E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems

Rebecca Taft♦, Essam Mansour♣, Marco Serafini♣, Jennie Duggan★, Aaron J. Elmore▲
Ashraf Aboulnaga♣, Andrew Pavlo♠, Michael Stonebraker♦
♦MIT CSAIL, ♣Qatar Computing Research Institute, ★Northwestern University, ▲University of Chicago
♠Carnegie Mellon University
{rytaft, stonebraker}@csail.mit.edu,
{emansour, mserafini, aaboulnaga}@qf.org.qa, jennie.duggan@northwestern.edu,
aelmore@cs.uchicago.edu, pavlo@cs.cmu.edu

ABSTRACT

On-line transaction processing (OLTP) database management systems (DBMSs) often serve time-varying workloads due to daily, weekly or seasonal fluctuations in demand, or because of rapid growth in demand due to a company's business success. In addition, many OLTP workloads are heavily skewed to "hot" tuples or ranges of tuples. For example, the majority of NYSE volume involves only 40 stocks. To deal with such fluctuations, an OLTP DBMS needs to be elastic; that is, it must be able to expand and contract resources in response to load fluctuations and dynamically balance load as hot tuples vary over time.

This paper presents E-Store, an elastic partitioning framework for distributed OLTP DBMSs. It automatically scales resources in response to demand spikes, periodic events, and gradual changes in an application's workload. E-Store addresses localized bottlenecks through a two-tier data placement strategy: cold data is distributed in large chunks, while smaller ranges of hot tuples are assigned explicitly to individual nodes. This is in contrast to traditional single-tier hash and range partitioning strategies. Our experimental evaluation of E-Store shows the viability of our approach and its efficacy under variations in load across a cluster of machines. Compared to single-tier approaches, E-Store improves throughput by up to 130% while reducing latency by 80%.

1. INTRODUCTION

Many OLTP applications are subject to unpredictable variations in demand. This variability is especially prevalent in web-based services, which handle large numbers of requests whose volume may depend on factors such as the weather or social media trends. As such, it is important that a back-end DBMS be resilient to load spikes. For example, an e-commerce site may become overwhelmed during a holiday sale. Moreover, specific items within the database can suddenly become popular, such as when a review of a book on a TV show generates a deluge of orders in on-line bookstores.

Proceedings of the VLDB Endowment, Vol. 8, No. 3

Such application variability makes managing DBMS resources difficult, especially in virtualized, multi-tenant deployments [10]. Enterprises frequently provision "siloed" workloads for some multiple of their routine load, such as 5–10× the average demand. This leaves resources underutilized for a substantial fraction of the time. There is a desire in many enterprises to consolidate OLTP applications onto a smaller collection of powerful servers, whether using a public cloud platform or an internal cloud. This multi-tenancy promises to decrease over-provisioning for OLTP applications and introduce economies of scale such as shared personnel (e.g., system administrators). But unless the demand for these co-located applications is statistically independent, the net effect of multi-tenancy may be more extreme fluctuations in load.

To date, the way that administrators have dealt with changes in demand on an OLTP DBMS has been mostly a manual process. Too often it is a struggle to increase capacity and remove system bottlenecks faster than the DBMS load increases [11]. This is especially true for applications that require strong transaction guarantees without service interruptions. Part of the challenge is that OLTP applications can incur several types of workload skew that each require different solutions. Examples of these include:

Hot Spots: In many OLTP applications, the rate that transactions access certain individual tuples or small key ranges within a table is often skewed. For example, 40–60% of the volume on the New York Stock Exchange (NYSE) occurs on just 40 out of ∼4000 stocks [23]. This phenomenon also appears in social networks, such as Twitter, where celebrities and socialites have millions of followers that require several dedicated servers just to process their updates. The majority of the other users have only a few followers, and can be managed by a general pool of servers.

Time-Varying Skew: Multi-national customer support applications tend to exhibit a "follow the sun" cyclical workload. Here, workload demand shifts around the globe following daylight hours when most people are awake. This means that the load in any geographic area will resemble a sine wave over the course of a day. Time-dependent workloads may also have cyclic skew with other periodicities. For example, an on-line application to reserve camping sites will have seasonal variations in load, with summer months being much busier than winter months.

Load Spikes: A DBMS may incur short periods when the number of requests increases significantly over the normal expected volume. For example, the volume on the NYSE during the first and last ten minutes of the trading day is an order of magnitude higher than at other times. Such surges may be predictable, as in the NYSE

system, or the product of "one-shot" effects. One encyclopedia vendor experienced this problem when it put its content on the web and the initial flood of users after the announcement caused a huge load spike that crashed the service [13].

The Hockey Stick Effect: Web-based startups often see a "hockey stick" of traffic growth. When (and if) their application becomes popular, they will have an exponential increase in traffic that leads to crushing demand on its DBMS. This pitfall also impacts established companies when they roll out a new product.

Given these issues, it is imperative that an OLTP DBMS be *elastic*. That is, it must be able to adapt to unanticipated changes in an application's workload to ensure that application throughput and latency requirements are met, while continuing to preserve transactional ACID guarantees. It is this last part that makes this problem particularly challenging. NoSQL systems, such as Cassandra and Amazon's DynamoDB, are able to scale in/out a DBMS cluster easily because they do not support full SQL transactions.

An elastic OLTP DBMS, in addition to preserving ACID guarantees, must adapt to workload changes without manual intervention. This ensures that the system can react immediately to a change in the workload; if the DBMS has to wait for a human to perform corrective actions, the event that caused the problem may have passed.

We contend that the only way to mitigate all of the different skew types is to use fine-grained (i.e., tuple-level) partitioning. This allows the DBMS to identify sets of tuples that are most frequently accessed and assign adequate resources for their associated transactions. Previous work on dynamic scaling of OLTP DBMSs has only been able to migrate large chunks of data [22, 29]. If a significant volume of requests accesses only a few hot tuples, then such a one-level chunking strategy will fail if hot tuples are hashed to the same chunk or allocated to the same range. Also, prior research has often presumed that the database fits entirely on a single server [10] or that chunks can be replicated in a consistent manner [14].

In this paper, we present *E-Store*, a planning and reconfiguration system for shared-nothing, distributed DBMSs optimized for transactional workloads. Our main contribution is a comprehensive framework that addresses all of the issues discussed. Instead of monitoring and migrating data at the granularity of pre-defined chunks, E-Store dynamically alters chunks by extracting their *hot tuples*, which are considered as separate "singleton" chunks. Introducing this *two-tiered* approach combining fine-grained hot chunks and coarse-grained cold chunks is the main technical contribution enabling E-Store to reach its goals. E-Store supports automatic on-line hardware reprovisioning that enables a DBMS to move its tuples between existing nodes to break up hotspots, as well as to scale the size of the DBMS's cluster.

E-Store identifies skew using a suite of monitoring tools. First it identifies when load surpasses a threshold using a lightweight algorithm. When this occurs, a second monitoring component is triggered that is integrated in the DBMS to track tuple-level access patterns. This information is used in our novel two-tier data placement scheme that assigns tuples to nodes based on their access frequency. This approach first distributes hot tuples one at a time throughout the cluster. Then it allocates cold tuples in chunks, placing them to fill in the remaining capacity on each cluster node.

We integrated our framework into the H-Store DBMS [15], a distributed NewSQL system designed for OLTP workloads, and measured the system's performance using three benchmarks. We demonstrate that under skewed workloads, our framework improves throughput by up to 4× and reduces query latency by 10×.

The remainder of the paper is organized as follows. In Section 2, we first provide an overview of our target OLTP system and demonstrate the significance of skew in this context. We then present E-Store's high-level architecture in Section 3, followed by a detailed discussion of its components in Sections 4 and 5 and its experimental evaluation in Section 6. Section 7 summarizes related work and Section 8 concludes the paper.

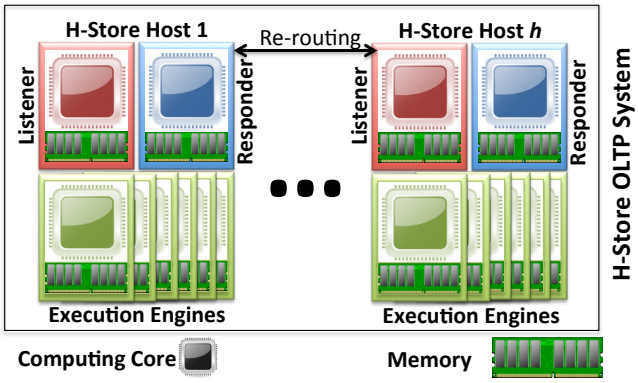


Figure 1: The H-Store Architecture.

2. BACKGROUND, MOTIVATION, AND APPLICABILITY OF RESULTS

We begin with an overview of the underlying architecture of H-Store and then provide a motivating example of the importance of fine-grained elasticity in distributed DBMSs. Lastly, we present our approach to elasticity and discuss the applicability of our results to other environments.

