MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems

Yu-Shan Lin
Nation Tsing Hua University, Taiwan, ROC
yslin@datalab.cs.nthu.edu.tw

Shao-Kan Pi
Nation Tsing Hua University, Taiwan, ROC
skpi@datalab.cs.nthu.edu.tw

Meng-Kai Liao
Nation Tsing Hua University, Taiwan, ROC
mkliao@datalab.cs.nthu.edu.tw

Ching Tsai
Nation Tsing Hua University, Taiwan, ROC
ctsai@datalab.cs.nthu.edu.tw

Aaron Elmore
University of Chicago, Chicago, IL, USA
aelmore@cs.uchicago.edu

Shan-Hung Wu
Nation Tsing Hua University, Taiwan, ROC
shwu@cs.nthu.edu.tw

ABSTRACT

Recent *deterministic database systems* have achieved high scalability and high availability in distributed environments given OLTP workloads. However, modern OLTP applications usually have changing workloads or access patterns, so how to make the resource provisioning elastic to the changing workloads becomes an important design goal for a deterministic database system. *Live migration*, which moves the specified data from a source machine to a destination node while continuously serving the incoming transactions, is a key technique required for the elasticity. In this paper, we present MgCrab, a live migration technique for a deterministic database system, that leverages the *determinism* to maintain the consistency of data on the source and destination nodes at very low cost during a migration period. We implement MgCrab on an open-source database system. Extensive experiments were conducted and the results demonstrate the effectiveness of MgCrab.

PVLDB Reference Format:
Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, Shan-Hung Wu. MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems . *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

Distributed, deterministic database systems [18,38,39] have been proposed to meet the requirements of modern online transaction processing (OLTP) applications. The *determinism*, i.e., the database state on each machine/node being deterministic to a total transaction order at any time, saves a system from using an expensive agreement protocol (e.g., two phase commit, 2PC) to achieve strong consistency between replicas [38], and enables lightweight processing of distributed transactions across partitions [38,39]. This makes the system highly available and scalable, and a superb container of various OLTP databases, either as a large database storing data of global users or a large collection of tenants having small footprints (both in terms of size and workload requirements).

Modern OLTP applications usually have changing workloads or access patterns [9,29,36]. For example, users of a database around the world may become active at different times due to timezone differences and may access their data differently; on the other hand, a tenant in a multi-tenant database may suddenly become very hot as the owner application gains flash crowds originating from viral popularity. *Elastic load balancing*, which moves data around the provisioned machines and increases/reduces the provisioning in response to changes of workload to maintain the system performance, becomes an important design goal of a deterministic database system [10]. Elasticity also helps minimize the operation cost when the system is deployed on a cloud platform that charges per use.

Live migration [3, 12–15, 24, 34] is a fundamental step toward elasticity. Given a *migration plan* for moving some data from a source node to a destination node, it moves the data while continuously serving the incoming transactions so to keep the system alive (available). Currently, live migration techniques can be divided into those running an incoming transaction on the source machine [3,12,14,24], destination machine [15,34], or on either one of the two machines [13].[1] The source-based techniques [3,12,14,24] have the advantage that the transaction can run smoothly in the beginning of migration because the data are likely to be available locally and the cache is warm. However, these approaches have a termination problem—the data migration may never end as the updates made by transactions running on the source node need to be constantly sent to the destination node (by shipping logs or snapshots). To terminate the migration, these techniques usually introduce a short period of downtime and then copy the final updates from source to destination. After that, the destination node starts serving transactions and the system becomes available again. The destination-based approaches [15,34] avoid such system downtime by letting the destination node serve transactions immediately after the migration starts. Nevertheless, the transaction execution will be slow in the beginning of the migration period since most data are not available at the destination yet and need to be pulled from the source node. This results in degraded system throughput. Squall [13], the third approach, extends the destination-based methods by carefully tracking the location of data being migrated

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 45th International Conference on Very Large Data Bases, August 2019, Los Angeles, California.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx

Copyright 2018 VLDB Endowment 2150-8097/18/10... $ 10.00.

DOI: https://doi.org/TBD

[1] This categorization is slightly different from the one used in previous work as we focus on where to run a *single* transaction arriving after the migration starts.

and allowing a transaction to run on the source node if the data accessed by the transaction locate entirely on the source node. This mitigates the performance drop as some transactions in the system can run faster now.

However, in practice, Squall still renders a considerable performance drop during the migration period [13]. The system may fail to meet an SLA and incur significant financial losses [2]. Furthermore, with Squall and most existing live migration techniques, there is a trade-off between performance and migration time (i.e., the duration of the migration period)—the shorter the migration time, the larger the performance drop can be during the migration period. This is because the migration process usually contends with running transactions on the migrating data. For example, suppose the data being migrated are broken into chunks and transmitted one-by-one. On the destination node, the process that persists a data chunk C will block all transactions accessing C. To meet the SLA, one could set the chunk size small such that the migration process will block fewer transactions (as fewer transactions may access data in C) for a smaller amount of time (as it takes less time to persist C). But this can prolong the migration period as there is a higher communication overhead to transmit a larger number of small chunks than a small number of large chunks. The prolonged migration time may prevent the system from reacting to the changing workload in a timely manner. There is a crucial need for a new live-migration technique that is more transparent (in terms of performance degradation) and that avoids such a trade-off.

In this paper, we present MgCrab, a technique for live migration in a shared-nothing distributed, deterministic database system. Unlike most existing live migration approaches that execute an incoming transaction on either the source or destination machine, MgCrab executes the single transaction on *both* nodes.[2] The client who issues the transaction will get results once *any* of the nodes completes the transaction. We call the node which answers faster the *winner* node. At first glance, running a transaction on both machines does not seem to be a good idea because it wastes computing resource and introduces the complexity of synchronizing results between nodes for consistency. However, we argue that this can actually lead to better resource utilization, as well as a simpler design. Migration typically happens in either a scaling-out or consolidation scenario, where at least one machine is under-utilized (the destination node in the scaling-out cases and both nodes in the consolidation cases). Running the transaction on both nodes does *not* saturate the system. In addition, by leveraging determinism, we can let the transaction produce exactly the same results on the source and destination nodes without using expensive agreement protocols (e.g., 2PC) to maintain consistency. We can also avoid the complex ownership tracking [13] used to decide which machine runs a transaction best.

MgCrab offers the following advantages as compared to the previous work. First, it allows the transactions to run on the source node to avoid a performance drop at the beginning of the migration period. When the destination node catches up later on, it hands over the winner node *seamlessly* on a per transaction basis (whenever the transaction on destination completes more quickly) without incurring downtime or aborted transactions. Second, we let the destination node "compute" the updates of the migrated data (by executing the transactions) instead of waiting for the updates shipped from the source. This saves the shipping cost (e.g., latency) and avoids the termination problem as in the source-based approaches. More importantly, MgCrab employs the *two-phase background pushes* that migrate chunks *without* blocking any transaction on the source node. This significantly improves the stability of system performance during the migration. The following summarizes our contribution:

- We present MgCrab that lets *both* the source and destination machines execute every incoming transaction during the migration period, and discuss how determinism can simplify the design of such a live-migration technique.
- Based on MgCrab, we propose the two-phase background pushing that avoids a common performance bottleneck in existing systems when migrating the cold data.
- We prove the correctness of MgCrab by showing its *safety* (i.e., correctness in the presence of failure) and *liveness* (i.e., migration eventually completes, despite failures occur) guarantees.
- We discuss many practical considerations, such as supporting distributed transactions, range queries, pipelining, etc., to make MgCrab work under various situations.
- We implement MgCrab on an open-source deterministic database system and conduct extensive experiments. The results show that MgCrab offers a much more stable performance during the migration period as compared to the state-of-the-art live migration techniques.

The rest of this paper is organized as follows. Section 2 reviews the architecture of a deterministic database system and existing migration techniques. Section 3 introduces MgCrab while Section 4 discusses some practical considerations. Section 5 evaluates the performance of MgCrab and Section 6 reviews more related work. Finally, Section 7 concludes the paper.

2. BACKGROUND

We first review the architecture of Calvin [38], an example of the deterministic database systems targeted by MgCrab. We then show how current live migration techniques are insufficient for this type of systems. Although we use Calvin in this paper, MgCrab is applicable to any deterministic database system, such as H-Store [18].

2.1 System Architecture

Calvin is designed for workloads that contain transactions only invoked as predefined stored procedures. As shown in Figure 1, data are fully replicated across multiple data centers separated geographically for availability. A data center has a cluster of machines, each of which holds only one partition of data to avoid hot spots. When transaction requests arrive, the sequencers in the system communicate with each other to decide a single *total order* (i.e., an order that all machines agree on) of these requests, and forward the requests to all schedulers following the total order. The scheduler on each machine, after receiving a request, analyzes the request and decides whether the transaction will touch any data stored locally, and if so, forwards the request to the executor residing on the same machine. Then, the executor processes the transaction while ensuring that the results will be the same as those of processing the transaction (and all its preceding transactions) following the total order. In other words, the database state on each node at any time is *deterministic* to the total ordering. This enables lightweight distributed transactions [38] across partitions (at the serializable isolation level) and low-cost strong consistency between replicas. The latter comes with the fact that two replicas having

[2] We use the word "crab" in the name to emphasize where a transaction runs before (on source), during (on both), and after migration (on destination), which proceeds like a walking crab.

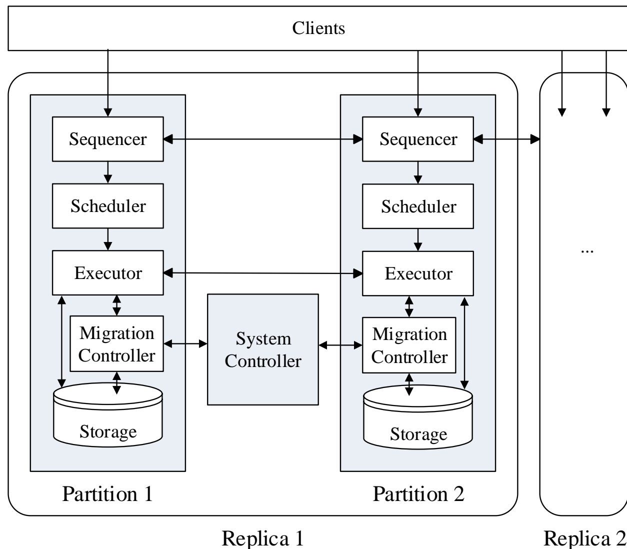


Figure 1: Cooperative transaction execution in a deterministic database system.

identical initial data will produce the exactly same results for every transaction (due to the determinism), and therefore need *no* agreement protocol (e.g, 2PC) to commit a transaction. Note that the delay incurred by the total-ordering protocol (e.g., Paxos [21] or Zab [17]) for ordering a transaction does *not* count into the contention footprint of the transaction in concurrency control. Therefore, such delay has no impact on the system throughput. Also, note that this architecture supports any storage engine (either memory- or disk-based) with the CRUD interface.

Elastic Load Balancing. Studies [13, 39] have shown that a deterministic database system is susceptible to load imbalances, and for this reason, it is important for the system to have components for elastic load balancing. We assume that inside each replica (data center), there is a system controller that collects statistics of system components in the background and decides when and how the data should be re-partitioned to fit the current workload. The controller periodically sends a *partition plan* to every partition in the replica. A partition plan specifies the data partition for every node so each partition can spot distributed transactions. After receiving the partition plan, the migration controller on each partition compares the old and new partition plans to derive *migration plans*, each of which specifies partial data being migrated from a source node to a destination node. A partition needs to infer only the migration plans that are relevant to itself (i.e., those where the partition is either the source or destination). Then, for each migration plan, the migration controller on the source node triggers a migration task that runs a live migration algorithm. After all migration tasks are completed, the migration controller on each partition sends an acknowledgment to the system controller so the system controller can generate the next partition plan.

2.2 Migration Cost

Data migration has a cost that includes the following [10]: downtime (system or database service outage), number of transactions aborted, increase in transaction latency, drop in throughput, and transfer of extra data (in addition to those specified in the migration plan). Also, there is a trade-off between the migration time and the migration cost. Normally, the shorter the migration time, the higher the migration cost. A live migration technique is expected to minimize the migration cost while avoiding its trade-off with migration time.

