P-Store: An Elastic Database System with Predictive Provisioning

Rebecca Taft*
rytaft@csail.mit.edu
MIT

Nosayba El-Sayed†
Marco Serafini
nelsayed@hbku.edu.qa
mserafini@hbku.edu.qa
Qatar Computing Research Institute - HBKU

Yu Lu
yulu3@illinois.edu
University of Illinois Urbana-Champaign

Ashraf Aboulnaga
aaboulnaga@hbku.edu.qa
Qatar Computing Research Institute - HBKU

Michael Stonebraker
stonebraker@csail.mit.edu
MIT

Ricardo Mayerhofer
Francisco Andrade
ricardo.mayerhofer@b2wdigital.com
francisco.jose.andrade@gmail.com
B2W Digital

ABSTRACT

OLTP database systems are a critical part of the operation of many enterprises. Such systems are often configured statically with sufficient capacity for peak load. For many OLTP applications, however, the maximum load is an order of magnitude larger than the minimum, and load varies in a repeating daily pattern. It is thus prudent to allocate computing resources dynamically to match demand. One can allocate resources reactively after a load increase is detected, but this places additional burden on the already-overloaded system to reconfigure. A predictive allocation, in advance of load increases, is clearly preferable.

We present P-Store, the first elastic OLTP DBMS to use prediction, and apply it to the workload of B2W Digital (B2W), a large online retailer. Our study shows that P-Store outperforms a reactive system on B2W's workload by causing 72% fewer latency violations, and achieves performance comparable to static allocation for peak demand while using 50% fewer servers.

ACM Reference Format:
Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3190650

*With Cockroach Labs at time of publication.
†Work done while part of a joint MIT-QCRI postdoc program.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190650

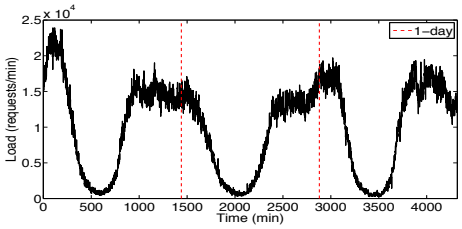


Figure 1: Load on one of B2W's databases over three days. Load peaks during daytime hours and dips at night.

1 INTRODUCTION

Large, modern OLTP applications increasingly store their data in public or private clouds. Many of these applications experience highly variable and spikey traffic patterns. Prior work on elastic databases has shown how a DBMS can automatically adapt to unpredictable workload changes to meet the throughput and latency requirements of its clients [26, 27, 31]. However, these systems suffer from poor performance during reconfiguration because reconfiguration is only triggered when the system is already under heavy load. Such performance issues make elastic database systems unworkable for companies that require high availability and fast response times. These issues could be avoided if reconfiguration were started earlier, but that requires knowledge of the future workload. Fortunately, OLTP workloads often follow a cyclic, predictable pattern. This paper shows how to take advantage of these patterns to reconfigure the DBMS *before* performance issues arise.

Online retail companies in particular have extremely predictable aggregate load, essentially following a sine wave throughout the course of the day. Furthermore, the difference between the crest and the trough of the wave is large. In this paper, we examine the workload of one such online retail company, B2W Digital (B2W) [2]. Figure 1 shows the B2W workload over three days. As can be seen, the peak load is about 10× the trough.

If companies like B2W could take advantage of predictive modeling to use exactly as many computing resources as needed to manage their workload, they could reduce the average number of servers

needed for their database by about half. In the case of a private cloud, these servers could be temporarily repurposed for some other application in the organization. In a public cloud, the reduction in servers translates directly into reduced expenses.

In this paper, we focus on workload prediction and dynamic provisioning for distributed, shared nothing OLTP database systems, an area that has not been explored by previous work. There has been a lot of work in the general area of provisioning in data centers, Infrastructure-as-a-Service cloud systems and web applications (e.g. [15, 16, 23, 28, 29, 34]). These are *stateless* services, where the startup cost of adding a new server is fixed. Shared nothing databases pose harder challenges because they are *stateful* services, which require selective state migration. Some work has targeted database systems, but it focused either on analytics workloads [10, 18, 25], which have much looser performance requirements than OLTP systems, or multi-tenant settings composed of many small database instances that can be served by a single server [4].

This paper examines how to realize cost savings by reconfiguring the database *proactively* before the load exceeds the capacity of the system. Previous work has shown that reconfiguring a main-memory OLTP database in reaction to detected skew or overload can significantly improve performance [27, 31]. But responding *reactively* to overload entails performing data migration while the system is running at its maximum capacity. While data is in flight, the system must still accept transactions and preserve consistency guarantees. Depending on the amount of data that must be migrated, reconfiguration can take anywhere from a few seconds to ten minutes or more [11]. In the absence of additional available resources, migration interferes with the execution of regular transactions and degrades the performance of the system. This is not an option for online retail companies because their customers will experience slower response times at the start of a load spike when the database tries to reconfigure the system to meet demand. Many from industry have documented that slow response times for end users leads directly to lost revenue for the company [5, 6, 20–22]. In a sense, the start of the overload period is exactly the wrong time to begin a reconfiguration, which is a weakness of all reactive techniques.

This paper presents P-Store, the first elastic OLTP DBMS to use state-of-the-art time-series prediction techniques to forecast future load on the database. Instead of waiting for the database to become overloaded, it proactively reconfigures the database when the system still has sufficient resources to migrate data and serve client transactions concurrently and without performance interference. To decide when to start a new reconfiguration and how many machines to allocate, P-Store uses a novel dynamic programming algorithm. The algorithm produces a schedule that minimizes the number of machines allocated while ensuring sufficient capacity for the predicted load, which is a constantly moving target.

This paper shows that our methods can predict real workloads from B2W (as well as Wikipedia [14]) with high accuracy. We demonstrate the superiority of P-Store over a state-of-the art purely reactive system, E-Store [31]. Overall, we envision a composite strategy for elastic provisioning in shared nothing OLTP DBMSs, which is a combination of complementary techniques: (i) *predictive provisioning*, the focus of this paper; (ii) *reactive provisioning* to react in real time to unpredictable load spikes; and (iii) *manual provisioning* for rare one-off, but expected, load spikes (e.g. special promotions for B2W). The evaluation shows that our approach, which combines predictive and reactive techniques, is sufficient to nearly always ensure adequate capacity – even during the load spikes on Black Friday, B2W's biggest promotional sale day of the year. Thus, manual provisioning is not strictly necessary, but may still be used as an extra precaution for rare, important events. The major contributions of this work are:

- A dynamic programming algorithm to determine when and how to reconfigure a database given predictions of future load.
- A scheduling algorithm for executing a reconfiguration, as well as a model characterizing the elapsed time, cost and effective system capacity during the reconfiguration.
- An analysis showing the effectiveness of using Sparse Periodic Auto-Regression (SPAR) for predicting database workloads.
- An open-source benchmark for an online retail application [30].
- A comprehensive evaluation using a real dataset and workload.

2 BACKGROUND

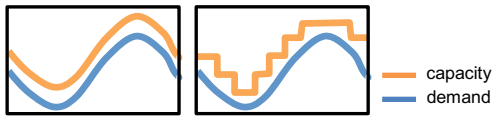
Increasingly, OLTP applications are moving to main-memory, multi-node DBMSs because of their dramatically superior performance. As such, in this study we use H-Store [19], a multi-node, shared nothing main-memory OLTP DBMS. An H-Store cluster consists of one or more physical machines (also referred to as servers or nodes), each of which contains one or more logical data partitions.

Tables in H-Store are split horizontally into disjoint sets of rows, which are each assigned to one of the data partitions (for simplicity we do not consider replication). The assignment of rows to partitions is determined by one or more columns, which constitute the *partitioning key*, and the values of these columns are mapped to partitions using either range- or hash-partitioning. Transactions are routed to specific partitions based on the partitioning keys they access.

H-Store is extremely efficient as long as data is not heavily skewed, there are few distributed transactions, and there are enough CPU cores and data partitions to handle the incoming requests. Previous database elasticity studies such as Accordion [26], E-Store [31] and Clay [27] have shown how scaling out and reconfiguring H-Store with a live migration system can help alleviate performance issues due to skew, distributed transactions, and heavy loads.

Accordion moves data at the granularity of fixed, pre-defined data chunks, whereas E-Store and Clay work at finer granularity. The E-Store system works by constantly monitoring the CPU utilization of each database partition. If an imbalance is detected, detailed monitoring is enabled for a short period of time to determine which data is "hot" and therefore causing the imbalance. Next, E-Store creates a new partition plan to relocate these hot tuples and balance the workload. Finally, a live migration system such as Squall [11] reconfigures the database to match the new partition plan. E-Store does not consider distributed transactions, so it is designed for workloads in which most transactions access a single partitioning key. Clay generalizes the E-Store approach to multi-key transactions. Rather than moving individual hot tuples, Clay moves "clumps" of hot and cold tuples that are frequently accessed together in order to balance the workload without creating new distributed transactions.

One major downside of the existing elastic database systems is that they are all **reactive**, meaning they do not reconfigure the database until performance issues are already present. At that point, it may be too late to reconfigure the system without severely impacting



(a) Ideal Capacity (b) Actual Servers Allocated

Figure 2: Ideal capacity and actual servers allocated to handle a sinusoidal demand curve.

performance, as discussed in the previous section. To alleviate this performance problem, we have designed P-Store, the first **predictive** elastic database system which reconfigures the database before performance issues arise. Throughout the rest of the paper we describe in detail how the system works, and demonstrate in our evaluation its superiority to reactive approaches.

3 PROBLEM STATEMENT

We now define the problem that predictive elasticity seeks to solve. We consider a DBMS having a *latency constraint*. The latency constraint specifies a service level agreement for the system: for example, that 99% of the transactions must complete in under 500 milliseconds. The *predictive elasticity problem* we consider in this paper entails minimizing the cost C over T time intervals:

$$C = \sum_{t=1}^{T} s_t \tag{1}$$

where s_t is the number of servers in the database cluster at time t. For convenience, we have included these and other symbols used throughout the paper in a table in Appendix A.

