Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases

Aaron J. Elmore[1], Vaibhav Arora[2], Rebecca Taft[3]
Andrew Pavlo[4], Divyakant Agrawal[2,5], Amr El Abbadi[2]
[1]University of Chicago, [2]University of California, Santa Barbara
[3]MIT CSAIL, [4]Carnegie Mellon University, [5]Qatar Computing Research Institute
aelmore@cs.uchicago.edu, {vaibhavarora,agrawal,amr}@cs.ucsb.edu,
rytaft@mit.edu, pavlo@cs.cmu.edu

ABSTRACT

For data-intensive applications with many concurrent users, modern distributed main memory database management systems (DBMS) provide the necessary scale-out support beyond what is possible with single-node systems. These DBMSs are optimized for the short-lived transactions that are common in on-line transaction processing (OLTP) workloads. One way that they achieve this is to partition the database into disjoint subsets and use a single-threaded transaction manager per partition that executes transactions one-at-a-time in serial order. This minimizes the overhead of concurrency control mechanisms, but requires careful partitioning to limit distributed transactions that span multiple partitions. Previous methods used off-line analysis to determine how to partition data, but the dynamic nature of these applications means that they are prone to hotspots. In these situations, the DBMS needs to reconfigure how data is partitioned in real-time to maintain performance objectives. Bringing the system off-line to reorganize the database is unacceptable for on-line applications.

To overcome this problem, we introduce the Squall technique for supporting live reconfiguration in partitioned, main memory DBMSs. Squall supports fine-grained repartitioning of databases in the presence of distributed transactions, high throughput client workloads, and replicated data. An evaluation of our approach on a distributed DBMS shows that Squall can reconfigure a database with no downtime and minimal overhead on transaction latency.

1. INTRODUCTION

The need for scalable main-memory DBMSs is motivated by decreasing memory costs and an increasing demand for high-throughput transaction processing systems. Such systems eschew legacy disk-oriented concurrency control and recovery mechanisms of traditional DBMSs [37] and alleviate the contention on shared data-structures [17,26,31,40]. But some OLTP databases are larger than the amount of DRAM that is available on a single node. Furthermore, modern web/mobile applications require always-available data with the ability to support sudden shifts in access patterns. The inability to react to changes in usage or mitigate potential downtime can cause significant financial losses for service providers [7].

This argues for the use of a distributed DBMS architecture where the database is deployed in memory-only storage on a cluster of shared-nothing nodes. These scalable DBMSs, colloquially known as *NewSQL* [10], achieve high performance and scalability without sacrificing the benefits of strong transactional guarantees by spreading the databases across nodes into disjoint *partitions*. Recent examples of these systems include H-Store [24] (and its commercial version VoltDB [6]), MemSQL [2], and SQLFire [5].

Even if a database resides entirely in memory across multiple partitions, the DBMS is not immune to problems resulting from changes in workload demands or access patterns. For example, sudden increases in the popularity of a particular item in the database can negatively impact the performance of the overall DBMS. Modern distributed systems can, in theory, add and remove resources dynamically, but in practice it is difficult to scale databases in this manner [23]. Increasing system capacity involves either *scaling up* a node by upgrading hardware or *scaling out* by adding additional nodes to the system in order to distribute load. Either scenario can involve migrating data between nodes and bringing nodes off-line during maintenance windows [18].

Previous work has shown how to migrate a database from one node to another incrementally to avoid having to shut down the system [8, 15, 19, 35]. These approaches, however, are not suitable for partitioned main-memory DBMSs. In particular, they are predicated upon disk-based concurrency control and recovery mechanisms. This means that they rely on concurrent data access to migrate data using heavy-weight two-phase locking and snapshot isolation methods to ensure correctness [19, 35]. More proactive migration techniques rely on physical logging from standard replication and recovery mechanisms [8, 15]. But this dependency on the DBMS's replication infrastructure places additional strain on partitions that may be already overloaded. Above all, these approaches are inappropriate for partitioned main-memory DBMSs that execute transactions serially [37], since the system does not support concurrent access or physical logging.

Even if one adapted these approaches to move partitions between nodes, they are still not able to split one partition into multiple partitions. For example, if there is a particular entity in a partition that is extremely popular (e.g., the Wu Tang Clan's Twitter account), then instead of migrating the entire partition to a new node it is better to move those individual tuples to their own partition [38].

A better (but more difficult) approach is to dynamically *reconfigure* the physical layout of the database while the system is live. This allows the system to continue to process transactions as it migrates

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723726.

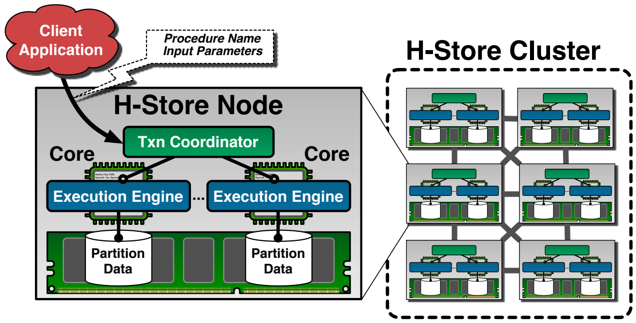


Figure 1: The H-Store architecture from [33].

data and immediately relieves contention on hotspots. Some distributed NoSQL DBMSs, such as MongoDB [3], support splitting and migration of partitions to new nodes when the system needs to re-balance [20]. But accomplishing this is easier when the DBMS does not support atomic operations on multiple objects. Another approach is to pre-allocate multiple "virtual" partitions for each real partition at start-up and then migrate some of the virtual partitions to new nodes for re-balancing the load [34]. The downside of this is that the DBMS has no control of the contents of these virtual partitions, and thus there is no way to know whether the migration will result in the desired change in performance until *after* the virtual partitions have been migrated. To the best of our knowledge, no DBMS today supports the fine-grained, tuple-level load balancing that is needed for the system to be truly autonomous.

Given the lack of solutions for this problem, we present the **Squall** migration system for partitioned OLTP DBMSs. Our key contribution in this paper is an efficient mechanism for performing fine-grained live reconfiguration by safely interleaving data migration with executing transactions. We also present several optimizations that reduce the migration costs for a variety of workloads. To evaluate our work, we implemented Squall in the H-Store [1] DBMS and measured the system's performance using two OLTP workloads. Our results demonstrate that Squall is able to reconfigure a DBMS with no downtime and a minimal decrease in throughput.

The rest of the paper is organized as follows. We start in Section 2 with an overview of the DBMS architecture targeted by our work and the reconfiguration scenarios that are important in this operating environment. We then present Squall in Section 3, discuss the details of data reconfiguration management in Section 4, and outline several optimizations in Section 5. Section 6 outlines how fault tolerance is enabled in Squall. We then provide a thorough evaluation of Squall in Section 7. Finally, we describe related work in Section 8, and conclude in Section 9.

2. BACKGROUND

We begin with an overview of the architecture of H-Store, an example of the type of distributed DBMS that we target in this work. We then show how these DBMSs are susceptible to load imbalances and how a naïve migration approach to reconfiguring a database is insufficient. Although we use H-Store in our analysis, our work is applicable to any partitioned, main memory OLTP DBMS.

2.1 H-Store Architecture

H-Store is a distributed, row-oriented DBMS that supports serializable execution of transactions over main memory partitions. We define an H-Store instance as a cluster of two or more nodes deployed within the same administrative domain. A *node* is a single physical computer system that contains a transaction coordinator that manages one or more partitions.

H-Store is optimized for the efficient execution of workloads that contain transactions invoked as pre-defined stored procedures. Each stored procedure is comprised of (1) parameterized queries and (2) control code that contains application logic intermixed with invocations of those queries. Client applications initiate transactions by sending the procedure name and input parameters to any node in the cluster. The partition where the transaction's control code executes is known as its *base partition* [33]. The base partition ideally will have most (if not all) of the data the transaction needs [32]. Any other partition involved in the transaction that is not the base partition is referred to as a *remote partition*.

As shown in Fig. 1, each partition is assigned a single-threaded *execution engine* that is responsible for executing transactions and queries for that partition. A partition is protected by a single lock managed by its coordinator that is granted to transactions one-at-a-time based on the order of their arrival timestamp [9, 13, 41]. A transaction acquires a partition's lock if (a) the transaction has the lowest timestamp that is not greater than the one for the last transaction that was granted the lock and (b) it has been at least 5 ms since the transaction first entered the system [37]. This wait time ensures that distributed transactions that send their lock acquisition messages over the network to remote partitions are not starved. We assume that standard clock-skew algorithms are used to keep the nodes' CPU clocks synchronized.

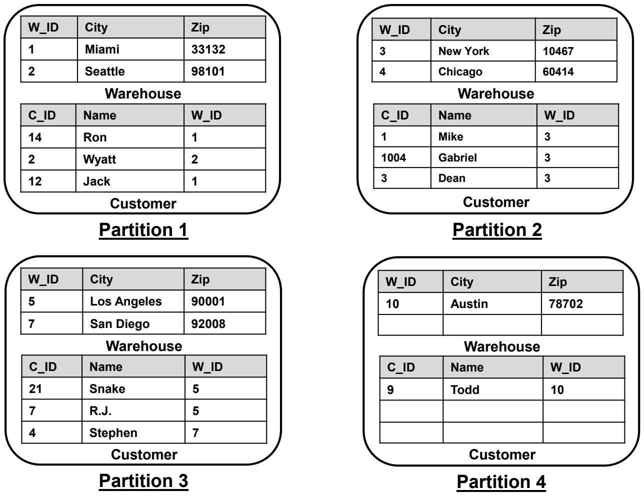
Executing transactions serially at each partition has several advantages for OLTP workloads. In these applications, most transactions only access a single entity in the database. That means that these systems are much faster than a traditional DBMS if the database is partitioned in such a way that most transactions only access a single partition [32]. The downside of this approach, however, is that transactions that need to access data at two or more partitions are slow. If a transaction attempts to access data at a partition that it does not have the lock for, then the DBMS aborts that transaction (releasing all of the locks that it holds), reverts any changes, and then restarts it once the transaction re-acquires all of the locks that it needs again. This removes the need for distributed deadlock detection, resulting in better throughput for short-lived transactions [22].

All data in H-Store is stored in main memory. To ensure that transactions' modifications are durable and persistent, each node writes asynchronous snapshots of the entire database to disk at fixed intervals [25,37]. In between these snapshots, the DBMS writes out a record to a redo-only *command log* for each transaction that completes successfully [27]. In addition to snapshots and command logging, main memory databases often use replication to provide durability and high availability. Each partition is fully replicated by another secondary partition that is hosted on a different node [27].

