Logical Timestamps in Distributed Transaction Processing Systems

by

Yu Xia

B.Eng., Tsinghua University (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2018

Signature redacted

Certified by .....
Srinivas Devadas
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Logical Timestamps in Distributed Transaction Processing Systems

by

Yu Xia

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

With the increasing demand on large scale in-memory databases, researchers have applied sharding techniques to split the data load into multiple disjoint data servers to make it possible to run the system on commodity machines and avoid the need of a single large DRAM which is believed to be inefficient. A typical transaction on such systems from time to time will need to query data from another server.

Distributed transactions are such transactions with remote data access. They usually suffer from high network latency (compared to the internal overhead) during data operations on remote data servers, and therefore lengthen the entire transaction executiont time. This increases the probability of conflicting with other transactions, causing high abort rates. This, in turn, causes poor performance.

In this work, we constructed **Sundial**, a distributed concurrency control algorithm that applies logical timestamps seaminglessly with a cache protocol, and works in a hybrid fashion where an optimistic approach is combined with lock-based schemes.

Sundial tackles the inefficiency problem in two ways.

Firstly, Sundial decides the order of transactions on the fly. Transactions get their commit timestamp according to their data access traces. Each data item in the database has logical leases maintained by the system. A lease corresponds to a version of the item. At any logical time point, only a single transaction holds the 'lease' for any particular data item. Therefore, lease holders do not have to worry about someone else writing to the item because in the logical timeline, the data writer needs to acquire a new lease which is disjoint from the holder's. This lease information is used to calculate the logical commit time for transactions.

Secondly, Sundial has a novel caching scheme that works together with logical leases. The scheme allows the local data server to automatically cache data from the remote server while preserving data coherence.

We benchmarked Sundial along with state-of-the-art distributed transactional concurrency control protocols. On YCSB, Sundial outperforms the second best protocol by 57% under high data access contention. On TPC-C, Sundial has a 34% improvement over the state-of-the-art candidate. Our caching scheme has performance gain comparable with hand-optimized data replication. With high access skew, it speeds the workload by up to 4.6×.

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor of Electrical Engineering and Computer Science

Acknowledgments

This work was supported (in part) by the U.S. National Science Foundation (CCF-1438955).

I would like to thank my thesis advisor Prof. Srinivas Devadas, whose help is always available whenever needed. I am also grateful to my colleague Xiangyao Yu, who proposed the main idea. In our original paper, Prof. Andrew Pavlo, Prof. Daniel Sanchez, and Larry Rudolph contributed plenty of valuable advice and guidance. Besides these, all the help from my parents and my friends are also much appreciated.

Chapter 1

Introduction

The transaction programming model is friendly for users who work on large parallel and concurrent systems like databases. Due to its useful properties including *atomicity, consistency, isolation, and durability*, the programmers do not have to spend significant time worrying much about data racing and conflict detections, task scheduling, and most importantly the complexity of multi-threaded program design. There has been an ongoing research effort on optimizing On-Line Transactional Processing (OLTP) database systems.

As the scale of data increases, the computational workload and the storage demand often go beyond the capability of a typical single server. When this happens, people often turn to a distributed database management system, where the data storage is often split into multiple disjoint shards, named *partitions*. Each partition is held on an independent server, sharing nothing but communication channels with other servers. If fortunately a transaction only access data locally, it is processed without any difference from a typical local transaction. However, chances are that some transactions visit multiple servers for data access, leading to a downgraded performance. The reasons fall into two categories. On one hand, remote data access itself will bring high network latency, making the transaction execution much slower. On the other hand, longer execution time naturally makes it more likely to conflict with another transaction, causing more aborts and re-trials. Thus, the performance for distributed transactions are often not up to par.

Recent related research that aims to ameliorate the performance of distributed concurrency control algorithms focuses mainly on two aspects, designing a better protocol, or

optimizing the hardware. New protocols often improve the system overhead in synchronizing among transactions [47, 59, 31, 48], and therefore the abort rate drops. However, we have observed that they still have limited performance because of higher protocol complexity. Hardware optimizations include utilizing special techniques like *Remote Direct Memory Access* that enables low network latency on a remote data access. However, such specialized hardware tends to increase the overall cost. In our system, we want to achieve good performance on average commodity servers.

For caching schemes, people have proposed to replicate read-intensive data items across multiple servers [20], so that some transactions do not have to perform remote data fetching and can be processed as local transactions. However, such replication usually requires human intuition on hotspot data items or systematically profiling the whole database workload. Ideally, the database system should automatically and dynamically provide a caching scheme to replicate this data without human intervention.

To fulfill these two ideas, we designed **Sundial** [67], an in-memory distributed concurrency control protocol that outperforms existing approaches in standard benchmarks. The **Sundial** project started before I joined the group. In this work, Xiangyao Yu primarily did significant work including coming up with the core idea extended from his previous work [66] and maintaining the codebase from [4]. My contribution in this work mainly lies in

- Proposing potential optimizations to improve the performance of the system (will be discussed in Chapter 5) as well as related implementations and experiments;
- Conducting a series of evaluation experiments of the main system over well-known benchmarks;
- Designing a variant of YCSB benchmark to capture the behaviors in social networks which reveals the non-trivial performance gain from the caching scheme of Sundial;
- Comparing Sundial with the most closely related baseline MaaT [47]. This includes reproduction of the codebase of MaaT, extending the MaaT system into the multi-server-multi-threaded model to ensure fairness (since Sundial is designed to run in multiple threads on every server), as well as other discussion, experiments and analysis related to MaaT;

- Adding the idealized MVCC as an imaginary baseline to indicate the performance bounds of a series of MVCC systems in the same environment.
- And other efforts in maintaining the codebase.

To reduce the problem of long network latency, Sundial naturally integrates with a data caching in a server's local memory without sacrificing serializability, and decides whether it should use cached data based on access patterns and heuristics.

To reduce the extra overhead of coordinating transactions, Sundial combines lock-based and optimistic concurrency control protocols. It reduces unnecessary aborts by reordering transactions dynamically that have read-write conflicts.

The technique we used in Sundial is *logical leases*, which, with a little more information stored per data item, enable the database system to dynamically re-arrange the logical order among transactions based on their data access patterns, while enforcing serializability.

For each data item, we attach a logical lease to it. A logical lease consists of two logical timestamps marking the starting point and ending point of the lease. Only within this range is the tuple valid. The database system finds a commit timestamp for the transaction such that it overlaps with all the logical leases of the tuples of the specific versions accessed by the transaction. It has been shown in prior work that logical leases are effective in improving overall performance and concurrency for both hardware cache coherence [65] and concurrency control protocols on a single machine with multicore processors [66]. As far as we know, Sundial is the first database system to integrate logical leases with a natural caching scheme working seamlessly with the concurrency control protocol in a distributed and shared-nothing setting.

We implemented Sundial and evaluated it with standard benchmarks along with three state-of-the-art candidates as baselines: Google F1 [53], MaaT [47], and typical two-phase locking with the Wait-Die scheme [12, 34]. We used two YCSB and TPC-C workloads with different configurations. Sundial achieves up to 57% higher throughput and 41% lower latency than the best baselines. We also evaluated the caching scheme and it improved the throughput by up to 4.6× for read-intensive workloads with skew.

Chapter 2

Background and Related Work

Transactional processing systems require that the concurrency controls maintain the *atomicity* and *isolation* of transactions. Atomicity makes sure that either the full transaction is applied or none of the changes made is applied to the database. Isolation specifies when a transaction's changes are publicly available to other transactions. There are different levels of isolations. In Sundial, we fulfill *serializability*. Conflicting transactions generate the same effects as if they are executed sequentially.

While serializability provides a strong guarantee and easy-to-understand programming features for programmers, procotols with such property usually come with large overhead in a database system. This has been shown in previous studies of concurrency control in both the multi-core processor setting [64] and distributed setting [34]. This observation is even more obvious in a distributed database system because of high network roundtrip latencies between geographically scattered database servers.

Existing concurrency control protocols [12] mainly follow two paradigms. One is two-phase locking (2PL for short) and the other one is timestamp reordering (T/O). People often consider 2PL [12, 30] schemes as pessimistic protocols as they by default assume more frequent conflicts and aborts. So they act in a conservative way that a transaction accesses a tuple only after the lock of that tuple is acquired with corresponding permission (e.g., read or write). Timestamp reordering decides the commit order of transactions with logical timestamps on the fly. A well-known category of concurrency control schemes is called Optimistic Concurrency Control, where the system executes transactions with an optimistic

assumption that no conflict might occur and performs conflict checking and resolution (usually just direct aborts) after execution. OCC allows the transactions to execute in a non-blocking manner, but it is expected to bring more aborts and more re-trials. Multi-Version Concurrency Control (MVCC) is another typical class of concurrency control algorithms, where the system dynamically maintains and keeps track of more than one version of data to reduce conflicts. Both OCC and MVCC are special cases of T/O. Sundial combines the pros of OCC and 2PL by applying different techniques for different types of conflicts (namely, write-after-write conflicts and read-after-write conflicts). Sundial is also MVCC-aware. It contains some optimizations with the spirit of MVCC, which will be discussed in details in Chapter 5. Sundial further prevents some of the unnecessary aborts due to read-after-write conflicts via utilizing logical leases (see Chapter 3).

2.1 Distributed Concurrency Control

A number of attempts have been made to design distributed concurrency control protocols [19, 28, 47, 53, 61]. Spanner [19] and F1 [53] follow the 2PL paradigm. MaaT [47] and Lomet et al. [46] are based on T/O. Among these protocols, MaaT [47] shows the most similarities with Sundial. They both coordinate transactions with logical timestamps and calculate the commit order of them dynamically. However, they use different techniques. MaaT assigns timestamp intervals to transactions and requires explicit manipulation of the logical time range of each transaction, which entails much higher overheads and causes more unnecessary aborts. Sundial does not let transactions interfere with other transactions directly. It works with the timestamp meta-data (i.e., a logical lease) assigned to each data item to indirectly infer the logical order. Futhermore, an efficient caching scheme is used in Sundial to reduce cost from the network latency of remote data access, whereas the design of MaaT does not support caching.

The work of Lomet et al. [46] presented a design of concurrency control protocol that applies the multi-version technique and the system decides the commit timestamp of a transaction using timestamp ranges similar to [47]. The protocol can be deployed to a single-server multicore architecture or a distributed cluster of servers. The protocol

dynamically detects conflicts and whenever there is one, the timestamp ranges of involved transactions are shrunk accordingly to eliminate the conflict. Sundial prevents such shrinking operations by applying logical leases. Furthermore, the database system working with Lomet et al.'s procotol needs to store multiple versions of data. We analyzed and compared the performance of Sundial and Lomet et al.'s construction, and our results are presented in Chapter 6.

2.1.1 Multicore Concurrency Control

Usually people will naturally try to extend and apply an idea used in multicore setting to distributed scenarios. Intensive research has been done in how to achieve efficient transaction processing on single-server multicore systems [38, 43, 49, 60, 62, 63, 68, 66]. Some of them are not out-of-box ready for the distributed setting because some operations become expensive due to the latency in a network roundtrip, as an example. We integrate some of the ideas from these works into Sundial.

In this work, we mainly look into the protocols for distributed concurrency control, and therefore those protocols which are specifically designed for single-server multi-core systems are not compared to.

2.1.2 Data Replication and Cache Coherence

Data replication [20, 50] is well-known to be effective for frequently accessed read-only data. When people replicate these data across servers, this replication can reduce unnecessary network requests and system overheads.

However, data replication requires human knowledge of spotting and identifying those frequently accessed data, or a systematic profiling on the workloads, both of which are daunting tasks for rapidly evolving databases. When the database updates these hot spots, all the servers hosting replicas of these data needs to be updated carefully to maintain the consistency, which is quite expensive. Furthermore, data replication increases the total memory usage, and is problematic if the size of replicated data is large.

Compared to data replication, a caching scheme is usually a better and more flexible

choice to take advantage of hot spot data, especially when the database system is of very large scale, with complicated data relationships, evolving and changing fast, or simply does not support easy human interventions. With hot spot data cached, a query that reads such data does not need to visit the distant data partition and thus avoids a network round trip, reducing both network latency and traffic.

The most critical challenge to build a caching scheme is to guarantee *cache coherence* (i.e., making sure that the data being read is still valid and updated). Cache coherence is one of the most 'classic' problems in the fields of computer architecture and computer systems. Significant efforts have been made to design and implement elegant caching schemes with cache coherence for multi-core and multi-socket architectures [9, 54, 70], distribtued shared memory systems [37, 42], distributed storage systems [33], and databases [8, 32, 51]. Conventionally, concurrency control algorithms and caching schemes are studied and designed independently in database systems. Both of them are difficult to construct and verify [55].

In Chapter 4, we will show that Sundial provides a harmonious and lightweight combination of both a concurrency control algorithm and a caching scheme, unifying them into a single protocol with the help of logical leases.

2.1.3 Integrating Concurrency Control and Cache Coherence

In a series of work on data sharing systems [52] such as IBM DB2 [36], Oracle RAC [15], Oracle RDB [45], and Microsoft Hydra [13], the idea of integrating concurrency control and cache coherence has been intensively studied. In these examples, the data is stored in a shared structure in distributed servers. Each server can read and write all the data in the database, with a cache buffer of recent accesses. The protocol guarantees the coherence of the cache. Any data fetched from the cache are not stale.

Sundial differs from these data sharing systems in two ways. First, data sharing systems are usually built on shared-disk architectures while Sundial can be run on commodity shared-nothing servers. Sundial scales better than usual data sharing systems. Second, the coherence protocol and the concurrency control scheme typically follow the 2PL paradigm,

whereas Sundial applies logical leases for both the concurrency control protocol and the cache coherence scheme, which is unified into a single lightweight and simple protocol.

G-store [22] is another related work that supports transaction processing over a key-value store. The system executes a protocol to transfer the ownership of all the keys that the transaction is going to access before the transaction actually starts. Ownership transfer in G-Store is equivalent to acquiring exclusive data copy in a coherence protocol; however, both reads and writes require exclusive ownership, which means transactions on different servers cannot read the same key concurrently.

The disadvantages of this include: (1) The DBMS needs to do static analysis or some other preprocessing to get the knowledge of what keys the transaction is going to access beforehand. (2) If more than one transaction wants to read a particular key, the system has to postpone all but one of them. In Sundial, we overcame these drawbacks by supporting concurrent data access and dynamic working sets.

Chapter 3

Sundial Concurrency Control

In this section we presents the details of the Sundial as a distributed concurrency control protocol. To make it easy to explain, we assume that the distributed system is built on top of a cluster of homogeneous commodity servers linked with a typical Local Area Network (LAN). The data is evenly split and deployed across these servers. Each data item gets assigned to a *home server*, where the data piece physically lies on. Every server can execute transactions as a coordinator, a.k.a., the transaction starts from this server, or the server helps process remote data access from a transaction coordinated at another server on behalf of it. The server that starts the transaction is called the *coordinator* of the transaction. Other servers which help process the remote data access requests of the transaction are called the *participants* of the transaction.

3.1 Logical Leases

In Sundial, a logical lease consists of two logical timestamps, marking the beginning and the ending of the lease. The logical timestamps in the systems specify a *partial logical order* among concurrent transactions. A logical lease corresponds to a version of the data item (although we do not actually keep multiple versions of the data), which is valid within the lease. If a transaction requests to write a new version of the data item, it has to acquire a new lease which logically comes after the current one. The two logical timestamps are represented by two 64-bit timestamps, *wts* and *rts*. *wts* is the logical timestamp when the

tuple was recently modified. *rts* marks the end of the lease, meaning the furthest future by which the system is sure that the version is valid. It is natural that we require for every lease $wts \leq rts$. A transaction can write to the data after the current version of the data 'expired' in terms of logical order. Namely, the transaction needs to write to the tuple at a timestamp no earlier than *rts* + 1. Note that since the leases are in logical timespan, the transaction that performs the writing does not necessarily have to wait in physical time for the lease to expire. It simply jumps ahead in logical time beyond the lease.

