Industry and Applications Paper

Elastic scaling in VectorH

Industrial Paper

Steffen Kläbe
TU Ilmenau, Germany
steffen.klaebe@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

Stephan Baumann
Actian Germany GmbH
stephan.baumann@actian.com

Michael Rink
Actian Germany GmbH
michael.rink@actian.com

ABSTRACT

Cloud infrastructures allow to dynamically adapt to workload changes by provisioning additional resources or deprovisioning resources to reduce costs. This offers also opportunities for scalable distributed data management. However, elastic scaling in databases requires to migrate or even repartition data. In this work, we present an approach implemented in Actian's MPP solution VectorH that speeds up the elastic resizing process by minimizing partition reassignments while still achieving load balancing. Moreover, we describe a buffer matching and pre-filling technique to further increase performance after the resize step. The experimental evaluation shows that our solution significantly outperforms the non-elastic way of scaling using a system restart by a factor of 2 up to 4 and reduces downtimes during resizing to less than one minute.

1 INTRODUCTION

Cloud computing gained extraordinary importance over the past several years with the advent of enabling technologies like distributed systems, virtualization or fast network. While consumers moved their business applications to the cloud environment, important technology companies like Amazon, Google or Microsoft directed their focus towards cloud technologies, competing with each other for the leadership position in this promising new market. One of the decisive properties for the success of cloud computing is elasticity, providing users with flexibility to meet requirement changes in the fast moving technology world we live in. In the context of the cloud environment, elasticity describes a system's ability to adapt to changes in user demands and can be achieved in every system layer, e.g. storage, network or computing power. Therefore, developing software that is optimized for the cloud environment requires the software to support elastic changes in the underlying environment.

Actian Avalanche is the Software as a Service version of Actian VectorH [4], a massively parallel processing (MPP) solution for data analytics. The software is deployed in public cloud environments like Amazon web services (AWS) or Microsoft Azure, which both offer an elastic environment. As the VectorH MPP solution was originally designed to run on a static cluster, it does not exploit the opportunities provided by those environments. Furthermore, the currently implemented partition management is not designed to cope with non-static clusters and needs to be overhauled.

As the cloud environment offers elasticity, the main goal is to make VectorH able to utilize the provided elasticity of computing and storage resources. For this purpose, we focus on three major goals:

- **Develop an elastic resize feature:** Nodes need to be **efficiently** added or removed from a VectorH cluster during the system uptime, avoiding the drawbacks of a full restart. This is the key feature to enable elastic scaling.
- **Provide a partition strategy applicable for the cloud environment:** Partitioning is the key concept of VectorH to distribute work among nodes. As scaling the system becomes a frequent operation in the elastic environment, the partition strategy has to support elastic scaling by minimizing partition reassignments to nodes while providing load balancing.
- **Smoothen the performance after a cluster resize:** Although the partition strategy in intended to minimize partition reassignments, each cluster resize operation involves changes in data responsibilities. These changes should be transparent to the user by providing the full system performance once the scaling process finished.

The remainder of this paper is organized as follows. We give an overview over related work in Section 2, including a discussion of commercial products available on the market and how they provide elasticity. Section 3 provides an overview over the basic concepts of VectorH, before describing the design of the elastic scaling feature in Section 4. We compare different approaches for partition management and present the implemented approach in Section 5. Based on experiments, we describe the design of the buffer matching and filling mechanism in Section 6, which is an optimization on top of elastic scaling. Finally, we evaluate our solution in several experiments in Section 7, before concluding and giving an outlook in Section 8.

2 RELATED WORK

Achieving elasticity for cloud database systems is solved in different ways in the research community. ElasTraS [6] as an example is explicitly designed for the cloud environment and supports elasticity by the separation of storage and compute resources. The system distinguishes between high level transaction managers (HTM) and owning transaction managers (OTM). While HTMs handle user connections and execute queries using their local caches, OTMs are

responsible for the actual data access. Depending on the current load, both HTM and OTM can be scaled independently and tasks are distributed among all running managers. As a different approach, Albatross [7] uses virtualization and live migration of databases to provide elasticity. The system aims at multi-tenant databases, which often face the problem of efficiently migrating single tenants instead of the whole system. To accelerate migration performance Albatross creates a database snapshot, which is placed in a network storage, and only migrates caches and active transaction states to a newly provisioned system. Afterwards, the system is switched using an atomic handover operation. In addition to elasticity, the system ensures serializability and correctness in failure cases. The experiments show that this approach does not abort any active transactions, harms query latency only in a negligible way and makes the database unavailable only for a small time window of about 300ms. A similar concept is used by ShuttleDB [1]. In contrast to Albatross, ShuttleDB can be seen as a middleware to make an arbitrary database management system elastic. It uses virtualization techniques to be transparent from the actual database instance and therefore works with common database systems without changes in their engine. On top of the live migration concept for single tenants, ShuttleDB offers automated elasticity. The system monitors query latencies of database tenants, automatically chooses tenants to scale and migrates them to another instance after deciding for a suitable migration strategy.

While ElasTraS, Albatross and ShuttleDB are mainly evolved from a cloud provider point of view, meaning maximizing utilization while fulfilling user demands, the Kingfisher system presented in [14] deals with elasticity from a customer point of view. Exploiting cloud pricing models, Kingfisher dynamically provisions virtual server capacity while being cost-aware. Using monitoring and forecasting of the query workload in combination with solving a linear optimization problem, the system minimizes infrastructure costs (e.g. cores, servers) and transition costs (time and costs to change environment).

In the field of commercial systems, Snowflake [5] was build from scratch for the cloud environment. It uses so called micro-partitions of several MB size to automatically partition and cluster data. Based on that, work distribution among nodes is realized using consistent hashing. In combination with work stealing, this approach automatically handles node failures as well as elastic scaling, while also minimizing the reassignment of micro-partitions as a property of the consistent hashing algorithm. As a second example, Amazon Redshift [9] divides compute nodes into the abstract concept of slices and assigns data to these slices. For scaling, the system offers two possibilities. While the "classic resize" deploys a new cluster in the background and sets the system in a read-only mode for several hours, the "elastic resize" reduces the downtime by saving a snapshot to the cloud file system, adding/removing nodes and reassigning work by reshuffling the abstract slices between nodes. This way, the downtime for the scaling process is reduced to several minutes. Nevertheless, user queries are on hold during the scaling. Third, Googles BigQuery relies on the concept of overpartitioning to avoid repartitioning in the elastic environment. In the case of scaling, tasks in the Dremel execution engine [13] are resized and data is read again from the storage layer, trusting in the speed of Google's Jupiter network technology and the Colossus storage system.

3 VECTORH OVERVIEW

Actian VectorH [4] is the scale-out version of the Vectorwise/X100 system [3] running on Hadoop clusters. It offers high and scalable query performance by exploiting opportunities of modern CPUs (e.g. SIMD, caching) with its vectorized execution engine. As a key for parallel processing of data the system uses hash partitioning and exploits co-located foreign-key joins for efficient node-local join processing, while partitions are assigned to nodes using a round-robin assignment. Furthermore, it reduces I/O costs by advanced compression methods and data skipping.

The Hadoop distributed file system (HDFS) is used as the storage layer and provides fault tolerance and scalability. Although HDFS is append-only, VectorH offers efficient updates by using Positional Delta Trees [11]. In order to efficiently read data from HDFS, the system is aware of data locality and replication. Nevertheless, processing data that is already in memory is another key to high query performance. Therefore, VectorH allocates a configurable bufferpool on system startup and maintains buffered blocks over different buffer policies [15].