A solution to the predictive elasticity problem must indicate *when* to initiate a reconfiguration of the database (if at all) and the *target* number of servers for each reconfiguration. In order to minimize the cost in Equation (1) and respect the latency constraint, our system should try to make the database's capacity to handle queries as close as possible to the demand while still exceeding it. In the ideal case, the capacity curve would exactly mirror the demand curve with a small amount of buffer (see Figure 2a). In reality, we can only have an integral number of servers at any given time, so the actual number of servers allocated must follow a step function (see Figure 2b). This must be taken into consideration when minimizing the gap between the demand function and the capacity curve.

An additional complexity in the predictive elasticity problem is that this step function is actually an approximation of the capacity of the system. The effective capacity of the system does not change immediately after a new server is added; it changes gradually as data from the existing servers is offloaded onto the new server, allowing it to serve queries. The next section describes how P-Store manages this complexity and solves the predictive elasticity problem.

4 ALGORITHM FOR PREDICTIVE ELASTICITY

This section describes P-Store's algorithm for predictive elasticity. First, we describe preliminary analysis that is required to extract key DBMS parameters, such as the capacity of each server (Section 4.1). Then we introduce the key assumptions and discuss the applicability of the algorithm (Section 4.2). Next, we introduce P-Store's algorithm to determine a sequence of reconfigurations that minimizes cost and respects the application's latency constraint (Section 4.3). Finally, we show how the timing and choice of reconfigurations depend on the way reconfigurations are scheduled (Section 4.4). The predictive elasticity algorithm requires predictions of future load, and we describe the P-Store load prediction component in Section 5. Section 6 describes how we put the components together in P-Store.

4.1 Parameters of the Model

Our model has three parameters that must be determined empirically for a given workload running on a given database configuration:

(1) Q: Target throughput of each server. Used to determine the number of servers required to serve the predicted load.

(2) $\hat{Q}$: Maximum throughput of each server. If the load exceeds this threshold, the latency constraint may be violated.

(3) D: Shortest time to move all of the data in the database exactly once with a single sender-receiver thread pair, such that reconfiguration has no noticeable impact on query latency. Reconfigurations scheduled by P-Store will actually move a *subset* of the database with *parallel* threads, but D is used as a parameter to calculate how long a reconfiguration will take so it can be scheduled to complete in time before a predicted load increase. We assume that D increases linearly with database size.

All of these parameters can be determined through offline evaluation based on the latency constraint of the system as defined in Section 3.

Q and $\hat{Q}$ can be determined empirically by running representative transactions from the given workload on a single server, and steadily increasing the transaction rate over time. At some point, the system is saturated and the latency constraint is violated. We set $\hat{Q}$ to 80% of this saturation point to ensure some "slack". Q should be set to some value below $\hat{Q}$ so that normal workload variability does not cause load to exceed $\hat{Q}$. We set Q to 65% of the saturation point in most of our evaluation, but we show through simulation that clients can vary Q to prioritize either minimizing cost or maximizing performance.

D is determined by fixing the transaction rate per node at $\hat{Q}$ and executing a series of reconfigurations, where in each reconfiguration we increase the rate at which data is moved. At some point, the reconfiguration starts to impact the performance of the underlying workload and lead to violations of the latency constraint because there are not enough CPU cycles to manage the overhead of reconfiguration and also execute transactions. D is set to the reconfiguration time of moving the entire database at the highest rate for which reconfiguration has no noticeable impact on query latency, plus a buffer of 10%. The buffer is needed because D will actually be used to calculate the time to move subsets of the database (not the entire thing), and the data distribution may not be perfectly uniform.

4.2 Applicability

The proactive reconfiguration algorithm we will describe in Section 4.3 relies on several assumptions. We have validated these assumptions for the B2W workload, and we believe that they are widely applicable to many OLTP applications, as confirmed by the publicly available data on the Wikipedia workload [14]. The key assumptions are:

- **Load predictions are accurate to within a small error.** Section 5 shows how SPAR, the default predictive model used by P-Store, works well for several workloads. But our algorithm can be combined with any predictive model if it is well suited for a given workload.

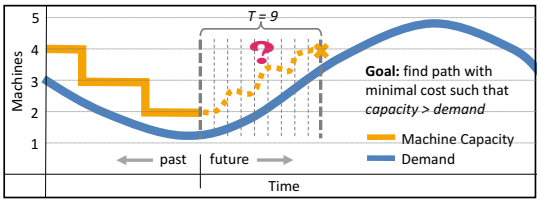


Figure 3: Goal of the Predictive Elasticity Algorithm.

- **The workload mix is not quickly changing.** This is a reasonable assumption for most OLTP workloads, in which the set of transactions and their distribution do not change very often. When they do change, we can simply measure Q and $\hat{Q}$ again.
- **The database size is not quickly changing.** This is true of many OLTP applications that keep only active data in the database. Historical data is moved to a separate data warehouse. Any significant size increase or decrease requires re-discovering D.
- **The workload distribution is (approximately) uniform across the data partitions.** Different tuples in a database often have skewed access frequencies, but this skew is smoothed out when the tuples are randomly grouped into data partitions with a good hash function. As a result, the load skew among data partitions (and thus servers) is generally much lower than the load skew among tuples.
- **The data is distributed uniformly across the data partitions.** Similar to the previous point, some keys may have more data associated with them than others, but differences are generally smoothed out at the partition level.
- **The workload has few distributed transactions.** Transactions accessing multiple rows with different partitioning keys may become distributed if data placement changes. Many partitioned database systems, including H-Store, require that distributed transactions are few to achieve (almost) linear scalability.

Although P-Store is implemented in a main memory setting, the ideas presented here should be applicable to most partitioned OLTP DBMSs with some parameter changes.

4.3 Predictive Elasticity Algorithm

Our algorithm for proactive reconfiguration must determine when to reconfigure and how many machines to add or remove each time. This corresponds to finding a series of *moves*, where each move consists of adding or removing zero or more machines ("doing nothing" is a valid move).

Formally, a move is a reconfiguration going from B machines before to A machines after (adding $A-B$ machines on scale-out, removing $B-A$ machines on scale-in, or doing nothing if $A=B$). Each move has a specified starting and ending time. We will use the variables B and A and the notion of *move* throughout the paper.

At a high level, our algorithm adapts to a constantly changing load by planning a series of moves from the present moment to a specified time in the future. For simplicity, we discretize that period of time into T time intervals. Each move therefore lasts some positive number of time intervals (rounded up to the nearest integer). Figure 3 shows an example of the high level goal of the algorithm. In this schematic, P-Store predicts the load $T=9$ time intervals ahead, and the goal is to find a series of moves starting at $B=2$ machines at $t=0$ and ending at $A=4$ machines at $t=9$, such that capacity exceeds demand and cost is minimized. Minimizing cost requires scale-out moves to be delayed as much as possible. However, they also must be started as early as necessary to ensure sufficient resources to migrate data without disrupting the regular database workload. The algorithm has three functions: *best-moves*, *cost* and *sub-cost*, which we discuss next.

4.3.1 Top-Level Algorithm. The *best-moves* function in Algorithm 1 is the top-level algorithm to find the optimal sequence of moves. It receives as input a time-series array of predicted load L of length T (generated by P-Store's online predictor component, the topic of Section 5), as well as N_0, the number of machines allocated at time $t=0$, and the target average transaction rate per node Q from Section 4.1. The output of the algorithm is an optimal sequence M of contiguous, non-overlapping moves ordered by starting time.

Algorithm 1: Calculate best series of moves for given time-series array of predicted load L, starting with N_0 nodes.

```
1  Function best-moves(L, N0, Q)
       Input: Time-series array of predicted load L of length T, machines
              allocated initially N0, target avg. txn rate per node Q
       Output: Best series of moves M
       // Calculate the maximum number of machines ever
       // needed to serve the predicted load
2      Z ← max(⌈max(L)/Q⌉, N0);
3      for i ← 1 to Z do
           // Initialize matrix m to memoize cost and
              best
           // series of moves
4          m ← ∅;
5          if cost(T, i, L, N0, Z, m) ≠ ∞ then
6              t ← T; N ← i;
7              while t > 0 do
8                  Add (t, N) to M;
9                  t ← m[t, N].prev_time;
10                 N ← m[t, N].prev_nodes;
11             Reverse M;
12             return M;
       // No feasible solution
13     return ∅;
```

Algorithm 1 first calculates Z, the number of machines needed to serve the maximum predicted load in L (Line 2). Next, Algorithm 1 iteratively tries to find a *feasible* series of moves ending with as few machines as possible, starting with $i=1$ and incrementing by one each time, with a maximum of Z (Lines 3 to 12). A sequence of moves is "feasible" if no server will ever be overloaded according to the load prediction L. The *cost* function returns the minimum cost of a feasible sequence of moves ending with i servers at time T (Line 5; the internals of the *cost* function will be discussed in the next section). If no feasible sequence exists, the function returns an infinite cost. Otherwise, the cost function populates a matrix m of size $T \times Z$ with the optimal moves it has found: $m[t,A]$ contains the last optimal move that results in having A servers at time t. The element $m[t,A]$ contains the time when the move starts, the initial number of servers for the move, and the cost during the move, i.e., the average number of servers used multiplied by the elapsed time.

As soon as Algorithm 1 finds a series of moves that is feasible (i.e., with finite cost), it works backwards through the matrix m to build a series of memoized best moves (Lines 6 to 10). Then, it reverses the list of moves so they correspond to moving forward through time, and it returns the reversed list (Lines 11 and 12). It is not necessary to continue with the loop after a feasible solution is found, because all later solutions will end up with more machines (i.e., higher cost).