To enforce serializability, the concurrency protocol scheme decides *commit_ts* of a transaction such that it falls within the leases of all the tuples the tranasction has ever accessed. For each committed transaction, we calculated a commit timestamp *commit_ts* for it. The system requires that for each transaction with commit timestamp *commit_ts*, and each data tuple dt it read, we have $dt.wts \leq commit\_ts \leq dt.rts$, where $dt.wts$ and $dt.rts$ represents the logical lease the transaction acquired for the exact version it has accessed. In the logical timestamp space, the transaction gets executed *atomically* at the timestamp *commit_ts*. The system picks *commit_ts* by solving an inequality group with all the coefficients from the metadata (a.k.a., the logical lease information) of the data items. Thus, it does not require any direct interaction or coordination between transactions across the whole system. When we cannot find a valid solution for the inequality group, we do not immediately abort the transaction. We will try to see if any of the data that the transaction read could extend its lease to a larger timestamp, for example, because it has not been written and assigned to a new logical lease. While such an extention does not always succeed, it can reduce the unnecessary aborts. We will discuss this optimization in Chapter 5.

To more clearly explain our scheme, Fig. 3-1 illustrates an simple example of the high-level idea behind Sundial's concurrency control scheme. The system is going to execute two transactions *T1* and *T2*. Among these two, *T1* reads tuples **A** and **B** with logical leases [0, 1] and [1, 2] respectively (in this work we represent a logical lease that starts from x and ends at y by the symbol [x, y]). *T1* also writes to **D** and creates a new lease of [1, 1] for the new version of the data item. *T1* commits at timestamp 1 as it is the earliest feasible solution that overlaps with all the logical leases of the data versions *T1* accessed.

T2 writes **A** so it creates a new version for **A** with a new lease [2, 2]. The starting point

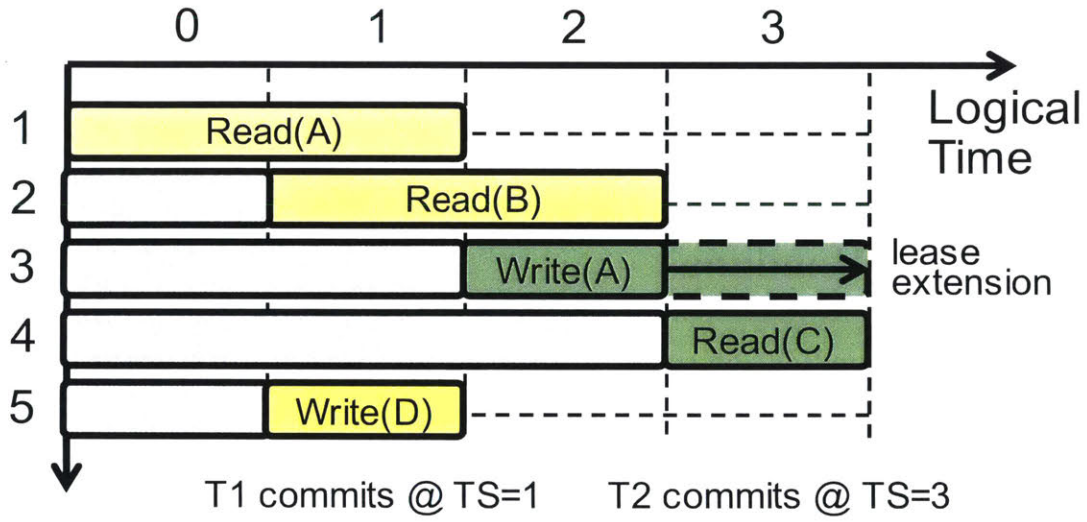


Figure 3-1: Logical Lease Example – Example schedule of two transactions, *T1* and *T2*, accessing tuples **A**, **B**, **C**, and **D**. *T1*'s and *T2*'s operations are shadowed in light yellow and dark green respectively. The horizontal and vertical axis means the order of logical timstamp and physical time, respectively. Since the logical timestamps are discrete, we represent each timestamp as an interval on the horizontal axis in the figure.

of the new lease equals the previous *rts* + 1, meaning that the new version is valid after the previous lease expires. *T2* reads **C** at [3, 3] because the current version of C was written by some other transaction and its logical lease starts from logical timestamp 3. Notice that these logical leases do not overlap, meaning there is no solution to the inequality group. In this case, as mentioned earlier, the scheme extends the end of the lease on **A** from 2 to 3 such that both logical leases are valid at 3. If no other transaction has written to **A** at timestamp 3, the lease extension succeeds. *T2* has a valid solution of its commit timestamp at 3.

Another observation in this simple example is that *T2* has already modified **A** before *T1* commits, whereas **A** has already read by *T1*. In conventional OCC protocols, the system will check the data items that *T1* touched, making sure no one has modified them. In this case, we can see that the writing operation from *T2* caused *T1* to abort. In Sundial, with the support of logical timestamps, such a situation does not cause *T1* to abort because Sundial serializes transaction in logical timestamp order rather than physical time order. Therefore, the transactions can jump in physical time, as long as their logical timestamp order is sound and valid. As shown in this example, although *T1* commits with a smaller logical timestamp, it commits after *T2* in physical time order. This dynamic scheduling feature allows the

database system to gain performance by enabling more transactions to commit compared to traditional OCC protocols and reducing unnecessary aborts caused by read-write conflicts. The abort rate is therefore lowered and concurrency is improved.

3.2 Conflict Handling

Sundial applies two kinds of approaches to resolving two categories of conflicts between transactions being processed. Previous work that embeds similar hybrid schemes exist both in database field [25, 56] and software tranasctional memory [27]. In Sundial, we handle *read-write conflicts* with an OCC style protocol, and *write-write conflicts* by the pessimistic 2-phase locking scheme. For write-write conflicts, our design decision results from an observation we made about database applications, that writing operations mainly consists of read-modify-writes. Two such writing operations on the same target would certainly cause a conflict, if no extra semantic knowledge about the operation like commutativity is exploited. Handling such conflicts with OCC will result in all but one of the conflicting transactions to abort and most of the computation resources are unfortunately wasted. For read-write conflicts, using OCC prevents a transaction from wasting time in waiting for locks, especially on some hot spot data items, which are likely to cause a bottleneck in transaction execution and greatly reduce the parallelization. More importantly, this allows the system to dynamically change the execution order of transactions and give up early on some of the transaction executions. Combined with the techniques of logical timestamps, the system can further reduce unnecessary aborts.

To better explain this, Fig. 3-2 shows an example to illustrate the difference between 2-phase locking, traditional OCC, and Sundial in dealing with read-write conflicts. In 2-phase locking, the database system notices that the transaction *T1* requests to perform a read operation. It then acquires a lock for *T1* before it actually reads it. Another transaction *T2* tries to write to the exact same tuple. The lock held by *T1* blocks the progress of *T2* from its writing operation. This increases the execution time of *T2* because all the following operations of *T2* are in turn affected.

Things are different when it comes to traditional OCC protocols. A transaction does

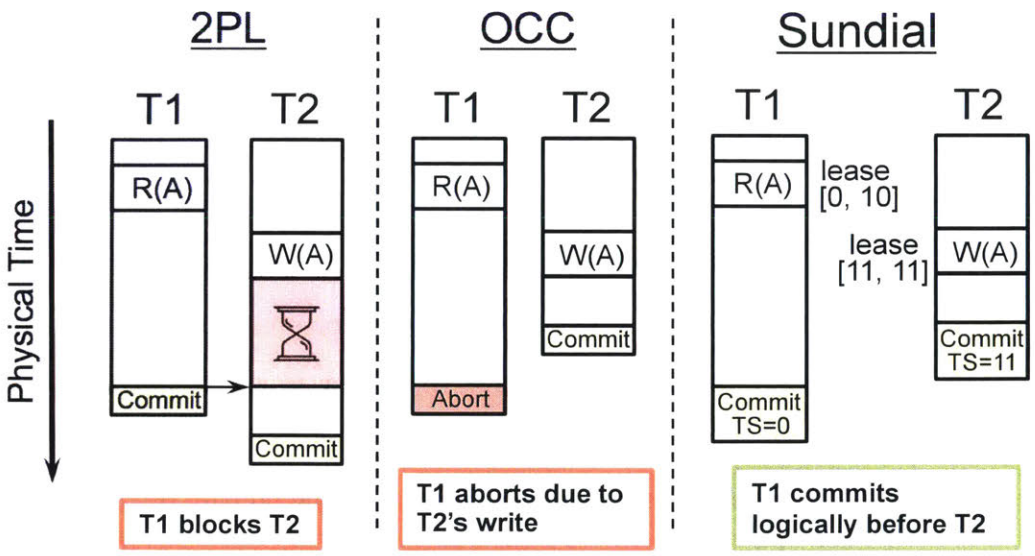


Figure 3-2: Read-Write Conflict Example – Example schedule of two transactions with a read-write conflict in 2-phase locking, traditional OCC, and Sundial.

not need to acquire a lock before accessing data items. Therefore, *T2* finishes and commits before *T1*, although it wrote to the data tuple `A` in the database. When *T1* finishes execution, the system finds that the data tuple `A` which *T1* accessed earlier has been changed to a new value by *T2*. Then, because of this conflict, the OCC protocol aborts the transaction *T1* since the effects proposed by *T1* might not be serializable any more.

However, in this simple schedule example, both *T1* and *T2* could have successfully committed because there exists a logical commit order where *T2* commits after *T1* that still preserves serializability. The concurrency control scheme behind Sundial is able to discover this order with the help of logical leases and enforce it. For example, *T1* accessed the version of `A` with a lease of `[0, 10]`. It commits at a timestamp inside the range, a.k.a., at timestamp 0 (here we assume there is no other constraint for *T1*). *T2* performs its writing operation to `A` and creates a new logical lease of [11, 11]. Then *T2* commits at timestamp 11 (similarly, we assume there is no other constraint for *T2*). Although *T1* in physical time committed after *T2*, in logical order *T1* commits before *T2*. Both transactions are approved to commit and take effect onto the data storage.

3.3 Protocol Phases

Fig. 3-3 shows the lifecycle of a typical distribuetd transaction executed in Sundial. It consists of three phases-(1) **execution**, (2) **prepare**, and (3) **commit**. The coordinating server (the first column) starts the transaction. In the execution phase, the coordinator server follows the content of the transaction and executes it. Whenever necessary, the server sends requests on behalf of the transaction to access remote data items. Then, the coordinator starts the prepare phase after the transaction finishes execution. The prepare phase and commit phase compose a typical 2-phase commit process. In the prepare phase, the coordinator sends *prepare* requests to all the remote servers that have been accessed in the transaction execution. Each server responds with a message indicating whether or not the corresponding transaction can be committed according to their local knowledge. If all the responses are positive, the coordinator server pushes the transaction into the commit phase, where it starts a new network roundtrip to all the remote servers that participated and finishes the transaction. Otherwise, the database system aborts the transaction, giving up changes before they are applied, and releasing all the locks. The system is designed to tolerate server crash-stop failure as well as network failure. The prepare phase and the commit phase apply the typical logging strategy widely used in 2-phase commitments. Details of logging are illustrated in Fig. 3-3.

During the commit phase of a transaction, the changes are applied into the database. Only at this point, the metadata associated with the previous data version is erased and modified into a new logical lease corresponding to the new version. As mentioned above, the system might extend the logical lease of a data item when needed and lease overlap will be caused. In this case, the *rts* of the logical lease are modified but no new data version is created. The logical leases for any particular data item is disjoint and non-overlapping. The *wts* and the *rts* of a tuple increase monotonically while the transaction processing progresses. The database system piggybacks the necessary logical timestamp information in its network messages. We now present a detailed explanation in the following parts of this section.

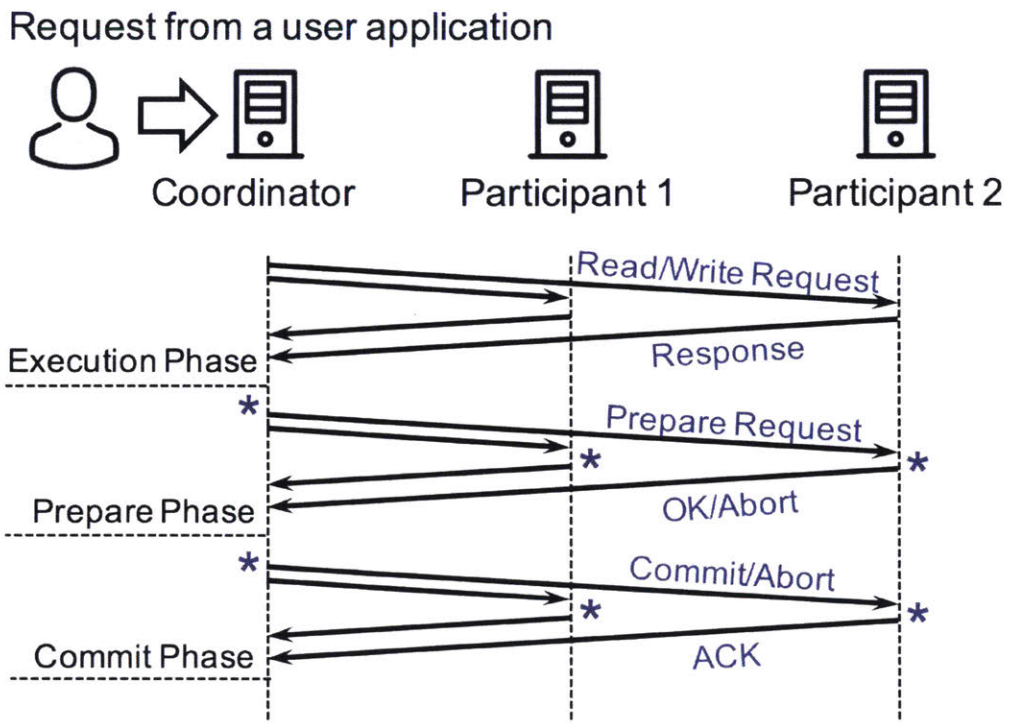


Figure 3-3: The lifecycle for a Distributed Transaction – A typical distributed transaction in Sundial goes through the execution phase, prepare phase and commit phase.

3.3.1 Execution Phase

The content of the transaction is executed in the execution phase. The database system performs the logics of the transaction and processes read and write requests on behalf of the transaction. Each data item is assigned a logical lease. The logical lease consists of two 64-bit integers, *wts* and *rts*. In Sundial, the protocol handles write-and-write conflicts using the 2-phase locking scheme (of the wait-and-die flavor) [12]. To implement this scheme, the meta-data field also maintains the *lock owner* and a variable length *wait-list* of transactions waiting for the lock. In this work, we denote the metadata together with the data of a tuple in the database `DB` with the symbol `DB[key]`, where `key` is the primary key of the tuple.

```
DB[key] = {wts, rts, owner, waitlist, data}
```

Sundial resolves read-write conflicts with an OCC scheme. Therefore, the system keeps track of the *read set* (RS) and the *write set* (WS) of each transaction. Different from traditional OCC's, we not only record the data itself in a read/write set element, but also the

logical timestamp of the data version. Formally, each element in the read/write set looks like below.

$$\texttt{RS[key]} = \{\texttt{wts, rts, data}\},$$
$$\texttt{WS[key]} = \{\texttt{data}\},$$

where `data` represents a full copy (though sometimes for a large data item, we might store an incremental change instead) of the database tuple that the transaction reads or writes.