For data exchange among cluster nodes, VectorH uses the Message Passing Interface (MPI) for implementing exchange operators described in the Volcano model [8]. The MPI library offers point-to-point communication as well as collective communication and is based on the concept of groups. Nodes within a group are identified using a rank starting at 0 and they are able to communicate with each other using a so called intra-communicator related to this group. In addition to that, so called inter-communicators allow communication among groups.

In order to scale a VectorH installation, the cluster configuration has to be changed and the system has to perform a restart, which has two major drawbacks. During the start process the system replays the write-ahead log, which might be a long-running operation depending on the log size. In addition to that, allocated memory is freed during the system shutdown. As a result, buffers are empty after a restart and data needs to be read again from storage, which impacts the performance of the first queries. Therefore, we need a solution to scale a running MPI application without performing a full restart. In addition to that, we want to avoid a performance degradation after the scaling operation by adapting the buffer management using the buffer matching and filling mechanism presented in Section 6 and by being aware of this issue when assigning work to nodes in a scaled environment.

Scaling can be invoked by the user to achieve one of the following goals. Either performance should be increased while keeping the data size fixed, or the system should be enabled to handle larger data sizes (while keeping performance constant). While the first goal refers to Amdahl's law, the second one is the use case for Gustafson's law (both in [10]). For VectorH, work distribution among cluster nodes is realized using partitioning, where the optimal number of partitions per table is approximately the total number of parallel threads the cluster offers. Assuming a

homogeneous cluster, this is equal to the number of nodes multiplied with the number of physical cores per node. In order to support both mentioned scaling cases while avoiding an expensive repartitioning, we use the concept of overpartitioning, initially splitting tables into more partitions than necessary for the initial cluster configuration. The chosen number of partitions is crucial in terms of performance and the ability to scale the cluster size. Figure 1 shows the runtime of the TPC-H [2] query set on scale factor 1000 GB as a function of the number of partitions. The experiments were made on a cluster of 4 to 6 nodes with 24 cores each, providing a parallelism of 96 to 144 threads. First, the results show that increasing the cluster size while keeping the data size constant can lead to a performance benefit. Second, one can observe that increasing the number of partitions above the optimal partitioning of one partition per thread comes with an increasing performance penalty, which is caused by the introduction of an Union operator on top of table scans. Furthermore, increasing the number of partitions heavily impacts update performance, as update operations have to be performed on more fine-grained partitions. As a consequence, the number of partitions should not be chosen too large. Third, one can observe that VectorH is able to handle underpartitioning to a certain degree by performing node-local splits during table scans, achieving the assignment of one partition per thread. However, this resplitting harms node-local join processing and is therefore not a desired behaviour.

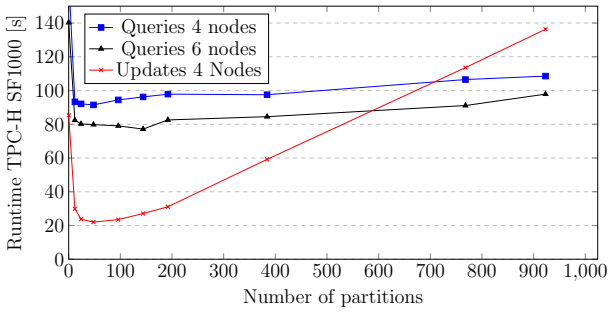

Figure 1: Dependency of Partitioning on TPC-H SF1000 runtime

4 ELASTIC SCALING

This section presents the design and implementation a dynamic cluster resize functionality for VectorH. With this feature we want to face the problem of scaling VectorH and solve the first goal stated in Section 1. Two new functions, *add_nodes* and *remove_nodes*, were implemented using the VectorH syscall functionality, allowing to issue commands against the system without forming SQL queries.

The new cluster resize functionality exploits the opportunities of the MPI group and communicator management. The basic approach of adding nodes using MPI routines is illustrated in Figure 2. Starting from an existing group of current nodes with an intra-communicator (A), a new group of processes is spawned by starting the application on the newly provisioned cluster nodes using the MPI_COMM_SPAWN routine. All nodes of the new process group are able to communicate with each other using their own intra-communicator (B) and with the old group of nodes using the inter-communicator (C) returned by the MPI routine. In order to abstract from these different types of communication in a second step, both groups form a new intra-communicator (D) using the MPI_INTERCOMM_MERGE routine and replace group communicators (A), (B) and (C). After new nodes are added in the *add_nodes* call, the master node has to broadcast the current partition mappings as described in Section 5 in order to provide the new nodes with the correct initial partition assignments. Afterwards the current nodes have to follow the new nodes' startup, as there are various synchronization points within the startup procedure. To simulate a collective start of all servers, the current servers have to perform all of these synchronizations to make the added servers finish their initialization, before performing the buffer matching mechanism described in Section 6. The cluster resize functionality has to be compatible with the following optimization made in VectorH. In order to reduce memory consumption, the creation of storage objects and minmax indexes is skipped for partitions a node is not responsible for. After adding nodes, this responsibility assignment changes as some partitions of the current nodes get assigned to the newly added ones. As a result, current nodes hold structures for partitions they are not responsible for anymore. Each node checks for these kind of unused structures by iterating over all tables in their catalogs, dropping information and freeing memory whenever possible. This could also be done in a lazy way by checking for unused information within a partition responsibility check during query execution. But as these checks are very frequent operations and are called multiple times within each query, the decision was made to cleanup the unused structures directly as part of the *add_nodes* operation.

Removing nodes reverses the presented mechanism. In the first step the nodes are divided into two distinct groups S and R with corresponding intra-communicators using a MPI_COMM_SPLIT routine. While nodes of group S perform a collective shutdown as a second step, the nodes of group R form the new cluster. Passing a list of hostnames to the function call, each node checks whether it is included in S and should terminate or not. It is currently not allowed to terminate the master node, so *remove_nodes* returns an error in this case. The remaining nodes (group R) now update their partition mappings as described in Section 5. Removing nodes assigns more partitions to the remaining nodes, and, as the nodes were not responsible for these partitions before, the creation of storage structures and minmax indexes was skipped. In order to reconstruct the missing information, an adapted replay of the write ahead log is performed. Within this replay, all log actions are skipped except those related to storage and minmax indexes for tables the node is now responsible for and was not responsible for before. This information is provided by the partition manager.

With the design of the elastic cluster resize feature we achieved the possibility to scale the VectorH cluster without restarting the existing nodes. We therefore ensured that the scaled system state is similar to the state before the scaling in terms of communication, metadata and catalog state. Furthermore, we automatically achieve support for updates that are resident in in-memory PDT structures

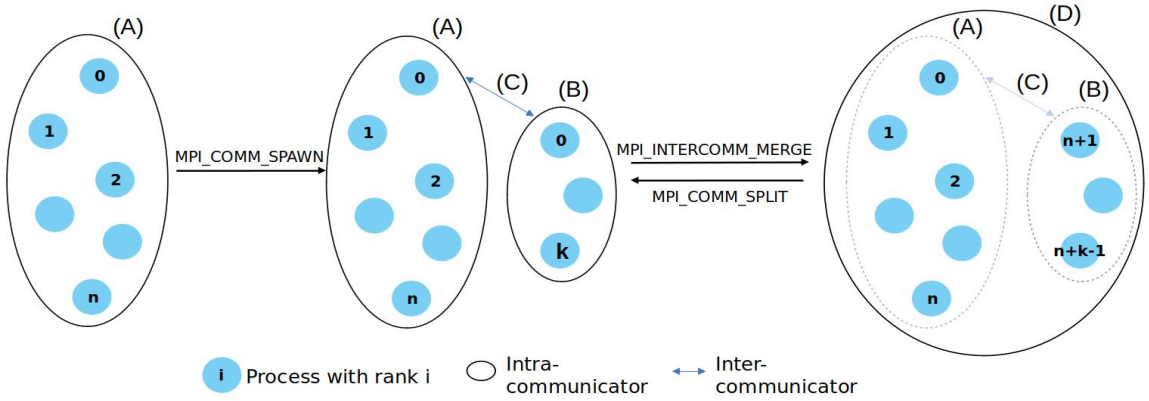

Figure 2: Schematic overview of cluster resize

mentioned in Section 3, as the log replay that is included in both *add_nodes* and *remove_nodes* also covers PDT log actions. As a result, the process reconstructs the update information needed to ensure data consistency.