2.2 Database Partitioning

A *partition plan* for a database in a distributed OLTP DBMS is comprised of (1) partitioned tables, (2) replicated tables, and (3) transaction routing parameters [32]. A table can be horizontally partitioned into disjoint fragments whose boundaries are based on the values of one (or more) of the table's columns (i.e., *partitioning attributes*). Alternatively, the DBMS can replicate non-partitioned tables across all partitions. This table-level replication is useful for read-only or read-mostly tables that are accessed together with other tables but do not partition in accordance with other tables. A transaction's routing parameters identify the transaction's base partition from its input parameters.

Administrators deploy databases using a partition plan that minimizes the number of distributed transactions by collocating the records that are used together often in the same partition [14, 32]. A plan can be implemented in several ways, such as using hash, range, or round-robin partitioning [16]. For this paper, we modi-



W_ID	City	Zip
1	Miami	33132
2	Seattle	98101

Warehouse

C_ID	Name	W_ID
14	Ron	1
2	Wyatt	2
12	Jack	1

Customer

Partition 1

W_ID	City	Zip
3	New York	10467
4	Chicago	60414

Warehouse

C_ID	Name	W_ID
1	Mike	3
1004	Gabriel	3
3	Dean	3

Customer

Partition 2

W_ID	City	Zip
5	Los Angeles	90001
7	San Diego	92008

Warehouse

C_ID	Name	W_ID
21	Snake	5
7	R.J.	5
4	Stephen	7

Customer

Partition 3

W_ID	City	Zip
10	Austin	78702

Warehouse

C_ID	Name	W_ID
9	Todd	10

Customer

Partition 4

Figure 2: Simple TPC-C data, showing `WAREHOUSE` and `CUSTOMER` partitioned by warehouse IDs.

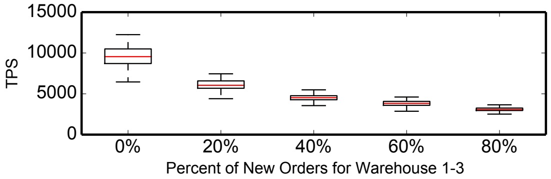


Figure 3: As workload skew increases, the number of new order transactions increasingly access 3 warehouses in TPC-C and the collocated warehouses experience reduced throughput due to contention.

fied H-Store to use range partitioning. We discuss how to support alternative partitioning schemes in Appendix C.

Fig. 2 shows a simplified TPC-C database partitioned by the plan in Fig. 5a. The `WAREHOUSE` table is partitioned by its id column (`W_ID`). Since there is a foreign key relationship between the `WAREHOUSE` and `CUSTOMER` tables, the `CUSTOMER` table is also partitioned by its `W_ID` attribute. Hence, all data related to a given `W_ID` (i.e., both `WAREHOUSE` and `CUSTOMER`) are collocated on a single partition. Any stored procedure that reads or modifies either table will use `W_ID` as its routing parameter. Since there is a foreign-key relationship between `CUSTOMER` and `WAREHOUSE`, the `CUSTOMER` table does not need an explicit mapping in the partition plan. We will use this simplified example throughout the paper for exposition.

2.3 The Need for Live Reconfiguration

Although partitioned DBMSs like H-Store execute single-partition transactions more efficiently than systems that use a heavyweight concurrency control scheme, they are still susceptible to performance degradations due to changes in workload access patterns [32]. Such changes could either cause a larger percentage of transactions to access multiple partitions or partitions to grow larger than the amount of memory available on their node. As with any distributed system, DBMSs need to react to these situations to avoid becoming overloaded; failing to do so in a timely manner can impact both performance and availability in distributed DBMSs [20].

To demonstrate the detrimental effect of overloaded partitions on performance, we ran a micro-benchmark using H-Store with a three-node cluster. For this experiment, we used the TPC-C benchmark with a 100 warehouse database evenly distributed across 18 partitions. We modified the TPC-C workload generator to create a hotspot on one of the partitions by having a certain percentage of transactions access one of three hot warehouses instead of a uniformly random warehouse. Transaction requests are submitted from up to 150 clients running on a separate node in the same cluster. We postpone the details of these workloads and the execution environment until Section 7. As shown in Fig. 3, as the warehouse selection moves from a uniform to a highly skewed distribution, the throughput of the system degrades by ∼60%.

This shows that overloaded partitions have a significant impact on the throughput of a distributed DBMS like H-Store. The solution is for the DBMS to respond to these adverse conditions by migrating data to either re-balance existing partitions or to offload data to new partitions. Some of the authors designed E-Store [38] for automatically identifying when a reconfiguration is needed and to create a new partition plan to shuffle data items between partitions. E-Store uses system-level statistics (e.g., sustained high CPU usage) to identify the need for reconfiguration, and then uses tuple-level statistics (e.g., tuple access frequency) to determine the placement of data to balance load across partitions. E-Store relies on Squall to execute the reconfiguration. Both components view each other as a black-box; E-Store only provides Squall with an updated partition plan and a designated leader node for a reconfiguration. Squall makes no assumptions on the plans generated by a system controller other than that all tuples must be accounted for.

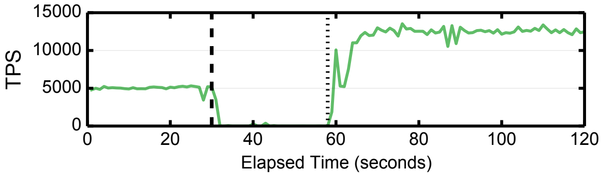


Figure 4: A Zephyr-like migration on two TPC-C warehouses to alleviate a hot-spot effectively causes downtime in a partitioned main-memory DBMS.

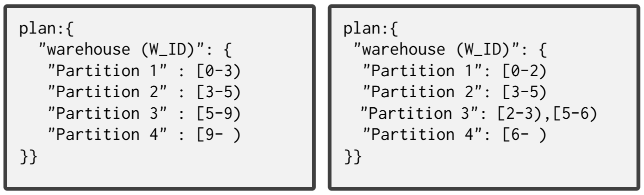
In building E-Store we evaluated a Zephyr-like migration for load-balancing (cf. Section 7 for a detailed description) where destination partitions reactively migrate tuples as needed and periodically pull large blocks of tuples. As shown by Fig. 4, however, this approach results in downtime for the system, and is therefore not an option for modern OLTP systems. This disruption is due to migration requests blocking transaction execution and a lack of coordination between partitions involved in the migration.

These issues highlight the demand for a live reconfiguration system that seeks to minimize performance impact and eliminate downtime. We define a *live reconfiguration* as a change in the assignment of data to partitions in which data is migrated without any part of the system going off-line. A reconfiguration can cause the number of partitions in the cluster to increase (i.e., data from existing partitions are sent to a new, empty partition), decrease (i.e., data from a partition being removed is sent to other existing partitions), or stay the same (i.e., data from one partition is sent to another partition).

3. OVERVIEW OF SQUALL

Squall is a technique for efficiently migrating fine-grained data in a strongly consistent distributed OLTP DBMS. The key advantage of Squall over previous approaches is that it does not require the DBMS to halt execution while all data migrates between partitions, thereby minimizing the impact of reconfiguration. The control of data movement during a reconfiguration is completely decentralized and fault-tolerant.

Squall is focused on the problem of how to perform this reconfiguration safely. In particular, Squall ensures that during the reconfiguration process the DBMS has no false negatives (i.e., the system assumes that a tuple does not exist at a partition when it actually does) or false positives (i.e., the system assumes that a tuple exists



```
plan:{
  "warehouse (W_ID)": {
    "Partition 1" : [0-3)
    "Partition 2" : [3-5)
    "Partition 3" : [5-9)
    "Partition 4" : [9- )
}}
```

(a) Old Plan

```
plan:{
 "warehouse (W_ID)": {
  "Partition 1": [0-2)
  "Partition 2": [3-5)
  "Partition 3": [2-3),[5-6)
  "Partition 4": [6- )
}}
```

(b) New Plan

Figure 5: An example of an updated partition plan for the TPC-C database shown in Fig. 2.

at a partition when it does not). Determining when a reconfiguration should occur and how the partition plan should evolve are addressed in the E-Store project [38]. We assume that a separate system controller monitors the DBMS and then initiates the reconfiguration process by providing the system with the new partition plan when appropriate [18, 38].

Squall processes a live reconfiguration in three stages: (1) initializing the partitions' tracking data structures, (2) migrating data between partitions, and (3) identifying when the reconfiguration process has terminated. In this section, we provide an overview of these three steps. We discuss Squall's data migration protocol in further detail in Section 4. We then present optimizations of this process in Section 5, such as eager data requests and a divide-and-conquer technique for reconfiguration plans.

3.1 Initialization

The initialization phase synchronously engages all of partitions in the cluster to start a new reconfiguration. As part of this step, each partition prepares for reconfiguration and identifies the tuples it will be exchanging with other partitions.

A reconfiguration starts when the DBMS is notified by an external controller that the system needs to re-balance. This notification identifies (1) the new partition plan for the database and (2) the designated *leader* node for the operation. The leader is any node in the cluster that contains a partition affected by the reconfiguration. If the reconfiguration calls for a new node to be added to the cluster, then that node must be on-line before the reconfiguration can begin.

To initiate the reconfiguration, the leader invokes a special transaction that locks every partition in the cluster and checks to see whether it is allowed to start the reconfiguration process. This locking is the same as when a normal transaction needs to access every partition in the system. The request is allowed to proceed if (1) the system has terminated all previous reconfigurations and (2) the DBMS is not writing out a recovery snapshot of the database to disk. If either of these conditions is not satisfied, then the transaction aborts and is re-queued after the blocking operation finishes. This ensures that all partitions have a consistent view of data ownership and prevents deadlocks caused by concurrent reconfigurations.

Once all of the partitions agree to start the reconfiguration, they then enter a "reconfiguration mode" where each partition examines the new plan to identify which tuples are leaving the partition (outgoing) and which tuples will be moving into the partition (incoming). These incoming and outgoing tuples are broken into ranges based on their partitioning attributes. As we discuss in Section 5, this step is necessary because Squall may need to split tuple ranges into smaller chunks or split the reconfiguration into smaller sub-reconfigurations for performance reasons.

