Elastic Data Partitioning for Cloud-based SQL Processing Systems

Lipyeow Lim
University of Hawai‘i at Mānoa
Honolulu, HI 96822, USA
lipyeow@hawaii.edu

***Abstract*—One of the key advantages of cloud computing is the elasticity in which computing resources such as virtual machines can be increased or decreased. Current state-of-the-art shared-nothing parallel SQL processing systems, on the other hand, are often designed and optimized for a fixed number of database nodes. To take advantage of the elasticity afforded by cloud computing, cloud-based SQL processing systems need the ability to repartition the data easily when the number of database nodes is scaled up or down. In this paper, we investigate the problem of supporting elastic partitioning of data in cloud-based parallel SQL processing systems. We propose several algorithms and associated data organization techniques that minimizes the re-partitioning of tuples and the movement of data between nodes. Our experimental evaluation demonstrates the effectiveness of the proposed methods.**

I. Introduction

Cloud computing platforms whether public (eg. Amazon EC2) or private is gaining acceptance as an economical way of sharing and managing computing resources [1]. In the scientific computing community, where users deal with large amounts of data (terabytes and above), running large parallel databases on a cloud computing platform is especially attractive given the exponential growth of scientific data and the severely limited (financial) resources [2], [3]. A key advantage of a cloud computing platform is the ability of scientific users to pay for the computing resources they need, when they need it [4]. To take advantage of this “elasticity” afforded by a cloud computing platform, a cloud-based parallel SQL processing system needs to be able expand and shrink the number of database nodes with ease [5]. Unfortunately, conventional parallel SQL processing systems are designed to run on a dedicated cluster with a relatively fixed number of machines. Changing the number of machines requires a re-deployment of the parallel database which typically involves a mostly manual process of (1) re-partitioning the data, (2) moving the partitioned data to the right database nodes and (3) loading the partitioned data.

In this paper we take a first stab at the problem of supporting elastic data partitioning in a cloud-based parallel DBMS. We consider shared-nothing parallel DBMSs deployed on a cluster of virtual machines in a cloud computing platform. Given that the data is partitioned among p database nodes in a parallel DBMS, how do we efficiently scale this parallel database up or down to q nodes? (Replication of data fragments is deferred to our future work). Can we avoid scanning through each tuple to re-partition the data ? Can we minimize moving data between nodes during the re-organization ? We describe a brute force, naive method for performing the reorganization and propose three novel methods that are significantly more efficient in terms of the number of tuples that need to be re-partitioned and the number of tuples that need to be moved.

The naive method (Method N) is based on a simple matching between the current partitions or fragments and the target partitions or fragments. For range partitioning, these partitions or fragments can be represented as range intervals. Current ranges that straddle multiple target ranges require re-partitioning. Deciding which tuples to move to what database nodes depends on how nodes are assigned to the target ranges. In the worst case, when resizing a parallel database to more nodes, the naive method needs to re-partition all the tuples. We proposed a chunk-based method (Method C) that organizes data using fine-grain partitions called chunks. Partitioning of the data for a parallel database is then required to respect chunk boundaries, thus eliminating the need for re-partitioning tuples during reorganization. Method N and C both respect the partitioning constraint of the partitioning function. For example, for equi-width partitioning function, the equi-width constraint is preserved after the data is reorganized for the target number of database nodes. In Method C the alignment to chunk boundaries may cause the equi-width property to be approximated, but it is still preserved. On one hand, the preservation of the partitioning constraint is a good property from a parallelization and load balancing perspective, on the other hand, it causes significant data move during reorganization.

To minimize data movements, we propose a tree-based method (Method T) that uses a tree of partitions. The tree encodes a set of allowed partitions, thereby avoiding the arbitrary partitions in Method N and Method C that necessitates the large amount of data movement. Method T sacrifices the preservation of the partitioning constraint for a simple way of deciding which fragment to split or which two fragments to merge. Partition boundaries remain more stable across reorganization, thus minimizing data movements.

We also propose a hash-based method (Method H) that specifically addresses hash-based partitioning (in contrast to range partitioning). A deeper analysis of Method H reveals that it is a specific variant of Method T.

Our contributions.

- We introduce and highlight the important problem of supporting elastic data partitioning in a shared-nothing, cloud-based parallel database system. The best of our knowledge this problem has not been addressed in the literature before.
- We describe a straightforward 'naive' algorithm for the problem and propose three algorithms that significantly reduces the need for re-partitioning and data movements. Method C uses the idea of chunks and boundary alignment to eliminate re-partitioning. Method T uses the idea of precomputing a tree of possible partitioning to minimize data movements. Method H extends the previous methods to handle the hash partitioning type.
- We perform a comprehensive empirical evaluation of the proposed methods and present a subset of representative results in this paper.

The rest of the paper is organized as follows. Section II formulates the problem and describes the proposed methods in greater detail. Section III presents an experimental evaluation of the proposed methods. Section IV discusses related work. We draw conclusions in Section V.

II. Our Approach

A. Preliminaries

A relation T is a set of tuples that is (horizontally) partitioned [6], [7] into a set of p partitions or fragments,

$$T = T_1 \cup T_2 \cup \ldots \cup T_p.$$

In a shared nothing parallel DBMS, each data fragment T_i resides on a database node i in the cluster. Each fragment T_i is itself a set of tuples. A partitioning function $\mathcal{F}$ is a mapping of T to $\{T_1, T_2, \ldots, T_p\}$ given p and can be described by several parameters.

Partitioning Type. Two partitioning types are currently in use in state-of-the-art parallel databases. Range partitioning uses a set of disjoint ranges on the values of the partitioning key (described next) to decide which tuple belong to which partition. Hence, the data partitions from a range partitioning can be equivalently represented using the set of disjoint ranges. Hash partitioning, on the other hand, uses a hash function on the partitioning key to compute a hash value for a tuple that is used to decide which partition the tuple belong to. In both partitioning types, the set of ranges or the hash function is typically stored in the database catalog so that queries and updates can be routed to the appropriate database nodes.

Partitioning Key. A partitioning key consists of one or more columns of a relation chosen to be the basis of partitioning the relation. For range partitioning, a comparison function must exist to sort the values of the partitioning key into a total ordering. Without loss of generality, we consider single-column partitioning keys, because the columns in a multi-column partitioning key can always be replaced with a single concatenated column.