The detailed pseudo code of the execution phase is shown in Algorithm 1. We design the execution phase to listen to the read and write events of the transaction. The code marked in gray shadow is related to the caching scheme in Sundial. We will explain it later in Chapter 4.

Whenever the transaction initiates a read request, the database will check if the data item is already in the read set or the write set of the transaction. If yes, the database system simply feeds the request with the data on hand (lines 2–5), reflecting the latest update of the data. Otherwise, the system sends a remote procedure call (RPC_n) to the server hosting that data item. The RPC will read the metadata and data atomically, and return them to the home server to add them to the read set of the transaction (lines 10–12). The coordinator then updates the *commit_ts* of the transaction to be no smaller than the *wts* of the tuple (line 14). This corresponds to the inequality that the commit timestamp should be larger than the *wts* of all the logical leases it acquired (i.e., the commit event happens after the time when the data version was written, in the logical order). The other part of the inequality (i.e., the commit event should happen before the end of the lease) will be enforced later. The procedure returns the data to the transaction execution at the end (line 15).

Similarly, for the write operation the database system does a check if the tuple is already in the write set of the transaction. If found, the database system simply updates the local data in the write set to reflect the lastest change (line 17). Otherwise, it locks the tuple by initiating an RPC to the server hosting that data item (lines 18–20). The RPC returns a message indicating whether the lock is successfully acquired. And if so, the *wts* and *rts* of the locked tuple will be piggybacked in the return message. Otherwise, if the data tuple

Algorithm 1: Execution Phase of Transaction T – *T.commit_ts* is initialized to 0 when T begins. Caching related code is highlighted in gray (cf. Chapter 4).

```
1  Function read(T, key)
2      if key ∈ T.WS:
3          return WS[key].data
4      elif key ∈ T.RS:
5          return RS[key].data
6      else:
7          if key ∈ Cache and Cache.decide_read():
8              T.RS[key].{wts, rts, data} = Cache[key].{wts, rts, data}
9          else:
10             n = get_home_node(key)
11             # read_data() atomically reads wts, rts, and data and return them back
12             T.RS[key].{wts, rts, data} = RPC_n::read_data(key)
13             Cache[key].{wts, rts, data} = T.RS[key].{wts, rts, data}
14         T.commit_ts = Max(T.commit_ts, T.RS[key].wts)
15         return RS[key].data

16 Function write(T, key, data)
17     if key ∉ T.WS:
18         n = get_home_node(key)
19         # lock() tries to lock the tuple DB[key]; if locking is successful, it returns wts and rts
             of the locked tuple
20         {success, wts, rts} = RPC_n::lock(key)
21         if not success or (key ∈ T.RS and wts ≠ T.RS[key].wts):
22             Abort(T)
23         else:
24             T.commit_ts = Max(T.commit_ts, rts + 1)
25     T.WS[key].data = data
```

is already locked by another transaction, the transaction aborts. If the lock is successfully acquried but coordinator finds that the locked key exists in the read set of the transaction, and the *wts* from the remote server does not match the *wts* recorded in the read set, the system will also abort the transaction since in this case someone else has already acquired the lock, written another version of the data, and released the lock. The system will advance the commit timestamp of the transaction to *rts* + 1 (i.e., the *wts* assigned to the new version that is going to take effect in the database if the transaction successfully commits), and add a new element to the write set of the transaction with the data intended in this write operation. The content in the write set will only be visible after the transaction is committed (Section 3.3.3).

As mentioned earlier, read-write conflicts in Sundial are processed in a non-blocking fashion. Even if a tuple is locked, a read operation from a transaction is still permitted. And that read will return the corresponding logical lease associated with the version stored in the database at that time. In this case, apparently someone else is trying to write to that exact tuple, and the read operation happens before the data change takes effect. The database system will make sure that the commit timestamp of the reading transaction is going to be smaller than that of the writing transaction (i.e., the lock owner of that data tuple). And thereby both transactions are able to commit as long as they successfully handle other conflicts.

3.3.2 Prepare Phase

Following the paradigm of 2-phase commitment, the prepare phase determines if all the remote servers involved, as well as the coordinator, agree that inequalities originated from the read and write operations of the transaction are satisfied at the *commit_ts* calculated at the coordinator server. The logic of the prepare phase is shown in Algorithm 2; *validate_read_set* is executed at the coordinator. *renew_lease* is a remote procedure call actually executed at the participating servers.

At this time, the transaction is holding a lock for each of the tuples recorded in the write set. No other transaction could have the chance of modifying the data tuple. The database system only validates those tuples that were accessed but not written. Notice that some keys might appear in the record elements of the read set of the transaction but also exist in the write sets. For these keys, the database system does not have to perform validations for them because the locks for them are being held. For each key in the read set but not the write set (line 2), the system checks if *commit_ts* is within the time interval of the logical lease, i.e., $RS[key].wts \leq commit\_ts \leq RS[key].rts$. Since during the read operation in the execution phase we already make sure that $RS[key].wts \leq commit\_ts$, we only have to check the right part of the inequality, i.e., $commit\_ts \leq RS[key].rts$. If any part of the validation fails, before aborting the transaction, the system does one more thing to try to save it. The coordinator server sends an RPC to the home server of the tuple to extend the

Algorithm 2: Prepare Phase of Transaction T – *validate_read_set()* is executed at the coordinator; *renew_lease()* is executed at the participants.

```
1  Function validate_read_set(T)
2      for key ∈ T.RS.keys() \ T.WS.keys():
3          if commit_ts > T.RS[key].rts:
4              n = get_home_node(key)
5              # Extend rts at the home server
6              resp = RPC_n::renew_lease(key, wts, commit_ts)
7              if resp == ABORT:
8                  Cache.remove(key)
9                  return ABORT
10     return COMMIT

11 # renew_lease() must be executed atomically
12 Function renew_lease(key, wts, commit_ts)
13     if wts ≠ DB[key].wts or (commit_ts > DB[key].rts and DB[key].is_locked()):
14         return ABORT
15     else:
16         DB[key].rts = Max(DB[key].rts, commit_ts)
17         return OK
```

lease (lines 4–6) (i.e., to see if the *rts* could be increased). If the lease cannot be extended, the server aborts the transaction.

At the remote server, when a request of extending the logical lease is received, the server performs the *renew_lease* function for lease renewal. The logic is simple. If the current *wts* in the database is different from the *wts* observed by the transaction during the execution phase, or the tuple has been locked by another transaction, then the database system will return an ABORT signal to the sender that the extension could not be done. Otherwise, the logical lease can be extended to be at least *commit_ts* by the server. And a OK signal is returned to the coordinator server. If all the servers involved returned a OK message, the coordinator pushes the transaction into the commit phase. Otherwise, the transaction is aborted.

Note that both *wts* and *rts* of a tuple never decrease.

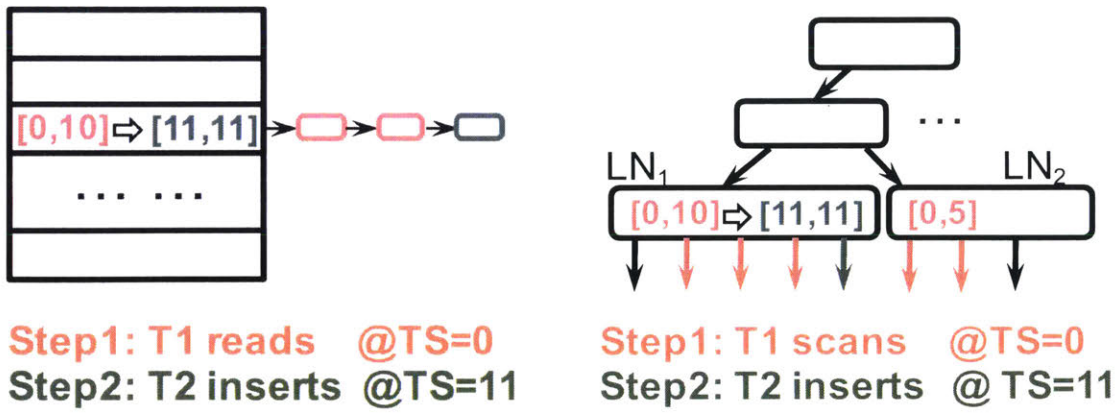
Algorithm 3: Commit Phase of transaction T – The transaction is committed or aborted here.

```
1  Function commit(T)
2      for key ∈ T.WS.keys():
3          n = get_home_node(key)
4          RPC_n::update_and_unlock(key, T.WS[key].data, T.commit_ts)
5          Cache[key].wts = Cache[key].rts = T.commit_ts
6          Cache[key].data = T.WS[key].data

7  Function abort(T)
8      for key ∈ T.WS.keys():
9          n = get_home_node(key)
10         # unlock() releases the lock on DB[key]
11         RPC_n::unlock(key)

12 Function update_and_unlock(key, data, commit_ts)
13     DB[key].data = data
14     DB[key].wts = DB[key].rts = commit_ts
15     unlock(DB[key])
```

3.3.3 Commit Phase

If the coordinator decides to abort the transaction, the system executes a cleanup procedure for it to release all the locks. Otherwise, the transaction is committed and all the temporary modifications in the write set are applied into the database, with the locks released. Algorithm 3 shows the logic of this phase. The function *abort* is executed for aborting transactions and *commit* is executed for committing transactions.

If the transaction is a read-only transaction, the system goes through a shortcut where the whole commit phase is skipped. Or, If a transaction does not perform any write operation on a remote server, the coordinator will skip this server during the commit phase and that server, observing the transaction being read-only, will simply drop the transaction (since whether it aborts or not will not make any difference to that server's point of view). Such small optimzations help reduce network traffics. Finally, for either committed or aborted transactions all the servers involved drop the transaction's local read and write set, and release the memory.



(a) Hash Index (b) B+tree Index

Figure 3-4: Concurrent Index Read/Scan and Insertion in Sundial – Logical leases are embedded into the leaf nodes of a B+tree index to enable high concurrency in index accesses.

3.4 Indexes and Phantom Reads

Beyond regular reads and writes, indexes must support inserts and deletes. As a result, an index requires extra concurrency control mechanisms for correctness. Specifically, a *phantom read* occurs if a transaction reads a set of tuples twice using an index but gets different sets of results due to another transaction's inserts or deletes to the set. Serializability does not permit phantom reads. This section discusses how Sundial avoids phantom reads.

By treating inserts and deletes as writes to index nodes, and lookups or scans as reads to index nodes, the basic Sundial protocol can be extended to handle index accesses. This way, each index node (e.g., a leaf node in a B+tree index or a bucket in a hash index) can be treated as a regular tuple and the protocol discussed in Section 3.3 can be applied to indexes. The logical leases can also be maintained at a finer granularity (e.g., each pointer in the leaf node or each block in a bucket) to avoid unnecessary aborts due to false sharing; in Sundial, we attach a logical lease to each index node. In order to ensure that multiple index lookups/scans to the same index node return consistent results, later accesses must verify that the index node's version has not changed.

Fig. 3-4 shows two examples of transactions concurrently accessing a hash index (Fig. 3-4a) and a B+tree index (Fig. 3-4b). In the hash index example, *T1* reads all the tuples mapped to a given bucket with a lease of [0, 10]. *T2* then inserts a tuple into the same bucket, and updates the lease on the bucket to [11, 11]. *T1* and *T2* have a read-write conflict but do not

block each other; both transactions can commit if each finds a commit timestamp satisfying all the accessed leases.

In the B+tree index example, *T1* performs a scan that touches two leaf nodes, LN_1 and LN_2, and records their logical leases ([0, 10] and [0, 5]). After the scan completes, *T2* inserts a new key into LN_1. When *T2* commits, it updates the contents of LN_1, as well as its logical lease to [11, 11]. Similar to the hash index example, both *T1* and *T2* may commit and *T1* commits before *T2* in logical time order. Thus, concurrent scans and inserts need not cause aborts.

3.5 Fault Tolerance

Sundial tolerates single- and multi-server failures through the two-phase commit (2PC) protocol, which requires the coordinator and the participants to log to persistent storage before sending out certain messages (Fig. 3-3). The logical leases in Sundial, however, require special treatment during 2PC: failing to properly log logical leases can lead to non-serializable schedules, as shown by the example in Listing 3.1.

The example contains two transactions (*T1* and *T2*) accessing two tuples, **A** and **B**, that are stored in servers 1 and 2, respectively. When server 2 recovers after crashing, the logical leases of tuples mapped to server 2 (i.e., tuple **B**) are reset to [0, 0]. As a result, the DBMS commits *T1* at timestamp 0, since the leases of tuples accessed by *T1* (i.e., **A** of [0, 9] and **B** of [0, 0]) overlap at 0. This execution, however, violates serializability since *T1* observes *T2*'s write to **B** but not its write to **A**. This violation occurs because the logical lease on **B** is lost and reset to [0, 0] after server 2 recovers from crash. If server 2 had not crashed, *T1*'s read of **B** would return a lease starting at timestamp 10, causing *T1* to abort due to non-overlapping leases and a failed lease extension.

Listing 3.1: Serializability violation when logical leases are not logged – Tuples A and B are stored in servers 1 and 2, respectively

```
T1 R(A)                  lease [0, 9]
T2 W(A), W(B), commit    @ TS=10
Server 2 crashes
Server 2 recovers
```

```
T1 R(B)                lease [0, 0]
```

One simple solution to solve the problem above is to log logical leases whenever they change, and restore them after recovery. This, however, incurs too much writing to persistent storage since even a read operation may extend a lease, causing a write to the log.

We observe that instead of logging every lease change, the DBMS can log only an *upper-bound timestamp* (UT) that is greater than the end of all the leases on the server. After recovery, all the leases on the server are set to [UT, UT]. This guarantees that a future read to a recovered tuple occurs at a timestamp after the *wts* of the tuple, which is no greater than UT. In the example shown in Listing 3.1, *T1*'s read of **B** returns a lease of [UT, UT] where UT is greater than or equal to 10. This causes *T1* to abort due to non-overlapping leases. Note that UT can be a loose upper bound of leases. This reduces the storage and logging overhead of UT since each upper bound is logged once only when the maximum lease exceeds the last logged.

3.6 External Consistency

External consistency (or linearizability [35]) is an isolation level not required in traditional DBMSs but supported in some systems [18, 3]. It brings the following salient feature to a DBMS – if a transaction starts after the commit of a previous transaction, the later transaction is also after the previous transaction in the logical serialization order. This feature is useful when the two transactions have a causal relationship due to some external event (i.e., a phone call) outside the database. However, supporting external consistency typically requires synchronizing the clocks across the database servers which can be prohibitive.

Sundial does not support external consistency by default due to its usage of logical time. But external consistency can be supported by exposing the logical timestamps to the external world. Specifically, the commit timestamp of a transaction can be returned to the end users, who forward it to dependent transactions. A dependent transaction feeds this timestamp to the database as its minimum commit timestamp thereby forcing the commit order.

To enforce the commit order between transaction *T1* and *T2*, we can feed *T1*'s commit timestamp as an input argument to *T2*. This sets *T2*'s initial *min_commit_ts* to *T1*'s *commit_ts*.

Therefore, *T2* will commit with a greater commit timestamp and be ordered after *T1*.

There are other distributed DBMSs that also support external consistency (e.g., Spanner [19] and CockroachDB [3]). These systems, however, use globally synchronized clocks (based on atomic clocks or NTP) for consistent time. Compared with these designs, Sundial does not require any global clock synchronization for external consistency. One downside of Sundial, however, is that the commit timestamps of the predecessor transactions have to be sent to successor transactions. These messages can, however, piggyback on an external communication channel between these transactions.