2.1 System Architecture

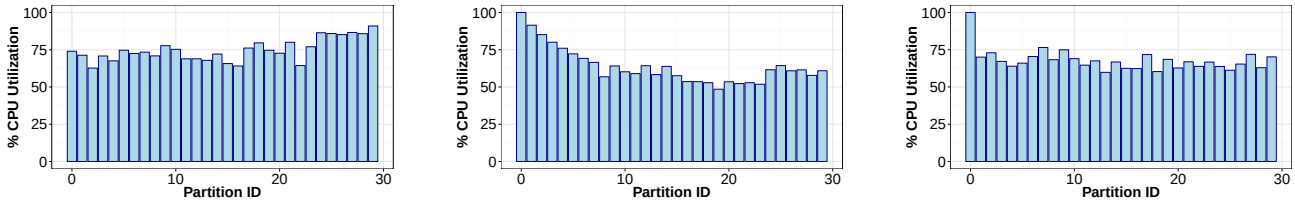
H-Store is a distributed, main-memory DBMS that runs on a cluster of shared-nothing compute nodes [18]. Fig. 1 illustrates the H-Store architecture. We define a DBMS instance as a cluster of two or more nodes deployed within the same administrative domain. A *node* is a single computer system that manages one or more data partitions. H-Store supports *replicating* tables on all servers, which is particularly useful for small read-only tables. In this work, however, we focus on horizontally partitioned tables, where the tuples of each table are allocated without redundancy to the various nodes managed by H-Store.

Each partition is assigned to a single-threaded *execution engine* that has exclusive access to the data at that partition. This engine is assigned to a single CPU core in its host node. When a transaction finishes execution, the engine can work on another transaction. Each node also contains a coordinator that allows its engines to communicate with the engines on other nodes.

H-Store supports ad-hoc queries but it is optimized to execute transactions as stored procedures. In this paper, we use the term *transaction* to refer to an invocation of a stored procedure. A stored procedure contains *control code* (i.e., application logic) that invokes pre-defined parameterized SQL commands. A client application initiates a transaction by sending a request (a stored procedure name and input parameters) to any node. Each transaction is *routed* to one or more partitions and their corresponding servers that contain the data accessed by a transaction.

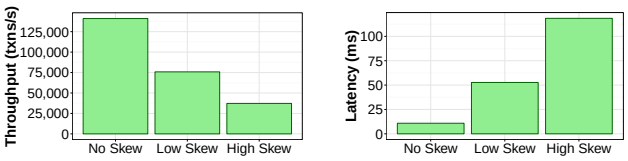
2.2 The Need for Elasticity

The demands of modern OLTP applications require the DBMS to maintain high throughput and low latency with near-continuous

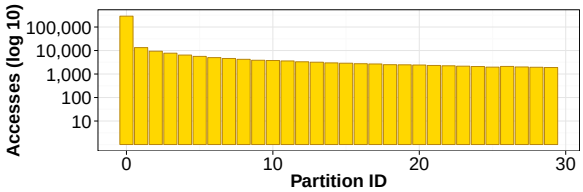


(a) No Skew **(b)** Low Skew **(c)** High Skew

Figure 2: Partition CPU utilization for the YCSB workload with varying amounts of skew. The database is split across five nodes, each with six partitions.



(a) Throughput **(b)** Latency



(c) Tuple Accesses per Partition with High Skew

Figure 3: Latency and throughput measurements for different YCSB workloads with varying amounts of skew. In Fig. 3c, we show the total tuple accesses per partition over a 10 second window for the **high skew** workload.

availability. This is challenging in the presence of skewed data access and changes in load. If an application's access pattern changes significantly, then some of the DBMS's nodes will become overloaded while others will be idle. This will degrade the system's performance despite potentially having sufficient total resources.

To illustrate the impact of skew on an OLTP DBMS, we conducted an initial experiment using the YCSB workload [2] on a five-node H-Store cluster. For this setup, we used a database with 60 million tuples that are each 1KB in size (∼60GB in total) that are deployed on 30 partitions (six per node). Additional details of our evaluation environment are described in Section 6. We then modified the YCSB workload generator to issue transaction requests with three access patterns:

1. **No Skew:** A baseline uniform distribution.
2. **Low Skew:** A Zipfian distribution where two-thirds of the accesses go to one-third of the tuples.
3. **High Skew:** The above Zipfian distribution applied to 40% of the accesses, combined with additional "hotspots", where the remaining 60% of the accesses go to 40 individual tuples in partition 0.

For each skew pattern, we ran the workload for ten minutes and report the average throughput and latency of the system. We also collected the average CPU utilization per partition in the cluster.

We see in Fig. 3 that the DBMS's performance degrades as the amount of skew in the workload increases: Fig. 3a shows that throughput decreases by 4× from the no-skew to high-skew workload, while Fig. 3b shows that latency increases by 10×. To help understand why this occurs, the graph in Fig. 3c shows the number of tuples that were accessed by transactions for the high-skew workload. We see that partition 0 executes more transactions than the other partitions. This means that the queue for that partition's engine is longer than others causing the higher average latency. Also, other partitions are idle for periods of time, thereby decreasing overall throughput.

This load imbalance is also evident in the CPU utilization of the partitions in the cluster. In Fig. 2, we see that the variation of CPU utilization among the 30 partitions increases proportionally to the amount of load skew. Again, for the high skew workload in Fig. 2c, partition 0 has the most utilization because it has the highest load.

2.3 Applicability

It is well known that transaction workloads that span multiple nodes execute slower than ones whose transactions access the data at a single node [26]. Hence, whenever possible an application designer should look for a database design that makes all transactions local to a single node. When this is not possible, the designer should make as few transactions as possible span multiple nodes.

Obviously any data rearrangement by E-Store could change the number of multi-node transactions. Therefore, E-Store must consider what data elements are accessed together by transactions when making decisions about data placement and load balancing. This presents a complex optimization environment. Hence, in this paper we focus on an important subset of the general case. Specifically, we assume all non-replicated tables of an OLTP database form a tree-schema based on foreign key relationships. Although this rules out graph-structured schemas and *m-n* relationships, it applies to many real-world OLTP applications [32].

A straightforward physical design for tree schemas is to partition tuples in the root node and then co-locate every descendant tuple with its parent tuple. We term this a *co-location* tuple allocation strategy. It is the best strategy as long as the majority of transactions access the schema tree via the root node and descend some portion of it during execution by following foreign key relationships. We term this access pattern *root-to-leaf order*. For example, in TPC-C, tuples of all non-replicated tables have a foreign key identifier that refers to a tuple in the `WAREHOUSE` table. Moreover, 90% of the transactions access the database in root-to-leaf order. As a result, partitioning tables based on their `WAREHOUSE` id and co-locating descendant tuples with their parent minimizes the number of multi-partition transactions.

In this paper, we assume that we start with a co-location allocation, and our elasticity problem is to find a second co-location allocation that balances the load and does not overload nodes. Next, we must migrate data without going off-line to move to the second allocation. The general case of graph schemas and general transactions is left for future work. For ease of explanation and without loss of generality, we will assume in the following that all the tables in the database have only one partitioning attribute.

Although the techniques discussed in this paper are implemented for H-Store, we claim that its speculative execution facilities [27] are competitive with other concurrency control schemes and that its command logging system has been shown to be superior to data logging schemes [19]. E-Store's elasticity techniques are generic

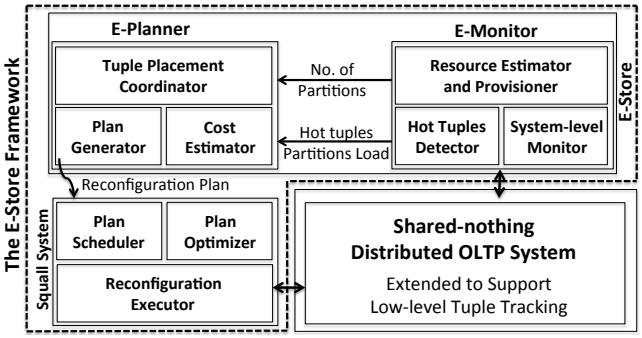


Figure 4: The E-Store Architecture.

and can be adapted to other shared-nothing DBMSs that use horizontal partitioning for tree structured schemas, whether or not the interface is through stored procedures.

3. THE E-STORE FRAMEWORK

To ensure high performance and availability, a distributed DBMS must react to changes in the workload and dynamically reprovision the database without incurring downtime. This problem can be broken into three parts:

1. How to identify load imbalance requiring data migration?
2. How to choose which data to move and where to place it?
3. How to physically migrate data between partitions?

The E-Store framework shown in Fig. 4 handles all three issues for OLTP applications. It is comprised of three components that are integrated with the DBMS. An overview of how E-Store rebalances a distributed DBMS is shown in Fig. 5. To detect load imbalance and identify the data causing it, the *E-Monitor* component communicates with the underlying OLTP DBMS to collect statistics about resource utilization and tuple accesses. This information is then passed to the *E-Planner* to decide whether there is a need to add or remove nodes and/or re-organize the data. The E-Planner generates a reconfiguration plan that seeks to maximize system performance after reconfiguration while also minimizing the total amount of data movement to limit migration overhead.