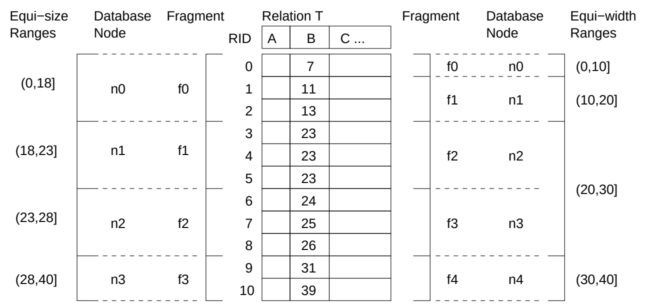


Figure 1. Range partitioning on relation T with partitioning key $T.B$. The left side illustrates equi-size partitioning, the right side illustrates equi-width partitioning.

Partitioning constraint. A partitioning constraint specifies conditions that the partitions of a partitioning function need to satisfy. The following are three partitioning constraints that apply to range partitioning.

- The *equi-width* constraint requires the range of values that the partitioning key takes for each partition to be equal.
- The *equi-size* constraint requires the size of each partition (eg. in the number of tuples) to be equal.
- The *equi-load* constraint requires the workload on each partition (eg. average number of query hits per unit time) to be equal.

Figure 1 illustrates how a relation T can be range partitioned with the equi-width and equi-size partitioning constraint.

Equi-width partitioning computes the ranges based purely on the minimum and maximum possible values of the partitioning key. When the data distribution of the partitioning key is fairly uniform, the number of tuples associated with each range would also be fairly uniform. When some skew is present, the variance in the number of tuples per range becomes significant. Equi-size partitioning attempts to remedy this problem by picking variable width ranges to ensure that the number of tuples associated with each range is approximately uniform. Obviously both partitioning constraints can be foiled by extremely skewed distributions. For example, when one particular value of the partitioning key occurs in 50% of the relation, equi-size partitioning cannot split a single value into two "ranges". Equi-load partitioning is similar to equi-size except that instead of balancing the number of tuples per range, it balances some load measure such as number of query hits etc. Note that other partitioning constraints are possible depending on the application and requirements.

A partitioning function $\mathcal{F}(T, a, p)$ partitions a relation T into p fragments according to some partitioning constraint on a user-chosen partitioning attribute a. In this paper, the relation T and the partitioning attribute a often remains constant in the equations and algorithms, hence, we will omit relation name and partitioning attribute arguments or subscripts for readability. We denote the fragments resulting from applying the partitioning function $\mathcal{F}(p)$ as Π_p. In the case of range partitioning, the resulting set of ranges or intervals on the partitioning attribute is denoted as $\mathcal{R}_p$. In the case of hash partitioning, the notation $\mathcal{R}_p$ will denote the set of bit strings that encode the address of the hash bucket. In most cases, there will be a one-to-one correspondence between ranges (or hash buckets) and fragments. However, when there is significant skew in the values of the partitioning key, it is possible that two fragments (hence database nodes) are required for a single range interval (or hash bucket). However, for ease of exposition, we assume a one-to-one correspondence between ranges (or hash buckets) and fragments, and between fragments and database nodes. Hence fragments and ranges (or hash buckets) will often be used interchangeably in the context of range partitioning. Investigating strategies for managing skewed non-one-to-one correspondences is part of our future work.

Problem Statement. Without loss of generality consider a single relation T that is initially partitioned into p fragments and needs to be resized to q fragments. Given

- a relation T,
- a partitioning function $\mathcal{F}$ on a fixed partitioning key,
- an initial number of partitions p,
- an initial mapping of data fragments to p database nodes $Node_p(\cdot)$,
- a target number of partitions q,

find a mapping of $\{T_1, T_2, \ldots, T_p\}$ to $\{T_1, T_2, \ldots, T_q\}$, and a mapping of the q data partitions to database nodes that minimizes the number of tuples re-partitioned, and the number of tuples moved between database nodes.

B. Method N: Naive Resize

Consider a table T that is initially partitioned into p data partitions using range partitioning function $\mathcal{F}$ on some partitioning key. These p data partitions are then assigned to p database nodes. Suppose we want to resize the number of partitions to q. The following steps illustrates the naive approach.

1) Find the set of q ranges according to $\mathcal{F}$.
2) Assign existing and/or new database nodes to the q ranges using a heuristic such as maximizing the number of common tuples — the ASSIGNNODES algorithm (pseudocode in Appendix A Algorithm 1).
3) Match up the current and the new ranges to decide which tuples need to be re-partitioned and/or moved to a different node — the NAIVERESIZERANGE algorithm (pseudocode in Appendix A Algorithm 2)

Step 1 is straightforward. Step 2 tries to assign existing database nodes to the q partitions and where $q > p$, new nodes are assigned as well. We formalize Step 2 in the *node assignment problem*: Given a p-node database with ranges $\mathcal{R}_p$ and a target number of nodes q with ranges $\mathcal{R}_q$, we wish to find the assignment of nodes to the target ranges $\mathcal{R}_q$.

In general, we would like to assign the p nodes and add or remove the $|p-q|$ nodes in order to optimize some measure (in this case, maximize the number of common tuples between the assigned ranges). The intuition then is to sort all pairs of overlapping ranges between $\mathcal{R}_p$ and $\mathcal{R}_q$ according to the measure and assign $Node_p(r_i)$ to s_j where r_i and s_j have the most number of tuples in common. The algorithm examines all pairs of ranges between the current p ranges and the target q ranges. The pairs are sorted in decreasing order of the number of common tuples. The current nodes that have not been assigned are assigned to the range with the largest number of common tuples. After all current nodes have been assigned, target ranges that do not have an assigned node are assigned new nodes. The conceptual cross product in the ASSIGNNODES algorithm to find all overlapping pairs can be implemented very efficiently using a linear scan similar to merging two sorted lists. Finding the number of common tuples can also be approximated using histogram techniques [8] in practice.

