# Elastic Scale-out for Partition-Based Database Systems

Umar Farooq Minhas Rui Liu Ashraf Aboulnaga
Kenneth Salem Jonathan Ng Sean Robertson
University of Waterloo, Canada
{ufminhas, r46liu, ashraf, kmsalem, jywng, sdrobert}@cs.uwaterloo.ca

*Abstract*— **An important goal for database systems today is to provide elastic scale-out, i.e., the ability to grow and shrink processing capacity on demand, with varying load. Database systems are difficult to scale since they are stateful – they manage a large database, and it is important when scaling to multiple server machines to provide mechanisms so that these machines can collaboratively manage the database and maintain its consistency. Database partitioning is often used to solve this problem, with each server machine being responsible for one partition. In this paper, we propose that the flexibility provided by a partitioned, shared nothing parallel database system can be exploited to provide elastic scale-out. The idea is to start with a small number of server machines that manage all partitions, and to elastically scale out by dynamically adding new server machines and redistributing database partitions among these servers. We present an implementation of this approach for elastic scale-out using VoltDB – an in-memory, partitioned, shared nothing parallel database system. Our main goal in this paper is to identify several manageability problems that arise when using this approach for elastic scale-out. The paper presents some of these problems and outlines a research agenda for this area.**

## I. Introduction

A software system is said to be scalable if it is able to handle increasing load simply by using more computing resources. A system can be *scaled up* by adding more computing resources, such as memory or CPUs, to the machine that it runs on. *Scale-out* permits a system to handle even larger workloads by adding more machines (which we call *nodes* in this paper), for example in a cluster. The computing platforms of today, namely computing clouds and private clusters, enable applications to access a large number of physical nodes and scale out and in *elastically*, by adding more nodes when the load increases and removing them when the load decreases. *Stateless* systems, such as web and application servers, can be scaled elastically by simply running more instances on additional physical nodes provisioned on demand. On the other hand, *stateful* systems, such as database management systems (DBMSes), are hard to scale elastically because of the requirement of maintaining consistency of the database that they manage. There is currently a lot of interest in elastically scaling database systems, and a good way to solve this problem is to start with scalable (but not elastic) database systems.

Scalable database systems enable multiple nodes to manage a database, but the set of nodes is static. These systems distribute the database among the nodes by relying on *replication* [1], [2], [3], [4] or *partitioning* [5], [6]. Replication comes with a cost due to data duplication and maintaining consistency among the replicas. Partitioning is also non-trivial since it requires users to define how to partition the database, and it makes dealing with multi-partition transactions problematic. However, partitioning seems to be gaining popularity as a way to scale database systems (e.g., [7], [8]), and in this paper we argue that partitioned, shared nothing databases are a good starting point for DBMS elasticity.

Our basic approach is to start with a small number of nodes that manage the database partitions, and to add nodes as the database workload increases. These new nodes could be (spare) physical servers that are dynamically provisioned from a local cluster. Alternatively, in cloud computing environments such as Amazon's EC2 [9], the new nodes could be dynamically provisioned virtual machines. When adding new nodes, the database partitions are redistributed among the nodes so that the workload gets shared among a larger set of nodes. Conversely, when the workload decreases, the number of nodes can be reduced and partitions can be moved to a smaller set of nodes. Redistributing partitions under this approach can be costly and difficult to manage because of its disruptive effect on transaction processing. Thus, this approach requires efficient mechanisms for partition redistribution.

We have implemented this elastic scale-out and scale-in approach in VoltDB [10], including the required efficient partition redistribution mechanism. VoltDB is a parallel shared nothing partition-based database system. Like other shared nothing database system [11], [12], [13], [14], VoltDB divides the database into disjoint partitions based on some partitioning key and distributes these partitions to the nodes of a cluster. In this paper, we describe the changes that we made to VoltDB to enable us to dynamically add new nodes to the cluster and redistribute partitions, and we experimentally demonstrate the effectiveness of our elastic scale-out and scale-in mechanisms under varying load. We also show that this approach to database elasticity gives rise to a number of research problems in the area of database provisioning and manageability. A main goal of this paper is to discuss some of these problems and present potential research directions in this area.

There is an increasing body of work in the area of elastically scalable data stores, in particular, for cloud computing environments. Key-value stores [15], [16], [17] can be scaled elastically, similar to our solution. At a superficial level, such

systems bear some resemblance to relational database systems in that they also store data as rows and columns. However, these systems provide a very simple interface that allows clients to read or write a *value* against a given *key* from the store i.e., SQL is not supported, hence the name "NoSQL". In addition, they typically support only single-row atomic updates, not general application-defined transactions like those that are supported by relational database systems. Some key-value stores provide only eventual consistency guarantees. These limitations make it easier for these systems to scale. In contrast, our system implements elastic scalability in an ACID compliant DBMS (i.e., VoltDB) that provides full SQL functionality.