After a partition completes this local data analysis, it notifies the leader and waits to learn whether the reconfiguration will proceed. If all of the partitions agree to proceed with the reconfiguration, then the leader sends out acknowledgement to all of the partitions to begin migrating data. Squall only uses this global lock during the initialization phase to synchronize all partitions to begin reconfiguration and does not migrate any data. Since the reconfiguration transaction only modifies the meta-data related to reconfiguration, the transaction is extremely short and has a negligible impact on performance. For all our trials in our experimental evaluation, the average length of this initialization phase was ∼130 ms.

3.2 Data Migration

The easiest way to safely transfer data in a distributed DBMS is to stop executing transactions and then move data to its new location. This approach, known as *stop-and-copy*, ensures that transactions execute either before or after the transfer and therefore have a consistent view of the database. But shutting down the system is unacceptable for applications that cannot tolerate downtime, so stop-and-copy is not an option. It is non-trivial, however, to transfer data while the system is still executing transactions. For example, if half of the data that a transaction needs has already been migrated to a different partition, it is not obvious whether it is better to propagate changes to that partition or restart the transaction at the new location. The challenge is in how to coordinate this data movement between partitions without any lost or duplicated data, and with minimal impact to the DBMS's performance.

To overcome these problems, Squall tracks the location of migrating tuples at each partition during the reconfiguration process. This allows each node's transaction manager to determine whether it has all of the tuples that are needed for a particular transaction. If the system is uncertain of the current location of required tuples, then the transaction is scheduled at the partitions where the data is supposed to be according to the new plan. Then when the transaction attempts to access the tuples that have not been moved yet, Squall will reactively pull data to the new location [19, 35]. Although this on-demand pull method introduces latency for transactions, it has four benefits: (1) it always advances the progress of data migration, (2) active data is migrated earlier in the reconfiguration, (3) it requires no external coordination, and (4) it does not incur any downtime to synchronize ownership metadata.

In addition to the on-demand data pulls, Squall also asynchronously migrates additional data so that the reconfiguration completes in a timely manner. All of these migration requests are executed by a partition in the same manner as regular transactions. Each partition is responsible for tracking the progress of migrating data between itself and other partitions. In other words, each partition only tracks the status of the tuples it is migrating. This allows it to identify whether a particular tuple is currently stored in the local partition or whether it must retrieve it from another partition.

3.3 Termination

Since there is no centralized controller in Squall that monitors the process of data migration, it up to each partition to independently determine when it has sent and/or received all of the data that it needs. Once a partition recognizes that it has received all of the tuples required for the new partition plan, it notifies the current leader that the data migration is finished at that partition. When the leader receives acknowledgments from all of the partitions in the cluster, it notifies all partitions that the reconfiguration process is complete. Each partition removes all of its tracking data structures and exits the reconfiguration mode.

4. MANAGING DATA MIGRATION

The migration of data between partitions in a transactionally safe manner is the most important feature of Squall. As such, we now discuss this facet of the system in greater detail.

We first describe how Squall divides each partition's migrating tuples into ranges and tracks the progress of the reconfiguration

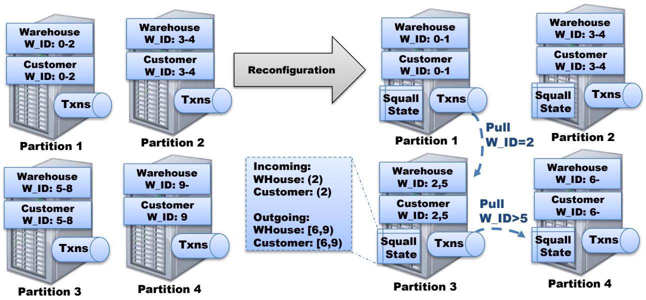


Figure 6: As a system's partition plan changes, Squall tracks the progress of the reconfiguration at each node to ensure correct data ownership.

based on these ranges. We then describe the two ways that Squall moves data: *reactive migration* and *asynchronous migration*. The former moves tuples when transactions need them. This allows the system to migrate hot tuples early in the reconfiguration without the use of complex usage modeling [36]. The latter is when the system periodically sends data to ensure that the reconfiguration eventually completes with minimal impact on the DBMS's performance.

To help explain this process, we will refer to Figs. 5 and 6 as our running example. For a particular tuple that is migrating from one partition to another, we refer to the partition that is losing that tuple as the *source* partition and the partition that receives the tuple as the *destination* partition. Although a partition can be both a source and destination during a reconfiguration, we refer to a partition as either a source or destination for a particular tuple.

4.1 Identifying Migrating Data

When a new reconfiguration begins, Squall calculates the difference between the original partition plan and the new plan to determine the set of incoming and outgoing tuples per partition. This allows a partition to determine when it can safely execute a transaction that accesses migrating data without any external metadata.

Migrating tuples are organized into *reconfiguration ranges* that specify a table name, the partitioning attribute(s) of the table, the minimum-inclusive key, maximum-exclusive key, and the old/new partition IDs. Tables in distributed OLTP DBMSs are partitioned by one or more columns [14, 32]. Without loss of generality, however, we discuss reconfiguration ranges for the single column case.

Each partition locally organizes the ranges into incoming and outgoing ranges based on the previous and new partition IDs. For example, in the reconfiguration shown in Figs. 5 and 6, the incoming warehouses for partition 3 are noted as:

$$(\texttt{WAREHOUSE}, \texttt{W\_ID} = [2, 3), 1 \rightarrow 3)$$

This means that partition 3 receives warehouse 2 from partition 1. Likewise, the following range identifies that partition 3 sends all warehouses with an ID of 6 or greater to partition 4:

$$(\texttt{WAREHOUSE}, \texttt{W\_ID} = [6, \infty), 3 \rightarrow 4)$$

These rules cascade for all tables that are not explicitly listed in these partition plan entries. That is, any table with a foreign-key relationship to one of the tables identified in an entry will have its tuples migrated based on these rules as well. In our TPC-C example, the CUSTOMER table is partitioned by its WAREHOUSE ID, thus partition 3 would also have the following implicit rule that sends all customers with a W_ID of 6 or greater to partition 4:

$$(\texttt{CUSTOMER}, \texttt{W\_ID} = [6, \infty), 3 \rightarrow 4)$$

All of the tables in TPC-C that are partitioned on their WAREHOUSE id, such as DISTRICT, ORDERS, and STOCK, are handled similarly. We note that this makes predicting how long the migration will take for a partition difficult. Since these tables are not partitioned by a primary or unique key, the number of tuples associated with a range can be far larger than the cardinality of the partition keys encapsulated by the range (e.g., there can be thousands of customers associated with a single W_ID).

Each of the above ranges is derived deterministically from the original and new partition plans. This means that each partition can independently calculate its local set of incoming and outgoing ranges from the updated plan that it received in the initialization phase. The advantage of this approach is that a partition only has to track the list of ranges migrating to or from itself. This reduces the amount of global state that Squall maintains during the reconfiguration and facilitates several other enhancements to improve performance. We discuss these additional optimizations for this phase in Section 5, including how to split ranges to reduce the amount of data associated with each of them.

4.2 Tracking Reconfiguration Progress

Squall tracks the progress of data migration for all incoming and outgoing ranges at a particular partition. It maintains a table at each partition to record the current status of these ranges:

NOT STARTED: The data associated with the range has not yet migrated to/away from this partition, and therefore all data associated with the range is located at the source partition.

PARTIAL: Some of the data for the range has been migrated, and some of the tuples may be currently in-flight between the source and destination partitions.

COMPLETE: All of the data for the range has migrated to the destination partition.

Continuing with our example from Section 4.1, a status of NOT STARTED for the CUSTOMER table range indicates that all customers with a W_ID of 6 or greater are present only at partition 3. This means that any transaction that needs to access the tuples for these customers will do so at partition 3.

Since a transaction could execute a query that contains a predicate at a different granularity than the reconfiguration range, Squall allows for the initial reconfiguration ranges to be split into sub-ranges by a query. For the same WAREHOUSE range example with a NOT STARTED status, assume that a transaction arrives at partition 4 that executes the following query:

```
SELECT * FROM WAREHOUSE WHERE W_ID >= 6 AND W_ID <= 7
```

The original range includes customers with W_ID > 7, thus the entire range is not needed for this transaction. In this scenario, partition 4 would split the original range $[6, \infty)$ into the following two ranges both with the status of NOT STARTED:

$$(\texttt{WAREHOUSE}, \texttt{W\_ID} = [6, 8), 3 \rightarrow 4)$$
$$(\texttt{WAREHOUSE}, \texttt{W\_ID} = [8, \infty), 3 \rightarrow 4)$$

When partition 4 requests the data associated with the first range, partition 3 similarly splits its original range to reflect the requested range. Once the data for the sub-range is migrated, both partitions update the status of the first range to COMPLETE.

For transactions with queries that access an individual key through an equality predicate (e.g., W_ID = 7), Squall must find the range that the key belongs to in its tracking table to determine which partition has that data. Since many OLTP workloads are comprised of transactions that access tuples through single keys, Squall also supports recording the movement of individual tuples at the key level through its tracking table. This enables faster lookups at runtime than scanning the partition plan entries to determine whether a key is in a range. It also reduces the amount of work required for range splitting and simplifies the tracking of keys with non-discrete domains (e.g., strings or floating-point values).

When a transaction reads or updates tuples, Squall ensures that all required data is present. Using our TPC-C example from above, assume that no data has yet been migrated (i.e., all of the entries in the tracking table are set to NOT STARTED). A transaction then arrives at partition 4 that executes a query to update the WAREHOUSE tuple with W_ID = 7. After failing to find a matching key entry in the tracking table and checking that the corresponding range is not marked as COMPLETE, partition 4 issues a request for (WAREHOUSE, W_ID = 7). Once the warehouse is migrated, both partitions set the corresponding WAREHOUSE range in the status table to PARTIAL and add a key-based entry for (WAREHOUSE, W_ID = 7) in the tracking table with the status of COMPLETE.

4.3 Identifying Data Location

Under normal operation, when a new transaction request arrives at the DBMS, the system evaluates the routing parameters for the stored procedure that the transaction wants to execute with the current plan to determine the base partition for that request [29, 32]. But during a reconfiguration, Squall intercepts this process and uses its internal tracking metadata to determine the base partition. This is because the partition plan is in transition as tuples move and thus the location of a migrating tuple is uncertain. The mechanisms in Section 4.2 ensure safe execution of transactions on migrating data, whereas the following techniques for determining where to execute transactions are for improving the DBMS's performance.