For physically moving data, E-Store leverages a migration system for H-Store, called *Squall* [8]. Squall uses the new reconfiguration plan generated by the E-Planner to decide how to physically move the data between partitions while the DBMS continues to execute transactions. This allows the DBMS to remain on-line during the reconfiguration with only minor degradation in performance.

We now describe how E-Store moves data across the cluster during a reorganization and its two-tier partitioning scheme that assigns data to partitions. We then discuss the details of the E-Monitor and E-Planner components in Sections 4 and 5, respectively.

3.1 Data Migration

The migration system for E-Store uses an updated version of the plan optimizer, scheduler, and executor from the Squall system [8]. Squall provides on-line reconfiguration for distributed DBMSs that can update the physical layout of partitioned data with minimal latency overhead and no system downtime. It is installed on every DBMS node in the cluster, and is able to directly interact with low-level scheduling in the system. To start a migration, Squall uses the reconfiguration plan generated by the E-Planner that includes a list of objects to move and the partition that will act as the leader during the process. This leader is responsible for coordinating reconfiguration state between partitions in the cluster and determining when the migration has completed.

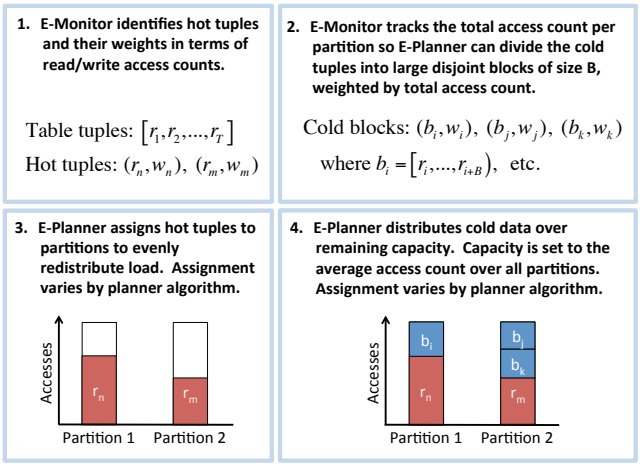


Figure 5: The steps of E-Store's migration process.

Since [8] was published, Squall has been updated to include an optimizer that decides the order that data is migrated. The optimizer splits the reconfiguration plan into sub-plans. In the case of applications with many partition keys, such as Voter [33] and YCSB, Squall calculates the ranges of keys that need to be moved and places the ranges that have the same source and destination partitions into the same sub-plan. For applications with fewer unique partition keys, this method generates sub-plans that move an excessive amount of data for each key. For example, in TPC-C moving a single `WAREHOUSE` id will end up moving many tuples because of our hierarchical co-location strategy. In this case, Squall further subdivides single-key ranges by using secondary and tertiary partitioning attributes, thereby limiting the amount of data moved in each sub-plan.

After producing the sub-plans, the optimizer prioritizes them based on which ones send data from the most overloaded partitions to the least overloaded partitions. This splitting ensures that overloaded partitions are relieved as quickly as possible. It also allows periods of idle time to be inserted between the execution of each sub-plan to allow transactions to be executed without the overhead of Squall's migrations. In this way, any transaction backlog is dissipated. We found that 100 sub-plans provided a good balance between limiting the duration of reconfiguration and limiting performance degradation for all of the workloads that we evaluated.

To execute a sub-plan, the leader first checks whether there is an ongoing reconfiguration. If not, it atomically initializes all partitions with the new plan. Each partition then switches into a special mode to manage the migration of tuples while also ensuring the correct execution of transactions during reconfiguration. During the migration, transactions may access data that is being moved. When a transaction (or local portion of a multi-partition transaction) arrives at a node, Squall checks whether it will access data that is moving in the current sub-plan. If the data is not local, then the transaction is routed to the destination partition or is restarted as a distributed transaction if the data resides on multiple partitions.

3.2 Two-Tiered Partitioning

Most distributed DBMSs use a *single-level* partitioning scheme whereby records in a data set are hash partitioned or range partitioned on a collection of keys [4, 26]. This approach cannot handle fine-grained hot spots, such as the NYSE example. If two heavily traded stocks hash to the same partition, it will be difficult to put them on separate nodes. Range partitioning also may not perform well since those two hot records could be near each other in the

sort order for range-partitioned keys. One could rely on a human to manually assign tuples to partitions, but identifying and correcting such scenarios in a timely manner is non-trivial [11].

To deal with such hot spots, E-Store uses a *two-tiered* partitioning scheme. It starts with an initial layout whereby root-level keys are range partitioned into blocks of size B and co-located with descendant tuples. We found that a block size of $B = 100,000$ keys worked well for a variety of workloads, and that is what we used in the Voter and YCSB experiments in this paper. For TPC-C, which has only a few root keys, we set $B = 1$.

Given this initial partitioning of keys, E-Store identifies a collection of k keys with high activity, where k is a user-defined parameter. For most workloads we found that setting k to the top 1% of keys accessed during a specified time window produced good results, as discussed in Section 6.2. These keys are extracted from their blocks and allocated to nodes individually. In short, we partition hot keys separately from cold ranges. The framework is illustrated in Fig. 5. While this approach works well with any number of root-level keys, workloads with a large number of root-level keys will benefit the most. Thus, our two-tiered partitioning scheme is more flexible than previous one-tiered approaches because it accommodates both hot keys and cold ranges.

4. ADAPTIVE PARTITION MONITORING

In order for E-Store's reorganization to be effective, it must know when the DBMS's performance becomes unbalanced due to hotspots, skew, or excessive load. The framework must also be able to identify the individual tuples that are causing hotspots so that it can update the database partitioning scheme.

A major challenge in continuous monitoring for high-performance OLTP DBMSs is the overhead of collecting and processing monitoring data. The system could examine transactions' access patterns based on recent log activity [31], but the delay from this off-line analysis would impact the timeliness of corrective action [11]. To eliminate this delay the system could monitor the usage of individual tuples in every transaction, but this level of monitoring is expensive and would significantly slow down execution.

To avoid this problem, E-Store uses a two-phase monitoring component called the *E-Monitor*. As shown in Fig. 4, E-Monitor is a standalone program running continuously outside of the DBMS. During normal operation, the system collects a small amount of data from each DBMS node using non-intrusive OS-level statistics [17]. Once an imbalance is detected, the E-Monitor component triggers per-tuple monitoring that is implemented directly inside of the DBMS. After a brief collection period, E-Monitor switches back to lightweight mode and sends the data collected during this phase to E-Planner to generate a migration plan for the DBMS. We now describe these two monitoring phases in more detail.

4.1 Phase 1: Collecting System Level Metrics

In the first phase, E-Monitor periodically collects OS-level metrics of the CPU utilization for each partition on the DBMS's nodes (each partition corresponds to a core). Such coarse-grained, high-level information about the system is inexpensive to obtain and still provides enough actionable data. Using CPU utilization in a main memory DBMS provides a good approximation of the system's overall performance. However, monitoring adherence to service-level agreements [9] would provide a better idea of application performance, and we are considering adding support for this in E-Store as future work.

When E-Monitor polls a node, it retrieves the current utilization for all the partitions at that node and computes the moving average over the last 60 seconds. E-Monitor uses two thresholds, a *high-watermark* (e.g., 90%) and a *low-watermark* (e.g., 50%), to control whether corrective action is needed. These thresholds are set by the DBA, based on a trade-off between system response time and desired resource utilization level. If a watermark is exceeded, E-Monitor triggers a phase of more detailed tuple-level monitoring.

4.2 Phase 2: Tuple-Level Monitoring

Once E-Monitor detects an imbalance, it starts the second phase of tuple-level monitoring on the entire cluster for a short period of time. The framework gathers information on the hot spots causing the imbalance to determine how best to redistribute data. Since E-Store focuses on tree-structured schemas and their co-location strategies, monitoring only the root tuples provides a good approximation of system activity and minimizes the overhead of this phase.

We define the hot tuples to be the top-k most frequently accessed tuples within the time window W. A tuple is counted as "accessed" if it is read, modified, or inserted by a transaction. For this discussion, let $\{r_1, r_2, \ldots, r_m\}$ be the set of all tuples (records) in the database and $\{p_1, p_2, \ldots, p_c\}$ be the set of partitions. For a partition p_j, let $L(p_j)$ denote the sum of tuple accesses for that partition and $TK(p_j)$ denote the set of the top-k most frequently accessed tuples. Thus, a tuple r_i is deemed "hot" if $r_i \in TK$.