ElasTras [18] is an elastically scalable, fault-tolerant, transactional DBMS for the cloud, and is closest to our work. Like VoltDB, ElasTras uses database partitioning and performs scaling at the granularity of a partition. However, ElasTras relies on a shared storage tier which allows for a simpler mechanism for data migration. VoltDB stores the data completely in-memory on each host, requiring a more elaborate mechanism for data migration (Section III). Furthermore, ElasTras relies on schema-level database partitioning and limits update transactions to a single partition. VoltDB, on the other hand, uses table level partitioning and allows multi-partition transactions at the cost of reduced performance for only these transactions. Aside from these differences, our work is very similar to ElasTras and the research challenges that we outline in this paper are applicable to both of these systems.

The rest of this paper is organized as follows. In Section II, we present an overview of VoltDB. Section III presents the changes that we made to VoltDB to make it elastically scalable. In Section IV, we present some open research problems that need to be addressed in order to implement a self-managing database system that is able to automatically provision itself by scaling elastically with varying load. Section V then focuses on the specific problem of data placement with elastic scale-out. Section VI concludes the paper.

## II. Overview of VoltDB

VoltDB [10] is a in-memory database system derived from H-Store [19]. VoltDB has a shared nothing architecture and is designed to run on a multi-node cluster. It divides the database into disjoint partitions and makes each node responsible for a subset of the partitions. Only a node that stores a partition can directly access the data in that partition, and such nodes are therefore sometimes called the "owners" of that partition. This shared nothing partitioning has been shown to provide good scalability while simplifying the DBMS implementation.

VoltDB has been designed to provide very high throughput and fault tolerance for transactional workloads. It does so by making the following design choices: a) all data is stored in main memory, which avoids slow disk operations, b) all operations (transactions) that need to be performed on the database are pre-defined in a set of stored procedures that execute on the server, i.e., ad hoc transactions are not allowed, c) transactions are executed on each database partition serially, with no concurrent transactions within a partition, thereby eliminating the need for concurrency control, and d) partitions are replicated for durability and fault tolerance.

A VoltDB cluster comprises one or more nodes connected together by a local network, each running an instance of the VoltDB server process. Each node stores one or more database partitions in main memory. The VoltDB server at each node is implemented as a multi-threaded process. For each partition hosted on a VoltDB node, a separate thread within the server process manages that partition and is responsible for executing transactions against that partition. The best practice for achieving high performance with VoltDB is to keep the number of partitions on a node slightly smaller than the number of CPU cores on that node. This way, each thread managing a partition will always be executing on its own CPU core, and there will still be some cores for the operating system and other administrative tasks. Thus, threads never contend with other threads for cores. Furthermore, these threads never wait for disk I/O since the database is in memory, and never wait for locks since transactions are executed serially. This means that the CPU cores can be running at almost full utilization doing useful transaction processing work, which leads to maximum performance.

Clients connect to any node in the VoltDB cluster and issue requests by calling pre-defined stored procedures. Each stored procedure is executed as a database transaction. VoltDB supports two types of transactions: **1) single-partition transactions,** as the name implies, are the ones involving only a single database partition and are therefore very fast, and **2) multi-partition transactions,** which require data from more than one database partition and are therefore more expensive to execute. Multi-partition transactions can be completely avoided if the database is cleanly partitionable, which is the case for the kinds of transactional workloads that VoltDB targets.

In order to tolerate node failures, VoltDB replicates partitions to more than one node. The replication factor can be specified as part of the initial configuration of the VoltDB cluster. A VoltDB cluster configured with $k$-safety will have $k+1$ instances of every partition, and can tolerate at most $k$ node failures. Transactions are executed on the different replicas of a partition in the same serial order, and VoltDB waits for all copies of a transaction to finish before acknowledging its success to the client.

## III. Enabling Elastic Scale-Out with VoltDB

In order to implement elastic scale-out, we need to provide mechanisms to: 1) dynamically add new nodes to the cluster, and 2) move database partitions from one node to another. One of the reasons for choosing VoltDB for elastic scale-out is that it already provides much of the needed functionality. This simplifies the implementation of our scale-out mechanism.

### A. Growing the size of the cluster

In large cluster deployments, node failures are common. Therefore, VoltDB implements an efficient way of identifying

failed (down) nodes and a *node rejoin* mechanism which introduces a node back into the cluster after it recovers from failure.

In our implementation, we violate the VoltDB best practices and assign more partitions to a node than the number of CPU cores on that node. This means that VoltDB threads will contend for CPU cores, but this is acceptable in a lightly loaded system. As the load on the system increases and we need to scale out, we will introduce new nodes into the cluster and move some partitions to these new nodes. When the system scales out to its limit, we will have one VoltDB thread per core. We made the design decision to build our system with the notion of a maximum scalability limit since this notion greatly simplifies the scale-out mechanism. The simplification stems from the fact that VoltDB can be made aware in advance of the number of nodes that can potentially participate in the cluster, and the mapping of partitions to these nodes. Knowing this information in advance enables us to use VoltDB's failure handling mechanism for managing scale-out as we describe next.