Chapter 4

Sundial Data Caching

Caching a remote partition's data in a server's local main memory can reduce the latency and network traffic of distributed transactions, because reads that hit the local cache do not contact the remote server. Caching, however, causes data replication across servers; it is a challenging task to keep all the replicas up to date when some data is updated, a problem known as cache coherence [10]. Due to the complexity of maintaining coherence, existing distributed DBMSs rarely allow data to be cached across multiple servers. As we now present, Sundial's logical leases enable such data caching by integrating concurrency control and caching into a single protocol.

Fig. 4-1 shows an overview of Sundial's caching architecture in a DBMS. The system only caches tuples for reads and a write request updates both the tuple at the home server and the locally cached copy. For a read query, the DBMS always checks the coordinator's local cache. For a hit, the DBMS decides to either read the cached tuple or ignore it and send a query to the tuple's home server. Later in Section 4.2, we will show that the decision depends on the scenario.

To avoid centralized bottlenecks, Sundial organizes the cache into multiple banks. A tuple's bank is determined by hashing its primary key. Each bank maintains the metadata for the tuples it contains using a small index. When the bank is full, tuples are replaced following a *least-recently-used* (LRU) policy.

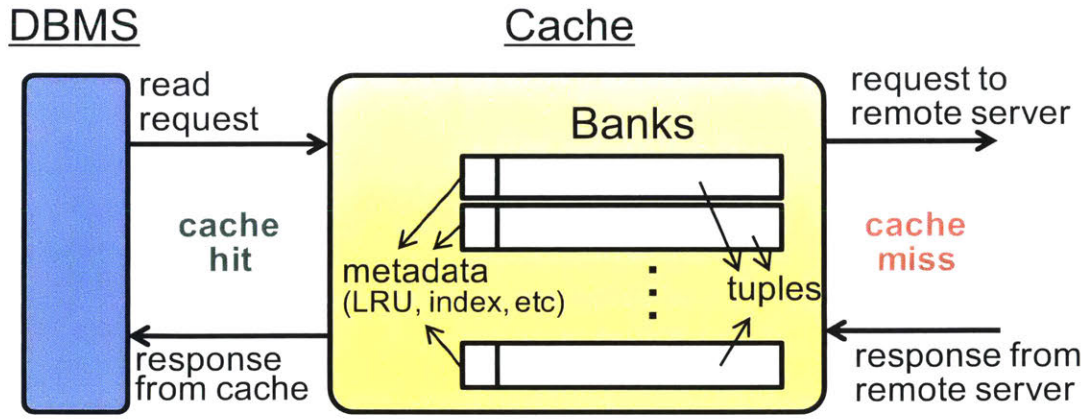


Figure 4-1: Caching Architecture – The cache is at the network side, containing multiple banks with LRU replacement.

4.1 Cache Coherence with Logical Leases

In a system where data can be cached at multiple locations, a cache coherence protocol enforces that the value at each location is always up-to-date; namely, when one copy is updated, the change must propagate to all the copies. Existing coherence protocols either (1) require an invalidation mechanism to update or delete all the shared copies, or (2) check the data freshness by contacting the home server for each read request.

The downside of the invalidation-based approach is the complexity and the performance overhead of broadcasting each tuple update. The downside of the checking approach is that each read request incurs the round-trip latency (though data is not transferred if the cached copy is up-to-date), reducing some of the benefits of caching.

The caching mechanism in Sundial is a variant of the checking approach mentioned above. However, instead of checking freshness for each read request, the use of logical leases reduces the number of checks. Specifically, a transaction can read a cached tuple as long as its *commit_ts* falls within the lease of the tuple, even if the tuple has been changed at the home server—the transaction reading the "stale" cached copy can still be serializable with respect to other transactions in logical time order. In this sense, Sundial relaxes the requirement of cache coherence: there is no need to enforce that all cached copies are up-to-date, only that serializability is enforced. Logical leases provide a simple way to check serializability given the read and write sets of a transaction, regardless of whether the

reads come from the cache or not.

Supporting caching requires a few changes to the protocol presented so far (Section 3.3); they are shadowed in gray in Algorithms 1–3. During the execution phase, a remote read request checks the cache (Algorithm 1, lines 7–8) and either reads from the cache (Section 4.2) or requests the home server home server (line 13). In the validation phase, if a lease extension fails, the tuple is removed from the cache to prevent repeated failures in the future (Algorithm 2, line 8). Finally, if a transaction commits, it updates the data copies in both the home server and the local cache (Algorithm 3, lines 5–6). These are relatively small protocol changes.

The caching mechanism discussed so far works for primary key lookups using an equality predicate. But the same technique can also be applied to range scans or secondary index lookups. Since the index nodes also contain leases, the DBMS caches the index nodes in the same way it caches tuples.

4.2 Caching Policies

We now discuss different ways to manage the cache at each server and their corresponding tradeoffs.

Always Reuse: This is the simplest approach, where the DBMS always returns the cached tuple to the transaction for each cache hit. This works well for read-intensive tuples, but can hurt performance for tuples that are frequently modified. If a cached tuple has an old lease, it is possible that the tuple has already been modified by another transaction at the home server. In this case, a transaction reading the scale cached tuple may fail to extend the lease of that tuple, which causes the transaction to abort. These kinds of aborts can be avoided if the locally cached stale data is not used in the first place.

Always Request: An alternative policy is where the DBMS always sends a request to the remote server to retrieve the tuple, even for a cache hit. In this case, caching does not reduce latency but may reduce network traffic. For a cache hit, the DBMS sends a request to a remote server. The request contains the key and the *wts* of the tuple that is being requested. At the home server, if the tuple's *wts* equals the *wts* in the request, the tuple cached in the

requesting server is the latest version. In this case, the DBMS does not return the data in the response, but just an acknowledgment that the cached version is up-to-date. Since data comprises the main part of a message, this reduces the total amount of network traffic.

Hybrid: `Always Reuse` works best for read-intensive workloads, while `Always Request` works best for write-intensive workloads. Sundial uses a hybrid caching policy that achieves the best of both. At each server, the DBMS maintains two counters to decide when it is beneficial to read from the cache. The counter *vote_cache* is incremented when a tuple appears up-to-date after a remote check, or when a cached tuple experiences a successful lease extension. The counter *vote_remote* is incremented when a remote check returns a different version or when a cached tuple fails a lease extension. The DBMS uses `Always Reuse` when the ratio between *vote_cache* and *vote_remote* is high (we found 0.8 to be a good threshold), and `Always Request` otherwise.

4.3 Read-Only Table Optimizations

Care is required when the logical lease of a cached tuple is smaller than a transaction's *commit_ts*. In this case, the DBMS has to extend the tuple's lease at its home server. Frequent lease extensions may be unnecessary, and hurt performance. The problem is particularly prominent for read-only tables, which in theory do not require lease extension. We now describe two techniques to reduce the number of lease extensions for read-only tables. The DBMS can enable both optimizations at the same time. We evaluate their efficacy in Section 6.4.1.

The first optimization tracks and extends leases at table granularity to amortize the cost of lease extensions. The DBMS can tell that a table is read-only or read-intensive because it has a large ratio between reads and writes. For each table, the DBMS maintains a *tab_wts* that represents the largest *wts* of all its tuples. The DBMS updates a table's *tab_wts* when a tuple has greater *wts*. A read-only table also maintains *tab_rts*, which means all tuples in the table are extended to *tab_rts* automatically. If any tuple is modified, its new *wts* becomes `Max`(*rts* +1, *tab_rts* +1, *wts*). When the DBMS requests a lease extension for a tuple in a read-only table, the DBMS extends all the leases in the table by advancing *tab_rts*. The

tab_rts is returned to the requesting server's cache. A access hit of a tuple in that table considers the lease to be [`wts`, `Max(tab_rts, rts)`].

Another technique to amortize lease extension costs is to speculatively extend the lease to a larger timestamp than what a transaction requires. Instead of extending the *rts* (or *tab_rts*) to the *commit_ts* of the requesting transaction, Sundial extends *rts* to *commit_ts* + δ for presumed read-only tables. Initially being 0, δ is incrementally increased over time as the DBMS gains more information that the table is indeed read-only. This reduces the frequency of lease extensions since it takes longer time for an extended lease to expire.

Chapter 5

Discussion

Given Sundial's concurrency control protocol and caching scheme described in the previous sections, we now characterize the different types of transaction aborts that can occur in the DBMS and discuss potential solutions for reducing them. We also compare Sundial with dynamic timestamp allocation [11] algorithms and show Sundial's advantages over these previous approaches.

We will also compare Sundial with an implementation of an idealized Multi-Versioning Concurrency Control system. As we will see, even without the complexity of multi-versioning, Sundial can achieve the same level of performance. Then, we discuss Dynamic Timestamp Allocation (DTA) systems which also utilize logical timestamps to control concurrency conflicts. As a typical example of DTA schemes, MaaT [47] will be compared with Sundial in some detail. MaaT reorders transactions for higher concurrency.

5.1 Transaction Aborts

As described in Section 3.2, write-write conflicts are difficult for a DBMS to prevent and thus they are not the main focus of Sundial. These kinds of aborts are inevitable if we choose not to delay or abort the writing operations of other transactions. Therefore, we focus on aborts caused by read-write conflicts.

There are three conditions in Sundial that a transaction's commit timestamp must satisfy before the DBMS is allowed to commit it:

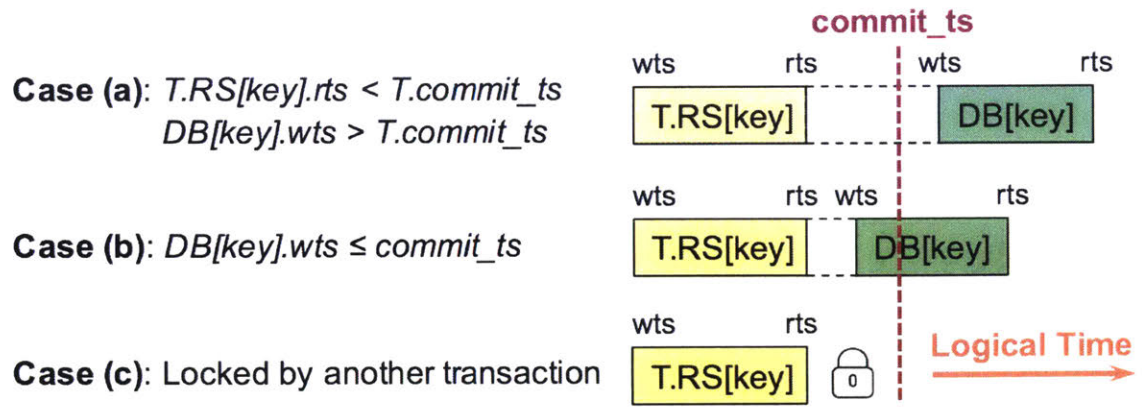


Figure 5-1: Transaction Aborts due to Read-Write Conflicts – Three cases where the DBMS aborts a transaction in Sundial due to read-write conflicts.

$$T.commit\_ts \geq tuple.wts, \qquad \forall tuple \in T.RS \tag{5.1}$$

$$T.commit\_ts \leq tuple.rts, \qquad \forall tuple \in T.RS \tag{5.2}$$

$$T.commit\_ts \geq tuple.rts + 1, \qquad \forall tuple \in T.WS \tag{5.3}$$

Condition (5.1) is always satisfied since *commit_ts* is no less than the *wts* of a tuple when it is read by a transaction (Algorithm 1) and that each write locks the tuple to prevent it from being changed by another transaction. Likewise, Condition (5.3) is also always satisfied because each write locks the tuple and updates the transaction's *commit_ts*, and no other transaction can modify the tuple's *rts* because of the lock.

Therefore, during the prepare phase, the DBMS can abort a transaction only if it fails Condition (5.2), i.e., the transaction fails to extend a lease since the tuple was locked or modified by another transaction (Algorithm 2). There are three scenarios where a transaction T fails Condition (5.2), which Fig. 5-1 illustrates:

Case (a): The tuple's *wts* in the database is greater than or equal to T's *commit_ts*. The DBMS must abort T because it is unknown whether or not the version read by T is still valid at T's *commit_ts*. It is possible that another transaction modified the tuple after $T.RS[key].rts$ but before *commit_ts*, in which case the DBMS has to abort T. But it is also possible that no such transaction exists such that the version in T's read set is still valid at

commit_ts and thus T can commit. This uncertainty could be resolved by maintaining a history of recent *wts*'s in each tuple [66].

Case (b): Another transaction already wrote the latest version of the tuple to the database before T's *commit_ts*. The DBMS is therefore unable to extend the lease of the transaction's local version to *commit_ts*. As such, the DBMS has to abort T.

Case (c): Lastly, the DBMS is unable to extend the tuple's *rts* because another transaction holds the lock for it. Again, this will cause the DBMS to abort T.

For the second and third conditions, Sundial can potentially avoid the aborts if the DBMS extends the tuple's lease during the execution phase. This reduces the number of renewals during the prepare phase, thereby leading to fewer aborts. But speculatively extending the leases also causes the transactions that update the tuple to jump further ahead in logical time, leading to more extensions and potential aborts.

5.2 A WOCC variant of Sundial

It is natural to consider the use of OCC-style schemes for write-write conflicts. We call such a variant of Sundial the WOCC variant. In this section we discuss the potential aborts introduced by the WOCC variant and why we stick to a hybrid design in Sundial.

For the WOCC variant of Sundial, a transaction only locks the write set after it finishes the execution phase. Since the lifespan of the lock does not depend on the execution logic, we can expect fewer aborts caused by locks. Although this can reduce the kinds of aborts of case (c) in Fig. 5-1, it raises the possibility of other potential aborts. An obvious kind of extra aborts happens when some data item in the write set get written by another transaction. The conventional practice of OCC-style protocols will find such conflicts and abort the transaction in validation phase. However, due to the fact that we allow transactions to extend a lease (a.k.a., to extend the *rts* of the data item), we expect a non-trivial kind of extra aborts in the system.

After the local preparation, the coordinator sends a timestamp which is valid locally to all the remote servers involved in the transaction. Upon receiving the timestamp, a remote server goes through the read set and write set of its sub-transaction and checks if

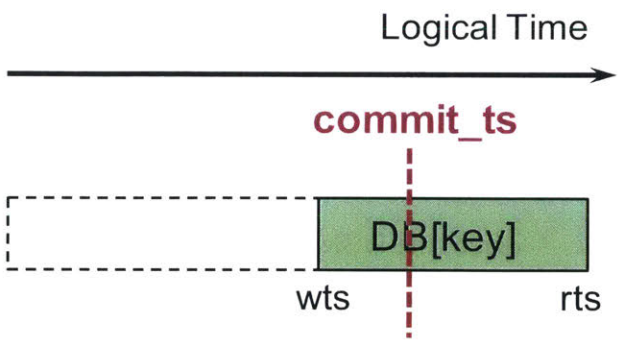


Figure 5-2: Extra aborts caused by remote execution. Another transaction has extended the *rts* of the data item.

the timestamp is valid. If it finds any of the inequalities in (5.1)-(5.3) not satisfied, it will abort the transaction and inform the coordinator. During the validation, in addition to the kinds of aborts discussed above, we expect an extra case of aborts. Shown in Fig. 5-2, the remote server might find the timestamp proposed by the coordinator to be less than (*rts* + 1) of a tuple in its write set (because another transaction has already extended the *rts* of the tuple), which violates the condition in (5.3). The conditions in (5.1) and (5.2) are guaranteed by the coordinator. Notice that since during the execution phase the remote server will return the *rts* and *wts* to the coordinator to make sure the coordinator knows them locally before the preparation phase, the timestamp proposed by the coordinator always satisfies the restrictions by these tuples although they are on the remote servers.