If the data that a transaction needs is moving and either the source or destination partition is local to the node receiving the transaction, then Squall will check with the local partition to determine whether the required tuple(s) are present (i.e., the source partition has the range marked NOT STARTED or the destination partition has the range marked COMPLETE). If Squall deems that the tuples' locations are uncertain, then it forwards the request to the destination partition. Since a transaction could have been (correctly) scheduled at a source partition before tuples were migrated, Squall traps transaction execution before it starts to verify that those required tuples were not migrated out while the transaction was queued. In this scenario, the transaction is restarted at the destination partition.

Scheduling the transaction at the partition that always has the required tuples improves performance by avoiding a migration when a transaction has already arrived at the same local site. But in all other cases, it is better to send the transaction to the destination partition and then use Squall's reactive migration mechanism to pull data as it is needed. This alleviates overloaded partitions more quickly by migrating hot tuples first and minimizes the communication overhead of partitions trying to determine a tuple's location.

4.4 Reactive Migration

Squall schedules each transaction at its corresponding destination partition, even if the system has not completed the migration of the data that the transaction will access when it executes. Likewise, when a transaction invokes a query, Squall examines the request to determine whether the data that it accesses has migrated to its destination partition. In this scenario, the destination partition blocks the query while the data is reactively pulled from the source partition. This blocks all transactions that access these partitions during this time, thereby avoiding consistency problems (e.g., two transactions modify the same the tuple while it is moving).

To perform the reactive migration, Squall issues a *pull request* to the source partition. This request is queued just like a transaction and thus acquires exclusive locks on both source and destination partitions. As both partitions are locked during the migration of data items, no other transaction can concurrently issue a read or update query at these partitions, thereby preventing any transaction anomalies due to the reconfiguration.

The pull request is scheduled at the source partition with the highest priority so that it executes immediately after the current transaction completes and any other pending reactive pull requests. Squall relies on the DBMS's standard deadlock detection to prevent cyclical reactive migrations from stalling the system, since all transactions are blocked while the source partition is extracting the requested data. Additionally, transaction execution is also blocked at the destination partition when it loads the requested data. Since the blocking caused by reactive migrations can be disruptive for active transactions, Squall also employs an additional asynchronous migration mechanism to minimize the number of reactive pulls.

4.5 Asynchronous Migration

After Squall initializes the list of incoming data ranges for each partition (cf. Section 3.1), it then generates a series of asynchronous migration requests that pull data from the source partitions to their new destination. During the reconfiguration, Squall periodically schedules these requests one-at-a-time per partition with a lower priority than the reactive pull requests.

When the source partition receives an asynchronous pull request, it first marks the target range as PARTIAL in its tracking table. It then sub-divides the request into tasks that each retrieve a fixed-size chunk to prevent transactions from blocking for too long if Squall migrates a large range of tuples. If the last task did not extract all of the tuples associated with the range, then another task for the asynchronous pull request is rescheduled at the source partition. As Squall sends each chunk to the destination partition, it includes a flag that informs the destination whether it will send more data for the current range. For the first chunk that arrives at the destination for the range, the destination lazily loads this data and marks the range as PARTIAL in its tracking table. This process repeats until the last chunk is sent for the current range.

Squall will not initiate two concurrent asynchronous migration requests from a destination partition to the same source. Additionally, before issuing an asynchronous pull request, Squall checks if the entire range has been pulled by a prior reactive migration (i.e., marked as COMPLETE). If so, the request is discarded; otherwise the full request is issued as only matching data at the source will be returned. Additionally, if a transaction attempts to access partially migrated data, then this forces a reactive pull to migrate the remaining data or flush pending responses (if any).

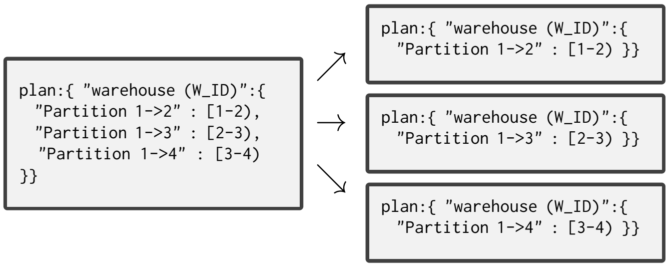
Since limiting the amount data per extraction and interleaving regular transaction execution is paramount to minimizing performance impact, the time between asynchronous data requests and the asynchronous chunk size limit are both controlled by Squall. These settings allow Squall to balance the trade-off between time to completion and impact of reconfiguration. We explore the optimal setting for these parameters in Section 7.6.

5. OPTIMIZATIONS

We next present several optimizations for improving the runtime performance of the DBMS during a reconfiguration with Squall.

5.1 Range Splitting

As described in Section 4.1, Squall computes the initial transfer ranges of tuples for a new reconfiguration by analyzing the difference between the previous and new partition plans. If the new partition plan causes a large number of tuples to move to a new location, Squall may schedule transactions on the destination partition when



```
plan:{ "warehouse (W_ID)":{
  "Partition 1->2” : [1-2),
  "Partition 1->3” : [2-3),
   "Partition 1->4” : [3-4)
}}
```

```
plan:{ "warehouse (W_ID)”:{
  "Partition 1->2” : [1-2) }}
```

```
plan:{ "warehouse (W_ID)”:{
  "Partition 1->3” : [2-3) }}
```

```
plan:{ "warehouse (W_ID)”:{
  "Partition 1->4” : [3-4) }}
```

Figure 7: A sample reconfiguration plan split into three sub-plans.

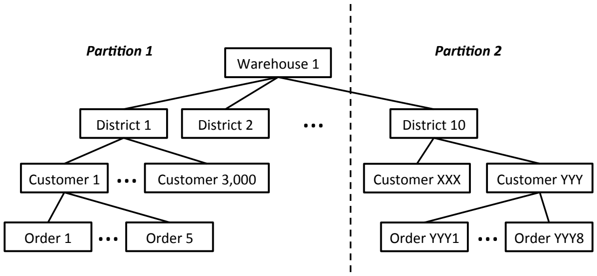


Figure 8: During reconfiguration in TPC-C, Squall uses secondary partitioning to split the DISTRICT table to avoid moving an entire WAREHOUSE entity all at once. While migration is in progress, the logical warehouse is split across two partitions, causing some distributed transactions.

they would be better served running on the source partition. For example, the following range entry requires movement of 100k tuples from the WAREHOUSE table from partition 3 to partition 4:

(WAREHOUSE, id = $[1, 100000), 3 \rightarrow 4)$

Although Squall's asynchronous migration mechanism would split this range into smaller chunks, once the first chunk is extracted the entire range is marked as PARTIAL in the source and destination partitions' tracking tables. This will cause all transactions that access data within this range to be routed to the destination partition and any query that accesses data within this range will potentially cause a reactive pull, even if the data was already migrated.

For this reason, during the initialization phase, Squall splits contiguous ranges into smaller sub-ranges, each with an expected size equal to the chunk size. Assuming that the average size of each tuple is 1 KB in our above example, with a chunk size limit of 1 MB the range would be set to $[1, 1000)$,$[1000, 2000)$,...,$[99000, 100000)$. Such smaller ranges result in fewer partial ranges, and thus reduce the likelihood that transactions are blocked unnecessarily.

5.2 Range Merging

In addition to splitting large contiguous ranges, Squall also combines small, non-contiguous ranges together to reduce the number of pull requests. For example, suppose a hotspot has formed on partition 1 for keys $[1, 10)$. The controller's load-balancing algorithm may distribute keys to other partitions in a round-robin manner. In this example, assume keys $(1, 3, 5, 7, 9)$ are migrated from partition 1 to partition 2. This would result in five migration pull requests between the partitions for a small amount of data per request. Since there is overhead and disruption to service associated with each reactive pull, issuing these small requests individually is sub-optimal. Therefore, if a table is partitioned on a unique key and has a fixed tuple size, Squall will merge small ranges into a single pull request that is composed of multiple ranges. The size of this merged range is capped to half of the chunk size limit.

5.3 Pull Prefetching

We also extend Squall's reactive pull mechanism to eagerly return more data from the source partition than is requested each time. When a pull request requests a key that is (1) partitioned on a unique column, (2) has fixed-size tuples (e.g., no varchar fields), and (3) has split reconfiguration ranges (cf. Section 5.1), Squall eagerly returns the entire range instead of the single requested key. In our experience, the additional time to extract additional tuples is substantially less than the time to schedule and process additional pull requests for the remaining of tuples in the range.

5.4 Splitting Reconfigurations

Executing a reconfiguration as a single step can cause performance problems due to contention on a single partition. Such contention can occur when a single partition is the source for many destination partitions (e.g., when moving tuples out of a hotspot partition). In this scenario, multiple destination partitions make concurrent migration requests to the same source partition. Each of these requests blocks transaction execution, thus increasing the load on the already-overloaded source partition. This contention can also occur for certain workloads, such as TPC-C, when there is a large amount of data associated with an individual partition key. These request convoys greatly degrade the DBMS's performance beyond what normally occurs during a reconfiguration.

To avoid this problem, Squall throttles data movement by splitting a large reconfiguration into smaller units. Squall first identifies the ranges of keys that need to move in order to transition to the new partition plan. It then splits these ranges into a fixed number of sub-plans where each partition is a source for at most one destination partition in each sub-plan. This is different than the splitting of reconfiguration ranges described in Section 5.1. because it reduces the number of partitions that retrieve data from a single partition.

As shown in the example in Fig. 7, the plan on the left moves data from partition 1 to partitions 2, 3, and 4. The plan is then divided into three separate sub-plans that each migrate data from partition 1 to just one partition at a time. The reconfiguration leader is responsible for generating these sub-plans and ensuring that all partitions move through the sub-plans together. The advantage of this approach is that it does not require additional coordination from a partition that is already overloaded.

For databases with a tree-based schema [37], Squall can also split reconfigurations at a finer granularity using secondary partitioning attributes. This allows the system to split up the migration of root-level keys that would otherwise require a significant number of related tuples to be migrated simultaneously. For example, the tables in TPC-C are typically partitioned by the WAREHOUSE id. But each warehouse in TPC-C may have over one million tuples associated with it. Each warehouse contains 10 DISTRICT records, however, so by partitioning the tables using their DISTRICT ids, Squall can split a warehouse into 10 pieces to limit the overhead of each data pull (cf. Fig. 8). Splitting the ranges in this way increases the number of distributed transactions in TPC-C, but avoids blocking execution for extended periods by throttling migrations.

6. FAULT TOLERANCE

Distributed DBMS like H-Store ensure high availability and fault tolerance through replicating partitions on other nodes [37]. Each partition is fully replicated at another node and must remain in sync with the primary during reconfiguration. Squall fully integrates with these master-slave replication schemes.

