Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems

Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, Shan-Hung Wu
Nation Tsing Hua University
Taiwan, R.O.C.
{yslin,ctsai,tylin,yschang}@datalab.cs.nthu.edu.tw,shwu@cs.nthu.edu.tw

ABSTRACT

Deterministic database systems have been shown to significantly improve the availability and scalability of a distributed database system deployed on a shared-nothing architecture across WAN while ensuring strong consistency. However, their scalability and performance advantages highly depend on the quality of data partitioning due to the reduced flexibility in transaction processing. Although a deterministic database system can employ workload driven data (re-)partitioning and live data migration algorithms to partition data, we found that the effectiveness of these algorithms is limited in complex real-world environments due to the unpredictability of machine workloads. In this paper, we present Hermes, a deterministic database system prototype that, for the first time, does *not* rely on sophisticated data partitioning to achieve high scalability and performance. Hermes employs a novel transaction routing mechanism that jointly optimizes the balance of machine workloads, data (re-)partitioning, and live data migration by looking into the queued transactions to be executed in the near future. We conducted extensive experiments which show that Hermes is able to yield 29% to 137% increase in transaction throughput as compared to the state-of-the-art systems under complex real-world workloads.

ACM Reference Format:
Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452827

1 INTRODUCTION

There have been recent proposals for deterministic database systems [4, 15, 25, 32, 35, 36, 40] that guarantee if a system is given the same transactional input, all nodes in the system will always end in the same, consistent final state. This guarantee has been shown [35, 36] to significantly improve the availability and scalability of a distributed database system deployed on a shared-nothing architecture across WAN while ensuring strong consistency; this is mainly because it eliminates the need for an agreement protocol (e.g., 2PC) between replicas or partitions when processing a transaction. The benefits drive the development of new commercial database systems such as VoltDB [4] and FaunaDB [2] that target high-performance applications at scale.

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. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '21, June 18–27, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452827

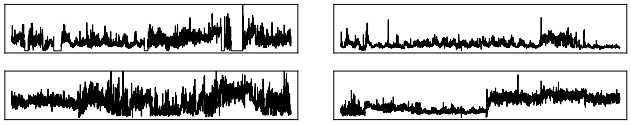


Figure 1: 30-day workloads of some nodes in a cluster owned by Google show unpredictable, episodic changes at small time scales and changes due to dynamic machine provisioning. Axes x and y represent the elapsed time and CPU loads, respectively.

However, the availability and scalability advantages of a deterministic database system come at costs. One major drawback is the reduced flexibility in dynamically reordering transactions [30, 35]. This leads to a serious performance drop when, for example, the transaction needs to wait for the next command from a user, or to wait until the accessing data is brought from disk or remote nodes into the buffer pool. Therefore, modern deterministic database systems [1, 2, 4, 13, 36] usually 1) drop the support of ad-hoc queries and take only the stored procedures as input, 2) use main-memory storage or a large buffer pool, and 3) assume the presence of highly optimized data partitions (also called shards) that minimize the occurrence of distributed transactions and maximize the balance of machine loads. This paper attempts to relax the last assumption.

It is not easy to obtain good data partitions in complex real-world systems such as a multi-tenant/cloud database system serving various applications around the world [10] or a trading system used by the NYSE [22]. The workloads of each machine in these systems are usually *unpredictable* and *highly dynamic*. In order to investigate the effectiveness of current data (re-)partitioning mechanisms, we use the YCSB benchmark with the loading traces of a cluster owned by Google [28] to create a complicated and highly fluctuating OLTP workload, and emulate the system performance using a deterministic database system, Calvin [36]. We leave the details of the experiment settings to Section 5. Figure 1 shows the workload traces of some machines. We can see that the machine workloads contain many fluctuations and unpredictable spikes and shifts, which are the results of episodic events and changes of machine provisioning in the cluster.

The "look-back" approaches. We first consider data partitioning and re-partitioning algorithms that analyze system statistics in the past. We implement Clay [31], a state-of-the-art online data (re-)partitioning approach that traces the workload and migrates "clumps" of data when the system does not meet an SLA. Figure 2

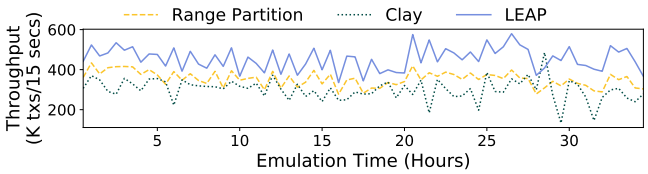


Figure 2: The performance of a deterministic database system, Calvin [36], with some state-of-the-art data (re-)partitioning approaches under the Google workload.

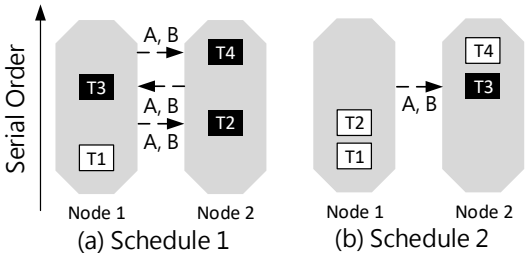


Figure 3: An example of the ping-pong problem. Each block represents a transaction, and each dashed line between nodes represents a data migration.

shows how the transaction throughput changes over time when the system uses Clay to manage its data partitions. To our surprise, Clay does not significantly outperform a naive range partitioning. This is mainly due to the episodic events, which limit the effectiveness of a "look-back" scheme because the events are not predictable from the past. Furthermore, the look-back approaches usually require an auxiliary data migration step [9, 19] to actually migrate the data. Under changing workloads, this dedicated migration step may incur a long delay that makes the data partitions outdated.

The "look-present" approaches. Another branch of studies [7, 18] aims to overcome the problem of unpredictable workloads by focusing on the present. Instead of precomputing data partitions, these approaches migrate records to a single node in an on-demand manner for each individual transaction so that the transaction and the later transactions accessing the same set of records become single-node transactions. We implement a state-of-the-art approach called LEAP [18], whose performance is also shown in Figure 2. We can see that this approach performs better than the naive range partition and Clay, but the improvement is not significant as expected. With further investigation, we observe that when there were many distributed transactions in the workload, LEAP tried to group records together to benefit most from temporal locality. As a result, almost all the active records were migrated to a single node, which creates a bottleneck. On the other hand, if a look-present approach chooses to balance machine loads, it may suffer from the *ping-pong* problem. Figure 3 illustrates two example schedules of four consecutive transactions that access records $\{A, B\}$ on a two-node system, where node 1 has $\{A, B\}$ initially. Both schedules evenly distribute the transactions to two nodes, but the first schedule requires more data migrations than the second. Unfortunately, in practice a look-present approach will more likely produce the first schedule for load balancing since it has no knowledge about T_3 and T_4 while processing T_2.

In this paper, we present Hermes, a deterministic database system prototype based on the shared-nothing architecture that achieves high transaction throughput without relying on sophisticated data partitioning. In Hermes, the transaction routing module, which is common in a distributed DBMS and was conventionally used to balance machine loads, plays additional roles of dynamic data partitioning and migration. It decides the route of a transaction by *looking into the future*. Specifically, we propose a *prescient transaction routing* algorithm that jointly optimizes load balancing, dynamic data (re-)partitioning, and live data migration by analyzing the read- and write-sets of successively queued transactions to be executed in the near future in the deterministic transaction processing flow.[1] The reasons why the concerns of data partitioning and migration can be controlled by the routing module are that 1) the transaction processing is deterministic, so given a sequence of transaction routes, the router can "foresee" how the corresponding transactions will be executed, including the cross-machine data movements triggered by distributed transactions, and 2) like the look-present approaches, Hermes migrates data on the fly with remote reads and writes of distributed transactions, so both data partitioning and migration will be deterministic to the transaction routes. As a result, Hermes can produce the schedule shown in Figure 3(b) that simultaneously balances machine loads, minimizes the number of distributed transactions, and avoids ping-pong data migrations. Hermes also equips each node with a *fusion table* that tracks the partitioning of hot records globally. This table makes the system perform stably in cases of server scale-out and consolidation because it not only facilitates the migration of hot data but also prevents the migration of cold data (triggered by a server scale-out or consolidation event) from interfering with normal transaction processing. The support of dynamic machine provisioning was neglected in traditional transaction routing literature, but is an important feature nowadays for large-scale, multi-tenant systems. The following summarizes our contributions:

- We present Hermes, a deterministic database system prototype. To the best of our knowledge, Hermes is the first deterministic database system whose performance does not rely on high-quality data partitioning unavailable in complex real-world applications.
- We propose the *prescient transaction routing* algorithm that jointly optimizes load balancing, dynamic data (re-)partitioning, and live data migration by looking into the future.
- We propose using a *fusion table* to isolate the migration of hot and cold data. This allows Hermes to maintain stable performance in cases of server scale-out and consolidation, which are common nowadays in a large-scale deployment.
- We discuss practical considerations and prove (in our supplementary materials [3]) the correctness of the proposed techniques.
- We conduct extensive experiments to demonstrate the effectiveness of Hermes and its techniques. The results show that

[1] See Section 2 for why transactions queue up on modern deterministic database systems [1, 2, 4, 13, 36] and how the read- and write-sets of each transaction are obtained.