When tuple-level monitoring is enabled, the DBMS initializes an internal histogram at each partition that maps a tuple's unique identifier to the number of times a transaction accessed that tuple. After the time window W has elapsed, the execution engine at each node assembles L and TK for its local partitions and sends them to E-Monitor. Once E-Monitor receives this information from all partitions, it generates a global top-k list. This list is used by E-Store's reprovisioning algorithms to build a reconfiguration plan. This monitoring process enables E-Monitor to collect statistics on all root-level tuples. The accesses that do not correspond to top-k tuples can be allocated to their correct blocks and summed to obtain block access frequencies.

The DBA should configure the monitoring time window for this phase to be the shortest amount of time needed to find hot tuples. The optimal value for W depends on the transaction rate and the access pattern distribution. Likewise, it is important to choose the right size for k so that enough tuples are identified as "hot." There is a trade-off between the accuracy in hot spot detection versus the additional overhead on an already overloaded system. We analyze the sensitivity of E-Store to both parameters in Section 6.2.

5. REPROVISIONING ALGORITHMS

After E-Monitor collects tuple-level access counts, E-Planner uses this data to generate a new partitioning scheme for the database. We now discuss several algorithms for automatically generating a two-level partitioning scheme. We first discuss how E-Planner decides whether to increase or decrease the number of nodes in the cluster. We then describe several strategies for generating new reconfiguration plans to reorganize the database.

All of the reconfiguration plans generated by E-Planner's algorithms begin by promoting any tuples that were newly identified as hot from block allocation to individual placement. Likewise, any tuple that was previously hot but is now identified as cold is demoted to the block allocation scheme. Then the new top-k hot tuples are allocated to nodes. Moving hot tuples between nodes requires little network bandwidth and can quickly alleviate load imbalances, so E-Planner performs these allocations first. If there is still a predicted load imbalance, E-Planner allocates cold blocks as a final step. Our algorithms currently do not take the amount of main memory into account when producing reconfiguration plans, and this is left as future work.

5.1 Scaling Cluster Size Up/Down

Before starting the reprovisioning process, E-Planner determines whether to maintain the DBMS's present cluster size or whether to add or remove nodes. One way to make this decision is by using the CPU utilization metrics collected during monitoring. If the average CPU utilization across the whole cluster exceeds the high-watermark, the framework can allocate new nodes and uses the extra partitions on these nodes in the placement algorithms. In the same way, if the average utilization is less than the low-watermark, it will decommission nodes. E-Store currently only supports changing the DBMS's cluster size by one node for each reconfiguration round.

5.2 Optimal Placement

We developed two different reprovisioning strategies derived from the well-known "bin packing" algorithm. Both of these approaches use an integer programming model to generate the optimal assignment of tuples to partitions. Since these algorithms take a long time, they are not practical for real-world deployments. Instead, they provide a baseline with which to compare the faster approximation algorithms that we present in the subsequent section.

We now describe our first bin packing algorithm that generates a two-tier placement where individual hot tuples are assigned to specific partitions and the rest of the "cold" data is assigned to partitions in blocks. We then present a simplified variant that only assigns blocks to partitions.

Two-Tiered Bin Packing: This algorithm begins with the current load (access count) on each partition and the list of hot tuples. The integer program has a decision variable for each possible partition-tuple assignment, and the constraints allow each hot tuple set to be assigned to exactly one partition. In addition, there is a decision variable for the partition assignment of each block of B cold tuples. The program calculates each partition's load by summing the access counts of its assigned hot and cold tuple sets. The final constraint specifies that each partition has an equal share of the load, $\pm\varepsilon$. Therefore, if A is the average load over all partitions, the resulting load on partition p_j must be in the range $A-\varepsilon \leq L(p_j) \leq A+\varepsilon$. The planner's objective function minimizes tuple movement while adhering to each partition's capacity constraints, thereby favoring plans with lower network bandwidth requirements.

For each potential assignment of a hot tuple r_i to partition p_j, there is a binary decision variable $x_{i,j} \in \{0,1\}$. Likewise, for each potential assignment of a cold tuple block b_k to partition p_j, there is a variable $y_{k,j} \in \{0,1\}$. In the following equations, we assume a database with n hot tuples, d cold tuple blocks, and c partitions. As defined in Section 4.2, $L(r_i)$ is the load on tuple r_i. Our first constraint requires that each hot tuple set is assigned to exactly one partition, so for each hot tuple set r_i,

$$\sum_{j=1}^{c} x_{i,j} = 1 \tag{1}$$

Likewise, for each cold block b_k,

$$\sum_{j=1}^{c} y_{k,j} = 1 \tag{2}$$

We seek a balanced load among the partitions, giving them a target load of $A \pm \varepsilon$, so for each partition p_j,

$$L(p_j) = \sum_{i=1}^{n}(x_{i,j} \times L(r_i)) + \sum_{k=1}^{d}(y_{k,j} \times L(b_k)) \geq A-\varepsilon \tag{3}$$

A second, similar constraint ensures that $L(p_j) \leq A+\varepsilon$. If a tuple is not assigned to its original partition according to the reconfiguration plan, it has a transmission cost of T. We assume that all machines in the cluster are located in the same data center, and therefore the transmission cost between any two partitions is the same. Thus, without loss of generality, we can set $T = 1$. We represent the transmission cost of assigning tuple r_i to partition p_j as a variable $t_{i,j} \in \{0, T\}$. Our objective function selects placements with reduced transmission overhead. Hence, it minimizes:

$$\sum_{i=1}^{n}\sum_{j=1}^{c}(x_{i,j} \times t_{i,j}) + \sum_{k=1}^{d}\sum_{j=1}^{c}(y_{k,j} \times t_{k,j} \times B) \tag{4}$$

Clearly, moving individual hot tuples is less expensive than transmitting blocks of B cold tuples.

One-Tiered Bin Packing: This is the same procedure as the 2-tiered algorithm but without using a list of hot tuples. Hence, rather than dividing the problem into hot and cold parts, all tuples are assigned using a single planning operation in which data is managed in blocks of size B. This scheme saves on monitoring costs as it does not require tuple tracking, but it is not able to generate a fine-grained partitioning scheme. One-tiered bin packing simulates traditional one-level partitioning schemes [26, 3]. This approach may perform well when data access skew is small, but it is unlikely to work in the presence of substantial skew.

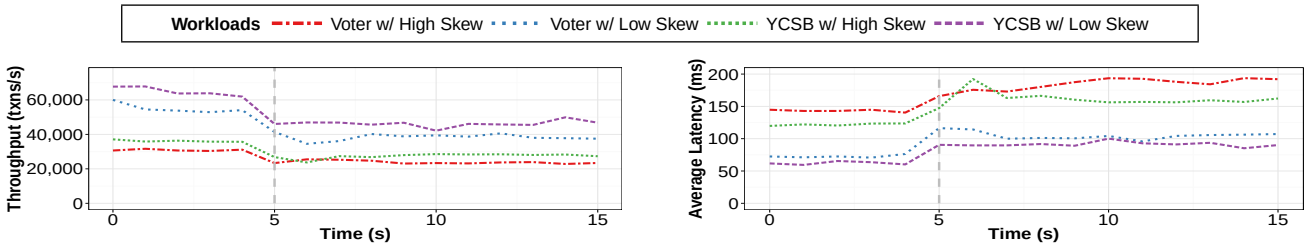
5.3 Approximate Placement

The bin packing algorithms provide a baseline for optimal reconfiguration of the database, but they are not practical for most applications. Because E-Store is intended for use in OLTP applications where performance is paramount, we set out to design algorithms capable of producing high quality partition plans in a much shorter timeframe. To this end, we implemented the following practical algorithms to assign hot tuples and cold blocks to partitions.

Greedy: This simple approach assigns hot tuples to partitions incrementally via locally optimal choices. It iterates through the list of hot tuples starting with the most frequently accessed one. If the partition currently holding this tuple has a load exceeding the average $A+\varepsilon$ as in Section 5.2, the Greedy algorithm sends the tuple to the least loaded partition. It continues to the next most popular tuple until all have been redistributed or no partitions have load exceeding $A+\varepsilon$. Although this algorithm operates in linear time, its usefulness is limited because this scheme only makes locally optimal decisions. It also does not move any blocks of cold tuples, which could impact its performance on workloads with lower levels of skew.

Greedy Extended: This algorithm first executes the Greedy algorithm for hot tuples. If one or more partitions are still overloaded after rebalancing, this scheme executes a similar operation with the cold blocks. One at a time (in order of decreasing load), each overloaded partition sends its hottest blocks of B cold tuples to the partition currently with the lowest load. This process repeats until all partitions have load within $A \pm \varepsilon$. The Greedy Extended algorithm's runtime is comparable to that of the standard Greedy algorithm.