Squall schedules any reconfiguration operation at both the primary and its secondaries, but all data movement is done through the primary since it cannot assume that there is a one-to-one match between the number of source replicas and destination replicas. For the data extraction requests, the primary notifies secondaries when a range is partially or fully extracted so the replica can remove tuples from its local copy. Using fixed-size chunks enable the replicas to deterministically remove the same tuples per chunk as their primary without needing to send a list of tuple ids.

For data loading, the primary forwards the pull response to its secondary replicas for them to load newly migrated tuples. Before the primary sends an acknowledgement to Squall that it received the new data, it must receive an acknowledgement from all of its replicas. This guarantees strong consistency between the replicas and that for each tuple there is only one primary copy at any time.

We now describe how Squall handles either the failure of a single node or an entire cluster during a reconfiguration.

6.1 Failure Handling

The DBMS sends heartbeats between nodes and uses watchdog processes to determine when a node has failed. There are three cases that Squall handles: (1) the reconfiguration leader failing, (2) a node with a source partition failing, and (3) a node with destination partition failing. A node failure can involve all three scenarios (e.g., the leader fails and has in- and out-going data). Since replicas independently track the progress of reconfiguration, they are able to replace a failed primary replica during reconfiguration if needed. If any node fails during reconfiguration, it is not allowed to rejoin the cluster until the reconfiguration has completed. Afterwards it recovers the updated state from its primary node.

During the data migration phase, the leader's state is synchronously replicated to secondary replicas. If the leader fails, a replica is able to resume managing the reconfiguration. After fail-over, the new leader replica sends a notification to all partitions to announce the new leader. Along with this notification, the last message sent by the leader is rebroadcast in case of failure during transmission.

When a node fails while migrating data, the secondary replica replacing the partition's primary replica reconciles any potentially lost reconfiguration messages. Since migration requests are synchronously executed at the secondary replicas, Squall is only concerned with requests sent to the failed primary that have not yet been processed. When a secondary replica is promoted to a primary, it broadcasts a notification to all partitions so that other partitions resend any pending requests to the recently failed site.

6.2 Crash Recovery

During a reconfiguration, the DBMS suspends all of its checkpoint operations. This ensures that the partitions' checkpoints stored on disk are consistent (i.e., a tuple does not exist in two partitions at the same time). The DBMS continues to write transaction entries to its command log during data migration.

If the entire system crashes after a reconfiguration completes but before a new snapshot is taken, then the DBMS recovers the database from the last checkpoint and performs the migration process again. The DBMS scans the command log to find the starting point after the last checkpoint entry and looks for the first reconfiguration transaction that started after the checkpoint. If one is found, then the DBMS extracts the partition plan from the entry and uses that as the current plan. The execution engine for each partition then reads its last snapshot. For each tuple in a snapshot, Squall determines what partition should store that tuple, since it may not be the same partition that is reading in the snapshot.

Once the snapshot has been loaded into memory from the file on disk, the DBMS then replays the command log to restore the database to the state that it was in before the crash. The DBMS's coordinator ensures that these transactions are executed in the exact order that they were originally executed the first time. Hence, the state of the database after this recovery process is guaranteed to be correct, even if the number of partitions changes due to the reconfiguration. This is because (1) transactions are logged and replayed in serial order, so the re-execution occurs in exactly the same order as in the initial execution, and (2) replay begins from a transactionally-consistent snapshot that does not contain any uncommitted data, so no rollback is necessary at recovery time [21, 27].

7. EXPERIMENTAL EVALUATION

We now present our evaluation of Squall. For this analysis, we integrated Squall with H-Store [1] with an external controller that initiates the reconfiguration procedure at a fixed time in the benchmark trial. We used the April 2014 release of H-Store with command logging enabled. We set Squall's chunk size limit to 8 MB and set a minimum time between asynchronous pulls to 200 ms. We also limit the number of reconfiguration sub-plans to be between 5 and 20, with a 100 ms delay between them. The experiments in Section 7.6 show our justification for this configuration.

The experiments were conducted on a cluster where each node has a Intel Xeon E5620 CPU running 64-bit CentOS Linux with OpenJDK 1.7. The nodes are in a single rack connected by a 1GB switch with an average RTT of 0.35 ms.

As part of this evaluation, we also implemented three different reconfiguration approaches in H-Store:

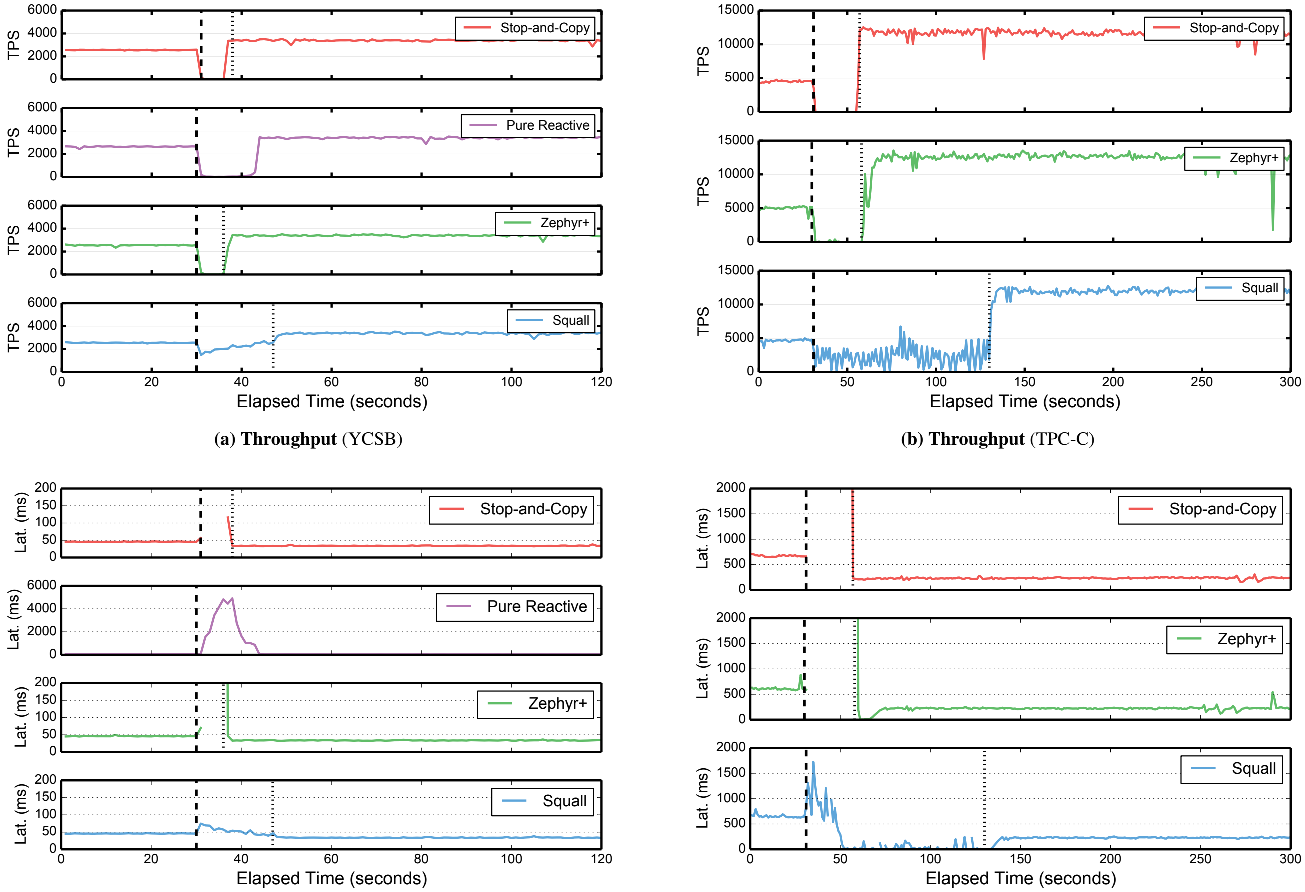
Stop-and-Copy: A distributed transaction locks the entire cluster and then performs the data migration. All partitions block until this process completes.

Pure Reactive: The system only pulls single tuples from the source partition to the destination when they are needed. This is the same as Squall but without the asynchronous migration or any of the optimizations described in Section 5.

Zephyr+: This technique combines the pure-reactive migration with chunked asynchronous pulls (cf. Section 4.5) and pull prefetching (cf. Section 5.3) to simulate pulling data in pages instead of individual keys.

The purely reactive reconfiguration approach is semantically equivalent to the Zephyr [19] migration system based on the following observations. First, Zephyr relies on a transaction execution completing at the source to begin a push-phase in addition to the reactive pull-based migration. In general for a reconfiguration, transaction execution at the source does not complete, so we stick to a pure pull-based approach. Second, Zephyr can only migrate a disk-page of tuples at a time (typically 4 KB). Squall on the other hand can migrate both single tuples and ranges. Third, there is no need to copy a "wireframe" between partitions to initialize a destination, as all required information (e.g., schema or authentication) is already present. In our experiments, the pure reactive technique was not guaranteed to finish the reconfiguration (because some tuples are never accessed) and pulling single keys at a time created significant coordination overhead. Thus, we also implemented the Zephyr+ approach that uses chunked asynchronous pulls with prefetching.

Our experiments measure how well these approaches are able to reconfigure a database in H-Store under a variety of situations, including re-balancing overloaded partitions, cluster contraction, and shuffling data between partitions. For experiments where Pure Reactive and Zephyr+ results are identical, we only show the latter.



(a) **Throughput** (YCSB)

(b) **Throughput** (TPC-C)

(c) **Mean Latency** (YCSB)

(d) **Mean Latency** (TPC-C)

Figure 9: Load Balancing – Due to a skewed workload, one partition distributes hot tuples to cold partitions. YCSB distributes 90 tuples across 14 partitions and TPC-C distributes all tuples associated with 2 warehouses to 2 different partitions.

7.1 Workloads

We now describe the two workloads from H-Store's built-in benchmark framework that we used in our evaluation. Transaction requests are submitted from 180 client threads running on separate nodes. Each client submits transactions to any DBMS node in a closed loop (i.e., it blocks after it submits a request until the result is returned). In each trial, the DBMS "warms-up" for 30 seconds and then measurements are collected for five minutes. Latencies are measured as the time from when the client submits a request to when it receives a response. The dashed vertical-line in our time-series graphs denotes the start of a reconfiguration and the light dotted line is the end of the reconfiguration.

