Hermes: Supplementary Materials

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

A CORRECTNESS & FAULT TOLERANCE

According to [3], a migration technique must ensure migration safety and liveness.

DEFINITION A.1. ***Safety***. *A migration technique is safe if it meets the following conditions: (1)* Transactional correctness*: Serializability is guaranteed among transactions during migrations; (2)* Transactional durability*: updates made by committed transactions are persistent; and (3)* Migration Consistency*: a failure during migrations does not leave the system and data in a inconsistent state.*

DEFINITION A.2. ***Liveness.*** *A migration technique ensures liveness if it meets the following conditions: (1)* Termination*: if the nodes participating in a migration are not faulty and can communicate with each other for a sufficiently long period during the migration, the process will terminate; and (2)* Starvation Freedom*: in the presence of one or more failures, each migrating data set always has at least one node that can execute its transaction.*

Hermes migrates data records when performing data fusion. Thus, in this section, we prove that Hermes ensures migration safety and liveness. Our failure model tolerates node crash failures and network partitions. We assume that there is no malicious node behavior and no data lost on persistent storage.

A.1 Correctness & Consistency

THEOREM A.1. ***Transactional correctness***: *Hermes guarantees serializability during migrations.*

PROOF. Calvin uses conservative ordered locking [7, 8] to ensure serializability with the total order determined by the sequencers. First, the locking protocol is known to be deadlock-free and to have no phantom problem [4]. Second, all transactions including migration transactions for cold data are processed by the sequencers and thus are totally ordered and follow the same locking protocol. Therefore, these transactions run under the same restrictions and are guaranteed to be serializable. □

Calvin provides the following guarantees:

AXIOM A.1. ***Run to complete***: *a transaction always commits or aborts due to its deterministic transaction logic.*

AXIOM A.2. ***Determinism***: *by giving the same sequence of transaction requests in a total order, the system and data state are deterministic on each node.*

With the above axioms, we prove the durability and consistency.

SIGMOD '21, June 18–27, 2021, Virtual Event, China
2021. ACM ISBN 978-1-4503-8343-1/21/06...$15.00
https://doi.org/10.1145/3448016.3452827

THEOREM A.2. ***Transaction durability***: *Changes made by committed transactions are persistent, even in the presence of any series of failures.*

PROOF. Because Hermes ensures that all transactions are totally ordered, and all the requests are logged, we can ensure transaction durability by applying Calvin's recovery algorithm [8] as follows. First, we roll back the system to a consistent state by performing UNDO. Then, with Axiom A.2, we recover the updates of the committed transactions by replaying the transactions in the request log in order. The axiom ensures that the data always meet the same states. □

THEOREM A.3. ***Migration consistency***: *Hermes ensures that (1) the updates made by transactions during the migrations are consistent; and (2) all the nodes have a consistent view of migration states at any logical time in the presence of any sequence of failures.*

PROOF. We prove the first guarantee by contradiction. There are two possibilities causing inconsistent states: (1) inconsistent updates made by transactions; and (2) aborts due to the system control. Because all transactions including the migration transactions for cold data have to acquire locks by following conservative ordered locking protocol before writing any value, with the serializability guarantee of Theorem A.1, there is no inconsistent writes made by transactions. Hence, the only possibility is the aborts due to the system. However, Axiom A.1 ensures no system abort, and thus it is impossible to have inconsistent writes.

The second guarantee will be proved by the following arguments. The only migration state that Hermes maintains is the ownership of the data. Hermes stores the ownership of hot data in the fusion table during the prescient routing and the ownership of cold data in the migration controller during cold migrations. According to the algorithm described in Section 3.2 and 3.3 in the main paper, all the queries and updates to these states are performed in the scheduler. By replaying the request log in order, with Axiom A.2, Hermes ensures that all the nodes have a deterministic and the same consistent view. Hence the proof. □

With Theorem A.1, A.3, and A.2, Hermes ensures safety during data migrations.

A.2 Liveness

THEOREM A.4. ***Termination***: *For either a hot migration (data fusion) or a cold migration, if the nodes participating in the migration are not faulty and can communicate with each other for a sufficiently long period during the migration, the process will terminate.*

PROOF. We discuss hot migrations and cold migrations separately.