To enable the number of nodes in the cluster to grow dynamically we introduce a new type of node called the *scale-out* node. A scale-out node is a node that initially has no database partitions mapped to it, and that looks to VoltDB like a "failed" node. We initialize the cluster with a fixed number of additional scale-out nodes. These nodes begin as inactive (i.e., they do not store data or process transactions), so the cluster will ignore them with little overhead. Note that at this point, a scale-out node is merely a data structure with no physical counterpart. When we need to actually introduce such a node into the cluster, we can bring up an actual physical server (or a virtual machine) that corresponds to a scale-out node. To dynamically grow the size of the cluster, we use a slightly modified version of VoltDB's node rejoin mechanism. At the completion of rejoin, the scale-out node will have transitioned from the "failed" state to the "active" state, and will have database partitions mapped to it. Effectively, we have made VoltDB think that this node was part of the cluster all along, but was just failed, and is now back up. However, at this point the node still has no data for any of the partitions. The partitions need to be moved from one of the active nodes, which is the second step of scale-out.

### B. Moving database partitions between nodes

Introducing a new node into the cluster requires copying data from one of the existing nodes to the new node. This data copying step is also required when recovering a failed node. For this purpose, VoltDB implements a *recovery* mechanism that runs after a node has rejoined the cluster. The recovery mechanism works at the database partition level, and copies the data from a *source* partition to a *destination* partition in a consistent manner. Note that source partitions reside on one of the active nodes, while destination partitions reside on the node that has rejoined the cluster. Once all the database partitions mapped to the rejoining node have been recovered, it can start accepting new transactions.

For scale-out, we made small changes to the recovery mechanism. Once the scale-out node has rejoined the cluster, it will run recovery on all the database partitions mapped to it to populate them. In the original recovery mechanism, after the data is copied from the source node to the destination node, both nodes are active and accept transactions. In contrast, once recovery is finished when scaling out, the source partitions are shutdown – reflecting the change of "ownership" for scaled-out partitions from the source node to the scale-out node.

The converse of the operations described above are used when scaling in because the load on the system has become lower: partitions are moved away from scale-out nodes and back to the original nodes, and when all the partitions on a scale-out node are moved back to the original nodes, that node is *shutdown* and returned to the pool of (inactive) scale-out nodes.

### C. Demonstrating elastic scale-out and scale-in

In this section, we present experiments to illustrate that the mechanism that we implemented in VoltDB can be used to efficiently grow and shrink the size of the VoltDB cluster elastically, depending on the load.

For these experiments, we use a modified, cleanly partitionable version of the TPC-C benchmark [20]. VoltDB is intended for transactional applications in which the database fits in the total memory of all nodes in the cluster. A TPC-C database grows continuously as transactions run. Thus, without modification, a TPC-C workload is not suitable for VoltDB, since the database will eventually grow beyond the memory limit. Because VoltDB can execute transactions very quickly, the memory limit will be reached very quickly - in less than 5 minutes in our server environment. Thus, to produce a transactional test application suited to VoltDB's memory constraint, we made several changes to TPC-C to avoid database growth. First, we removed the *History* table, which keeps track of payment history and thus continues to grow. Second, in the original TPC-C specifications, the *delivery* transaction removes rows from the *NewOrder* table. We modified the delivery transaction to remove corresponding rows from *Orders* and *OrderLine* table at the same time, to prevent these tables from growing. Finally, to make TPC-C cleanly partitionable by warehouse, we eliminated *new order* transactions that access remote warehouses.

The first experiment that we present here was run with a total of three physical hosts. We have a total of 8 unique database partitions, and we run this experiment without *k*-safety. One of the three hosts is used to run the TPC-C benchmark clients and the other two are used as VoltDB servers. We start the VoltDB cluster with just one physical host that stores all eight partitions. We call this the *primary* host. We have one spare *scale-out* host which will be inactive initially. Clients connect to the VoltDB cluster (which is just one primary host) and submit TPC-C transactions. Clients in this experiment are *open loop* clients that submit the requests at a fixed rate, which we call the *offered load*. Offered load is varied during the experiment to vary the load on the VoltDB