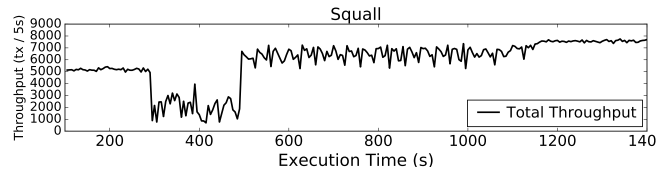


Figure 2: With TPC-C workload, Squall [13] renders a drop in throughput in the binning of the migration period.

2.3 Some Migration Techniques

We now discuss some typical data migration techniques.

Stop-and-Copy. This is arguably the simplest and most heavy-handed approach to migrate data. The system stops accepting transaction requests on the source node, moves the data from source to destination, and then re-start serving requests on the destination node. This technique incurs a long downtime lasting the entire migration period. It may also slow down the transactions and throughput on the destination node after service restarts, as the node may require some time to warm up the cache. However, in spite of its inefficiency, this technique is currently the only choice in many commercial database systems (e.g, MySQL) currently. Next, we review some live migration techniques.

Albatross [12]**.** This approach allows the incoming transactions to run on the source node after the migration starts. In addition to migrating data, the source node iteratively ships to the destination node the updates made by the transactions. When the data have been migrated and there are relatively few (or stable) updates, the source node performs a short stop-and-copy migration and hands over the transactions to the destination. Although being reduced, the downtime still exists and may not be acceptable to systems with SLAs. Furthermore, to reduce the migration time, one may let the source node perform the stop-and-copy earlier, resulting in a longer downtime as there are more pending data/updates that need to be shipped.

Zephyr [15]**.** This approach avoids the service downtime by letting the incoming transactions run on the destination node. The source node transmits data chunks to the destination node iteratively. If a transaction on a destination accesses data that are not transmitted yet, it pulls the data from the source node. This increases the transaction latency and the transaction will block more conflicting transactions. Note that the blocking can result in a significant drop in throughput in a deterministic database system [37] because, to ensure the determinism, conflicting transactions *cannot* be dynamically reordered at run time (conversely, a traditional database system can achieve this easily by using, strict two-phase locking, for example). Furthermore, Zephyr does not allow a transaction on the source node that was active at the start of migration to change the metadata (specifically, the index used to track the migration progress) during the migration period. If so, the transaction needs to abort. The upside is that there is no additional update that need to be transferred. Note that, on the destination node the data migration process that persists the data chunks may conflict with both the read-only and read-write transactions. Therefore, setting a larger chunk size in order to reduce the migration time may result in further degradation in performance.

Squall [13]**.** This approach extends Zephyr by allowing an incoming transaction to run on the source node if the required data are

Table 1: Comparison of some data migration techniques.

	Stop&Copy	Albatross [12] (Source)	Zephyr [15] (Dest.)	Squall [13] (Either)	MgCrab (Both)
Downtime	Yes	Yes	No	No	No
Aborted Transaction	Yes	No	Yes	Yes	No
Throughput Drop	High (after)	Low	High	Moderate	Low
Increase in Latency	High (after)	Low	High	Moderate	Low
Extra Data Transferred	No	Yes	No	No	No
Migration Time Trade-Off	None	Downtime	R-&RW-Txs	R-&RW-Txs	None

entirely present there.[3] Squall knows the location of data required by a transaction by splitting the records to migrate into ranges and tracking the migration status of each range. A transaction can run on the source node only if all the relevant ranges are not being migrated or have not been migrated. Compared to Zephyr, Squall improves the system performance in the beginning of the migration process by letting some transactions run on the source node without being delayed by data pulls. However, the improvement is limited by the fact that *any* record that has started migrating in a range will prevent *all* transactions accessing data in the range from running on the source node. So, as soon as the data migration begins, the number of transactions that can complete on the source node decreases rapidly, and the system can still suffer from the performance degradation at the beginning of the migration period, as shown in Figure 2 (see Section 5 for the settings and detailed description). Furthermore, on H-store, the router may route a transaction to the source node where the required ranges are migrating. In this case, the transaction needs to abort and restart on the destination node.

Table 1 summarizes the performance characteristics of the above data migration techniques. As we can see, none of the current approaches is ideal for a deterministic database system. There is a need for a new design.

3. MGCRAB

In this section, we introduce MgCrab using some simplified assumptions for ease of presentation. We assume no failures, no distributed transaction in the system, and that there is only one migration plan to be fulfilled. The extended design that works with distributed transactions and concurrent migration plans is discussed in Section 4, while the failure handling is described in Appendix A.

3.1 Overview

The design goal of existing live migration techniques is to let every transaction that arrives during the migration period run on the most appropriate machine such that the clients perceive minimal service interruption. However, as evidence with Squall [13], deciding a better machine to execute a transaction requires precise information about the location of migrating data, and this information is hard to track in practice. MgCrab takes an opposite approach—instead of relying on data locations to decide the better one, it runs every incoming transaction on *both* the source and destination machines. MgCrab's live migration proceeds in three phases: the initial, termination, and crabbing phases, as shown in Figure 3. Transactions arriving at these three phases are processed on the source node, destination node, and both respectively.

In the crabbing phase, MgCrab hides the migration cost from clients by letting each client see the results from the *winner* node (i.e., the node that completes the transaction issued by the client faster). The source and destination nodes see strongly consistent data, and thus always process a transaction in the same way. MgCrab leverages the *determinism* discussed in Section 2.1 to avoid the high cost of strong consistency.

MgCrab works alongside any data partitioning scheme (e.g., hash-based, fine-grained record-based [9], or fine-grained range-based [29] partitioning) used by the storage and any elastic load balancing algorithm [9, 29, 36] implemented in the system controller for the scheme (cf. Figure 1).

3.2 Initial Phase

Given a migration plan (specifying the data to migrate, a source node, and a destination node), this phase engages all partitions in a replica such that every node enters the crabbing phase synchronously and has enough information to execute transactions correctly. At first, the migration controller on the source node creates an `Init` transaction request that includes the migration plan as a parameter and sends the request to the sequencers in the system so that after being forwarded to the scheduler on every node, the transaction gets into the same place in the total order and can be processed atomically throughout the system. Each scheduler forwards the request to the executor on the same machine. The executor then modifies the metadata about the ownership of migrating data (by changing "source" to "both"). After completing this transaction, all partitions enter the crabbing phase and process later transactions in the total order using the new ownership information.

Note that *all* partitions have to process the `Init` transaction because any node other than the source or destination may encounter a distributed transaction accessing the migrating data later (to be discussed in Section 4). However, there is no need for the transaction to process/ship complex metadata (e.g, index wireframe [15]) in order to help track the locations of migrating data, because the locations can be inferred deterministically on individual machines (to be discussed in the next phase). As a result, the transaction is generally very short and has a negligible impact on performance.

3.3 Crabbing Phase

Once entering the crabbing phase, the scheduler on the destination node starts forwarding transaction requests to the local executor if the required data overlap with the data to be migrated (regardless of the actual locations of these data). So, both the source and destination nodes process these transactions now. MgCrab employs two techniques, namely the *foreground pushes* and *two-phase background pushes*, to migrate data when serving the transactions:

Foreground Pushes. When processing a transaction, the executor on the source node may push data to the destination node in order to 1) migrate the active data as soon as possible, and 2) help the same transaction complete on the destination executor. For example, suppose the source partition owns data $\{A, B, C\}$ and wants to migrate data $\{B, C\}$ to the destination partition, as shown in Fig-

[3] Squall focuses on the *live reconfiguration*, where multiple migration plans are executed concurrently, and consists of some other techniques orthogonal to this paper.

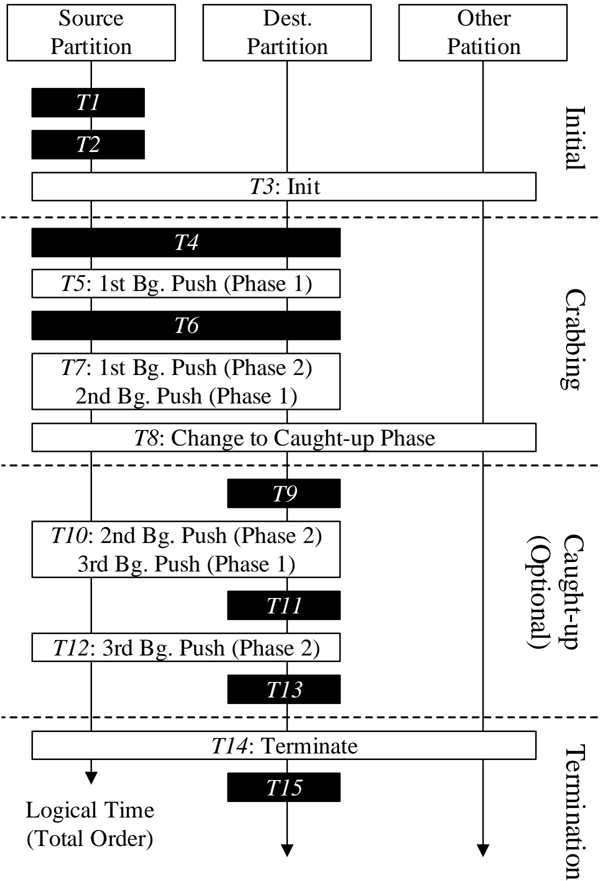


Figure 3: In MgCrab, a live migration task proceeds in the initial, crabbing, caught-up, and termination phases. User- and system-generated transactions are colored black and white respectively. The caught-up phase is described in Section 4.4.

ure 4. In the crabbing phase, a transaction T reading data $\{A, B\}$ and writing $\{C\}$ will be processed by both partitions (Figure 4(1)). In a deterministic database system, this transaction will be placed in the same position in the total order therefore when executing T, both the source and destination see a consistent snapshot of data. By virtue of this, the source executor reads its own data specified in T's read-set (Figure 4(2)) and pushes them to the destination executor (Figure 4(3)). Once obtaining $\{A, B\}$, the destination executor can execute and complete the transaction *locally*—there is no need for an agreement protocol (e.g., 2PC) since the deterministic system already ensured that these two partitions will always reach the same conclusion on whether to commit the transaction or not and, if yes, write the same value of C [37, 38]. To complete T, the source and destination executors write C independently (Figure 4(4)). Furthermore, the destination executor writes B and the both nodes mark on metadata that the data $\{B, C\}$ have been migrated.

After processing T, the destination executor needs no more pushes of B and C and can keep their states in sync with the source partition *at any logical time* by executing the later transactions deterministically following the total order. Note that the source executor does *not* delete B and C until in the termination phase (to be discussed later). This allows the source node to continuously serve later transactions during the migration period and hide the migration cost when the destination node is busy catching up with the

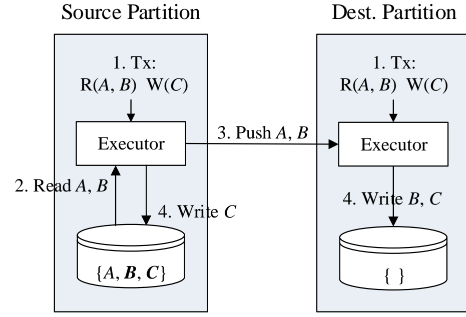


Figure 4: Execution of a transaction during the crabbing phase. Missing data are synchronously pushed to the destination partition.

source.

We call the above pushes generated during the transaction processing the *foreground pushes*. These pushes migrate hot/active data that are likely to be accessed again (due to the locality of access pattern) as early as possible. Nevertheless, the cold data may be left behind.

Two-Phase Background Pushes. One naive way to migrate the cold data is to break them into fixed-sized *chunks* and push the chunks one-by-one iteratively in the background. To preserve the determinism (and to ensure the correctness), one should push a data chunk *synchronously*—the migration controller on the source node creates a `BgPush` transaction request and sends it to the sequencers in the system so the transaction will be processed by both the source and destination partitions at the same logical time (following the total order). On the source node, the transaction reads and sends out the chunk; conversely, on the destination node the transaction waits for the chunk and then writes it into the storage. Before completing the transaction, both the source and destination nodes mark on metadata that the chunk has been migrated. By pushing a chunk synchronously, we can ensure that the locations of migrating data are always certain (after every transaction completes). So, there is no need to deal with the uncertainty of data locations, which may result in transaction aborts [13, 15].