First Fit: This approach globally repartitions the entire database using a heuristic that assigns tuples to partitions one at a time. It begins with the list of hot tuples sorted by their access frequency. The scheme places the hottest tuple at partition 0. It continues to add hot tuples to this partition until it has reached capacity, at which point the algorithm assigns tuples to partition 1. Once all the hot tuples have been placed, the algorithm assigns cold blocks to partitions, starting with the last partition receiving tuples. This approach favors collocating hot tuples and runs in constant time. In some circumstances it leads to better utilization of the DBMS's CPU caches,



(a) Throughput

(b) Latency

Figure 6: The impact of tuple-level monitoring on throughput and latency. Dashed lines at 5 seconds indicate the start of tuple-level monitoring.

because hot partitions serve fewer items. But it also makes the first partitions more vulnerable to overload because they are handling the hottest data. Moreover, because this algorithm does not make any attempt to minimize the movement of tuples during reconfiguration, the migration process may be expensive and cause temporary performance degradation.

6. EVALUATION

We now present our evaluation of the E-Store framework integrated with H-Store. We conducted an extensive set of experiments using large datasets and three different benchmarks to analyze the parameter sensitivity and performance of E-Store. We report our time-series results using a sliding-window average.

All of the experiments were conducted on a cluster of 10 Linux nodes connected by a 10Gb switch. Each node has two Intel Xeon quad-core processors running at 2.67GHz with 32GB of RAM. We used the latest version of H-Store with command logging enabled to write out transaction commit records to a 7200 RPM disk.

6.1 Benchmarks

We now describe the workloads that we used in our evaluation. For all three benchmarks, we examined three access patterns; no skew, low skew, and high skew.

Voter: This benchmark simulates a phone-based election application [33]. It is designed to saturate the DBMS with many short-lived transactions that all update a small number of records. The database consists of three tables. Two tables are read-only and replicated on all servers: they store the information related to contestants and map area codes to the corresponding locations. The third table stores the votes and it is partitioned; the telephone number of the voter is used as the partitioning attribute. An individual is only allowed to vote a fixed number of times. As mentioned above, we use three different types of skew: no skew, low skew, and high skew. Low skew simulates local interest in the contest, and is modeled by a Zipfian distribution where two-thirds of the accesses go to one-third of the tuples. High skew simulates highly localized interest where 30 phone numbers are responsible for attempting to vote 80% of the time. The remaining 20% of votes follow the Zipfian distribution described above. The 30 hot phone numbers will use up their allowed votes, but their continued effort to vote will strain database resources on their partition.

YSCB: The Yahoo! Cloud Serving Benchmark has been developed to test key-value data stores [2]. It consists of a single table partitioned on its primary key. In our experiments, we configured the YCSB workload generator to execute 85% read-only transactions and 15% update transactions, with no scans, deletes or inserts. Since Voter is write-heavy, we ran YCSB with a read-bias for balance. We used a database with 60 million tuples that are each 1KB (∼60GB in total). Again we ran no skew, low skew and high skew cases, using the definitions from Section 2.2.

TPC-C: This is an industry-standard benchmark for OLTP applications that simulates the operation of a wholesale parts-supply company [35]. The company's operation is centered around a set of warehouses that each stock up to 100,000 different items. Each warehouse has ten districts, and each district serves 3000 customers.

For these experiments, we ran TPC-C on a database with 100 warehouses. We again tested three different skew settings. For low-skew, we used a Zipfian distribution where two-thirds of the accesses go to one-third of the warehouses. For the high-skew trials, we modified the distribution such that 40% of accesses follow the Zipfian distribution described above, and the remaining 60% of accesses target three warehouses located initially on partition 0. As discussed in Section 2.3, 90% of the time customers can be served by their home warehouse, so if the tables are partitioned by their `WAREHOUSE` id, at most 10% of the transactions will be multi-partitioned [26].

6.2 Parameter Sensitivity Analysis

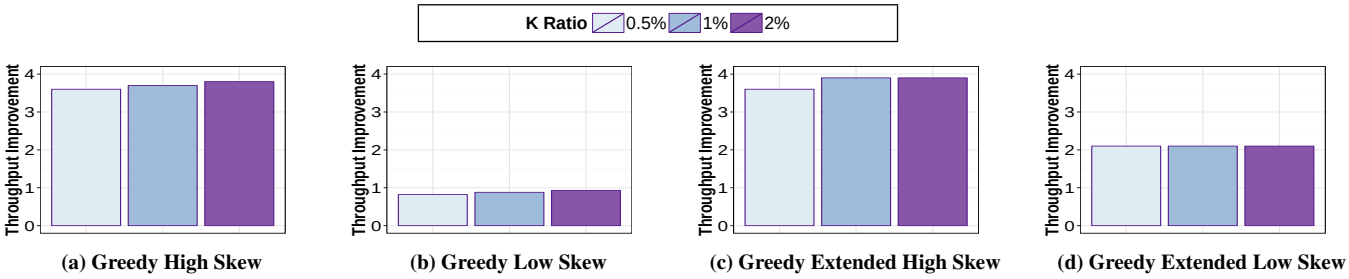
Once E-Store decides that a reconfiguration is needed, it turns on tuple-level monitoring for a short time window to find the top-k list of hot tuples. We analyzed the E-Store performance degradation in terms of throughput and latency due to this tracking. In each trial, we first executed the target workload for 60 seconds to let the system warm-up. We then collected the throughput and latency measurements. After five seconds, we enabled tuple-level monitoring with the top-k percentage of tracked tuples set to 1%. The results in Fig. 6 show that the monitoring reduces throughput by ∼25% for the high skew workload and ∼33% in the case of low skew. Moreover, the latency increases by about 45% in the case of low skew and about 28% in the case of high skew.

We next analyzed the sensitivity of the monitoring time window W and top-k ratio parameters. Fig. 7 shows the throughput improvement ratio (throughput after reconfiguration divided by throughput before reconfiguration) for the Greedy and Greedy Extended planners with time windows of variable length. The figure shows that the Greedy Extended algorithm is not sensitive to variation in the length of the time window. In contrast, the Greedy algorithm shows some sensitivity to the length of the time window since it is more dependent on the accuracy of the detected hot tuples set. Note that our measure of throughput after reconfiguration includes the monitoring and reconfiguration periods during which throughput is reduced, so a longer monitoring interval sometimes results in lower performance.

Lastly, we conducted an experiment for the top-k ratio, for $k =$ 0.5%, 1%, and 2%. Fig. 8 illustrates that both Greedy and Greedy Extended algorithms are not sensitive to variation in this parameter. As such, we use a time window of 10 seconds and top-k ratio of 1% for all the remaining experiments in this paper.

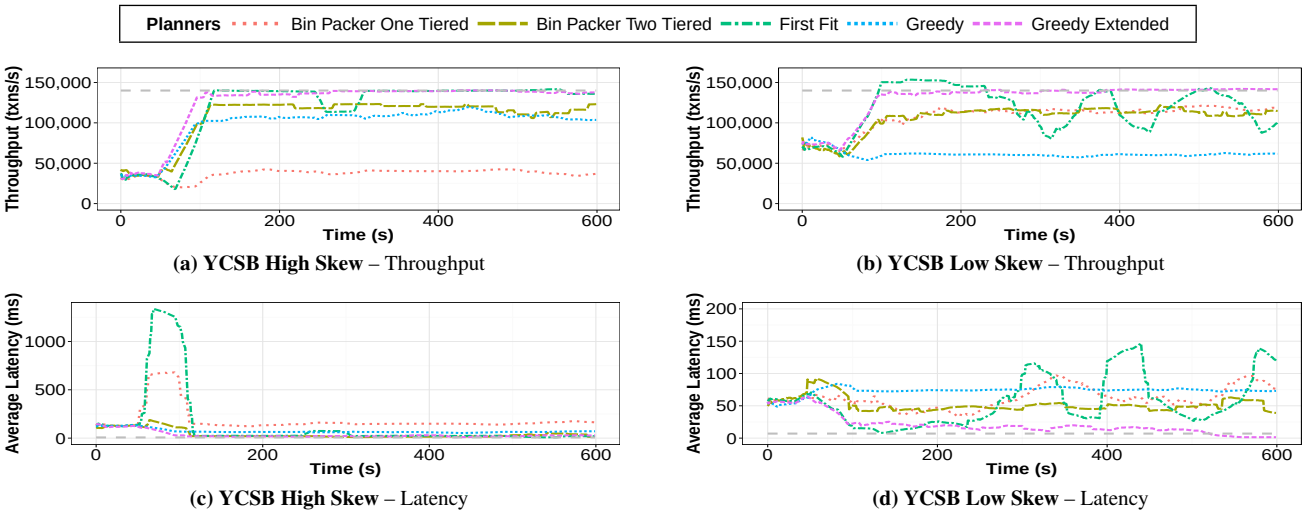
(a) Greedy High Skew (b) Greedy Low Skew (c) Greedy Extended High Skew (d) Greedy Extended Low Skew

Figure 7: Throughput improvement ratio for YCSB after reconfiguration with Greedy and Greedy Extended planners with different time windows.