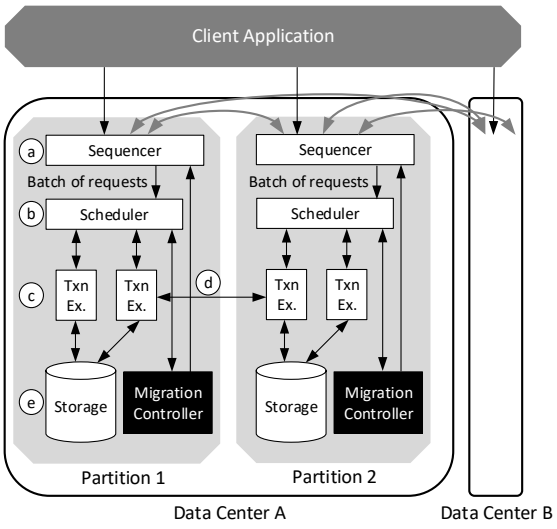


Figure 4: The system architecture of Calvin [36] (Txn Ex. stands for Transaction Executor). This system requires high-quality data partitions to deliver high performance.

Hermes is able to yield 29% to 137% increase in transaction throughput as compared to the state-of-the-art deterministic systems under complex real-world workloads. It also responds much more quickly to the workload changes and yields stable performance in the presence of dynamic hardware provisioning.

The rest part of the paper is organized as follows. We give some background knowledge of deterministic database systems in Section 2, followed by the introduction of Hermes in Section 3. We then discuss some practical considerations in Section 4, and evaluate the performance of Hermes in the next section. Section 6 reviews related work and finally, Section 7 concludes the paper.

2 BACKGROUND

In this section, we introduce Calvin [36], a deterministic database system that will be used as the baseline system when describing Hermes in later sections. Note that our proposed techniques can be applied to other deterministic database systems as well.

2.1 Deterministic Database Systems

Figure 4 shows the architecture of Calvin. Each node in a data center has a sequencer, a scheduler and multiple transaction executors. To make the results of transaction execution deterministic to the input, Calvin ensures that 1) all machines process transactions in the same total order, and 2) all sources of non-deterministic transaction aborts are eliminated. These goals are achieved by processing a (distributed) transaction as follows. First, the sequencers (Figure 4(a)) receive transaction requests issued from the clients and use a total ordering protocol such as Paxos [16, 17] or Zab [27] to determine a total order for these requests. Note that the sequencers usually order a total number of *batches of requests*, each made by an individual sequencer, in order to improve efficiency. Then, each sequencer forwards the totally ordered transaction requests to the scheduler residing on the same node (Figure 4(b)). The scheduler determines if the node should ignore or process the request. The scheduler will then forward the request to a transaction executor (Figure 4(c)) if the read- or write-set of the request overlap with the data stored locally. The transaction executor, after receiving the request, will start a transaction and obtain locks following the *conservative ordered locking protocol* to avoid deadlocks and non-deterministic transaction aborts. Note that if there is a distributed transaction that reads records from multiple nodes, all machines having the records will have to execute the transaction by reading the records and sending them to the machines owning the data to be written by the transaction (Figure 4(d)). Once the read-set is collected, the machine owning the data to be written performs transaction logic, writes the data it owns, and commits the transaction. Calvin also makes some changes to the storage engine to eliminate other sources of non-deterministic transaction aborts.

Note that Calvin, and most existing deterministic database systems, assume that the read-set and write-set of a transaction are known before the transaction starts. Since modern OLTP applications usually access the database via stored procedures, this may not be a too strong assumption. If the read-set and write-set cannot be directly determined from a stored procedure, Calvin will use an Optimistic Lock Location Prediction (OLLP) protocol that prepends light-weight reconnaissance transactions to the transaction corresponding to the stored procedure to determine the read-/write-set.

Calvin guarantees strong consistency and is able to ensure high system availability even when the nodes are deployed across the WAN. Every data center shown in Figure 4 contains a full replica of data and can be placed in a geographically separated region. With the help of determinism, there is no need for an expensive 2PC to ensure the consistency between the replicas. As compared to traditional, non-deterministic database systems, Calvin also offers the advantage that the distributed transactions can be processed in a more lightweight manner without the need for the 2PC protocol. This advantage can lead to increased scalability when data in storage are carefully partitioned such that distributed transactions are rare and workloads of machines are balanced.

However, without high-quality data partitions, Calvin and most existing deterministic database systems cannot improve the system throughput and scalability [30, 40]. This is because the conservative ordered locking protocol used by the transaction executors forbids conflicting transactions from being dynamically re-ordered, which is allowed in traditional 2PL. So, any stall in a transaction blocks all following conflicting transactions in the total order and leads to the clogging problem [35]. To avoid this problem, recent studies have proposed dynamic data re-partitioning [31, 33] and live data migration [8–10, 19] techniques trying to improve the quality of data partitions so as to minimize the stalls due to network delay in distributed transactions and overloaded machines. However, given real-world, complex workloads like the one shown in Figure 1, it is hard to find good data partitions using these techniques. It is thus crucial to devise a new approach that allows a deterministic database system to achieve high performance without relying on high-quality data partitions.

3 HERMES

In this section, we present Hermes. For ease of presentation, we use Calvin (see Section 2) as the baseline system, and assume that the read-set and write-set of a transaction are available before the transaction starts. Otherwise, the system runs OLLP [36] to first find out the read- and write-sets. We also assume that each machine node contains only one data partition, although it is easy to extend our design to a system, such as H-Store [13], where a node contains multiple partitions.

3.1 Overview

Observe that a deterministic database system such as Calvin compiles transaction requests into batches for better efficiency when totally ordering transactions. This gives us an opportunity to understand workloads *in the near future*. Based on this observation, we redesign a deterministic database system such that it can leverage the insights to future workloads to process transactions, partition data, and migrate data more efficiently and smoothly. We name this new architecture Hermes.

Hermes differs from Calvin in some key aspects. **Schedulers.** In the scheduler of each node (see Figure 4), we replace the transaction routing algorithm with the *prescient transaction routing* whose details will be given in Section 3.2. The original algorithm routes a transaction to all nodes storing records to be written by the transaction. In Hermes, a transaction is always routed to only one node (which we call the *master* node). Furthermore, instead of processing each transaction request one by one, the scheduler takes a batch of requests as input, analyzes the batch, and determines the routing schedule for all the requests in the batch at once. Note that, as long as the routing algorithm is deterministic, each scheduler can perform this action on its own *without* any additional network communication.[2] **Executors.** We also modified the transaction executor of each node such that a distributed transaction *migrates* the remote records it reads and writes to the master node. So, data migrations happen on the fly with the remote reads and writes performed by a distributed transaction on the master node. This technique is known as *data fusion* and has been used by existing look-present schemes [7, 18]. With the help of the prescient transaction routing, Hermes generalizes the idea of fusing the records of a single transaction to fusing the records of multiple transactions in a batch, avoiding the drawback of ping-pong data migration in the look-present approaches. **Fusion table.** Hermes also has a major system component called *fusion table*, which does not exist in Figure 4. In each scheduler, the prescient transaction routing defines fine-grained partitioning of data. Hermes employs a global fusion table, denoted as $\mathbb{F}$, to book-keep the partitioning. This table consists of multiple (record key, partition ID) pairs and needs to be accessed by all schedulers in the system. In order to ensure speedy accessing and to avoid additional network communication, Hermes replicates this table across all schedulers running on different nodes and leverages the *determinism* provided by a deterministic database system to ensure consistency between the replicas. Since the prescient transaction routing run by each scheduler is a deterministic algorithm, each replica always yields the same result given the same totally ordered transaction requests. Furthermore, Hermes puts the fusion table in the main memory of each node. To prevent the fusion table from becoming arbitrarily large, Hermes limits the size of the fusion table by using a deterministic replacement strategy to be described in Section 4.1 and tracking only the partitioning of *hot* records. Hermes uses a naive, static range partitioning to store cold data. Such a design has implications to system performance in the presence of dynamic machine provisioning. We will discuss this in Section 3.3.