Sometimes the aborts in this case are unnecessary. Notice that the coordinator always tries to propose the minimal timestamp to the remote servers. In reality, it might accept timestamps (slightly) larger than the proposed one (as long as the timestamp is less than *rts* of all the tuples in Read Set). Therefore, it is possible that there exists a larger timestamp that can actually satisfy all the conditions, both for the coordinator and the remote servers. If the coordinator tells the remote servers all the acceptable time ranges, it is possible that the transaction gets saved. In later sections we will explain this case in detail.

5.3 Abort-Reduction Techniques

In the above sections we showed some cases where some aborts are actually avoidable. In this subsection we are going to introduce some useful techniques to reduce some unnecessary aborts. Some of these are only applicable to the WOCC variant of Sundial.

5.3.1 Multi-Versioning

As shown in Figure 5-1(a), sometimes the *wts* of the latest version is strictly greater than (*rts* + 1) of our cached version. This actually hides the information we want to know, i.e., whether it is valid to commit at the timestamp $commit\_ts$. To solve this problem, we can dynamically keep m latest versions of *rts* and *wts* of the tuple. At each time that we need to check whether $commit\_ts$ is valid for the tuple, we iterate through the latest m versions and try to find a version v with $v.wts = T.RS[r].wts$. If we successfully find such a v, we can see if $v.rts$ is greater or equal to $commit\_ts$. Then, we can decide whether to abort the transaction or not. The implementation is easy and straightforward. It does bring $O(m)$ extra computation cost, but we assume the network communication is the bottleneck and therefore local computation cost is negligible. Notice that m is a parameter which can be tuned, or computed dynamically from runtime statistics with some heuristics. Intuitively, m should be large when the average lifespan of this tuple is small, and vice versa.

In Section 6.6 we will compare Sundial with an idealized Multi-versioning concurrency control system and will show that the number of aborts that can be saved by implementing multi-versions are actually negligible.

5.3.2 Extended Leases

To prevent other transactions from writing too aggressively, besides locking, we can also choose to "reserve" a certain amount of timespan in case we need to extend the *rts*. Specifically, during the execution phase of Sundial, we can try extending r.*rts* right away if we find some of the conditions of (5.2) are not satisfied. This will help reduce the amount of aborts shown in Figure 5-1. Also, for some tuple r' even if $commit\_ts \leq r'.rts$, we can still extend $r'.rts$ to $\max(r'.rts, commit\_ts + L)$, where L is the length of the lease, a parameter which

can be pre-set empirically or dynamically tuned from heuristics and statistics. Leases can help the algorithm reduce the possibility that in the future the monotonically non-decreasing $commit\_ts$ might violate the condition (5.2) for r'.

Range Information

For the WOCC variant, in order to reduce some unnecessary aborts at the remote server, during the preparation phase, the coordinator can send a range of acceptable timestamps (t_{min}, t_{max}) instead of a single minimum timestamp. Upon receiving the range, the remote server can compute the inequalities raised by sub-transactions and update the range as $(t'_{min}, t'_{max}) \subseteq (t_{min}, t_{max})$. Then, it sends (t'_{min}, t'_{max}) back to the coordinator. After the coordinator gets all the responses from the remote servers, it can decide whether the transaction can commit or not.

Notice that for the coordinator there are two ways to produce the range. If the coordinator wants to be conservative, it only needs to report $t_{min} = \max\{\max_{r \in T.RS} r.wts, \max_{w \in T.WS} w.rts + 1\}, t_{max} = \min_{r \in T.RS} r.rts$. Any time in this range is guaranteed safe to commit. However, if the coordinator wants to be aggressive, it can report $t_{max} = \min_{r \in T.RS \wedge DB[r].wts > r.rts} r.rts$, in other words, it only needs to upper bound the timestamps with those tuples that have already been written by other transactions and therefore impossible to get extensions for. If there are no such tuples, the coordinator can simply report $t_{max} = \infty$. However, for an aggressive coordinator the available timestamp range might change when waiting for the responses from the remote servers. So it needs to utilize other techniques to ensure the validity of the range. For example, it might want to lock the tuples that have not been written by others, or set leases on these tuples. Similarly, the remote server can also choose to be conservative or aggressive.

Generally, this will help the algorithm deal with the cases shown in Figure 5-2.

5.4 Executing-Phase Pre-Abort

We can detect if there is a feasible solution when executing the transaction.

5.4.1 Aggressive Locking

As mentioned before, a brute-force way to avoid these aborts is to stop other transactions writing these tuples. We can set locks on tuples in the read set before we start validation. We called this procedure *aggressive locking*. It will surely prevent all the cases in Figure 5-1, which raises the probability of the current transaction getting committed. However, it has several drawbacks. First of all, locking tuples in the read set will prevent other transactions from accessing them, which will hurt the overall performance when the task is write-intensive. Secondly, locking read tuples will increase the probability of having deadlocks.

5.4.2 Shared Locking

Different from *aggressive locking*, we will allow *rts* extension even if some other transactions have locked the tuple. We call this kind of locking as *shared locking*. Notice that extending a tuple's *rts* will not decrease the chance to get committed for any transactions with this tuple in their read sets, though it might affect the transactions who has this tuple in their write sets by postponing their writing time. In read-intensive tasks shared locking is expected to decrease the abort rate.

We found that the WOCC variant of Sundial indeed introduced more transaction aborts than the hybrid version of Sundial in write-intensive benchmarks. The WOCC variant also increases the complexity of the whole system framework. For the sake of simplicity and efficiency, we dropped the description of the WOCC variant in our conference paper [67].

5.5 Sundial vs. MaaT

Similar to Sundial, previous concurrency control protocols have also used timestamp ranges to dynamically determine the transactions' logical commit timestamps. The technique, first proposed as *dynamic timestamp allocation* (DTA), was applied to both 2PL protocols for deadlock detection [11] and OCC protocols [14, 40, 41]. More recently, similar techniques have been applied to multi-version concurrency control protocols [46], as well as to MaaT, a

single-version distributed concurrency control protocol [47].

In all these protocols, the DBMS assigns each transaction a timestamp range (e.g., 0 to ∞) when the transaction starts. After detecting a conflict, the DBMS shrinks the timestamp ranges of transactions in conflict such that their ranges do not overlap. If a transaction's timestamp range is not empty when it commits, the DBMS can pick any timestamp (in practice, the smallest timestamp) within its range as the commit timestamp; otherwise the transaction aborts.

Timestamp-range-based protocols have one fundamental drawback: they require the DBMS to *explicitly coordinate* transactions to shrink their timestamp ranges when a conflict occurs. In a distributed setting, they incur higher overhead than Sundial.

We use MaaT [47], a distributed DTA-based concurrency control protocol, as an example to illustrate the problem. In MaaT, the DBMS assigns a transaction with the initial timestamp range of $[0, +\infty]$. The DBMS maintains the range at each server accessed by the transaction. When a transaction begins the validation phase, the DBMS determines whether the *intersection* of a transaction's timestamp ranges across servers is empty or not. To prevent other transactions from changing the validating transaction's timestamp range, the DBMS *freezes* the timestamp range at each participating server. Many transactions, however, have ranges with an upper bound of $+\infty$. Therefore, after the DBMS freezes these ranges in the prepare phase, it must abort any transaction that tries to change the frozen timestamp ranges, namely, transactions that conflict with the validating transaction. We believe this problem is fundamental because simple fixes like adapting better time-range initialization scheme, better time-range shrinking upon conflicts, or better heuristics to decide commit timestamp within the range will not solve the problem from its root.

This problem also exists in other timestamp-range-based protocols. Chapter 6 shows that these aborts degrade the performance of MaaT.

5.6 Limitations of Sundial

As discussed above, the two advantages of Sundial are (1) improving concurrency in distributed transaction processing, and (2) lightweight caching to reduce the overhead of

remote reads. Sundial also has some limitations, which we now discuss, along with potential ways to mitigate them.

First, Sundial requires extra storage to maintain the logical leases for each tuple. Although this storage overhead is negligible for large tuples, which is the case for workloads evaluated in this work, it can be significant for small tuples. One way to reduce this overhead is for the DBMS to maintain the tuples' logical leases in a separate *lease table*. The DBMS maintains leases in this table only for tuples that are actively accessed. The leases of all 'cold' tuples are represented using a single (*cold_wts*, *cold_rts*). When a transaction accesses a cold tuple, the DBMS inserts an entry for it to the lease table with its lease assigned as (*cold_wts*, *cold_rts*). When a tuple with (*wts*, *rts*) is deleted from the lease table (e.g., due to insufficient space), the DBMS updates *cold_wts* and *cold_rts* to `Max`(*cold_wts*, *wts*) and `Max`(*cold_rts*, *rts*), respectively.

The second issue is that Sundial may not deliver the best performance for partitionable workloads. Sundial does not assume whether the workload can be partitioned or not and thus does not have special optimizations for partitioning. Systems like H-Store [6] perform better in this setting. Our experiments show that if each transaction only accesses its local partition, Sundial performs 3.8× worse than a protocol optimized for partitioning. But our protocol handles distributed (i.e., multi-partition) transactions better than the H-Store approaches.

Finally, the caching mechanism in Sundial is not as effective if the remote data read by transactions is frequently updated. This means the cached data is often stale and transactions that read cached data may incur extra aborts. A more detailed discussion can be found in Section 6.4.

Chapter 6

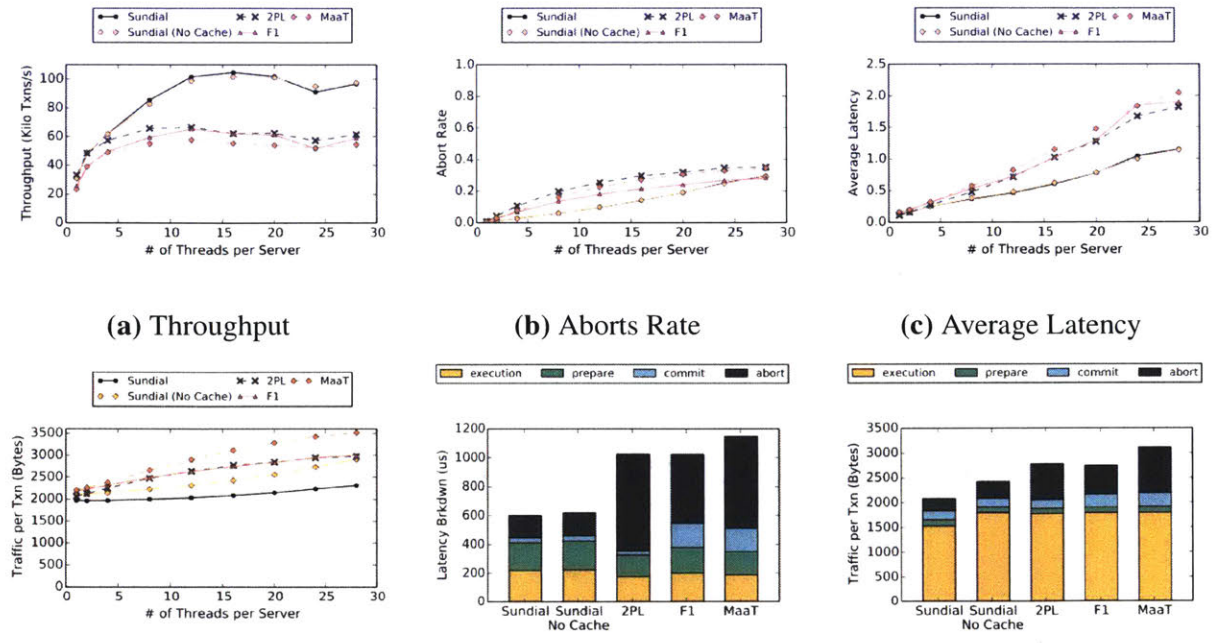
Experimental Evaluation

We now evaluate the performance of Sundial. We implemented Sundial in a distributed DBMS testbed based on the DBx1000 in-memory DBMS [64]. We have open-sourced Sundial and made it available at `https://github.com/yxymit/Sundial.git`.

Each server in the system has one input and one output thread for inter-server communication. The DBMS designates all other threads as workers that communicate with the input/output threads through asynchronous buffers. Workload drivers submit transaction requests in a blocking manner, with one open transaction per worker thread.

At runtime, the DBMS puts transactions that abort due to contention (e.g., lock acquisition failure, validation failure) into an abort buffer. It then restarts these transactions after a small back-off time randomly selected between 0–1 ms. The DBMS does not restart transactions caused by user-initiated aborts.

Most of the experiments are performed on a cluster of four servers running Ubuntu 14.04. Each server contains two Intel Xeon E5-2670 CPUs (8 cores $\times$ 2 HT) and 64 GB DRAM. The servers are connected together with a 10 Gigabit Ethernet. For the datacenter experiments in Sections 6.7 and 6.8, we use the Amazon EC2 platform. For each experiment, the DBMS runs for a warm-up period of 30s, and then results are collected for the next 30s of the run. We also ran the experiments for longer time but the performance numbers are not different from the 1 minute runs. We assume the DBMS logs to battery-backed DRAM.



(a) Throughput **(b)** Aborts Rate **(c)** Average Latency

(d) Traffic **(e)** Latency Breakdown **(f)** Traffic Breakdown

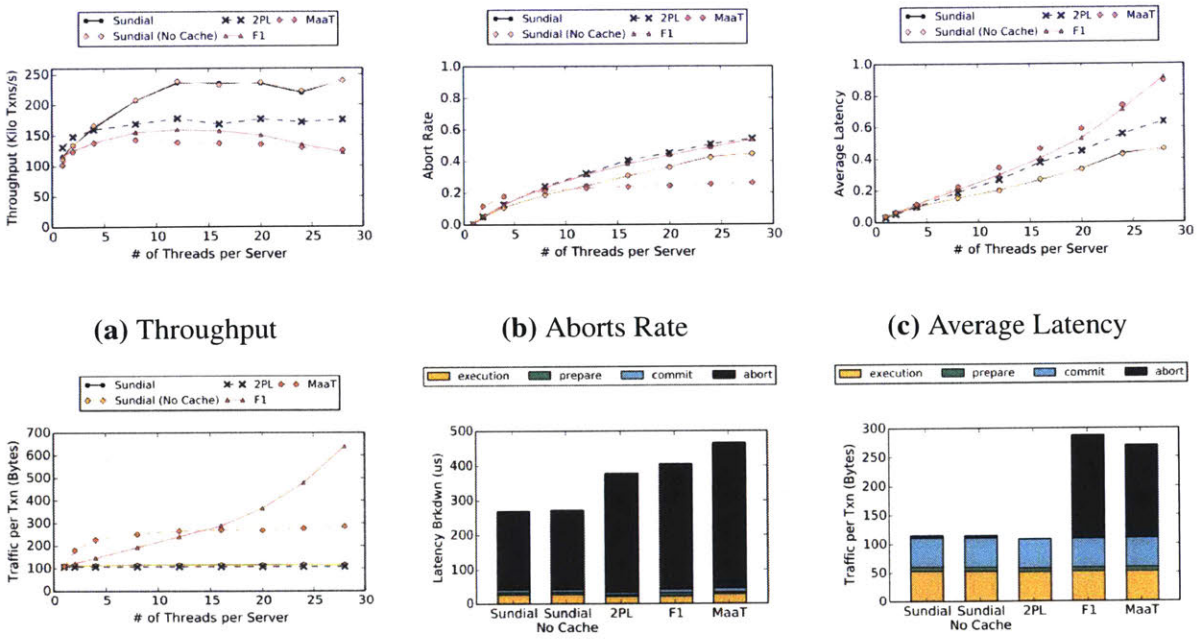
Figure 6-1: Performance Comparison (YCSB) – Runtime measurements when running the concurrency control algorithms for the YCSB workload. The throughput, abort rate, average latency and traffic is evaluated on a commodity 4-server cluster. The latency and traffic breakdown are measured on a single server in the cluster with 16 threads.

