((node.left, depth, lS));
18     queue.add((node.right, depth, rS));
```

Query Processing

A hyper partitioning query processor considers the filter predicates in incoming queries and filters 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 any of the partitions in right subtree of the node don't need to be scanned.

Using Spark, for instance, a job can be constructed where relevant partitions are split into tasks, a set of partitions such that the total size is not more than 4GB. Each task reads the blocks from HDFS in bulk and iterates over the tuples in main-memory. A tuple is returned if it matches the predicates in the query. 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.

This section described hyper partitioning and processing selection queries over a hyper partitioned input. The following section describes techniques for processing join queries over two or more hyper partitioned inputs.

Hyper Joins

Hyper partitioning may end up partially partitioning tables 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, i.e., repartition 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.

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 $\mathscr{M}_i$ has memory to hold 2 partitions to build hash tables on A. Consider building a hash table over A_1 and A_3 on $\mathscr{M}_1$; we will need to read B_1, B_2 and B_3. We then build another hash table over A_2 on $\mathscr{M}_2$ 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 $\mathscr{M}_1$ and another one over A_3 on $\mathscr{M}_2$ 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, finding the optimal collection of partitions to read is NP-Hard. However, heuristically solving the problem can still provide significant performance gains over shuffling. 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. Hyper join provides this property and is designed to move fewer blocks throughout the cluster than a complete shuffle join when tables are not co-partitioned.

The rest of this section formulates hyper join as an optimization problem, presents an optimal solution based on mixed integer programming, introduces an approximate algorithm which can run in a much shorter time, discusses hyper joins for multiple join predicates, and finally shows a two-phase partitioning technique to add join attributes into the robust partitioning tree.

Problem definition

Consider 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 hyper partitioning. Let $V = \{v_1, v_2, \ldots, v_n\}$ be 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

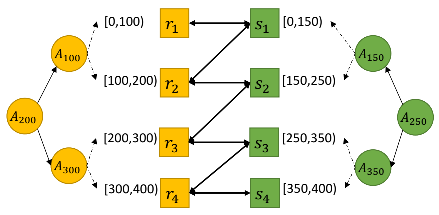


Fig. 3 Illustrating hype join.

subsets of the blocks of R and its union is all blocks in R. Each p_i is constrained to be able to fit into memory of the node performing the join. $\tilde{v}(p_i)$ is used 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 $\delta(v_i) = \sum_{k=1}^{m} v_{ik}$ indicate the number of bits set in v_i. Given a partition p_i, $C(p_i)$ defines the cost of joining p_i with all partitions in S as the number of bits set in $\tilde{v}(p_i)$, i.e., $C(p_i) = \delta(\tilde{v}(p_i))$. This corresponds to the number of blocks to be read to join p_i. Next, the cost function $C(P)$ over a partitioning is defined as 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 3, with table $R = \{r_1, r_2, r_3, r_4\}$ and table $S = \{s_1, s_2, s_3, s_4\}$ and assume $|P| = 2$, i.e., that there is 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. From the figure, r_1 needs to join with s_1, r_2 needs to join with s_1, s_2, etc. Therefore, $V = \{v_1 = 1000, v_2 = 1100, v_3 = 0110, v_4 = 0011\}$. A hash table could be built over multiple yellow partitions to share some disk access of green partitions. For example, a hash table could be built 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, the size of p_i is constrained, 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.

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.,

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

Optimal Algorithm

This section describes a mixed integer programming formulation which can generate the minimal partitioning. Given the maximum number of data blocks B that can be used to build a hash table due to available worker memory, the total number of hash tables to be built are $c = \lceil n/B \rceil$. For each data block r_i from relation R and each partition p_k, the assignment of r_i to partition p_k is indicated with a binary decision variable $x_{i,k} \in \{0,1\}$. Likewise, for each data block s_j from relation S, a binary decision variable $y_{j,k} \in \{0,1\}$ indicates if the j-th bit of $\tilde{v}(p_k)$ is 1.

The first constraint in Problem 1 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,

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

Given a partitioning P, for each partition p_k, every overlapping data block from relation S must also be 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. The proof for NP-hardness of the problem can be found in Lu et al (2017).

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

Fig. 4 A bottom-up approximate solution.

Approximate Solution

Taking B data blocks from relation R with smallest $\delta(\tilde{v}(\mathscr{P}))$ is NP-hard and there is no algorithm for $n^{1-\varepsilon}$-approximation for any constant $\varepsilon > 0$. However, an approximate bottom-up algorithm, as shown in Figure 4, can provide practical runtimes.