If no feasible solution is found (Line 13), that means the initial number of machines N_0 is too low, and it is not possible to scale out fast enough to handle the predicted load. This can happen if, for example, there is a news event causing a flash crowd of customers on the site. There are a couple of options in this case: (1) reactively increase the data migration rate to meet capacity demands by moving larger chunks at a time (which will incur some latency overhead due to data migration), or (2) continue to move data at the regular rate and suffer some overhead due to insufficient capacity. By default, P-Store reacts to unexpected load spikes with option (2). The evaluation in Section 8.2 shows the performance of these two strategies.

4.3.2 Finding an Optimal Sequence of Moves. We now describe the *cost* and *sub-cost* functions. We first introduce some notation. To minimize the cost of the system over time T, we must find a series of moves spanning time T such that the predicted load never exceeds the effective capacity of the system, and the sum of the costs of the moves is minimized. In order to plan a move from B to A machines, we need to determine how long the move will take. The function $T(B,A)$ expresses this time, which depends on the specific reconfiguration strategy used by the system. We will discuss how to calculate $T(B,A)$ in Section 4.4.2. We also need to find out the moves that minimize cost. The cost of a move is computed by the function $C(B,A)$, which will be described in Section 4.4.3.

In order to determine the optimal series of moves, we have formulated the problem as a dynamic program. This problem is a good candidate for dynamic programming because it carries optimal substructure. The minimum cost of a series of moves ending with A machines at time t is equal to the minimum cost of a series of moves ending with B machines at time $t - T(B,A)$, plus the (minimal) cost of the last optimal move, $C(B,A)$.

This formulation is made precise in Algorithms 2 and 3. Algorithm 2 finds the cost of the optimal series of feasible moves ending at a given time t and number of machines A. Line 2 of Algorithm 2 checks the constraints of the problem, in particular that t must not be negative, and if $t = 0$ the number of machines must correspond to our initial state, N_0. It also checks that the predicted load at time t does not exceed the capacity of A machines. We assume that the cost of latency violations (see Section 3) is extremely high, so for simplicity, we define the cost of insufficient capacity to be infinite. Moving forward through the algorithm, recall that the matrix element $m[t,A]$ stores the last optimal move found by a call to *cost*. But its secondary purpose is for "memoization" to prevent redundant computation. Accordingly, Algorithm 2 checks to see if the optimal set of moves for this configuration has already been saved in m, and if so, it returns the corresponding cost (Lines 3 and 4). Finally, we come to the recurrence relation. The base case corresponds to $t = 0$, in which we simply return the cost of allocating A machines for one time interval (Lines 5 and 6). The recursive step is as follows: find

Algorithm 2: Find a feasible sequence of moves with minimum cost that ends with A nodes at time t. Memoize best moves in m.

```
1  Function cost(t, A, L, N0, Z, m)
       Input: Current time interval t, no. of nodes A, time-series array of
              predicted load L of length T, machines allocated initially
              N0, max. no. of machines available to allocate Z, matrix m
              of size T × Z to memoize cost and best series of moves
       Output: Min. cost of system after time t, ending with A nodes
       // penalty for constraint violation or
          insufficient capacity
2      if t < 0 or (t = 0 and A ≠ N0) or L[t] > cap(A) then return ∞;
3      if m[t,A] exists then                          /* check memoized cost */
4          return m[t,A].cost;
5      if t = 0 then                                          /* base case */
6          m[t,A].cost ← A;
7      else                                          /* recursive step */
8          B ← arg min_{1≤i≤Z}(sub-cost(t, i, A, L, N0, Z, m));
           // a move must last at least one time interval
9          if T(B,A) = 0 then T(B,A) ← 1;
10         m[t,A].cost ← sub-cost(t, B, A, L, N0, Z, m);
11         m[t,A].prev_time ← t − T(B,A);
12         m[t,A].prev_nodes ← B;
13     return m[t,A].cost;
```

the cost of the optimal series of moves ending with $B \rightarrow A$, for all B, and choose the minimum (Line 8). There is one caveat for the case when $B = A$ (the "do nothing" move). Since the time and cost of the move are both 0, we need to artificially make the move last for one time step, with a resulting cost of B (Line 9).

Algorithm 3 finds the cost of the optimal series of moves ending at a given time t with the final move from B to A machines. It first adjusts the time and cost of the move for the case when $B = A$, as described previously for Algorithm 2 (Line 2). Next it checks that the final move from $B \rightarrow A$ would not need to start in the past (Lines 3 to 5). Finally, it checks that for every time interval during the move from $B \rightarrow A$, the predicted load never exceeds the *effective capacity* of the system (`eff-cap`), which is the capacity of the system while a reconfiguration is ongoing (Lines 6 to 9). We will describe how to compute effective capacity in Section 4.4.4. If all of these checks succeed, it makes a recursive call to Algorithm 2 and returns the cost of the full series of moves (Line 10).

4.4 Characterizing Data Migrations

In order to find an optimal series of moves, the previous algorithms need to evaluate individual moves to find the optimal choice at different points in time. This section provides the tools to perform such an evaluation. There are four key questions that must be answered to determine the best move: (1) How to schedule data transfers in a move? (2) How long does a given reconfiguration take? (3) What is the cost of the system during reconfiguration? (4) What is the capacity of the system to execute transactions during reconfiguration?

The answers to the last three questions correspond to finding expressions for three functions used in Section 4.3, respectively: $T(B,A)$, $C(B,A)$, and `eff-cap`. We answer these questions next.

Algorithm 3: Calculate minimum cost of system after time t when the last move is from B to A nodes.

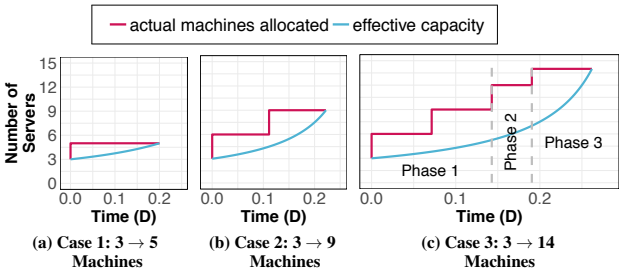
```
1  Function sub-cost(t, B, A, L, N_0, Z, m)
       Input: Current time interval t, no. of machines B before last move,
              no. of machines A after last move, time-series array of
              predicted load L of length T, machines allocated initially
              N_0, max. no. of machines available to allocate Z, matrix m
              of size T × Z to memoize cost and best series of moves
       Output: Min. cost of system after time t, with last move from B to
               A nodes
       // a move must last at least one time interval
2      if T(B,A) = 0 then T(B,A) ← 1, C(B,A) ← B;
3      start-move ← t − T(B,A);
4      if start-move < 0 then
           // this reconfiguration would need to start
           // in the past
5          return ∞;
6      for i ← 1 to T(B,A) do
7          load ← L[ start-move + i ];
8          if load > eff-cap(B, A, i/T(B,A)) then
               // penalty for insufficient capacity
               // during the move
9              return ∞;
10     return cost(start-move, B, L, N_0, Z, m) + C(B,A);
```

4.4.1 Executing a Move. In order to model moves it is necessary to understand the way they are executed. An important requirement is that at the beginning and end of every move, all servers always have the same amount of data. So initially B machines each have $1/B$ of the data, and at the end, A machines each have $1/A$ of the data. Since we consider (approximately) uniform workloads, spreading out the data evenly is best for load balancing. We enforce this invariant for each reconfiguration by sending an equal amount of data from every sender machine to every receiver machine.

Another important aspect in a move is the degree of parallelism that we can achieve. To minimize performance disruption, we limit each partition to transfer data with at most one other partition at a time. This means that the maximum number of parallel data transfers is limited to whichever is smaller: the number of sender partitions or the number of receiver partitions. Thus, the maximum number of parallel data transfers that can occur when scaling from B machines before to A machines after, with P partitions per machine is:

$$max_{\parallel} = \begin{cases} 0 & \text{if } B = A \\ P * min(B, A-B) & \text{if } B < A \\ P * min(A, B-A) & \text{if } B > A \end{cases} \quad (2)$$

We are now ready to describe how moves are performed. In the following exposition, we will consider the specific case of scale out, since the scale in case is symmetrical. For simplicity and without loss of generality, we will assume one partition per server. To minimize the time for each move, moves are scheduled such that the system makes full use of the maximum available parallelism given by Equation (2). In addition, moves add new servers as late as possible in order to minimize the cost while the move is ongoing.



(a) Case 1: $3 \rightarrow 5$ Machines (b) Case 2: $3 \rightarrow 9$ Machines (c) Case 3: $3 \rightarrow 14$ Machines

Figure 4: Servers allocated and effective capacity during migration, assuming one partition per server. Time in units of D, the time to migrate all data with a single thread.

Phase 1, Step 1	$1 \rightarrow 4, 2 \rightarrow 5, 3 \rightarrow 6$
	$1 \rightarrow 5, 2 \rightarrow 6, 3 \rightarrow 4$
	$1 \rightarrow 6, 2 \rightarrow 4, 3 \rightarrow 5$
Phase 1, Step 2	$1 \rightarrow 7, 2 \rightarrow 8, 3 \rightarrow 9$
	$1 \rightarrow 8, 2 \rightarrow 9, 3 \rightarrow 7$
	$1 \rightarrow 9, 2 \rightarrow 7, 3 \rightarrow 8$
Phase 2	$1 \rightarrow 10, 2 \rightarrow 11, 3 \rightarrow 12$
	$1 \rightarrow 11, 2 \rightarrow 12, 3 \rightarrow 10$
Phase 3	$1 \rightarrow 12, 2 \rightarrow 13, 3 \rightarrow 14$
	$1 \rightarrow 13, 2 \rightarrow 14, 3 \rightarrow 11$
	$1 \rightarrow 14, 2 \rightarrow 10, 3 \rightarrow 13$

Table 1: Schedule of parallel migrations when scaling from 3 machines to 14 machines.