However, the downside is that a `BgPush` transaction blocks all conflicting transactions on both the source and destination nodes. Even worse, the `BgPush` transaction is usually longer than normal user transactions due to the relatively large chunk size. This leads to the *clogging* [33, 37] where the blocked transactions continue to block incoming conflicting transactions like a chain reaction and finally results in a drastic drop of system performance, as shown in Figure 13(a). The same issue occurs in Squall [8] as well..

To solve this problem, we split a `BgPush` transaction into two phases, each accomplished by a distinct transaction, such that the source node can keep hiding the migration cost from users. The first transaction reads the chunk on the source node and sends it to the destination node, and then creates and schedules the second transaction into the system before it completes. The second transaction, after being forwarded by sequencers following the total order, inserts the buffered chunk to the storage of the destination node and then updates the metadata about the chunk location on both nodes. Between the first and the second phases, the sequencer may assign other transactions in the total order, as shown in Figure 3. The point of such a split is that both the first ($T5$) and second ($T7$) transactions can be made very lightweight on the source node to prevent the clogging of system performance. The second transaction is clearly lightweight on the source node as it modifies only the

metadata, so we focus on the first one. Notice that the first transaction ($T5$) is read-only on the source node. We let the node read the chunk *without* acquiring read locks.[4] Because of this, the first transaction does not block the following transactions ($T6$ and so on) and the source node can maintain the system performance. On the other hand, this may lead to a dirty chunk (to be stored into the storage of the destination node) in the second phase ($T7$), as the transactions in between ($T6$) may touch data in the chunk. Thanks to the crabbing mechanism, the destination node knows which data have been modified since the first phase ($T5$) by actively executing transactions, and can simply *skip those modified data* when storing the chunk in the second transaction ($T7$) to maintain consistency. The effectiveness of two-phase background pushes is demonstrated in Figure 13(b).

3.4 Termination Phase

After all data have been migrated, the migration controller on the destination node receives an empty background push and creates a `Terminate` transaction, as shown in Figure 3, that changes the ownership of the migrating data from "both" to "destination" on every partition. Like the `Init` transaction, this transaction is very short and has a negligible impact on the performance. After it completes, the source node stops executing transactions accessing the migrating data.

Recall that, during the crabbing phase, the migrating data are *not* deleted on the source node after they are pushed to the destination. When processing the `Terminate` transaction, the source needs to schedule a process that deletes the data. However, this process can run *in the background after* the `Terminate` transaction completes, resulting in little impact on performance. Alternatively, the source node can choose to keep the data if it foresees upcoming consolidation tasks. When data are moving back, the source node can locally replay the request logs (see Appendix A) to make the data up-to-date faster. We leave this alternative approach to the future work.

3.5 Correctness

MgCrab guarantees the *safety* and *liveness* [15].

THEOREM 3.1 (SAFETY). *MgCrab is* safe*, meaning that it meets the following conditions: (a)* transaction correctness*: transactions run with the correct ACID guarantees during migration; and (b)* migration consistency*: a failure during migration does not leave the data and system state inconsistent.*

THEOREM 3.2 (LIVENESS). *MgCrab is* live *as it satisfies the following requirements: (a)* termination*: if the source and destination nodes are not faulty and can communicate with each other for a sufficiently long period during migration, the migration will terminate; and (b)* starvation freedom*: every transaction that accesses the migrating data can eventually be processed, despite failures that may occur.*

Due to space limitations, the formal proof is left to Appendix A. Note that MgCrab does *not* require the REDO logs and a complex ARIES-like recovery algorithm [25] to bring a failed machine back to the latest state. Thus, it works nicely with exisitng deterministic systems [18, 38, 39] that replay the *request logs* during the recovery. Furthermore, MgCrab supports fast checkpointing such as the CALC [32] and Zig-Zag [5] algorithms.

[4]For readers familiar with Calvin: this read-only transaction ($T5$) is *not* the same as the low-isolation snapshot-read transactions [38], as the former needs to be totally ordered.

3.6 Migration Cost Analysis

The cost of MgCrab is summarized in Table 1.

Downtime. There is no downtime because the clients see the results from the winner nodes. The handover of winner nodes is seamless on a per transaction basis.

Aborted Transaction. Since the source can continue executing transactions and the data are always available during the migration period, there is no transaction that needs to abort due to the live migration.

Throughput Drop and Increase in Latency. There is no expensive agreement protocol (e.g., 2PC) in MgCrab for ensuring data consistency between the source and destination nodes. Furthermore, a foreground push can be performed very fast by a transaction on the source node as the source does *not* wait for the same transaction on the destination node to receive and process the data. If the destination node lags behind in physical time, the migration controller on the destination receives and puts the pushed data into the buffer and let the transaction read them locally later. Also, the transaction latency perceived by a client is at least as short as that on the source node. Therefore, MgCrab has little impact on the transaction latency. The same argument applies to the system throughput.

Extra Data Transferred. Since the data locations are certain to every transaction, there is no "false" push that transmits additional data not required by the destination. Moreover, unlike most existing source-based approaches [3, 12, 14, 24], the updates to the migrating data are *not* transmitted to the destination. In MgCrab, the destination node computes these updates by executing the incoming transactions deterministically. In effect, MgCrab trades computing for communication overhead. This is worthwhile for most OLTP applications as the transactions are generally short and the communication overhead (in distributed transactions) is a major reason for the slowdown of the system performance [13, 39].

Migration Time Trade-Off. To shorten the migration time, one can set a larger chunk size for the background pushes. Calvin [38] uses the conservative two-phase locking in concurrency control. Therefore, a `BgPush` transaction (in phase two) on the destination node blocks all conflicting read-only and read-write transactions. A larger chunk size can impact the performance of the destination node. However, the overall system performance depends on the winner node. A `BgPush` transaction (in either phase one or two) on the source node does not block other transactions, thus the source node is likely to win and renders the system performance less affected by large chunks. Furthermore, there are optimizations that make the system even less sensitive to the large chunks. We will discuss this in Section 4.

4. PRACTICES

In this section, we discuss how to apply MgCrab to real world situations and present some optimizations.

4.1 Pipelining of 2-Phase Background Pushes

A naive implementation of two-phase background pushes may suffer from a drawback: since each background push spans two totally ordered transactions, the overall migration time may be prolonged. To avoid this issue, we pipeline the two-phase background pushes. That is, we merge the second phase of the previous background push and the first phase of the next background push into one transaction, as shown by $T7$ in Figure 3. This transaction is still lightweight on the source node. Furthermore, the number of (totally ordered) transactions for two-phase background pushes now becomes similar to that of one-phase pushes. Figure 13(c) demonstrates the effectiveness of the pipelining.

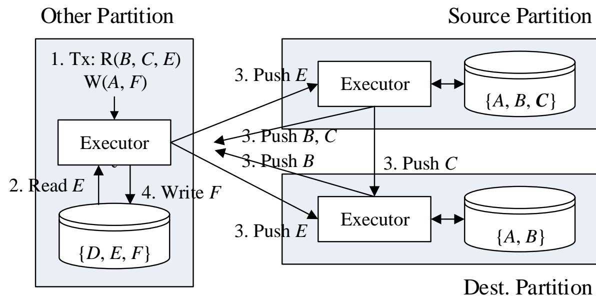


Figure 5: Execution of a distributed transaction during the crabbing phase. The other partition sees the pushes from the winner partition.

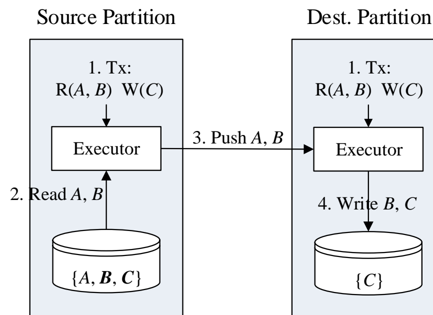


Figure 6: Execution of a transaction during the caught-up phase. Source partition does not access the migrated data and only pushes data not available on the destination.

4.2 Supporting Distributed Transactions

MgCrab can readily integrate with the default transaction processing in Calvin [38]. Normally (without migration), if two nodes have data to be accessed by a distributed transaction, their executors first read the data in the transaction's read-set respectively from their own storage, then push to each other the missing data on the opposite node. Once getting all the data in the read-set, both executors run the transaction following the transaction logic and then write the results belonging *only* to their respective data partition into the storage. There is no need for synchronization (e.g, distributed 2PL or 2PC) as the transaction is processed deterministically and the two executors always write consistent data.

When data are being migrated from a source to a destination node (in the crabbing phase in MgCrab) and the source gets involved in a distributed transaction, each other node involved in the same transaction can process the transaction by simply *regarding the source and destination nodes as a single node*. For example, suppose a distributed transaction T reading data $\{B, C, E\}$ and writing $\{A, F\}$ is being executed by the source node in an environment where the source node holds data $\{A, B, C\}$ with C being migrated, the destination node holds data $\{A, B\}$, and the other node holds data $\{D, E, F\}$, as shown in Figure 5(1). The other node observes that the record C is migrating by checking the ownership metadata (which was updated by an `Init` transaction, as discussed in Section 3.2). Then it reads the record E (Figure 5(2)) and pushes the record to *both* the source and destination nodes (Figure 5(3)) by treating them as a single node. Similarly, the source and destination nodes read $\{B, C\}$ and push the data to the other node (Figure 5(3)). If the data are not available yet on the destination node, the destination simply omits the data from its push. After receiving $\{B, C\}$, the other node can proceed to execute T and write F independently (Figure 5(4)). Similarly, the source and destination nodes write A and $\{A, C\}$ respectively and there is no need for a synchronization protocol to maintain the consistency of data written by the three nodes. Note that the other node receives two pushes (Figure 5(3)). When there is no missing data in the destination node's push (in the later migration period), the other node can benefit from the *winner node* (as clients do) to process T more quickly.

4.3 Supporting Range Queries

Like Squall [13], MgCrab supports the fine-grained range partitions [36] in the storage as well as transactions with range queries. In addition, it can be extended to support range queries more efficiently. Suppose there is a query that reads a large range of data. In the crabbing phase, the executor on the source node will push the entire data in the range to the destination node when processing the enclosing transaction. This foreground push, with a large data volume, may prolong the transaction on the source node and result in performance degradation (especially in the beginning of the crabbing phase). One can minimize this problem by letting the source and destination nodes switch to a *master-slave mode* for this particular type of transactions. Unlike in the normal case (which we called the *multi-master mode*) where the destination node executes the transaction actively and writes its own results, the destination node does *not* execute the transaction in the master-slave mode. Instead, it waits for the results pushed from the source and then writes the received data it owns into the storage. Although this results in additional data being transmitted, the advantage is that it avoids pushing large data in the foreground. The consistency is not sacrificed, and the destination node can write the same data as if it was in the multi-master mode.

Note that the switching needs to be deterministic on a per transaction basis. To ensure this, we let the source and destination nodes deterministically enter the master-slave mode if the read-set of a transaction is larger than a data chunk pushed in the background. Also note that the data locations are still always certain—even if a range is partially migrated, the locations of the corresponding data chunks are tracked in MgCrab. Different from Squall where the source node needs to abort a transaction if the transaction accesses a range that is partially migrated, the source node can always process the transaction in MgCrab due to the "crabbing."

Chunk Reordering. The master-slave mode introduced in Section 4.3 may slow down the migration of active data because only the data written by a transaction and owned by the destination are migrated. To speed up transmitting the active data, one can let the background-push process *reorder* the data chunks. When the source executor enters the master-slave mode for a transaction, it flags the data chunks that overlap with the transaction's read-set such that the background-push process will push these chunks first (specifically, in a higher priority following a deterministic rule).

4.4 Slow or Fast Destination

In MgCrab, the winner nodes of transactions that arrive during the crabbing phase may change over time. In the very beginning of the crabbing phase, the source node is likely to be the winner for each transaction, as the destination node may not have sufficient data and need to wait for the foreground pushes. However, as the destination node accumulates more and more data, the winner will be decided by the running speed of individual machines (taking into account the hardware speed, workload, etc.). So far, we assume that the destination node runs at a similar speed with the source. Next

we discuss how to extend MgCrab to handle destinations that are either too slow or fast.