[2] See Section 2 for how a scheduler routes a transaction request and how the Executor on the same node runs the transaction deterministically.

3.2 The Prescient Transaction Routing

Given that the transaction processing is deterministic to the total transaction ordering and that data are migrated alongside distributed transactions, a transaction routing algorithm in Hermes can control not only the machine loads but also data partitioning and live migration. We propose the prescient transaction routing algorithm that looks into the read- and write-sets of transactions in a batch to 1) minimize (the cost of) distributed transactions while simultaneously balancing machine loads, and 2) avoid the ping-pong data migration problem shown in Figure 3.

3.2.1 Objective. Given a batch of transaction requests $\mathbb{B} = \{T_i\}_{i=1}^{b}$, where b is the batch size and T_i is a transaction request, and the current data partitioning $\mathbb{P}_0 = \{P_i\}_{i=1}^{n}$, where n is the number of machines in the system and P_i is a data partition owned by a node, we formally define the goal of the prescient transaction routing as

$$\begin{aligned} &\arg\min_{\mathbb{B}', 1 \le x_1, \cdots, x_b \le n} \sum_{i=1}^{b} r(x_i; T_i \in \mathbb{B}', \mathbb{P}_{i-1}), \\ &\text{subject to } l(P) \le \theta, \forall P \in \mathbb{P}_b, \end{aligned} \tag{1}$$

where $\mathbb{B}'$ is a permuted batch where transaction requests are reordered, x_i is the route (destination machine ID) of the transaction request T_i in $\mathbb{B}'$, $\mathbb{P}_{i-1}$ is the updated data partitioning after executing transactions $T_1, \cdots, T_{i-1}$ with on-the-fly data migrations, $r(x_i; T_i, \mathbb{P}_{i-1})$ is the number of *remote* records in the read-set of the transaction request T_i given the latest data partitioning $\mathbb{P}_{i-1}$ if we route T_i to the node x_i, and $l(P)$ is the load of a partition P in the final partitioning $\mathbb{P}_b$ after all transactions in $\mathbb{B}$' are routed. For simplicity, we define $l(P)$ as the number of transactions routed to partition P in $\mathbb{P}_b$. Note that minimizing $\sum_{i=1}^{b} r(x_i; T_i, \mathbb{P}_{i-1})$ minimizes the the number of *both* remote reads in distributed transactions and data migrations. The constraints in Eq. (1) ensure that the load of every partition does not exceed a given threshold θ. We define the threshold as

$$\theta = \left\lceil \frac{b}{n} \times (1 + \alpha) \right\rceil,$$

where $\alpha \ge 0$ is a configurable parameter that denotes the tolerance to imbalanced loads. We use a ceiling function to ensure that the trivial routing plan that evenly distributes requests to machines is always a feasible solution to Eq. (1).

3.2.2 Algorithm. The solutions $x_1, \cdots, x_b$ to Eq. (1) depend on each other. If we considered all possible transaction ordering in $\mathbb{B}$ and all possible routes, there will be $b! \times n^b$ plans to evaluate. For a system with $n = 20$ nodes and a batch of $b = 20$ transactions, the scheduler has to evaluate 10^{26} plans, which is unlikely (if not impossible) to

Algorithm 1: The Prescient Transaction Routing

Input: $\mathbb{B} = \{T_i\}_{i=1}^{b}$, $\mathbb{P}_0 = \{P_i\}_{i=1}^{n}$ described by the fusion table $\mathbb{F}$ and static range config, and α
Output: $\mathbb{B}'$ and $x_1, \cdots, x_b$

1 **begin**
2 $\quad\mathbb{B}' \leftarrow \emptyset$; $l_i \leftarrow 0, \forall i = 1, \cdots, n$;
3 $\quad$// Step 1: Order and routes requests by minimizing remote reads.
4 $\quad$**for** $i \leftarrow 1$ **to** b **do**
5 $\quad\quad$find transaction T_i, $T_i \in \mathbb{B}$ and $T_i \notin \mathbb{B}'$, and x_i such that $r(x_i; T_i, \mathbb{P}_{i-1})$ is minimal;
6 $\quad\quad\mathbb{B}' \leftarrow \mathbb{B}' \cup \{T_i\}$;
7 $\quad\quad$add entries with read/write-sets of T_i as keys and x_i as value to $\mathbb{F}$ to get $\mathbb{P}_i$;
8 $\quad\quad l_{x_i} \leftarrow l_{x_i} + 1$;
9 $\quad$**end**
10 $\quad$// Step 2: Finds overloaded and underloaded nodes
11 $\quad\theta \leftarrow \left\lceil \frac{b}{n} \times (1 + \alpha) \right\rceil$;
12 $\quad O \leftarrow \{i : l_i > \theta\}$; $U \leftarrow \{i : l_i < \theta\}$;
13 $\quad$// Step 3: Reroutes requests to balance loads
14 $\quad\delta \leftarrow 1$;
15 $\quad$**while** $|O| > 0$ **do**
16 $\quad\quad$// Iterates through $\mathbb{B}'$ backward
17 $\quad\quad$**for** $i \leftarrow b$ **to** 1 **do**
18 $\quad\quad\quad$**if** $x_i \in O$ **then**
19 $\quad\quad\quad\quad x' \leftarrow$ findNewRoute$(T_i, U, \mathbb{P}_b, \delta)$;
20 $\quad\quad\quad\quad$**if** $x' \neq null$ **then**
21 $\quad\quad\quad\quad\quad l_{x_i} \leftarrow l_{x_i} - 1$; $l_{x'} \leftarrow l_{x'} + 1$;
22 $\quad\quad\quad\quad\quad$update O and U;
23 $\quad\quad\quad\quad\quad$update $\mathbb{P}_b$ by changing the values of entries in $\mathbb{F}$ having read/write-sets of T_i as keys from x_i to x';
24 $\quad\quad\quad\quad\quad x_i \leftarrow x'$;
25 $\quad\quad\quad\quad$**end**
26 $\quad\quad\quad$**end**
27 $\quad\quad\quad$**if** $|O| = 0$ **then break**
28 $\quad\quad\quad\delta \leftarrow \delta + 1$;
29 $\quad\quad$**end**
30 $\quad$**end**
31 $\quad$**return** $\mathbb{B}'$ *and* $x_1, \cdots, x_b$
32 **end**

be done in realtime. Hermes employs a greedy algorithm shown in Algorithm 1 to efficiently find an approximate solution to Eq. (1).

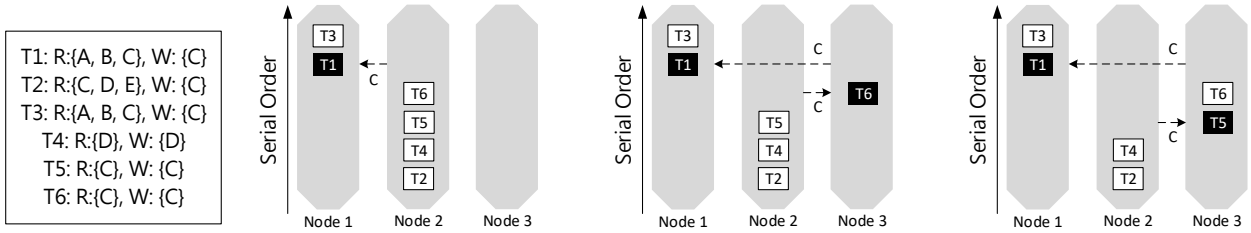
The algorithm consists of three major steps. In the first step (lines 4-9), the scheduler reorders and routes transaction requests by minimizing the number of remote reads. It greedily selects the first reordered transaction T_1 and its route x_1 such that $r(x_1; T_1, \mathbb{P}_0)$ is minimal, and then repeats this process until all transactions are reordered and routed. We consider only the remote reads here because each write will be local and result in a data migration. After the T_i and x_i are found in an iteration, the scheduler updates the fusion table $\mathbb{F}$ by adding records with read- and write-sets of T_i as keys and x_i as value. This updates $\mathbb{P}_i$, which will affect the ordering and routing of subsequent transaction requests in later iterations. In the second step (lines 11-12), the scheduler finds out the overloaded partitions O and underloaded partitions U by counting the number of assigned transactions in each partition. In the last step (lines 14-30), it tries to move some transaction requests from the overloaded partitions to underloaded ones in order to balance the loads of different machines. When it decides which node a request T_i should be routed to, it calls a subroutine **findNewRoute**$(T_i, U, \mathbb{P}_b, \delta)$. This function returns another partition $x' \in U$ for T_i that will not results in more additional "remote edges" than δ. The remote edges include 1) the remote reads of T_i if T_i is routed to x' and 2) the reads to T_i's write-set performed by subsequently ordered transactions in $\mathbb{B}'$ that are not routed to x'. At line 17, the scheduler iterates the requests in $\mathbb{B}'$ backward so that it tends to move later requests which affects less subsequent transactions. If there are still overloaded partitions after moving all eligible transactions, it will relax δ and try to move transactions to balance the loads again. This process repeats until all the constraints in Eq. (1) are satisfied.