Parameter	Default Value
Total Number of Records	200,000,000
Number of Client Threads	4,000
Number of Servers	20
Buffer Pool Size per Server	6 GB
Number of Records Accessed per Txn.	2
Distributed Txn. Rate	50%
Read-write Txn. Rate	50%
Zipfian Parameter	0.99
Batch Size of Hermes	1000
Sink Size of T-Part	1000
The Size Limit of The Fusion Table	10,000,000

Table 1: Default parameters of the experiments under the Google workload.

Because each hot migration created by data fusion in Hermes is a single migration, which is performed by an individual transaction, the migration terminates as long as the transaction finishes. We also know that, according to Axiom A.1, a transaction in a deterministic database system always commits or aborts. In either situation, Hermes ensures that the transaction always migrates the data to the destination. Hence, a hot migration always terminates.

A cold migration terminates if all the chunks in the given migration plan are migrated. As described in Section 3.3 in the main paper, we migrate each chunk using a dedicated migration transaction. The limited number of data chunks guarantees the limited number of migration transactions. Since a transaction always runs to completion in a deterministic database system, the cold migration always terminates. If a failure happens, Theorems A.2 and A.3 ensures that the system and data eventually recover to the latest states. Hence, a cold migration will terminate even in the presence of failures. ◻

THEOREM A.5. ***Starvation Freedom****: in the presence of one or more failures, each migrating data set always has at least one node that can execute its transaction.*

PROOF. First, Theorem A.3 guarantees that all the nodes have a consistent view of migration states, which includes the ownership states, in any logical time even if a series of failures happens. In addition, since the prescient routing is a deterministic algorithm, the scheduler on every node always generate the same routing plan, which always assign each transaction a node to run on. Therefore, the assigned node is always aware of the responsibility of executing the transaction. Hence the proof. ◻

With Theorems A.4 and A.5, Hermes ensures liveness for migrations.

B MORE EXPERIMENTS

In this section, we present the additional experiments that provide more scenarios and aspects than the experiments in the paper.

B.1 Details of Google Workloads

First, we provide more details about how we obtained the timelines of loads from the logs of Google's clusters and the idea behind the complex Google workload.

In order to retrieve the loads of Google's machines, we downloaded a collection of logs from Google's website [6]. However, we found that the logs only record i) when a task starts and ends, ii) which machine the task was performed on, and iii) some matrices for overall usage of the machine such as average CPU usage. To obtain the timeline of CPU usage for each machine, we summed up all the CPU usage of the tasks running on the machine at every logged time point. Please refer to Figure 1 in Section 1 of the paper for some samples of our results. After we have the timelines, we followed the design described in Section 5.2.2 in the paper to create the complex Google workload. Table 1 summarizes the default parameters for the Google workload.

The idea behind the workload is the daily access patterns of on-line shopping websites and stock trading systems. For example, an on-line shopping website usually has multiple categories of products, each of which contains some popular products, which can be modeled by a Zipfian distribution in YCSB. A local YCSB transaction models a user who puts two products from the same category into his shopping cart. On the other hand, a distributed YCSB transaction models a user who puts a product from a category and another product from all possible categories (which can be modeled by a global Zipfian distribution) to his cart. In addition, each category may also has a different degree of hotness. We model the hotness among categories using the load traces that we obtained above to create varying and unpredictable patterns. As another example, this workload can also model the behaviors in stock trading systems. For example, a local YCSB transaction can represent two users who trade their stocks from the same broker so that the system needs to update their records from the same partition; and a distributed YCSB transaction can represent two users who trade stocks from two different brokers. Each broker may have a different degree of hotness like the categories of an on-line shopping website. We can simulate all above patterns in our complex Google workload.

B.2 More Experiments under Complex Workloads

Now, we present additional experiments under the Google workload that are not shown in Section 5.2 of the paper due to space limitation.

B.2.1 Impact of Distributed Transactions. In the first set of experiments, in order to investigate the impact of distributed transactions, we varied the ratio of distributed transactions in the Google workload. As we can see in Figure 1(a), Hermes leads with an 18-300% speedup when more than 25% transactions are distributed. The improvement becomes significant when there are more distributed transactions, thus showing that Hermes is much more capable of handling distributed transactions because the prescient routing finds a better data placement to localize these transactions. On the other hand, when there are no distributed transaction, the throughput of Hermes is slightly lower than that of G-Store and LEAP. This is because the original partitioning plan works best in this scenario, but the prescient routing and data fusion may move some records for load balancing, which may create some distributed transactions.