Slow Destination. This may happen, for example, in consolidation scenarios where data are migrated to a machine with an existing workload in order to shutdown the source. If its aggregated workload is too high, the destination node may never catch up during the entire crabbing phase, resulting in the waste of computing resources. To help the destination node catch up, MgCrab introduces an optional phase, called the *catching-up phase*, between the initial and crabbing phases. In this phase, the source and destination nodes execute *every* transaction using the master-slave mode introduced in Section 4.3 so that the destination node can use its full power to catch up with the source before entering the crabbing phase. Another way to prevent a slow destination is to let the system controller (see Figure 1) generate better migration plans based on the statistics and SLA [9, 29, 36].

Fast Destination. This is common in scale-out scenarios where the source node is overloaded and/or the destination node is a new machine. When the destination node runs faster in the later crabbing phase, executing transactions on the source offers little help to performance but rather wastes the computing resources. To conserve computing on the source node, MgCrab introduces another optional phase, called the *caught-up phase*, between the crabbing and termination phases. In this phase, the source node *stops* updating data that have been migrated and executes a transaction only when the required data are *not* available on the destination. Following the example shown in Figure 4, suppose now the source partition has already migrated $\{C\}$ to the destination via previous transactions, as shown in Figure 6. Because it no longer updates C, if there is a transaction T that reads $\{A, B\}$ and writes C (Figure 6(1)), the source node only reads and pushes $\{A, B\}$ to the destination (Figure 6(2)(3)). After that, it marks on the metadata that B has been migrated and does not access B anymore. This not only makes T run faster on the source node, but also allows the source node to *skip all later transactions not accessing* A, thereby more quickly alleviating the overloaded source. Note that the data $\{B, C\}$ on the source node have no impact on the consistency (as they are no longer read) and will be deleted as usual after the termination phase.

The source and destination nodes need to enter the caught-up phase at the same logical time to preserve the determinism. MgCrab accomplishes this by issuing a special transaction request, which is totally ordered with other user transactions. Figure 3 shows the switching from the crabbing phase to the caught-up phase. After the switching, all later user transactions that touch the migrating records will run on the destination node. In practice, this switching can happen as early as 1) the amount of data pushed in the foreground converges, and 2) there is no pending tagged data chunk in the background (see Section 4.3).

4.5 Concurrent Migration Plans

Once the system controller broadcasts a new partition plan (see Figure 1), the machines may infer more than one migration plans. These migration plans can be executed concurrently to speed up the reconfiguration of data partitioning. Generally, there is no problem in running multiple instances of MgCrab on multiple nodes. However, if a node involves in too many migration tasks at the same time, it may have to execute a large portion of transactions in the system (because of the "crabbing") and become a hotspot. One way to avoid this problem is to 1) group the migration tasks such that every node involves in only few migration tasks in each group [13]; and 2) run the tasks in different groups serially.

4.6 Supporting Disk Storage

As discussed in Section 3.3, a background push in MgCrab is very lightweight on the source node thus can minimize the impact on system performance. This offers a significant advantage in a system equipped with the disk-based storage where reading/writing a data chunk from/to the disk-based storage is very slow. MgCrab can be further optimized to *prefetch* [38] the data chunks on the source node to reduce the processing time of the `BgPush` transactions in the first phase. In addition, the destination node can write the data chunk into the storage *asynchronously* in the second phase in order to speed up and catch up with the source node faster. The asynchronous writes do not have an impact on the correctness in the presence of failures as the chunks can be recreated by replaying the request logs during the recovery process (see Appendix A).

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of MgCrab. We implement MgCrab on top of an open-source database system. We also implement Calvin [38] for distributed deterministic transaction processing. Details about the system and our implementation can be found in Appendix B. The chunk size of the background pushes is set to 15000 records by default. The experiments are conducted on a cluster where each machine is equipped with an Intel Core i5-4460 3.2 GHz CPU and 32 GB RAM and running 64-bit CentOS 7. The nodes in the cluster are interconnected by a 1Gbps switch. For comparison, we also implement the following two live migration approaches.

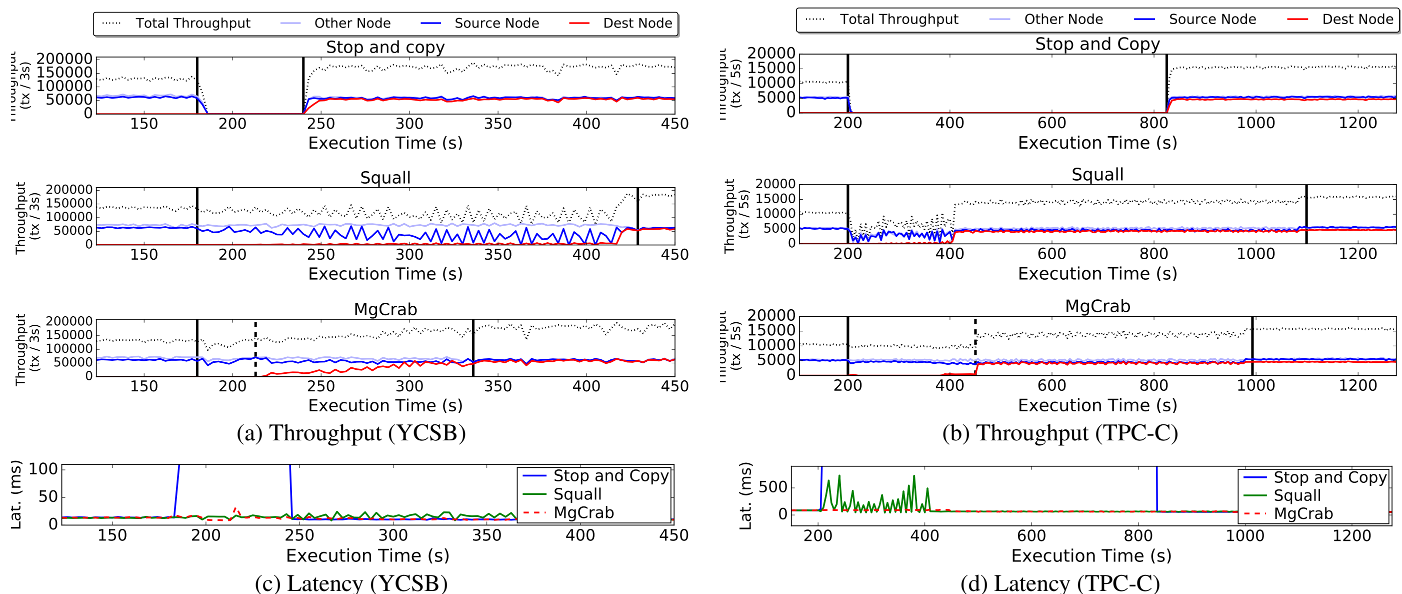
Stop-and-Copy: when a migration starts, the migration controller on the source node issues a distributed transaction that locks both the source and destination nodes and then performs data migration. All conflicting transactions are blocked until this transaction completes.

Squall [13]**:** a migration begins with a distributed transaction that updates the metadata on the source and destination nodes. Then, each transaction request accessing the migrating data is routed (via the scheduler) to the destination node by default. However, if the data are all available on the source node, the transaction will be routed to the source node. Squall has been shown to outperform other live migration techniques, like Zephyr [15], and is designed specifically for deterministic database systems. One key difference between the original Squall on H-Store [18] and our implementation is that the transactions on H-Store are processed sequentially by a single thread; while transactions on the Calvin/our system are processed concurrently by multiple threads. Also, Squall proposes some techniques for *live reconfiguration* targeting multiple concurrent migration plans. Since our focus is on improving the execution of a single migration plan, we do not implement these live reconfiguration techniques.

5.1 Scenarios & Workloads

Our experiments measure how well the above approaches are able to fulfill a single source, single destination migration plan in different scenarios, including *scaling out* and *cluster consolidation*. We use a cluster with 3 nodes. In the scaling out scenario, the source node, destination node, and the other node hold two, zero and one data partitions initially and the goal is to migrate one partition of data from the source to the empty destination. In the consolidation scenario, each node holds one partition initially and the goal is to migrate the partition held by the source node to the destination.

We employ two OLTP workloads, the YCSB and TPC-C, in our evaluation. Transaction requests are submitted from 600 and 180 client threads running on 3 nodes in YCSB and TPC-C workloads



(a) Throughput (YCSB) (b) Throughput (TPC-C)

(c) Latency (YCSB) (d) Latency (TPC-C)

Figure 7: System performance during scaling out. The first and second vertical solid lines denote the start and end of migration respectively. In MgCrab, the vertical dash line denotes the beginning of the caught-up phase.

respectively. Each client submits transaction requests to a partition node in a closed loop, i.e., it blocks after submitting a request until the result is returned. In MgCrab, the client can issue the next transaction request once getting the result from the winner node (either the source or destination node) during the migration. In each experiment trial, we warm up the DBMS for 150 seconds before collecting the measurements. Latencies are measured as the duration from the time a client submits a transaction request to when it receives the result.

YCSB [8]: the Yahoo! Cloud Serving Benchmark (YCSB) is a collection of workloads that simulate a large-scale online service. We use a YCSB database consisting of a single table with 1.5 million records (half a million records in each partition). Each tuple has 10 columns including the primary key. Each column of a tuple contains a randomly generated string of 100 bytes. The workload consists of 85% read-only transactions and 15% read-write transactions. Each read-only transaction reads two records in the same partition. And each read-write transaction reads and updates a single record and then inserts another new record to the same partition. The keys of the records are selected from the Zipfian distribution specified by YCSB.

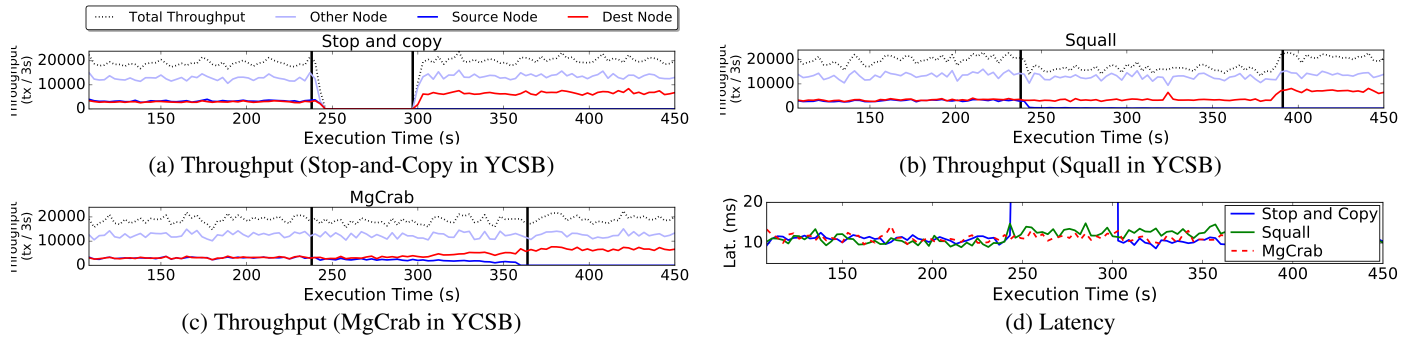
TPC-C [31]: the TPC-C benchmark simulates a warehouse management system. It consists of nine tables and five types of transactions. Here we focus on the New-Order transactions and Payment transactions, which are both read-write transactions and together contribute 88% of the standard TPC-C workload. We create 20 cold warehouses and one hot warehouse, where the hot warehouse receives transaction requests from 10 times more clients than a cold warehouse does. In the scaling out scenario, the migrating partition consists of the hot warehouse, and the remaining partitions contains 10 cold warehouses each. As compared to the YCSB workload, TPC-C has much longer transactions, and there are roughly 10% of the transactions that are distributed across multiple partitions.

5.2 Scaling Out

We first evaluate how well the live migration approaches perform in the scaling out scenario, where the source node is overloaded and the goal is to transfer half of its workload to the destination node via data migration.