At the end of Step 2, we have the current ranges $\mathcal{R}_p$ with their node assignments and the target ranges $\mathcal{R}_q$ with their node assignments. Step 3 uses the NAIVERESIZERANGE Algorithm to match the current and target ranges in order to move and/or re-partition the tuples associated with a current node. The algorithm first deals with all the current ranges that are completely contained in a target range. The tuples associated with these fully contained ranges do not need to be re-partitioned, but moved to the target node if their target node is different. The algorithm then proceeds to deal with those current ranges that are mapped to two or more target ranges. The tuples associated with these current ranges need to be re-partitioned and moved to the target nodes (if their target node is different from their current node).

Example 1 (Naive Resize): Consider the following ranges and node assignments for decreasing the number of database nodes from p=5 to q=4,

$$\begin{aligned}
\mathcal{R}_p &= \{[0,8],(8,16],(16,24],(24,32],(32,40]\}. \\
\mathcal{R}_q &= \{[0,10],(10,20],(20,30],(30,40]\}. \\
Node_p(\cdot) &= \{1,2,3,4,5\}, \\
Node_q(\cdot) &= \{1,2,4,5\}.
\end{aligned}$$

The NAIVERESIZERANGE algorithm would first iterate over target ranges $\{(0,10],(30,40]\}$, because there exists some current ranges that are fully contained in these target ranges,

$$(0,8] \subset (0,10] \quad \text{and} \quad (32,40] \subset (30,40].$$

However, since these subsumed ranges map to the same database nodes, no movement of tuples are required. The second part of the algorithm would iterate over the current ranges $\{[8,16],(16,24],(24,32]\}$ because these ranges straddle at least two target ranges each.

- For current range $(8,16]$, the tuples will be re-partitioned to $\{(0,10],(10,20]\}$ and tuples for $(0,10]$ will be moved to node 1.
- For current range $(16,24]$, the tuples will be re-partitioned to $\{(10,20],(20,30]\}$ and moved to node 2 and node 4 respectively.
- For current range $(24,32]$, the tuples will be re-partitioned to $\{(20,30],(30,40]\}$ and the tuples for $(30,40]$ will be moved to node 5.

Compared to moving tuples from one database node to another database node, re-partitioning is a relatively expensive operation. Our experiments on the Amazon EC2 cloud showed that moving a TPC-H lineitem tuple between two instances in the same U.S. east coast zone takes 0.0225 ms on average and partitioning a lineitem tuple on a small on-demand instance takes 0.0478 ms on average. Partitioning requires every tuple to be scanned from disk, the value of the partitioning key accessed for partitioning, and then written out to disk. The re-partitioned tuples can then be moved in a batch fashion to the corresponding target nodes. Consider the case where the same equi-width partitioning function is used to compute both the current and target ranges. If the number of target partitions q is greater than the number of current partitions p, it is clear that all p partitions need to be re-partitioned. If q is less than p, the number of current partitions that need to be re-partitioned is dependent on the greatest common divisor between p and q. More formally,

$$\text{\# partitions to be split} = \begin{cases} 0 & p = q \\ q & p < q \\ q - gcd(p,q) & p > q \end{cases} \quad (1)$$

In fact, this characterization holds as long as the same range partitioning function is used to compute the current and target ranges, irrespective of the partitioning constraint. Observe that the number of partitions that need to be split becomes zero whenever p is a multiple of q and hence $gcd(p,q) = q$. This observation leads us to a method of resizing without re-partitioning tuples.

C. Method C: Chunk-based Elastic Resize

The key ideas to a method of elastic resizing of a cloud-based parallel DBMS without splits are (1) pre-partition the data into k fine-grained partitions called *chunks* ($k \geq p, k \geq q$), and (2) any range partitioning function must align ranges to chunk boundaries. Since any changes to the ranges are always aligned to chunk boundaries, the tuples within a chunk are never re-partitioned.

Initialization. An additional initialization step is required. The data table is initially partitioned into k chunks using any existing range partitioning function. Subsequent partitioning will need to use an approximate range partitioning function that aligns ranges to chunk boundaries.

Choosing k. The parameter k should be chosen as the maximum number of database nodes that would ever be deployed until the next major database re-organization when the data can be re-partitioned according to a new k. A large k would result in small chunks that could better isolate changes when resizing from p to q, but it also translates to more chunks that need to be managed by the database catalog.

Alignment to chunk boundaries. A straightforward technique for chunk boundary alignment would be to include a chunk in a range if at least 50% of the chunk overlaps with the range. For example, to decide if a range $(0,11.5]$ should include the chunk $(10,12]$, we compute the percentage overlap $(11.5-10)/2.0 = 75\%$ and since the overlap is significant, the range is adjusted to include the chunk, i.e., $(0,12]$. The subsequent range then starts at 12.

Resizing. The following steps outlines the chunk-based elastic resize approach.

1) Find the set of q ranges according to $\mathcal{F}$.
2) Align the q ranges to chunk boundaries.
3) Assign existing and/or new database nodes to the q ranges using the AssignNodes algorithm.
4) Match up the current and the new ranges to decide which chunks need to be moved to a different node using the chunk-based ElasticResizeRange algorithm (pseudocode in Appendix A Algorithm 3)

Steps 1-2 are straightforward. Step 3 uses the same node assignment algorithm to obtain a node assignment for the q ranges. Step 4 uses the ElasticResizeRange algorithm which is a modified version of the NaiveResizeRange algorithm. The main difference lies in how the two algorithms deal with current ranges that straddle two or more target ranges. In the chunk-based method, re-partitioning of tuples is no longer needed. Instead, ElasticResizeRange iterates through each chunk in the current range, finds the target range that the chunk maps to, and move the chunk to the node for that target range if the chunk is on a different node. In another sense, we have replaced tuple-based re-partitioning to chunk-based re-partitioning. If each chunk holds an average of $\frac{n}{k}$ tuples, where n is the total number of tuples in the relation, then chunk-based processing represents a factor of $\frac{n}{k}$ improvement in re-partitioning cost.

Example 2 (Chunk-based Resize): Consider the following ranges and node assignments,

$$\begin{aligned} \mathcal{R}_k &= \{(0,2.5],(2.5,5],\ldots,(37.5,40]\}, \\ \mathcal{R}_p &= \{[0,10],(10,20],(20,30],(30,40]\}, \\ \mathcal{R}_q &= \{[0,7.5],(7.5,15],(15,25],(25,32.5],(32.5,40]\}, \\ Node_p(\cdot) &= \{1,2,3,4\}, \\ Node_q(\cdot) &= \{1,2,5,3,4\}. \end{aligned}$$

Since there are no current ranges that are fully contained in a target range, ELASTICRESIZERANGE simply iterates through each range in $\mathcal{R}_p$ and perform the following.