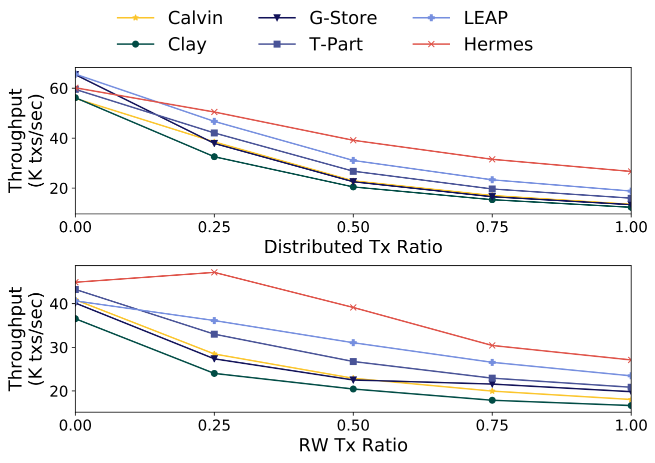

Figure 1: The impact of varying (a) the distributed transaction ratio and (b) the read-write transaction ratio of Google workloads.

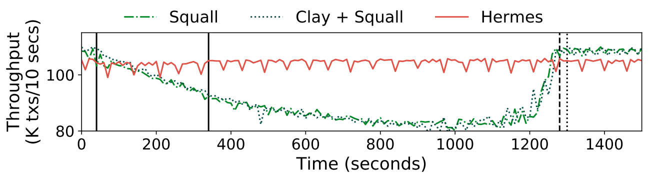

Figure 2: The changing of throughput during the consolidation scenario. The first solid vertical line indicates the event of starting to remove a node, the second solid vertical line indicates the end of data migration in Hermes, and the dash and dotted vertical lines indicate the end of the migration in Squall and Clay+Squall, respectively.

Nevertheless, the throughput is still comparable to those of Calvin and the other baselines.

B.2.2 Impact of Read-write Transactions. Next, we conducted the experiments of varying the ratio of read-write transactions. Figure 1(b) shows the impact of different ratios of read-write transactions. We can clearly observe that Hermes outperforms with a 6-70% speedup over the baselines when there are read-write transactions, but the throughput decreases when the read-write ratio comes to 0%. This is because we implement an optimization described in Section 3.2 of the paper that the system only updates data partitions when a transaction performs writes, and thus there is no change in data partition on the read-only workloads. Nevertheless, Hermes still performs slightly better than the other baselines since the prescient routing can still find a better routing plan that balances the transaction loads.

B.3 Dynamic Machine Provisioning with Consolidation Scenario

In Section 5.4 of the paper, we compared the system performance of Hermes with other baselines during the scale-out scenario of dynamic machine provisioning. We now test the performance during the consolidation of the system from 4 nodes to 3 nodes. We use the same workload as we described in Section 5.4 but make the system under loaded by letting each client wait 100 milliseconds before issuing a new request, and the goal is to migrate the tenant that was migrated out from the first node during the scale-out back to the first node. As we can see in Figure 2, the throughput does not change after the migrations because the machines are under-loaded in the first place. We observe that, as in the scale-out experiment, Squall and Clay also experience severe performance degradation because they migrate some hot records that may block normal transactions. Although the machines are not fully loaded, this stall still hurts the throughput of the system due to the increase in latency. On the other hand, Hermes keeps the throughput smooth and stable. This shows that the cold data migrations of Hermes have a very negligible impact to transaction throughput in the consolidation scenario as well.

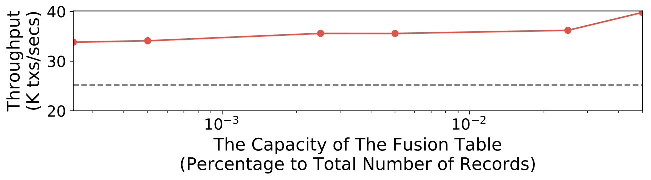

Figure 3: The impact of the capacity of the fusion table. The horizontal dash line indicates the throughput of our baseline, Calvin.