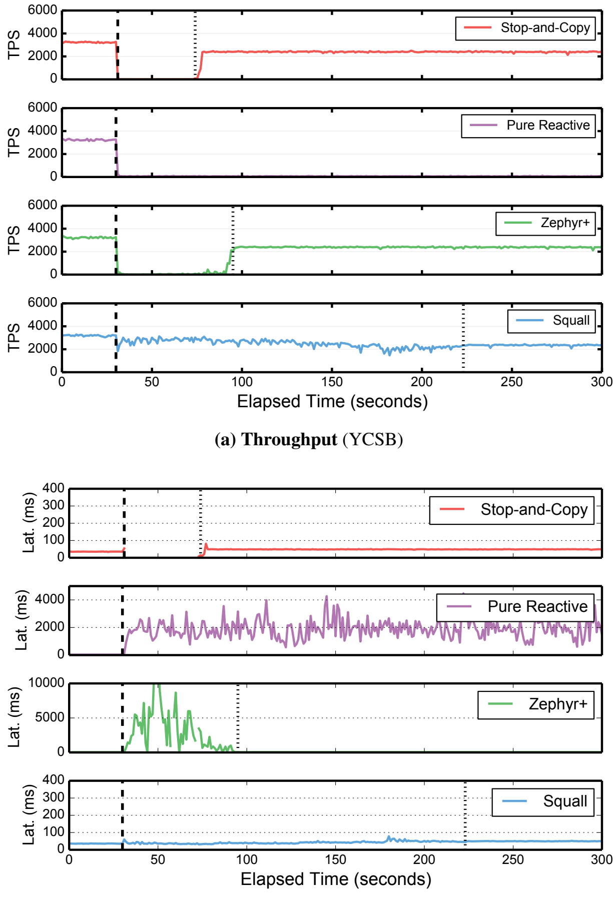
YCSB: The Yahoo! Cloud Serving Benchmark is a collection of workloads that are representative of large-scale services created by Internet-based companies [12]. For all of the YCSB experiments in this paper, we use a YCSB database containing a single table with 10 million records. Each YCSB tuple has a primary key and 10 columns each with 100 bytes of randomly generated string data. The workload consists of two types of transactions; one that updates a single record for 15% of operations and one that reads a single record for the remaining 85% of operations. Our YCSB workload generator supports executing transactions with either a uniform access pattern or with Zipfian-skewed hotspots.

TPC-C: This benchmark is the current industry standard for evaluating the performance of OLTP systems [39]. It consists of nine tables and five procedures that simulate a warehouse-centric order processing application. A key feature of this benchmark is that roughly 10% of all transactions touch multiple warehouses, which typically results in a multi-partition transaction. We use a database with 100 warehouses partitioned across three nodes.

7.2 Load Balancing

We first evaluate how well the reconfiguration approaches are able to reconfigure a skewed workload. In these tests, we create a hotspot on a single partition and then migrate hot tuples away from the overloaded partition. For YCSB, we evenly partition 10 million tuples between the four nodes and create a hotspot on 100 tuples. For TPC-C, we use a 100 warehouse database split across three nodes and then create a three-warehouse hotspot on one partition. We used a simple load-balancing algorithm that distributes hot tuples to other partitions in a round-robin manner [38].

The YCSB results in Figs. 9a and 9c show that Squall has a lower impact on the DBMS's performance than the other methods, but it takes longer to complete the reconfiguration. Initially, Squall causes the DBMS's throughput to drop by ∼30% in the first few seconds when the reconfiguration begins. This is because transactions are redirected to the new destination partition for the hotspot's tuples and then that data is pulled from their source partition. After this initial dip, the throughput steadily increases as the overloaded partition becomes less burdened until the reconfiguration completes after ∼20 seconds. The DBMS's performance for the other methods is strikingly different: they all halt transaction execution for 5–15 seconds and increase response latency significantly even though



(a) **Throughput** (YCSB)

(b) **Latency** (YCSB)

Figure 10: Cluster Consolidation – Contracting from four nodes to three nodes, with all remaining partitions receiving an equal number of tuples from the contracting node.

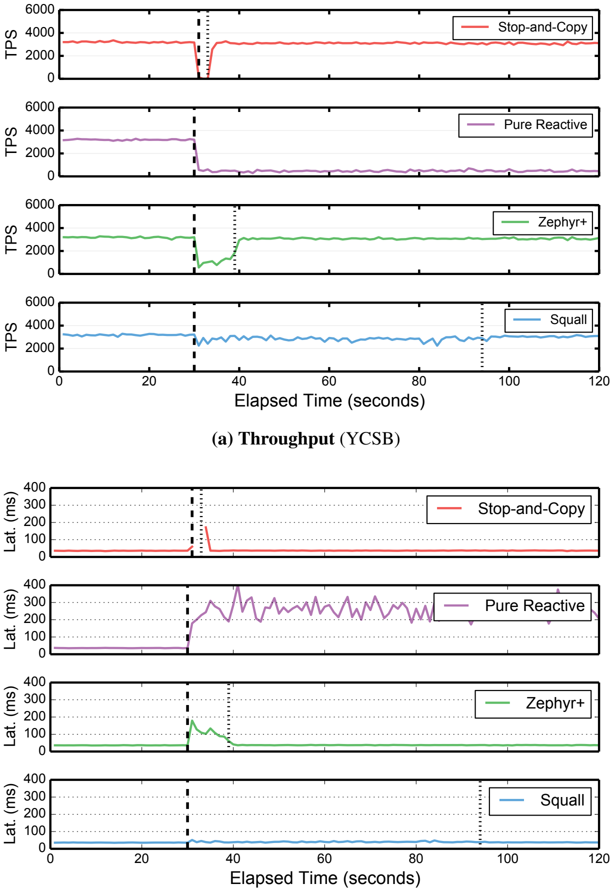
the amount of data being migrated is small. For Stop-and-Copy, this downtime results in thousands of aborted transactions. The Pure Reactive and Zephyr+ migrations cause the DBMS to "hold" transactions during the migration, which results in clients waiting indefinitely for their requests to be processed. This demonstrates that the difficulties in re-balancing an overloaded partition are not only dependent on the amount of data that is moved.

For TPC-C, Figs. 9b and 9d again show that Stop-and-Copy and Zephyr+ block execution for 24 and 30 seconds, respectively. These results also show the DBMS's performance to oscillate during reconfiguration with Squall. This is because the transactions access two large tables that cause Squall to send on-demand pull requests that retrieve a lot of data all at once. It takes the system 500–2000 ms to move the data and update indexes for these tables, during which the partitions are unable to process any transactions. This causes their queues to get backed up and results in short periods during the reconfiguration when no transaction completes. It is these distributed transactions in TPC-C that increase the likelihood that a partition will block during data migration. As described in Section 5.4, splitting a single reconfiguration into smaller sub-plans minimizes contention on a single partition and reduces the amount of data pulled in a single request by splitting up large ranges. These results demonstrate how disruptive a large pull can be for methods that lack this functionality.

7.3 Cluster Consolidation

We next measure the performance impact of the reconfiguration methods when the number of nodes in the cluster contracts for a fixed workload. For this experiment, we start with a YCSB database that is partitioned across four nodes. The tuples are evenly divided among the partitions and clients access tuples with a uniform distribution. After 30 seconds, the system initiates a reconfiguration that removes one of the nodes. This causes the DBMS to move the data from the removed node to the other three remaining nodes.

The results in Fig. 10 show that Pure Reactive never completes the reconfiguration and the DBMS's throughput is nearly zero. This is because transactions access tuples uniformly, and thus every transaction causes another on-demand pull request to retrieve a single tuple at a time. The DBMS's throughput also drops to nearly zero with Zephyr+ during the reconfiguration because the partitions on the remaining nodes all try to retrieve data at the same time. This causes the system to take longer to retrieve the data for each pull request. Squall alleviates this bottleneck by limiting the number of concurrent partitions actively involved in the reconfiguration through splitting the reconfiguration into many steps. This results in Squall's reconfiguration taking approximately 4× longer than Stop-and-Copy. We contend that this trade-off is acceptable given that the DBMS is not down for almost 50 seconds during the reconfiguration. Likewise, we believe that Squall's consistent performance impact shows that it is well-suited for both load balancing and contraction reconfigurations that do not have a tight deadline.



(a) **Throughput** (YCSB)

(b) **Latency** (YCSB)

Figure 11: Data Shuffling – Every partition either loses 10% of its tuples to another partition or receives tuples from another partition.

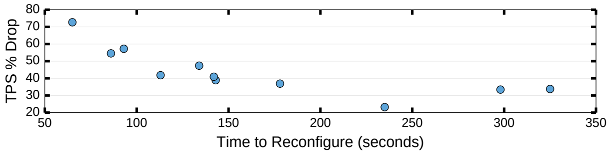


Figure 12: Duration vs. Degradation – Measuring the mean throughput percentage decrease while in reconfiguration compared to out of reconfiguration for a YCSB workload contracting the number of nodes. The decrease in reconfiguration time is controlled by limiting the number of sub-plans and delay between asynchronous requests.

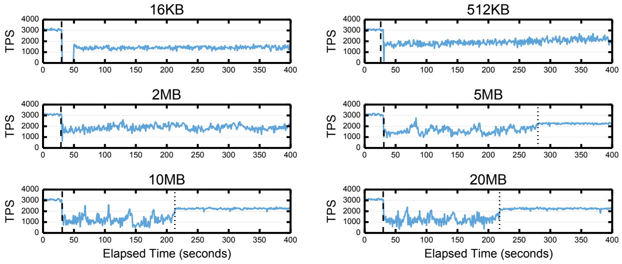


Figure 13: Chunk Size Analysis – The impact of chunk sizes on time to complete reconfiguration and throughput for YCSB workload.

7.4 Data Shuffling

In this experiment, we evaluate how well the methods are able to shuffle data between pairs of partitions. This is to simulate moderate data reconfiguration without creating migration dependencies on a single partition. We again use a YCSB database with 10 million tuples evenly split on 16 partitions and execute a workload that accesses tuples with a uniform distribution. The reconfiguration plan then causes eight pairs of partitions to exchange 10% of their tuples with each other (i.e., 62,500 tuples per partition).

The throughput and latency measurements in Fig. 11 show that Stop-and-Copy is able to quickly migrate data in only a few seconds. This downtime may still be unacceptable for highly available applications: if a system needs "five nines" (99.999%) of uptime, each week the system can only afford 6 seconds of downtime. As in Section 7.3, Pure Reactive never completes because of transactions' uniform access patterns and the overhead of small data pulls. The DBMS's performance with Zephyr+ is better than the previous trials, largely due to the lack of multiple partitions pulling data from a single partition. Moving data from partition to another is the ideal case for Zephyr [19]. But again we see that despite requiring longer to complete, Squall only causes a ∼10% decrease in throughput. We explore this issue further in the next experiment.

