T-Part: Partitioning of Transactions for Forward-Pushing in Deterministic Database Systems

Shan-Hung Wu, Tsai-Yu Feng, Meng-Kai Liao, Shao-Kan Pi, Yu-Shan Lin
National Tsing Hua University
shwu@cs.nthu.edu.tw, kayfeng@appier.com,
{mkliao,skpi,yslin}@netdb.cs.nthu.edu.tw

ABSTRACT

Deterministic database systems have been shown to yield high throughput on a cluster of commodity machines while ensuring the strong consistency between replicas, provided that the data can be well-partitioned on these machines. However, data partitioning can be suboptimal for many reasons in real-world applications. In this paper, we present T-Part, a transaction execution engine that *partitions transactions* in a deterministic database system to deal with the unforeseeable workloads or workloads whose data are hard to partition. By modeling the dependency between transactions as a *T-graph* and continuously partitioning that graph, T-Part allows each transaction to know which later transactions on other machines will read its writes so that it can *push forward* the writes to those later transactions *immediately after committing*. This forward-pushing reduces the chance that the later transactions stall due to the unavailability of remote data. We implement a prototype for T-Part. Extensive experiments are conducted and the results demonstrate the effectiveness of T-Part.

1. INTRODUCTION

Modern OLTP workloads have two salient features. First, most transactions are short and drawn from predefined stored procedures consisting of no user interaction/stall [27]. Applications use stored procedures to cut down on the number of round trips between themselves and the database systems. Second, the data being accessed change over time and may not be easily partitionable [9,31]. For example, it is difficult to partition the data for social networking applications, as their schemas often consist of many n-to-n relationships.

The first feature motivates re-examination of the *deterministic database systems* [15, 29, 30], a type of distributed OLTP database systems that requires different machines (holding data partitions or replicas) to process all relevant transactions in the same total order (or an equivalent one). This allows transactions to produce the same results on different machines, avoids costly 2PC for data synchronization, and significantly increases system throughput on a large cluster of commodity machines while guaranteeing strong consistency between replicas. On the other hand, the totally ordered execution of transactions makes the throughput vulnerable

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06. . . $15.00
DOI: http://dx.doi.org/10.1145/2882903.2915227

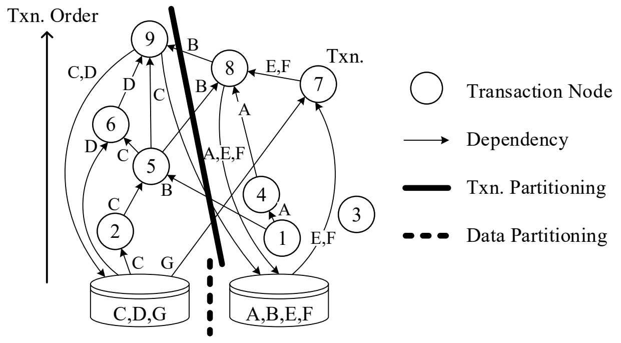


Figure 1: Partitioning of transactions and the forward-pushing.

to long transactions [29], as short transactions ordered after a long one cannot be reordered on-the-fly to complete first and to prevent the long transaction from hurting throughput.

While OLTP transactions are generally short, there are two common cases that a transaction will be prolonged in a distributed database system: when it is executed on an overloaded machine/storage and when the data being accessed span multiple machines. To work around these cases, most existing deterministic database systems require that the data in storage are well-partitioned *at all times* so the loads of machines are balanced and the number of distributed transactions is minimized.

Unfortunately, in practice it is hard to keep the data well-partitioned at all times. The second feature of modern OLTP workloads makes static, coarse-grained data partitioning insuffficent, and motivates studies on dynamic, fined-grained data partitioning driven by workloads [9,20,21,34]. The basic idea is to employ the data access patterns in the workload traces to determine better partitions. However, due to its "looking back" nature, the workload-driven data partitioning only finds good partitions *in the past*, and gives no guarantee on the quality of partitions when facing the changing workloads. Furthermore, due to the large volume of data, frequent data partitioning (so to keep up with changing workloads) may incur unacceptable computing/data migration cost.

In this paper, we present *T-Part*, a transaction execution engine for deterministic database systems that achieves high throughput in the presence of dynamic workloads and imperfect data partitions. Observe that before executing a transaction, each machine in a deterministic database system needs to assign that transaction a place in the total order (follow in which all machines execute transactions) and to analyze the read and write sets of that transaction. Therefore, the dependency between *pending transactions* (i.e., transactions that have entered the system but not yet executed) can be known early. Using this dependency, T-Part *partitions pending transactions*, by first establishing a *T-graph* whose nodes and edges denote transactions and their dependency respectively, and then finds the partitions of that graph. Figure 1 gives an example of transaction partitioning. T-Part ensures that 1) nodes/transactions spread evenly among partitions, so that machines, each assigned a partition of transactions, are evenly loaded; and 2) there exists as few cross-partition edges as possible, so that execution of transactions requires less remote data access.

Based on the partitioning results, each transaction can know exactly which later transactions will read its writes, and can "push forward" those writes *immediately after its execution*. For example, the transaction 1 in Figure 1 can push forward the object B to transaction 5 right after committing. So transaction 5, upon execution, need not pull data from other machines. This early forward-pushing reduces the *synchronization between machines* (i.e., one machine stops to wait for the others to catch-up and send data it needs), a major cause of slowdown in a deterministic database system when executing distributed transactions.

T-Part also opens up numerous opportunities for sophisticated optimization of data movement and transaction execution. In terms of optimizing data movement, T-Part moves data by *looking forward*. The read/write sets of pending transactions are modeled into the T-graph. So, finding partitions of pending transactions effectively finds the partitions of the in-memory data *to be accessed in the near future*. In addition, the number of nodes/transactions in the T-graph is much smaller than the number of records in the storage. Thus, transaction partitioning can be performed much more frequently than data partitioning.[1] The above augments the workload-driven data partitioning techniques [9, 20, 21, 34] and make the system robust to changing workloads. Following summarizes our contributions:

- We propose T-Part, which allows transactions to push forward their writes to later transactions early to avoid machine synchronization. T-Part is compatible with existing deterministic database systems [15, 29, 30] and works alongside any storage with the CRUD interface and any data partitioning scheme.
- We discuss numerous performance optimization techniques enabled by T-Part, such as the principles to obtain a better T-graph.
- We implement a prototype for T-Part based on our implementation of Calvin [30]. We devise efficient partitioning algorithms that can be run by individual machines continuously without accounting for more than 0.25% of the transaction latency in general. Furthermore, these algorithms are deterministic to the total order of transactions. So different machines need not communicate with each other to reach consensus on the partitioning. T-Part supports systems with fully distributed architecture.
- Extensive experiments are conducted and the results show that T-Part is able to make the baseline systems (with and without fine-grained, dynamic data partitioning) much more scalable when given the hard-to-partition workloads like TPC-E. In particular, it is able to eliminate more than 50% distributed transactions having network stall (due to unavailable remote data on local machines).

The rest of paper is organized as follows. Section 2 discusses the advantages and limitations of existing deterministic database systems. Section 3 introduces T-Part while Section 4 describes optimization techniques for improving the system performance. We discuss implementation challenges/pitfalls in Section 5. Section 6 evaluates the performance of T-Part and Section 7 discusses related work. Finally, Section 8 concludes the paper.

[1]In fact, the transaction partitioning is continuous in T-Part.

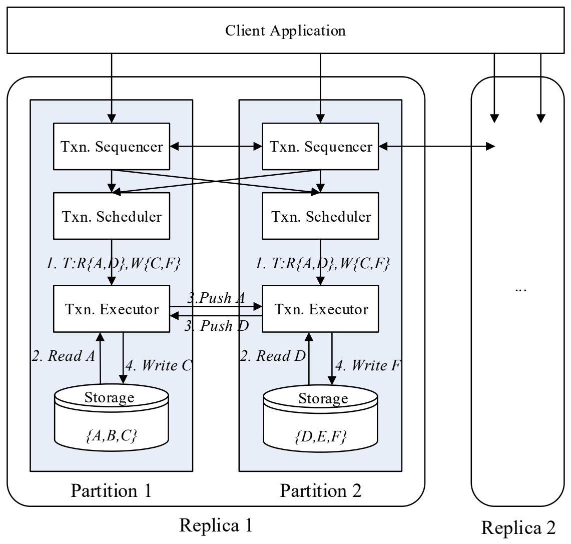


Figure 2: System architecture of Calvin [29, 30]. Each scheduler receives *all* transactions from the sequencers but only forwards a *part* of them (that touch data in the local storage) to the local executor.

2. BACKGROUND AND CHALLENGES

In this section, we review the deterministic database systems and discuss their practical limitations when facing imperfect data partitions.