Note that in step 1, the scheduler tends to route a transaction to the partition that contains most data in the read-set of the transaction. Therefore, the lines 7 and 23 and can be simplified to updating $\mathbb{F}$ using only the write-set of T_i as keys. After receiving $\mathbb{B}'$ and the routes produced by the scheduler, the executer (Figure 4(c)) can migrate only the data written by transactions. This allows a data record to be concurrently read (and shared locked using the conservative ordered locking protocol) by multiple, non-conflicting transactions on different nodes without contending for its ownership.

3.2.3 An Example. We use an example to demonstrate how exactly the scheduler routes a batch of requests. Suppose that there are three partitions in three server nodes, and tuple {A, B} are stored in node 1 and tuple {C, D, E} are stored in node 2. The scheduler on each node then receives a batch of requests shown in Figure 5(a) where each row represents a transaction request with its read-set and write-set. We assume that parameter $\alpha = 0$ so that threshold $\theta = 2$. After the scheduler performs the first step, ordering and routing routes based on the counts of remote reads, it generates the plan shown in Figure 5(b). We can see that transaction 1 (T_1) and 3 (T_3) are reordered to the end of the sequence because this sequence generates the least number of remote reads, which is only one in the case. If we fixed the order, there might be the ping-pong problem in the original sequence where transaction 1, 2 and 3 may migrate tuple C back and forth. This shows that reordering transactions can help Hermes avoid the ping-pong issue.

In the second step, the scheduler identifies node 2 as an overloaded node since there are 4 requests routed to it. In order to reduce the load of node 2, it tries to reassign transaction requests on it. According to the algorithm, it will reassign an request only if the assignment adds no more number of remote edges than $\delta = 1$. Moving transaction 6 (T_6) to node 3 only creates one more remote edge, and thus the scheduler reassigns T_6 to node 3 as shown in Figure 5(c). However, node 2 still has higher load than the threshold so that the scheduler tries to move transaction 5 (T_5) as well. At this time, it finds that moving T_5 will not create additional network transmission because T_5 will migrate tuple C and T_6 can reuse the same tuple. This shows that this algorithm can balance the loads by reassigning a group of transaction requests with temporal locality. As a result, it moves two transaction requests but only adds one more data migration as Figure 5(d) demonstrates.

3.2.4 Cost Analysis. Now we analyze the computational cost of Algorithm 1. There are three main steps in the algorithm, and



(a) a batch of requests (b) the plan after the first step (c) the plan after the first reassignment (d) the final plan

Figure 5: An example of routing a batch of requests. The dash lines between nodes represent network transmissions.

the most expensive step is the last step, which has $O(a^2b^2n)$ time complexity, where a is the maximum size of a read-set among all the requests in $\mathbb{B}$. This is much faster than the brute-force search of $O(b!n^b)$ time complexity. In our experiments, we observe that Algorithm 1 takes only a few milliseconds on average for each transaction under a complex workload with $n = 20$ nodes and $b = 1000$ requests per batch, which only accounts for 4% of the overall latency. We treat this as a reasonable overhead since some previous work [20, 38] also trade throughput gain with a few milliseconds delay on OLTP workloads. Furthermore, Hermes performs this algorithm in the scheduler so that this latency is *not* counted into the contention footprint of transaction execution. The prescient transaction routing has little impact on the system performance.

3.3 Dynamic Machine Provisioning

So far, we assume that the number of machines in a system is fixed. In practice, one may add or remove machines corresponding to the workload changes, but this makes the dynamic data (re-)partitioning even more challenging. Next, we show that Hermes readily supports dynamic machine provisioning.

Adding or removing a node involves moving a data partition P on a node to another, where the partition contains both hot and cold records. Existing data (re-)partitioning algorithms [31, 33] usually assign the entire P to a new node. However, this makes it hard for a live migration technique [8–10, 19] to migrate the partition in a per-transparent manner because the migration process will touch hot data that are being accessed by current normal transactions. Hermes migrates P using a hybrid approach, where the (hot) data in the fusion table are migrated using data-fusion. This can be easily done by including the added node/partition or excluding the removed node in the fusion table. On the other hand, the (cold) data not in the fusion table, denoted as P^-, are migrated using existing coarse-grained data (re-)partitioning and live migration algorithms.

To migrate the hot data in P, the schedulers (Figure 4(b)) should be aware of the change of the physical layout. Hermes notifies the schedulers of such changes by issuing a special transaction that will be totally ordered so that the schedulers will include the added node or exclude the removed node in a consistent manner. To migrate the cold data P^-, Hermes uses the asynchronous migration technique proposed by Squall [9]. The basic idea is to break the P^- into multiple chunks and then migrate each chunk using a dedicated migration transaction. With the help of the fusion table, we can significantly decrease the chance that the transactions for migrating chunks conflict with normal transactions,[3] making the system performance more resilient to changes of machine provisioning.

4 PRACTICAL CONSIDERATIONS

In this section, we discuss some practical considerations of the Hermes design.

4.1 Condensing a Fusion Table

We discuss how to control the size of the table in this Section.

The most straightforward method is to compress the table. A previous work [34] discusses how to partition a database using a lookup table which has a similar structure to that of our fusion table. Their report shows that the table can be compressed with a 2.2×~250× compression factor using Huffman encoding. The specific compression ratio depends on the characteristics of the workloads. However, the main disadvantage of this method is to trade the space with computing power. Since a lookup table is a read-intensive structure, the delay of decompression may dramatically hurt the performance of the entire system.

Another direction is to limit the number of key-value pairs stored in the fusion table. Once the number exceeds the limit, it has to evict a key-value pair based on a replacement strategy. The strategy can be any deterministic replacement strategy such as First-In-First-Out (FIFO) or Least Recently-Used (LRU). In addition to the pair, the system has to migrate the corresponding record back to its original partition. To implement this strategy, we first make the scheduler check the size of the fusion table when it routes a transaction request. If the size exceeds the defined threshold, it will evict some keys out from the fusion table as an evicted key-set. Then, the scheduler will add the evicted key-set to the write-set of the transaction, which has to migrate the evicted records back to their original partitions. Note that this transaction can soon return to the client before it migrates the records back, which means that the migration will not create any additional delay to the client who issues the transaction.

We believe that limiting the size of the fusion table is reasonable for many OLTP workloads since a large number of OLTP workloads only has small portions of hot data. This was shown in a study [39] in which 99.94% of workloads in Wikipedia only accessed 5% of data in the whole database. Also, our experiments described in Section 5 show that Hermes still outperforms previous work under Google workloads even if we limit the size of the fusion table to under 2.5% of the database size.

[3] A chunk-migration transaction may still conflict with a normal user transaction if the user transaction accesses the cold data in the migrating chunk, but this is rare.

4.2 Handling Transaction Aborts

Since a deterministic database system has eliminated all types of non-deterministic events that may change the results of execution, random aborts caused by the system will not happen in Hermes. We only need to consider the aborts due to the transaction logic as defined by users. For example, a user might request to abort if the stock level of an item is less than the amount it needs. This kind of abort may happen even in stored procedures. When a transaction decides to abort, Hermes follows the traditional UNDO process to roll back the modification of the transaction. However, the aborted transaction still has to migrate and push records according to the original plan generated by the prescient routing so that the following transactions will find all the records where they expect.

4.3 Handing System Failures

Hermes may encounter unexpected failures such as power failures or software errors. In order to handle such failures, we follow the strategy used by Calvin [36]. Each node must maintain UNDO logs for its storage and command logs [21] for the transaction requests. Each node may also periodically create consistent checkpoints [29] to reduce the time for recovery. When a node fails, a replica of the node can immediately take over. After we restart the failed node, it first undoes the modifications until it reaches the state of the latest consistent checkpoint. Then, it uses the command logs to replay the prescient routing and data fusion for executing the transactions so that it can recover the system to the latest state. If the node is participating in migrating cold data when it fails, the system can also recover the migration states by replaying the command log since the process of cold migrations is deterministic and the requests of transactions for cold migrations are also logged. Note that it may also need to synchronize its command log with another machine to update the log.