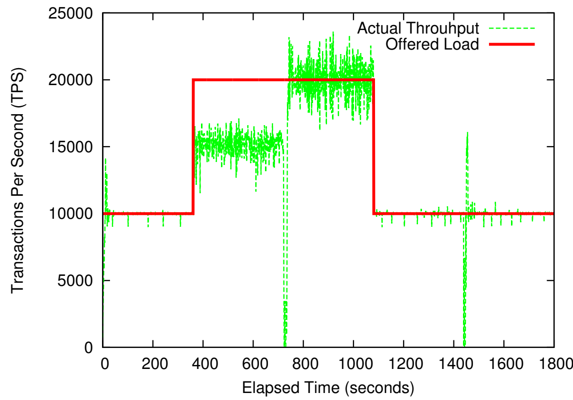


Fig. 1. Cluster Throughput with Increasing Load

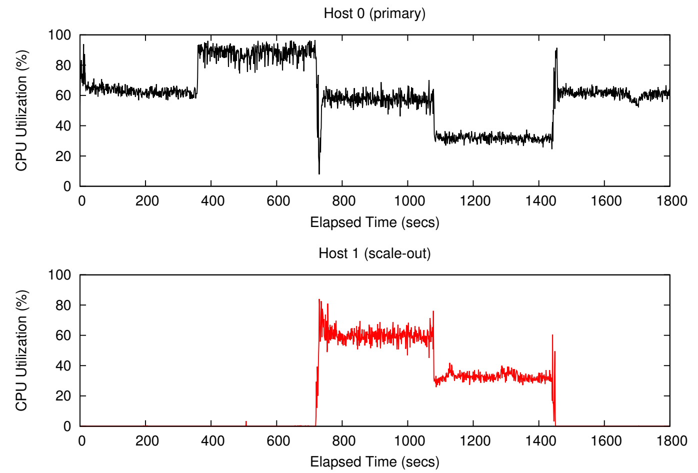


Fig. 2. Per Host CPU Utilization with Increasing Load

cluster. We run the test for a total of 30 minutes and measure the throughput of the cluster in terms of transactions per second (TPS). Our goal is to demonstrate that we can handle increases and decreases in the offered load by adding or removing hosts to the VoltDB cluster.

Figure 1 shows the offered load and the actual throughput of the cluster. Figure 2 shows the CPU utilization of each host (primary and scale-out) during the duration of the experiment. Initially, there is just one host, which is offered a fixed load of 10,000 requests/sec. The single host is able to service this load successfully, with about 60% CPU utilization (Figure 2). At about 400 seconds into the test, we increased the offered load to 20,000 requests/sec. A single host is not able to handle this load, and can provide only 15,000 TPS, as shown by the green line in Figure 1. We also see that the primary host (Host 0) is highly loaded now with about 90% CPU utilization (Figure 2). At about 750 seconds into the test, we do a scale-out operation by adding another host (Host 1) to the VoltDB cluster. Note that before this point, Host 1 was idle thus its CPU utilization was at 0% (Figure 2). During the scale-out operation, we move 4 out of 8 partitions from the primary host (Host 0) to the scale-out host (Host 1). The amount of data moved was about 1 GB. After the scale-out operation, the cluster is once again able to successfully service the offered load of 20,000 TPS. We reduce the offered load back to 10,000 requests/second at around 1100 seconds. Now both hosts are very lightly loaded at around 30% CPU utilization (Figure 2). We perform a scale-in operation at around 1450 seconds, once again leaving only one node in the cluster that stores all partitions. The green line in Figure 1 shows that our scale-out and scale-in mechanisms have a minimal transient effect on throughput. When doing a scale-out or scale-in there is a small drop in throughput (as expected) but the drop lasts only for a very short time.

The second experiment, which is similar to the first, was run with a total of five physical hosts. Like before, one of the three hosts is used to run the TPC-C benchmark clients but now we have four hosts in the VoltDB cluster instead of two. We start the VoltDB cluster with just one physical host, the *primary* host. When the offered load is increased, the number of hosts in the VoltDB cluster is also increased, up to a total of four hosts (including the primary host). This approach ensures that the cluster is always adequately provisioned to handle the offered load, thus proactively avoiding any overload conditions. Similarly, when the offered load is decreased, we decrease the number of hosts in the VoltDB cluster to avoid underload. In total, we perform three scale-out operations at points A, B, and C in Figure 3, and we perform three scale-in operations at points D, E, and F. In addition to showing the offered load and scale-out/scale-in points, Figure 3 shows the throughput during this experiment. Figure 4 shows the CPU utilization of the different hosts over time. The experiment starts with 16 TPC-C partitions on one host, and through scale-out we get to four hosts, and then back to one host through scale-in. Figure 5 shows the mapping of partitions to hosts at different points in the experiment, along with the expected mapping of partitions to cores within a host. (The mapping of partitions to cores is managed by the operating system and not by VoltDB.) The total amount of data migrated during the scale-out operations is approximately 3 GB.

Note that a single host in our system can handle up to 10,000 TPS. With a total of four hosts, the system is able to effectively handle approximately 40,000 TPS during the peak. As the offered load starts to subside around 1,000 seconds into the test, the cluster gradually becomes over-provisioned, so it is still able to handle the offered load even after the scale-in operations at points D, E, and F in Figure 3. Figure 4 shows that none of the hosts in the cluster is ever overloaded because of our anticipatory scale-out policy. In this experiment, we add and remove nodes at fixed pre-defined points that we know will match the pre-defined offered load. In a more realistic setting, a DBMS that is able to scale elastically would implement a controller that would automatically add or remove hosts from the cluster whenever it anticipates an overload or underload condition.

These experiments show that the changes we implemented in VoltDB allow us to effectively handle varying load by dynamically growing and shrinking the size of the VoltDB cluster.