- For current range $(0, 10]$, the chunk $(7.5, 10]$ will be moved to node 2.
- For current range $(10, 20]$, the chunks $(15, 17.5]$ and $(17.5, 20]$ will be moved to node 5.
- For current range $(20, 30]$, the chunks $(20, 22.5]$ and $(22.5, 25]$ will be moved to node 5.
- For current range $(30, 40]$, the chunk $(30, 32.5]$ will be moved to node 3.

A natural question to ask next is whether the data movements can be minimized. If we insist on the target ranges satisfying the same partitioning constraint, then the answer is unfortunately no. If we are able to accept target ranges that partially satisfy the partitioning constraint, tree-based techniques can be used to reduce the amount of data movement.

D. Method T: Tree-based Elastic Resize

In the previous methods, the data of the partitioned relation is re-distributed to satisfy the partitioning constraint on the target ranges resulting in much data movements. In the tree-based method, we relax the need to satisfy the partitioning constraint. In addition to the key ideas introduced in the chunk-based method, the tree-based method relies on the idea of a pre-computed hierarchy of ranges (see Fig. 2) where each partitioning into p or q ranges corresponds to finding a cover of (internal) tree nodes that subsumes all the leaf nodes.

Initialization. Before the parallel database is deployed on p nodes, the following steps are used to initialize the tree of ranges.

1) Partition T into $k > p$ special fragments which we shall call chunks. Choose k to be a power of 2.
2) Construct a binary tree in bottom-up fashion where each leaf is associated with one of the k chunks.
3) Find a set of p (internal) nodes that cover all the leaves. Each of the p fragments consists of the chunks associated with the leaves descended from that (internal) node. Algorithm 4 FINDTREECOVER can be used.

Finding a tree cover. The key algorithm in the tree-based method is the FINDTREECOVER algorithm (pseudocode in Appendix A Algorithm 4). Given a tree of ranges $\mathcal{T}$, the current set of ranges $\mathcal{R}_p$, and a target number of ranges q, we wish to find a set of q tree nodes that subsumes or cover all the leaf nodes. Note that each leaf node is associated with a chunk and hence a chunk range. The current set of ranges $\mathcal{R}_p$ is aligned to chunk boundaries and hence has a one-to-one correspondence to a set of tree nodes. In this section we will refer to (chunk aligned) ranges and tree nodes synonymously because of this one-to-one correspondence. The algorithm finds a cover as follows. Start with the current ranges as an initial cover. Let c be the number of tree nodes in the current cover. If $c < q$, increase number of tree nodes in the cover by finding a tree node in the current cover to split. If $c > q$, decrease number of tree nodes in the cover by finding the two sibling tree nodes in the current cover to merge. Repeat until $c = q$.

How do we pick the nodes for splitting or merging ? The heuristic used for picking nodes can be used to approximate the partitioning constraint.

- Pick the tree node with the widest range to split. Pick the two sibling nodes with the narrowest range to merge.
- Pick the tree node with the most number of tuples to split. Pick the two sibling nodes with the least number of tuples to merge.
- Pick the tree node with the most number of query hits to split. Pick the sibling nodes with the least number of query hits to merge.

Note that using heuristics such as the above requires maintaining a priority queue which could be expensive. A computationally simple heuristic would be to split the leftmost, highest node in the tree that is in the current cover and to merge the rightmost, lowest two siblings in the tree that is in the current cover.

Returning to our discussion of Step 3 of the initialization, we can simply call FINDTREECOVER using the root node of the tree as the current cover and the algorithm will keep splitting till p nodes are obtained.

Resize. The following steps outline the tree-based resize method for resizing from p partitions to q partitions. The chunk ranges $\mathcal{R}_k$ and the tree of ranges $\mathcal{T}$ are assumed to be stored in the database catalog.

1) Find the q ranges given the tree $\mathcal{T}$ using the FINDTREECOVER algorithm (pseudocode in Appendix A Algorithm 4)
2) Resize the parallel database system using the ELASTICRESIZERANGE algorithm (Appendix A Algorithm 3)

Note that an alternative to the steps outline above is to keep track of the current ranges as a list of tree nodes. Let c be the number of current ranges. If $c < q$, increase number of database nodes by finding a tree node in the current ranges to split, and move the chunks associated with the right child to a new database node. If $c > q$, decrease number of database nodes by finding the two sibling tree nodes to merge, and move the chunks associated with right sibling to the left sibling (the database node associated with the left sibling gets "promoted", i.e., associated with the parent tree node). Repeat until $c = q$. Note that since data movement is perform one tree node at a time, it is possible that a particular chunk can be moved multiple times compared to the Step 2 using the ELASTICRESIZERANGE algorithm.

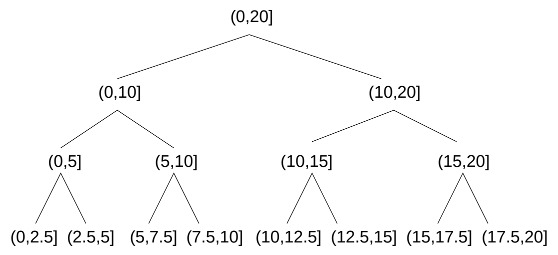


Figure 2. Example of a tree of ranges.