The algorithm starts from an empty set of partitions P and an empty partition $\mathscr{P}$. It iteratively adds a data block r_i into $\mathscr{P}$ with smallest $\delta(r_i \vee \tilde{v}(\mathscr{P}))$ until there are B blocks in partition $\mathscr{P}$ or no data block left in relation R. It then adds $\mathscr{P}$ 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 the minimum cost block (requiring a scan of the non-placed blocks) needs to be computed n times.

Joins Over Multiple Relations

Hyper join technique can be extended to multiple inputs. Consider TPC-H query 3. If the join order is (`lineitem` $\bowtie$ `orders`) $\bowtie$ `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, only `tempLO` needs to be shuffled based on `custkey`, and then hyper join can be used instead of an expensive shuffle join, in which both `tempLO` and `customer` need to be shuffled.

With more relations to join, shuffle join over two intermediate outputs of hyper joins 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, changing the join order to (`lineitem` ⋈ `part`) ⋈ (`orders` ⋈ `customer`) can use hyper join twice and a shuffle join over the intermediate results.

Two-phase Hyper Partitioning

Hyper join leverages hyper partitioning, however, the robust partitioning tree described so far, partitions data based solely on the selection predicates. Thus, 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. Two-phase partitioning tackles this challenge by injecting the join attributes into the partitioning tree, as depicted in Figure 5. The first phase splits on join attributes (shown in orange), while the second phase splits on selection attributes (shown in blue). During the first phase, median values of the join attributes are used to recursively split the dataset into two. During the second phase, the join partitions are further partitioned on selection attributes using the standard hyper partitioning.

Consider the left partitioning tree in Figure 3 as an example. There are two

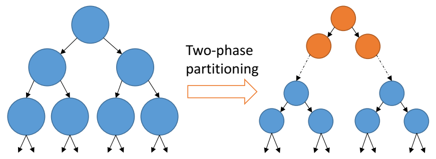


Fig. 5 Illustrating two-phase partitioning

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 $[350,400)$.

Robust Repartitioning

Hyper partitioning and hyper join allow an analyst to quickly get started with her ad-hoc queries. However, the analyst also want the partitioning to adapt as her analysis progresses, e.g., drilling down web click data into successively smaller age groups, to provide even better query performance. *Robust repartitioning* provides the mechanisms to achieve this. When a query is submitted a repartitioning optimizer explores alternative partitioning trees to find the best one and decides whether repartitioning is worthwhile. The optimized plan only accesses data which is to be read by input queries, i.e., data that is not read by queries during repartitioning is not accessed. This has two benefits: (i) data that is not touched by any query is never repartitioned, and (ii) query processing and repartitioning share scans reducing the cost of repartitioning.

The rest of this section describes the cost model used, introduces three basic transformations used to transform a given partitioning tree, describes a divide-and-conquer approach to consider all possible alternatives generated from the transformation rules for inserting a single predicate, discusses how to handle multi-predicate queries, and lastly shows a smooth repartitioning technique to adapt to changing join predicates. 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.

Cost Model

Consider a window (W) of queries that happened in the past X hours. X is a parameter in adaptive query executor and it determines how quickly the system reacts to workload changes. 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 $\text{lookup}(T,q)$ 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:

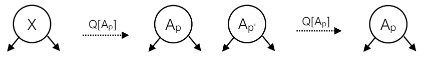
$$\text{RepartitioningCost}(T,q) = c \sum_{b \in B} n_b$$

where c is the write-multiplier i.e., how expensive writes are compared to a read. 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 single node in the tree and a query sequence of the form $\sigma_{A<2}, \sigma_{B<2}, \sigma_{A<2}, \sigma_{B<2} \ldots$. In this case, the data is not constantly repartitioned. After doing it once, say on A, the total cost goes down and hence the repartitioning on B would not happen as Benefit $<$ RepartitioningCost.

Tree Transformations

A set of transformation rules allow exploring the space of possible plans when repartitioning the data. Consider a query predicate of the form $A \leq p$, denoted as A_p. Only partitioning transformations that are local, i.e., that do not involve rewriting the entire tree, are considered. These local transformations are cheaper and amortize the repartitioning effort over several queries. The three basic transformations are discussed below.

(1) Swap. replaces an existing node in the partitioning with the incoming query predicate A_p. As only the accessed data is repartitioned, we consider swapping only those nodes whose left *and* right children are fully accessed by the incoming query. Applying swap on an existing node involves reading both sub-branches, and restructuring all partitions beneath the left subtree to contain data



(a) Different Attribute (b) Same Attribute

Fig. 6 Node swap in the partitioning tree.

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.

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 results in a much higher cost.

(2) Pushup. pushes 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 rearranges the internal nodes without any modification to the leaf nodes.