When executing moves, there are three possible strategies that P-Store uses to maximize parallelism, exemplified in Figure 4. The first strategy is used when B is greater than or equal to the number of machines that need to be added (see Figure 4a). In this case, all new machines are added at once and receive data in parallel, while sender partitions rotate to send them data.

In the second case, the number of new machines is a perfect multiple of B, so blocks of B machines will be allocated at once and simultaneously filled. This allows for maximum parallel movement while also allowing for just-in-time allocation of machines that are not needed right away (see Figure 4b).

The third case is the most complex because the move is broken into three phases (see Figure 4c). The purpose of the three phases is to keep the sender partitions fully utilized throughout the whole reconfiguration, thus minimizing the length of the move. Table 1 reports all the sender-receiver pairs in the example of Figure 4c.

During the first phase in Figure 4c, servers are added in blocks of $B = 3$ at a time. Each of the original three servers sends data to every new server, in a round robin manner. During phase two, three new servers are added, bringing the total up to 12. During this phase, the sender servers send data to the new receivers, but they are filled only partly (see Table 1). By the end of phase two, each sender server has communicated with only two of the three new receiver servers. Finally, servers 13 and 14 are added during phase three. Because the previous three receiver servers were not completely filled in phase two, all the three sender servers can send data in parallel. This schedule enables the full reconfiguration to complete in the 11 rounds shown in Table 1, while minimizing overhead on each partition throughout. Without the three distinct phases, the reconfiguration shown would require at least 12 rounds.

4.4.2 Time for a Move. After detailing how a move is scheduled, we are ready to calculate the time $T(B,A)$ that it takes to move from B to A servers. Recall that D is the time it takes to move the entire database using a single thread, as defined in Section 4.1. We have discussed previously that moves are scheduled to always make full use of the maximum parallelism given by Equation (2). Therefore, the entire database can be moved in time $D/max_{\|}$. If we consider the actual fraction of the database that must be moved to scale from B machines to A machines, we obtain that the reconfiguration time is:

$$T(B,A) = \begin{cases} 0 & \text{if } B = A \\ \frac{D}{max_{\|}} * (1 - \frac{B}{A}) & \text{if } B < A \\ \frac{D}{max_{\|}} * (1 - \frac{A}{B}) & \text{if } B > A \end{cases} \quad (3)$$

4.4.3 Cost of a Move. As defined in Equation (1), cost depends on the number of machines allocated over time. Therefore, we define the cost of a reconfiguration as follows:

$$C(B,A) = T(B,A) * \texttt{avg-mach-alloc}(B,A) \quad (4)$$

where $T(B,A)$ is the time for reconfiguration from Equation (3) and $\texttt{avg-mach-alloc}(B,A)$ is the average number of machines allocated during migration. The full algorithm to find the latter is described in Appendix B. The algorithm generalizes the three cases discussed in Section 4.4.1, and calculates the average machines allocated based on which case $B \to A$ corresponds to.

4.4.4 Effective Capacity of System During Reconfiguration. Finally, we calculate the effective capacity of the system during a reconfiguration. The total capacity of N machines in which data is evenly distributed is defined as follows:

$$\texttt{cap}(N) = Q * N \quad (5)$$

During a reconfiguration, however, data is not evenly distributed. We refer to the effective capacity of the system during reconfiguration as `eff-cap`. Assume that a node n keeps a fraction f_n of the total database, where $0 \le f_n \le 1$. Since we consider (approximately) uniform workloads, node n receives a fraction f_n of the load, which is $\texttt{eff-cap} * f_n$ when the system is running at full capacity. The total load on the system cannot be so large that the target capacity Q of a server is exceeded, so we have that $\texttt{eff-cap} * f_n \le Q$ and:

$$\texttt{eff-cap} \le Q/f_n \quad \forall n \in \{n_1 \dots n_N\} \quad (6)$$

This implies that the server having the largest f_n, i.e., the largest fraction of the database, determines the maximum capacity of the system. We can thus define the effective capacity of the system after a fraction f of the data is moved during the transition from B machines to A machines as:

$$\texttt{eff-cap}(B,A,f) = \begin{cases} \texttt{cap}(B) & \text{if } B = A \\ \texttt{cap}(1/(\frac{1}{B} - f * (\frac{1}{B} - \frac{1}{A}))) & \text{if } B < A \\ \texttt{cap}(1/(\frac{1}{B} + f * (\frac{1}{A} - \frac{1}{B}))) & \text{if } B > A \end{cases} \quad (7)$$

Let us consider each case individually. The first case is simple: no data is moving. The second case applies to scaling out, where B nodes send data to $(A-B)$ new machines. Throughout reconfiguration, capacity is determined by the original B machines, which each initially have $1/B$ of the data. After reconfiguration, they will have $1/A$ of the data, so as fraction f of the data is moved to the new machines, each of the B machines now has $(1/B - f * (1/B - 1/A))$ of the data. The inverse of this expression corresponds to the number of machines in an evenly loaded cluster with equivalent capacity to the current system, and Equation (5) converts that machine count to capacity. The third case in Equation (7) applies to scaling in and follows a similar logic to the second case.

To illustrate why it is important to take the effective capacity into account when planning reconfigurations, Figure 4 shows the effective capacity at each point during the different migrations presented at the beginning of this section. For a small change such as Figure 4a, the effective capacity is close to the actual capacity, and it may not make a difference for planning purposes. But for a large reconfiguration such as Figure 4c, the effective capacity is significantly below the actual number of machines allocated. This fact must be taken into account when planning reconfigurations to avoid underprovisioning. Algorithm 3 performs this function in our Predictive Elasticity Algorithm.

5 LOAD TIME-SERIES PREDICTION

The decision about how and when to reconfigure requires an accurate prediction of the aggregate workload. This section describes the time-series techniques we use for accurately modeling and predicting the aggregate load on the system. As shown in Figure 1, B2W's traffic exhibits a strong diurnal pattern due to the customers' natural tendencies to shop during specific times of the day. However, we also find that there is variability on a day-to-day basis due to many factors, from seasonality of demand to occasional advertising campaigns. Capturing these short- and long-term patterns when modeling the load is critical for making accurate predictions. As previously discussed, database reconfiguration usually requires several minutes, so prediction of load changes must be done at that scale. To this end, we exploit *auto-regressive* (AR) prediction models, which are capable of capturing time-dependent correlations in the data. More precisely, we use *Sparse Periodic Auto-Regression* (SPAR) [7].

Informally, SPAR strives to infer the dependence of the load on long-term periodic patterns as well as short-term transient effects. Therefore, SPAR provides a good fit for the database workloads that P-Store is designed to serve, since the AR component captures the observed auto-correlations in the load intensity over time (e.g. due to diurnal trends), and the sparse-periodic component captures longer-term seasonal periods (e.g. due to weekly or monthly trends).

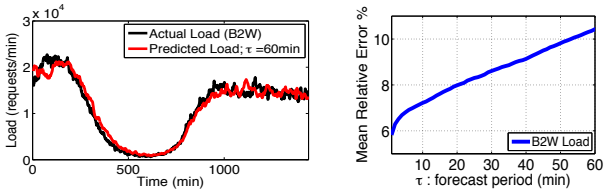
We now discuss fitting SPAR to the load data. We measure the load at time slot t by the number of requests per slot. Here each slot is 1 minute, so $T = 1440$ slots per day. In SPAR we model load at time $t + \tau$ based on the periodic signal at that time of the day and the offset between the load in the recent past and the expected load:

$$y(t+\tau) = \sum_{k=1}^{n} a_k y(t + \tau - kT) + \sum_{j=1}^{m} b_j \Delta y(t-j) \quad (8)$$

where n is the number of previous periods to consider, m is the number of recent load measurements to consider, a_k and b_j are parameters of the model, $0 \le \tau < T$ is a forecasting period (how long in the future we plan to predict) and

$$\Delta y(t-j) = y(t-j) - \frac{1}{n} \sum_{k=1}^{n} y(t - j - kT)$$

measures the offset of the load in the recent past to the expected load at that time of the day. Parameters a_k and b_j are inferred using linear least squares regression over the training dataset used to fit the model.



(a) 60-minute ahead SPAR predictions during a 24-hour period.

(b) Prediction accuracy vs. forecasting period τ.

Figure 5: Evaluation of SPAR's predictions for B2W.

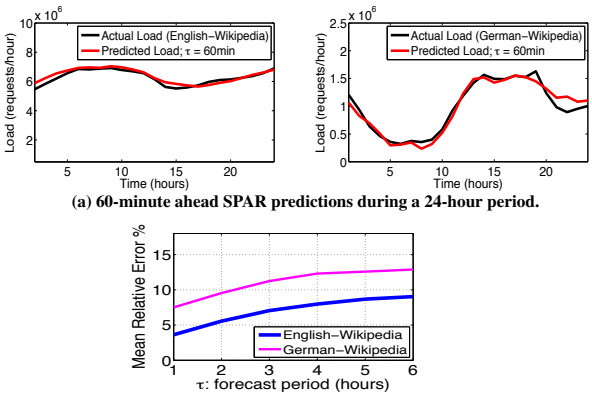
SPAR Predictions for B2W: We now analyze the utility of using SPAR to model and predict the aggregate load in several months' worth of B2W load traces (Section 7 provides more details about the data). We used the first 4 weeks of the data to train the SPAR model. After examining the quality of our predictor under different values for the number of previous periods n and the number of recent load measurements m, we find that setting $n = 7$ and $m = 30$ is a good fit for our dataset. This means that we use the previous week for the periodic prediction, and the offset from the previous 30 minutes to indicate how different the current load is from the "average" load at that time of day.