Example 3 (Tree-based Resize): Consider partitioning a table T with partitioning attribute values from $(0, 40]$. Suppose we choose $k = 16$ and use the equi-width partitioning function to obtain the chunk ranges,

$$\mathcal{R}_k = \{(0, 2.5], (2.5, 5], \ldots, (37.5, 40]\} .$$

The binary tree of ranges for $\mathcal{R}_k$ is illustrated in Fig. 2. Suppose we wish deploy the parallel database with $p = 4$ database nodes initially. Using the split-leftmost-highest node heuristic, we obtain the following covering ranges,

$$\mathcal{R}_p = \{[0, 10], (10, 20], (20, 30], (30, 40]\} .$$

Suppose, we wish to increase the number of database nodes to $q = 5$. The FINDTREECOVER algorithm picks a range from among $\mathcal{R}_p$ to split. If we use the split-leftmost-highest node heuristic, the range $(0, 10]$ will be split into $(0, 5]$ and $(5, 10]$, resulting in

$$\mathcal{R}_q = \{[0, 5], (5, 10], (10, 20], (20, 30], (30, 40]\} .$$

If we use the heuristic that picks the range with the most tuples, then using the data in Fig. 1, the range $(20, 30]$ will be chosen, because it contains 6 tuples. Hence,

$$\mathcal{R}'_q = \{[0, 10], (10, 20], (20, 25], (25, 30], (30, 40]\} .$$

Using ELASTICRESIZERANGE on $\mathcal{R}'_q$, a new database node is instantiated and only the two chunks $\{(25, 27.5], (27.5, 30]\}$ is moved to the new database node.

E. Implications for Query Performance

Consider a simple query that scans the entire partitioned relation $T = T_1 \cup T_2 \cup \ldots \cup T_p$. The running time of a table scan query $Q(T)$ is the time of the longest scan on each of the data partitions, $Q(T) = \max_{i=1}^{p} Q(T_i)$. The performance of table scans is dependent on the size of each data partition T_i which in turn is dependent on the partitioning constraint. A detailed exposition of the relationship between parallel query performance and partitioning constraint is beyond the scope of this paper. When the partitioning key is relatively uniform, equi-width partitioning would ensure relatively balanced partitions. Method N and C tries to preserve the partitioning constraint and hence would also preserve the query performance characteristics. Method T, on the other hand, does not try to preserve the partitioning constraint. The variance in the partition sizes would depend on the heuristic used to pick splitting/merging nodes. For the simple split-leftmost-highest and merge-rightmost-lowest heuristic, the difference in partition sizes for uniform data can be bounded by a factor of two, since the partitions for a p-node database correspond to tree nodes at the $\lfloor \log p \rfloor$ and $\lceil \log p \rceil$ levels only. While this is true for equi-width range partitioning on uniform data, hash partitioning, on the other hand, is inherently random and the partition sizes are not balanced in general.

III. EXPERIMENTAL EVALUATION

We evaluated the effectiveness and efficiency of the proposed methods on the TPC-H benchmark data as well as synthetically generated data from the uniform and Zipf distributions.

For the TPC-H data, we used the lineitem table generated using a scale factor of 0.01 and 1.0 (henceforth denoted as TPC-H(0.01) and TPC-H(1.0)). The lineitem table with scale 0.01 contains 60,175 rows and about 7.5 MB of data, while the table with scale 1.0 contains 6,001,215 rows and about 759 MB of data. We experimented using the attributes `L_ORDERKEY, L_PARTKEY, L_SUPPKEY` as the partitioning key.

The synthetic data are generated by drawing the values of the partitioning key from the uniform distribution and the Zipf distribution. A ‘uniform’ dataset is generated by drawing 100,000 random variates between zero and 500 from a uniform distribution. A ‘Zipf 1’ dataset is generated by drawing 10,000 random variates in the same range from a Zipf distribution with exponent one. A ‘Zipf 2’ dataset is generated by drawing 10,000 random variates in the same range from a Zipf distribution with exponent two. Note that since Zipf random variates represent the rank and not the data value itself, we randomly permute the generated ranks to simulate values. For random datasets, experiments are run over 10 randomly generate datasets with the same parameters in order to observe average behavior.

Four performance metrics are used: (1) elapsed running time in a serial execution, (2) elapsed running time in a parallel execution, (3) the number of tuples moved from one node in the cluster to another, (4) the number of tuples re-partitioned. The number of tuples moved and re-partitioned are expressed as a percentage of the total number of tuples in the table. The number of tuples re-partitioned counts the number of tuples of which the value of the partitioning key needs to be examined in order to map the tuples to the appropriate partition. We present a subset of our experimental results due to space constraints.

A. End-to-End Efficiency

In this experiment, we measure the elapsed time of using our proposed method to resize a cloud-based parallel

	Method N	Method C	Method T
Algorithm	3.369	3.526	0.006
Re-partition	363.657	0	0
Move+Load	266.452	261.236	226.848
Total Serial	533.478	264.762	226.854
Total Parallel	321.391	134.290	120.899

Table I
TIME (SECONDS) TO SCALE A 2-NODE DATABASE TO 11 NODES ON EC2.

	Method N	Method C	Method T
Algorithm	3.382	3.909	0.002
Re-partition	167.892	0	0
Move+Load	203.558	204.611	85.867
Total Serial	533.478	208.520	85.869
Total Parallel	66.730	39.795	22.608

Table II
TIME (SECONDS) TO SCALE AN 11-NODE DATABASE TO 7 NODES ON EC2.

database from p nodes to q nodes running on the Amazon EC2 cloud. For initialization, we partition and load the TPC-H(1.0) dataset on a cluster of p IBM DB2 database instances running on small, on-demand instances. The partitioning key is `L_ORDERKEY` and the partitioning constraint is equi-width. We then use the proposed method to elastically resize the database to q instances. If $q > p$, we assume that additional EC2 instances are already started and do not count the time required for EC2 instances to be started. We also assume that the data fragment at each database node also exists as a CSV file at that node. We measure the following,

- the running time of the proposed method,
- the time to re-partition the data fragment for target partitions at each node,
- the time to send the re-partitioned data to the target node,
- the time to load the re-partitioned data into the database instance at the target node,
- the total time required if all the steps were executed serially, and
- the total time required if the re-partitioning, move, and load steps are executed concurrently at each node in parallel.

Table I shows the results for p=2, q=11. For Method C and T, 128 chunks were used. The first observation is that the time required for executing the proposed methods is negligible compared to the time required to re-partition, move or load the data fragments. Among the proposed methods, Method T is the speediest. Second, re-partitioning is more expensive to moving and loading data. Since q is larger than p, every tuple needs to be re-partitioned under Method N. Method C and T keep track of tuples at the chunk level, and thus avoid the costly re-partitioning operation. Third, the time for moving and loading is roughly the same for Method N and C, but Method T is almost 40 seconds faster. This is due to number of tuples that needed to be moved under the three methods: 4,909,377 for Method N, 4,875,307 for Method C, and 4,125,866 for Method T. Fourth, since p=2, the level of parallelism is limited to two.