(a) Greedy High Skew (b) Greedy Low Skew (c) Greedy Extended High Skew (d) Greedy Extended Low Skew

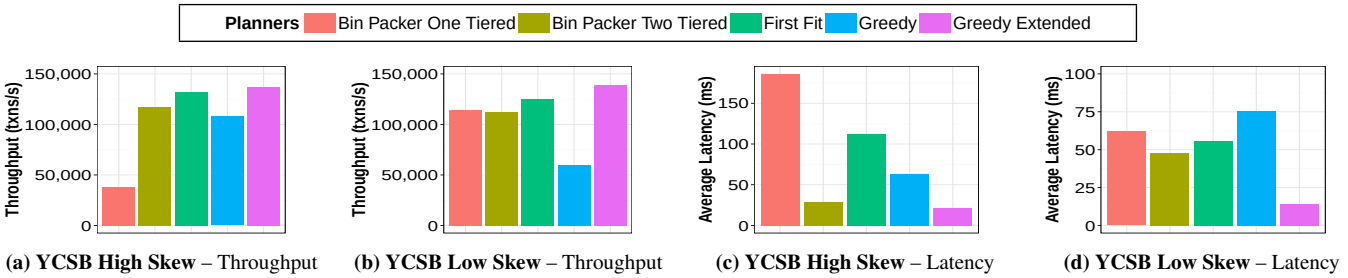
Figure 8: Throughput improvement ratio for YCSB after reconfiguration with Greedy and Greedy Extended planners with different top-*k* ratios.



(a) **YCSB High Skew** – Throughput (b) **YCSB Low Skew** – Throughput

(c) **YCSB High Skew** – Latency (d) **YCSB Low Skew** – Latency

Figure 9: Comparison of all our tuple placement methods with different types of skew on YCSB.



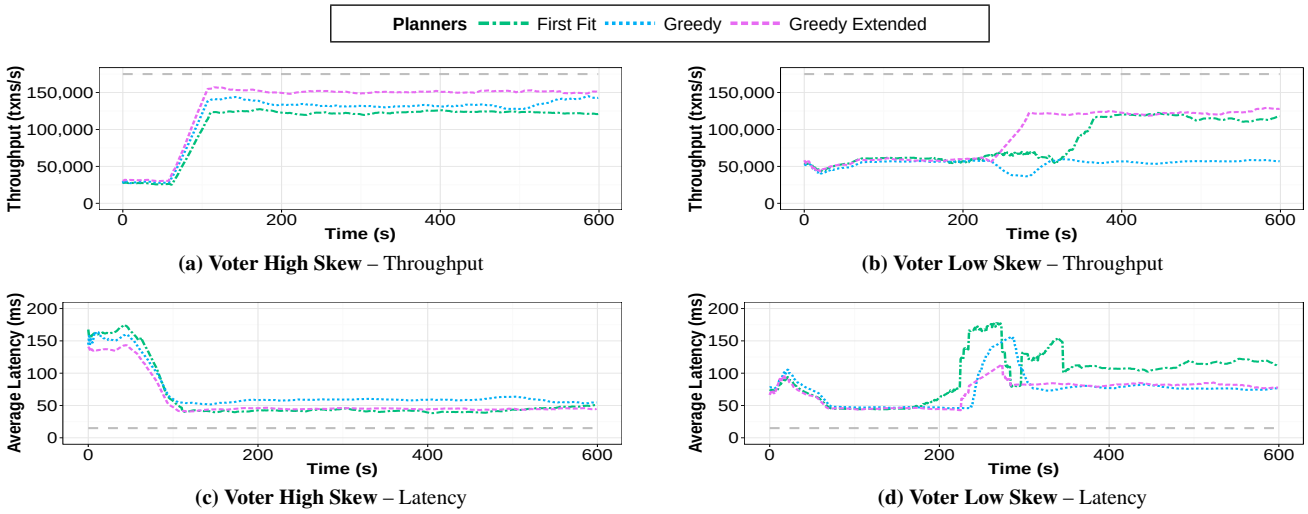
(a) **YCSB High Skew** – Throughput (b) **YCSB Low Skew** – Throughput (c) **YCSB High Skew** – Latency (d) **YCSB Low Skew** – Latency

Figure 10: YCSB throughput and latency from Fig. 9 averaged from the start of reconfiguration at 30 seconds to the end of the run.

6.3 One-Tiered vs. Two-Tiered Partitioning

We next compared the efficacy of the plans generated by the one- and two-tiered placement algorithms. For this experiment, we used the YCSB workload with low and high skew. We implemented both of the bin packing algorithms from Section 5.2 inside of the E-Planner using the GLPK solver[1]. Since these algorithms find the optimal placement of tuples, this experiment compares the ideal scenario for the two different partitioning strategies. The database's tuples are initially partitioned uniformly across five nodes in evenly sized chunks. E-Store moves tuples among these five nodes to cor-

[1] http://www.gnu.org/s/glpk/



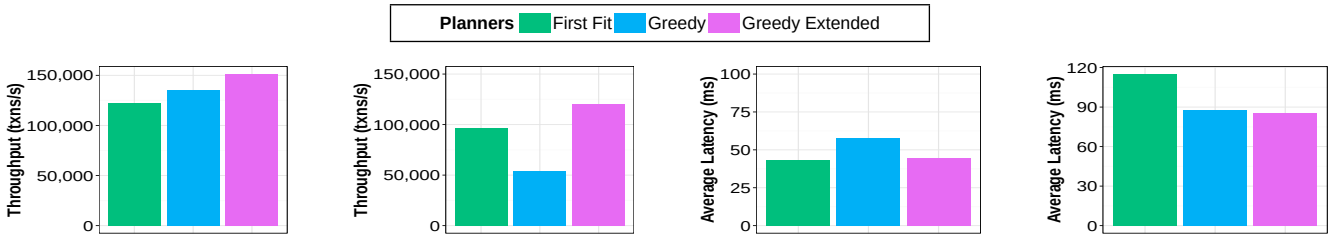
(a) Voter High Skew – Throughput

(b) Voter Low Skew – Throughput

(c) Voter High Skew – Latency

(d) Voter Low Skew – Latency

Figure 11: Comparison of approximate tuple placement methods with different types of skew on Voter.



(a) Voter High Skew – Throughput

(b) Voter Low Skew – Throughput

(c) Voter High Skew – Latency

(d) Voter Low Skew – Latency

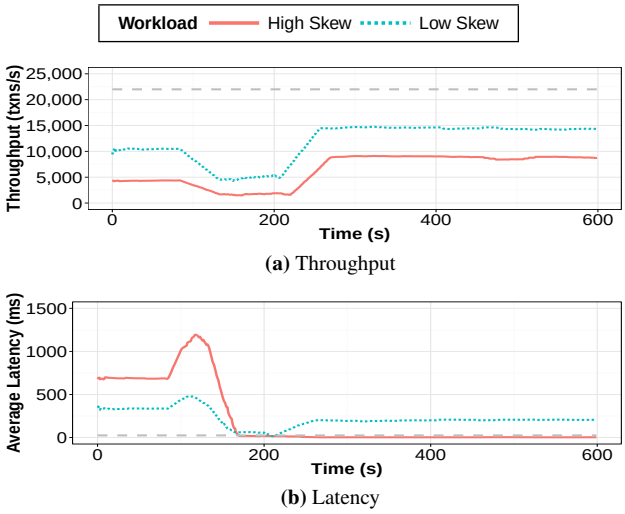
Figure 12: Voter throughput and latency from Fig. 11, averaged from the start of reconfiguration at 30 seconds to the end of the run.

rect for the load imbalance. E-Monitor and E-Planner run as standalone processes on a separate node.

Figs. 9 and 10 show the results of running the two Bin Packer algorithms (and others to be discussed in the next section) on the various types of skew to balance load across the five nodes for the YCSB workload. Note that the time to compute the optimal plan is exceedingly long for an on-line reconfiguration system like E-Store (see Table 1). Thus for these experiments, we terminated the solver after 20 hours; we did not observe a noticeable improvement in the quality of the plan beyond this point.

In Fig. 9 and all subsequent performance vs. time plots, tuple-level monitoring starts 30-seconds after the beginning of the plot. The 20 hours to compute the placement plan for the One- and Two-Tiered bin packer algorithms is not shown in the plots, for obvious reasons. The horizontal dashed gray line indicates system performance with no skew (a uniform load distribution). The goal of E-store is to achieve the same level of performance as the no-skew case even in the presence of skew. The drop in throughput and increase in latency around 30 seconds is due to the overhead of reconfiguration.

Both algorithms perform comparably well in the case of low skew, however the DBMS achieves a lower latency more quickly with the two-tiered approach. Moreover, the two-tiered approach performs better in the high skew workload since it identifies hot spots at the individual tuple level and balances load by redistributing those tuples. The two-tiered approach is able to balance load such that throughput is almost the same as the no skew workload.