To demonstrate the accuracy of our SPAR predictor, Figure 5a depicts the actual B2W load and the SPAR predictions for a 24-hour period (outside of the training set), when using a forecasting window of $\tau = 60$ minutes. We also report the average prediction accuracy as a function of forecasting period τ in Figure 5b; the mean relative error (MRE) measures the deviation of the predictions from the actual data. We find that the prediction accuracy decays gracefully with the granularity of the forecasting period τ.

SPAR Predictions for Less Periodic Loads: To better understand if SPAR's high-quality predictions for B2W's periodic load can be obtained for other Internet workloads with less periodic patterns and varying degrees of predictability, we examined traces from *Wikipedia* for their per-hour page view statistics [14]. We focus on the page requests made to the two most popular editions, the English-language and German-language Wikipedias [33].

Similar to our analysis for B2W, we trained SPAR using 4 weeks of Wikipedia traces from July 2016 (separately for each language trace), then evaluated SPAR's predictions using data from August 2016. The results in Figures 6a and 6b show that SPAR is able to accurately model and predict the hourly Wikipedia load for both languages. Even for the less predictable German-language load, the error remains under 10% for predicting up to two hours into the future, and within only 13% for forecast windows as high as 6 hours.

Discussion: What is a Good Forecasting Window? Note that the forecast window τ only needs to be large enough for the first move returned by the dynamic programming algorithm described in Section 4 to be correct; by the time the first reconfiguration completes, the predictions may have changed, so the dynamic program must be re-run anyway. But in order for the first move to be correct, τ must be at least $2D/P$, the maximum length of time needed for *two* reconfigurations with parallel migration. This allows the algorithm to ensure that the first move will not leave the system in an under-provisioned state for subsequent moves (e.g., if it plans a "scale in" move, it knows there will be time to scale back out in advance of any predicted load spikes). This typically means a value of τ in the range of tens of minutes. Figures 5 and 6 show that SPAR is



(a) 60-minute ahead SPAR predictions during a 24-hour period.

(b) Prediction accuracy vs. forecasting period τ.

Figure 6: Evaluation of SPAR's predictions for another workload with different periodicity and predictability degrees: Wikipedia's per-hour page requests.

sufficiently accurate for such values of τ, with error rates under 10%.

We have explored other time-series models, such as a simple AR model and an auto-regressive moving-average (ARMA) model. Overall, we find that AR-based models work well, but that SPAR usually produces the most accurate predictions under different workloads (as it captures different trends in the data). For example, under τ = 60 minutes, the MRE for predicting the B2W load is 10.4%, 12.2%, and 12.5% under SPAR, ARMA, and AR, respectively.

6 PUTTING IT ALL TOGETHER

P-Store's techniques are general and can be applied to any partitioned DBMS, but our P-Store implementation is based on the open-source H-Store system [19]. It uses H-Store's system calls to obtain measurements of the aggregate load of the system. Data migrations are managed by H-Store's elasticity subsystem, Squall [11].

The P-Store system combines all the previous techniques for time-series prediction, determination of when to scale in or out, and how to schedule the migrations. We have created a "Predictive Controller" which handles online monitoring of the system and calls to the components that implement each of these techniques. For convenience, we call these components the Predictor, the Planner and the Scheduler, respectively.

P-Store has an active learning system. If training data exists, parameters a_k and b_j in Equation (8) can be learned offline. Otherwise, P-Store constantly monitors the system over time and can actively learn the parameter values. The Predictor component uses Equation (8) and the fitted parameter values to make online predictions based on current measurements of H-Store's aggregate load.

As soon as P-Store starts running, the Predictive Controller begins monitoring the system and measuring the load. When enough measurements are available, it makes a call to the Predictor, which returns a time series of future load predictions. The Controller passes this data to the Planner, which calculates the best series of moves.

Given the best series of moves from the Planner, the Controller throws away all but the first (similar to the idea of receding horizon control [24]). It passes this first move along with the current partition plan to the Scheduler, which generates a new partition plan in which all source machines send an equal amount of data to all destination

machines, as described in Section 4.4.1. This partition plan is then passed to the external Squall system to perform the migration.

If the Planner calls for a scale-in move, the Controller waits for three cycles of predictions from the Predictor to confirm the scale-in. If after three cycles the Planner still calls for a scale-in, then the move is executed. This heuristic prevents unnecessary reconfigurations that could cause latency spikes.

After a move is complete, the Controller repeats the cycle of prediction, planning, and migration. If at any time the Planner finds that there is no feasible solution to manage the load without disruption, the Scheduler is called to create a partition plan to scale out to the number of machines needed to handle the predicted spike, and Squall is called in one of two ways as described at the end of Section 4.3.1: move data faster and suffer some latency during migration, or move at the regular rate and wait longer to reach the desired capacity.

7 B2W DIGITAL WORKLOAD

B2W is the largest online retailer in Brazil. This study is based on a large, rich dataset that includes logs of every transaction on the B2W shopping cart, checkout and stock inventory databases over a period of several months. The transaction logs include the timestamp and the type of each transaction (e.g., GET, PUT, DELETE), as well as unique identifiers for the shopping carts, checkouts and stock items that were accessed.

Using B2W's Workload to Evaluate P-Store: To model B2W's workload in H-Store, we have implemented a benchmark driven by the company's traces. This benchmark includes nearly all the database operations required to run an online retail store, from adding and removing items in customers' shopping carts, to collecting payment data for checkout. More details about the benchmark are available in Appendix C. The cart and checkout databases have a predictable access pattern (recall Figure 1) due to the daily habits of B2W's customers. The stock database is more likely to be accessed by internal B2W processes, however, causing a spikey, unpredictable access pattern. There may be predictive models that would be appropriate for this spikey workload, but we leave such a study for future work. Our evaluation considers only data and transactions from the cart and checkout databases. This is consistent with the deployment used in production at B2W: the stock data is stored in a different database, on a different cluster of machines from the cart and checkout data. The business logic involving multiple data sources happens at the application layer, not at the database layer.

When replaying the original cart and checkout transactions, we make a couple of modifications to enable us to experimentally demonstrate our proactive elasticity algorithms with H-Store and Squall. First, we increase the transaction rate by 10× so we can experience the workload variability of a full day in just a few hours. This allows us to demonstrate the performance of P-Store over several days within a reasonable experimental timeframe. Second, we add a small delay in each transaction to artificially slow down execution. We do this because H-Store is much faster than the DBMS used by B2W and can easily handle even the accelerated workload with a single server. Slowing down execution allows us to demonstrate the effectiveness of P-Store by requiring multiple servers.

We train our prediction model using 4-weeks' worth of historical B2W data, stored in an analytic database system. The SPAR parameters a_k and b_j from Equation (8) are calculated offline using the training data and can be easily updated online periodically. In our experiments, we found that updating these parameters once per week is usually sufficient. The remaining parameters in SPAR are updated online based on current load information extracted periodically from H-Store.

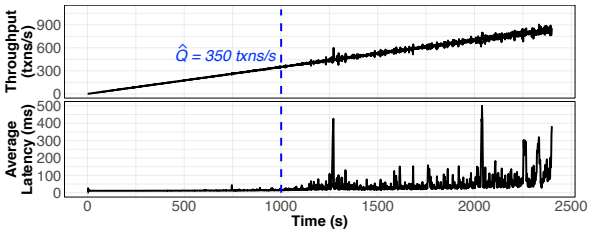


Figure 7: Increasing throughput on a single machine. Blue line indicates maximum throughput $\hat{Q}$.

8 EVALUATION

To evaluate P-Store we run the B2W benchmark described in the previous section. All of our experiments are conducted on an H-Store database with 6 partitions per node deployed on a 10-node cluster running Ubuntu 12.04 (64-bit Linux 3.2.0), connected by a 10 Gbps switch. Each machine has four 8-core Intel Xeon E7-4830 processors running at 2.13 GHz with 256 GB of DRAM.

8.1 Parameter Discovery

Before running P-Store, we need to understand certain properties of the B2W workload and its performance on our system. In particular, we must confirm the workload is close to uniform, and we must determine the target and maximum throughput per machine Q and $\hat{Q}$ and time to migrate the whole database D as described in Section 4.1.

In the B2W workload, each shopping cart and checkout key is randomly generated; so there is minimal skew in transactions accessing the cart and checkout databases. Furthermore, after hashing the keys to partitions with MurmurHash 2.0 [17], we found that the access pattern and data distribution are both relatively uniform across partitions. In particular, with 30 partitions over a 24-hour period, the most-accessed partition receives only 10.15% more accesses than average, and the standard deviation of accesses across all partitions is 2.62% of the average. The partition with the most data has only 0.185% more data than average, and the standard deviation is 0.099% of the average. This level of skew is not even close to the skew described in [27, 31], in which 40% or more of the transactions could be routed to a single partition. Therefore, the assumption that we have a uniform database workload is reasonable.

To discover the values for Q and $\hat{Q}$, we run a rate-limited version of the workload with a single server and identify the transaction rate at which the single server can no longer keep up. As shown in Figure 7, for the B2W workload running on an H-Store cluster with 6 partitions per server, this happens at 438 transactions per second. As described in Section 4.1, we set $\hat{Q}$ to 80% of this maximum, or 350 transactions per second. Q is set to 65% of the maximum, or 285 transactions per second.

To discover D, we run the following set of experiments: With an underlying workload of $\hat{Q}$ transactions per second, we start with the data on a single machine and move half of the data to a second machine, tracking the latency throughout migration. We perform this

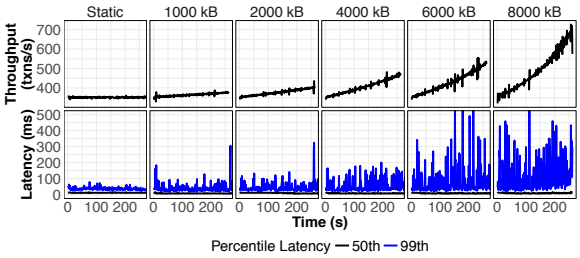


Figure 8: 50th and 99th percentile latencies when reconfiguring with different chunk sizes compared to a static system. Total throughput varies so per-machine throughput is fixed at $\hat{Q}$.