Table II shows the results for p=11, q=7. For Method C and T, 128 chunks were used. In this case, Method T is the clear winner, since it minimizes the number of tuples that need to be moved. In fact, only 4 out of the 11 partitions need to be moved. For Method N and C, 6 out of the 11 partitions need to be split and moved. The number of tuples moved are 3,272,833 for Method N, 3,187,478 for Method C, and 1,500,796 for Method T. The amount of concurrency for Method N, C, T is 6,6,4 respectively which accounts for the improved parallel time compared to the previous case.

In both cases, we observe speedups of 2 and 3 for Method C and Method T over the naive Method N. We also note that the number of tuples moved and the number of tuples re-partitioned are good predictors of performance; hence, we focus on these two measures for the rest of the paper.

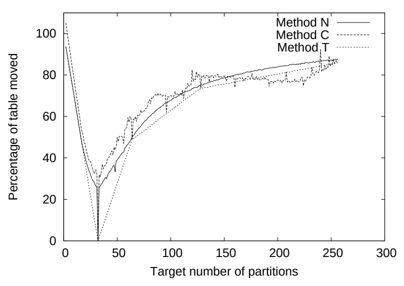
B. Elasticity Characteristics

We wish to understand how "elastic" we can scale our cloud-based parallel database from a fixed p number of nodes to varying q numbers of nodes. We measure the number of tuples moved and re-partitioned as a percentage of the total number of tuples for p=32 and $q \in \{2, 3, 4, \ldots, 256\}$. For Method C and T, 256 chunks were used.

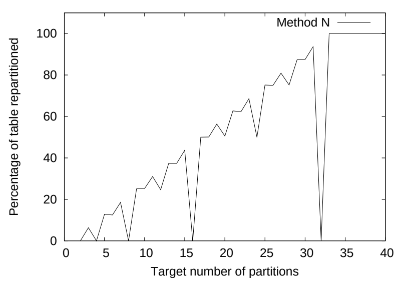
Fig. 3(a) and Fig. 3(b) shows the results for TPC-H(0.01) dataset. Observe the general shape of the curves in Fig. 3(a). As $|q - p|$ increases, the number of tuples to be moved increases. The ideal curves should have gradual slopes as $|q-p|$ increases. Note that Method T is almost always better in terms of the number of tuples moved, but Method N and T are about the same. Recall that Method N requires re-partitioning whereas Method C and T does not. Hence Method C is still superior to Method N. Fig. 3(b) shows the periodic nature of the re-partitioned tuples as governed by Eqn. 1. Recall that when $q > p$, all tuples need to be re-partitioned under Method N.

Fig. 3(c)-3(f) shows the results for synthetically generated data. The synthetic datasets 'Uniform', 'Zipf 1', and 'Zipf 2' represents datasets with increasing skew. Observe that the curves shift lower with increasing skew. While this seems to be a positive, but puzzling result, it can be explained by a small number of partitions holding a large portion of the data in skewed datasets. For equi-width partitioning, the effect is that the likelihood of moving a partition with a small number of tuples increases with skew, resulting in less tuples moved in total. Fig 3(f) seems to indicate that skew does not affect the number of re-partitioned tuples significantly for Method N.

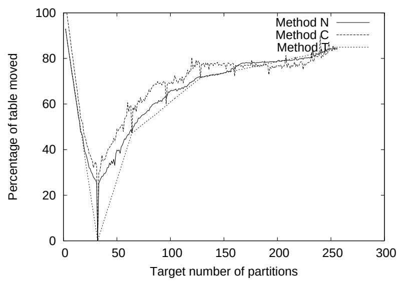
Number of Chunks. We also investigated the effect of the number of chunks (the parameter k in Method C and



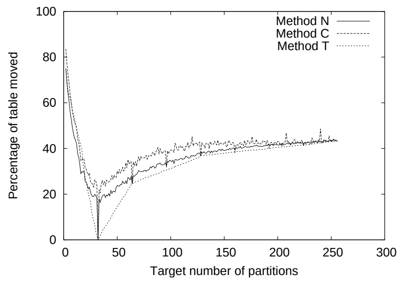
(a) TPC-H(0.01) : Tuples moved



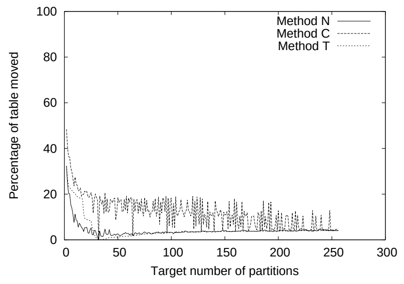
(b) TPC-H(0.01) : Tuples re-partitioned



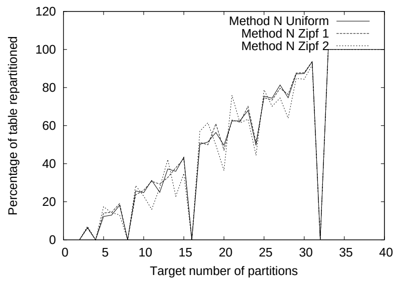
(c) Uniform : Tuples moved



(d) Zipf 1 : Tuples moved



(e) Zipf 2 : Tuples moved



(f) Uniform+Zipf : Re-partitioned Tuples

Figure 3. Elasticity of scaling from 32-nodes to $\{2, 3, 4, \ldots, 256\}$-nodes for TPC-H(0.01) and synthetic datasets.

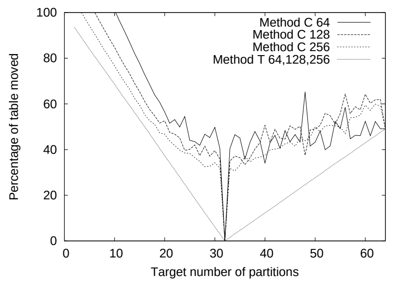


Figure 4. Elasticity of scaling from 32-nodes to $\{2, 3, 4, \ldots, 256\}$-nodes for TPC-H(0.01) dataset over varying numbers of chunks.