Our implementation relies on the following assumption. For the *add_nodes* functionality binaries, configuration files and user data are accessible from the new nodes in the same way (especially using the same directory paths) as for the already running nodes. Triggering the operation over a front-end, e.g., a web-based management tool, would invoke starting the new nodes (dependent on the cloud service provider) and synchronizing necessary data before the *add_nodes* operation is started. A functionality for synchronizing the VectorH installation directory already exists and is used for cluster deployment, so this assumption is no restriction.

Besides assumptions the presented implementation comes with some limitations. First, the resize functionality must be issued in a separate session with no concurrent sessions. The system is able to block/hold incoming session requests when a scaling request is made. Second, the hosts to be removed are restricted to be the ones with the highest ranks. As all nodes except the master node are treated equally and the user does not call this function directly but through a frontend only providing the cluster size he wants to reach, the frontend can choose the nodes to remove as the ones that where added most recently. This ensures that only the highest ranks are selected. In case this is considered as too restrictive, one could extend the algorithm such that it first rearranges the nodes before the resize operation and assigns ranks in a way that fulfills the restriction. Third, it is currently not allowed to remove the master node as worker nodes are not able to replace a missing master node (which also holds for master node failure).

5 PARTITION MANAGEMENT

In this section, we present a partition management approach that is suitable for the elastic cloud environment, which corresponds to the second goal stated in Section 1.

5.1 Partition assignment approaches

We start by comparing basic approaches for partition assignment. The comparison is based on the following requirements, prioritized from most important to least important:

(1) **Load balancing:** Assigning an equal number of partitions to each node is crucial to achieve an optimal query performance. As partitions are already built using the partitioning method, the assignment strategy can only affect the number of partitions per node, not the size of each partition.
(2) **Lookup time:** The mapping *partition* $\rightarrow$ *node* is evaluated numerous times within each query and should therefore be an efficient operation.
(3) **Update time:** As resizing the cluster changes partition assignments, the structures of the partition assignment should be updatable in an efficient way.

Besides these main objectives, the partition assignment strategies must fulfill the following side conditions for performance reasons:

- Keep co-locality of foreign-key related tables
- Minimize the number of reassigned partitions on cluster resize

Keeping the co-locality of related tables is a key for achieving optimal query performance by exploiting node-local joins and is therefore an important demand. Reassigning partitions has several effects and should therefore be minimized. First, it leads to storage access for reading the partition, as the data is not present in the node's buffer. Second, nodes have to update their catalog information when becoming responsible or loosing the responsibility for a new partition, as described in Section 4.

We can state a lower bound for the minimum number of partitions that have to be reassigned on cluster resize. Let d be the total number of partitions for an arbitrary table and we assume that partition assignment is balanced before a resize operation. When adding n nodes to an existing cluster of n_{old} nodes with $n_{new} = n_{old} + n$, it is clear that every node has to be responsible for $\frac{d}{n_{new}}$ partitions after resizing to achieve load balancing. We assume that n_{new} is a divider of d and if not, every node gets one additional partition until the remaining partitions are assigned. As every new node get's $\frac{d}{n_{new}}$ partitions, the minimum total number of reassigned partitions is $\frac{d}{n_{new}} \cdot n$. For removing n nodes from an existing cluster of n_{old} nodes it can be easily seen that $\frac{d}{n_{old}} \cdot n$ partitions have to be reassigned, as every node was responsible for $\frac{d}{n_{old}}$ partitions before

resizing. Overall we can state the lower bound of

$$reassigned\_partitions \geq \frac{d}{n_{max}} \cdot n$$

with $n_{max} = max\{n_{new}, n_{old}\}$ for adding/removing n nodes.

Round-robin assignment: Round-robin assignment is the currently used assignment strategy in VectorH. It is clear that this strategy provides load balancing and as it can be evaluated using an arithmetic operation, has a very fast lookup time while not needing additional data structures. Nevertheless, with respect to the additional cluster resize functionality, round-robin is one of the worst choices as it reassigns nearly all partitions when not resizing the cluster with a factor that is a power of two, as shown in Figure 3. Doubling the number of nodes leads to a reassignment of half of the partitions, which is basically the minimum for achieving load balancing. But increasing the number of nodes by a factor being an arbitrary number and not being a power of two leads to a reassignment of nearly all partitions. Therefore, this strategy is not applicable for the cloud environment.

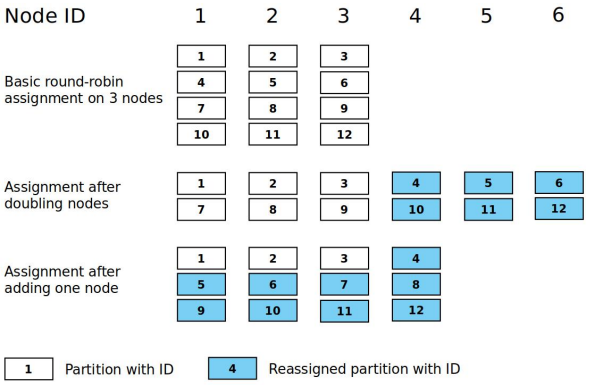

Figure 3: Round-robin partition assignment on cluster resize

Consistent hashing: The requirement of minimizing the number of reassignments on cluster resize directly leads to the method of consistent hashing, which places elements and buckets on a logical ring representing the set of hash values produced by the hash function. In order to improve load balancing, buckets can be replicated on the logical ring. It is shown that removing/adding one hash bucket leads to a reassignment of $\frac{k}{n}$ keys, with k being the total number of keys and n being the number of hash buckets [12]. With keys being partitions and hash buckets being nodes, this is the lower bound of reassigned partitions we stated before. If a node is removed, only the partitions assigned to the removed node have to be reassigned and adding a node leads to a reassignment of all partitions between the added node and its last predecessor on the logical ring. Consistent hashing can be implemented by holding a sorted array or list of nodes, so a lookup operation to find a node responsible for a partition takes time $O(\log(n \cdot r))$ with r being the replication factor and using binary search on that list. Updating the number of nodes leads to resorting the list with a complexity of $O(n \cdot \log(n))$, for example using insertion sort for adding a few nodes or merge sort for adding a sorted list of nodes. Holding the list consumes memory in the size of $O(n \cdot r)$, which is independent from the number of partitions.

Explicitly storing and maintaining the assignment mapping: This approach tries to provide a minimal lookup time and a best possible load balancing by explicitly maintaining and storing the mapping $[d] \rightarrow [n]$ from the set of partitions to the set of nodes for each possible partitioning using a partition manager structure. The mapping can be stored as an array with the size d, providing lookup time $O(1)$. Tables with equal number of partitions d_i build an equivalence class $i \in Eq$, so as a side effect, this guarantees co-location of foreign-key related tables when assuming them to have equal partition numbers (otherwise node-local joins would not be possible anyway). Maintaining the mapping explicitly ensures that the best possible load balancing is achieved. As a drawback, this approach has a quite high memory consumption of $O(\sum_{i \in Eq} d_i)$, which is especially not independent from the number of partitions and increases with the number of distinct numbers of partitions. Nevertheless, the number of different partitionings and therefore the number of equivalent classes is typically small in user scenarios.