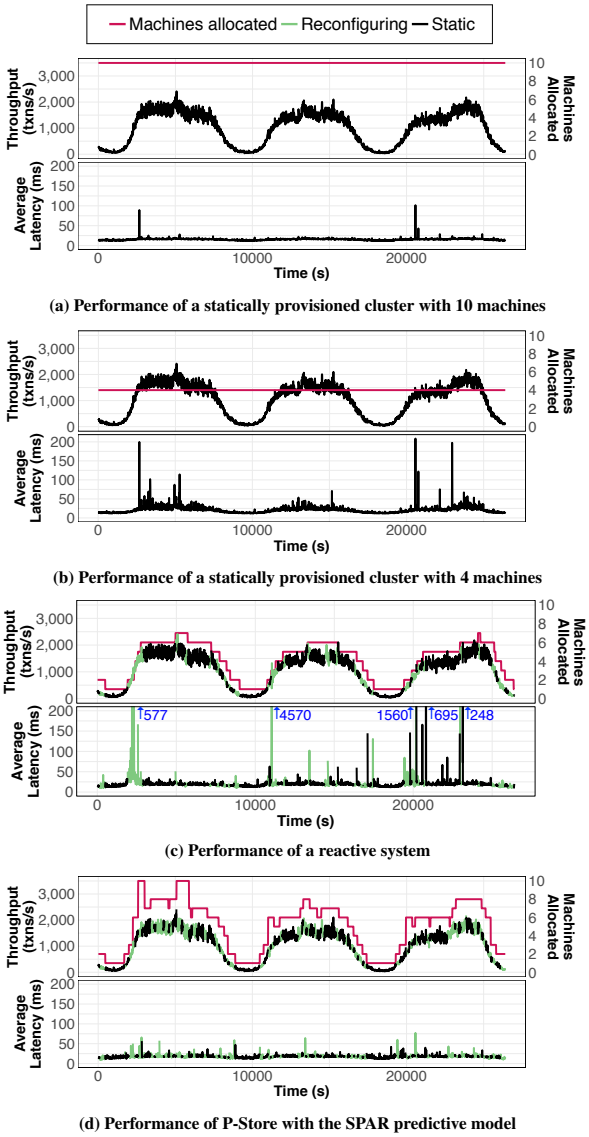
same experiment several times, varying the migration chunk size each time. We also vary the overall transaction rate to ensure the rate on the source machine stays fixed at $\hat{Q}$, even as data is moved. As shown in Figure 8, the 99th percentile latency when moving 1000 kB chunks is slightly larger than that of a static system with no reconfiguration, but still within the bounds of most acceptable latency thresholds. Moving larger chunks causes reconfiguration to finish faster, but creates a higher risk for latency spikes. In the 1000 kB experiment we moved one half of the entire 1106 MB database of active shopping carts and checkouts in 2112 seconds. Therefore, we set D to 4646 seconds (including the 10% buffer), or 77 minutes. We define the *migration rate* R as the rate at which data is migrated in this setting, which is 244 kB per second[1]. Since P-Store actually performs parallel migration and a single migration never moves the entire database, most reconfigurations last between 2 and 7 minutes.

8.2 Comparison of Elasticity Approaches

In this section, we compare the performance and resource utilization of several different elasticity approaches. Unless otherwise noted, all experiments are run with the B2W benchmark replaying transactions from a randomly chosen 3-day period, which happened to fall in July 2016. With a 10× speedup, this corresponds to 7.2 hours of benchmark time per experiment. For visual clarity, the charts show throughput and latency averaged over a 10 second window. To account for load prediction error, we inflate all predictions by 15%.

As a baseline for comparison, we run the benchmark on H-Store with no elasticity. If the cluster size is sufficient to manage the peak load comfortably, we would expect few high latency transactions but many idle servers during periods of low activity. Figure 9a shows this scenario when running the B2W benchmark on a 10-node cluster. Average latency is low, with only two small spikes during the first and third days. Presumably these spikes are caused by transient workload skew (e.g., one partition receives a large percentage of the requests over a short period of time). The red line at the top of the chart shows that with 10 machines allocated and a capacity per machine of $\hat{Q} = 350$ transactions per second, there is plenty of capacity for the offered load. If we reduce the number of servers to 4, the number of idle machines drops but the number of high latency transactions increases (Figure 9b). Companies like B2W cannot tolerate these latency spikes, and so provision for peak load.

[1] A data movement rate of 244 kB per second may seem low given a setting of 1000 kB chunks, but the reported chunk size is actually an upper bound; the actual size of most chunks is much smaller. Squall also spaces the chunks apart by at least 100 ms on average.



(a) Performance of a statically provisioned cluster with 10 machines

(b) Performance of a statically provisioned cluster with 4 machines

(c) Performance of a reactive system

(d) Performance of P-Store with the SPAR predictive model

Figure 9: Comparison of elasticity approaches.

We also run the benchmark with the reactive elasticity technique used by E-Store [31]. We choose E-Store over Clay [27] because in the B2W benchmark each transaction accesses only one partitioning key. Figure 9c shows the performance of this technique on the B2W workload. Light green sections of the throughput and latency curves indicate that a reconfiguration is in progress, while black sections indicate a period of no data movement. The red line shows the number of machines allocated at each point in time and the corresponding machine capacity (effective capacity is not shown, but it is close to the full machine capacity). This technique correctly reacts to the daily load variations and reconfigures the system as needed to meet demand. However, it leads to higher latency at the start of each load increase due to the overhead of reconfiguration at peak capacity.

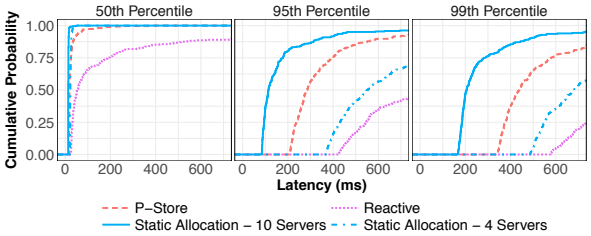


Figure 10: Comparison of elasticity approaches in terms of the top 1% of 50th, 95th and 99th percentile latencies.

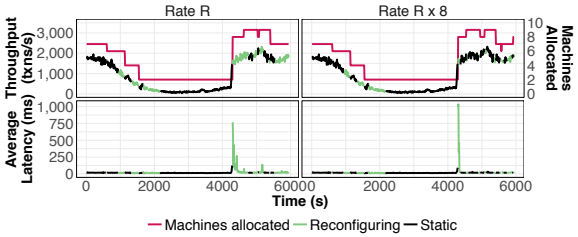


Figure 11: Comparison of two different rates of data movement when P-Store reacts to an unexpected load spike.

Elasticity Approach	# Latency Violations 50th %ile	95th %ile	99th %ile	Average Machines Allocated
Static allocation with 10 servers	0	13	25	10
Static allocation with 4 servers	0	157	249	4
Reactive provisioning	35	220	327	4.02
P-Store	0	37	92	5.05

Table 2: Comparison of elasticity approaches in terms of number of SLA violations for 50th, 95th and 99th percentile latency, and average machines allocated.

Finally, we show that P-Store comes closest to solving the problem outlined in Section 3. Figure 9d shows P-Store running on the B2W benchmark. We see many fewer latency spikes than the reactive experiment because P-Store reconfigures the system in advance of load increases and provides more headroom for transient load variations and skew (notice that the red line indicating machine capacity is always above the throughput curve).

Figure 10 compares the four different elasticity approaches studied in terms of CDFs of the top 1% of 50th, 95th and 99th percentile latencies measured each second during the experiments shown in Figure 9. Curves that are higher and far to the left are better, because that indicates that latency is generally low. The reactive approach clearly performs the worst in all three plots because it reconfigures at peak capacity, making latency spike. Although static allocation with four servers outperforms P-Store for 50th percentile latency, it is much worse for 95th and 99th percentile latencies. Static allocation with 10 servers performs best in all three plots.

Table 2 reports the number of SLA violations as well as the average number of machines allocated during the experiments shown in Figure 9. We define SLA violations as the total number of seconds during the experiment in which the 50th, 95th, or 99th percentile latency exceeds 500 ms, since that is the maximum delay that is unnoticeable by users [1]. Static allocation with 10 machines unsurprisingly has the fewest latency violations, but it also has at least 2× more machines allocated than all the other approaches. Furthermore, Section 8.3 will show that 10 servers are insufficient to handle load spikes such as those seen on Black Friday. Static allocation with 4 machines has the smallest number of machines allocated, but it has many SLA violations for the tail latencies. Reactive provisioning performs even worse, with 13× more 99th percentile latency violations than static allocation for peak load, because it reconfigures when the system is at peak capacity. P-Store performs well, using about 50% of the resources of peak provisioning, while causing about one third of the latency violations of reactive provisioning. P-Store has more latency violations than the peak-provisioned system because there is less capacity to handle transient workload skew, particularly when it coincides with data movement. This will be less of a problem when running at normal speed (as opposed to 10× speed), because the system will need to reconfigure less frequently. Users can also configure P-Store to be more conservative in terms of the target throughput per server Q. Section 8.3 will show how users can vary Q to prioritize cost or performance.

The predictive algorithms alone are sufficient as long as there are no unexpected load spikes. When the predictions are incorrect, however, P-Store must do one of the two options described in Section 4: continue scaling out at rate R, or reactively increase the rate of migration to scale out as fast as possible. Figure 11 compares these two different approaches in the presence of a large unexpected spike during a day in September 2016. When scaling at rate R, the numbers of latency violations for the 50th, 95th, and 99th percentile are 16, 101, and 143, respectively. When scaling at rate $R \times 8$, however, the numbers are 22, 44, and 51. Although the average latency at the start of the load spike is higher when scaling at rate $R \times 8$, the total number of seconds with latency violations is lower.