5 EXPERIMENTAL EVALUATION

In this section , we evaluate the performance of Hermes and compare it with previous work.

5.1 Systems & Environment

We implemented Hermes on an open source deterministic database system written in Java whose architecture are similar to Calvin [36]. We ran the following experiments on a cluster consisting of 31 commodity machines: 20 machines as server nodes, 10 machines as client nodes, and 1 dedicated machine as the leader in the Zab total-ordering protocol [27]. Each machine in the cluster was equipped with an Intel Core i5-4460 3.2 GHz CPU, 24 GB RAM, and a 240 GB SSD. We connected the machines with a 10Gbps switch and ran each of the following experiments after warming up the system for 120 seconds so that it produces a stable throughput.

5.2 Dynamic Data Re-Partitioning under Complex Workloads

In this section, we evaluate the performance of Hermes and the baselines given highly complicated, fast changing, and unpredictable workloads, as shown in Figure 1.

5.2.1 Baselines. We implemented the following baselines.

Calvin (vanilla). We introduced Calvin [36] in Section 2 and implemented it as a baseline. We also used it as the base system when implementing other previous work. Calvin executes transactions in a *multi-master* scheme where a transaction is routed to all the machines (called masters) that have records to be written by the transaction. This scheme eliminates the need of writing records across network but takes more resources to execute transactions than a single-master scheme.

G-Store+ (look-present). G-Store [7] is a *look-present* approach for NoSQL DBMSs. It dynamically groups records and provides atomic access to a group for clients. However, G-Store requires the clients to define which keys to be grouped and when to disband the group. Since Calvin already knows the read-set and write-set of each transaction, we adapted G-Store to Calvin by trivially grouping the keys in the read-set and write-set for each individual transaction and disbanding the group immediately after the transaction commits. We also altered the execution model of Calvin for G-Store to a single-master scheme where a transaction is routed to only one machine (called master) that has the majority of records accessed by the transaction, thus the master must pull the records not located on its partition and write them back to their original partitions.

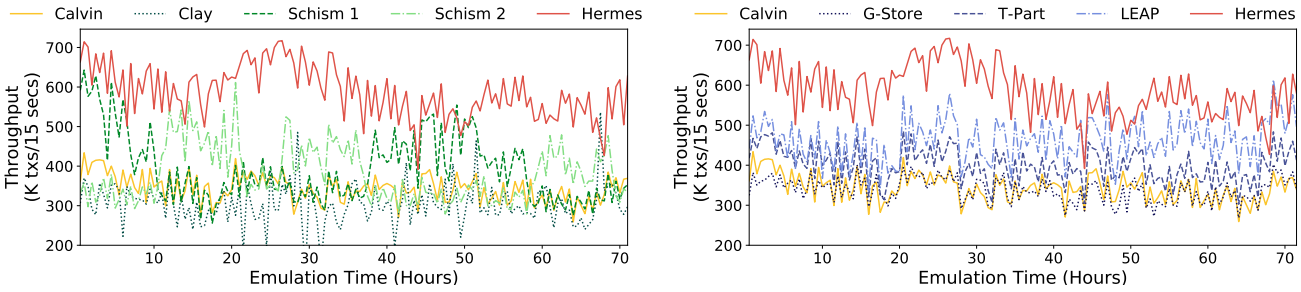
LEAP (look-present). LEAP [18] is another *look-present* approach that not only groups keys to a single master but also migrates data records to the master node for each individual transaction so that it eliminates the need of writing records back to their partitions and benefits from temporal locality. However, it does not consider load balancing and ping-pong problems.

T-Part (transaction-routing-only). T-Part [40] is a transaction processing engine designed for deterministic database systems, which executes transactions in a single-master scheme like G-Store. It optimizes transaction routing for minimizing the cost of distributed transactions while balancing loads. In addition, it proposes the *forward-pushing* technique that allows a transaction to push its writes to later transactions in the same batch of transactions in order to reduce the synchronization cost, which also eliminates the need of writing records back within the batch. However, since T-Part does not migrate data, the records transferred between transactions must be written back to their original partitions once there is no later transaction that needs the records in the same batch.

Clay (look-back). Clay [31] is a generator of data re-partition plans for partitioned databases. Clay monitors workloads and breaks down a database into small pieces (called clumps) according to the hotness and co-access frequency of data[4]. It then generates plans to migrate those dynamic blocks in order to balance machine loads. Note that, unlike LEAP, which migrates records every time that a master reads a record from another partition for a transaction, Clay migrates records only when it detects overloaded machines and starts a dedicated migration phase, which may be more heavyweight than the migration scheme used by LEAP and Hermes. As in the Clay paper, we pair up Clay with Squall [9], a live-migration technique, to move data according to the plans at runtime.

Schism (off-line, look-back). Schism [6] is an *off-line data partitioning* scheme. It models a database as a graph, where the nodes represent the records and the edges represent co-access frequency

[4]In our implementation, we generate a clump by using data ranges instead of keys because generating key-based, fine-grained clumps takes too much time. The size of the range depends on workloads.



(a) Hermes vs. Look-back Approaches

(b) Hermes vs. On-line Approaches

Figure 6: The performance of Hermes and the baselines under the complex Google workloads.

of the endpoints in transactions. It then partitions the graph using a graph partitioning algorithm. Schism partitions data from scratch and does not support incremental data re-partitioning. Periodically applying Schism to a system with changing workloads will result in a large number of data migrations. Therefore, we only used it to indicate the "optimal" data partitioning at a particular time within an experiment period.

5.2.2 Google Workloads. Based on the workload trace of machines in Google's data centers [28], we created a complex OLTP workload whose characteristics are shown in Figure 1. We defined transactions and databases by following the Yahoo! Cloud System Benchmark (YCSB) while letting the final workload of each machine look similar to that of Google. Each database has one table, and each record in the table is 1KB in size and has 10 fields. We populated 200M records for the following experiments. The metadata, the indices and the data table of the database totally occupies 360GB disk space, which is hard to be loaded into the memory of a single commodity machine. All approaches, except Schism, used range partitions (where each range has 10M records) as the initial partitions, where each machine has a 18GB database.

Each transaction accesses 2 records in the database. There are two types of transactions: the first type are read-only transactions, where each transaction reads two records; the second type are read-write transactions, where each transaction performs read, modify, and write on two records. Both types of transactions are further divided into the local and distributed transactions. A local transaction first selects a partition following a time-varying distribution proportional to the machine loads logged by Google and then accesses two records following the Zipfian distribution in the partition. This allows the local transactions to reflect the workload spike, skewness, and dynamics of Google's machines. A distributed transaction accesses a record using the same access pattern of a local transaction and another record selected from a global, two-sided Zipfian distribution defined on all keys in the whole database. The global distribution changes its peak over time repeatedly from the first to the last record in order to simulate the behavior of active users around the world in 24 hours. The changing global hot records, together with complex per-machine workloads, make the optimal data partitioning opaque and dynamic. We set the ratio of distributed transactions and read-write transactions both to 50%.

We replayed a 3-day log consisting of 20 machines of Google's cluster. However, since it is too time consuming to run each emulation for 3 days, we downscaled the emulation from 3 days to 2160 seconds by sampling every two minutes. Each emulating machine has 24GB RAM, where 6GB is allocated for buffer pool and the rest is used by the Java VM (to store objects such as the fusion table in Hermes). For more details, please see the supplementary file [3].

5.2.3 Overall Performance. Figures 6(a)(b) show how the system throughput of Hermes and the baselines changes over time under the complex Google workloads. In Figure 6(a), the state-of-the-art "look-back" data re-partitioning approach, Clay, does not significantly outperform Calvin with only static range partitions. This is mainly due to that 1) there are many episodic events which are not predictable from the past, and 2) the dedicated data-migration phase incurs delays so Clay fails to update data partitions in time. To see the optimistic performance of the look-back scheme, we study the performance of Schism. We first randomly select two periods of 12 hours long, run Schism offline given the workloads in these two periods to determine their respective "optimal" data partitioning, and then equip Calvin with the optimal data partitioning (marked as Schism 1 and Schism 2 for the two periods, respectively). In Figure 6(a), we can see that Schism 1 and 2 work well during the selected periods (40th to 52nd hours for Schism 1, and 10th to 22nd hours for Schism 2), but none of them fits the workloads in the long term. Furthermore, if Schism is run periodically, we can see from the difference between Schism 1 and 2 that there will be a high data migration cost when the system updates the "optimal" data partitioning calculated offline. The look-back approaches does not work well with the complex Google workloads.