A pushup transformation is checked every time a swap transformation is performed. The idea is to move important predicates (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 nodes in the upfront partitioning. When such a dormant node is pushed down, subsequent predicates can swap them in an incremental fashion, affecting fewer branches, thus making the tree transformations more robust.

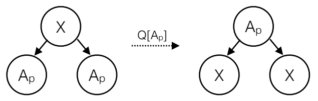


Fig. 7 Node pushdown in partitioning tree.

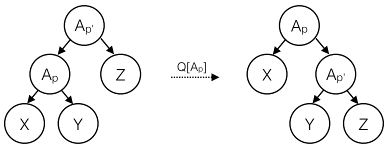


Fig. 8 Node rotation in partitioning tree.

(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 it is always performed wherever possible.

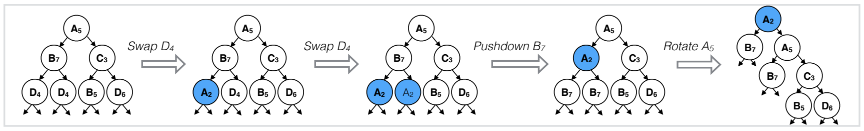


Fig. 9 Introducing predicate A_2 into the partitioning tree.

Above three partitioning tree transformations can be combined to capture a fairly general set of repartitioning scenarios. Figure 9 shows an example, where first nodes D_4 is swapped with incoming predicate A_2 at the lower level, then A_2 is pushed-up one level above, and finally it is rotated with nodes A_5 and C_3. In the process, only half the leaves are repartitioned. Thus, in larger trees, repartitioning mostly happens on small fractions of the data modifying a few subtrees locally.

Divide-And-Conquer Repartitioning

Given a query with predicate A_p and a partitioning tree T, there are many different combinations of transformations that need to be considered. However, 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.,

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

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, the benefit and cost of that transformation is considered and the one which has the best benefit-to-cost ratio is picked. Table 1 shows the cost and benefit estimates for the different transformations. For the swap transformation, denoted as $P_{\text{swap}}(n, n')$, the query costs are recalculated. 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. Applying none of the transformations at a given node is denoted as $P_{\text{none}}(n)$. This approach helps to significantly reduce the candidate set of modified partitioning trees.

Above divide-and-conquer algorithm has a 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. More details on the algorithm can be found in Shanbhag et al (2017).

Repartitioning with Multiple Predicates

A predicate of the form $A \leq p$ 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 $A > p'$.

Now consider a query with two predicates A_p and A_{p2}. The brute force

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	$P_{\text{rotate}}(p, p')$	$C(P(n_{\text{left}\mid\text{right}}))$,for p' on $n_{\text{left}\mid\text{right}}$	$B(P(n_{\text{left}\mid\text{right}}))$,for p' on $n_{\text{left}\mid\text{right}}$
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.

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 of set of remaining nodes to be replaced by A_{p2}. Thus, the number of choices grows exponentially with the number of predicates. A greedy approach is to try to insert each predicate in the query into the partitioning tree. The best among the best plans obtained for different predicates is picked and the corresponding predicate is removed from the predicate set. Likewise, the remaining predicates are inserted 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.

Smooth Repartitioning

A key limitation of the repartitioning techniques presented so far is that they do 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. After the initial two-phase partitioning, with new incoming queries containing a new join attribute, the partitioning tree should also 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. Furthermore, 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. Smooth repartitioning addresses these challenges by maintaining multiple partitioning trees, building each when a new popular join attribute is seen, and migrating blocks between them. The key goal is to adapt partitioning trees in a way that facilitates joins while still maintaining the performance advantages of partitioning for selection queries.

Smooth partitioning creates a new partitioning (initially empty) tree, when it observes a query with a new join attribute. 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. Smooth repartitioning 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, smooth repartitioning can be configured to wait to create a new

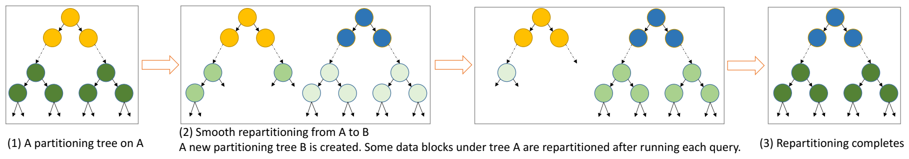

(1) A partitioning tree on A
(2) Smooth repartitioning from A to B
A new partitioning tree B is created. Some data blocks under tree A are repartitioned after running each query.
(3) Repartitioning completes

Fig. 10 Illustrating smooth repartitioning.