2.1 Deterministic Database Systems

Assuming that all transactions are short and defined as stored procedures without user stall, recent studies [15, 29, 30] show that the deterministic database systems can be advantageous to high scalability and availability. Here, we briefly review the architecture of Calvin [30], shown in Figure 2, based on which we build T-Part. In this architecture, data are fully replicated across multiple data centers separated geographically for availability. A data center has a cluster of machines. To avoid hot spots, each machine holds a partition of data. Upon arrival of a transaction request, it is assigned a place in the total order and replicated to all data centers, both done by the sequencers in the system. The sequencers forward the replicated requests to all schedulers by following the total order. Each scheduler, when receiving a request, determines its read set and write set by analyzing the stored procedure,[2] and forwards the request to the local executor if the read and write sets cover any data stored locally. By *determinism* we mean that 1) all executors in the system process transactions following the total order (or an equivalent one) decided by the sequencers; and 2) there is no reason other than the stored procedure logic that can cause the transaction to abort. So, the commit/abort decision and the written values of each transaction are deterministic. Note that each scheduler receives *all* transactions from the sequencers but

[2]In order to determine the read and write sets of a transaction before its execution, the scheduler may break the transaction into multiple ones. For more details, please refer to [29, 30].

each executor processes only *partial* transactions which touch data in the local storage. The above system offers some nice features:

Lightweight distributed transactions. Execution of distributed transactions (i.e., transactions accessing data on multiple partitions) can be simplified with the aid of *peer-pushing* and *local writes*. For example, suppose a transaction T reading data $\{A, D\}$ and writing $\{C, F\}$ is being executed by two machines holding partitions 1 and 2 respectively (Figure 2(1)). Both machines recognize that this transaction is a distributed transaction and, additionally, which objects are missing on the other side in order to execute that transaction. They both perform local reads first (Figure 2(2)), and then *push* to the peer those missing objects (Figure 2(3)). Once obtaining $\{A, D\}$, these machines can execute the transaction *locally*—there is no need for some expensive data synchronization protocols (e.g., 2PL and 2PC), as the determinism ensures that these two machines will always reach the same conclusion on whether to commit the transaction or not and, if so, write the same values for C and F. Furthermore, each machine needs only write objects local to its storage (Figure 2(4)).

Cross-WAN data replication with strong consistency. By replicating the transaction *requests* rather than the post-execution *writes*, the system needs *no* agreement protocol (e.g., 2PC or the group-commit protocol [17]) between replicas to commit a transaction. Note that the delay incurred by the total-ordering protocol (e.g., Paxos [8] or its variants [14, 22]) for ordering a transaction does *not* count into the contention footprint of that transaction during the concurrency control. Therefore, such delay has no impact on the system throughput. Data centers can be placed across WAN to survive from geographical disasters. Furthermore, by determinism, different replicas will always see/produce the same snapshot of data at a given total order, preserving strong consistency.

The system has a fully distributed architecture, thereby preventing a single point failure. In addition, it supports any storage engine (either memory- or disk-based) with the CRUD interface. Study [30] demonstrates its improved performance over traditional (R*-like) systems on a large cluster of commodity machines.

2.2 The Synchronization Problem

The totally ordered execution of transactions makes the system throughput vulnerable to long transactions [29], as short transactions ordered after a long one cannot be reordered on-the-fly in the concurrency control module to prevent the long transaction from blocking others. Although OLTP workloads do not tend to have long transactions, in practice a transaction can be prolonged due to either a hot machine, slow I/Os (over hot data), or remote data access. This leads to the *synchronization problem.* Recall that when processing the transaction T (Figure 2(3)), the participating machines need to collect the read set from remote ones. If *any* machine falls behind, then *all* others need to wait for that machine to catch up and to push the missing data. Later transactions conflicting with T will be blocked until the progress of participating machines becomes synchronous, which could damage the throughput badly. Figures 8(a) and (c) show the impacts of distributed transactions and skewness of machine loads on the throughput of our Calvin implementation, respectively. We can see that the occurrence of distributed transactions and workload skewness significantly degrade the throughput. The settings and further discussions will be detailed in Section 6.

One way to avoid the synchronization problem is to find good data partitions such that 1) data access spreads evenly among partitions, so that the machines have balanced loads without hot spot; and 2) the need for cross-partition data access is minimized, so that the system encounters as few distributed transactions as possible. However, as discussed in Section 1, it is hard to find good data partitions at all times in many OLTP workloads. It is crucial to have a new transaction processing technique for deterministic database systems that can cope with the synchronization problem itself.

3. OVERVIEW OF T-PART

In this section, we introduce *T-Part*, a new transaction execution engine for deterministic database systems.

T-Part consists of two components, the *scheduler* and *executor*, for the architecture shown in Figure 2. This is how T-Part works in general:

As in Calvin, a T-Part scheduler receives all transactions [3] from the sequencers and decides which of them will be routed to the executor on the local machine. It analyzes and models transaction dependency as a *T-graph*, finds the balanced partitioning of that graph, and forwards to the local executor a *push plan* along with the transactions assigned to the current machine. Then, the T-Part executor processes the received transactions and *push forward* their writes to other machine by following the push plan.

T-Part is designed to work alongside any deterministic database system with architecture similar to the one shown in Figure 2 (Calvin), any storage with the CRUD interface, and any data partitioning scheme. The outputs of T-Part schedulers and executors are deterministic to the total order of transactions so to support fully distributed architecture. Since the T-Part schedulers face all transactions (requests), they are designed to be very efficient in computing. Note that unlike Calvin, each transaction will be executed only once in some executor.

Next, we provide more details about some important steps.

3.1 Constructing T-Graph

To maximize the throughput of a total-ordering protocol, each sequencer in existing deterministic database systems [29, 30] periodically compiles its requests arriving within a time interval into a *batch* (in which the order of requests are decided), and uses the total-ordering protocol to determine the total order of that batch only. Therefore, the totally ordered transactions actually come to the schedulers *in batches.* This allows the schedulers to analyze the dependency between transactions before forwarding them to the executors.[4]

By analyzing the read set and write set of each transaction,[5] the T-Part scheduler models the transactions and their dependency as the nodes and edges respectively in a graph, called *T-graph.* Since we are looking for the chances of pushing writes from one machine to another, we model the write-read conflicts (wr-conflicts for short) as edges. Figure 3(a) gives an example T-graph based on the following transactions:

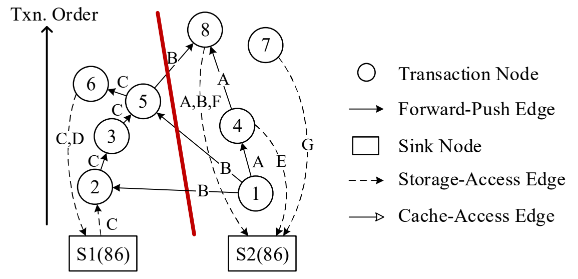
$$\begin{array}{ll} T1: W\{A, B\}, & T2: R\{B, C\}, W\{C\}, \\ T3: R\{C\}, W\{C\}, & T4: R\{A\}, W\{A, E\}, \\ T5: R\{B, C\}, W\{B, C\}, & T6: R\{C\}, W\{D\}, \\ T7: W\{G\}, & T8: R\{A, B\}, W\{F\}. \end{array}$$

An *forward-push edge* from $T1$ to $T2$ means "a wr-conflict such that if $T1$ and $T2$ are assigned to difference machines (after the T-graph is partitioned), the machine executing $T1$ could push the

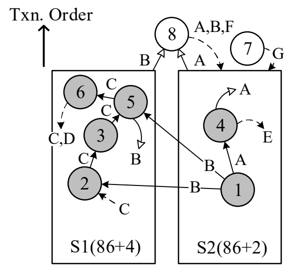
[3]Specifically, transaction requests. For simplicity, we do not distinguish transaction requests from transactions if doing so does not lead to confusion.

[4]If the transactions come individually, the schedulers can still buffer each transaction for a short time to analyze its dependency with the others. The buffering delay does *not* add to the contention footprint of transactions, and has no impact to the system throughput.

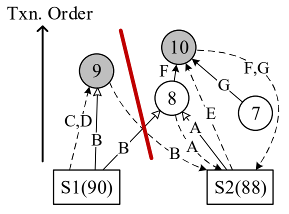
[5]As in traditional deterministic database systems [29,30], to obtain the read and write sets T-Part may break a dependent transaction into multiple ones, interleaved by the "reconnaissance" queries.



(a)



(b)



(c)

Figure 3: T-graph. (a) An example. (b) Sinking of transactions 1 to 6. (c) Arrival of transactions 9 and 10.

commonly accessed data (object B) to the machine where $T2$ resides. "

Nodes and edges are associated with weights. The weight of a node represents the processing cost of a transaction while the weight of an edge denotes the synchronization cost. For simplicity, here we assume that all node/edge weights equal to 1. We will discuss other weighting rules in Section 4.1.