8.3 Simulation of Elasticity Approaches

It is not practical to run the B2W benchmark for longer than a few days; running the benchmark with just three days of data (as in Figure 9) requires at least 7.2 hours per experiment. Therefore, to compare the performance of the different allocation strategies and different parameter settings over a long period of time, we use simulation. We make use of the large amount of B2W data available by simulating running each strategy multiple times with different parameters over a four-and-a-half month period from August to December 2016. This period includes Black Friday as well as several other periods of increased load (e.g., due to periodic promotions or load testing), allowing for a more comprehensive comparison between allocation strategies. We use the discovered values of D and $\hat{Q}$ from Section 8.1 and we study the effect of varying the value of Q, which has the effect of increasing or decreasing the "buffer" between the actual load and the effective capacity provided by each allocation strategy. This in turn affects the total cost of the allocation, as well as the likelihood of being caught with insufficient capacity in case of an unexpected load spike.

Figure 12 shows the performance of each strategy in terms of the percentage of time with insufficient capacity and overall cost (calculated from Equation (1)). Each point represents a full simulation with a specific allocation strategy and value of Q. The simulations with settings corresponding to the four different benchmark runs from Figure 9 are annotated on the chart. The x-axis is shown with a

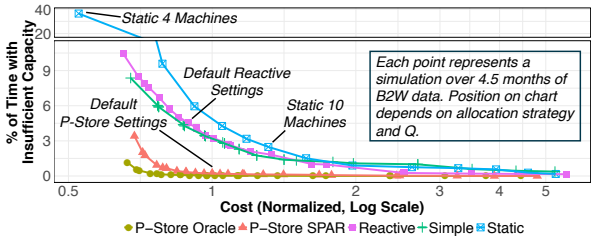


Figure 12: Performance of different allocation strategies and values of Q simulated over 4.5 months of B2W's load.

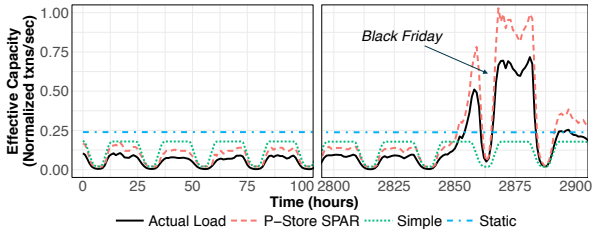


Figure 13: Actual load on B2W's DB and effective capacity of three allocation strategies simulated over two 4-day periods.

logarithmic scale and normalized to the cost of the P-Store simulation using default parameters (i.e., predictions are inflated by 15%, Q is set to 65% of the maximum throughput, etc.). Varying Q has the effect of prioritizing either sufficient capacity or lower cost, thus creating a "capacity-cost" curve for each allocation strategy.[2]

Figure 12 compares P-Store with SPAR to four other allocation strategies. "P-Store Oracle" shows the performance of P-Store given a perfect prediction. The percentage of time with insufficient capacity is not zero because the predictions are at the granularity of five minutes, and instantaneous load may have spikes. Of course "P-Store Oracle" is not possible in practice, but it shows the upper bound of P-Store's performance. "P-Store SPAR" is not far behind, and Figure 12 demonstrates that P-Store's default parameters achieve a good tradeoff between cost and capacity. The purple line shows a reactive allocation strategy, similar to the strategy used in Figure 9c. It is possible to limit capacity violations with a reactive approach by increasing the "buffer" of allocated machines, but this results in a higher-cost solution. The "Simple" strategy increases machines in the morning and decreases them at night. It seems like it could work (see the green dotted line in Figure 13, left), but it breaks down as soon as there is any deviation from the pattern (see Figure 13, right). Increasing the number of machines allocated each day reduces the time with insufficient capacity, but vastly increases the cost. The worst option is the "Static" approach (see Figures 9a, 9b and 13). Similar to the "Simple" model, it is inflexible and not resilient to large load spikes. In contrast, P-Store uses both predictive and reactive techniques to effectively handle the load surge on Black Friday (see Figure 13, right).

[2]Varying P-Store's prediction inflation parameter has the same effect as varying Q since both parameters affect the "buffer" between actual load and effective capacity. Both parameters affect the position along P-Store's capacity-cost curve shown in Figure 12.

9 RELATED WORK

This work follows on several previous papers on database elasticity. We have discussed reactive approaches such as Accordion [26], E-Store [31], and Clay [27] in Section 2. TIRAMOLA is an elastic system for NoSQL databases which models resize decisions as a Markov Decision Process [32]. Cumulus [13] is another project that, similar to Clay, attempts to minimize distributed transactions through adaptive repartitioning. It currently does not support elasticity.

Many recent papers have modeled cyclic workloads and load spikes for management of data centers, Infrastructure-as-a-Service cloud systems, and web applications [15, 16, 23, 28, 29, 34]. Many of the systems described are elastic and include a control-loop for proactively provisioning resources in advance of load increases. The model for most of these systems is that servers and other cloud resources have some amount of fixed initialization cost, but once initialized they are available to serve requests at full capacity. We study a more complex model of proactive provisioning specific to shared nothing databases, in which the effective capacity of newly allocated servers is limited by the speed of data re-distribution.

There has been some recent work on modeling workloads for elastically scaling databases [10], but it has focused on long-term growth for scientific databases rather than cyclic OLTP workloads. Holze et al. model cyclic database workloads and predict workload changes [18], but they do not use these models to proactively reconfigure the database. PerfEnforce [25] predicts the amount of computing resources needed to meet SLAs for a particular OLAP query workload. It does not take into account the time to scale out to the new configuration, nor the impact on query performance during scaling. ShuttleDB implements predictive elasticity for a multi-tenant Database-as-a-Service system, but unlike our system it only moves entire databases or VMs [4]. To our knowledge, there is no other system that solves the problem that P-Store addresses: proactive scaling for a distributed, highly-available OLTP database.

In this paper, we use Squall [11] for database migration. There are other approaches to live database migration [3, 8, 9, 12], all focusing on reducing the overhead of migration while maintaining ACID guarantees. We show that with careful performance tuning, it is possible to virtually eliminate overhead for certain workloads.

10 CONCLUSION

This paper presented P-Store, a novel database system that uses predictive modeling to elastically reconfigure the database *before* load spikes occur. We defined the problem that P-Store seeks to solve: how to reduce costs by deciding when and how to reconfigure the database. To explain how P-Store solves this problem, we described a novel dynamic programming algorithm for scheduling reconfigurations. We presented a time-series model that can accurately predict the load for different applications. Finally, we tested the end-to-end system by running a real online retail workload in H-Store and using our predictive models to decide when and how to reconfigure, thus demonstrating the cost savings that can be achieved with P-Store.

Overall, P-Store improves on existing work on database elasticity by scaling proactively rather than reactively, but it does not manage skew as E-Store and Clay do. Future work should investigate combining these ideas to build a system which uses predictive modeling for proactive reconfiguration, but also manages skew.

REFERENCES

[1] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. 2014. Impact of Response Latency on User Behavior in Web Search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. 103–112.

[2] B2W Digital 2017. B2W Digital. https://www.b2wdigital.com. (2017).

[3] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. 2012. Cut Me Some Slack: Latency-Aware Live Migration for Databases. In *Proceedings of the 15th international conference on extending database technology*. 432–443.

[4] Sean Kenneth Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. 2014. ShuttleDB: Database-Aware Elasticity in the Cloud. In *IEEE International Conference on Autonomic Computing*. 33–43.

[5] Deborah Barnes and Vijay Mookerjee. 2009. Customer delay in e-Commerce sites: Design and strategic implications. *Business Computing* 3 (2009), 117.

[6] Jake Brutlag. 2009. Speed Matters for Google Web Search. https://services.google.com/fh/files/blogs/google_delayexp.pdf. (2009). [Online; accessed: 16-Mar-2017].

[7] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. 2008. Energy-aware Server Provisioning and Load Dispatching for Connection-intensive Internet Services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 337–350.

[8] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud. *ACM Transactions on Database Systems* 38, 1 (2013), 5:1–5:45.

[9] 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 (2011), 494–505.

[10] Jennie Duggan and Michael Stonebraker. 2014. Incremental Elasticity for Array Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 409–420.

[11] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 299–313.

[12] Aaron J Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 301–312.

[13] Ilir Fetai, Damian Murezzan, and Heiko Schuldt. 2015. Workload-Driven Adaptive Data Partitioning and Distribution - The Cumulus Approach. In *IEEE International Conference on Big Data*. 1688–1697.

[14] Wikimedia Foundation. 2017. Wikipedia page view statistics. https://dumps.wikimedia.org/other/pagecounts-raw. (2017).

[15] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. 2007. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *Proceedings of the IEEE 10th International Symposium on Workload Characterization*. 171–180.

[16] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. Press: Predictive Elastic Resource Scaling for Cloud Systems. In *Proceedings of the IEEE International Conference on Network and Service Management*. 9–16.

[17] V Holub. 2010. Java implementation of MurmurHash. https://github.com/tnm/murmurhash-java. (2010). [Online; accessed: 29-Mar-2017].

[18] Marc Holze, Ali Haschimi, and Norbert Ritter. 2010. Towards Workload-Aware Self-Management: Predicting Significant Workload Shifts. In *Proceedings of the IEEE 26th International Conference on Data Engineering Workshops*. 111–116.

[19] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[20] Kissmetrics. 2017. How Loading Time Affects Your Bottom Line. https://blog.kissmetrics.com/loading-time/?wide=1. (2017). [Online; accessed: 28-Feb-2017].

[21] Greg Linden. 2006. Make Data Useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf. (2006). [Online; accessed: 28-Feb-2017].

[22] Greg Linden. 2006. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. (2006). [Online; accessed: 16-Mar-2017].