7.5 Reconfiguration Duration vs. Degradation

The results from the experiments thus far show that there is an inherent trade-off in live reconfiguration between reconfiguration duration and performance degradation. One of Squall's goals is that it seeks to efficiently amortize the impact of data migration by using a combination of on-demand data pulls and periodic asynchronous data pulls. We have tuned Squall to minimize performance impact for a variety of workloads through the aforementioned parameters. But if an administrator wants a shorter reconfiguration time, then they can tune Squall to be more aggressive in moving data at the cost of a greater disruption in the DBMS's performance. To better understand this trade-off, we used the same reconfiguration scenario with the YCSB workload from Section 7.3. We then vary how quickly Squall completes the reconfiguration by decreasing the number of sub-plans it generates and the amount of time the system waits between issuing asynchronous pull requests.

The graph in Fig. 12 shows the reconfiguration duration plotted against the DBMS's throughput degradation relative to its throughput before the reconfiguration starts. As expected, the faster reconfigurations cause a greater drop in performance. This indicates that a longer reconfiguration time may be preferable for applications concerned with high availability. One could argue, however, that there are two downsides to an extended reconfiguration: (1) snapshots may not be taken while reconfiguration is in progress, so recovery could take longer in case of failure, and (2) new reconfigurations may not be started while an existing reconfiguration is in progress, so the system may take longer to respond to changes in the workload. We contend that this trade-off is acceptable given the performance improvement during reconfiguration, and furthermore, we do not consider either of these downsides to be a major issue. In our experience, most H-Store users rely on replicas for fault tolerance, and almost never recover from snapshots. Even in the worst-case scenario where recovery from a snapshot is required, reconfiguration is not likely to significantly extend the timeline. Regarding the second potential downside, if the load-balancing algorithm is effective, there should not be a need for frequent reconfigurations. Even if the system must reorganize the database often, we do not expect the duration to be an issue as the system controller needs to observe a steady post-reconfiguration state before load-balancing [38].

7.6 Sensitivity Analysis

Lastly, we evaluate how Squall's tuning parameters described in Sections 4 and 5 affect the DBMS's performance and the length of the reconfiguration. For all of these experiments, we use the YCSB workload with the same reconfiguration plan used in Section 7.3 that contracts the cluster from four to three nodes.

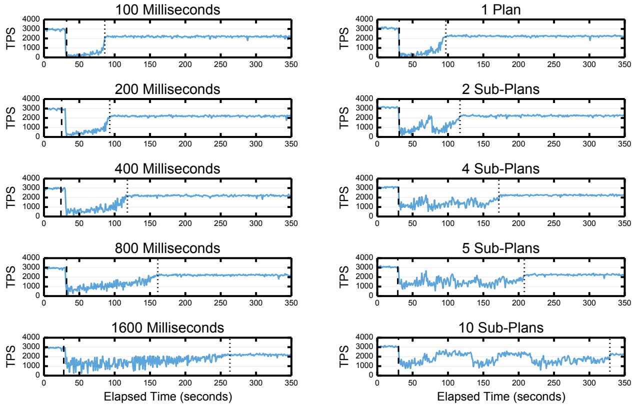
We begin with comparing the different maximum chunk sizes for pull requests. The results in Fig. 13 show that when the chunk size is 2 MB or less, reconfiguration takes an unacceptably long time. Above 10 MB, however, the total length of reconfiguration does not improve significantly and throughput degrades slightly. For this reason, we select 8 MB as our default chunk size.

Next, we measure Squall's performance with different settings for (1) the delay between asynchronous requests and (2) the number of sub-plans that it generates for a reconfiguration plan. These are the two settings that have the greatest affect on the DBMS's performance during the reconfiguration and the time it takes for Squall to complete data migration. For the former, the results in Fig. 14a show that separating asynchronous pulls by 200 ms provides the best balance of throughput versus reconfiguration time. In the same way, the results in Fig. 14b show that creating five sub-plans is the optimal choice for this workload which we set as our minimum. Although we do not show experiments here due to space constraints, we found that separating consecutive sub-plans by 100 ms also achieves this nice balance.

These experiments show that our chosen settings are appropriate for the YCSB workload. We found that they also proved to be ideal for TPC-C. Depending on the application, administrators may choose to tune these parameters to further minimize reconfiguration time at the expense of performance during migration, or alternatively stretch out the reconfiguration time in order to maximize system performance. We defer the problem of automating this tuning process in Squall as future work.

8. RELATED WORK

There are several existing protocols for migrating data between non-partitioned DBMSs [18]. The simplest approach, *stop-and-copy*, halts all execution on the source node and performs a tuple- or



(a) Asynchronous Pull Delays

(b) Number of Sub-plans

Figure 14: Reconfiguration Parameter Analysis – The impact in Squall on the time to complete reconfiguration and throughput for two parameters: (14a) the delay between asynchronous pulls and (14b) the number of sub-plans generated. The target workload is YCSB using the cluster consolidation plan Section 7.3. The max chunk size is 8 MB.

page-level copy to the destination. All new requests are re-directed to the destination node. An alternative, *flush-and-copy* [15], first flushes dirty records and then marks the source node as read-only during the migration. Any transaction that performs an update is restarted at the destination node.

Synchronous migration [18] relies on replication to synchronize the source and destination. In this migration, the destination is added as a new eager replica for the source. Depending on the implementation, a mechanism, such as log shipping or snapshots, is used to bring the destination "up to speed" with the source. Once the persistent state of each node is fully synchronized, transactions are now eagerly committed at source and destination. The source now can use a fail over mechanism to make the destination the primary replica and complete the migration process.

Several *live migration* techniques have been proposed to migrate databases with minimized interruption of service and downtime. Designed for systems with shared storage, Albatross [15], copies a snapshot of transaction state asynchronously to a destination node. Updates to the source are then iteratively shipped to the destination, until either a period of no updates arrives or a conditional convergence occurs. A small period of downtime at the source is initiated while the remaining states are copied to the destination. Slacker [8] is another approach that is optimized for minimizing the impact of migration in a multi-tenant DBMS by throttling the rate that pages are migrated from the source to destination. Slacker uses recovery mechanisms to stream updates from the source to the destination. To avoid straining the other tenants at migrating nodes, a PID controller monitors average transaction latency to adjust throttling the network connection used to stream the updates.

Zephyr [19] allows concurrent execution at the source and destination during migration, without the use of distributed transactions. A wireframe of the database's schema (e.g., an index's physical data structures) are copied to the destination upon initialization. As transactions at the source node complete, new transactions are routed to the destination, resulting in concurrent transaction execution at both nodes. During migration, requests at the destination that require leaf nodes force a pull on the data page from the source; any transaction at the source accessing a page that has been migrated to the destination must restart at the destination. 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 [35] 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 employs both the reactive and asynchronous data migration techniques that we use in Squall but for a distributed MySQL cluster [23]. Minhas et al. propose a method for VoltDB that uses statically defined virtual partitions as the granule of migration [28]. This technique was also explored in other work from the same research group [34].

Intra-machine reconfiguration has been explored in the context of partitioning data structures between cores to improve concurrency [31]. While the goals are similar to Squall, the movement of data and constraints encountered are drastically different.

NuoDB [4] is a disk-oriented, distributed DBMS that splits all components of a database (e.g., tables, indexes, meta-data, etc.) into "atoms" that are spread out across a shared-nothing cluster [30]. For example, tuples for a single table are combined into 50KB atoms. These atoms then migrate to nodes based on query access patterns: when a node processes a query, it must retrieve a copy of the atoms with the data that the query needs to access from a storage node. Over time, the atoms that are used together in the same transaction will end up on the same executor node. The system does not need a migration scheme such as Squall, but it is unable to do fine-grained partitioning.

There are several ways that a DBMS could manage the dynamic location information of migrating data during a reconfiguration: (1) a globally consistent tracker that maintains the active owning partition for migrating data [23], (2) a distributed transaction is invoked at both potentially owning partitions, (3) transactions are scheduled at the partition according to the old plan, which tracks modifications, or (4) transactions are scheduled at the partition according to the new plan, which *reactively* pulls data on-demand [19,35]. With the first option, in a large system with many partitions and tuples, maintaining a global view of this information of individual tuples at each partition becomes untenable due to the overhead of communicating ownership between partitions. Furthermore, the storage overhead of tracking these tuples can be expensive in a main memory DBMS, especially if the reconfiguration was initiated because a node was running out of memory, because an expensive migration might exacerbate the problem [20]. The second option will increase latency due to additional nodes being involved in a transaction [14] and does nothing to move the migration forward. The third option has been used in prior live migration solutions [11,15], but it requires the old node to ship modifications to the new node and requires a synchronized downtime to transfer ownership and ship modifications to the new owner.

9. CONCLUSION

We introduced a new approach, called Squall, for fine-grained reconfiguration of OLTP databases in partitioned, main memory DBMSs. Squall supports the migration of data between partitions in a cluster in a transactionally safe manner even in the presence of distributed transactions. We performed an extensive evaluation of our approach on a main memory distributed DBMS using OLTP benchmarks. We compared Squall with a naive stop-and-copy technique, a pure-reactive reconfiguration, and a reconfiguration similar to the live-migration technique, Zephyr. The results from our experiments show that while Squall takes longer than the baselines, it can reconfigure a database with no downtime and a minimal overhead on transaction latency. The benefits of Squall are most profound when there is a significant amount of data associated with a single partition attribute or many partitions are migrating data from a single source.

10. REFERENCES

[1] H-Store. http://hstore.cs.brown.edu.

[2] MemSQL. http://www.memsql.com.

[3] MongoDB. http://mongodb.org.

[4] NuoDB. http://www.nuodb.com.

[5] VMware vFabric SQLFire. http://www.vmware.com/go/sqlfire.

[6] VoltDB. http://www.voltdb.com.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[8] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüs, and P. J. Shenoy. "Cut me some slack": latency-aware live migration for databases. In *EDBT*, pages 432–443, 2012.

[9] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *VLDB*, pages 285–300, 1980.

[10] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39:12–27, 2011.

[11] C. Clark et al. Live migration of virtual machines. In *NSDI*, pages 273–286, 2005.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[13] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, pages 21–34, June 2012.

[14] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[15] 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):494–505, May 2011.

[16] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[17] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, pages 1243–1254, 2013.

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

[19] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, pages 301–312, 2011.