6.1 Workloads

We use two different OLTP workloads in our evaluation. All transactions execute as stored procedures that contain program logic intermixed with queries. We implement hash indexes since our workloads do not require table scans.

YCSB: The Yahoo! Cloud Serving Benchmark [17] is a synthetic benchmark modeled after cloud services. It contains a single table that is partitioned across servers in a round-robin fashion. Each partition contains 10 GB data with 1 KB tuples. Each transaction accesses 16 tuples as a mixture of reads (90%) and writes (10%) with on average 10% of the accesses being remote (selected uniformly at random). The queries access tuples following a power law distribution controlled by a parameter (θ). By default, we use θ=0.9, which means that 75% of all accesses go to 10% of hot data.

TPC-C: This is the standard benchmark for evaluating the performance of OLTP DBMSs [58]. It models a warehouse-centric order processing application that contains



(a) Throughput **(b)** Aborts Rate **(c)** Average Latency

(d) Traffic **(e)** Latency Breakdown **(f)** Traffic Breakdown

Figure 6-2: Performance Comparison (TPC-C) – Runtime measurements when running the concurrency control algorithms for the TPC-C workload. The throughput, abort rate, average latency, and traffic is measured on a commodity 4-server cluster. The latency and traffic breakdown are measured on a single server in the cluster with 16 threads.

five transaction types. All the tables except `ITEM` are partitioned based on the warehouse ID. By default, the `ITEM` table is replicated at each server. We use a single warehouse per server to model high contention. Each warehouse contains 100 MB of data. For all the five transactions, 10% of `NEW-ORDER` and 15% of `PAYMENT` transactions access data across multiple servers; other transactions access data on a single server.

6.2 Concurrency Control Algorithms

We implemented the following concurrency control algorithms in our testbed. All the codes are available online.

2PL: We used a deadlock prevention variant of 2PL called *Wait-Die* [12]. A transaction waits for a lock only if its priority is higher than the current lock owner; otherwise the DBMS will abort it. We used the current wall clock time attached with the thread id as the metric of priority. This algorithm is similar to the approach that Google Spanner [19] used

for read-write transactions.

Google F1: This is an OCC-based algorithm used in Google's F1 DBMS [53]. During the read-only execution phase, the DBMS tracks a transaction's read and write set. When the transaction begins the commit process, the DBMS locks all of the tuples accessed by the transaction. The DBMS aborts the transaction if it fails to acquire any of these locks or if the latest version of any tuple is different from the version that the transaction saw during its execution.

MaaT: This is a state-of-the-art distributed concurrency control protocol discussed in Section 5.5 [47]. We integrated the original MaaT source code into our testbed. We also improved the MaaT implementation by (1) reducing unnecessary network messages, (2) adding multi-threading support to the original single-thread-per-partition design of MaaT, and (3) improving its garbage collection.

Sundial: Our proposed protocol as described in Sections 3 and 4. We enabled all of Sundial's caching optimizations from Chapter 4 unless otherwise stated. Each server maintains a local cache of 1 GB. Sundial by default uses the hybrid caching policy.

6.3 Performance Comparison

We perform our experiments using four servers. For each workload, we report throughput as we sweep the number of worker threads from 1 to 28. After 28 threads, the DBMS's performance drops due to increased context switching. To measure the benefit of Sundial's caching scheme, We run Sundial with and without caching enabled. In addition to throughput measurements, we also provide a breakdown of transactions' latency measurements and network traffic. These metrics are divided into Sundial's three phases (i.e., execution, prepare, and commit), and when the DBMS aborts a transaction.

The results in Figs. 6-1a and 6-2a show that Sundial outperforms the best evaluated baseline algorithm (i.e., 2PL) by 57% in YCSB and 34% in TPC-C. Caching does not improve performance in these workloads in the current configuration. For YCSB, this is because the fraction of write queries per transaction is high, which means that the DBMS always sends a remote query message to the remote server even for a cache hit. As such, a

transaction has the same latency regardless of whether caching is enabled. In TPC-C, all remote requests are updates instead of reads, therefore Sundial's caching does not help.

Fig. 6-1b, Fig. 6-1c and Fig. 6-2b, Fig. 6-2c presents the variance of the abort rate and the average latency of Sundial in YCSB workload and TPCC workload, respectively, when the number of threads changes. Sundial achieves better abort rate and average transaction execution latency compared to other baselines. The only exception is that in the TPCC workload, MaaT achieves better abort rate when the number of threads per server is larger than 12. We think this is caused by the limited parallelizability of MaaT system, as we can notice the throughput of MaaT is decreasing slightly when the number of threads per server goes up. Turning on or off the cache scheme does not affect the result much (in fact, the lines representing Sundial with and without cache schemes almost always overlap with each other) in both YCSB and TPCC.

From Fig. 6-1d and Fig. 6-2d we can see that in YCSB, turning on the caching scheme saves around 23% of the total network traffic. In the TPCC benchmark, the caching scheme does not help (later we will discuss our caching scheme in TPCC workload in more details). The network traffic consumed by Sundial is small compared to that required by other baselines.

Figs. 6-1e and 6-1f show the latency and network traffic breakdown of different concurrency control protocols on YCSB at 16 threads. The `Abort` portions represent the latency and network traffic for transaction executions that later abort; this metric measures the overhead of aborts. It is clear from these results that Sundial performs the best because of fewer aborts due to its dynamic timestamp assignment for read-write conflicts. Enabling Sundial's caching scheme further reduces traffic in the execution phase because the DBMS does not need to send back data for a cache hit. Section 6.4 provides a more detailed analysis of Sundial's caching mechanism.

Another interesting observation in Fig. 6-1e is that F1 and MaaT both incur higher latency in the commit phase than 2PL and Sundial. This is because in both 2PL and Sundial, the DBMS skips the commit phase if a transaction did not modify any data on a remote server. In F1 and MaaT, however, the DBMS cannot apply this optimization because they have to either release locks (F1) or clear timestamp ranges (MaaT) in the commit phase,

	Sundial	2PL	F1	MaaT
Throughput (Kilo Txns/s)	134.26	136.57	100.80	97.95

Figure 6-3: Performance Comparison (YCSB with Low Contention) – Performance of different protocols evaluated with YCSB at low contention (90% read) and with all accesses uniformly random.

which requires a network round trip.

The latency and traffic breakdown of TPC-C (Figures 6-2e and 6-2f) show trends similar to YCSB in that Sundial achieves significant gains from reducing the cost of aborts. Since only one warehouse is modeled per server in this experiment, there is high contention on the single row in the WAREHOUSE table. As a result, all algorithms waste significant time on execution that eventually aborted. Both 2PL and Sundial incur little network traffic for aborted transactions because contention on the WAREHOUSE table happens at the beginning of each transaction. In 2PL, the DBMS resolves conflicts immediately (by letting transactions wait or abort) before sending out any remote queries. Sundial also resolves write-write conflicts early; for read-write conflicts, Sundial's logical leases allow it resolve most conflicts without having to abort transactions.

We also evaluated these protocols with YCSB at low contention (90% read, all accesses are uniformly random), and found that Sundial and 2PL have the same performance, which is around 35% better than that of F1 and MaaT. Specific results are shown in Fig. 6-3. The performance gain comes from the optimized commit protocol as discussed above.

6.4 Caching Performance

We describe the experimental results of different aspects of Sundial's caching in this section.

6.4.1 Caching with Read-Only Tables

We first measure the effectiveness of Sundial's caching scheme on databases with read-only tables. For this experiment, we use the TPC-C benchmark which contains a read-only table (i.e., ITEM) shared by all the database partitions. To avoid remote queries on ITEM, the DBMS replicates the table across all of the partitions. Table replication is a workload-specific optimization that requires extra effort from the users [50, 20]. In contrast, caching

is more general and transparent, thereby easier to use. We use two configurations for the ITEM table:

- **Replication (Rep):** The DBMS replicates the table across all the partitions, thereby all accesses to the table are local.
- **No Replication (NoRep):** The DBMS hash partitions ITEM on its primary key. A significant portion of queries on this table have to access a remote server.

For the configuration without table replication, we test two different caching configurations:

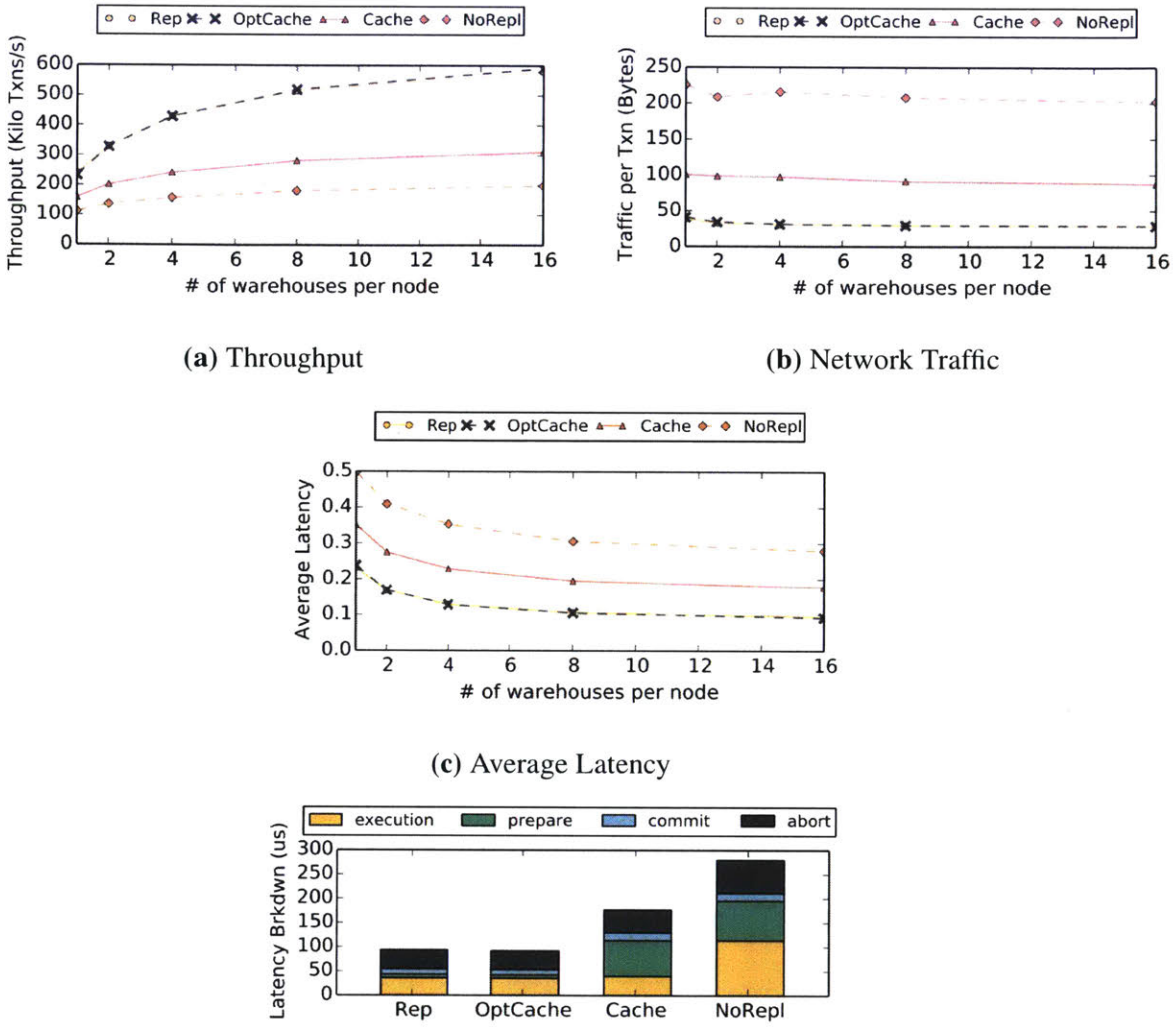
- **Default Caching (Cache):** The original caching scheme described in Section 4.1.
- **Caching with Optimizations (OptCache):** Sundial's caching scheme with the read-only optimizations from Section 4.3.

According to Fig. 6-4, the DBMS incurs a performance penalty when it does not replicate the ITEM table. This is because the table is accessed by a large fraction of transactions (i.e., all NewOrder transactions which comprise 45% of the workload) that become distributed if ITEM is not replicated. The performance gap can be closed with caching, which achieves the same performance benefit as manual table replication but hides the complexity from the users.

From Fig. 6-4, we observe that the read-only table optimizations from Section 4.3 are important for performance. Without them, a cached tuple in the ITEM table may require extra lease extensions during the prepare phase. This is because contentious tuples have rapidly increasing *wts*; transactions accessing these tuples have large *commit_ts*, leading to lease extensions on tuples in ITEM. These lease extensions increase both network delays and the amount of network traffic, which hurts performance. With the read-only table optimizations, the DBMS extends all leases in the ITEM table together which amortizes the extension cost.

6.4.2 Caching with Read-Write Tables

In this section, we measure the effectiveness of caching for read-write tables. We use a variant of the YCSB workload to model a social network application scenario. A transaction writes tuples following a uniformly random distribution and reads tuples following a power



(a) Throughput

(b) Network Traffic

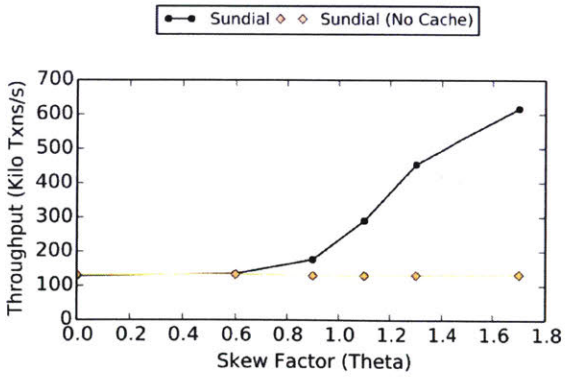
(c) Average Latency

(d) Latency Breakdown (16 warehouses per server)

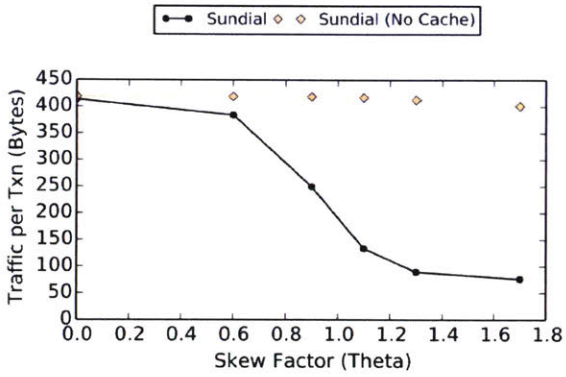
Figure 6-4: Caching with Read-Only Tables – Performance of different TPC-C configurations in Sundial with caching support. We can see that without caching the system experiences a performance loss compared to **Rep**, and this loss can be compensated with **OptCache**.

law distribution. This is similar to the social network application where the data of popular users are read much more often than data of less popular users. We sweep the skew factor (i.e., θ) for the read distribution from 0.6 to 1.7. The percentage of read queries per transaction is 90% for all trials.