YCSB. Initially, the source, destination, and the other nodes have 1, 0, and 0.5 million records respectively. We migrate 0.5 million records (one partition) from the source to destination nodes. Each partition has 200 client threads submitting requests to it from client nodes. The changes of system performance during the migration are shown in Figures 7(a)(c). It is obvious that Stop-and-Copy has the worst performance (zero throughput and extremely high latency) during the whole migration period. But it gives a lower bound of the migration time because the data transfer can be done much more efficiently by a dedicated process than live migration approaches. Squall is alive but gives dropped performance during the migration. Squall uses reactive polls to migrate hot data in the foreground and one-phase background/asynchronous pushes to migrate cold data in the background. We observe that the performance drop is mainly due to the one-phase background pushes rather than the reactive polls because YCSB transactions are generally short and the delays of reactive polls do not clog up short transactions easily. However, the one-phase background push transactions are much longer and block many other transactions. On the other hand, the two-phase background pushes in MgCrab are lightweight on the source node and do not conflict with foreground transactions there. This allows the source node to maintain the performance during the migration period. Moreover, the fewer blocked foreground transactions speed up the migration of hot data. Therefore, the total migration time is reduced. MgCrab also gives a latency comparable to Squall's.

TPC-C. The source, destination, and the other nodes contain 11, 0, and 10 warehouses respectively. One warehouse on the source node is hot and will be migrated to the destination node. Each cold warehouse has 6 client threads submitting requests to it, and the hot warehouse has 60 client threads acting with it. Figures 7(b)(d) show how the performance changes during the migration period. It is not surprising that the Stop-and-Copy gives similar results as in the last experiment. However, Squall renders a significant performance drop after the migration starts, and the major reason differs



(a) Throughput (Stop-and-Copy in YCSB)

(b) Throughput (Squall in YCSB)

(c) Throughput (MgCrab in YCSB)

(d) Latency

Figure 8: System performance during cluster consolidation.

from the one in YCSB. Generally, TPC-C transactions are much longer than those in YCSB and have about a 10% chance to interact with remote partitions. In Squall, a data object is owned by a single machine. The reactive migration, which pulls data from the source node, prolongs distributed transactions by involving the destination node in these transactions. This slows down not only the source and destination nodes *but also the other nodes* (see Figure 7(b)), thereby resulting in the drop of entire system throughput and increased latency. In contrast, the distributed transactions in MgCrab are not prolonged by the foreground pushes, because it is the *winner node* that responds to the clients and interacts with the other node in the distributed transactions (see Section 4.2). So, the latency and system throughput become much more stable.

In addition, the asynchronous pushes in Squall result in oscillation in throughput at the later part of the migration period. This is because each asynchronous push is handled by a (long) distributed transaction that can easily block other transactions. On the other hand, the two-phase background pushes in MgCrab does not block normal transactions and has little impact on the system throughput. We can see that after the background pushes starts (the first vertical line in MgCrab in Figure 7(b)), the throughput remains stable since both the phase 1 and phase 2 background push transactions can be processed very efficiently on the source node. However, once MgCrab switches to the caught-up phase (the second dash line in MgCrab in Figure 7(b)), the background pushes start to impact the system throughput because the destination node takes over the transaction processing of the migrating data. Note that we choose to start the caught-up phase at the time when the data pushed in the foreground converges.

In Figure 7(d), we can see that with Squall the latency increases during the migration period. This is because of the destination node busying inserting data chunks pushed in the background. On the other hand, the latency of MgCrab remains stable. As stated in Section 3.1, MgCrab lets both the source and destination nodes respond to clients, so the latency observed by a client is determined by the winner node, which is the source node in this case. MgCrab let the source node hide the cost of migrations on the destination node.

5.3 Consolidation

Next, we evaluate the performance of migration approaches during cluster consolidation using the YCSB workload. We make all nodes underloaded so that we could shut down a node in order to save resources. The source, destination, and the other nodes have one partition initially and our goal is to move the partition on the source node to the destination node (and to shutdown the source node later). The other node has 50 client threads submitting transaction request to it, while the source and the destination node only has 10 client threads each. The results are shown in Figure 8. We can see that the performance is unchanged before and after the migration in all figures because the system is not fully loaded. However, the consolidation allows the system to use fewer machines.

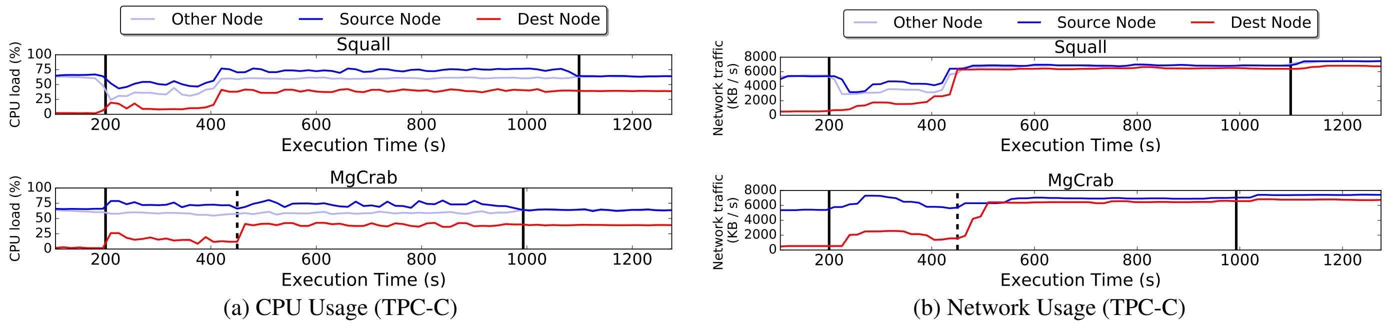
Just as in the scaling out scenario, the Stop-and-Copy blocks all transactions and makes the system temporarily available. In Squall, the performance drops again when the reactive and asynchronous migration begins. The reason is similar to the one in the scaling out scenario—the background pushes block many transactions on the source node. The problem is more severe here because the destination node is underloaded and pulls fewer data from the source node, leaving more data cold and be migrated through background pushes, which in turn blocks more transactions and results in a severe drop in throughput on the source node. On the other hand, the two-phase background pushes in MgCrab prevent foreground transactions from being blocked on the source node and allows the source node to smoothly hand over the workload to the destination node.

5.4 Resource Utilization

Executing transactions on both the source and destination nodes seems to imply doubled resource utilization (and saturated system). However, this is *not* true. Instead, MgCrab improves resource utilization. We compare the CPU and network utilization of Squall and MgCrab, and the results are shown in Figure 9.

As we can see, Squall and MgCrab consume resources very differently on the source and the other node during the migration. In Squall, we see a clear drop in resource utilization after the reactive and asynchronous migration starts. There are two major reasons for this. First, the reactive migration lets the destination node pull migrating data from the source, but since only one machine can own the data, this prolongs the distributed transactions that originally lie between the source and the other node. The network latency in these distributed transactions counts into the contention footprint, thereby slowing down the entire system, as evidenced by the drop of resource utilization on the other node. Second, the reactive pull and asynchronous push transactions may conflict with the user transactions. This seriously impacts the performance of the source node on serving clients and results in the drop of resource utilization on the source node.

MgCrab avoids both of the above problems. By allowing the source and destination machines to execute transactions concurrently, the distributed transactions between the source and the other node are not prolonged by the destination. This maintains the performance of the other node, as evidenced by its stable resource utilization between the first solid vertical line and the dash line. Furthermore, neither the phase 1 nor phase 2 background push transactions conflict with the normal transactions. Thus, after the back-



(a) CPU Usage (TPC-C)

(b) Network Usage (TPC-C)

Figure 9: The CPU and network usage during the migration in the scaling out scenario with TPC-C workload. See Figure 7 for the explanations of vertical lines.

ground pushes start, the source performance remains stable, as evidenced by the resource utilization as well.

There is something else noteworthy about MgCrab. We see slightly higher loads (in both CPU and network) on the source node during the migration period. This is the amount of effort that the source node spends on identifying and pushing data to the destination node. We can also see this in the later migration period in Squall (for asynchronous pushes).

5.5 Concurrent Migration Plans

In this section, we conduct two larger scale experiments to validate the efficiency of MgCrab. In the first experiment, we run 3 concurrent migration plans on a 12-node cluster, where different migrations involve different machines; while in the second experiment, we run 6 concurrent migration plans on an 18-node cluster, where each pair of destination nodes share the same source node. We use the TPC-C workloads with the scaling-out scenario in the following experiments.

Figure 10 shows the system throughput during the migration period. As we can see, the impact of migrations to system performance becomes larger when there are more nodes involved. And, in a complex situation (Figure 10(b)), both Squall and MgCrab incur a performance drop in the beginning of the migration period. This is mainly due to the overhead of initializing migration states. However, MgCrab still renders better performance than the baselines.

6. FURTHER RELATED WORK

In this section, we briefly discuss some related work not mentioned in Section 2.3.

The study [12] proposes the *flush-and-copy* technique as an alternative to the stop-and-copy. During the migration, it first flushes dirty records and then marks the source node as read-only. Any read-write transaction on the source node is aborted and restarted on the destination node. Although saving the system from going down entirely, the read-only mode may not be acceptable to OLTP applications.

Elmore et al. [14] propose the *synchronous migration* that relies on replication techniques to migrate data. Like MgCrab, this technique replicates (instead of moving) data during migration. However, it uses the log shipping and an eager replication protocol to sync the progress between the source and destination. These syncing methods do *not* take advantages of a deterministic database system [13] and introduce not only extra transmitted data but also high communication overheads. Furthermore, the study employs a failover mechanism to hand over all transactions at once, whereas in MgCrab the handover is on a per transaction basis (via the winner nodes).

In addition to Albatross [12], some studies aim to improve the efficiency of the source-based approaches. Slack [3] minimizes the impact of migration by throttling the rate that the data chunks (specifically, pages) are migrated from the source to destination. The throttling also takes into account the impact of migration on other tenants in a multi-tenant database system. Slacker uses recovery mechanisms to stream updates. Madeus [24] employs a middleware that analyzes the operations performed on the source node and propagates only the necessary operations to the destination to synchronize the two nodes. Operations are propagated concurrently to improve the communication efficiency. This study is optimized for the snapshot isolation.

The destination-based approach ProRea [34] extends Zaphyr [15] by proactively migrating hot records to the destination at the start of migration; it also focus on snapshot isolation. Another work, Rocksteady [20], is also a destination-based approach, which transfers the ownership of migrating records at the beginning of the migration and processing incoming client requests at the destination node. This approach reduces the time of the migration by issuing multiple pull requests and taking the advantages of the thread model of RAMCloud [27]. However, unlike MgCrab, the work focuses on key-value stores, and it does not consider how to ensure consistency and isolation while processing transactions.

In addition to Squall [13], studies [1, 23] propose some optimization techniques to improve the efficiency of *live reconfiguration* where multiple migration tasks run concurrently in the system. Wildebeest [1] applies both the reactive and asynchronous data migration techniques to a distributed MySQL cluster. Minhas et al. [23] propose a method for VoltDB that uses predefined virtual partitions as the granule of migration. Although being orthogonal to this paper, some techniques may be helpful in applying MgCrab to a live reconfiguration task on these particular platforms.

PLP [28] proposes an intra-machine data migration technique that moves data between the partitioned data structures used by different cores to improve concurrency. Elastic load balancing has also been discussed in the virtual machine-based (VM-based) approaches [11, 35], where each tenant is contained in a dedicated VM. These approaches usually focus on the "scale-up" scenarios and make use of the VM migration techniques [7, 22] for resource planning/allocation. The movement of data and the constraints faced by the above techniques are very different from the inter-machine data migration techniques we target in this paper.