On the other hand, as Figure 6(b) shows, the existing "look-present" approaches, including G-Store and LEAP, improve the throughput by 2% and 50% respectively because they take advantage of temporal locality. G-Store needs to put the records accessed by a distributed transaction back to their owner data partitions after the transaction terminates, thus has higher costs. LEAP leaves the records at where they are accessed (by a distributed transaction) so it improves the throughput better. However, Hermes still outperforms LEAP because LEAP may suffer from ping-pong issues and does not consider load balancing. The transaction routing approach, T-Part, also improves the performance by trying to balance loads among machines. In addition, its forward pushing technique lets a transaction directly send its records to later transactions that read

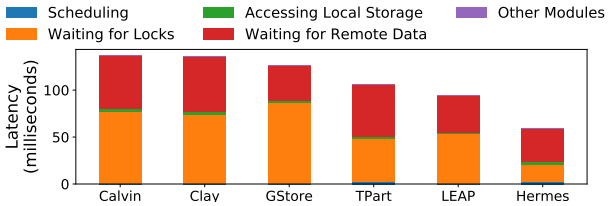


Figure 7: The breakdown of average latency.

the records; this eliminates the need for accessing remote storage, so T-Part can also benefit from temporal locality. In spite of this strength, Hermes outperforms T-Part because the prescient routing algorithm in Hermes not only routes transactions (which optimizes transaction execution as in T-Part) but also re-partitions and migrates data on the fly. In particular, T-Part has to put records back into their original partitions while Hermes uses data-fusion to avoid this overhead. To sum up, Hermes addresses all the issues above and outperforms all the baselines by 29%~137%.

5.2.4 Latency breakdown. To understand how Hermes improves performance, we tracked the time spent of each transaction action by using code injection. Figure 7 shows the latency breakdown of a transaction in each system. We have three observations. First, Hermes reduces both the average wait time for remote data and locks by 30% and 120%, respectively, because the prescient routing generally yields better data partitions that minimizes distributed transactions and balances loads. This explains why Hermes gives the best overall transaction throughput (Figure 6). Although G-Store and LEAP also reduce the wait time by grouping records on demand, both systems have no ability to balance the loads among machines. LEAP has slightly better performance since it migrates records such that transactions can benefit from temporal locality and hot nodes can shift off a little load. Second, while T-Part does not reduce wait time greatly, it has a better performance than most of the baselines. This is because T-Part has the ability to balance loads using transaction routing so that it utilizes the CPU resource better than other baselines. Third, the latency of scheduling a transaction in Hermes is about 2 milliseconds, which is 4% of the overall latency. The latency represents the time of analyzing transactions, performing the prescient routing and scheduling an Executor thread. This justifies our claim in Section 3.2 that the latency is almost negligible.

5.2.5 Resource Utilization. We also recorded the CPU and network usage.[5] Figure 8 illustrates the average CPU and network usage consumed by each transaction among the nodes. There are some notable observations. First, the CPU usage of T-Part is slightly higher than the usage of LEAP. This, again, shows that T-Part has a better capacity to balance loads among machines. However, since T-Part has to put data records back into their original partitions, its improvement is limited by the communication costs. Second, the network usage of Clay sometimes gets higher than that of other baselines. This is because Clay performs dedicated data migrations in order to meet its new data partitions. Finally, Hermes has a better ability

[5]It is normal for a machine to have a low CPU usage under the Complex Google workload because 50% of the transactions are distributed and there is a high chance of network stall.

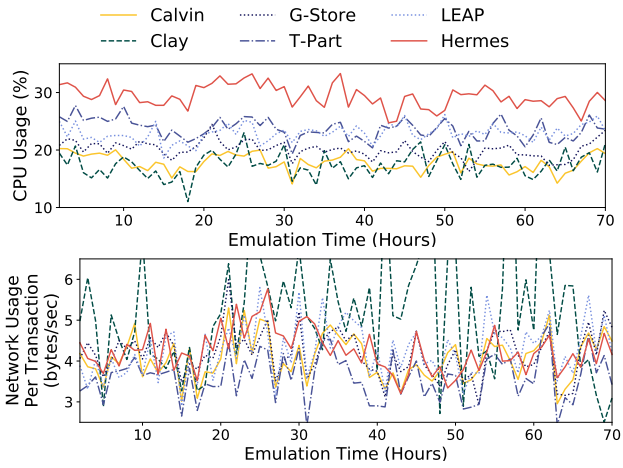


Figure 8: The changing of (a) average CPU usage and (b) network usage per transaction with the Google workload.

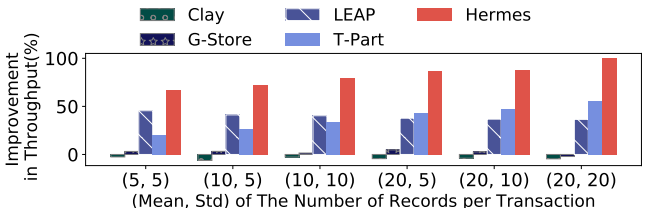


Figure 9: Impact of transaction length. Std stands for standard deviation.

to balance the loads among machines so that it can utilize more resources than the other baselines. In addition, the network usage of Hermes is almost the same with (and sometimes even lower than) that of other baselines. This shows that Hermes not only balances loads but also reduces the number of distributed transactions.

5.2.6 Impact of Transaction Length. Next, we conducted more experiments to evaluate the impact of transaction length with the Google workload. In order to create a workload with the transactions having diverse length, we made the number of records accessed by each transaction randomly sampled from a normal distribution with different means and standard deviations. Figure 9 shows the improvement in throughput over Calvin with six different settings. We can see that Hermes consistently improves performance given different combinations of the means and standard deviations. Moreover, it works even better under the workloads with higher means. This is because longer transactions implies longer blocking time for conflicting transactions, which enlarges the contention footprint. Therefore, the benefits of reducing synchronization across machines and balancing loads become more obvious.Sensitivity Analysis of the Batch Size

In order to understand how the prescient routing affects the performance of Hermes, we ran one more experiment by varying the number of batched requests (called batch size) in Hermes with the Google workload as shown in Figure 10. While increasing the batch

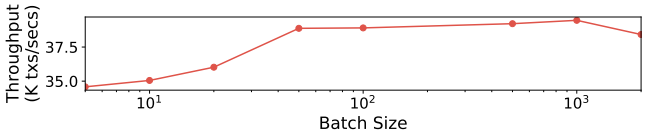


Figure 10: The trade-off between the batch size and the performance under the Google workload.

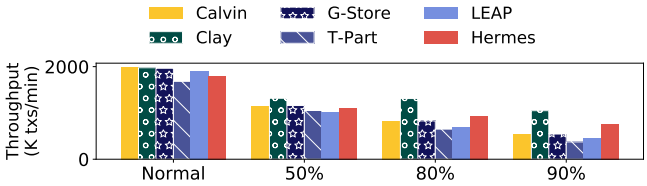


Figure 11: The average throughput of Hermes and baselines on the TPC-C benchmark with different degrees of hot-spot concentration.

size can increase the throughput to a certain degree (by allowing the prescient routing to obtain a better routing plan), the performance drops when the batch size becomes too large. This is because a large batch increases the CPU utilization of the prescient routing algorithm, which slows down the entire system. Therefore, we can not arbitrarily increase the batch size, but we can also see how significantly the prescient routing improves the performance when we choose a good batch size.

5.3 Dynamic Data Re-Partitioning under Simpler Workloads

Next, we study if Hermes is applicable to other, simpler workloads.

5.3.1 The TPC-C Benchmark. We first run Hermes and the baselines on the TPC-C benchmark [24], which has a complicated schema and transactions but well-partitioned data. The benchmark simulates a warehouse management system, which consists of nine tables and five types of transactions. We use only the New-Order and Payment transactions in these experiments since they contribute 88% of the workload and form its main characteristics. We use 20 machines and load 20 warehouses for each machine, and thus there are 400 warehouses in the database. In order to create a hot spot in the workload, we make 4000 clients send requests to the system in a close loop and modify the workload so that a significant proportion of the requests concentrate on the warehouses in the first node [9, 31, 33]. We test three degrees of concentration, namely 50%, 80%, and 90%, and the ordinary workload in our experiments.