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 more queries arrive with the new join attribute, smooth repartitioning repartitions more data into the new partitioning tree using the following algorithm. It first calculates the percentage of two types of queries in the query window and the data in each of the partitioning trees. 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, data from the old partitioning tree is moved to the new one, again by randomly selecting blocks and moving them.

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, a new partitioning tree for B is created 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, both the partitioning trees are maintained with different join attributes. As more queries with join attribute B appear in the query window, more data from the old partitioning tree is repartitioned to the new one. The above procedure is iterated 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 only the partitioning tree for join attribute B is maintained, which is depicted by the last subfigure 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.)

Conclusion

This chapter described new advancements in data partitioning for modern applications that are ad-hoc in nature and do not have any upfront query workload. The key ideas presented include the notion of robustness, the concept of hyper partitioning for creating a robust partitioning tree without upfront query workload, a hyper join technique to efficiently process join queries over hyper partitioned data, and a set of robust repartitioning techniques to steadily adapt the partitioning tree to changes in the workload.

Robust data partitioning revisits the design of a database in the face of modern ad-hoc query workloads, recalibrating the database systems to the expectations of modern users — good performance from the first query itself and adaptively improving from there on.

References

Agrawal S, Narasayya V, Yang B (2004) Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In: SIGMOD

Aly AM, Mahmood AR, Hassan MS, Aref WG, Ouzzani M, Elmeleegy H, Qadah T (2015) Aqwa: adaptive query workload aware partitioning of big spatial data. PVLDB

Ananthanarayanan G, Ghodsi A, Shenker S, Stoica I (2011) Disk-locality in datacenter computing considered irrelevant. In: HotOS

Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM

Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquin R (2011) Incoop: MapReduce for Incremental Computations. In: SoCC

Binnig C, Crotty A, Galakatos A, Kraska T, Zamanian E (2016) The End of Slow Networks: It's Time for a Redesign. PVLDB

Cudré-Mauroux P, Wu E, Madden S (2010) TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets. In: ICDE

Curino C, Jones E, Zhang Y, Madden S (2010) Schism: a Workload-Driven Approach to Database Replication and Partitioning. PVLDB

Dittrich J, Quiané-Ruiz JA, Jindal A, Kargin Y, Setty V, Schad J (2010) Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). PVLDB

Eldawy A, Mokbel MF (2015) Spatialhadoop: A mapreduce framework for spatial data. In: ICDE

Eltabakh MY, Tian Y, Özcan F, Gemulla R, Krettek A, McPherson J (2011) CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. PVLDB

Ghandeharizadeh S, DeWitt DJ (1994) MAGIC: A Multiattribute Declustering Mechanism for Multiprocessor Database Machines. IEEE Trans Parallel Distrib Syst

Ghemawat S, Gobioff H, Leung ST (2003) The Google File System. SIGOPS Oper Syst Rev

Graefe G (2003) Sorting And Indexing With Partitioned B-Trees

Idreos S, Kersten M, Manegold S (2007) Database Cracking. In: CIDR

Idreos S, Kersten M, Manegold S (2009) Self-organizing Tuple Reconstruction In Column-stores. In: SIGMOD

Lu Y, Shanbhag A, Jindal A, Madden S (2017) Adaptdb: adaptive partitioning for distributed joins. PVLDB

Nehme R, Bruno N (2011) Automated Partitioning Design in Parallel Database Systems. In: SIGMOD

Nishimura S, Das S, Agrawal D, El Abbadi A (2011) MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In: MDM

Pavlo A, Curino C, Zdonik S (2012) Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In: SIGMOD

Quamar A, Kumar KA, Deshpande A (2013) SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In: EDBT

Shanbhag A, Jindal A, Madden S, Quiané-Ruiz J, Elmore AJ (2017) A Robust Partitioning Scheme for Ad-hoc Query Workloads. In: SOCC, pp 229–241

Sleator DD, Tarjan RE (1985) Self-adjusting Binary Search Trees. Journal of the ACM

Sun L, Franklin MJ, Krishnan S, Xin RS (2014) Fine-grained Partitioning for Aggressive Data Skipping. In: SIGMOD

Wang J, Wu S, Gao H, Li J, Ooi BC (2010) Indexing Multi-dimensional Data in a Cloud System. In: SIGMOD

Zhou J, Bruno N, Wu MC, Larson PA, Chaiken R, Shakib D (2012) SCOPE: parallel databases meet MapReduce. PVLDB

Zilio DC, Rao J, Lightstone S, Lohman G, Storm A, Garcia-Arellano C, Fadden S (2004) DB2 Design Advisor: Integrated Automatic Physical Database Design. PVLDB