Comparison: Table 1 compares the approaches of consistent hashing and partition manager. The partition manager approach outperforms consistent hashing in the most important categories load balancing and lookup time, while also minimizing partition reassignment. Assuming that the number of different partitionings is quite small and hence the number of equivalence classes is small, the time for updating the structure and the memory consumption is justifiable. As an example, a database consisting of 1000 tables sharing the same partitioning schema of 1000 partitions would lead to a memory consumption of around 5KB for 1000 4-byte-integer values and 1000 boolean values regarding the partition manager design shown in Section 5.2. Even for 1000 different partitioning schemas with a maximum of 2000 partitions each we would get a few MB of memory consumption. Therefore, the decision has been made towards the partition manager approach.

5.2 Partition manager design

The partition manager structure, illustrated in Figure 4 maintains *partition mapping* objects for each equivalent class, which consist of the number of *partitions*, a *mapping* array and an *is_moved* array, both of the size of the specific number of partitions. Each position i in *mapping* holds the node ID of the node responsible for partition i. The mappings are adapted during each cluster resize operation to maintain load balancing. In addition to that, the boolean *is_moved* value at position i indicates, whether the partition was moved during the last cluster resize operation, which is important to determine partitions to replay from the log when removing nodes or to delete the storage objects from when adding nodes. On top of the *partition mapping* objects, the partition manager maintains a hash table of *partition mapping* pointers to efficiently find the mapping for a given number of partitions.

For implementing a partition assignment, two assumptions are stated. First, it is assumed that co-local partitions have the same partition ID. This assumption is fulfilled by the hash partitioning method, as tuples with same keys

	Consistent hashing	Partition manager
Load balancing	Good, not guaranteed	Best possible
Partition reassignment	Minimized	Minimized
Lookup time	$O(\log(nodes \cdot replication))$	$O(1)$
Update	Re-sort array	Adapt every mapping $O(equi\_classes \cdot partitions)$
Memory consumption	$O(nodes \cdot replication)$	$O(equi\_classes \cdot partitions)$

Table 1: Comparison of partition assignment strategies

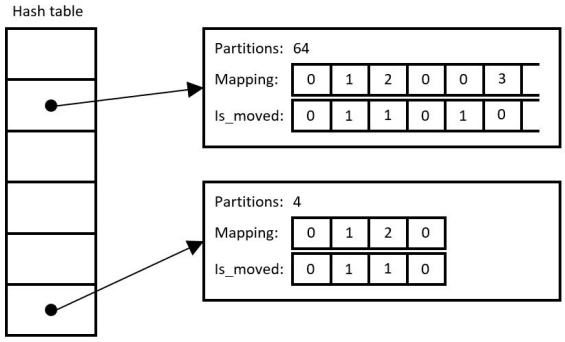

Figure 4: Partition manager overview

(either primary or foreign keys) are mapped to the same hash value, which is used as partition ID. Second, it is assumed that tables with a foreign key relationship have the same number of partitions specified. Having an unequal number of partitions while using the currently implemented hash partitioning violates the requirement of co-locality. Especially, it is not ensured by the current hash function that having a table T with k times the number of partitions than it's join partner S results in a partitioning that maps one partition of S to exactly k partitions of table T.

Initialization: During server startup, the partition manager structure is initialized to a global variable by creating an empty hash table. When the partition manager gets queried for a partition mapping that is not present in its hash table, a partition mapping object for the queried number of partitions is created and inserted into the hash table using the number of partitions d as key. The *mapping* is initialized using a round-robin strategy, so for partition ID i we get $mapping[i] = i \bmod d$. The choice of this initial strategy is arbitrary and could be replaced by any other strategy that provides a balanced assignment for n nodes. The *is_moved* array is initialized with *FALSE* at every position. The described process has time complexity $O(d)$ to initialize a single partition mapping.

Lookup: Knowing the structure of the partition manager, the lookup implementation is straight forward by a single hash table access and a single array access. Due to the design, the lookup operation has complexity $O(1)$.

Update: Whenever adding or removing nodes, all partition mappings have to be adapted. Therefore, we iterate over all entries in the partition manager's hash table and adapt every mapping using an algorithm divided into the following steps:

(1) Compute the optimal load balancing for the new cluster state by computing *partitions_per_node* and a *remainder* if the number of nodes is not a divider of the number of partitions.
(2) Compute a *diffs* array, with *diffs*$[i]$ indicating wether node i has to get additional partitions (positive entry) or get partitions removed (negative entry) to achieve the computed load balancing. The sum over all entries in the *diffs* array is 0, as the total number of partitions does not change.
(3) Adapt the actual partition mapping by iterating over the *mapping* array. If we find a partition whose responsible node has a negative *diffs* entry, we move the partition to a node with a positive *diffs* entry.

With step three being the dominant step in terms of runtime, the algorithm runs in $O(d)$. As we adapt the mapping of every equivalence class, the update operation has a total runtime of $O(\sum_{i \in Eq} d_i)$.

Exchange: After new nodes are added to the system, they check during startup whether they are added to an existing cluster and if so, they prepare to receive the current partition assignments. The current nodes trigger this exchange functionality within the execution of the *add_nodes* query. The exchange operation is implemented using MPI_BCAST (broadcast), so it has to be performed collectively. After broadcasting the number of mappings, the number of partitions and the *mapping* array are broadcasted for each partition mapping. This way it is ensured that all nodes call the broadcast the same number of times.

As a result, the chosen approach minimizes the reassignment of partitions in the case of scaling while providing load balancing and efficient lookup and update functions.

6 BUFFER MATCHING AND FILLING

We now motivate the problem of decreased performance after a cluster resize operation and present the design and the implementation of the buffer matching and filling mechanism, solving the performance problem and therefore being a solution to the third goal stated in Section 1.

For a brief intermediate evaluation of the cluster resize functionality and the partition manager described in Sections 4 and 5, the TPC-H benchmark was run on scale factor 300 using 8 of the 16 cluster nodes described in Section 7. Afterwards, the remaining 8 nodes were added and the benchmark was run again. Overall, it was observed that queries are slower after adding nodes than before. As an example, we pick query 1 of the benchmark, which is a selection and aggregation query on the lineitem table. The runtimes are shown in Figure 5. The query was run on 8 nodes with filled buffers (run 1) before adding 8 additional nodes and running the query several times again (runs 2,

3, and 4). Doubling the number of resources, we would expect a speedup up to factor 2, but the results clearly show an increase in runtime for the first run after the resize operation. However, the performance increases as expected with more consecutive query runs.