T). Fig. 4 shows the number of tuples moved over different values of k for the TPC-H(0.01) dataset. The parameters p and q are set at $p{=}32$ and $q{\in} \{2, 3, 4, \ldots, 256\}$. Method T is not sensitive to the number of chunks at all, because once the tree of ranges is constructed, the tuple movements are fixed. Method C, on the other hand, moves less tuples with larger k, because more chunks translates to smaller chunk size and better isolation of the fragments that need to be split and moved.

IV. Related Work

Recent work on exploiting cloud computing platform for datamanagement has focused mainly on building systems and identifying challenges [9], [10], [11], [1], [5], [12]. Our work complements these effort in that the proposed elastic data partitioning techniques could be applied to cloud data management systems that use a partitioned, relational data model.

A closely related area of theoretical research is the problem of data migration in storage area networks (SAN) [13], [14], [15]. The data migration problem computes a migration plan for moving data objects over a network of storage devices from one configuration to another. Even if we conceptualize the tuples of our partitioned relation as "data objects" in the data migration problem, there are significant differences. First, our problem is more general and includes not just the data migration problem as a subproblem, but the subproblem of computing the next configuration given certain constraints. Second, previous theoretical formulation of the data migration problem has assumed that each device can only handle one object at one time. This is true in the context of SAN, but not true in the context of a parallel SQL processing system. Each node in the SQL processing system is capable of multi-threading and processing multiple objects at a time. Third, data repartitioning is the most expensive operation in our problem and not data movement across a network.

The chunking technique that we used to eliminate tuple repartitioning is similar in spirit to techniques used in text indexing. Glimpse [16] uses chunking to reduce the granularity of the pointers in an inverted index so that the pointers point to pages rather than to individual words in the text. Similarly, landmarks are used in [17], [18] to speed up update processing in inverted indexes.

Data partitioning is also closely related to histogram construction [8], [19]. Histograms partitions data into a set of buckets and approximate the number of tuples in each bucket with an average. We adopted some of the concepts and terminology for data partitioning from partitioning in histograms [19]. In contrast to histogram partitioning problem, this paper addresses the orthogonal problem of migrating one set of partitions to another. Our proposed Method T uses a tree of partitions which in the case of range partitioning type is similar to the hierarchical ranges used in [20] for selectivity estimation.

To the best of our knowledge, the problem of partitioning and organizing data in support of elasticity in a parallel DBMS has not been addressed before.

V. Conclusion

We have highlighted an interesting problem of supporting elastic data partitioning in cloud-based parallel databases in order to exploit the elasticity feature of a cloud computing platform. We proposed several methods to scale parallel databases from p nodes to q nodes. Method N does a naive and brute force matching of the existing p data partitions to the target q data partitions while preserving the partitioning constraint. Method C pre-partitions the data into fine-grain chunks and manages data partitions that are aligned and mapped to the chunks in order to avoid re-partitioning each data tuple. Method T improves on data movement by using a hierarchy of pre-computed partitions. Scaling from p to q must use a covering set of partitions from the tree. We have implemented our proposed methods in a prototype and have demonstrated its effectiveness and efficiency through extensive experimentation. As future work, we plan to investigate the issue of skewed data distributions as well as extending the current techniques to leverage replicated data fragments.

References

[1] D. Agrawal, S. Das, and A. E. Abbadi, “Big data and cloud computing: New wine or just new bottles?” *PVLDB*, vol. 3, no. 2, pp. 1647–1648, 2010.

[2] L. Wang, J. Tao, M. Kunze, A. Castellanos, D. Kramer, and W. Karl, “Scientific cloud computing: Early definition and experience,” in *High Performance Computing and Communications (HPCC)*. IEEE, sep. 2008, pp. 825 –830.

[3] S. Loebman, D. Nunley, Y.-C. Kwon, B. Howe, M. Balazinska, and J. Gardner, “Analyzing massive astrophysical datasets: Can pig/hadoop or a relational dbms help?” in *Cluster Computing and Workshops (CLUSTER)*. IEEE, aug. 2009, pp. 1–10.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “A view of cloud computing,” *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[5] S. Das, D. Agrawal, and A. E. Abbadi, “Elastras: An elastic transactional data store in the cloud,” *CoRR*, vol. abs/1008.3751, 2010.

[6] M. T. Özsu and P. Valduriez, *Principles of distributed database systems (2nd ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1999.

[7] D. J. DeWitt and J. Gray, “Parallel database systems: The future of high performance database systems,” *Commun. ACM*, vol. 35, no. 6, pp. 85–98, 1992.

[8] Y. Ioannidis, “The history of histograms (abridged),” in *VLDB*. VLDB Endowment, 2003, pp. 19–30.

[9] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, “HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads,” *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 922–933, 2009.

[10] A. Aboulnaga, Z. Wang, and Z. Y. Zhang, “Packing the most onto your cloud,” in *CloudDB ’09: Proceeding of the first international workshop on Cloud data management*. ACM, 2009, pp. 25–28.

[11] D. Agrawal, A. E. Abbadi, S. Antony, and S. Das, “Data management challenges in cloud computing infrastructures,” in *DNIS*, 2010, pp. 1–10.

[12] S. Das, D. Agrawal, and A. E. Abbadi, “G-store: a scalable data store for transactional multi key access in the cloud,” in *SoCC*, 2010, pp. 163–174.

[13] J. Hall, J. Hartline, A. R. Karlin, J. Saia, and J. Wilkes, “On algorithms for efficient data migration,” in *Proceedings of the 12th annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’01. Society for Industrial and Applied Mathematics, 2001, pp. 620–629.

[14] E. Anderson, J. Hall, J. Hartline, M. Hobbs, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes, “An experimental study of data migration algorithms,” in *Algorithm Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, vol. 2141, pp. 145–158.

[15] S. Khuller, Y.-A. Kim, and Y.-C. J. Wan, “Algorithms for data migration with cloning,” in *Proceedings of the 22th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS ’03. ACM, 2003, pp. 27–36.

[16] U. Manber and S. Wu, “Glimpse: A tool to search through entire file systems,” in *USENIX Winter*, 1994, pp. 23–32.

[17] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. C. Agarwal, “Dynamic maintenance of web indexes using landmarks,” in *WWW*, 2003, pp. 102–111.

[18] ——, “Efficient update of indexes for dynamically changing web documents,” *World Wide Web Journal*, vol. 10, no. 1, pp. 37–69, March 2007.