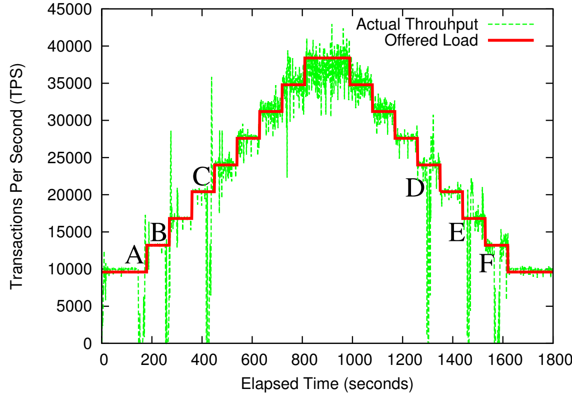


Fig. 3. Cluster Throughput with Increasing Load and Multiple Scale-out Operations

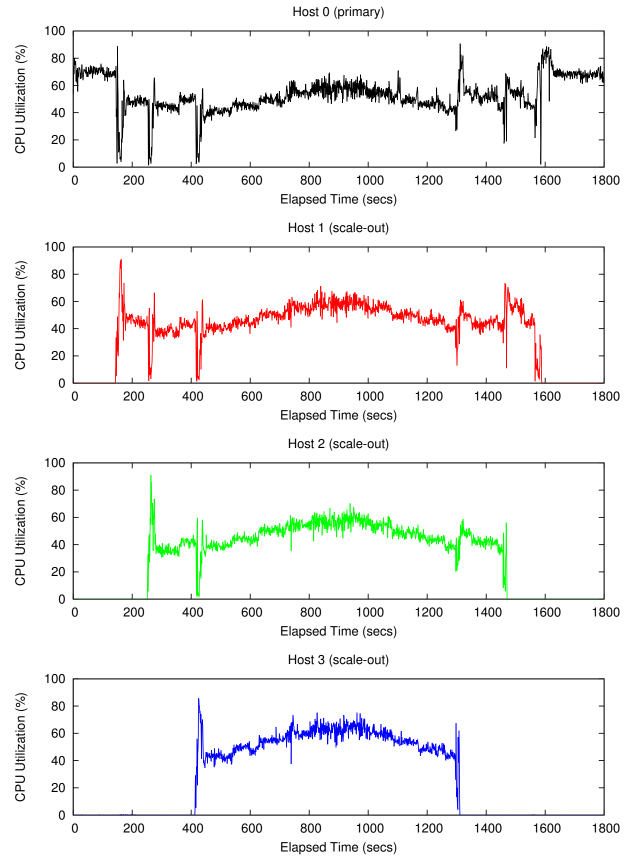


Fig. 4. Per Host CPU Utilization with Multiple Scale-out Operations

## IV. Research Problems

We envision a number of different research challenges that need to be addressed to implement a database management system that can scale elastically in response to a time-varying workload. In this section, we discuss a number of these challenges. Some of these challenges are specific to the VoltDB scale-out implementation presented earlier while the rest are more generally applicable to any approach for elastic scale-out.

### A. *When to scale-out/in?*

A DBMS needs to be equipped with mechanisms that are able to trigger scale-out/scale-in at the required times. Timing of these scaling decisions is critical. Responding to a load spike too early or too late will result in an over- or under-provisioned system, respectively. A simple *reactive* strategy is to monitor some observable system parameter such as per host CPU utilization. If CPU utilization of a particular host is greater than a max threshold (say 80%), it is considered to be overloaded and thus a scale-out operation is performed. Similarly, if the load on a particular host is less than a min threshold (say 40%), that host is considered to be underloaded and so the DBMS can perform a scale-in operation to minimize wasted server resources. On the other hand, a *proactive* approach would require building performance models that are able to accurately predict load on the DBMS given some information about its workload and resource allocation. Using such an approach, a DBMS would be able to predict when load will exceed server capacity and thus scale out proactively. These are just two examples of approaches that can be used to answer the question: when to scale-out/scale-in? More research is needed to establish which of these approaches (or some other approach) works best for which scenarios.

### B. *Minimizing the cost of scaling (speeding up scaling)*

Scaling out requires some administrative work, for example, adding a new node to the cluster, copying data to this new node, and warming it up before client requests are redirected to it. Each of these steps needs to be optimized in order to minimize the cost of the scaling operation. The most important and probably the most costly part of scaling is the data movement from one physical host to another physical host. We could develop cost models that estimate the cost of data movement for a scaling operation, e.g., I/O cost of transferring disk blocks, and the cost of network traffic. Optimizing the cost of the scaling operation with these models in place would require choosing a data movement "plan" that has the least cost, as estimated by these cost models. Such a plan can take advantage of the fact that each VoltDB partition is replicated $k + 1$ times on different hosts for $k$-safety.

Also, the actual implementation of the scaling operation will play an important role in determining its cost. For example, in our implementation of scaling for VoltDB we explored two different strategies. A naive, and very inefficient strategy is to do the scale-out operation as a single, multi-partition transaction that blocks all partitions on both the source and destination nodes and thus has an adverse effect on system throughput. On the other hand, our optimized implementation of scale-out divides the task into multiple phases making it more efficient and less disruptive. We believe that such design choices will exist in any DBMS that is being designed to be elastically scalable and so should be evaluated carefully.