T-Part works with any data partitioning strategy in the storage, and models each data partition held by a machine using a *sink node*. For example, $S1$ and $S2$ in Figure 3(a) represent two data partitions held by machines 1 and 2, with data:

$$S1 : \{C, D\}, \quad S2 : \{A, B, E, F, G\},$$

respectively. The weight of a sink node denotes the load of the corresponding machine. It is defined as the sum of weights of nodes that have already been sent to the executor on that machine, but not committed yet. In our example, we assume that both machines 1 and 2 have 86 running transactions, so the weights of $S1$ and $S2$ are 86. If a transaction reads an object not written by previous transactions, it needs to read the object from the storage on some machine (e.g., $T2$ reads C from $S1$). This is modeled in T-graph by a *storage-access edge* from the corresponding sink node (where the object belongs to) to the transaction. Conversely, when a transaction writes an object that will not be read nor overwritten by later transactions, it needs to write back the object to the storage (e.g., $T4$ writes E back to $S2$). This is modeled by another storage-access edge in the opposite direction. The storage-access edges can be weighted differently than the forward-push edges to reflect the storage access cost. Note that, even if $T6$ does not write C, it needs to write back C to storage as C was written by previous transactions but had not been written back yet. Similarly, $T8$ needs to write back A and B. We will discuss this behavior in Section 4.2.

3.2 Partitioning T-Graph

After obtaining a T-graph, the T-Part scheduler finds the balanced partitioning of that graph, as shown by the thick line in Figure 3(a), such that: 1) the sums of the weights of nodes in different partition are similar, so that each executor, once assigned a partition of nodes/transactions, is not overloaded; and 2) the sum of the weights of cross-partition edges is minimized, so that execution of transactions requires the least remote data access. Note that, by taking the weights of storage-access edges into account, T-Part tends to *move transactions to machines holding the data to be accessed.*

The balanced graph partitioning is an NP-hard problem. In practice, there exist software such as METIS [16] that can solve the problem efficiently using heuristics. However, our problem here differs from the conventional one in that it has *disconnectivity* constraints; that is, two sink nodes (denoting two data partitions on different machines) *cannot* be put into the same partition. So we cannot directly apply the software to our problem. We devise a new partitioning algorithm based on the *streaming graph partitioning* [26], which is very efficient and gives quality partitions comparable to those of METIS. Our results show that the algorithm accounts for no more than 0.25% of the transaction latency in general. We will discuss more about the challenges/pitfalls and the algorithm in Section 5.1.

3.3 Sinking & Push-Plan Generation

As transactions keep arriving from the sequencers, the partition assignment of each transaction changes over time. The T-Part scheduler periodically *sink* a batch[6] of the earliest transactions in T-graph by fixing their assignment and merging them into the sink nodes, as shown in Figure 3(b) (suppose the batch size is 6). The weight of each sink node is increased by the sum of weights of the transactions sunk to that node. Notice that some edges are transformed into the *cache-access edges* during the sinking process. We will discuss such transformation in Section 3.4.

After sinking transactions, the T-Part scheduler generates a *push plan* for the local executor. For example, the plan for the executor on machine 1 is as follows:

$T2$: Read B from cache; C from storage. Write C to cache.
$T3$: Read C from cache. Write C to cache.
$T5$: Read B, C from cache. Write B, C to cache.
$T6$: Read C from cache. Write C, D to storage.

And the plan for machine 2 is:

$T1$: Write A to cache. Push B to $T2$ and $T5$ on $S1$.
$T4$: Read A from cache. Write A to cache; E to storage.

The scheduler then forwards the plan and the sunk transactions to the local executor. Each executor receives only the plan and transactions local to a T-graph partition.

It is important to note that each sinking process needs be *deterministic* so that different schedulers under a distributed architecture (like the one shown in Figure 2) do not generate inconsistent push plans. To ensure the determinism, we require a scheduler to trigger the sinking process *whenever it sees a fixed number of totally ordered transactions in T-graph.* We call this number the *sink size*. By taking advantage of total ordering, the schedulers will generate consistent plans during each sinking process, as they are based on the same transactions in the system. However, this creates a subtle issue: the schedulers will stop sinking transactions if there are not enough transactions available in the T-graph due to the silence of clients (and sequencers too). To solve this issue, we require each

[6]The batch size need not be the same as the size of request batches from sequencers.

sequencer to add *dummy requests* into every batch to be sent to the total-ordering protocol if there are not enough requests from the clients. The schedulers discard these dummy requests when generating a push plan.

3.4 Push-Plan Execution

Upon receiving a batch of sunk transactions along with their push plan, the T-Part executor processes the transactions by following the plan.

The T-Part executor implements a key-value *cache area* (above the buffer manager of the storage engine) in memory to store the objects that are either 1) written by the earlier transactions on the same machine (e.g., $T3$ reads C in Figure 3(b)); or 2) pushed from the remote machines (e.g., $T5$ reads B). Upon execution, a transaction either reads an object from the storage (if that object has not been written by earlier transactions) or the cache area (otherwise). The transaction stalls if the object is not available in memory yet. Note that once accessed by a transaction, an object will stay in the cache until it is not used by any transaction in the T-graph. This prevents repeated storage access to the hot records, and avoids unnecessary buffer/index/file manipulation. We will discuss the cache management in more detail in Section 5.2 and its implication to failure handling in Section 5.4.

T-Part does *not* use the conservative locking for concurrency control as in Calvin. Rather, it makes use of the cache manager to implement a version-based deterministic concurrency control. The idea is to let each transaction access a unique version of a targeted record that is created by some of it's preceding transactions which writes the record last. If that version is not available yet, the current transaction stalls. More details are given in Section 5.2.

Unlike Calvin, each transaction in T-Part is processed by only one executor, and it does *not* wait for the *peer-pushing* from other machines, which requires precise machine synchronization, to obtain the remote data it needs. In T-part, transactions access remote data from *previous* transactions. This allows the data to be prepared *early* so that they are likely to be available in the local cache of machines executing later transactions. Consider those transactions that are sunk in a batch. When processing a transaction in that batch, the executor knows which later transactions in the same batch will read the objects written by current transaction. The executor can push those objects to later transactions *immediately after the current transaction commits*. For example, in 3(b), right after committing $T1$ the executor local to $S2$ pushes forward the object B to $T2$ on another machine. This early *forward-pushing* reduces the need for machine synchronization.

Note that, since the partitions of unsunk nodes (e.g., $T7$ and $T8$ in Figure 3(b)) are subject to change, transactions (e.g., $T5$) writing objects that will read by the unsunk nodes do not know where to push the objects. During the sinking process, the T-Part scheduler transforms all forward-push edges (e.g., the edge from $T5$ to $T8$ in Figure 3(a)) from the sunk to unsunk nodes into the *cache-access edges* (e.g., the cache-write edge from $T5$ to $S1$ in Figure 3(b)). So transactions sunk in the next batch will read the objects from the cache (e.g., the cache-read edge from $S1$ to $T8$) rather than the storage.

The above transformation should not affect the partitioning decided earlier (otherwise transactions will not be executed optimally from the graph partitioning point of view). To ensure this, we leverage the fact that the cache-write edges have no impact on the quality of partitioning since they are local. So, the partitioning will be unchanged if the cache-read edges have the same weights as those of the corresponding forward-push edges before the transformation. Setting the weight of each cache-read edge to the weight of the original forward-push edge is realistic since these two edges share the same endpoints at which only the memory access is involved.

The T-graph construction, partitioning, and sinking processes discussed above repeat for the continuously arriving transactions. When building the T-graph, we have to write back the data read from the sink nodes that have been cached due to the above transformation. For example, suppose two new transactions arrive:

$$T9 : R\{B, C, D\}, W\{B\}, \quad T10 : R\{E, F, G\}.$$

The updated T-graph is shown in Figure 3(c). Notice that $T9$ needs to write back B to the storage holding $S2$, as B is read from the cache.

3.5 Merits

T-Part allows each transaction to *push forward* the data needed by the later transactions early. This early forward-pushing reduces the chance of transaction stalls due to the asynchronous machines and makes the system robust to the progress skew among machines.

T-Part also augments existing data partitioning by instructing the movement of data that will be accessed in the near future. Note the the number of nodes in a T-graph is much smaller than the number of records in storage. So the T-graph can be partitioned much more efficiently and frequently than the data in storage. This alleviates the need for frequent data partitioning in order to keep up with the changing workloads, which may incur high computing/data migration cost.

Furthermore, T-Part opens up numerous opportunities for the sophisticated optimization of transaction execution, to be discussed next.

4. OPTIMIZATION

In this section, we discuss techniques enabled by T-Part that can improve transaction execution.

4.1 Refining Weights

Let w_i be the weight of a node v_i, where i is an index in the total order of transactions to be executed, and $w_{i,j}$ be the weight of an edge $e_{i,j}$ from v_i to v_j. We did not observe significant performance gain in tuning the node weights. This is because OLTP transactions are usually short and lack of user interaction.

On the other hand, edge weights have impact on the system performance. The weight $w_{i,j}$ of an edge $e_{i,j}$ should reflect the machine synchronization cost, which can be roughly defined as the amount of time v_j stalls to wait for the push from v_i. Intuitively, the larger the *transaction distance* $(j - i)$, the lower the weight should be. However, in a real system, there are many reasons that can affect the progress of a machine, and it is very hard to know the exact cost between two specific nodes. Our approach is to regard $w_{i,j}$ as a function of $(j - i)$, and fit the function to the inverse of our measurements. Figures 4(a) and (b) show the average and maximum stalls we observed over different $(j - i)$'s, which can be fitted by the linear and Sigmoid functions (notice the jump around $(j - i) = 200$) respectively.[7] We choose the linear fitting in our prototype due to its simplicity, and leave the latter our future work.