(a) Throughput

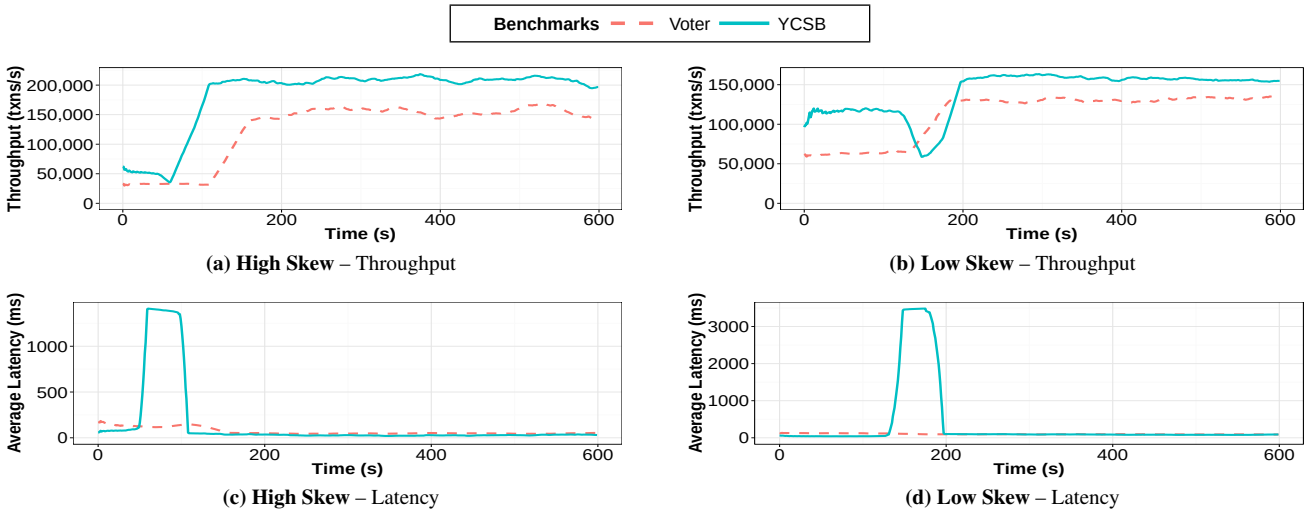
(b) Latency

Figure 13: The Greedy planner with different types of skew on a TPC-C workload. The dashed gray line indicates system performance with no skew (a uniform load distribution).

6.4 Approximate Placement Evaluation

The main challenge for our approximate placement algorithms is to generate a reconfiguration plan in a reasonable time that allows the DBMS perform as well as it does using a plan generated from the optimal algorithms. For these next experiments, we tested our three approximation algorithms from Section 5.3 on YCSB and Voter workloads with both low and high skew. All tuples are initially partitioned uniformly across five nodes. Then during each trial, E-Store moves both hot tuples and cold blocks between nodes to correct for load imbalance caused by skew.

Figs. 9 and 10 show the DBMS's performance using E-Store's approximate planners for the two different skew levels for YCSB.



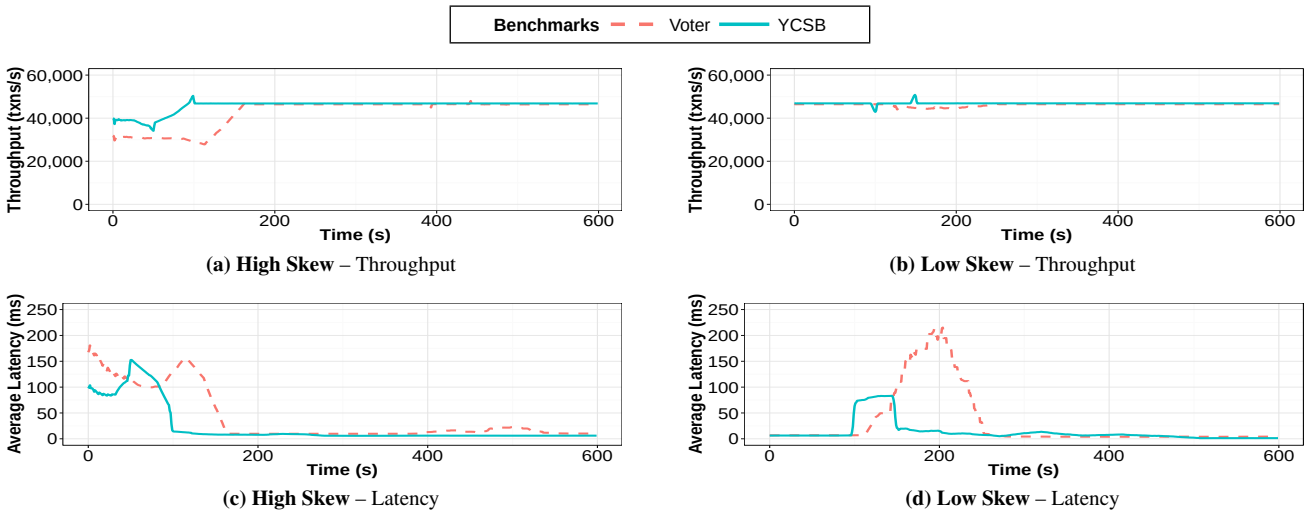
(a) High Skew – Throughput

(b) Low Skew – Throughput

(c) High Skew – Latency

(d) Low Skew – Latency

Figure 14: The Greedy Extended planner with different types of skew on Voter and YCSB workloads. In these experiments we overloaded the system, causing it to scale out from 5 to 6 nodes.



(a) High Skew – Throughput

(b) Low Skew – Throughput

(c) High Skew – Latency

(d) Low Skew – Latency

Figure 15: The Greedy Extended planner with different types of skew on Voter and YCSB workloads. In these experiments we underloaded the system, causing it to scale in from 5 to 4 nodes.

Planner	Low skew	High skew
One-tier bin packer	> 20 hrs	> 20 hrs
Two-tier bin packer	> 20 hrs	> 20 hrs
Greedy	835 ms	103 ms
Greedy Extended	872 ms	88 ms
First Fit	861 ms	104 ms

Table 1: Execution time of all planner algorithms on YCSB.

These results are consistent with our results with the Voter workload reported in Figs. 11 and 12.

In the case of high skew, all three approximate planners perform reasonably well, but Greedy Extended and Greedy stabilize more quickly since they move fewer tuples than First Fit. After stabilizing, Greedy Extended and First Fit both perform comparably to the two-tiered bin packer approach. Specifically, Fig. 9a shows a 4× improvement in throughput and Fig. 9c shows a corresponding 10× improvement in latency. Greedy Extended performs the best overall, however, since it avoids the spike in latency that First Fit exhibits as a result of moving a large number of tuples during reconfiguration.

In the case of low skew, Greedy Extended also produces the best reconfiguration plan since it reaches a stable throughput and latency that is better than the others more quickly. The plan generated by First Fit achieves good performance too, but it does not stabilize within the 10 minute window since it moves such a large amount of data. Reconfiguration of large chunks of data takes time because Squall staggers the movement of data to avoid overloading the system (recall Section 3.1).

In summary, Greedy Extended produces the same performance as the two-tiered bin packer approach and runs in just a few seconds. We note that because the Greedy algorithm only considers hot tuples, it does not generate good plans for workloads with low skew. This provides additional evidence of the importance of considering both hot tuples and cold blocks.

To gauge the effectiveness of E-Store on applications with few root nodes in the tree schema, we also ran two experiments with TPC-C. In our TPC-C experiments there are only 100 root tuples and all the other tuples are co-located with these ones. Hence, our Greedy Extended scheme is overkill and it is sufficient to use the Greedy allocation scheme, which only looks at hot tuples. In the TPC-C experiments, the 100 warehouses were initially partitioned across three machines in evenly sized chunks, with skew settings as described in Section 6.1. As shown in Fig. 13, E-Store improves

both the latency and throughput of TPC-C under the two different levels of skew. The impact of reconfiguration is larger for TPC-C than the other benchmarks for a few reasons. First, each warehouse id has a significant amount of data and tuples associated with it. Therefore, reconfiguring TPC-C requires more time and resources not only to move all data associated with each warehouse, but also to extract and load a large number of indexed tuples. Second, as roughly 10% of transactions in TPC-C are distributed, a migrating warehouse can impact transactions on partitions not currently involved in a migration. For these reasons, load-balancing TPC-C can require longer to complete, but it results in a significant improvement in both throughput and latency.

6.5 Performance after Scaling In/Out

We next measured E-Store's ability to react to load imbalance by increasing and decreasing the DBMS's cluster size. We tested both overloading and underloading the system with the two different levels of skew to prompt E-Store to scale out or in. We used the YCSB and Voter workloads again with tuples initially partitioned across five nodes in evenly sized blocks. We only evaluated plans using the Greedy Extended algorithm, since our previous experiments demonstrated its superiority for these workloads.