### C. *Exploiting replicas*

VoltDB requires that database partitions be replicated in order to provide fault tolerance and durability in the presence of failures of nodes in the VoltDB cluster. This requirement

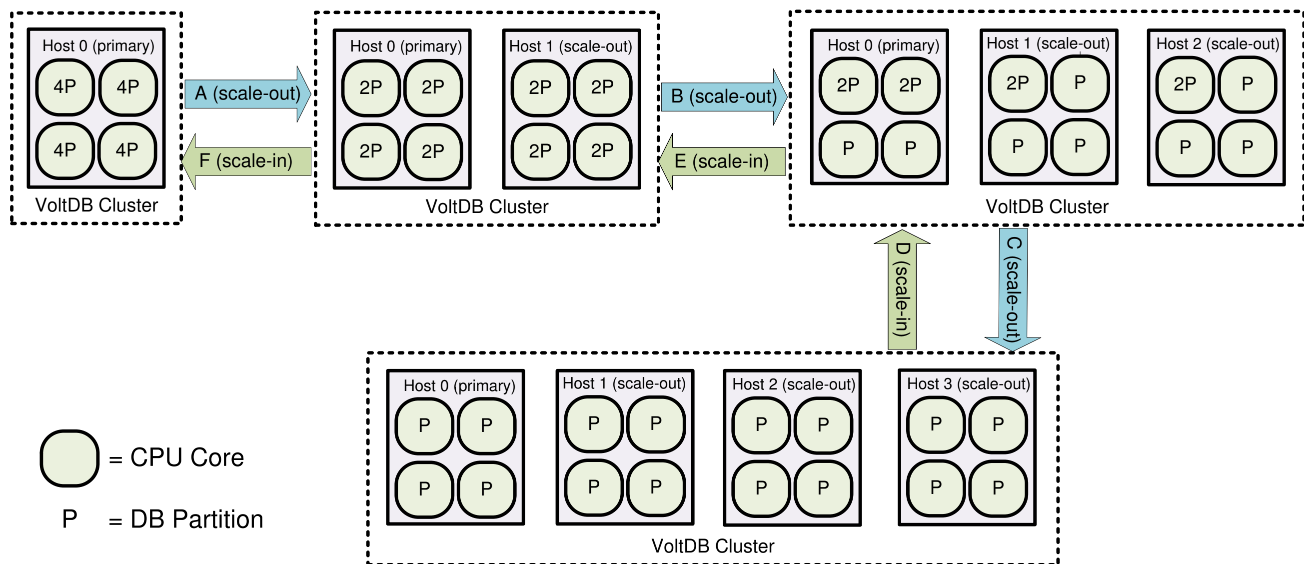


Fig. 5. Scale-out Illustration

not only allows the system to operate even when nodes fail, but may also present opportunity for optimizing scaling operations. In VoltDB, the loads on each replica of a database partition are exactly the same. One of the ways in which this "symmetry" of replica loads can be exploited is by minimizing the impact on transaction processing during a scaling operation. If a database partition is being moved, there will be a small window of unavailability for that partition. During that window new transactions intended for that partition can be redirected to any of its replicas, thus minimizing the impact on system throughput. These transactions can then later be applied to the copy of the partition that is being moved to "catch it up" and make it consistent with other copies, before allowing it to handle new transactions.

Note that VoltDB requires that no two replicas of a database partition can reside on the same physical host. With this policy, entire nodes may be replicas of each other regardless of the placement strategy used to distribute database partitions to nodes. For example, consider a case where there are two physical hosts and we need to place four partitions on them, with a replication factor $k = 1$, i.e., a total of eight partitions. With the requirement that no two replicas of a partition can reside on the same physical host, one host stores four partition, with the other host storing the replicas of those four partitions. What this means is that the load on these hosts will be exactly identical. If one of them is overloaded, the other will be as well, and vice versa. Can we manage the placement of partitions so that we can manage load in a better manner while implementing elastic scale-out? More research is needed to address this question.

Note that we present the research question of exploiting replicas in the context of VoltDB but many parallel DBMS systems today have a similar requirement to use replication for fault tolerance. Therefore, the discussion above applies to these other systems as well.

### D. *Partitioning and partition movement*

In our approach, we enable elasticity by moving database partitions for scale-out. The key idea is that a database is stored as a collection of partitioning units ($P_1$, $P_2$, ..., $P_n$). When the database server becomes overloaded, it transfers the "ownership" of some of these partitions to another node that is dynamically added to the cluster. Similarly, for scale-in, an underloaded node will transfer the ownership of its partitions to some other node in the cluster.

VoltDB effectively uses a two-stage mapping of database partitions to CPU cores. The first mapping is of database partitions to VoltDB's server threads. As mentioned earlier, at every node, for each database partition hosted at that node, VoltDB creates a server thread to manage that partition. In our implementation, we used a one-to-one mapping between database partitions and server threads but whether that is always the best choice is not entirely clear. An alternative is to map multiple database partitions to a single VoltDB server thread. The second mapping is of these server threads that manage database partitions to CPU cores. As mentioned earlier, recommended best practices for VoltDB suggest having no more server threads than the total number of CPU cores on each physical host. This strategy means that the operating system will most likely give each VoltDB thread its own dedicated core, which is one of the ways that VoltDB uses to achieve maximum performance. However, in our experience with VoltDB, *over-committing* CPU cores by assigning more server threads to a host than the number of cores that it has, and hence running more than one thread on each core, had minimal impact on system throughput (around 1%). These two levels of mapping might be specific to VoltDB but similar choices exist in other partitioned database systems. Coming

up with an optimal mapping is an interesting manageability problem.