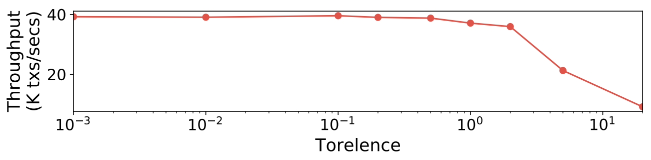

Figure 4: The impact of the tolerance setting of the prescient routing.

B.4 Sensitivity Analysis

In order to understand how each parameter of the prescient routing affects the performance of Hermes, we ran the following experiments by varying the parameters with the Google workload.

B.4.1 The Capacity of the Fusion Table. In Section 4.1 of the paper, we described an algorithm that condenses the size of the fusion table in order to control its memory footprint. We tested different capacity of the table and compared the resulting performances. As Figure 3 shows, although a larger capacity does lead to better performance (because Hermes can migrate more data for better data partitions), it suffices to have a fusion table of 50K keys, which consists of only 0.25% of all keys in the database. This is due to that most transactions in the workload only access a small portion of the database, a common scenario in real-world workloads [9].

B.4.2 The Tolerance Parameter. In Section 3.2 of the paper, we introduce a configurable parameter α for the prescient routing to decide a constraint that controls how balanced the loads of machines should be. As the value of α increases, it is easier for the routing algorithm to find a routing plan with less remote edges under the

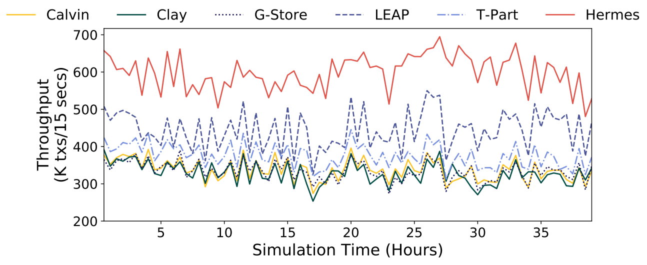

Figure 5: Impact of dynamic batch sizes.

constraint, but it also has higher chance to dispatch more transactions to a small portion of machines. Now, we evaluate how sensitive the prescient routing is to this parameter. As we can see in Figure 4, the algorithm is able to find an effective routing plan that manages a good throughput when $\alpha \leq 2$. Although the scheduler can find a plan with less remote edges as we relax the constraint, it may also stress more loads on a few machines. This is why we do not see any significant improvement in throughput when $\alpha \leq 2$. When $\alpha > 2$, the impact of overloaded machines becomes significant so that the overall throughput decreases.

B.4.3 Dynamic Batch Size. In Section 5.2.7, we presented the experiment that demonstrates how the size of a batch used by the prescient transaction routing significantly affects the system performance. However, if the batch size was not fixed, would it also affect the performance? To answer this question, we conducted a new experiment with the Google workload but added an interval between the requests such that the workload becomes sporadic and the batch size for the prescient routing varies from 10 to 1000 over time. Figure 5 shows the system throughput under the Google workload described in Section 5.2.2 of the paper. As we can see, Hermes still outperforms (up to 133%) all other baselines. This exceeds our expectations because the small batches would have made the prescient routing less effective. Looking into the resulting data partitions, we found that the routing plans generated by the large batches (even if they do not appear often) are enough to properly partition data. Note that the large batches together contain much more transactions than the small batches and thus have a stronger impact on performance.

B.4.4 Single-Request Batches. Hermes relies on batching to explore possible data partitions for the incoming transactions. If Hermes did not batch transactions, it would degenerate to LEAP, which does not batch but fuses records together for each transaction. In order to verify this statement, we ran two more experiments where the batch size is fixed to 1. In the first experiment, we used the normal Google workload described in the paper. Figure 6(a) shows the results. As one may expect, Hermes degenerates into LEAP because there is no future knowledge for Hermes to exploit when the batch size is 1. In the second experiment, we assume that the single-transaction batches are due to a light workload. We increase the intervals between the requests in the Google workload such that Hermes can only process a batch of size 1 at a time. Figure 6(b) shows the results. In this case, all approaches perform the same.