E-Store can scale out with minimal overhead in order to handle a system that is simultaneously skewed and overloaded. Fig. 14 shows the results of overloading the system and allowing E-Store to expand from five to six nodes. We also tested E-Store's ability to remove nodes when resources are underutilized. Fig. 15 shows the results of underloading the system and allowing E-Store to scale in from five to four nodes. These experiments show that E-Store maintains system performance when scaling in, other than a brief increase in latency due to migration overhead. In the case of high skew, E-Store actually improves performance due to load balancing, despite using fewer nodes and, hence, fewer partitions.

7. RELATED WORK

Recent work has explored the problem of supporting multi-tenant deployments in cloud-oriented DBMSs. This is exemplified by Salesforce.com's infrastructure that groups applications onto single-node Oracle DBMS installations [7]. Similar work in the Schism [4] and Zephyr [10] projects pack single-node DBMSs together on multi-tenant nodes. In contrast, we focus on elastically provisioning single applications onto multiple nodes in a distributed DBMS.

Essentially all data warehouse DBMSs use hash or range partitioning, and provide some level of on-line reprovisioning. Early work on load balancing by repartitioning for AsterData could reorganize a range-partitioned database [12]. Later in the 2000s, several NoSQL DBMSs were released that use consistent hashing, popularized in Chord [30], to assign tuples to shared-nothing nodes.

NuoDB [25] and VoltDB [36] are NewSQL DBMSs that partition data across multiple nodes in a computing cluster and support on-line reprovisioning. NuoDB uses physical "atoms" (think disk pages) as their unit of partitioning, while VoltDB uses hash partitioning. The key difference between our work and all of these products is that our two-tier partitioning scheme supports both fine- and coarse-grained tuple assignment, and our tightly-coupled approach balances the overhead of the migration of data and the expected performance improvement after the migration.

Hong et al. proposed a method, called *SPORE*, for self-adapting, popularity-based replication of hot tuples [14]. This method mitigates the effect of load imbalance in key-value DBMSs, such as memcached [24]. SPORE does not support ACID semantics nor scaling in/out the number of nodes. It replicates hot keys by renaming them and then this replication is performed randomly without considering underloaded nodes.

Accordion is a one-tiered elasticity controller that explicitly models the effect of distributed transactions on server capacity [29]. As with other one-tier approaches, Accordion relies on a pre-defined set of blocks that can be migrated but are never modified. Accordion is not able to handle situations where hotspots concentrate on a particular block and make it impossible for any server to process the load of that block. By contrast, E-Store's two-tiered approach is able to detect heavily accessed hot tuples within a block, isolate them, and redistribute them to underloaded nodes.

ElasTras is an elastic and scalable transactional database [5]. ElasTras utilizes a decoupled storage architecture that separates storage nodes from transaction manager nodes, each of which is exclusively responsible for a data partition. The focus is on fault tolerance, novel system architecture, and providing primitives for elasticity, such as the ability to add and move partitions [6]. However, ElasTras emphasizes support for multi-tenant databases and transaction execution is limited to a single transaction manager. Therefore, ElasTras cannot support databases that must be partitioned across several nodes. Conversely, load-balancing is accomplished by a greedy heuristic that migrates tenants from over-loaded nodes to the least-utilized nodes. Details for loadbalancing and partition splitting are not presented by the authors.

PLP is a partitioning technique that alleviates locking and logging bottlenecks in a shared-memory DBMS [34]. It recursively splits hot data ranges into fixed-size sub-ranges that are distributed among the partitions. This approach works well with hot ranges that are large, but requires many sub-range splits before it is able to isolate single hot tuples. As the number of ranges grows, monitoring costs grow too. PLP continuously monitors the load on each of the newly created sub-ranges, which has a non-negligible performance impact during regular execution. E-Store is focused on repartitioning for distributed DBMSs, and supports scaling in/out as well as load balancing across multiple servers. E-Store normally uses a lightweight monitoring protocol, and turns on more detailed monitoring only when needed and for a short period of time. This makes it possible to immediately isolate hot spots without having to go through multiple repartitioning cycles.

Several live migration techniques have been proposed to move entire databases from one node to another with minimized interruption of service and downtime. Designed for systems with shared storage, Albatross [6] copies a snapshot of transaction state asynchronously to a destination server. In addition, Slacker [1] is another approach that is optimized for minimizing the impact of migration in multi-tenant DBMSs by throttling the rate that pages are migrated from the source to destination. Zephyr [10] allows concurrent execution at the source and destination during migration, without the use of distributed transactions. Although Zephyr does not require the nodes to be taken off-line at any point, it does require that indexes are frozen during migration. ProRea [28] extends Zephyr's approach, but it instead proactively migrates hot tuples to the destination at the start of the migration.

Previous work has also explored live reconfiguration techniques for partitioned, distributed DBMSs. Wildebeest employed both reactive and asynchronous data migration techniques for a distributed MySQL cluster [16]. In [22] a method is proposed for VoltDB that uses statically defined virtual partitions as the granule of migration. Lastly, Squall [8] allows fine-grained on-line reconfiguration of partitioned databases. In theory, E-Store can use any of these transport mechanisms; our prototype uses a modified version of Squall since it already supports H-Store.

8. CONCLUSION

E-Store is designed to maintain system performance over a highly variable and diverse load. It accomplishes this goal by balancing tuple accesses across an elastic set of partitions. The framework consists of two sub-systems, E-Monitor and E-Planner. E-Monitor identifies load imbalances requiring migration based on CPU utilization, and tracks for a short time window the most-read or -written "hot" tuples. E-Planner chooses which data to move and where to place it. For E-Planner, we developed smart heuristics to make intelligent decisions on how to balance the workload across a distributed OLTP DBMS. E-Planner generates the reconfiguration plan in milliseconds, and the result is a load-balanced system. Moreover, E-Store allows OLTP DBMSs to scale out or in efficiently. Our experiments show that E-Store can start reconfiguring the database after approximately 10 seconds of detecting load skew or a load spike. Reconfiguration results in increasing throughput by up to 4× while reducing latency by up to 10×.

There are several possible directions for future research on E-Store. The first is to extend the framework to support more complex workloads and applications that have many multi-partition transactions. Supporting multi-partition transactions requires extending E-Monitor to collect information about partitions spanned by a transaction, and E-Planner to explicitly consider their cost. Another direction is developing techniques to reduce the overhead of E-Monitor for more complex workloads. One possible approach is to use approximate frequent item counting algorithms such as SpaceSaving [21] or LossyCount [20]. Finally, it would be interesting to extend the planning algorithms to take memory consumption into account so that reconfiguration never places more data on a node than its memory capacity.

9. REFERENCES

[1] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. J. Shenoy. "cut me some slack": latency-aware live migration for databases. In *EDBT*, 2012.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[3] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *PVLDB*, 3(1-2), 2010.

[4] C. Curino, E. P. C. Jones, S. Madden, and H. Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[5] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1):5:1–5:45, 2013.

[6] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8), 2011.

[7] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, 2005.

[8] A. J. Elmore. *Elasticity Primitives for Database as a Service*. PhD thesis, University of California, Santa Barbara, 2013.

[9] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Towards an elastic and autonomic multitenant database. In *NetDB*, 2011.

[10] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD*, 2011.

[11] N. Folkman. So, that was a bummer. `http://is.gd/SRF0sb`.

[12] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[13] J. Gaw. Heavy Traffic Crashes Britannica's Web Site – Los Angeles Times. `http://lat.ms/1fXLjYx`, 1999.

[14] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *SOCC*, 2013.

[15] H-Store: A Next Generation OLTP DBMS. `http://hstore.cs.brown.edu`.

[16] E. P. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis, MIT, 2012.

[17] D. Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, USA, 2007.

[18] R. Kallman et al. H-store: A high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2), 2008.

[19] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[20] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[21] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[22] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. Elastic scale-out for partition-based database systems. In *ICDE Workshops*, 2012.

[23] A. Nazaruk and M. Rauchman. Big data in capital markets. In *ICMD*, 2013.

[24] R. Nishtala et al. Scaling memcache at facebook. In *NSDI*, 2013.

[25] NuoDB. `http://www.nuodb.com`.

[26] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[27] A. Pavlo, E. P. C. Jones, and S. Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB*, 5(2):85–96, 2011.

[28] O. Schiller, N. Cipriani, and B. Mitschang. ProRea: Live Database Migration for Multi-Tenant RDBMS with Snapshot Isolation. In *EDBT*, 2013.

[29] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12), 2014.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[31] R. Stoica, J. J. Levandoski, and P.-A. Larson. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[32] M. Stonebraker et al. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[33] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull*, 36(2), 2013.

[34] P. Tözün, I. Pandis, R. Johnson, and A. Ailamaki. Scalable and dynamically balanced shared-everything oltp with physiological partitioning. *The VLDB Journal*, 22(2):151–175, 2013.

[35] The TPC-C Benchmark, 1992. http://www.tpc.org/tpcc/.

[36] VoltDB. `http://www.voltdb.com`.