Figure 11 shows the average throughput of Hermes and the baselines. When facing the ordinary TPC-C workload (marked as *Normal*), all approaches give similar throughput because the database is already well partitioned (simply based on the warehouse) and the loads are balanced. Hermes gives slightly lower throughput due to the overhead of batch processing and analysis. Nevertheless, its performance is still comparable to that of other baselines. As the transaction requests concentrate on the first node (lowering the quality of warehouse-based data partitioning), all approaches give

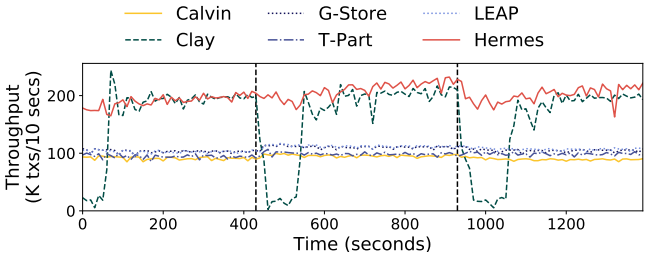


Figure 12: The throughput of Hermes and baselines under the multi-tenant workload with a changing hot spot. The vertical dash lines indicate the time where the hot spot changes from one node to another.

degraded throughput. However, Hermes and Clay starts to outperform other baselines because both of them are able to balance the loads among machines by migrating hot warehouses out from the first node. This verifies that Hermes is capable of data re-partitioning even when transactions and data schema are complicated. Note that Clay yields slightly higher throughput than Hermes in this case because the modified TPC-C workloads with hot spots still have access patterns that Clay can exploit by looking back. Specifically, Hermes may distribute the records of a warehouse on multiple partitions due to the load balancing concern in the prescient routing, but the ideal solution is to put those records together. Clay is able to capture this pattern because it collects a longer workload trace (than a mere batch) to determine the optimal partitioning. Note that Clay needs a dedicated migration phase to move the records, which may take long time to update data partitioning and cause negative impact on the throughput. This drawback does not affect the performance here because the TPC-C workloads are static over time.

5.3.2 The Multi-tenant Workload with a Changing Hot Spot. Next, we test Hermes under a multi-tenant workload that is dynamic over time but has a simpler schema and no distributed transaction. In this workload, each server has 4 non-overlapping tenant databases, each of which contains 2.5M YCSB records, and each transaction performs the read, modify, and then write operations on two records randomly selected from a single tenant following a Zipfian distribution (with the skewness parameter $\theta = 0.9$). We create 4 servers and 800 client threads submitting requests to the servers in a closed loop. As in the modified TPC-C workloads, 90% of the workloads concentrate on the tenants of one of the server nodes. However, we change the concentration target from one node to another every 500 seconds to simulate that different tenants serve different users around the world and the users become active at different time.

Figure 12 shows how the throughput of Hermes and the baselines change over time underthis workload. Since Calvin does not balance loads dynamically, it perform the worst. T-Part gives only slightly higher throughput because it cannot migrate data and its load-balancing ability is limited in the absence of distributed transactions. In contrast, LEAP can smoothly migrate records, but it cannot not balance loads. Clay is the only baseline that gives comparable performance against Hermes, which demonstrates its effectiveness of load balancing and data migration by identifying the hot records and its co-accessed records from system statistics. However, Clay

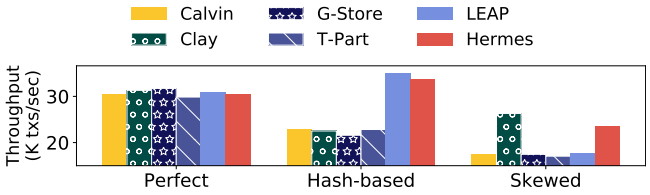


Figure 13: The impact of initial partitioning. Please see Figure 11 for labels.

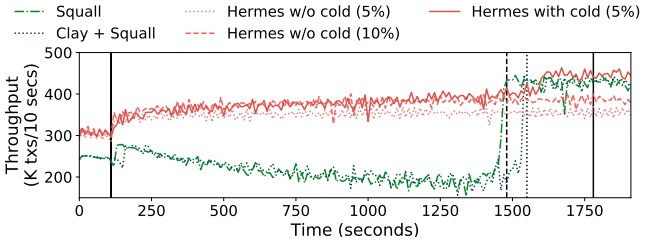


Figure 14: The changing of throughput during the scale-out scenario. The first solid vertical line indicates the event of adding a new node, the second solid vertical line indicates the end of data migration in Hermes, and the dash and dotted vertical lines indicate the end of the migration in Squall and Clay+Squall, respectively.

needs time to collect the statistics and generate a migration plan, and as such, it reacts to workload changes more slowly than Hermes. Furthermore, the data migrations block normal user transactions. These lead to the drops of throughput right after the hot spot changes.[6] Hermes gives relatively stable performancebecause it migrates data on the fly with distributed transactions, which minimizes the data migration costs. Moreover, it adapts to the changing workload quickly and starts to improve the performance earlier than Clay after the hot-spot changes.

5.3.3 Impact of Initial Partitioning. Following Section 5.3.2, we evaluate if Hermes is robust to the initial data partitioning. We use three different initial partitioning plans to load the data into the database: 1) *perfect* range partitioning, 2) *hash-based*, which decides the partition of a key using a hash function, and 3) *skewed* range partitioning, which is also a range partitioning but puts the data of first 7 tenants (about 43% of data in the database) in a single node. Note that the hash-based partitioning creates distributed transactions.

It is not surprising to see that all approaches give satisfactory performance with the perfect initial data partitions, as shown in Figure 13. With the hash-based initial data partitions, LEAP and Hermes outperform the other baselines. However, LEAP does not perform well with the skewed initial data partitions. This is because LEAP merges records according to co-access patterns, but the co-accessed records in the skewed initial partitioning do not separate across different partitions. So, LEAP keeps the workload skewed. In contrast, Clay performs best with the skewed initial partitioning but does not work well with the hash-based initial data partitions. We observe that Clay cannot find a better data partitioning plan in the hash-based scenario because all nodes are equally loaded in the long term and thus migrating a tenant to another node relieves a temporal hot spot but creates another. Among all the approaches, Hermes consistently gives good performance. This shows that the data re-partitioning and migration abilities of Hermes is robust to the initial data partitioning.

5.4 Dynamic Machine Provisioning

In the following experiments, we evaluate the performance of Hermes when the machine provisioning changes, which is common in a large cluster. Here, we consider a scale-out scenario where a system has to dynamically add a machine to a cluster with 3 machine nodes in order to relieve a hot spot via data migration. We use the multi-tenants workloads described in Section 5.3.2 with single hot spot tenant in the first node, which receives 25% of total workloads.

We consider two baselines that are able to perform dynamic data migrations. **Squall.** Squall [9] is a state-of-the-art live migration mechanism that uses reactive pulling to migrate hot data accessed by transactions while using background jobs to migrate cold data. Note that Squall is a migration executor (which decides how to migrate data) instead of a migration planner (which decides what data to migrate). We pair up Squall with the planner for migrating cold data of Hermes. **Clay+Squall.** We use Clay as the migration planner and Squall as the migration executor. This baseline also evaluates the quality of the migration plan generated by Clay.

In the experiments, the system controller sends a notification of adding a node after the system warms up. Squall immediately starts a migration with a given migration plan, whereas Clay first monitors the workloads for 30 seconds, generates a plan, and then starts a migration using the generated plan. Hermes, on the other hand, not only starts to migrate cold data but also notifies the scheduler of the change of machine provisioning such that the prescient routing will consider routing transactions to the new node. The chunk size used by the cold migrations in Hermes and Squall are both 1000 records. The cold data migration plan of Hermes simply migrates the hot tenant (the first quarter of a range of keys in the first node) to the new node. Squall uses the same plan. For ablation study, we also consider two simplified versions of Hermes that do not perform cold migrations and only migrate hot data using data fusion. The first one has a fusion table that can cache 5% of the records in the database (marked as *w/o Cold (5%)*) while the second one has a larger fusion table that can cache 10% of the records (marked as *w/o Cold (10%)*).