4.2 Modeling Principles

[7]Note there are some peaks at large distance. In each of these cases, v_i is a (relatively) long transaction and should excluded from the fitting. Also, there are two levels in Figure 4(b) when the distance is above 200. These are contributed by the remote and local pushes respectively. Since we care only about the remote pushes, the Sigmoid function should be fitted to the higher level.

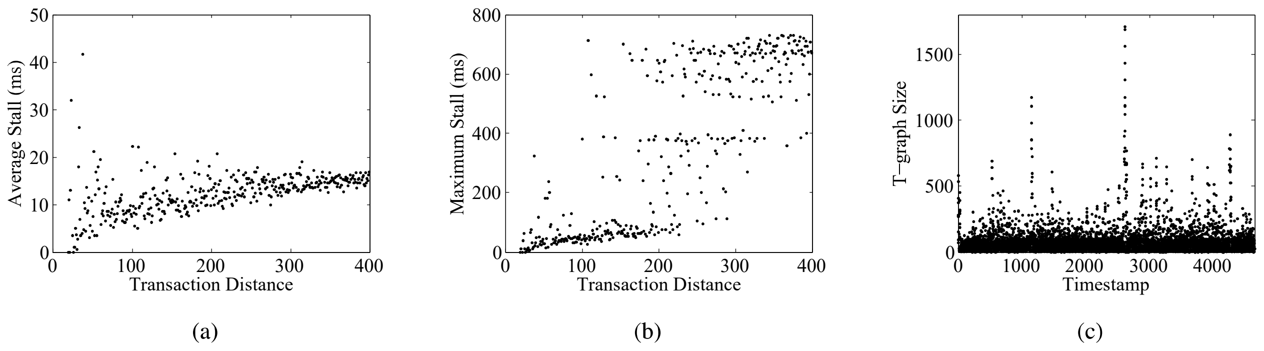


Figure 4: Measurements of a system over 9 Amazon EC2 machines, giving throughput 350,000 txn/min under a microbenchmark. (a) Average stall. (b) Maximum stall. (c) T-graph size.

We can model a given set of transactions into different T-graphs. For example, in Figure 3(a), $T5$ can either read B from $T1$ or $T2$. T-Part chooses to create an edge from $T1$ to $T5$ based on the *reading-from-the-earliest* principle, which prefers the source node to be the earliest transaction that sees the version of data required by the destination node. The intuition behind this is that, if the edge becomes a cross-partition edge after partitioning, then the source node can push the object as early as possible to prevent the stall at destination.

T-Part also employs the *writing-back-the-latest* principle when creating the storage-write edges, which requires only the latest node writing an object to have an storage-write edge. For example, in Figure 3(a), despite that all transactions $T2$, $T3$, $T5$ and $T6$ update the object C, only $T6$ needs to write it back to the storage. This avoids unnecessary storage-access edges that could mislead T-Part to bad partitions, and does not damage the recoverability of transactions as long as their operations and commit decisions are logged.

4.3 Plan Optimization

After generating a push plan and before sending it to the executor, the scheduler can optimize the plan by eliminating the cross-partition edges if local reads are possible. For example, in Figure 3(b), we can eliminate a cross-partition edge from $T1$ to $T5$ by creating another local edge from from $T2$ to $T5$. Plan optimization further reduces the synchronization between machines. Note that each machine only needs to optimize the plans for its owning partition.

5. IMPLEMENTATION

In this section, we briefly discuss some of the key engineering challenges and design decisions we made when building T-Part.

5.1 Real-Time Partitioning

As discussed in Section 3.2, the problem of partitioning T-graph differs from conventional balanced graph partitioning in that the former has the disconnectivity constraints prohibiting the sink nodes from going to the same partition.

To partition the T-graph, our initial direction is to reduce the problem to the conventional one, then apply the off-the-shelf graph partitioning software like METIS [16]. We first introduce a virtual node, called the *pin node*, for each sink node and connect them using a virtual edge, called the *tie edge*. Then, by giving sufficiently large weights to all the tie edges, we can ensure that each pair of the sink and pin nodes will go to the same partition. Furthermore, by giving sufficiently large weights to the pin nodes we can ensure that two pins never go to the same partition, neither do the sinks.

However, this approach has some shortcomings. First, the large pin weights dilute the weights of normal nodes, so we may not find very balanced partitions. Second, whenever new nodes arrive, we need to repartition the T-graph from scratch, which is not preferred in a large-scale system.

We therefore take another direction in extending the *streaming graph partitioning* algorithms [26]. These algorithms assume that the nodes enter a graph as a stream, and build partitions incrementally. We extend a greedy algorithm (called "weighted deterministic greedy" in [26]) that is shown to work well for various types of graphs. Our partitioning algorithm works as follows.

ALGORITHM 1 (T-PART PARTITIONING). *First, assign a sink node to each partition. Upon arrival of a node v, calculate the total weight of edges between v and each partition. Assign v to the partition having the smallest total weight.*[8] *If there is a tie, then assign v to the partition having the smallest total weight of nodes in that partition. Break tie again by assigning v deterministically to a partition with smaller ID (which could be the ID of the machine).*

The performance comparison of our METIS- and Streaming-based algorithms under our simplified TPC-E workload is given as follows:

	Streaming-based			METIS-based		
#Txn	Time	Cut	Skew	Time	Cut	Skew
100	0.14	474	114	5.1	464	25
1000	1.1	4914	94	18.1	4704	201
10000	12.7	59125	46	185.8	72909	242

The time, cut, and skew denote the time (in milliseconds) required to update partitions after arrival of a batch of nodes, the number of cross-partition edges, and the maximum difference between the loads of machines (in total weight of nodes on a machine) respectively. These metrics are measured against different numbers of unsunk transactions in the system. Normally, the number of unsunk transactions (i.e., the size of T-graph) is under 200, as shown in Figure 4(c). Based on the above facts, the streaming-based algorithm offers both speed and quality advantages in spite of its simplicity.

5.2 Cache Management

As discussed in Section 3.4, each executor in T-Part implements a key-value cache area to store the hot data and to prevent repeated

[8]The total weight of a partition can be normalized by the capacity of the corresponding machine. Our deployment consists of homogeneous machines. So we do not perform the normalization.

storage access. We now discuss how T-Part manages the cache area to keep its minimum size small and, when necessary, to trade more cache space for improved performance.

The value of an cache entry is an object value. Each cache access in a push plan needs to specify a key. For example, the push plan for machine holding $S2$ in Section 3.4 can be formally written as:

$T1$: Write cache: <A, $T1$, $T4$>;
Push to $S1$: <B, $T1$, $T2$>, <B, $T1$, $T5$>.
$T4$: Read cache: <A, $T1$, $T4$>;
Write cache: <A, $Sink1$>; storage: E.

Each forward-push edge involves entries with keys of the form: <*obj key, source txn, destination txn*>, a triple consisting of the primary key of the object, the order of source transaction who writes the object, and the order of destination transaction who will read the object. By the definition of its key, each entry of this kind will be read by only the destination transaction. So, after reading an object from the cache area, the destination transaction can invalidate the enclosing entry immediately.

The cache-access edges (e.g., the edge from $T4$ to $S2$ carrying A in Figure 3(b)) involve entries with keys of the form: <*obj key, sink number*>, where a sink number p denotes the p-th sinking process. Assuming that the above plan is generated by the first sinking process, $T4$ writes A using the key <A, $Sink1$>. The purpose of the sink number is to ensure that the later transactions to be sunk in the next batch will read the correct version of data. For example, in Figure 3(c), there will be a line:

$T8$: Read cache: <A, $Sink1$>, <B, $Sink1$>

in the plan generated by the next sinking process to instruct $T8$ to read A and B from the batch in the right front. Upon reading the these objects, $T8$ can invalidate the corresponding entries immediately.

Version-based Deterministic CC. Since each transaction stalls whenever the desired entry is not available in the cache area, T-Part does *not* require the conservative 2PL for concurrency control as in Calvin. Instead, the serializability is guaranteed by allowing each transaction to know which specific version (recorded in the *source txn* field of a cache entry for an forward-push edge, for example) of the object it needs to access. To ensure the serializability and recoverability, a transaction can only write to cache if it knows that it will commit eventually.

Sticky Caching. From the above, all essential cache entries are invalidated once they are accessed. This helps keeping a limited cache size. Actually, we observe that, the total size of the essential cache entries on each machine is proportional to the working set of the transactions assigned to that machine.

We can introduce more cache entries to improve the performance of storage access. Consider the plan for $T6$:

$T6$: Read cache: <C, $T5$, $T6$>;
Write storage: C, D.