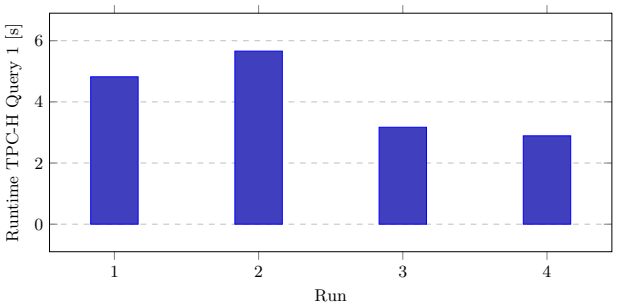

Figure 5: Runtime of TPC-H SF300 Query 1 in consecutive runs with run 1 on 8 nodes and runs 2,3 and 4 on 16 nodes after a cluster resize

From a user perspective, this behavior is not satisfactory, as he pays for extra resources without getting any immediate benefit. The observed performance is a direct consequence of the buffer management. Figure 6 shows the qualitative performance (runtime and output cardinality) of the scan operation of TPC-H query 1 for all threads on all nodes. The right half of the plot, which covers the 8 nodes added by the cluster resize, shows a runtime that is up to 5 times slower with respect to the left half of the plot, which covers the 8 nodes that were already running before the cluster resize. This is caused by filled buffers of the old nodes, while the added nodes start with empty buffers. In order to smoothen this runtime, we present the buffer matching mechanism to fill the buffers of added nodes during the cluster resize step. Doing so, we move the overhead of filling the buffer from user queries to the cluster resize operation, which is more justifiable to the user, as the elapsed time between the user toggling a cluster resize until finishing the resize also involves demanding new cluster nodes from the cloud service provider, which is also a potentially long-running operation and dependent on the actual provider. General approaches for buffer pre-filling must answer the following questions:

- Which data should be chosen to fill into the buffers?
- How is data brought to the buffers?

Especially the second question is important for the cloud setup. As cloud storage systems offer lower bandwidth and higher latencies than node-local storages, sending buffered data between nodes is also worth considering next to reading data from storage.

The chosen solution for this problem is our buffer matching mechanism. In order to discuss the mechanism from an abstract point of view, we identify a sender side and a receiver side, dividing the set of nodes into two distinct sets. Nodes of the sender side are characterized by losing the responsibility of partitions and having blocks in their buffer they are not responsible for anymore, while nodes of the receiver side become responsible for new partitions and do not have any buffered data for them. Note that because of our partition manager design in Section 5, a node is either a sender or a receiver. During data exchange, each node of the sender side can possibly have a connection to each node of the receiver side.

Block selection: The first step of buffer matching is to identify for each node the set of blocks that needs to be sent to other nodes, as well as each block's specific destination. First we get a list of all blocks currently resident in the buffer memory sorted by importance. The importance of a block is determined by the actually used buffer replacement policy. In addition to that, we identify the destination of the blocks by querying the partition manager described in Section 5 to get the responsible node. Blocks that belong to a partition for which the node remains responsible are not sent and therefore dropped from the list. All other blocks are appended to a list of blocks per receiver, so as the result of the block selection step, each sender node holds a (potentially empty) list of blocks for each receiver node. One special case needs to be handled. Due to data distribution or due to buffering blocks of only a few partitions caused by selection predicates, it might occur that receiver nodes are intended to receive more blocks than they can actually fit into their buffers. The calculated cardinality difference is balanced between all senders to this receiver node and each sender is informed about the number of blocks to send before starting to send data. As the block lists are sorted by importance, the sender just drops the end of the list in this case.

Data exchange: We now want to answer the question how buffer data is brought to nodes. As reading data from cloud storage might be slow compared to usual local disks or network transfer, the decision was made towards explicitly sending data to nodes over the network. The implemented data exchange mechanism follows three basic steps:

(1) Exchange the number of blocks to transfer between each sender and receiver node.
(2) Exchange block metadata.
(3) Exchange buffer content.

The first step is important to establish synchronization between senders and receivers. Each node of the receiver side has to know about the number of blocks to receive from each node of the sender side. After the block selection step, each sender node holds a list of blocks per receiver. The length of these lists is shared with the respective receivers using MPI_GATHER routines, called within a loop over all added nodes. A receiver node with rank i becomes the receiver of the MPI_GATHER call in exactly one loop iteration. In this round, all other nodes send the length of their list i, indicating the number of blocks to send to node rank i. As a result, node i has the complete information about the number of blocks to receive after success of loop iteration i. In the second step, we transfer the blocks metadata (e.g., used bytes, columnID or the ID of the commit creating the block) to the receiver nodes. It is important to note that metadata and actual data of a block cannot be transferred all-in-one using a single MPI call by default, as metadata and actual data are not placed in consecutive memory areas due to the VectorH buffer management, which preallocates the whole buffer memory during startup. Constructing an additional structure holding both metadata and buffer data would lead to copying major parts of the buffer, which is not desirable. Sending

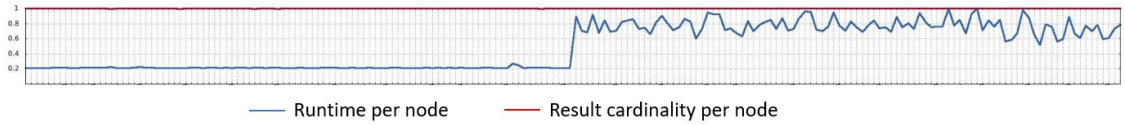

Figure 6: Qualitative TPC-H scan performance of query 1 for run 2

the metadata is important for two reasons. First, the respective block can be searched in the receiver's catalog. VectorH uses a replicated catalog, so a each block is already created on the receiver side. Second, the metadata contain information about the buffer content, like the actual data size or a flag to indicate wether data has been changed. This information is created during data load, so it needs to be sent as we do not load data from storage. Sending the metadata arrays is realized using non-blocking MPI point-to-point communication. This eliminates the need for explicit synchronization in this step. After receiving the metadata, each receiver performs catalog lookups to get pointers to the block structures and demands memory in the buffer memory for each block, before the blocks are inserted into the buffer replacement policy. In the third step, we transfer the actual buffer data. As the data for blocks is not placed in consecutive memory areas, they have to be sent one-by-one. Using non-blocking MPI communication like in the second step would therefore lead to k communication calls per sender and receiver with k being the number of blocks to transfer between sender and receiver, making it difficult to handle for the MPI environment when scaling the problem up. Therefore, we use synchronous, blocking communication for this step. To avoid deadlocks and reduce waiting time, we handle the communication in a multi-threaded way. For each point-to-point connection between a sender and a receiver node, having a non-zero number of blocks to transfer, both sender and receiver node open a separate thread running their side of the communication, resulting in a communication network. Each thread blocks until the respective counterpart of the communication is called. Upon receiving buffer data for one block, the data is copied to the buffer memory of the receiver node.

Fault tolerance: The buffer matching mechanism is an addition to the cluster resize functionality. The success of the buffer matching step is not indispensable for the success of the whole cluster resize operation, but should not lead to a undefined state on the occurrence of errors. Therefore, the mechanism is designed to be fault tolerant. We distinguish between different times of error occurrence. If an error is detected before the block metadata in the catalog of a receiver is changed, we can simply perform a collective abort, as no durable changes were done yet. This is realized using a synchronization point between the second and the third step. If an error occurs during data exchange within third step, we have to ensure that the system handles the block's buffer content in the right way. After receiving a single block's data and copying it to the buffer memory, we verify the correctness of the data using an already existing magic number in the block's data. This magic number is a fixed constant which is used to discover transmission failures. This mechanism could be further improved using a checksum. If the verification succeeds, the block is flagged to be in memory. If the verification fails or an error occurs during communication, the current block is flagged as "LOAD", leading the system to not use the buffer content before an IO-thread loads the data from storage (and adjusts the metadata again). Furthermore, all pending blocks that have not been transferred yet are also flagged as "LOAD". All other communication threads are not affected and may succeed.