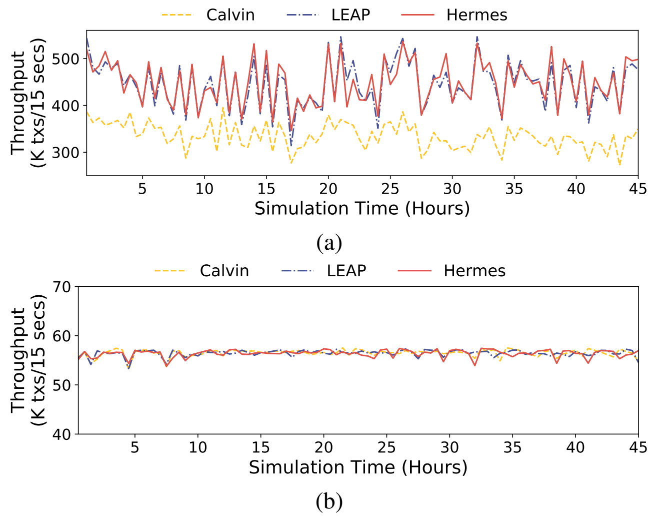

Figure 6: The comparison with Hermes, LEAP and Calvin under (a) the normal (heavy) Google workload and (b) the light Google workload.

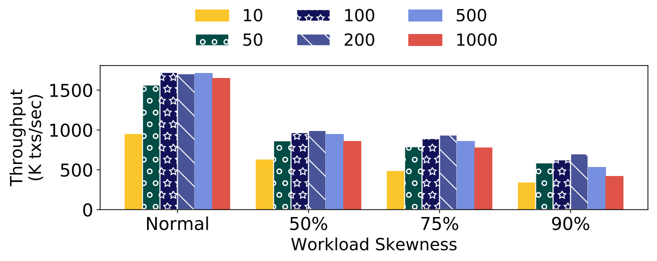

Figure 7: The impact of batch sizes under the TPC-C workload.

B.4.5 The Batch Size for the TPC-C benchmark. We have tested how batch size affects performance under the Google workload. Next, we evaluate how this parameter impacts the performance under the TPC-C benchmark. We ran an experiment as shown in Figure 7. We found that the best batch size for TPC-C is much smaller than that for the Google workload. In particular, under the TPC-C workload with 90% skewness, Hermes gives 67% improvement in throughput when we reduce the batch size from 1000 to 200. This shows that a larger batch i) does not provide additional useful information than a small batch because the TPC-C workload is static, and ii) introduces more overhead (e.g., CPU computation) to the prescient routing.

B.5 Considering More Factors in the Prescient Transaction Routing

The prescient transaction routing of Hermes reduces the number of distributed transactions and balances the loads among machine with fine-grained transaction scheduling. It also avoids the ping-pong problem that we described in Section 1 of the paper. In order to keep the routing algorithm fast, we design a simple algorithm that only models the number of remote edges and the number of accessed

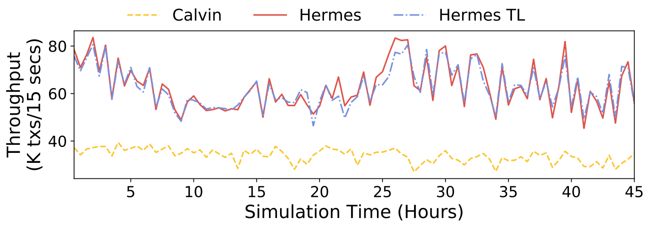

Figure 8: The performance of Hermes with the original prescient routing algorithm (marked as *"Hermes"*) and the modified algorithm that considers transaction length (marked as *"Hermes TL"*) under the complex Google workloads.