If a new transaction $T20$ reading C enters the system after $T6$ is sunk, then it needs to read C from the storage, causing unnecessary storage access. We observe that such "immediate storage reads after write" are frequent under some workloads, where there exists locality in data access but the access interval of an object is longer than that of the sinking process. T-Part deals with this problem by creating a *sticky entry* that caches the data being written back for a fixed small amount of time. For example, it first changes the plan for $T6$ to the following:

$T6$: Read cache: <C, $T5$, $T6$>;
Write sticky cache: <C>, <D>;
Write storage: C, D.

Instead of waiting $T6$ to commit or waiting the C from being read form the storage, $T20$ now can obtain C immediately by searching for the stick entry containing C.

5.3 Handling Transaction Aborts

Next, we discuss how T-Part handles transaction aborts while ensuring the correctness of transaction execution. Recall that a deterministic database system eliminates all the cases of system-initiated transaction aborts apart from that the transaction aborts initiated by its own logic [29, 30]. So, each transaction, upon execution, can know whether its net effect will persist in the system or not by simply looking at its own commit/abort decision hard-coded in the transaction logic. Leveraging this, we require each transaction in T-Part to *read the objects it writes*, and, if it decides to abort, push the read data forward. So, whether a transaction commits or not will *not* change the structure of a T-graph (but only the values of the pushed data), and the T-Part schedulers can partition the T-graph without worrying the commit/abort decisions of transaction execution in the future.

5.4 Handling Failure

The properties of deterministic database systems that simplify the task of fault tolerance have been discussed in [30]. First, if machine failure happens, the failover process can be done immediately because a failed machine can be taken over by an active replica. Second, by logging each transaction request, the failed machine can recover by acquiring a snapshot of data from replicas (if they exist) and/or *replaying* the logged requests to get the latest data back. There is no need for REDO-logging the transaction operations. Moreover, various checkpointing methods are supported, such as the Zig-Zag algorithm [7].

T-Part inherits all the above properties from existing deterministic database systems. We only summarize the difference here. Recall from Section 5.2 that all storage access is actually done by the write-back procedures rather than normal transactions. In T-Part, only the operations of write-back procedures need to be UNDO-logged. Normal transactions do not need any log .

When a failed machine decides to replay transactions (during, say, the REDO phase in the recovery procedure), it cannot replay the transactions that were assigned to itself alone, as 1) the partitioning information is lost; and 2) some transactions need to read the pushes from other machines, which already happened. In T-Part, the transaction requests are logged only after they are partitioned, and each machine *logs only those requests that are assigned to itself.* Furthermore, T-Part requires each executor to create a *PUSH-log* upon receiving a push in order to remember the object value. Therefore, each machine in T-Part can replay its transactions locally during the recovery.

6. PERFORMANCE EVALUATION

In this section, we evaluate the performance of T-Part.

Implementation and Systems. We implement T-Part on a distributed database system called ElaSQL [2], a Java implementation of Calvin [30]. Note that the Calvin source code available from GitHub [1] focuses on the distributed transaction handling and simplifies many low-level database operations (e.g., storage access, buffering, latching, logging, indexing, etc.) by using simulations. ElaSQL, on the other hand, employs VanillaDB [5] as its core on each machine. VanillaDB is a single-node, IBM-System-R-like database system for research purpose. Both ElaSQL and VanillaDB have *no* simulated component. So we believe that building T-Part on ElaSQL can give more realistic performance results.

We implemented Zab [14, 22], a well-known simplification of Paxos, as our total ordering protocol in our communication module. In order to avoid the possible overuse of CPU caused by the leader in Zab, we pull the leader out of the database nodes as a standalone node. We compare the default implementation of ElaSQL (which we denote "Calvin") and Calvin with T-Part (denoted as "Calvin+TP") in the following experiments.

6.1 Scalability

We first evaluate the scalability of T-Part by using the well-known TPC-C and TPC-E benchmarks. To make the results comparable to those published in the Calvin paper [30], we run experiments on machines/environment similar to that adopted in the Calvin paper. We rent 46 Amazon EC2 C3.xlarge dedicated instances, each promises 7.5 GB of memory and 14 EC2 Compute Units[9] (4 virtual cores with 3.5 EC2 Compute Units each), and reserve 15 instances to run the clients (named Remote Terminal Emulator (RTE) in the TPC terminology), 30 instances as DB nodes, and one as the Zab leader. Note that, although we strive to create a comparable environment, the actual capability/provision of an EC2 instance changes overtime thus our environment may still be different from that used in the Calvin paper.

6.1.1 TPC-C

The TPC-C benchmark simulates a warehouse management system, and its data are known to be partitionable based on warehouses because each transaction has only 10% probability to access the data in more than one warehouses. While T-Part focuses on the hard-to-partition data, we show that it can also achieve the same level of transaction throughput as Calvin does.

Figure 5(a) shows the system throughput of the New-Order transactions given by T-Part and Calvin over different numbers of machines.[10]As we can see, both Calvin and T-Part can scale out up to 30 machines. This implies that T-Part incurs little overhead on a deterministic database system where data are well-partitioned. It is safe to turn it on even with easy workloads.

Note that our Calvin implementation, without simulated components, verifies the trends reported in the original Calvin paper [30]. However, the absolute values of our throughput readings are generally only one-third of those reported in [30]—the per-core throughput of our implementation and the study is 208 and 625 transactions per second in the above figure, respectively.[11] This give a sense how much the other components, such as index maintenance and file system operations, may impact the performance of a deterministic database system.

6.1.2 TPC-E

TPC-E emulates the behavior of customers, brokers, and markets in a financial brokerage system, and, as compared to TPC-C, has more complicated and long-running transactions, non-uniform data access, and *hard-to-partition data*.

Because there is no well-known best partitioning method for TPC-E, we partition each table horizontally based on the hash value of the primary key of each record. Among all types of TPC-E transactions, we focus on the Trade-Order and Trade-Result transactions, as they are two most representative transaction types. Normally, almost all transactions of TPC-E are distributed transactions. And a record read by a transaction could be on any machine; that is, almost all operations of a TPC-E transaction are remote operations. Moreover, the EGen program provided by TPC generates non-uniform customer ID, thus the data access pattern is skewed. The above characteristics fit well to other types of applications such as the social networking services.

The throughput of T-Part and Calvin given up to 30 nodes are shown in Figure 5(b). We can see that Calvin can only scale out up to 4 machines. When the number of machines go beyond five, Calvin starts to saturate. On the other hand, T-Part is still scalable, and the linear scalability preserves up to 22 machines.[12] The reason for this drastic change in trend is twofold. First, the forward-pushing technique in T-Part significantly reduces the chance of transaction stalls due to unavailable remote data, and mitigates the machine synchronization problem in the presence of workload skewness. We will further verify the effectiveness of forward-pushing in later sections. Another reason, less obvious but still important, is that T-Part does *not* let multiple machines execute the same distributed transaction collaboratively (via peer-pushing, as described in paragraph 2 in Section 2.1). Instead, it assigns each transaction to only one machine, and that machine collects necessary data from either previous transactions or storage on different machines. So, the benefit is that it saves the computing resources from processing the same transaction logic multiple times. In TPC-C, this benefit doesn't make much difference because each transaction is short and the portion of distributed transactions is low. However, in TPC-E where the portion of distributed transactions is high and each distributed transaction accesses data spanning almost *all* machines, Calvin requires almost all machines to participate in every distributed transaction, resulting in huge overhead.

Parameter	Default Value
#Records per Machine	1,000,000
#RTEs (i.e., #client processes)	4000
#Server / #Client Machines	20 / 10
Buffer Size per Server Machine	390 MB
#Records Accessed per Txn.	10
#Remote Records per Distributed Txn.	9
#Write Records per Read-write Txn.	5
Distributed Txn. Rate	1.0
Read-write Txn. Rate	0.5
Skewed Txn. Rate	0.3
Txn. Conflict Rate	1% (10k hot rec.)
Sink Size (T-Part parameter)	100

Table 1: Default parameters of the Microbenchmark.

6.2 Comparison with Data Partitioning / Moving Methods

Here we demonstrate the advantages of T-Part over conventional data partitioning methods in the presence of hard-to-partition workloads. Figure 6 compares the 10-minute-average throughput of T-Part against those of various data partitioning/moving techniques

[9]One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

[10]We focus on the New-Order transactions because it is a major limiting factor of the system scalability in TPC-C. For more discussions, please refer to [29].

[11]Note that the capability of machines we used on AWS differs from that of machines used in [30]. The machines they used are of the C1.xlarge type while we were using the C3.xlarge machines. According to the documentation, each C1.xlarge machine consists of 8 virtual cores and a C3.xlarge has 4. So, if we consider the per-core throughput of TPC-C on 10 nodes, the original Calvin and our implementation give 625 txs/sec and 208 txs/sec respectively.

[12]Notice that the performance of T-Part fluctuates when the number of machines is large. This is because that not all EC2 instances yield equivalent performance, and this issue has been reported in the Calvin paper [30] as well as other industrial articles [4]. Given hard-to-partition workloads like TPC-E, any slowdown of a single machine could have an impact on the overall system performance. And, as the number of machines increases, this impact can be more significant, as we can see in the figure.