Integration: The described mechanism is integrated into the *add_nodes* and the *remove_nodes* call. For adding nodes, the sender side is formed by the current nodes, as they lose responsibilities for partitions and may have buffered data for them, while all newly added nodes form the receiver side. During the system startup of the added nodes, buffer matching is integrated after the partition mapping exchange and the log replay, but before the server is able to handle user connections. This way the server already has the full catalog information. The current nodes perform buffer matching after following the server startup communication of the new nodes. For scaling the cluster size down, the removed nodes form the sender side of the buffer matching mechanism, while the remaining nodes form the receiver side. In order to enable removed nodes to determine the target of their buffered blocks, they update their partition mappings according to Section 5. Afterwards, they perform buffer matching while the remaining nodes perform it during execution of the *remove_nodes* call. In order to isolate the buffer matching communication from all other communication, a separate MPI communicator is build, being only valid during the buffer matching step. Finishing the buffer matching step, this communicator is destroyed. To provide the user the possibility to toggle the buffer matching mechanism on/off, an additional parameter is introduced into the VectorH configuration API.

Optimizations: After describing the basic ideas behind the buffer matching mechanism, we want to introduce two additional optimizations to the concept: the deletion of unused blocks at the sender side and the use of an alternative data exchange implementation. The strategies of the buffer policies are designed to keep the most important elements in the buffer by using priority queues. After performing buffer matching, blocks a sender node is not responsible for anymore may remain in its buffer queues. Due to the behavior of the strategies, these blocks are displaced at some time in the future. Nevertheless, a block may remain a long time in the queues once it is categorized as very important, depending on the actual displacement strategy. As a consequence, this buffer page is useless for a long time, blocking possibly important blocks from finding their way into the buffer. Therefore, we explicitly drop sent blocks from the buffer replacement policy on the sender side. The third step of the buffer matching mechanism uses blocking MPI calls to send the block's data one-by-one, as the buffered data of multiple blocks are not placed in

consecutive memory areas (in this case, one call pointing to the start of the memory area would suffice). The MPI environment can be configured to use several communication protocols and uses the Transmission Control Protocol (TCP) in the VectorH integration. Therefore, each call to send/receive a data block invokes communication setup, as well as the common TCP slow start phase, which is unnecessary overhead. As an optimization, we introduce a second, socket-based data exchange implementation. Similar to the described mechanism in the third step of the data exchange step, each sender-receiver pair with non-zero number of blocks to be transferred opens a thread on each side. Instead of starting MPI communication, the nodes establish a TCP stream socket connection. The receiver node creates a socket, sends the socket address information to the sender node using MPI and listens for an incoming connection. The sender node connects to the socket and sends data over the socket. As this single connection keeps alive until all data is sent, the overhead of communication establishment and slow start phase is reduced compared to the MPI implementation. In order to provide the same level of fault tolerance, each side of the socket checks the socket status using *select* before sending/receiving data. Furthermore, data blocks can be send in chunks, and only after a full block is received, the receiver verifies the block. Similarly to the fault tolerant behavior, blocks are flagged on connection or communication errors.

7 EVALUATION

During the evaluation, we want to prove the superiority of the implemented cluster resize feature over the inelastic way of scaling. Furthermore, we want to show that the usage of the buffer matching and filling mechanism improves query performance after a cluster resize operation and is, therefore, a useful extension. Due to expenditure reasons, the evaluation was done on a private cluster of 16 nodes, each with the following configuration:

- AMD Opteron Processor 3380 @2600MHz with 4 modules of 2 cores each
- 32 GB DDR3 RAM
- 3.5 TB disk space, distributed among 4 HDDs
- CentOS-7 64 Bit

The nodes are connected over a 1GBit/s ethernet connection and Hadoop 2.7.1 is installed on the cluster. Comparing the hardware to resources available on Amazon Web Services (AWS), this setup should be slower than all available EC2 instances. Therefore, the measured runtimes in the experiments can be seen as an upper bound and we expect our implementation to perform better on any AWS clusters with equal number of nodes. In addition, the TPC-H benchmark [2] on scale factor 1000 GB provided test data and test queries. This benchmark covers a well-understood synthetical workload in order to evaluate and compare data warehouse solutions with a dataset inspired by real world applications. The large tables of the benchmark are partitioned into 192 partitions, which is an overpartitioning for the cluster of 16 nodes with 8 cores each.

Scaling performance: The first experiment evaluates the performance of the implemented cluster resize feature. For the investigation of the upscaling process we start with a cluster of 4 nodes with filled buffers and vary the number of added nodes, while we start with 16 nodes and vary the number of removed nodes for the downscaling process. For these experiments, we keep the size of the bufferpool at 10 GB and the block size at 1 MB. Figure 7 illustrates the results of the experiment. One can observe that the runtime for adding nodes without using buffer matching increases in a linear way with the number of added nodes. The main reason for this behavior is the collective startup of the nodes. Starting more nodes at the same time increases the impact of the various synchronization points within the startup process. The buffer matching mechanism adds a nearly constant overhead to the measured *add_nodes* runtime. In a more detailed consideration one can observe that the buffer matching mechanism shows its minimum runtime when adding 8 nodes. Adding more nodes also increases the buffer matching data exchange parallelism (the number of receiver nodes per sender node), so the minimum runtime is expected to be at the number of physical cores, which is 8 in the used hardware setup. The downscaling runtime shows a slight increase when removing more nodes, which is caused by the adapted log replay the remaining nodes have to perform. Within this step, removing more nodes leads to more log entries that have to be replayed in the remaining nodes. The buffer matching mechanism was not applied for the downscaling process, as buffers where totally filled before scaling, so there was no buffer space left in the remaining nodes to receive blocks from removed nodes. Overall we can state that downscaling takes significantly less time than upscaling, because the synchronization effort for downscaling is lower. Once the nodes are split into two groups within the *remove_nodes* process, the group of removed nodes can simply perform a shutdown.

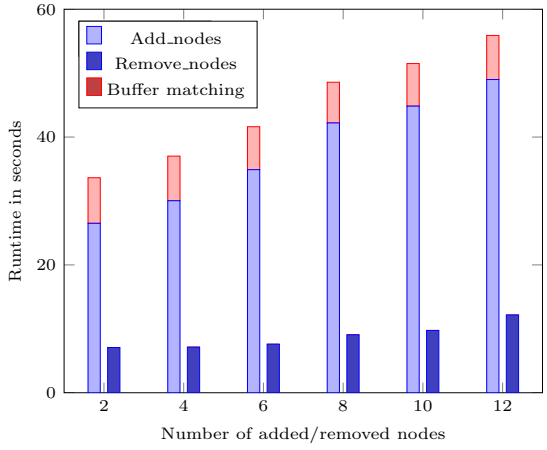

Figure 7: Scaling performance for adding and removing nodes

Scaled query performance: This experiment investigates the impact of the cluster resize operations on query performance. For this we repeatedly run a query while changing the cluster size between runs. As the main impact is expected to be within the scan operators, we use a query scanning two columns of the lineitem table. In order to fit the data into the buffer of the smallest cluster configuration, we limit the number of tuples to 500 million using a selection predicate, while setting the buffer

memory to 20 GB per node. This way we avoid I/O access that would create an unintended bias in the measurements. Moreover we minimize the network traffic by applying an aggregation on each column, which is executed locally on each partition and results in a single tuple that needs to be send to the master node. Figures 8 and 9 show the measured query runtimes. Regarding the case of upscaling, we can observe that adding nodes accelerates the query runtime as expected. However, the behavior varies between different buffer matching configurations. With activated buffer matching, query runtime drops and stays on the same level for the respective query runs after cluster resize, because the buffers already contain the needed data. On the contrary, not using buffer matching leads to significantly slower queries, especially for the first run after a cluster resize. The reason for this behavior is that added nodes have to read data from storage. In the following runs the query performance improves as buffers of added nodes fill. For the case of downscaling, we have to distinguish between two use cases. On the one hand, the reason a user triggers a downscale operation can be that the system is in an overprovisioning state, so the system underutilizes the provided hardware. In this case, removing server capacity should not have an impact on the systems performance. For VectorH we neglect this case as we aim to have more partitions than nodes at every point in time. If this condition is violated, a repartition operation is triggered by the system. On the other hand, the user could invoke the downscaling to save costs while accepting slower system performance, e.g., when the expected load becomes less during specific times of a day. This case is expressed by Figure 9. When scaling down the cluster, we can observe a performance degradation, again varying between buffer matching configurations. Similar to the upscaling case, the runtime of the first query run after cluster resize is significantly slower when not using buffer matching, as remaining nodes become responsible for data of removed nodes and have to read it from storage. With activated buffer matching, data is sent to the remaining nodes, leading to an immediately fast runtime after resize. Overall, this experiment proves that using the buffer matching mechanism during cluster resize perceptibly increases query performance after resizing.