Beyond transactional and operational databases, several stream processing systems explore the use of live migration for elastic scaling. While the goals of these approaches are similar to MgCrab and

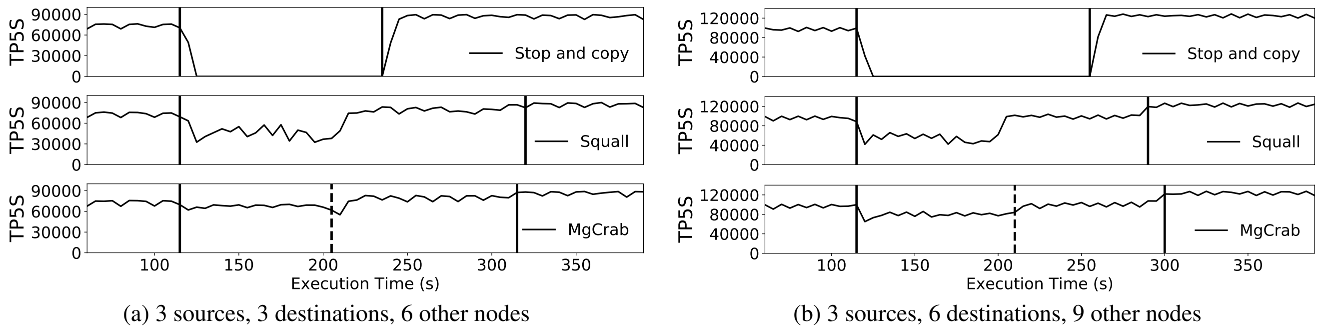


Figure 10: The impact of concurrent migration plans. See the description of Figure 7 for the meaning of vertical lines. TP5S stands for transactions per 5 seconds.

other transactional migration techniques, their assumptions and requirements differ. Many systems, such as [26], support simple migration through persistent queues and partitioning operators, but do not support live migration [40]. Fernandez et al. [6] describe how to dynamically scale-out stateful stream operators (e.g. join) by partitioning the operator's data and intelligently using check-points and replay from upstream operators for explicit state management of an operator. TimeStream [30] proposes "resilient substitution" that tracks the state of an operator and it's output dependencies, which allows a provenance style replay to selectively rebuild an operator, with checkpoints minimizing the amount of work to recreate state. These projects differ in that the amount of data or state migrated is bounded by having operators only work on a window of data. ChronoStream [40] utilizes a strategy similar to Albatross [12] for migration, where a destination operator is rebuilt from a local or remote copy, and after a switch-over events are replayed at the destination. Migration for these stream processing systems mainly differs by targeting the migration of only operators' data, which can be significantly smaller than migrating all of the persistent state associated with a database or partition [40]. Additionally, stream processing systems typically do not update data in-situ and instead generate output based on immutable input data (streams). However, transactional database migration must consider that multiple partitions may invoke an in place update in the presence of multi-key transactions and secondary record lookups.

7. CONCLUSIONS

We present MgCrab, a live migration technique, for deterministic database systems. Different from most exiting migration techniques, MgCrab lets *both* the source and destination machines execute every incoming transaction during the migration period. We show how such a design avoids the complexity of data ownership tracking as well as distributed transactions that slow down the entire system right after the migration starts. We also propose the two-phase background pushes that prevent the migration transactions from blocking normal transactions. In addition, we point out some extensions and optimizations that can further improve the applicability of MgCrab as well as the system performance. Extensive experiments are conducted on a real system and the results demonstrate the effectiveness of MgCrab in both the scaling out and consolidation scenarios. MgCrab guarantees the *safety* and *liveness*. We proof these guarantees formally in Appendix A.

Further Extensions and Future Work. This paper opens up some interesting research directions. First, MgCrab is a live migration technique for a single source, single destination migration plan. One may extend the system controller (e.g., E-store [36]) such that it best utilizes MgCrab when generating a multi-source, multi-destination migration plan (i.e., a live reconfiguration plan). Furthermore, although MgCrab is proposed for deterministic database systems, one may integrate it into a non-deterministic system by requiring the system to enter a "deterministic mode" temporally during the migration. This helps avoid the complexity and drawbacks of non-deterministic migration techniques (e.g., [12, 15]) discussed in Section 1. The cost is that, during migration, the system will not be able to accept ad-hoc queries because all transactions must be totally ordered first. However, recent studies [16, 18, 33, 38] show that this cost may be acceptable to many OLTP applications as the clients (e.g., web/application servers) are usually optimized for performance and issue queries through stored procedures. The above are matters of our future inquiry.

8. REFERENCES

[1] Wildebeest. `http://zendigital.co/wildebeest`.
[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.
[3] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy. Cut me some slack: Latency-aware live migration for databases. In *Proc. of EDBT*, pages 432–443. ACM, 2012.
[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.
[5] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. of SIGMOD*, pages 265–276. ACM, 2011.
[6] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proc. of SIGMOD*, pages 725–736. ACM, 2013.
[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of NSDI*, pages 273–286. USENIX Association, 2005.
[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of SoCC*, pages 143–154. ACM, 2010.
[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. of the VLDB Endowment*, 3(1-2):48–57, 2010.

[10] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. on Database Systems (TODS)*, 38(1):5, 2013.
[11] S. Das, F. Li, V. R. Narasayya, and A. Christian Konig. Automated demand-driven resource scaling in relational database-as-a-service. In *Proc. of SIGMOD*, pages 1923–1934. ACM, 2016.
[12] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. of the VLDB Endowment*, 4(8):494–505, 2011.
[13] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proc. of SIGMOD*, pages 299–313. ACM, 2015.
[14] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Towards an elastic and autonomic multitenant database. In *Proc. of NetDB Workshop*. sn, 2011.
[15] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proc. of SIGMOD*, pages 301–312. ACM, 2011.
[16] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proc. of the VLDB Endowment*, 10(5):553–564, 2017.
[17] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proc. of Int'l Conf. on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.
[18] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment*, 1(2):1496–1499, 2008.
[19] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. of VLDB*, pages 134–143. VLDB Endowment, 2000.
[20] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 390–405. ACM, 2017.
[21] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
[22] H. Liu, H. Jin, X. Liao, L. Hu, and C. Yu. Live migration of virtual machine based on full system trace and replay. In *Proc. of HPDC*, pages 101–110. ACM, 2009.
[23] 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. IEEE, 2012.
[24] T. Mishima and Y. Fujiwara. Madeus: database live migration middleware under heavy workloads for cloud environment. In *Proc. of SIGMOD*, pages 315–329. ACM, 2015.
[25] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. on Database Systems (TODS)*, 17(1):94–162, 1992.
[26] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proc. of the VLDB Endowment*, 10(12):1634–1645, 2017.
[27] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.
[28] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. *Proc. of the VLDB Endowment*, 4(10):610–621, 2011.
[29] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. of SIGMOD*, pages 61–72. ACM, 2012.
[30] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: Reliable stream computation in the cloud. In *Proc. of EuroSys*, pages 1–14. ACM, 2013.
[31] The Transaction Processing Council. Tpc-c benchmark (version 5.11), 2010.
[32] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proc. of SIGMOD*, pages 1539–1551. ACM, 2016.
[33] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. of the VLDB Endowment*, 7(10):821–832, 2014.
[34] O. Schiller, N. Cipriani, and B. Mitschang. Prorea: live database migration for multi-tenant rdbms with snapshot isolation. In *Proc. of EDBT*, pages 53–64. ACM, 2013.
[35] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. *ACM Trans. on Database Systems (TODS)*, 35(1):7, 2010.
[36] 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 systems. *Proc. of the VLDB Endowment*, 8(3):245–256, 2014.
[37] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. of the VLDB Endowment*, 3(1-2):70–80, 2010.
[38] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proc. of SIGMOD*, pages 1–12. ACM, 2012.
[39] S.-H. Wu, T.-Y. Feng, M.-K. Liao, S.-K. Pi, and Y.-S. Lin. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In *Proc. of SIGMOD*, pages 1553–1565. ACM, 2016.
[40] Y. Wu and K.-L. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *Proc. of ICDE*, pages 723–734. IEEE, 2015.

APPENDIX

A. CORRECTNESS & FAULT TOLERANCE

A live migration technique should ensure the two requirements of safety and liveness. [15].

Definition A.1 (Safety). *A data migration technique is* safe *if it meets the following conditions: (a)* transaction correctness*: transactions run with the correct ACID guarantees during migration; and (b)* migration consistency*: a failure during migration does not leave the data and system state inconsistent.*

Definition A.2 (Liveness). *A data migration technique is* live *if the following conditions are met: (a)* termination*: if the source and destination nodes are not faulty and can communicate with each other for a sufficiently long period during migration, the migration will terminate; and (b)* starvation freedom*: every transaction that accesses the migrating data can eventually be processed, despite failures that may occur.*

In this section, we describe how MgCrab meets these conditions.

A.1 Consistency and Isolation

In Calvin, each executor processes transactions using multiple threads and employs the conservative ordered locking [37, 38] to ensure the serializability of transaction execution following the total order decided by the sequencers. The conservative ordered locking is known to be deadlock-free and has no phantom problem [4, 19].

Theorem A.1. *In MgCrab, the conflicting transactions are executed serially following the total order during migration.*

Proof. Based on the study [4], this can be easily proved by the fact that 1) the read and write sets of every transaction (an `Init`, `Terminate`, `BgPush`, or a user transaction[5]) can be known prior to execution; and 2) when running, each transaction first acquires all the locks for the read/write sets and the lock requests of different transactions are granted atomically across machines following the total order. □

Furthermore, Calvin provides the following primitive:

Axiom A.1 (Run to Completion). *Every transaction must run until it commits or* aborts due to deterministic program logic.

Together with the conservative ordered locking, the above axiom implies:

Axiom A.2 (Determinism). *After each transaction, the data and system state is* deterministic *as if the transaction and all its precedences were executed serially following the total order.*

We show that MgCrab preserves determinism in the following.

Theorem A.2. *In MgCrab, the data and system state is deterministic during migration.*

Proof. We prove this theorem by contradiction. Axiom A.2 ensures that the first transaction in MgCrab begins with a consistent data and system state. If the transaction completes but leaves the system state inconsistent, then, with the serializability guarantee given by Theorem A.1, the reasons could only be 1) the transaction logic writes inconsistent data; and/or 2) the transaction is aborted by the system. Based on our discussions in Sections 3 and 4 of the main text, each transaction in MgCrab writes data consistently regardless of the phase it is in and the mode it uses. Thus the cause can only be the latter. However, because this contradicts with Axiom A.1, the transaction leaves the system state consistent. Applying the above argument to the second transaction and so on, we obtain the proof. □

Theorem A.2 implies that transaction processing in MgCrab guarantees consistency and serializable isolation.

[5] See [37, 38] on how to obtain the read and write sets of a user transaction.

A.2 Fault Tolerance

We first explain how MgCrab ensures the atomicity and durability in transaction processing. Our failure model assumes that all messages are transferred using reliable communication channels that guarantee in-order, at most once delivery. We take into consideration node crashes and network partitions, and that a node failure does not lead to loss of data (e.g., logs) in the persistent storage. However, we do not consider malicious node behavior.

One salient feature of a deterministic database system is that the fault tolerance mechanisms can be greatly simplified [18, 37, 38]. Calvin lets multiple machines store the same copy of a data partition in a data center. One machine is selected as the *primary* partition to serve transactions in normal cases while the others, called *secondary* partitions, actively replicate the data/updates by running the same transactions. If the primary partition fails, a secondary partition takes over immediately (and deterministically). To recover a failed machine, Calvin relies on the UNDO logs kept during transaction processing to rollback transactions (either complete or incomplete) until a checkpoint. Then it replays the *request logs* kept during scheduling to bring the machine back to the latest state. The replay of logged transaction requests follows the same total order, and therefore is idempotent (see Axiom A.2). Note that transactions do not keep the REDO logs, as there is no need for the ARIES-like recovery algorithm [25]. Furthermore, Calvin supports fast checkpointing such as the CALC [32] and Zig-Zag [5] algorithms.

MgCrab leverages the above mechanisms to ensure high availability and fault tolerance. MgCrab lets the secondaries participate in a migration task in the exactly same way as the primary does. Source machines process a transaction in the same manner but only the primary machine pushes data to the destination machines (either in the foreground or background). This guarantees a strong consistency between 1) the source and destination primaries; and 2) a primary and its secondaries. If the source or destination primary fails during the migration, a secondary can take over immediately and deterministically. Furthermore, if network failure happens during a migration, the source node and the destination node can still keep processing transactions that are not conflicting with the current migration transaction. The migration transaction and its conflicting user transactions will halt, but they will not hurt the consistency of data and system states since their total order has been determined.