[19] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, “Improved histograms for selectivity estimation of range predicates,” *SIGMOD Rec.*, vol. 25, no. 2, pp. 294–305, 1996.

[20] N. Koudas, S. Muthukrishnan, and D. Srivastava, “Optimal histograms for hierarchical range queries (extended abstract),” in *PODS*. ACM, 2000, pp. 196–204.

APPENDIX

Algorithm 1 ASSIGNNODES($T, \mathcal{R}_p, \mathcal{R}_q$)

Input: T is the relation instance that is currently partitioned using $\mathcal{R}_p$ into p partitions, q is the new desired number of partitions, $\mathcal{R}_q$ is the set of ranges for the desired q partitions.
Output: An assignment of database nodes to each range in $\mathcal{R}_q$

```
1: Let $\mathcal{R}_p = \{r_1, \ldots, r_i, \ldots, r_m\}$
2: Let $\mathcal{R}_q = \{s_1, \ldots, s_j, \ldots, s_n\}$
3: Initialize all Done($r_i$) $\leftarrow false$
4: Initialize all $Node_q(s_j) \leftarrow \bot$
5: for all $(r_i, s_j) \in \mathcal{R}_p \times \mathcal{R}_q$ ranked in decreasing number of common tuples do
6:    if Done($r_i$)$=false \wedge Node_q(s_j)=\bot$ then
7:       $Node_q(s_j) \leftarrow Node_p(r_i)$
8:       Done($r_i$) $\leftarrow true$
9:    end if
10: end for
11: Assign new nodes to all unassigned $s_j \in \mathcal{R}_q$
```

Algorithm 2 NAIVERESIZERANGE($T, \mathcal{R}_p, \mathcal{R}_q$)

Input: T is the relation instance that is currently partitioned using $\mathcal{R}_p$ into p partitions, q is the new desired number of partitions, $\mathcal{R}_q$ is the set of ranges for the desired q partitions. Assume that node assignment for $\mathcal{R}_q$ has been computed.
Output: A migration path

```
1: Let $\mathcal{R}_p = \{r_1, \ldots, r_m\}$
2: Let $\mathcal{R}_q = \{s_1, \ldots, s_n\}$
3: for all $s_j \in \mathcal{R}_q$ do
4:    Let Contained$(s_j) = \{r_i : r_i \subseteq s_j\}$
5:    for all $r_i \in$ Contained$(s_j)$ do
6:       if $Node_p(r_i) \neq Node_q(s_j)$ then
7:          Move tuples associated with $r_i$ to $Node_q(s_j)$
8:       end if
9:    end for
10: end for
11: Let Straddle$(\mathcal{R}_q)$ = $\{r_i : \exists s_j, s_{j+1}, r_i \cap s_j \neq \emptyset, r_i \cap s_{j+1} \neq \emptyset\}$
12: for all $r_i \in$ Straddle$(\mathcal{R}_q)$ do
13:    Let $r_i$ straddle $s_j, s_{j+1}, \ldots, s_{j+l}$
14:    Repartition each tuple in $r_i$ onto the nodes for $s_j, s_{j+1}, \ldots, s_{j+l}$
15: end for
```

Algorithm 3 ELASTICRESIZERANGE($T, \mathcal{R}_k, \mathcal{R}_p, \mathcal{R}_q$)

Input: T is the relation instance that is currently partitioned using $\mathcal{R}_p$ into p partitions, q is the new desired number of partitions, $\mathcal{R}_q$ is the set of ranges for the desired q partitions. Assume that node assignment for $\mathcal{R}_q$ has been computed.
Output: A migration path

```
1: Let $\mathcal{R}_k = \{t_1, \ldots, t_l\}$
2: Let $\mathcal{R}_p = \{r_1, \ldots, r_m\}$
3: Let $\mathcal{R}_q = \{s_1, \ldots, s_n\}$
4: for all $s_j \in \mathcal{R}_q$ do
5:    Let Contained$(s_j) = \{r_i : r_i \subseteq s_j\}$
6:    for all $r_i \in$ Contained$(s_j)$ do
7:       if $Node_p(r_i) \neq Node_q(s_j)$ then
8:          Move chunks associated with $r_i$ to $Node_q(s_j)$
9:       end if
10:    end for
11: end for
12: Let Straddle$(\mathcal{R}_q)$ = $\{r_i : \exists s_j, s_{j+1}, r_i \cap s_j \neq \emptyset, r_i \cap s_{j+1} \neq \emptyset\}$
13: for all $r_i \in$ Straddle$(\mathcal{R}_q)$ do
14:    Let $r_i$ straddle $s_j, s_{j+1}, \ldots, s_{j+l}$
15:    for all chunks $t_u \subseteq r_i$ do
16:       Let $t_u \subseteq s_{j+z}$
17:       if Node($r_i$) $\neq Node_q(s_{j+z})$ then
18:          Move chunk $t_u$ to $Node_q(s_{j+z})$
19:       end if
20:    end for
21: end for
```

Algorithm 4 FINDTREECOVER($\mathcal{T}, \mathcal{R}_p, q$)

Input: The current ranges $\mathcal{R}_p$ for the p partitions, $\mathcal{T}$ is the binary tree of ranges, q is the new desired number of partitions. Assume $q < k$
Output: $\mathcal{R}_q$

```
1: Let $\mathcal{R}_p = \{r_1, \ldots, r_m\}$, each $r_i$ is a node in $\mathcal{T}$.
2: $\mathcal{R}_{cur} \leftarrow \mathcal{R}_p$.
3: while $q \neq |\mathcal{R}_{cur}|$ do
4:    if $q > |\mathcal{R}_{cur}|$ then
5:       Pick a node $r_i \in \mathcal{R}_{cur}$ to split
6:       Replace $r_i$ with Children$(r_i)$ in $\mathcal{R}_{cur}$
7:    else if $q < |\mathcal{R}_{cur}|$ then
8:       Pick two sibling nodes $r_i, r_j \in \mathcal{R}_{cur}$ to merge
9:       Replace $r_i, r_j$ with Parent$(r_i, r_j)$ in $\mathcal{R}_{cur}$
10:    end if
11: end while$\mathcal{R}_q \leftarrow \mathcal{R}_{cur}$
```