Figure 14 shows the results. We can see that the throughput of all approaches increases after the data migration finishes because of the increased overall computing power. However, Squall results in a severe performance drop during the migration period. This is because Squall migrates some hot records that block later transactions. Clay yields similar performance since it also uses Squall as its migration executor. Hermes performs much better. The throughput of Hermes immediately increases as it receives the notification of adding a new node (the first vertical line in the figure). Because Hermes uses the prescient routing and migrates data on the fly with distributed transactions, it is able to quickly relieve a hot spot by shifting a part

[6]The performance of Clay is optimistic here because we implement an optimization. In the multi-tenant workload, any two records in a tenant could be accessed by a transaction. Thus, Clay has to examine every record in the tenant to generate a "clump" (i.e., a group of records) to be migrated together, which takes a long time. To speed up the clump generation, we let Clay examine ranges instead of individual records.

of its workload to the new node. Interestingly, the performance can be improved by Hermes even without cold migration, demonstrating the benefits of fusing the hot records only. We can also see that with a large fusion table, which allows more hot data to be migrated, the throughput goes higher. However, migrating cold data is still beneficial because it results in higher throughput in the later stage of the migration period. With cold data migration, the transactions executing on the new node can have a higher probability to find cold data in the local storage. It is important to note that the cold data migrations in Hermes skip the hot data kept in the fusion table. Therefore, migrating cold data has no obvious negative impact on the performance in the early stage of the migration period.

5.5 More Experiments

We also performed more experiments, but we can not discuss the results here due to space limitation. Please check our supplementary materials [3] for more results.

6 RELATED WORKS

In this section, we review previous studies related to Hermes.

6.1 Data Fission

Data fission refers to the concept of partitioning the data in a system into multiple shards in order to increase the system performance. To support transactions with the ACID guarantees, the system needs to ensure the consistency between shards. Schism [6] uses a workload trace to model the database as a graph, whose nodes are records and edges are frequencies of co-accessing the endpoints. It then uses Metis [14] to partition the graph such that the number of cross partition edges are minimized and the distribution of the nodes are balanced. Sword [26] uses the same idea but also considers replication for fault-tolerance. Horticulture [23] explores possible sets of keys for partitioning and evaluates the quality of the sets using workload traces. JECB [37] goes one step farther by analyzing SQL statements in transactions. However, a fixed data partitioning can not accommodate the changing workloads, and thus E-Store [33] analyzes the workload trace and periodically migrates hot data critical to the performance. It first identifies the hot tuples in a given workload trace and redistributes those tuples to colder partitions. Clay [31] is the successor of E-Store but considers co-access frequencies between records. Compared with above work, the prescient routing in Hermes offers an advantage insofar as it does not analyze logs in the past. Instead, Hermes analyzes the current batch of transactions to decide the data partitions and can therefore deal with episodic workload changes happening in the near future. Furthermore, by integrating data migration with distributed transaction processing, it minimizes the impact of live data migration on transaction throughput.

6.2 Data Fusion

Data-fusion refers to the concept of dynamically grouping the initially scattered data for the current need (e.g., executing a transaction) in order to increase system performance. Megastore [5] allows clients to create entity groups and guarantees atomic access to the data in the entity group. However, a problem suffered by Megastore is that once a key is assigned to a group, it will not be allowed to change at runtime. In addition, Megastore requires the keys in the same group must be continuous in sorted order. Another work, G-Store [7], allows keys to be dynamically grouped while providing transactional access to a group without any restriction on the properties of keys, but it still requires clients to define the groups. LEAP [18] eliminates the need of intervention from clients by grouping keys according to the demand of a transaction. It also migrates data records accessed by the transaction to the master node, which benefits the later transactions that access the same records. Nonetheless, LEAP has the drawbacks described in Section 1. Hermes avoids these drawbacks by generalizing the idea of data-fusion to future transactions using the prescient routing.

6.3 Optimizations by Batching Transactions in Deterministic Database Systems

The idea of batching transactions for optimizations with deterministic execution is not new. Faleiro et al. [12] propose to delay the evaluation of a transaction and execute a batch of conflicting transactions at once in order to maximize cache locality. Another work, Piece-Wise Visibility [11], optimizes the execution of conflicting transactions by chopping a batch of transactions into sub-transaction pieces and scheduling the conflicting pieces to the same core such that only the conflicting parts of the transactions are blocked. Both of these techniques significantly increase the degree of concurrency between transactions. However, they target single-machine scenarios and cannot be straightforwardly adapted to distributed environments. That is, they would still face the data partitioning issue that we described in Section 2.1, which motivates our work. Hermes is orthogonal to these techniques and can be complementary to each other. In distributed environments, T-Part [40] minimizes the cost of executing distributed transactions with its forward-pushing technique where a transaction directly push its writes to later transactions in the same batch to eliminate the need of accessing storage. However, a transaction in T-Part still needs to put the data it reads/writes back to the origins because the data partitions are fixed. Hermes uses data-fusion to eliminate the need of writing data back. Due to space limitation, we cannot discuss all related work here. Please see our supplementary file [3] for more related work.

7 CONCLUSION

We present Hermes, a deterministic database system prototype that migrates data on the fly with distributed transaction processing. It uses the prescient transaction routing algorithm to analyze near-future transactions and jointly optimize load balancing, dynamic data (re-)partitioning, and live data migration. It also uses the fusion table to isolate the migration of hot and cold data. We conduct extensive experiments and the results show that Hermes does not require predefined high-quality data partitions to achieve high performance in various situations. It also responds much more quickly to the workload changes and yields stable performance in the presence of dynamic hardware provisioning. Hermes opens up new directions in joint design of transaction processing, data partitioning, and live migrations.

8 ACKNOWLEDGMENT

This work is supported by the MOST Joint Research Center for AI Technology and All Vista Healthcare (MOST 110-2634-F-007-013).

REFERENCES

[1] Elasql. http://www.elasql.org.

[2] Faunadb. https://fauna.com/.

[3] Hermes: Supplementary materials. http://www.cs.nthu.edu.tw/~shwu/pubs/shwu-sigmod-21-sup.pdf.

[4] Voltdb. https://www.voltdb.com/.

[5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

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

[7] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proc. of SoCC'10*, pages 163–174. ACM, 2010.

[8] 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 VLDB Endow.*, 4(8):494–505, 2011.

[9] 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'15*, pages 299–313, 2015.

[10] 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'11*, pages 301–312, 2011.

[11] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *Proc. of the VLDB Endow.*, 10(5), 2017.

[12] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proc. of SIGMOD'14*, pages 15–26. ACM, 2014.

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

[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[15] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication.

[16] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[17] L. Lamport et al. Paxos made simple. 2001.

[18] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang. Towards a non-2pc transaction management in distributed database systems. In *Proc. of SIGMOD'16*, pages 1659–1674. ACM, 2016.

[19] Y.-S. Lin, S.-K. Pi, M.-K. Liao, C. Tsai, A. Elmore, and S.-H. Wu. Mgcrab: transaction crabbing for live migration in deterministic database systems. *Proc. of VLDB Endow.*, 12(5):597–610, 2019.

[20] Y. Lu, X. Yu, and S. Madden. Star: scaling transactions through asymmetric replication. *Proceedings of the VLDB Endowment*, 12(11):1316–1329, 2019.

[21] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. of ICDE'14*, pages 604–615. IEEE, 2014.

[22] A. Nazaruk and M. Rauchman. Big data in capital markets. In *Proc. of SIGMOD'13*, pages 917–918. ACM, 2013.

[23] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proc. of SIGMOD'12*, pages 61–72, 2012.

[24] T. processing performance council. http://www.tpc.org/tpcc/.

[25] D. Qin, A. D. Brown, and A. Goel. Scalable replay-based replication for fast databases. *Proc. of VLDB Endow.*, 10(13):2025–2036, 2017.

[26] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *Proc. of EDBT'13*, pages 430–441, 2013.

[27] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *Proc. of LADiS'08*, page 2. ACM, 2008.

[28] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[29] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proc. of SIGMOD'16*, pages 1539–1551. ACM, 2016.

[30] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. of VLDB Endow.*, 7(10):821–832, 2014.

[31] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proc. of VLDB Endow.*, 10(4):445–456, 2016.

[32] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proc. of VLDB Endow.*, pages 1150–1160, 2007.

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

[34] A. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Proc. of ICDE'12*, pages 102–113, 2012.

[35] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. of VLDB Endow.*, 3(1):70–80, 2010.

[36] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proc. of SIGMOD'12*, pages 1–12, 2012.

[37] K. Q. Tran, J. F. Naughton, B. Sundarmurthy, and D. Tsirogiannis. JECB: a join-extension, code-based approach to OLTP data partitioning. In *Proc. of SIGMOD'14*, pages 39–50, 2014.

[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[39] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.

[40] S. Wu, T. Feng, M. Liao, S. Pi, and Y. Lin. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In *Proc. of SIGMOD'16*, pages 1553–1565, 2016.