Buffer matching performance: In this experiment, we want to evaluate the buffer matching performance for both implemented data exchange mechanisms, using MPI and using data streams over sockets. The two main parameters that have an impact on the buffer matching performance are buffer size and block size. While the buffer size affects the amount of data that is shipped during the buffer matching data exchange, the block size affects the granularity of shipped blocks and as a result also the communication overhead. Trying to touch as much data as possible, we use query 9 of the TPC-H benchmark for this experiment, as it touches five of the 8 tables in the benchmark and scans about seven billion tuples for the used scale factor of 1000 GB.

The plot in Figure 10 shows the runtime of the buffer matching mechanism as a function of the buffer size per node for both data exchange implementations, using a constant block size of 1 MB. For these measurements, we used the up-scale step from 8 to 16 nodes. First of all, the results show that the data exchange takes the major part (about

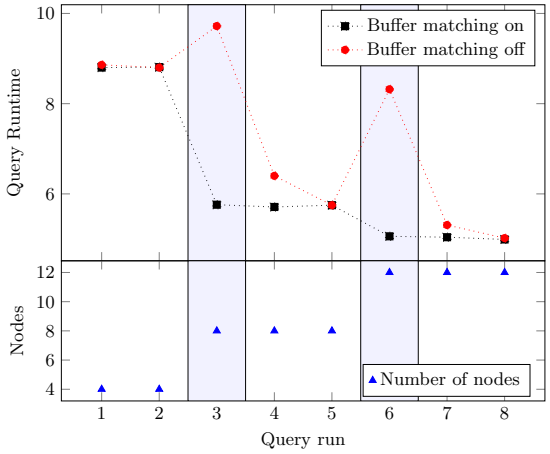

Figure 8: Scaled query performance after adding nodes (buffer matching impact highlighted)

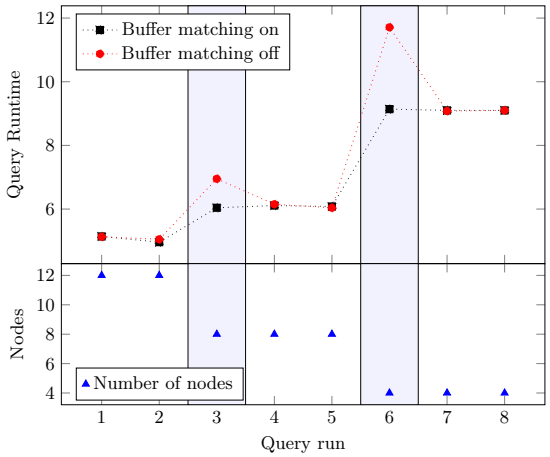

Figure 9: Scaled query performance after removing nodes (buffer matching impact highlighted)

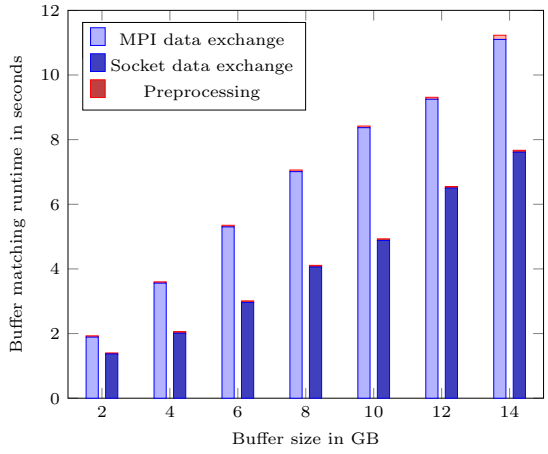

Figure 10: Buffer matching runtime for preprocessing and actual data exchange as a function of the buffer size for upscaling from 8 to 16 nodes with a constant block size of 1 MB

98%) of the whole buffer matching step. Furthermore, the plot illustrates the expected runtime increase when increasing the buffer size and shows that they correlate in a linear way. Comparing both data exchange implementations, we can observe that the socket implementation is faster in terms of runtime and also shows a smaller grow in runtime when increasing the buffer size. As discussed in Section 6, this behavior is presumably caused by the fact that the MPI implementation has to re-initiate the communication for each block, as the blocks may be randomly placed within the buffer memory space. The socket implementation on the other hand initiates the communication once before sending a data stream and therefore reduces the communication overhead.

In a further experiment we evaluate the impact of varying block sizes on the buffer matching mechanism. For this we keep the buffer size constant at 10 GB and we only consider the pure data exchange runtime, as we have seen in Figure 10 that preprocessing only takes a minor part of the overall runtime. Increasing the block size implies an increase of the necessary memory to allocate blocks. Therefore, we had to switch to scale factor 300 GB for this experiment as the overall available memory did not suffice for the largest tested block size of 8 MB and scale factor 1000 GB. Figure 11 shows the results of this experiment. For both implementations we can observe a slight increase in runtime when increasing the block size. This is caused by increased data volume that has to be exchanged, as larger block sizes come along with larger unused space or padding. Besides that, the socket implementation is not heavily impacted by varying block sizes, as data is simply written to stream sockets not considering any block boundaries. On the contrary, the MPI implementation profits from larger block sizes, as the communication overhead shrinks with the decreasing number of blocks to be sent. As a result, the difference in runtime between both implementations also shrinks with larger block sizes. Nevertheless, the choice of the block size also impacts other parts of the system, so this choice is usually fixed around a value of 1MB and can not be changed after database creation. For these block sizes, the socket implementation is surely the better choice compared to the MPI implementation.

Cluster resize usability: In the last experiment, we compare the implemented cluster resize functionality with the "inelastic" scaling, which involves the following steps:

(1) Shutdown of the system
(2) Adjustment of a list that holds the VectorH node names
(3) Restart of the system
(4) Run a query

These step are encapsulated in a script to reliably measure the runtime. We define the start state of the experiment as a running VectorH system with filled buffers. Furthermore, we define the end state as the moment we get a query result from a scaled VectorH instance. The runtime between start and end state is measured for the cluster resize feature with and without activated buffer matching, as well as for the inelastic scaling process. As query workload we choose query 1 and query 9 of the TPC-H benchmark. The buffer size is set to 10 GB per node and we investigate the cases of scaling from 8 to 16 nodes and vice versa.