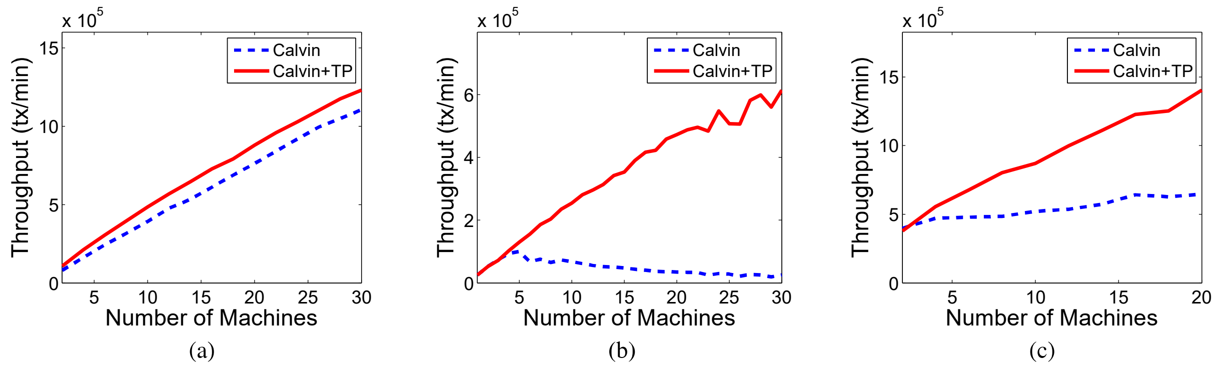


Figure 5: System throughput with the (a) TPC-C benchmark (New-Order transactions); (b) TPC-E benchmark; and (c) Microbenchmark (default parameters).

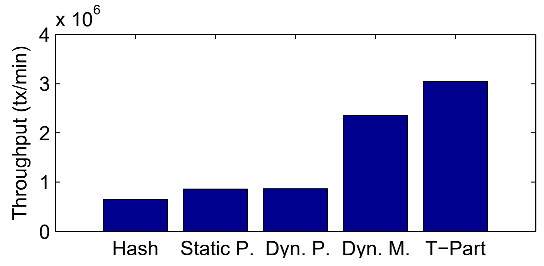


Figure 6: System throughput with (a) static hash-based data partitioning, (b) static graph-based data partitioning (Schism [9]), (c) dynamic graph-based data partitioning, (d) dynamic data movement (G-Store [10]), and (e) T-Part.

(other than the basic hash-based data partitioning) under the TPC-E workloads. The experiments are ran on 20 machines of our own. Each machine is equipped with an Intel Core i5-4460 3.2 GHz CPU and 8 GB RAM and running CentOS 7.

Static Graph-based Data Partitioning (Schism). In this experiment, we follow Schism [9] to model the trace of 300K transactions into a graph, then employ METIS [16] (a multi-way graph partitioning algorithm) to partition the graph and obtain data partitions. As shown in Figure 6(b), Schism gives about 60% performance boost as compared to the basic hash-based data partitioning.

Dynamic Graph-based Data Partitioning. One obvious way to further improve the quality of data partitions is to run Schism periodically. In this experiment, we re-generate the data partitions every 3 minutes. Note that we pre-generate all the data partitions offline, so the reported throughout will *not* take into account the data migration cost and be optimistic. Even so, the resultant throughput does not change much, as shown in Figure 6(c), because TPC-E does not render much temporal locality.

Dynamic Data Movement (G-Store). Another approach to deal with hard-to-partition data is to use the dynamic data movement approach proposed by G-Store [10], which transfers all relevant records to a single machine and execute the corresponding transactional operations on demand. However, this approach requires the clients to specify explicitly a certain group of data that is going to be accessed together in the near future *before* issuing the corresponding transactions, which is not possible in TPC-E. In fact, G-Store is proposed only for the NoSQL database systems. Therefore, we can only simulate this dynamic data movement approach by moving the read-sets and write-sets of a group of transactions to a single machine, executing the transactions, and then moving the records back to their original machines. To determine

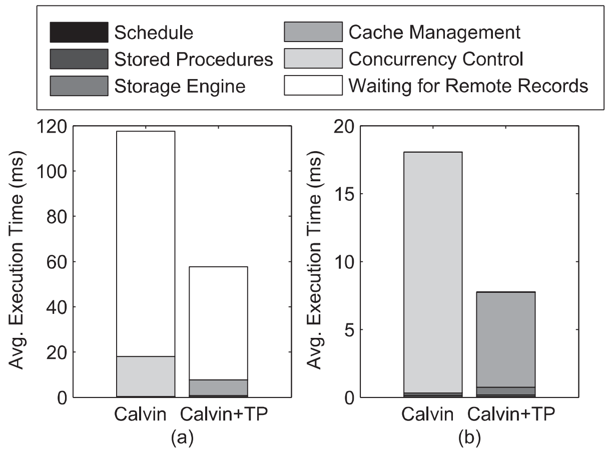


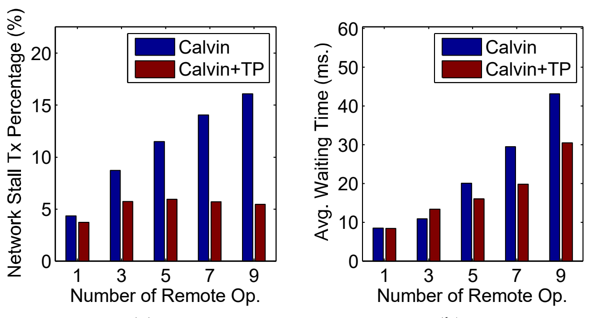
Figure 7: Breakdowns of the average execution time (a) with and (b) without transaction stalls due to the unavailable remote records.

the transaction group without users' specifications, we make it as a transaction partition in a T-graph. In effect, this simplification reduces to T-Part with the sink size 1, and, as shown in Figure 6(d), gives 270% performance boost as compared to the dynamic graph-based data partitioning. This demonstrates the importance of joint optimization of transaction execution and data movement.

T-Part. T-Part, with sink size larger than 1, additionally takes into account the locality of data access between subsequent transactions and move transactions around to avoid unnecessary storage reads/writes. This gives yet another 30% boost in system throughput, as shown in Figure 6(e), and demonstrates the effectiveness of T-Part in replacing the storage reads/writes with the forward-pushes.

6.3 Microbenchmark

To further investigate the factors that affect the T-Part performance, we implemented a Microbenchmark that has several adjustable parameters whose default values are summarized in Table 1. This Microbenchmark consists of one table that are horizontally and evenly partitioned across different machines. The size of each record is 164 bytes. We split each data partition into the *hot* set and *cold* set. We distinguish transactions from different aspects: 1) read-only vs. read-write; 2) local vs. distributed; and 3) non-skewed vs. skewed. A read-only transactions reads a constant 10 records; while a read-write transaction, after reading 10 records,



(a) (b)

Figure 9: The impact of the number of remote operations under a TPC-C-like workload. (a) Network-stalled transaction percentage. (b) Average waiting time.

randomly writes back 5 of them back to the storage. A transaction is deemed distributed if any of the accessed records is located on a remote machine. A skewed transaction has 50% probability of accessing remote records on machines that are numbered in the first one-fifth. We control the conflict rate of transactions by varying the size of the hot sets—the 10 records read by each transaction consists of one record in a hot set (called hot record) and 9 in a cold set. So, the smaller the hot sets, the higher transaction conflict rate.

Like the previous experiments, these experiments are also ran on the same machine described in Section 6.2. We carefully select the default parameter values so that we can re-produce the same trends shown in Figure 5(b) using the Microbenchmark. The throughput of T-Part and Calvin over the default Microbenchmark is shown in Figure 5(c). Looking into the parameters, we see high distributed transaction and skewed transaction rates. This is consistent with our observations in TPC-E, where almost all transactions are distributed and the skewness are high due to the non-uniform customer ID generator.

6.3.1 *Execution Time Breakdown*

Running Microbenchmark on our own machines allows us to break down the behavior of T-Part and Calvin in detail. We inject the probing code to the codebases of T-Part and Calvin to record the execution time of every major component. The results are shown in Figure 7. Note that we do not include the time spent in the total-ordering protocol Zab because it remains around 150 milliseconds in both Calvin and Calvin+TP.

We can see that, the main cause of the transaction delay is the time spent in waiting for remote records. And T-Part can reduce about 50% of this cost thanks to the partitioning of T-graph and the forward-pushing technique. Note that the delay of T-graph analysis and partitioning (covered in "Schedule" component) is almost neglectable (less than 0.05% of the overall delay). These together justify the effectiveness and efficiency of our run-time partitioning algorithm, as described in Section 5.1.

Note that the role of concurrency control is replaced by the (multi-version) cache management layer in T-Part, as described in Section 5.2. Comparing the cost of the cache management in T-Part against the concurrency control in Calvin, we can see that our multi-version design yields better implementation efficiency than conservative locking. Also note that, since T-Part spends less time in communication, the storage engine is better utilized thus contributing more to the delay.