[20] N. Folkman. So, that was a bummer. https://web.archive.org/web/20101104120513/http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/, October 2010.

[21] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[22] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[23] E. P. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis, MIT, 2011.

[24] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[25] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. *DPDS*, pages 177–187, 1988.

[26] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.

[27] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 604–615, March 2014.

[28] 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*, pages 281–288, 2012.

[29] C. N. Nikolaou, M. Marazakis, and G. Georgiannakis. Transaction routing for distributed OLTP systems: survey and recent results. *Inf. Sci.*, 97:45–82, 1997.

[30] NuoDB LLC. *NuoDB Emergent Architecture – A 21st Century Transactional Relational Database Founded On Partial, On-Demand Replication*, Jan. 2013.

[31] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: Page latch-free shared-everything oltp. In *PVLDB*, volume 4, pages 610–621, 2011.

[32] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[33] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proc. VLDB Endow.*, 5:85–96, October 2011.

[34] T. Rafiq. Elasca: Workload-aware elastic scalability for partition based database systems. Master's thesis, University of Waterloo, 2013.

[35] O. Schiller, N. Cipriani, and B. Mitschang. Prorea: live database migration for multi-tenant rdbms with snapshot isolation. In *EDBT*, pages 53–64, 2013.

[36] R. Stoica, J. J. Levandoski, and P.-A. Larson. Identifying hot and cold data in main-memory databases. In *ICDE*, pages 26–37, 2013.

[37] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*, pages 1150–1160, 2007.

[38] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proc. VLDB Endow.*, 8:245–256, November 2014.

[39] The Transaction Processing Performance Council. TPC-C benchmark (Version 5.10.1), 2009.

[40] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[41] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *HPTS*, 1997.

APPENDIX

A. CORRECTNESS

Transaction execution in a partitioned main-memory database provides correctness by ensuring that a single partition executor (transaction manager) is exclusively responsible for a set of tuples and transactions on these tuples execute in a serial order. Distributed transactions effectively lock each partition executor so that transactions can be executed using two-phase commit with exclusive access. The exclusive data access on a set of tuples with serial execution ensures that phantoms are not possible and transactions are serializable. Let x be the set of tuples that correspond to a single reconfiguration range, or the set of tuples from a single relation scheduled to move between partitions, identified by a range of partition keys (e.g. W_ID > 3 and W_ID < 5). Without loss of generality x can be a single tuple for a single partitioning key (e.g. a warehouse tuple with W_ID = 4). Let the current partition plan be i, and the previous partition plan be $i-1$. Let the logical **owner** of x in the partition plan i be some partition $p_n \in P$, determined by a function $O_i(x) \rightarrow p_n$. Since reconfigurations are not executed as atomic transactions and migrating data may be broken into chunks, the previous owning partition $O_{i-1}(x)$ may physically have none, some, or all of the tuples in x. Therefore, a partition p_n can **check** if it physically owns x at any given time by the function $C(x, p_n) \rightarrow (true, partial, false)$. Only a partition that physically owns x can execute a transaction that reads or writes tuples in x, regardless of whether another partition expects to own the tuple according to O_i. This leads us to following axiom.

AXIOM 1. *A partition p_n can read or write x if $(O_i(x) = p_n \vee O_{i-1}(x) = p_n) \wedge C(x, p_n) = true$.*

In the following discussion, we say that data is *extracted* from the source partition p_s and *loaded* into the destination partition p_d.

A partition p_s marks its physical ownership of x as $false$ or $partial$ as soon as the migration process fully or partially extracts x from p_s. Conversely, a partition p_d only marks ownership of x as $true$ once the migration process has finished loading every tuple in x into p_d. Since transaction execution and migration extraction operations are executed serially, a transaction can never execute on p_s at the same time as migrating data is being extracted. However, data can be loaded while a transaction is blocked for data migration on p_d. This provides us with the following axiom.

AXIOM 2. *Data extraction (may change the value of $C(x, p_s)$) and transaction execution are mutually exclusive.*

During reconfiguration, migrating tuples can temporarily be in "flight" or in a queue at the destination partition ($O_i(x)$), no two partitions can ever concurrently have physical ownership of x. This means there can be a period of time where no partition physically owns x, however in this scenario no partition can execute a transaction involving x until the tuples are fully loaded. Axiom 2 provides us with the following lemmas.

LEMMA 1. $(O_{i-1}(x) = p_s) \wedge (O_i(x) = p_d) \wedge (p_s \neq p_d) \wedge (C(x, p_s) = partial) \implies C(x, p_d) \in \{partial, false\}$

LEMMA 2. $(O_{i-1}(x) = p_s) \wedge (O_i(x) = p_d) \wedge (p_s \neq p_d) \wedge (C(x, p_s) = false) \implies C(x, p_d) \in \{true, partial, false\}$

LEMMA 3. $(O_{i-1}(x) = p_s) \wedge (O_i(x) = p_d) \wedge (p_s \neq p_d) \wedge (C(x, p_s) = true) \implies C(x, p_d) = false$

Using these lemmas it can be shown that at all times a set of tuples x are physically owned by at most one partition. Furthermore, a user transaction T cannot read or write tuples belonging to x unless x is physically owned by a partition. Any reconfiguration (or migration event) of x with respect to T will either completely follow or precede T. By Axiom 1 we know a transaction cannot operate on x without fully owning the set, and therefore a phantom problem could only arise if some tuples in x were extracted during transaction execution. By way of contradiction, suppose that a transaction starts at p_s which owns x ($C(x, p_s) = true$), but by commit time, p_s has lost some or all the tuples in x ($C(x, p_s) \in \{partial, false\}$). This would mean that the migration extraction was concurrent with the transaction, which contradicts Axiom 2. Additionally, Squall is not susceptible to lost updates as a record in x could only be inserted or updated by the owning partition by Axiom 1. This establishes that there are no phantom or lost update problems with Squall.

Since transactions and migration extractions are mutually exclusive and serially executed, we cannot have any other cyclic conflict dependencies between transactions involving migrations. A cycle in a conflict dependency graph would mean that transaction execution had an extraction included which cannot happen by Axiom 2. Therefore, we can assert that transaction execution remains serializable with Squall. It is worth noting that in some cases a distributed transaction that triggers a pull request between subordinates can cause a deadlock, depending on how partition locks are acquired. After the distributed transaction aborts, due to transaction order the migration pull is resolved before the distributed transaction restarts and hence avoiding livelock.

With regard to fault-tolerance, Squall is safe and will make progress if all failures are such that for every partition node only loses the primary replica or secondary replica(s). As described in Section 6.1, if partition fails, a replica partition can seamlessly resume the reconfiguration due to synchronously replicating the state and data of migrating tuples. If for a partition both the primary and the secondary replicas fails or if the entire cluster crashes, Squall will have to wait until the recovery of "last node to fail" for the reconfiguration to complete the reconfiguration. Otherwise the Squall is blocked.

B. EVALUATING OPTIMIZATIONS

Fig. 15 shows how the optimizations discussed in Section 5 work together to improve Squall's performance during reconfiguration. With none of the optimizations, throughput during reconfiguration is not much better than the baseline approaches discussed in Section 7 – which benefit from Squall's sizing and migration scheduling. Likewise, each individual optimization provides limited improvement beyond the unoptimized system. But by combining the optimizations, we show in Fig. 15 how they build upon each other to help Squall reconfigure the system with minimal impact on performance.

For example, the eager pull mechanism discussed in Section 5.3 will not perform well if the reconfiguration ranges are too large or too small. But by combining it with the range splitting and merging optimizations from Sections 5.1 and 5.2, we can ensure that the sizes of the reconfiguration ranges are optimal for eager pulls. Splitting the reconfiguration plan into several smaller plans as described in Section 5.4 serves to keep any single partition from becoming a bottleneck, and as a result further improves throughput during reconfiguration.

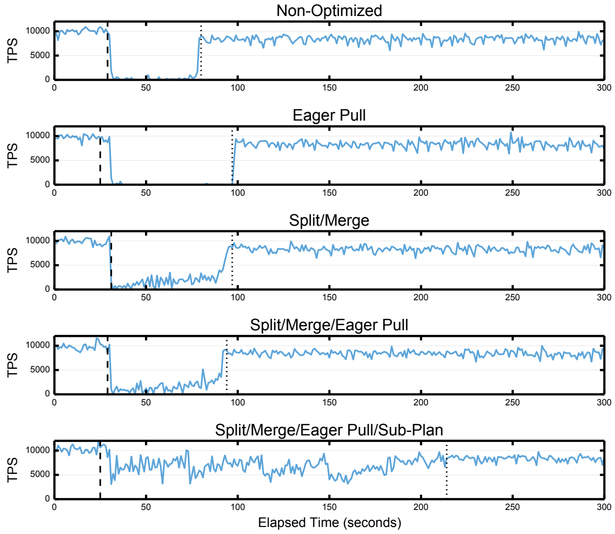


Figure 15: A demonstration of how each optimization discussed in Section 5 improves transaction throughput during reconfiguration. This shows the impact of the optimizations when the system contracts from 4 servers to 3 under a high throughput YCSB workload.

C. ALTERNATIVE PARTITIONING SCHEMES

Our current implementation of Squall assumes that the DBMS uses range-based partitioning to horizontally split tables. It is non-trivial to extend it to work with alternative partitioning schemes, such as hash and round-robin partitioning. The major difference, however, is that under these schemes the source partitions would be responsible for determining which tuples will migrate for a reconfiguration during the initialization phase. This is because it is the only place in the system that knows what data it has. After the source partition computes this information, it is sent to the leader as part of the acknowledgement that the partition is ready to proceed with the reconfiguration. The leader node then disseminates this this information to all partitions with the acknowledgment that reconfiguration is starting. With this, each partition build the list of their expected incoming tuples. This process is less efficient than our range-based scheme used in this paper because it requires that the partitions check *all* of their tuples to determine if they are to be moved. Using extensible or consistent hashing can reduce the number of tuples that need to be checked but would still require more than our approach.

D. ACKNOWLEDGEMENTS

This work was partially funded by the NSF under the IIS-1018637 grant and QCRI under the QCRI-CSAIL partnership. We would like to thank the reviewers, whose feedback significantly improved the paper. We also thank Ken Salem, Ashraf Aboulnaga, Marco Serafini, Essam Mansour, Jennie Duggan, Mike Stonebraker, and Sam Madden for their suggestions and feedback. Andy sends big ups to the Brown DB squadron, especially Kaiser Kraska, L.T.D. Ugur, and Big Steezy Z.