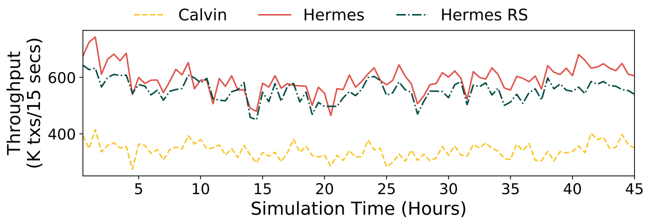

Figure 9: The performance of Hermes with the original prescient routing algorithm (marked as *"Hermes"*) and the modified algorithm that considers the size of records (marked as *"Hermes RS"*) under the complex Google workloads.

records for each transaction. However, it is possible to model more characteristics of a transaction into the algorithm. In this section, we add a few different possible factors to the algorithm and test the effectiveness of these modified algorithms with the Google workload described in Section 5.2.2 of the paper.

B.5.1 Considering Transaction Lengths. A transaction may access different number of records, which leads to different lengths of execution time. The current routing algorithm assumes that each transaction accesses the same number of records while balancing the loads among machines. This assumption may make the routing algorithm unable to distribute the loads accurately. Therefore, we modified the algorithm such that it can take the number of accessed records into account for each transaction during the load balancing. Note that it is easy to obtain the read-set and write-set of a transaction in advance in a deterministic database system as we described in Section 2 of the paper, so there is lightweight to estimate the number of accessed records.

In order to compare the modified algorithm with the original algorithm under a complex workload with diverse transaction length, we alter the transactions of the Google workload such that the number of records accessed by each transaction is randomly sampled from a Gaussian distribution with mean = 20 and std = 10. We choose this mean and standard deviation because this distribution is similar to the distribution of transaction lengths in the TPC-C benchmark. Figure 8 shows the system throughput given by the two algorithms over time. The modified version does not outperform the original algorithm. This is because that OLTP transactions are generally short and access only a small set of total records, and therefore the main bottlenecks are the communication and synchronization cost of distributed transactions.

B.5.2 Considering Record Size. Hermes uses data fusion to migrate records for each transaction such that the latter transactions may benefit from the temporal locality. However, the prescient routing models the cost of data migrations simply through the number of remote edges. This may not be accurate because each record may have different size in bytes. In order to understand whether it is necessary to consider size of records in the prescient routing. We modified the routing algorithm such that it weights the cost of moving remote records by the size of the records. We also modified the database of the Google workload so that each partition has a table with different size of records ranging from 200 Bytes to 1KB.

Figure 9 shows the results. Surprisingly, the modified algorithm does not improve the performance. After profiling the system, we found that the modified algorithm tends to move small records rather than large ones. This prevents the large but hot records from moving freely, and in turn leads to imbalanced buffer usage across machines. The machines having more large but hot records by chance would become overloaded easily because the records consumes more buffer space and incurs more I/Os. Since the bottlenecks of Hermes are the synchronization cost between machines, the overloaded machine slows down the entire system.

C MORE RELATED WORK

In this section, we review more previous studies related to Hermes that are not be able to be put in Section 6 of the paper due to space limitation.

Some shared-storage systems such as MemSQL [1] and NuoDB [2] designate transaction execution and data storage to separate machine nodes. To execute a transaction, a transaction execution (TE) node has to pull requested records from storage nodes. A TE node can cache records to gain benefit from temporal locality, which is similar to other data-fusion techniques. However, to ensure consistency while modifying a tuple, a TE node has to acquire write locks from all other TE nodes. Hermes relies on a deterministic architecture so that it does not need to coordinate with other nodes during transaction execution.

STAR [5] is a work that merges the idea of data-fusion and data-fission together. It first divides a database into partitions and then stores two replicas of the database in two different way. The *multi-machine* replica stores each partition on a separate machine while the *single-machine* replica stores all the partitions on a single machine. STAR then classifies transactions to single-partition transactions and multi-partition transactions by considering whether a transaction accesses the records stored on more than one partition. During the execution of transactions, it firsts executes single-partition transactions on the multi-machine replica, synchronize the records of both of replicas, and then execute multi-partition transactions on the single-machine replica. This scheme eliminates the need of coordination for distributed transactions, thus it significantly improves scalability. However, this scheme requires the memory of the machine that stores the single-machine replica to be big enough to store the entire database, which may not hold in practice. Hermes uses the prescient routing and data-fusion to (re-)partition database on the fly without such memory requirement.

REFERENCES

[1] Memsql. https://www.memsql.com/.

[2] Nuodb. https://www.nuodb.com/.

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] 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.