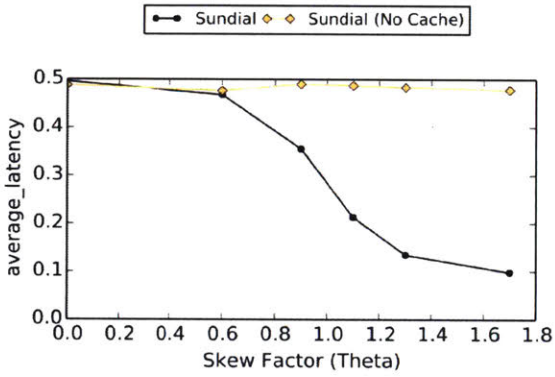
Fig. 6-5 shows the performance of Sundial with and without caching, in terms of throughput, network traffic, and latency breakdown. When the read distribution is less skewed (θ=0.6), caching does not provide much improvement because hot tuples are not read intensive enough. As the reads become more skewed, performance of Sundial with



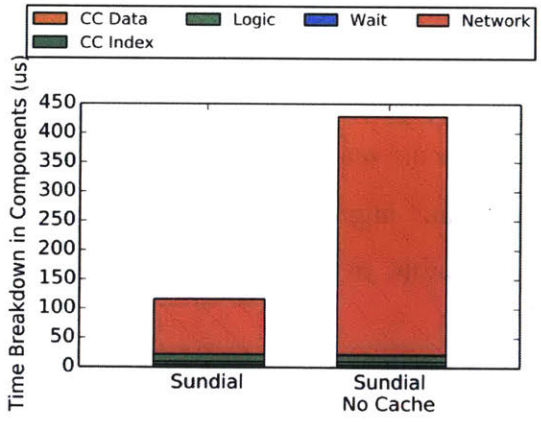
(a) Throughput



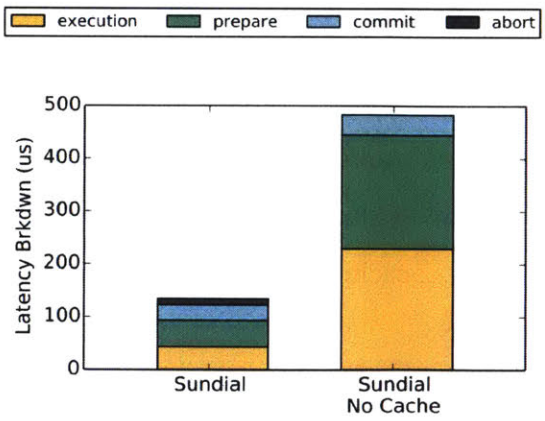
(b) Network Traffic



(c) Average Latency

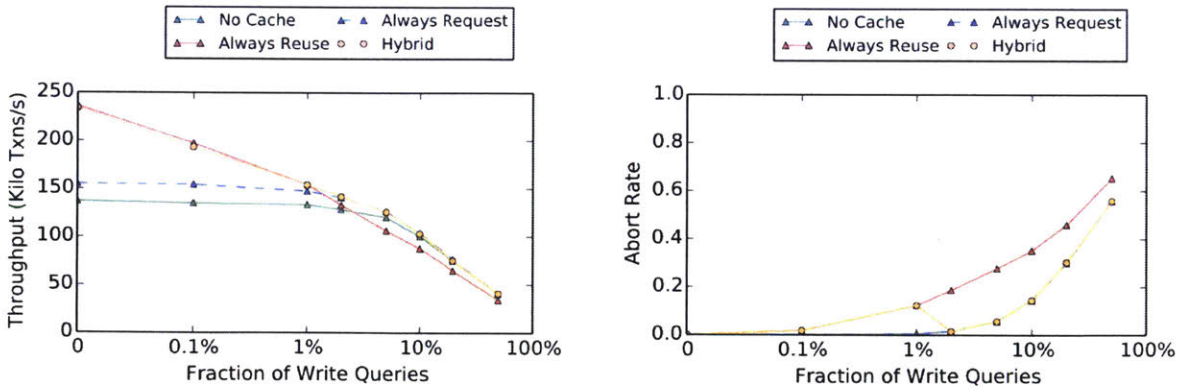


(d) Component Time Breakdown ($\theta = 1.3$)



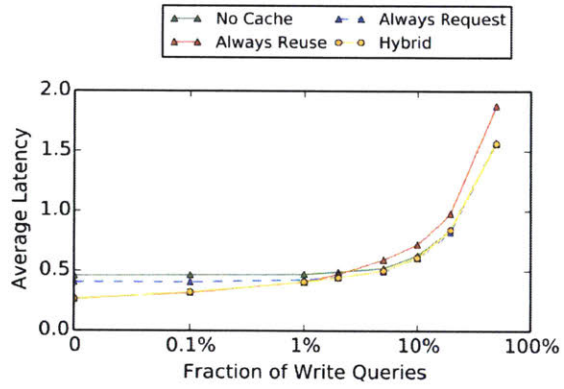
(e) Latency Breakdown ($\theta = 1.3$)

Figure 6-5: Caching with Read-Write Tables – Performance of Sundial with and without caching on YCSB as the skew factor of read distribution changes.

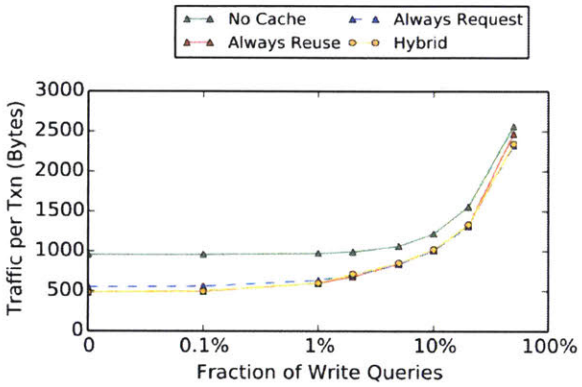


(a) Throughput

(b) Abort Rate



(c) Average Latency



(d) Network Traffic

Figure 6-6: Caching Policies – Performance measurements for Sundial with the caching policies from Section 4.2 when varying the percentage of write queries per transaction in the YCSB workload.

caching improves significantly because the hot tuples are read intensive and can be locally cached. With a high skew factor (θ=1.7), the performance improvement derived from caching is 4.6×; caching also reduces the amount of network traffic by 5.24× and transaction latency by 3.8×.

6.4.3 Cache Size

We now evaluate how sensitive the performance is to different cache sizes. Table 6.2 shows the throughput of Sundial on the same social-network-like workload as used in Section 6.4.2 with a skew factor $\theta = 1.3$.

Table 6.1: Throughput (in Kilo Txns/s) with Different Cache Sizes in Sundial.

Cache Size	0 MB	16 MB	64 MB	256 MB	1 GB	4 GB
Throughput	129	429	455	467	462	462

At this level of skew, performance is significantly improved even with a small cache size of 16 MB. Performance further increases as the cache gets bigger, until it plateaus with 256 MB caches.

6.4.4 Caching Policies

We now evaluate different caching policies in Sundial (see Section 4.2). We control the percentage of write queries in transactions. This changes whether or not the DBMS designates reading from cache as beneficial or not. For these experiments, both reads and writes follow a power law distribution with θ=0.9. We use the following caching configurations:

- **No Cache:** The Sundial's caching scheme is disabled.
- **Always Reuse:** The DBMS always reuses a cached tuple if it exists in its local cache.
- **Always Request:** The DBMS always sends a request to retrieve the tuple even if it exists in local cache.
- **Hybrid:** The DBMS reuses cached tuples for read-intensive tables only when caching is beneficial (cf. Section 4.2).

The results in Fig. 6-6 show that `Always Reuse` improves the DBMS's performance when the data is read-intensive. In this workload, the cached tuples are unlikely to be stale, thus there will be fewer unnecessary aborts. As the number of writes increases, however, many transactions abort due to reading stale cached tuples. The performance of `Always Reuse` is even worse than `No Cache` when more than 1% of queries are writes.

In contrast, `Always Request` never performs worse than `No Cache`. It has the same abort rate as no caching since a transaction always reads the latest tuples. The DBMS incurs lower network traffic than `No Cache` since cache hits on up-to-date tuples do not incur data transfer. For a read intensive table, `Always Request` performs better than `No Cache` but worse than

Abort Cases	Abort Rate
Case (a): *commit_ts* $<$ *DB*[*key*].*wts*	1.79%
Case (b): *commit_ts* $\geq$ *DB*[*key*].*wts*	1.56%
Case (c): tuple locked	6.60%
Aborts by W/W Conflicts	4.05%
Total	**14.00%**

(a) Sundial

Abort Cases	Abort Rate
Conflict with $[x, \infty)$	42.21%
Empty range due to other conflicts	4.45%
Total	**46.66%**

(b) MaaT

Figure 6-7: Measuring Aborts – Different types of aborts that occur in Sundial and MaaT for the YCSB workload. For Sundial, we classify the aborts due to read-write conflicts into the three categories from Section 5.1.

`Always Reuse`.

Lastly, the `Hybrid` policy combines the best of both worlds by adaptively choosing between `Always Reuse` and `Always Request`. This allows the DBMS to achieve the best throughput with any ratio between reads and writes.

6.5 Measuring Aborts

We designed this next experiment to better understand how transaction aborts occur in the Sundial and MaaT protocols. For this, we executed YCSB with the default configuration. We instrumented the database to record the reason why the DBMS decides to abort a transaction, i.e., due to what type of conflict. A transaction is counted multiple times in our measurements if it is aborted and restarted multiple times. To ensure that each protocol has the same amount of contention in the system, we keep the number of active transactions running during the experiment constant.

The tables in Fig. 6-7 show the percentage of transactions that the DBMS aborts out of all of the transactions executed. We see that the transaction abort rate under Sundial is 3.3$\times$ lower than that of MaaT. The main cause of aborts in MaaT is due to conflicts with the frozen range of $[x, \infty)$ where x is some constant. As discussed in Section 5.5, this happens when a transaction reads a tuple and enters the prepare phase with a timestamp range of $[x, \infty)$.

While the transaction is in this prepare phase, the DBMS has to abort any transaction that writes to the same tuple as it cannot change a frozen timestamp range. In Sundial, most of the aborts are caused by Case (c) in read-write conflicts, where a transaction tries to extend a lease that is locked by another writing transaction. There are also many aborts due to write-write conflicts. The number of aborts due to Cases (a) and (b) are about the same and lower than the other two cases.

6.6 Dynamic vs. Static Timestamps

One salient feature of Sundial is its ability to dynamically determine the commit timestamp of a transaction to minimize aborts. Some concurrency control protocols, in contrast, assign a static timestamp to each transaction when it starts and use multi-versioning to avoid aborting read queries that arrive later [25, 3, 43]. The inability to flexibly adjust transactions' commit order, however, leads to unnecessary aborts due to write conflicts from these late arriving transactions (i.e., writes arriving after a read has happened with a larger timestamp).

In this experiment, we compare Sundial without caching against a multi-version concurrency control (MVCC) protocol with varying amounts of clock skew between servers. Our MVCC implementation is idealized as it does not store or maintain the data versions, and therefore does not have associated memory and computation overhead (e.g., garbage collection) [62]. This allows us to compare the amount of concurrency enabled by Sundial and MVCC. We use `ptp` [7] to synchronize the servers' clocks and then adjust the drift from 1 to 10 ms.

In Fig. 6-8, we observe that even with no clock skew (less than 10 μs) that the DBMS performs worse with MVCC than with Sundial. This degradation is mostly caused by the extra aborts due to writes that arrive late. Sundial's dynamic timestamp assignment allows the DBMS to move these writes to a later timestamp and thus reduce these aborts. Increasing the clock skew further degrades the throughput of the MVCC protocol. This is because writes from servers that fall behind in time will always fail due to reads to the same tuples from other servers. The DBMS's performance with Sundial does not suffer with higher amounts of clock skew since its timestamps are logical.

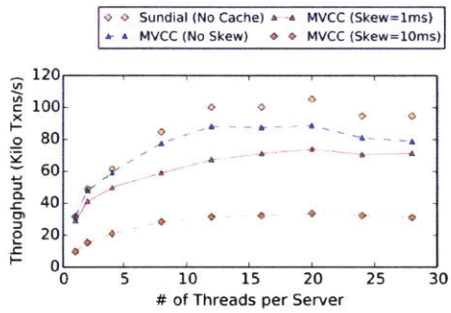


Figure 6-8: Dynamic vs. Static Timestamp Assignment – Performance comparison between Sundial and a baseline MVCC protocol that statically assigns timestamps to transactions.

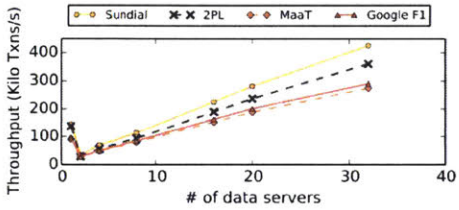


Figure 6-9: Scalability – Throughput of concurrency control algorithms for the YCSB workload on Amazon EC2.

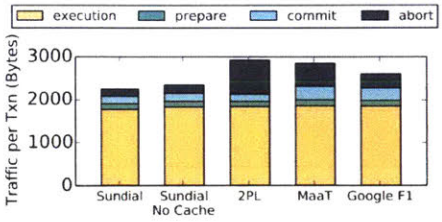
6.7 Scalability

For the final two experiments, we deployed DBx1000 on Amazon EC2 to evaluate its performance at a larger scale. We first study the scalability of the concurrency control protocols as we increase the number of servers in the cluster. Each server is an `m4.2xlarge` instances type with eight virtual threads and 32 GB main memory. We assign two threads to handle the input and output communications, and the remaining six threads as worker threads. We run the YCSB workload using the workload mixture described in Section 6.1.

The first notable result in Fig. 6-9 is that the performance of all the protocols drop to the same level when the server count increases from one to two. This is due to the overhead of the DBMS having to coordinate transactions over the network [34]. Beyond two servers, however, the performance of all of the algorithms increases as the number of servers increases. We see that the performance advantage of Sundial over the other protocols remains as the server count increases.

Protocol	Sundial	Sundial (No Cache)	2PL	MaaT	F1
Throughput	161	158	122	107	113

(a) Throughput (Txns/sec)



(b) Traffic per Transaction

Figure 6-10: Cross-Datacenter Transactions – Throughput and Traffic per Transaction when servers are placed in different data centers.

6.8 Cross-Datacenter Transactions

Lastly, we measure how Sundial performs when transactions have to span geographical regions. In the previous experiments, all of the servers are located in the same data center and therefore have low communication latency with each other. We deployed DBx1000 on Amazon EC2 in an eight-server cluster with each server located in a different datacenter[1]. We ran the YCSB workload again and compared the different concurrency control protocols.

Fig. 6-10 shows the throughput and per-transaction network traffic of the protocols in this environment. All of them suffer from much higher network latencies ($\sim$100 ms per round trip) due to cross-continent communications, which leads to three orders of magnitude reduction in performance. The average network latency is estimated as 140ms for a round trip. From the throughput we can see that Sundial performs better than the other 3 candidates. The results show that Sundial still outperforms the other baseline concurrency control protocols. Caching slightly improves the performance of Sundial due to the reduction of network traffic, which is especially expensive in this operating environment.

In Section 5.1 we discussed several cases of aborts which could be potentially avoided if we keep multiple versions of a data tuple. However, in this subsection we will see that even an idealized multi-versioning has little gain, which implies that the portion of such aborts is relatively negligible.

[1] North Virginia (us-east-1), North California (us-west-1), Ireland (eu-west-1), Frankfurt (eu-central-1), Singapore (ap-southeast-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), and Mumbai (ap-south-1).