Lastly, the loads placed on each partition might not be the same, as is implicitly assumed by VoltDB. It is possible that certain partitions might be "hotter" than some other partition, which may result in unbalanced load among the nodes of the cluster. Nodes hosting one or more hot partitions will always be more loaded than other nodes. With non-uniform partition loads, the initial placement of partitions and the decision which partitions to move to which nodes during a scaling operation become non-trivial. We discuss this specific issue in more detail in the following section.

## V. Parition Placement

The problem of partition placement is to find an "optimal" allocation of partitions to physical servers. Optimality in this case is usually defined in terms of load balancing or overall throughput maximization. The general partition placement problem has been shown to be NP-complete [21]. Various heuristic techniques are typically used to find a solution.

As noted in the previous section, if the load on partitions is not uniform, i.e., some partitions are considerably more loaded than other partitions, the data placement problem becomes even more important. We discuss this problem in the context of initial placement, and for scaling out, in the following sections.

### A. Initial placement

Figure 6 shows the result of a simulation that compares two data placement strategies in terms of overall system load-balancing when database partitions are not uniformly loaded. In our simulation, we use 40 partitions and the loads on these partitions are distributed according to a Zipf distribution. The x-axis in Figure 6 shows the value of the Zipf's skew parameter (the higher this parameter, the more skewed the distribution), and the y-axis shows the variance in per host load. The higher the variance, the more unbalanced the system is. We compare VoltDB's default placement strategy (i.e., round-robin) with a simple greedy heuristic that places the highest loaded partitions on least loaded hosts. As shown in Figure 6, for moderate skew (0.4) both strategies provide equally well balanced system. As the skew in partition load grows, the default (round-robin) strategy starts to perform worse than the simple heuristic. We see the greatest opportunity for optimization when Zipf's skew coefficient is around 1. As the skew increases beyond 1, both strategies start to perform worse. This is because with higher skew values, a few partitions always have the most load, so the overall variance in per host load will be very high regardless of the placement strategy used.

As shown by the simulation above, initial placement of database partitions matters when the load is non-uniform. Moreover, there is room for improvement over a simple strategy such as round-robin. An obvious research challenge is to devise a technique that is able to provide close to optimal load balance in the presence of skew in partition load. The problem of partition placement and re-distribution for load

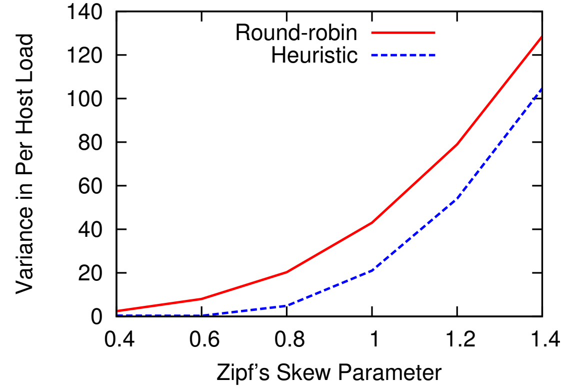


Fig. 6. Round-robin vs. Greedy Heuristic

balancing has been studied extensively in the past [5], [6], [21], [22]. This problem is also very similar to the problem of consolidating multiple database workloads with an aim to minimize the number of physical servers used and to maximize load balance [23]. So the question of initial placement of partitions on VoltDB nodes can simply be a question of choosing a suitable technique from the literature, probably with minor modifications for this specific problem of elastic scale-out.

### Problem formulation

As an example, we present one formulation for the initial partition placement problem for VoltDB as a non-linear binary integer programming problem.

*Notation:* Let $P$ denote the total number of unique partitions that are to be placed on $N$ physical hosts in the cluster, where each partition is replicated $k$ times. Let $X$ denote the assignment matrix of partitions to hosts, where the elements of this matrix are $X_{ij} \in \{0, 1\}$, where $i = 1, 2, ..., P$ and $j = 1, 2, ..., N$. A value $X_{ij} = 1$ means that partition $i$ is assigned to node $j$. Let $C_j$ denote the capacity of the $j$-th node in terms of the number of partitions it can host, and $L_j$ denote maximum CPU load allowed on the $j$-th node. Let $l_i$ denote CPU load generated by the $i$-th partition. Let $\mu$ denote the mean load across all nodes in the cluster. And finally, let $k$ denote the replication factor.

*Constraints:* Given the above notation, a feasible solution has to satisfy the following constraints:

1) Every replica of a partition must be assigned to exactly one server, and the servers must be different: $\sum_{j=1}^{N} X_{ij} = k$ for $i = 1, 2, \ldots, P$
2) A cluster node $j$ can host a maximum of $C_j$ partitions: $\sum_{i=1}^{P} X_{ij} \leq C_j$ for $j = 1, 2, \ldots, N$
3) The load of all the partitions assigned to a node $j$ should be less than $L_j$: $\sum_{i=1}^{P} X_{ij} * l_i \leq L_j$ for $j = 1, 2, \ldots N$

Note that all of the above constraints are *linear* constraints.