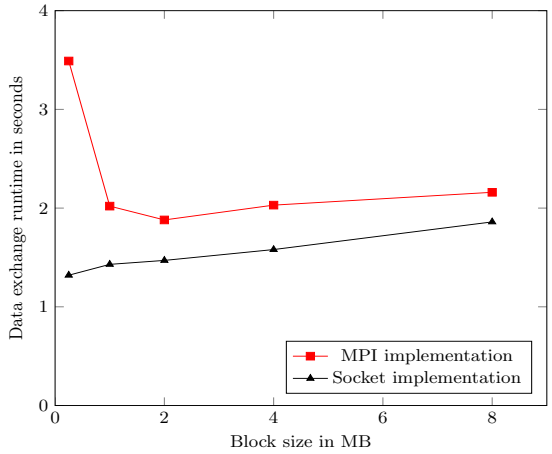

Figure 11: Buffer matching data exchange runtime as a function of the block size with constant buffer size of 10GB when upscaling from 8 to 16 nodes

	add_nodes		remove_nodes	
	Q1	Q9	Q1	Q9
BM on	57.74 s	77.55 s	32.02 s	51.06 s
BM off	58.70 s	87.95 s	32.56 s	52.04 s
Inelastic	167.68 s	223.30 s	123.20 s	147.94 s

Table 2: Runtime for scaling the system using the inelastic scaling process or the cluster resize feature with and without buffer matching (BM)

Table 2 shows the measured runtime results of the experiment. For all cases, the implemented cluster resize feature outperforms the inelastic scaling up to a factor 4. In addition to that, activated buffer matching shows a benefit in runtime for the cases of scaling the system up, caused by the buffer pre-filling. The amount of benefit is dependent on the actual query for this experiment, so query 9 shows a better speedup than query 1 using buffer matching. For the case of removing nodes, buffer matching has a negligible impact, as buffers of the remaining nodes are already filled. Therefore, a buffer merging strategy could be a possible optimization in the future. Moreover, we can state that adding nodes to the system is slower than removing nodes, both for inelastic scaling and the cluster resize feature. The reason for this is that started servers perform a collective startup with several synchronization points and do a full log replay. Increasing the number of servers, this startup time also increases, leading also the inelastic scale-up to be slower than the scale-down. In the contrary, removed nodes can shutdown independently after the remaining servers form a new communicator (see Section 4), not influencing the further query processing. This experiment is highly dependent on the cluster configuration (e.g. network speed) as well as the size of the write-ahead log that has to be replayed. Therefore, this experiment should not be used for a quantitative comparison, but is intended to show a qualitative difference between the scaling methods.

Overall the evaluation proves that the implemented cluster resize feature outperforms the "inelastic" scaling

method using a restart up to a factor of 4. The buffer matching mechanism shows to add a minor runtime overhead to the scaling step, but proves to have a major impact on the query performance. The first queries after scaling show a significant performance gain when using buffer matching, which is caused by the pre-filling of buffers. As a result, the user gets an immediate performance boost when deciding to scale the VectorH installation up, which is the behavior he expects when increasing his service cost. Furthermore, the experiments prove that the socket implementation improves the buffer matching data exchange step compared to the MPI implementation, which was the expectation this optimization was based on. As we evaluated our implementation using a private cluster, the time for acquiring resources from a cloud service provider is not included in our results.

8 CONCLUSION

In this paper, we presented our approach to adapt Actian VectorH for the elastic cloud environment. As the first goal, we implemented an elastic cluster resize feature for VectorH, enabling adding and removing nodes during system uptime and therefore avoiding the drawbacks of a full system restart, e.g. full log replay and empty buffers. For the implementation of the feature we utilized group and communicator management offered by the Message Passing Interface (MPI), which is used for node-to-node communication within VectorH. As a second contribution, we designed a partition manager that is suitable for the cloud environment. By using overpartitioning and explicitly managing partition-to-node mappings for equivalence classes of partitionings, the implemented solution minimizes partition reassignments, keeps partition co-locations, balances load on partition level and provides efficient lookup and update functions. The partition manager replaces the round-robin partition assignment in VectorH, which showed to be not suitable for the elastic cloud environment. While evaluating the cluster resize feature in combination with the implemented partition manager, queries did not show the expected speedup immediately after the resize, which was caused by empty buffers. As an optimization, we introduced the buffer matching and filling mechanism into the cluster resize feature. After changing partition mappings, nodes scan their buffers and send buffered data to other nodes in order to pre-fill their buffers, leading to immediate speedup after cluster resize. For the buffer matching data exchange, we implemented two different approaches using MPI communication and using data streams over sockets and evaluated them against each other.

The experiments showed that the elastic cluster resize feature significantly outperforms the inelastic scaling process using a system restart. Activating the buffer matching mechanism further increases the performance after the cluster resize, enabling the user to immediately profit from additional resources. Evaluating both buffer matching data exchange mechanisms, the socket implementation showed to be the better choice for all tested cases.

Future work includes the investigation on online scaling allowing concurrent read and/or write transactions during the scaling process, as well as query driven scaling, optimizing a query for a given goal (e.g., cost, runtime) by scaling the system automatically. In addition to that, further data exchange mechanisms, such as RDMA (remote direct memory access) based data exchange, will be evaluated to accelerate the buffer matching data exchange step. The buffer matching mechanism could also be further extended with a predictive buffer filling strategy for adding nodes or a buffer merging strategy for removing nodes.

REFERENCES

[1] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. 2014. ShuttleDB: Database-Aware Elasticity in the Cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*. USENIX Association, Philadelphia, PA, 33–43. https://www.usenix.org/conference/icac14/technical-sessions/presentation/barker

[2] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer International Publishing, Cham, 61–76.

[3] Peter Boncz, Marcin Zukowski, and Niels Nes. [n.d.]. MonetDB/X100: Hyper-Pipelining Query Execution. ([n. d.]).

[4] Andrei Costea, Adrian Ionescu, Bogdan Răducanu, MichałSwitakowski, Cristian Bârca, Juliusz Sompolski, Alicja Luszczak, MichałSzafrański, Giel de Nijs, and Peter Boncz. 2016. VectorH: Taking SQL-on-Hadoop to the Next Level. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1105–1117. https://doi.org/10.1145/2882903.2903742

[5] Benoit Dageville, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, and Martin Hentschel. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, San Francisco, California, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[6] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *ACM Trans. Database Syst.* 38, 1, Article 5 (April 2013), 45 pages. https://doi.org/10.1145/2445583.2445588

[7] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proceedings of the VLDB Endowment* 4, 8 (May 2011), 494–505. https://doi.org/10.14778/2002974.2002977

[8] G. Graefe. 1994. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (Feb. 1994), 120–135. https://doi.org/10.1109/69.273032

[9] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. ACM Press, Melbourne, Victoria, Australia, 1917–1923. https://doi.org/10.1145/2723372.2742795

[10] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (May 1988), 532–533. https://doi.org/10.1145/42411.42415

[11] Sandor Heman, Niels Nes, Marcin Zukowski, and Peter Boncz. [n.d.]. Positional Delta Trees to reconcile updates with read-optimized data storage. ([n. d.]), 11.

[12] Wolfgang Lehner and Kai-Uwe Sattler. 2013. *Web-Scale Data Management for the Cloud*. Springer New York, New York, NY. https://doi.org/10.1007/978-1-4614-6856-1

[13] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[14] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. 2011. A Cost-Aware Elasticity Provisioning System for the Cloud. In *2011 31st International Conference on Distributed Computing Systems*. 559–570. https://doi.org/10.1109/ICDCS.2011.59

[15] Michał Switakowski, Peter Boncz, and Marcin Zukowski. 2012. From cooperative scans to predictive buffer management. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1759–1770. https://doi.org/10.14778/2367502.2367515