6.3.2 *Distributed Transactions*

Next, we investigate the impact of distributed transactions on system throughout. We conduct two experiments: one with varying distributed transaction rate, and another with distributed transactions accessing a varying number of remote records. The results are shown in Figures 8(a) and (b) respectively. As we can see, T-Part leads to 60%~120% speedup when either the distributed transaction rate or the remote operation number is high. The improvement becomes significant when the distributed transaction rate is above 0.2 and when there are more than 5 remote records in a transaction. This justifies the effectiveness of our 1) T-graph partitioning, so the machine workloads be can indeed be balanced and the (cross machine) storage reads/writes can be optimized; and 2) forward-pushing of records, so the chance of transaction stall due to unavailable remote records can be reduced. Recall that "bad" data partitioning is a major reason of a high distributed transaction rate and a high number of remote operations. T-Part mitigates the synchronization problem of existing deterministic database systems and make them applicable to workloads where data are hard to partition.

Note that, when all transactions are local, the throughput of Calvin is little higher than T-Part. This is because 1) in the absence of distributed transactions, the cache management in T-Part becomes relatively costly; and 2) with default workload skewness (see Table 1), the schedulers in T-Part attempts to balance the load between machines (see Algorithm 1) and thus may create a small number of unnecessary distributed transactions. However, the gap is very small and we believe that it can be further reduced by optimizing the T-Part source code about cache management, or by further extending our real-time partitioning algorithm (to be described in Section 6.3.6).

6.3.3 *Network Stall*

To further investigate whether T-Part decreases the chance of network stall that happening when a distributed transaction needs a record from a remote machine. We vary the number of remote operations under a TPC-C-like workload and a TPC-E-like workload separately and record how many transactions need to wait for remote records and the average waiting time of them. We call these transactions the *network-stalled transactions* here. Followings are our results.

TPC-C-like Workloads. We create such workloads by setting the skewed transaction rate to 0.0 and the remote transaction rate to 0.1. The results are reported in Figure 9. As the number of remote operations increases, one would expect more network-stalled transactions since there are more remote records to wait. This is the case for Calvin, but *not* for Calvin+TP, as shown in Figure 9(a). As the number of remote operations increases, the T-Part gradually shifts its focus from balancing the machine workloads to minimizing the cross-partition edges when partitioning the T-graphs. So the number of network-stalled transactions decreases slightly. This validates the benefits of T-graph partitioning. In addition, Figure 9(b) shows that Calvin+TP is able to reduce more than 30% of the average waiting time when the number of remote records is high. This validates the effectiveness of the forward-pushing technique.

TPC-E-like Workloads. Now we step back to the default parameters of the Microbenchmark, which simulates the TPC-E. The results are shown in Figure 10. The percentage of the network-stalled transactions doesn't change significantly in Calvin. We believe this is because that the transaction throughput of Calvin has already been saturated (as shown in 5(b)). On the other hand, Calvin+TP gives a decreasing number of network-stalled transactions again, validating the benefits of T-graph partitioning in this case. And, as shown in Figure 10(b),[13] the forward-pushing tech-

[13]Note that the delay reported here is higher than that reported in Figure 7, because Figure 7 takes into account all transactions while we consider only the network-stalled transactions here.

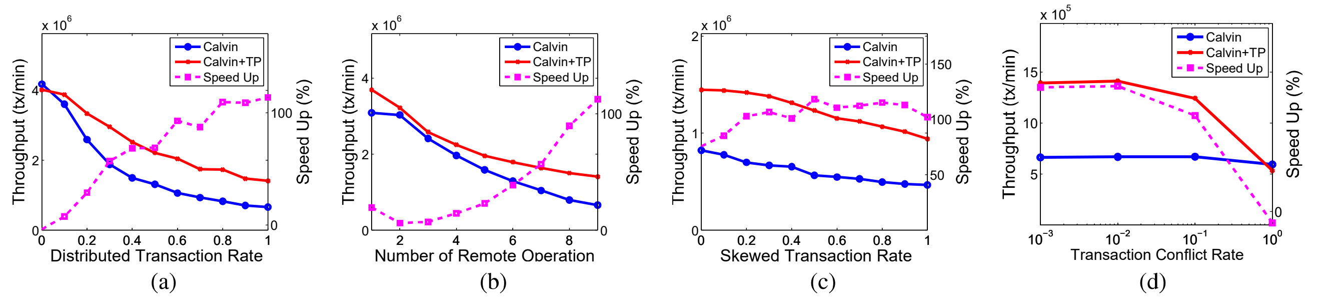


Figure 8: (a) Impact of distributed transaction ratio. (b) Impact of the number of remote operations in a distributed transaction. (c) Impact of skewness. (d) Impact of transaction conflict rate.

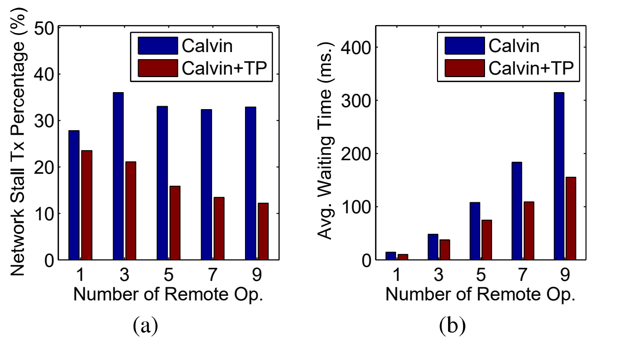


Figure 10: The impact of the number of remote operations under a TPC-E-like workload. (a) Network-stalled transaction percentage. (b) Average waiting time.

nique adopted by Calvin+TP is able to reduce more than 50% of the average waiting time when the number of remote records is high.

6.3.4 Workload Skewness

Another negative impact of "bad" partitioning is the imbalanced workloads across machines, as most records accessed by a transaction may reside on few particular machines. Theses machines would fall behind easily and make the synchronization problem worse. We study the impact of workload skewness on system throughput and conduct experiments with varying skew transaction rate. The results are shown in Figure 8(c). Again, T-Part significantly outperforms Calvin when the skewness is high. This justifies the effectiveness of Algorithm 1 on balancing machine loads.

6.3.5 Transaction Conflict Rate

Next, we study how transaction conflict rate affects the system throughput. We conduct experiments with varying hot set size (recall that each transaction access one hot record so we can change the size of hot set to control the transaction conflict rate).[14]

The results are given in Figure 8(d). While the main paper [30] of Cavlin reports that the transaction conflict rate has significant impact on performance under TPC-C-like workloads, we do *not* observe similar results here under TPC-E-like workloads. We believe this is because that Calvin has already been saturated by the communication overhead due to distributed transactions and numerous remote operations. On the other hand, the Calvin+TP performance can be affected by a high transaction conflict rate. This

[14]The actual transaction conflict rate may be slightly higher, because two transactions may conflict with each other when accessing the same cold record. However, due to the large amount of cold records, the chance may be too low to be noticeable.

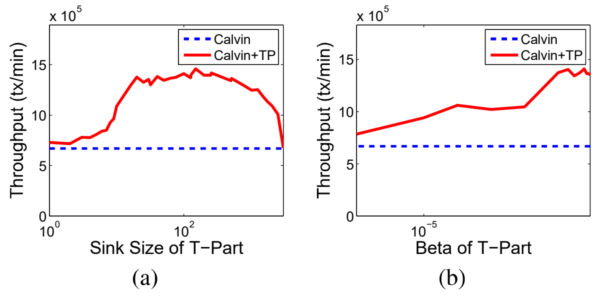


Figure 11: The effects of T-Part parameters: (a) Sink Size; (b) β.

is because that, at a high conflict rate, the T-graph becomes very dense and hard to partition. However, with mild conflict rates T-Part is still able to keep the T-graph partitioning beneficial, by reducing the cross-machine links.

6.3.6 Parameter Tuning

The sink size in T-Part is an adjustable parameter. We study the impact of this parameter by conducting experiments with varying sink size. The results are shown in Figure 11(a). We can see that either a too large or too small sink size has negative impact on the system throughput. When the sink size is too large, the T-graph becomes very large thus its partitioning overhead increases; on the other hand, when the sink size is too small, the T-graph may be too small to have enough forward-push edges to improve the efficiency of transaction executors. Note that except with extreme values, the sink size does not impact the system throughput too much. One can easily pick a value around 100 to have good performance.

Recall that Algorithm 1's goal is to minimize the cross-machine edges *and* to balance the workloads between machines. Given that most of the transaction execution time is spent in communication (see Figure 7), it may seem that minimizing the cross-machine edges is more important than balancing machine workloads. However, we observe that the latter is vital to good performance in the presence of hard-to-partition data. We extend Algorithm 1 by dispatching each transaction to the partition having the maximum weight:

$$(\text{total edge weight}) + \beta(\text{total node weight}),$$

where the total-edge-weight denotes the sum of weights of edges between the transaction and the partition, and the total-node-weight denotes the sum of weights of nodes/transactions that has been assigned to the partition previously. The smaller the β, the more T-Part prefer minimizing the cross-machine edges than balancing machine workloads. We conduct experiments with varying β and the results are shown in Figure 11(b). As we can see, the throughput is high only β is sufficiently large, justifying the importance of load balancing.