*Objective Function:* For load balancing, an objective function could be to minimize variance in load across all nodes in the cluster. Using the above notation, such an objective function can be expressed as:

$$\min \sum_{j=1}^{N} \left(\sum_{i=1}^{P} X_{ij} * l_i - \mu\right)^2$$

Note that in our formulation the objective function is *non-linear*. And because our decision variable (i.e., $X_{ij}$) is binary, we have to use non-linear binary integer programming to solve this problem. This is just one formulation of the partition placement problem. A different formulation might have linear constraints and a linear objective function, and thus may be solved by using any standard linear programming software.

### B. Placement for scaling out

The possibility of the database cluster growing and shrinking dynamically adds another dimension to the problem of partition placement. Previous research either does not deal with the data re-distribution problem for load balancing, or if it does, it works with a fixed number of physical servers. Clearly, this is not sufficient for the case where physical servers can be added or removed dynamically for elastic scaling. Finding a data placement for $M$ physical servers starting with a placement for $N$ servers ($M > N$) while minimizing data migration has a lot of room for innovation and research.

For the examples that we considered in this paper, we assumed that the database cluster will grow by one physical server at a time. Perhaps, for certain situations, a more efficient approach in terms of data migration cost may be to add more than one physical server at the time of scale-out. How to determine the number of physical servers to add at scale-out is another interesting research question. Once the number of servers to add has been determined, the next thing is to choose which partitions to move from which nodes, and to which new nodes. The strategy used for initial placement can be used here as well.

## VI. Conclusion

A self-managing database system should be able to automatically and elastically grow and shrink the computing resources that it uses in response to variations in load. Database systems today do not support this kind of elastic scale-out and scale in due to unique challenges involving the need to maintain consistency of the database while elastically scaling. In this paper, we outline a solution for database scale-out using a shared nothing partition-based parallel database system, namely VoltDB. We have presented the required changes that can be incorporated in a system like VoltDB to make it elastically scalable. More specifically, we present mechanisms to dynamically add more nodes to an existing VoltDB cluster and to move partitions from existing nodes to the new nodes in an efficient manner. We also discuss various research challenges that need to be overcome to build a truly elastic DBMS. We believe that a solution implemented using the techniques presented in this paper, and that addresses the research challenges envisioned, can provide effective elastic scaling for partitioned database systems.

## References


[1] C. Amza, A. L. Cox, and W. Zwaenepoel, "Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites," in *Middleware*, 2003, pp. 282–304.

[2] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, consistency, and practicality: are these mutually exclusive?" in *SIGMOD*, 1998, pp. 484–495.

[3] B. Kemme and G. Alonso, "Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *VLDB*, 2000, pp. 134–143.

[4] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso, "Database replication techniques: A three parameter classification," in *SRDS*, 2000, pp. 206–215.

[5] M. Mehta and D. J. DeWitt, "Data placement in shared-nothing parallel database systems," *VLDB Journal*, vol. 6, no. 1, pp. 53–72, 1997.

[6] G. P. Copeland, W. Alexander, E. E. Boughter, and T. W. Keller, "Data placement in bubba," in *SIGMOD*, 1988, pp. 99–108.

[7] "Database sharding whitepaper," www.dbshards.com/articles/database-sharding-whitepapers/.

[8] "Database Sharding at Netlog, with MySQL and PHP," http://nl.netlog.com/go/developer/blog/blogid=3071854.

[9] "Amazon Elastic Compute Cloud (Amazon EC2)," http://aws.amazon.com/ec2/.

[10] "VoltDB," http://voltdb.com/.

[11] "Teradata Database," http://www.teradata.com/t/products-and-services/database/teradata-13/.

[12] R. B. Melnyk and P. C. Zikopoulos, *DB2: The Complete Reference*. McGraw-Hill, 2001.

[13] "Microsoft SQL Server 2008," Microsoft, http://www.microsoft.com/sqlserver/2008/en/us/default.aspx. [Online]. Available: http://www.microsoft.com/sqlserver/2008/en/us/default.aspx

[14] "MySQL Cluster," http://www.mysql.com/products/database/cluster/.

[15] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *SOSP*, 2007, pp. 205–220.

[16] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, pp. 35–40, 2010.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *TOCS*, vol. 26, pp. 4:1–4:26, June 2008.

[18] S. Das, D. Agrawal, and A. El Abbadi, "Elastras: an elastic transactional data store in the cloud," in *HotCloud*, 2009.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: a high-performance, distributed main memory transaction processing system," *PVLDB*, vol. 1, no. 2, pp. 1496–1499, 2008.

[20] "The TPC-C Benchmark," http://www.tpc.org/tpcc/.

[21] D. Saccà and G. Wiederhold, "Database partitioning in a cluster of processors," in *VLDB*, 1983, pp. 242–247.

[22] P. M. G. Apers, "Data allocation in distributed database systems," *TODS*, vol. 13, pp. 263–304, 1988.

[23] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *SIGMOD*, 2011, pp. 313–324.