In an extreme case where a primary and all its secondaries fail at the same time, the migration halts and MgCrab instructs the system to stop accepting transactions that access the migrating data. The availability level here is the same as when the machines holding migrating data all fail in non-migration cases. To recover the machine and migration state, MgCrab rollbacks the transactions on *both* the source and destination nodes (including secondaries) until the last checkpoint, then replays the request logs after the checkpoint following the total order to bring the system back to the latest state. Note that the replay is idempotent because each migration operation is performed in transactions (e.g., `Init`, `Terminate`, and `BgPush` transactions) and Theorem A.2 guarantees determinism. After the machine and migration state recovers, MgCrab instructs the system to accept incoming transactions and continues the migration progress. It can be easily seen that MgCrab ensures the atomicity and durability of transactions.

Safety and Liveness. Next, we prove the safety and liveness of MgCrab.

Theorem A.3. *MgCrab is safe.*

Proof. From the above discussions, we can see that MgCrab runs transactions with full ACID guarantees. Furthermore, MgCrab

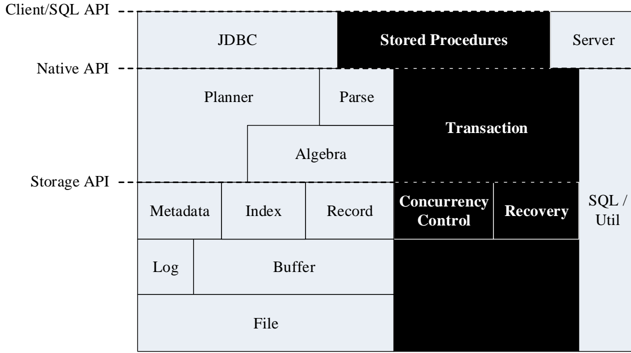


Figure 11: The system architecture of our core database system.

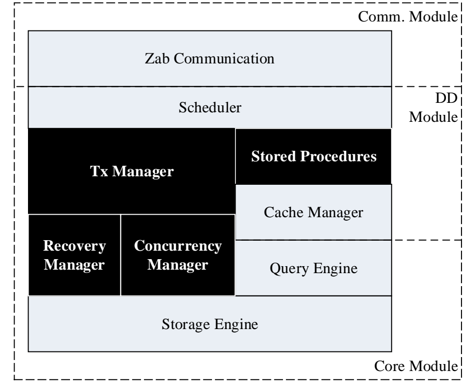


Figure 12: The system architecture of our distributed deterministic system.

performs all data migration operations in transactions (including `Init`, `Terminate`, `BgPush`, and user transactions), so we can prove the migration safety (Definition A.1(b)) by considering only the transaction correctness (Definition A.1(a)). Hence the proof. □

THEOREM A.4. *MgCrab guarantees liveness.*

PROOF. We first prove that the migration terminates if there is no failure (Definition A.2(a)). In MgCrab, the migration terminates after the `Terminate` transaction. This transaction must be created because the migrating data has a limited size so there must be an empty `BgPush` transaction that triggers the `Terminate` transaction (see Section 3.4 of the main text). Next, we show that transactions in MgCrab are free from starvation (Definition A.2(b)). After an arbitrary sequence of failures, the source and destination will have a consistent state because each replay of request logs is idempotent. Therefore, the migration can resume and every transaction arriving during the migration period will eventually be processed. We obtain the proof. □

B. SYSTEM ARCHITECTURE

We implement MgCrab on an open source database system and extend the system to be a distributed, deterministic database system following Calvin [38]. Each server node contains a core database system and a distributed system module. A core database system only deals with the data stored in the local storage. A distributed system module manages the actions that need to collaborate with other nodes. Both systems have been running for more than one year to serve our R&D prototypes, and are briefly introduced in the following.

B.1 Core Database Systems

Figure 11 shows the architecture of our core database system. The architecture is based on IBM System R, a classical DBMS developed in the 1970s. All the fundamental components of a single-node DBMS, such as file manager, buffer manager, record manager, concurrency control manager, query parser, query planner and query optimizer, are implemented in the system. It can be roughly divided into a query engine and a storage engine.

The query engine (between the Native API and Storage API) accepts SQL statements and translates them into executable plans represented by relational algebra. Each algebra has a (or several) predefined way to access the storage engine. The query planner is responsible for finding the "best" (i.e., of lowest cost) plan tree for each SQL statement. Note that we only slightly modify the code here without strictly following the standard (e.g., SQL-92), but it is enough to run most research benchmarks rigorously.

The storage engine (under the Storage API) takes the responsibility of storing and retrieving records from the file system. The system is a disk-based DBMS, and it puts all records on non-volatile storage and organizes the physical structure of files itself. To minimizing I/O times, we enlarge the buffer pool to cache file blocks in memory. The system employs multiple buffer replacing strategies, such as the Clock strategy, to reduce the chance of swapping. To ensure durability, the system logs all changes made by transactions and implements the Write-Ahead Logging. It also implements an ARIES-like recovery manager to handle failures. The system periodically performs quiescent checkpointing. For concurrency control, it implements the strict 2PL as well as the multiple-granularity locks to prevent phantoms due to insertions. In addition, it employs B-Tree indexes to speed up searches.

To support determinism, we modify the components of transactions processing (which are colored in black in Figure 11). We implement the conservative locking and record the parameters of stored procedures as described in Calvin [38]. We also enhance our stored procedure manager to so that it can extract the read and write sets from each transaction.

B.2 Distributed Deterministic Systems

Our distributed system implementation is based on Calvin [38]. Figure 12 shows the architecture of our distributed system, which consists of three main modules: the core module, the communication module, and the Distributed Deterministic (DD) module. The core module is basically a simplified version of the database system described in the last section. The communication module handles the group communication between servers and clients. When receiving requests from the clients, our system employs the Paxos protocol (precisely, the Zab protocol) to reach the global consensus about the order of each incoming requests so that all server nodes can process the transactions in the same order. The ordered requests will be forwarded to the DD module on every server for transaction management and concurrency control.

The DD module will process the total ordered requests, create the transactions and then make sure they will be executed in the same order on every server by leveraging the conservative lock technique. Finally, when it comes to reading or writing a record during the execution of a transaction, the core module will be called and we can obtain the records by accessing database (via an executable plan). In short, the communication module handles the

network communication, the DD module handles the transaction management, and the core module handles the jobs that a single node should do.

The communication module takes the responsibility of the group communication between servers and clients. We use an open-source group communication toolkit as the backbone of this module. Whenever a server receives a batch of requests, it will first deliver the requests to the communication module, and try to broadcast the messages to other servers. The Paxos protocol is used here to make sure all the servers reach a global consensus in a network of unreliable processors.

In distributed deterministic database system, transactions need to be executed in the same order, which is decided by the Paxos protocol in the communication module, across all server nodes. This is achieved by deterministic locking. The locking procedure is divided into two phases, the requesting phase and obtaining phase. A transaction first analyzes the read/write set and requests its locks sequentially, and the locks will be maintained and stored in a queue for each record. When a transaction actually starts execution, it can obtain the locks only if it is in the first place in the queue, and it will pop itself from the queue after releasing the locks. Since the locks are stored in the queues in total order, the transactions can be executed in parallel after the request phase without violating the global transaction order. In the requesting phase, a transaction will analyze the read/write set and decide which server should participate in this transaction. Only the participating servers need to request the locks and execute the transaction. If a transaction is distributed over multiple servers, it will have multiple participants. The participants which have any record in the locally stored write set will be the active participants while the ones that only have records in the read set will be the passive participants. When the transaction starts, it will first perform a local read on every participant node and then all nodes will forward the results to every active participant, which in turn will proceed to execute the transaction logic and then apply the local writes.

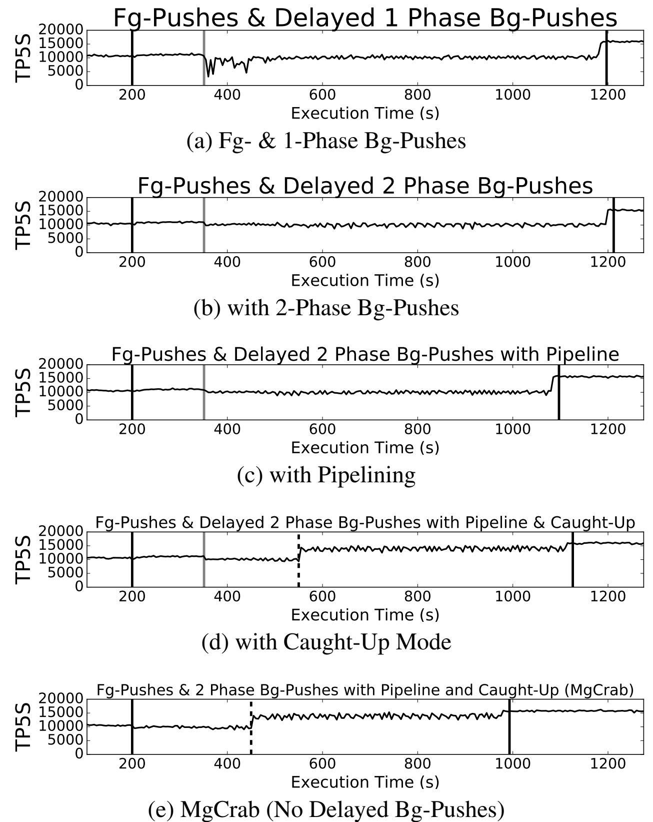
In order to implement MgCrab, we modify the components of transactions processing (which are colored in black in Figure 12). We alter the transaction routing such that transactions accessing the data in the migration range can run on both the source and destination node simultaneously. We also let the migration manager monitor the read/write set (through the transaction manager) to record which data have been migrated.

C. MORE EXPERIMENTS

This section shows the results of experiments that are not shown in Section 5 due to the space limitation.

C.1 Sensitivity Analysis

Here, we investigate how each part of the design discussed in Sections 3 and 4 improves performance. We use the same configuration described in Section 5.2 for the scaling out scenario with the TPC-C workload. We start from a vanilla MgCrab with only the foreground pushes (i.e., collaboratively executing transactions on both the source and destination nodes) and the one-phase background pushes. Then, we add two-phase background pushes and optimizations such as the pipelining of two-phase background pushes (see Section 3.3) and the caught-up phase (see Section 4.4) one by one and compare the performance of these variants. Note that we delay the starting time of background pushes here in order to demonstrate the individual impact of foreground pushes. Figure 13 shows the results. We can see from Figure 13(a) that the foreground pushes themselves are sufficient to maintain the system



(a) Fg- & 1-Phase Bg-Pushes

(b) with 2-Phase Bg-Pushes

(c) with Pipelining

(d) with Caught-Up Mode

(e) MgCrab (No Delayed Bg-Pushes)

Figure 13: A demonstration of how each part of the design discussed in Sections 3 and 4 improves the system throughput in the scaling out scenario with TPC-C workload. The gray lines denote the starts of background pushes and the dash lines denote the starts of caught-up phase.

performance right after the start of migration period. This justifies the effectiveness of our "crabbing" design as shown in Figure 3. However, the performance drops right after the start of one-phase background pushes. Replacing the one-phase background pushes with two-phase background pushes, we can see from Figure 13(b) that the two-phase background pushes indeed improve system performance by preventing the normal transactions (accessing hot data) from being blocked by the background pushes (migrating cold data). Breaking the background pushes into 2 phases, however, prolongs the migration time since the transactions running the first and second phases of a background push need to be totally ordered. We can see from Figure 13(c) that the pipelining of two-phase background pushes indeed reduces the migration time indeed. Finally, after the destination node has caught up, we remove redundant workload from the source node by enabling the caught-up mode. As we can see in Figure 13(d), the overall performance jumps because the source node can concentrate on its own transactions now. Figure 13(e) shows the complete version of MgCrab without delaying background pushes.