7. RELATED WORK

Our partitioning algorithms are based on the streaming graph partitioning algorithms [26].

7.1 Transaction Execution Altering

A main contribution of T-Part is the design of transaction execution framework that allows the forward-pushing and reduces machine synchronization in a deterministic database system. There are other studies that alter the execution of transactions to improve the system performance. NuoDB [3] can dispatch/route transactions to different machines on-the-fly. However, T-Part leverages the determinism so to allow more sophisticated optimization—the forward-pushing. G-Store [10] executes a group of transactions that access the same data (called a *key group*) on the same machine and move data around beforehand. It assumes that the data can be perfectly partitioned into key groups and each client can tell exactly which key group to access before issuing a transaction. T-Part, on the other hand, does not have such assumptions and finds the optimal "transaction groups" by partitioning the T-graph. Studies [12, 13] explicitly create transaction dependency graphs for multi-versioned deterministic database systems in order to increase data access locality in multi-core settings; while T-Part uses dependency graphs for increasing data access locality for multi-machine settings.

7.2 Pre-Fetching

The T-graph, having data-access links, is related to the pre-fetching techniques. In the pre-fetching techniques, the storage system pre-fetches a batch of blocks upon detecting some data access pattern. Smith [24] develop an algorithm using the sequentiality of data access. Nightingale et al. [18] implement a speculative execution for file systems so to increase the I/O throughput by masking the I/O latency. Soundararajan et al. [25] propose QuickMine, which improves pre-fetching by capturing application contexts, such as a transaction or query. There are many other work on improving the effectiveness of pre-fetching under particular settings. For example, PROMISE [23] improves pre-fetching by predicting the query behavior in OLAP systems, and Scout [28] improves pre-fetching by capturing the latent structure behind spatial data.

T-Part differs from most of the above work in that it decides not only the policy for pre-fetching (realized by the storage-read edges in a T-graph), but also 1) the relay of modified data (realized by the forward-push edges) and 2) the writeback (realized by the storage-write edges). This jointly optimizes the management of data in memory, in storage, and across machines (e.g., not to write back certain data in one machine to avoid repeated fetching *and* to reduce remote wait).

7.3 Content-Aware Request Dispatching

To partition the T-graph, T-Part employs the streaming-based partitioning algorithms that dispatches the nodes to partitions upon their arrival, therefore is related to the work on content-aware request dispatching. Studies, LARD [19] and HACC [35], propose content-aware request dispatching protocols in the middleware and dispatcher layers for clustered servers. In replicated database systems with lazy replication, Amza et al. [6] propose to route the read request to the replica with the up-to-date data and with the least load. Similar to T-Part, it totally orders the requests and processes the conflict transactions following this order. Zuikeviciute et al. [36] consider dispatching the update transaction to the machine which has processed some conflicting transaction in order to reduce the abort rate. Yamamuro et al. [32] explore the working set information of transaction to dispatch the requests. Tashkent+ [11] uses the query engine to estimate the working sets of each transaction type, and then assigns a set of transaction types (forming a transaction group) to replicas such that the memory is large enough to serve the composite working set. This work shares with T-Part a similar idea of partitioning the transaction workload (rather than partitioning the data). However, the partition unit in Tashkent+ is the coarse-grained transaction group.

T-Part differs from all of the above work in that it does *not* finalize the dispatch decisions on previous requests until it sink a batch of requests. This allows T-Part to change the dispatch decisions on previous requests to get better overall dispatch results.

7.4 Graph-based Data Partitioning

The T-graph modeling is related to the graph-based data partitioning techniques. Schism [9] models data objects and their common access in the past transactions as the nodes and edges of a graph respectively, and finds the partitions of this graph such that 1) nodes spread evenly among partitions, so that the machine loads are balanced; and 2) edges across partitions are the fewest, so the communication cost is minimized. SWORD [21] , an extension of Schism, supports dynamic data re-partitioning by swapping pairs of data in different partitions and chooses different replication strategies based on the data update frequency. It also proposes grouping the nodes to increase the efficiency of partitioning large graphs. Yang et al. [33] propose a two-level partitioning technique for large graphs, which uses the static primary and dynamic secondary partitions together to fit the changing workload. It replicates the hotspot in different graph partitions to decrease the cross-partition queries.

The design goals of T-Part are very different from the graph-based data partitioning. First, it deals with a much smaller graph but requires the graph to be partitioned in real-time (and continuously). Second, T-Part does not change the data placement, but how data are accessed and where transactions are executed.

8. CONCLUSIONS

This paper presents T-Part, a transaction execution engine for the deterministic database systems. T-Part partitions pending transactions to keep the balanced loads on machines while minimizing the communication cost between them. By following the push plan, each transaction can push forward the data to later transactions immediately after its execution. Our experiments demonstrate that T-Part reduces machine synchronization and makes the system robust to the unforeseeable workloads or workloads whose data are hard to partition.

There are some points in T-Part that deserves further investigation. First, as discussed in Section 4.1, the maximum stall due to transaction i that a transaction j will encounter depends on $(j - i)$ and can be approximated by a Sigmoid function. It remains unclear how the performance of T-Part will be affected if we set the edge weights using this model. Second, although not shown in Figure 2, data partitions may be replicated within a data center to survive from machine failure and/or to avoid hot spots due to reads. The replication of transactions in a T-graph may be further extended to work with the data replicas inside a data center. The above matters are our future inquiry.

9. ACKNOWLEDGMENT

We thank Mr. Ching Tsai (ctsai@netdb.cs.nthu.edu.tw) for helping setup environments and run experiments.

10. REFERENCES

[1] Calvin source code. https://github.com/yaledb/calvin.

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

[3] Nuodb. http://www.nuodb.com.

[4] The top 5 aws ec2 performance problems. http://www.datadoghq.com/wp-content/uploads/2015/06/Top-5-AWS-Ec2-Performance-Problems-Guide-Ebook.pdf.

[5] Vanilladb. http://www.vanilladb.org.

[6] C. Amza, A. L. Cox, and W. Zwaenepoel. Conflict-aware scheduling for dynamic content applications. In *USENIX Symposium on Internet Technologies and Systems*, volume 21, page 22, 2003.

[7] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. of SIGMOD*, pages 265–276. ACM, 2011.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *Proc. of PODC*, volume 7, 2007.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. of the VLDB Endowment*, 3(1-2):48–57, 2010.

[10] S. Das, D. Agrawal, and A. El Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 163–174, New York, NY, USA, 2010. ACM.

[11] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. *ACM SIGOPS Operating Systems Review*, 41(3):399–412, 2007.

[12] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. of the VLDB Endowment*, 8(11):1190–1201, 2015.

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

[14] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proc. of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment*, 1(2):1496–1499, 2008.

[16] 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, 1999.

[17] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. of VLDB*, pages 134–143, 2000.

[18] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 24(4):361–392, 2006.

[19] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *ACM Sigplan Notices*, volume 33, pages 205–216. ACM, 1998.

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

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

[22] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *Proc. of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 2. ACM, 2008.

[23] C. Sapia. Promise: Predicting query behavior to enable predictive caching strategies for olap systems. In *Proc. of Data Warehousing and Knowledge Discovery*, pages 224–233. Springer, 2000.

[24] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):223–247, 1978.

[25] G. Soundararajan, M. Mihailescu, and C. Amza. Context-aware prefetching at the storage server. In *Proc. of the USENIX Annual Technical Conference*, pages 377–390, 2008.

[26] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proc. of SIGKDD*, pages 1222–1230. ACM, 2012.

[27] 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 the VLDB Endowment*, pages 1150–1160. VLDB Endowment, 2007.

[28] F. Tauheed, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. Scout: Prefetching for latent structure following queries. *Proc. of the VLDB Endowment*, 5(11):1531–1542, 2012.

[29] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. of the VLDB Endowment*, 3(1-2):70–80, 2010.

[30] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proc. of SIGMOD*, pages 1–12. ACM, 2012.

[31] E.-J. van Baaren. Wikibench: A distributed, wikipedia based web application benchmark. *Master's thesis, VU University Amsterdam*, 2009.

[32] T. Yamamuro, Y. Suga, N. Kotani, T. Hitaka, and M. Yamamuro. Buffer cache de-duplication for query dispatch in replicated databases. In *Database Systems for Advanced Applications*, pages 352–366. Springer, 2011.

[33] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proc. of SIGMOD*, pages 517–528. ACM, 2012.

[34] G.-W. You, S.-W. Hwang, and N. Jain. Ursa: Scalable load and power management in cloud storage systems. *ACM Transactions on Storage*, 9(1):1, 2013.

[35] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. Hacc: An architecture for cluster-based web servers. In *Proc. of the USENIX Windows NT Symposium*, volume 3, pages 16–16. USENIX Association, 1999.

[36] V. Zuikeviciute and F. Pedone. Conflict-aware load-balancing techniques for database replication. In *Proc. of ACM SAC*, pages 2169–2173. ACM, 2008.