[23] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. 2012. Renewable and Cooling Aware Workload Management for Sustainable Data Centers. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 175–186.

[24] David Q Mayne and Hannah Michalska. 1990. Receding Horizon Control of Nonlinear Systems. *IEEE Trans. Automat. Control* 35, 7 (1990), 814–824.

[25] Jennifer Ortiz, Brendan Lee, and Magdalena Balazinska. 2016. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *Proceedings of the 2016 International Conference on Management of Data*. 2141–2144.

[26] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1035–1046.

[27] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[28] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 5:1–5:14.

[29] Matthew Sladescu. 2015. *Proactive Event Aware Cloud Elasticity Control*. Ph.D. Dissertation. University of Sydney.

[30] Rebecca Taft. 2017. B2W Benchmark in H-Store. https://github.com/rytaft/h-store/tree/b2w/src/benchmarks/edu/mit/benchmark/b2w. (2017).

[31] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment* 8, 3 (2014), 245–256.

[32] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris. 2013. Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 34–41.

[33] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Computer Networks* 53, 11 (2009), 1830–1845.

[34] Michail Vlachos, Christopher Meek, Zografoula Vagena, and Dimitrios Gunopulos. 2004. Identifying Similarities, Periodicities and Bursts for Online Search Queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 131–142.

Symbol	Definition
C	Cost of a DBMS cluster over T time intervals
T	Number of time intervals considered in calculation of C
s_t	Number of servers in the database cluster at time t
Q	The target average throughput of a single database server
$\hat{Q}$	The maximum throughput of a single database server
D	The time needed to move all data in the database once with a single thread
R	The rate at which data must be migrated to move the entire database in time D
B	Number of servers before a reconfiguration
A	Number of servers after a reconfiguration
move	A reconfiguration from A to B servers
L	Time-series array of predicted load of of length T
N_0	Number of nodes allocated at the start of Algorithm 1
Z	The maximum number of machines needed to serve the predicted load in L
m	A matrix to memoize the cost and best series of moves calculated by Algorithm 2
M	The sequence of moves returned by Algorithm 1
`cap`(N)	Returns the maximum capacity of N servers
$T(B,A)$	Returns the time for a reconfiguration from B to A servers
$C(B,A)$	Returns the cost of a reconfiguration from B to A servers
`eff-cap`(B,A,f)	Returns the effective capacity of the DBMS after fraction f of the data has been moved when reconfiguring from B to A servers
P	The number of partitions per server
$max_{\parallel}$	The maximum number of parallel migrations during a reconfiguration
`avg-mach-alloc`(B,A)	Returns the average number of machines allocated when reconfiguring from B to A servers
s	Minimum of B and A
l	Maximum of B and A
Δ	The difference between s and l
r	The remainder of dividing Δ by s
f_n	The fraction of the database hosted by node n
f	The fraction moved so far of the total data moving during a reconfiguration
τ	The SPAR forecasting window
a_k	SPAR coefficient for periodic load
b_j	SPAR coefficient for recent load
$y(t+\tau)$	SPAR forecasted load at time $t+\tau$
n	The number of previous periods considered by SPAR
m	The number of recent load measurements considered by SPAR

Table 3: Symbols and definitions used throughout the paper.

A SYMBOLS USED THROUGHOUT PAPER

For ease of reference, we provide in Table 3 a list of the symbols used throughout the paper in the order they appear.

B AVERAGE MACHINES ALLOCATED DURING RECONFIGURATION

As described in Section 4.4.3, the expected cost of a reconfiguration from B to A is equal to the time for reconfiguration, $T(B,A)$, multiplied by the average number of machines allocated, `avg-mach-alloc`(B,A). The full algorithm for `avg-mach-alloc`(B,A) is presented in Algorithm 4.

Algorithm 4: Calculate the average number of machines that must be allocated during the move from B to A machines with parallel migration

```
1  Function avg-mach-alloc(B, A)
       Input: Machines before move B, machines after move A
       Output: Average number of machines allocated during the move
       // Machine allocation symmetric for scale-in and
       // scale-out. Important distinction is not
       // before/after but larger/smaller.
2      l ← max(B,A) ;          // larger cluster
3      s ← min(B,A) ;          // smaller cluster
4      Δ ← l − s ;             // delta
5      r ← Δ%s ;               // remainder
       // ================================================
       // Case 1: All machines added or removed at once
       // ================================================
6      if s ≥ Δ then return l;
       // ================================================
       // Case 2: Δ is multiple of smaller cluster
       // ================================================
7      if r = 0 then return (2s+l)/2;
       // ================================================
       // Case 3: Machines added or removed in 3 phases
       // ================================================
       // Phase 1: N1 sets of s machines added and
       // filled completely
8      N1 ← ⌊Δ/s⌋ − 1 ;        // number of steps in phase1
9      T1 ← s/Δ ;              // time per step in phase1
10     M1 ← (s+l− r)/2 ;       // average machines in phase1
11     phase1 ← N1 ∗ T1 ∗ M1;
       // Phase 2: s machines added and filled r/s
       // fraction of the way
12     T2 ← r /Δ ;             // time for phase2
13     M2 ← l− r ;             // machines in phase2
14     phase2 ← T2 ∗ M2;
       // Phase 3: r machines added and remaining
       // machines filled completely
15     T3 ← s/Δ ;              // time for phase3
16     M3 ← l ;                // machines in phase3
17     phase3 ← T3 ∗ M3;
18     return phase1 + phase2 + phase3
```

Algorithm 4 takes into consideration that machine allocation is symmetric for scale-in and scale-out. The important distinction between the starting and ending cluster sizes, therefore, is not before/after but larger/smaller. And the delta between the larger and smaller clusters is equal to the number of machines receiving data from the smaller cluster when scaling out, or the number of machines sending data to the smaller cluster when scaling in. These values are assigned to l, s and Δ in Lines 2 to 4 of Algorithm 4. Line 5 assigns to r the remainder of dividing Δ by s, which will be important later in the algorithm.

Given these definitions, the algorithm considers the three cases discussed in Section 4.4.1. In the first case, the size of the smaller cluster is greater than or equal to Δ, which means that all new machines must be allocated (or de-allocated) at once in order to allow for maximum parallel movement (Line 6). In the second case, Δ is a perfect multiple of the smaller cluster, so blocks of s machines will be allocated (or deallocated) at once and simultaneously filled (or emptied). Thus, the average number of machines allocated is $(2s+l)/2$ (Line 7). In the third case we have three phases, and the calculation of the average number of machines is shown in Lines 8 to 18 of Algorithm 4.

C B2W BENCHMARK

This appendix provides more detail about the B2W Benchmark introduced in Section 7. The transaction logs from B2W include the timestamp and the type of each transaction (e.g., GET, PUT, DELETE), as well as unique identifiers for the shopping carts, checkouts and stock items that were accessed or modified. Since there is some important information not available in the logs (e.g., the contents of each shopping cart), we also use a dump of all of the B2W shopping carts, checkouts, and stock data from the last year. The data has been anonymized to eliminate any sensitive customer information, but otherwise it is identical to the data in production. Joining the unique identifiers from the log data with the keys in the database dump thus allows us to infer almost everything about each transaction, meaning we can effectively replay the transactions starting from any point in the logs. This allows us to run H-Store with the same workload running in B2W's production shopping cart, checkout and stock databases.

A simplified database is shown in Figure 14, and a list of the transactions is shown in Table 4. When a customer tries to add an item to their cart through the website, GetStockQuantity is called to see if the item is available, and if so, AddLineToCart is called to update the shopping cart. At checkout time, the system attempts to reserve each item in the cart, calling ReserveStock on each item. If a given item is no longer available, it is removed from the shopping cart and the customer is notified. The customer has a chance to review the final shopping cart before they agree to the purchase.

Although the data used in this work is proprietary to B2W, the H-Store benchmark containing the full database schema and transaction logic is not. The benchmark is open-source and available on GitHub for the community to use [30].

Stock Inventory

sku	description	available	reserved	purchased
123456	Harry Potter and the...	97	2	53
111111	Maytag front loadin...	43	0	13
...	...	...	...	...

Shopping Cart

cart_id	cust_id	timestamp
abcdef	000001	Aug 2, 2016 10:05:34
ababab	000002	Aug 5, 2016 11:12:13
...	...	...

Cart Lines

cart_id	sku	price
abcdef	123456	$10.99
abcdef	111111	$599.99
...	...	...

Checkout

cart_id	checkout_id	credit_card_no	expiration
abcdef	abcdefghi	1111111111111111	11/18
bcbcbc	bcbcbcdcdc	2222222222222222	07/17
...	...	...	...

Figure 14: Simplified database for the B2W H-Store benchmark

Transaction	Description
AddLineToCart	Add a new item to the shopping cart, create the cart if it doesn't exist yet
DeleteLineFromCart	Remove an item from the cart
GetCart	Retrieve items currently in the cart
DeleteCart	Delete the shopping cart
GetStock	Retrieve the stock inventory information
GetStockQuantity	Determine availability of an item
ReserveStock	Update the stock inventory to mark an item as reserved
PurchaseStock	Update the stock inventory to mark an item as purchased
CancelStockReservation	Cancel the stock reservation to make an item available again
CreateStockTransaction	Create a stock transaction indicating that an item in the cart has been reserved
ReserveCart	Mark the items in the shopping cart as reserved
GetStockTransaction	Retrieve the stock transaction
UpdateStockTransaction	Change the status of a stock transaction to mark it as purchased or cancelled
CreateCheckout	Start the checkout process
CreateCheckoutPayment	Add payment information to the checkout
AddLineToCheckout	Add a new item to the checkout object
DeleteLineFromCheckout	Remove an item from the checkout object
GetCheckout	Retrieve the checkout object
DeleteCheckout	Delete the checkout object

Table 4: Operations from the B2W H-Store benchmark