C.2 Migration Time Trade-Off

By default, we set the chunk size of a background push to 15000 records. Now, we share our evaluation of how chunk size affects the system performance using the TPC-C benchmark. We use the same configuration and workload described in Section 5.2 for the following experiments. The results are shown in Figure 14. We can see

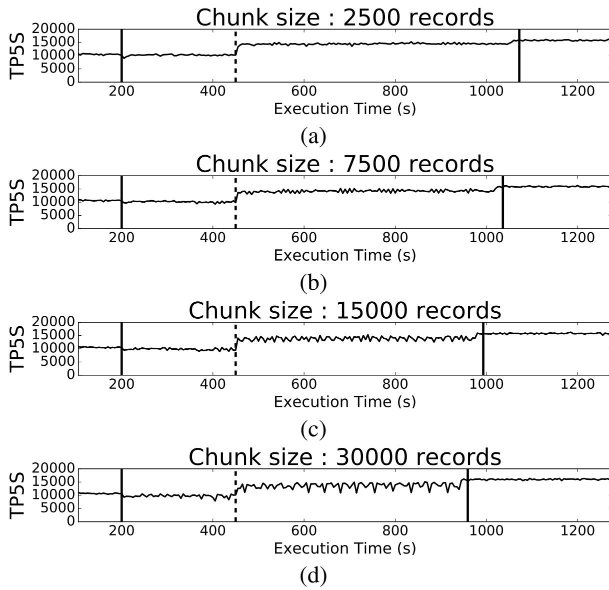


Figure 14: The trade-off between migration time and chunk size in the scaling out scenario with TPC-C workload. The vertical dash line represents the start of caught-up phase.

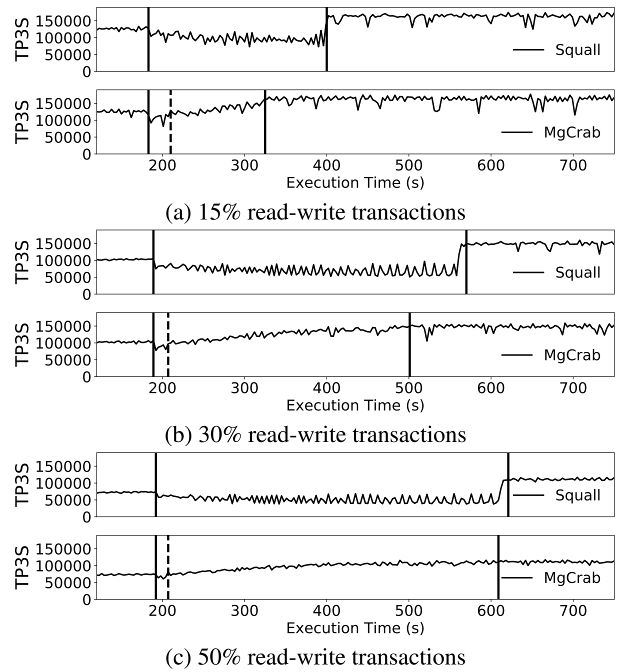


Figure 15: The impact of read-write transactions. The first and second vertical solid lines denote the start and end of migration respectively. In MgCrab, the vertical dash line denotes the beginning of the caught-up phase. TP3S stands for transactions per 3 seconds.

that, first, using the larger chunk size reduces the migration time. If the total amount of migrating data is fixed, sending the data using a larger chunk size can save the overhead of communication and storage access. Second, the larger chunk size has relatively little impact on the system throughput in the crabbing phase (before the vertical dash line) than in the caught up phase (after the vertical dash line) since the phase 1 and phase 2 transactions of a background push does *not* conflict with any user transaction on the source node, and the source node can "hide" the migration cost by executing user transactions actively. Third, the caught up phase where the destination node takes over the transaction processing (after the vertical dash line) should be used with caution as it may "expose" the drawback of using a larger chunk size: the second phase/transaction of a background push may be very long and block many user transactions. Most previous live migration approaches also suffer from this drawback.

In summary, MgCrab makes the migration time less sensitive to the chunk sizes. This mitigates the trade-off between the system performance and migration time so the system administrators can explore more flexible configurations and spend less time on fine tuning the system parameters. In our experiments, 15000 records for each chunk seems to strike the balance between the migration time and performance.

C.3 More YCSB experiments

In order to understand the impact of different types of workloads on MgCrab and the baselines, we adjust the parameters of the YCSB benchmarks to conduct the following experiments. The default settings of the benchmark is the same as in the scaling-out scenario in Section 5.1 except that we use 300 client threads in the experiments because 600 client threads create too heavy loads for some settings.

C.3.1 Impact of Read-Write Transactions

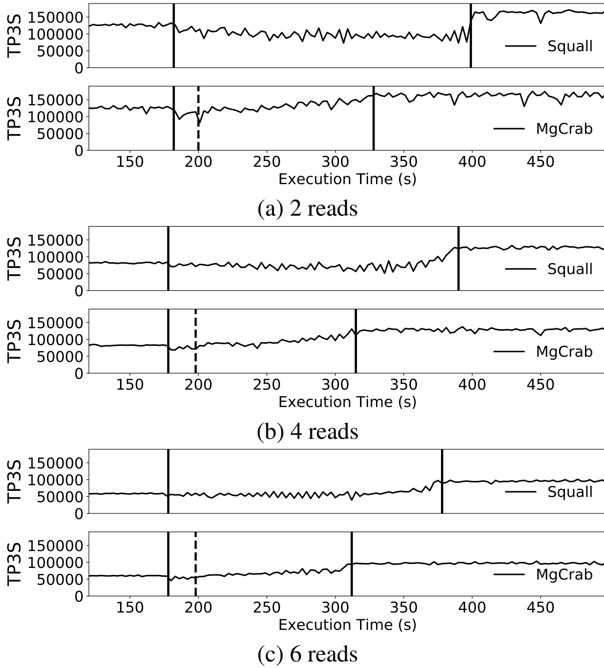
We vary the ratio of read-write transactions from 15% to 50%. We can see from Figure 15 that MgCrab outperforms Squall and has more stable throughput on all situations because the two phase background pushes of MgCrab are much more lightweight than the background pushes used by Squall. In addition, we observe that read-write transactions prolongs the migration time. This is because the read-write transactions create a high load for the storage engine on destination node and slows down background chunk insertion.

C.3.2 Impact of Read Set Size

Next, we evaluate the impact of the size of read-sets. We can see from Figure 16 that when there are more reads in a transaction, the migration time becomes shorter. This is because the storage engine on the destination node receives fewer writes and can spare more resources to process background pushes. Also, MgCrab has shorter migration time than Squall.

C.3.3 Impact of Distributed Transactions

In our original setting of the YCSB benchmarks, there is no distributed transaction. To create a distributed transaction, we make a transaction read one more record from another node. We conduct the experiments with 0%, 10% and 25% of distributed transactions. Figure 17 shows the results. We can see that distributed transactions certainly introduces considerable cost to both MgCrab and Squall. As the ratio of distributed transactions increases, Squall gives multiple performance drops during the migration because distributed transactions on the destination node slows down all other nodes. MgCrab mitigates this problem by letting both the source and destination nodes serve data to other nodes. In addition, the two phase background pushes are lightweight and reduce the overhead of migratiing cold data. We can see that the throughput remains stable after MgCrab enters caught-up phase even when there are lots of distributed transactions.

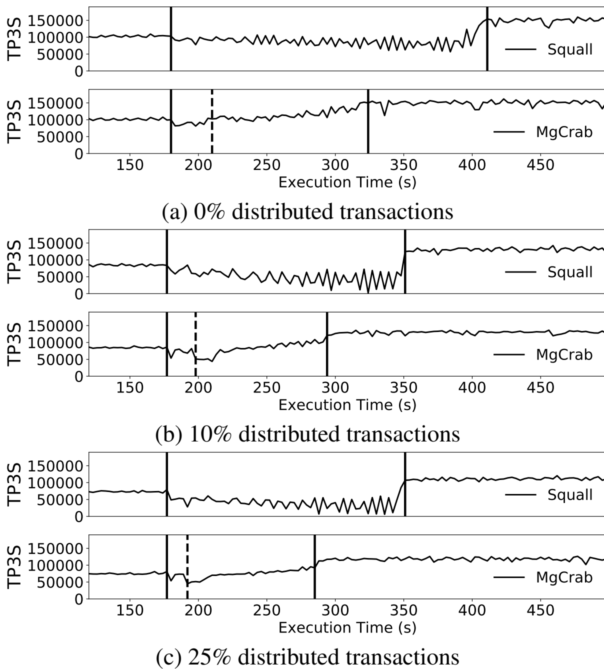


(a) 2 reads

(b) 4 reads

(c) 6 reads

Figure 16: The impact of the size of read-sets. See the description of Figure 15 for the meaning of vertical lines.

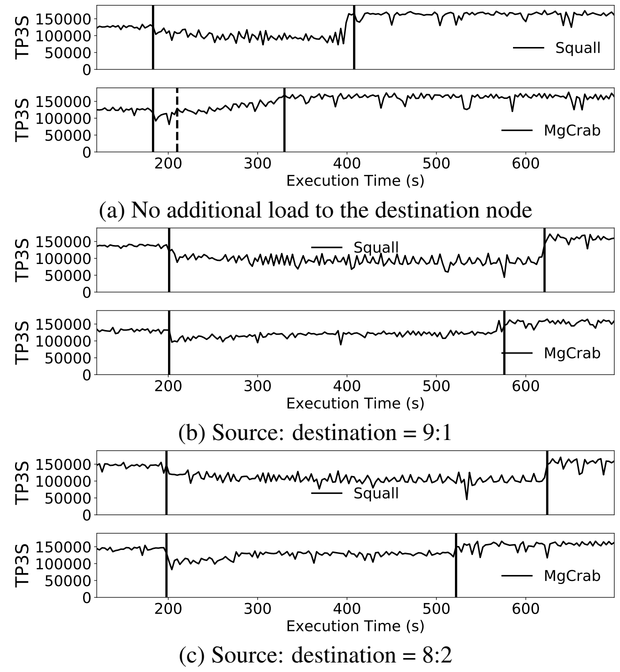


(a) 0% distributed transactions

(b) 10% distributed transactions

(c) 25% distributed transactions

Figure 17: The impact of varying the size of read-set. See the description of Figure 15 for the meaning of vertical lines.



(a) No additional load to the destination node

(b) Source: destination = 9:1

(c) Source: destination = 8:2

Figure 18: The impact of the degree of load-unbalance. See the description of Figure 15 for the meaning of vertical lines.

C.3.4 Impact of Load Balancing

In this experiment, we evaluate the performance of MgCrab when the source and destination nodes have different loads. In the original YCSB setting, the destination node has no workload and the source node migrates a partition to the destination such that both nodes will have the similar loads after the migration. We extend this setting by issuing initial loads to the destination node.

Figure 18 shows the results of the experiments. First, we can see that the migration time is much longer when there are loads on the destination node. Because the destination node has to handle not only the migration but also its original workloads. Second, we observe that the throughputs of both MgCrab and Squall drop at the beginning of the migration. The reason is that there is a large number of foreground pushes/pulls at the beginning of the migration. Inserting those data to the destination node slows down the transaction processing. However, MgCrab still gives higher throughput because the dual execution allows the the winner node to hide the latency of the slower node. Note that when the destination node has existing workloads, it is hard to tell which node will be faster. Also note that MgCrab does not enter the caught-up phase in Figures 18(b)(c) because the destination node, with a very busy storage engine, catches up with the source node only at the time the migration is about to end.

C.4 Comparison with Albatross

In this section, we compare MgCrab with a baseline, Albatross [12], that executes transactions on the source node. We adapt the idea of Albatross such that it can run on and take advantage of a deterministic database system. We call this adaptation Albatross+. We run the Albatross+ and MgCrab in the scaling-out scenario with the TPC-C workload described in Section 5.1.

We can see from Figure 19 that both Albatross+ and MgCrab have very stable throughput at the beginning of the migration because Albatross+ executes transactions on the source node and MgCrab

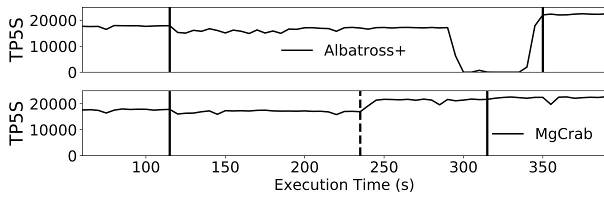


Figure 19: The changing of throughput during scaling-out a cluster under the TPC-C workload. See the description of Figure 15 for the meaning of vertical lines.

has the dual execution. However, Albatross+ has to perform an atomic hand-over to migrate the changes during the migration. This causes the partition temporarily unavailable and also blocks the transactions accessing the migrating partition. This further blocks other conflict distributed transactions and eventually halts the entire system. MgCrab, on the other hand, has stable throughput during the entire migration and does not require such an handover.