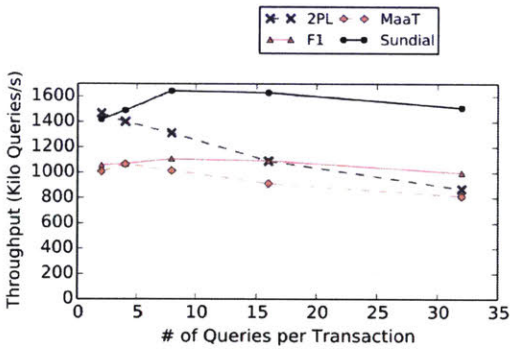


Figure 6-11: Sweep the number of queries per transaction in YCSB.

Idealized MVCC is an idealized multi-versioning concurrency control algorithm where we don't actually store previous versions but let every read on previous versions succeed. And because of this, idealized MVCC is not applicable to the TPC-C benchmark since the value fetched from previous read may affect the following writes. We only do the comparison for the YCSB benchmark. The result shows that the performance of Sundial and Idealized MVCC is at the same level with negligible difference.

6.8.1 Queries per Transaction

Fig. 6-11 shows the throughput as the number of queries changes in each transaction in the YCSB workload. Note that the y-axis shows the throughput in terms of queries per second which is different from the transactions per second metric used in other experiments. When each transaction only contains a single query, both 2PL and Sundial have good performance; 2PL performs slightly better due to lower overhead of metadata management. As the number of queries increases, contention also increases and the performance of 2PL quickly goes down, while the performance of Sundial remains at a high level.

6.9 Comparison to Dynamic Timestamp Range

In Chapter 2, we qualitatively discussed the difference between Sundial and Lomet et al. [46]. In this section, we quantitatively compare their performance and discuss why

Sundial performs better. Our implementation of [46] strictly follows procedures 1 to 3 in the paper of Lomet et al. [46]. To simplify our implementation, we made two changes to their protocol.

First, instead of implementing it in a multiversion database, we only maintained a single latest version for each tuple. For the YCSB workload that we used for the evaluation, this offers a performance upper bound of the protocol; this is similar to what we did for the idealized MVCC in Section 6.6.

Second, the original protocol in [46] requires a centralized clock which all threads have access to. There are two ways to adapt this design to a distributed environment – a centralized clock server which all transactions get the timestamp from, or synchronized distributed clocks. We picked the second design and used `ptp` [7] for clock synchronization.

We run both protocols with the YCSB benchmark – each transaction accesses 16 tuples where each access has 90% probability to be a read and 10% probability to be a write. Two different contention levels are tested – with low contention tuples are accessed with uniformly random distribution; with high contention tuples are accessed following a power law distribution with a skew factor of $\theta = 0.9$, meaning that 75% of accesses go to 10% of hot data.

Table 6.2: Performance comparison between Sundial and Lomet et al [46]

	Low Contention	High Contention
Sundial	130.3	99.6
Lomet et al. [46]	110.9	58.5

Table 6.2 shows the performance of the two protocols. At low contention, their performance is similar, with Sundial outperforming by 17.5%. In this setting, both protocols incur very few aborts. The performance improvement is mainly due to the lower cost of managing the metadata in Sundial, because there is no need to adjust the timestamp ranges for *other* transactions when conflicts occur. Compared to per-transaction timestamp ranges, the per-tuple logical leases are a more lightweight solution to dynamically determine transactions' commit timestamps.

At high contention, the performance difference between Sundial and [46] becomes a more significant 70%. In this setting, Sundial has lower abort rate and thus performs better.

One reason of this is that [46] has to decide how to shrink the timestamp ranges of conflicting transactions at the moment the conflict occurs. Failing to choose the best way of shrinking can hurt performance. Sundial, in contrast, delays the decision to the end of the transaction in an optimistic way, thereby avoiding some unnecessary aborts.

Chapter 7

Related Work

Most of the existing work on Distributed Transactional Database System improve their efficiency by graph partitioning to reduce the desired number of cross-partition transactions (such as [21] and [57]).

As a transactional distributed database system, Sundial is related to work in lots of areas.

7.1 Distributed Concurrency Control

Enforcing strong consistency (i.e., serializability) in a distributed transaction processing system is an active research area. Classic database systems incur high overheads to achieve strong consistency. NoSQL systems—exemplified by BigTable [16], Dynamo [23], and Cassandra [39]—favor weak consistency and shift the burden of writing correct concurrent programs to the programmers.

NewSQL is a class of database management systems that provide ACID transactions while trying to achieve the same level of performance of NoSQL systems. NewSQLs save the programmers from handling consistency issues and make them focus on writing transactions, which are much more natural and easy to understand. Examples of NewSQL include Clustrix [1], NuoDB [2], CockroachDB [3], Google Spanner [18], and F1 [53]. Many of these systems use 2PL as their concurrency control scheme. Some use multi-version concurrency control and rely on synchronized clocks across servers. Most of the distributed NewSQL DBMS were designed to achieve scalability by partitioning the data into parts.

And they try to make sure that every transaction only accesses tuples in a single part of data. From early distributed DBMSes like the GAMMA project from the University of Wisconsin-Madison [24] to DBMSes with heterogeneous architectures like MemSQL and NuoSQL, distributed transactional database systems achieved better performance.

Prior work has proposed many distributed concurrency control algorithms [47, 44, 26, 48, 29]. Sundial is unique in that it uses logical leases to exploit concurrency with low overhead. Among these previous algorithms, MaaT [47] is similar to Sundial in that it uses dynamic timestamp ranges for its concurrency control where each transaction starts with a range of $[0, \text{inf}]$ and adjusts its ranges when conflicts occur among transactions. One downside of MaaT is that transactions that have conflicts on the same tuples need to explicitly coordinate with each other to adjust the timestamp ranges, which increases complexity and may hurt performance. Sundial is simpler since each transaction only interacts with the tuples it accesses but not with other transactions.

Chapter 8

Future Work

We now discuss future research directions for extending Sundial to more system configurations and application scenarios.

Our evaluation of Sundial in this work used a 10 Gigabit Ethernet over standard TCP/IP network stack. Recent technology improvements significantly reduce network latency and the associated computation overhead through new software and hardware supports. Examples of them include remote direct memory access (RDMA) [28, 69] and Intel's data plane developer kit (DPDK) [5]. Leveraging these kernel bypass methods to further improve the performance of Sundial is part of our future research.

Additionally, the current implementation of Sundial also does not take high availability (HA) into account. To support HA, the DBMS must replicate data across multiple servers to ensure that if some of the servers go down, the backup servers can continue serving incoming transactions. We plan to extend the Sundial protocol to support these replicated environments. Based on the observation that Sundial efficiently supports caching and that caching is similar to data replication, we believe that logical leases can facilitate the management of data replication.

Finally, this work focuses on transactions in the form of stored procedures. We would like to study how Sundial performs with ad-hoc transactions in the future.

Chapter 9

Conclusion

We presented Sundial, a distributed concurrency control protocol that outperforms all other ones that we evaluated. Using logical leases, Sundial reduces the number of aborts due to read-write conflicts, and reduces the cost of distributed transactions by dynamically caching data from a remote server. The two techniques are seamlessly integrated into a single protocol as both are based on logical leases. Our evaluation shows that both optimizations significantly improve the performance of distributed transactions under various workload conditions.

Bibliography

[1] http://www.clustrix.com.

[2] The architecture & motivation for nuodb. http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf.

[3] CockroachDB. https://www.cockroachlabs.com.

[4] DBx1000. https://github.com/yxymit/DBx1000.

[5] Dpdk: Data plane development kit. http://dpdk.org.

[6] H-Store: A Next Generation OLTP DBMS. http://hstore.cs.brown.edu.

[7] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *Sensors for Industry Conference* (2008), pp. 1–300.

[8] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *ACM SIGMOD Record* (1995), 23–34.

[9] ANDERSON, D., AND TRODDEN, J. *Hypertransport System Architecture*. Addison-Wesley Professional, 2003.

[10] ARCHIBALD, J., AND BAER, J.-L. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *TOCS* (1986), 273–298.

[11] BAYER, R., ELHARDT, K., HEIGERT, J., AND REISER, A. Dynamic Timestamp Allocation for Transactions in Database Systems. In *DDB* (1982), pp. 9–20.

[12] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency Control in Distributed Database Systems. *CSUR* (1981), 185–221.

[13] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR* (2011), pp. 9–20.

[14] BOKSENBAUM, C., FERRIE, J., AND PONS, J.-F. Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems. *TSE* (1987), 409–419.

[15] CHANDRASEKARAN, S., AND BAMFORD, R. Shared Cache – the Future of Parallel Databases. In *ICDE* (2003), pp. 840–850.

[16] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[17] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *SoCC* (2010), pp. 143–154.

[18] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ET AL. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013).

[19] CORBETT, J. C., AND ET AL. Spanner: Google's Globally-Distributed Database. In *OSDI* (2012), pp. 251–264.

[20] CURINO, C., JONES, E., ZHANG, Y., AND MADDEN, S. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *VLDB* (2010), 48–57.

[21] CURINO, C., J. E. Z. Y., AND MADDEN, S. Schism: a workload-driven approach to database replication and partitioning. In *Proceedings of the VLDB Endowment 3* (2010), pp. 48 – 57.

[22] DAS, S., AGRAWAL, D., AND EL ABBADI, A. G-Store: a Scalable Data Store for Transactional Multi key Access in the Cloud. In *SoCC* (2010), pp. 163–174.

[23] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review 41*, 6 (2007), 205–220.

[24] DEWITT, D. J., GERBER, R., GRAEFE, G., HEYTENS, M., KUMAR, K., AND MURALIKRISHNA, M. *A High Performance Dataflow Database Machine*. Computer Science Department, University of Wisconsin, 1986.

[25] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD* (2013), pp. 1243–1254.

[26] DING, B., KOT, L., DEMERS, A., AND GEHRKE, J. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 262–275.

[27] DRAGOJEVIĆ, A., GUERRAOUI, R., AND KAPALKA, M. Stretching Transactional Memory. In *PLDI* (2009), pp. 155–165.

[28] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP* (2015), pp. 54–70.

[29] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 54–70.

[30] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The Notions of Consistency and Predicate Locks in a Database System. *CACM* (1976), 624–633.

[31] FALEIRO, J. M., AND ABADI, D. J. Rethinking Serializable Multiversion Concurrency Control. *PVLDB* (2015), 1190–1201.

[32] FRANKLIN, M. J., CAREY, M. J., AND LIVNY, M. Transactional Client-Server Cache Consistency: Alternatives and Performance. *TODS* (1997), 315–363.

[33] GRAY, C., AND CHERITON, D. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *SOSP* (1989), pp. 202–210.

[34] HARDING, R., VAN AKEN, D., PAVLO, A., AND STONEBRAKER, M. An Evaluation of Distributed Concurrency Control. *VLDB* (2017), 553–564.

[35] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[36] JOSTEN, J. W., MOHAN, C., NARANG, I., AND TENG, J. Z. DB2's Use of the Coupling Facility for Data Sharing. *IBM Systems Journal* (1997), 327–351.

[37] KELEHER, P., COX, A. L., DWARKADAS, S., AND ZWAENEPOEL, W. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX* (1994), pp. 23–36.

[38] KIM, K., WANG, T., JOHNSON, R., AND PANDIS, I. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD* (2016), pp. 1675–1687.

[39] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[40] LAM, K.-W., LAM, K.-Y., AND HUNG, S.-L. Real-Time Optimistic Concurrency Control Protocol with Dynamic Adjustment of Serialization Order. In *RTAS* (1995), pp. 174–179.

[41] LEE, J., AND SON, S. H. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *RTSS* (1993), pp. 66–75.

[42] LI, K., AND HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *TOCS* (1989), 321–359.

[43] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD* (2017), pp. 21–35.

[44] LIN, Q., CHANG, P., CHEN, G., OOI, B. C., TAN, K.-L., AND WANG, Z. Towards a non-2pc transaction management in distributed database systems. In *SIGMOD. ACM* (2016).

[45] LOMET, D., ANDERSON, R., RENGARAJAN, T. K., AND SPIRO, P. How the Rdb/VMS Data Sharing System Became Fast. Tech. rep., 1992.

[46] LOMET, D., FEKETE, A., WANG, R., AND WARD, P. Multi-Version Concurrency via Timestamp Range Conflict Management. In *ICDE* (2012), pp. 714–725.

[47] MAHMOUD, H. A., ARORA, V., NAWAB, F., AGRAWAL, D., AND EL ABBADI, A. MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud. *VLDB* (2014), 329–340.

[48] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distributed Transactions. In *OSDI* (2014), pp. 479–494.

[49] NEUMANN, T., MÜHLBAUER, T., AND KEMPER, A. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD* (2015), pp. 677–689.

[50] PAVLO, A., CURINO, C., AND ZDONIK, S. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *SIGMOD* (2012), pp. 61–72.

[51] PORTS, D. R., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI* (2010), pp. 279–292.

[52] RAHM, E. Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems. *TODS* (1993), 333–377.

[53] SHUTE, J., VINGRALEK, R., SAMWEL, B., HANDY, B., WHIPKEY, C., ROLLINS, E., OANCEA, M., LITTLEFIELD, K., MENESTRINA, D., ELLNER, S., ET AL. F1: A Distributed SQL Database that Scales. *VLDB* (2013), 1068–1079.

[54] SORIN, D. J., HILL, M. D., AND WOOD, D. A. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture* (2011), 1–212.

[55] STERN, U., AND DILL, D. L. Automatic Verification of the SCI Cache Coherence Protocol. In *CHARME* (1995).

[56] SU, C., CROOKS, N., DING, C., ALVISI, L., AND XIE, C. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 283–297.

[57] TATAROWICZ, A. L., CURINO, C., JONES, E. P., AND MADDEN, S. Lookup tables: Fine-grained partitioning for distributed databases. In *2012 IEEE 28th International Conference on Data Engineering* (2012), IEEE, pp. 102–113.

[58] THE TRANSACTION PROCESSING COUNCIL. TPC-C Benchmark (Revision 5.9.0), June 2007.

[59] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD* (2012), pp. 1–12.

[60] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-Memory Databases. In *SOSP* (2013).

[61] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *SOSP* (2015).

[62] WU, Y., ARULRAJ, J., LIN, J., XIAN, R., AND PAVLO, A. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *VLDB* (2017), 781–792.

[63] WU, Y., CHAN, C.-Y., AND TAN, K.-L. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD* (2016), pp. 1689–1704.

[64] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. pp. 209–220.

[65] YU, X., AND DEVADAS, S. Tardis: Timestamp based Coherence Algorithm for Distributed Shared Memory. In *PACT* (2015), pp. 227 – 240.

[66] YU, X., PAVLO, A., SANCHEZ, D., AND DEVADAS, S. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD* (2016), pp. 1629 – 1642.

[67] YU, X., XIA, Y., PAVLO, A., SANCHEZ, D., RUDOLPH, L., AND DEVADAS, S. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. In *Proceedings of the 44th International Conference on Very Large Data Bases* (2018), VLDB '18.

[68] YUAN, Y., WANG, K., LEE, R., DING, X., XING, J., BLANAS, S., AND ZHANG, X. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *VLDB* (2016), 504–515.

[69] ZAMANIAN, E., BINNIG, C., HARRIS, T., AND KRASKA, T. The End of a Myth: Distributed Transactions Can Scale. *VLDB* (2017), 685–696.

[70] ZIAKAS, D., BAUM, A., MADDOX, R. A., AND SAFRANEK, R. J. Intel® Quickpath Interconnect Architectural Features Supporting Scalable System Architectures. In *High Performance Interconnects (HOTI)* (2010), pp. 1–6.