Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System

Xiangyao Yu*, Yu Xia*, Andrew Pavlo♣
Daniel Sanchez♦, Larry Rudolph♠, Srinivas Devadas♦
*Massachusetts Institute of Technology, ♦Carnegie Mellon University, ♠Two Sigma Investments, LP
xyy@csail.mit.edu, yuxia@csail.mit.edu, pavlo@cs.cmu.edu
sanchez@csail.mit.edu, rudolph@csail.mit.edu, devadas@csail.mit.edu

ABSTRACT

Distributed transactions suffer from poor performance due to two major limiting factors. First, distributed transactions suffer from high latency because each of their accesses to remote data incurs a long network delay. Second, this high latency increases the likelihood of contention among distributed transactions, leading to high abort rates and low performance.

We present Sundial, an in-memory distributed optimistic concurrency control protocol that addresses these two limitations. First, to reduce the transaction abort rate, Sundial dynamically determines the logical order among transactions at runtime, based on their data access patterns. Sundial achieves this by applying logical leases to each data element, which allows the database to dynamically calculate a transaction’s logical commit timestamp. Second, to reduce the overhead of remote data accesses, Sundial allows the database to cache remote data in a server’s local main memory and maintains cache coherence. With logical leases, Sundial integrates concurrency control and cache coherence into a simple unified protocol. We evaluate Sundial against state-of-the-art distributed concurrency control protocols. Sundial outperforms the next-best protocol by up to 57% under high contention. Sundial’s caching scheme improves performance by up to 4.6× in workloads with high access skew.

1. INTRODUCTION

When the computational and storage demands of an on-line transactional processing (OLTP) database management system (DBMS) exceed the resources of a single server, organizations often turn to a distributed database management system (DBMS). Such systems split the database into disjoint subsets, called partitions, that are stored across multiple shared-nothing servers. If transactions only access data at a single server, then these systems achieve great performance [2, 3]. Performance degrades, however, when transactions access data distributed across multiple servers [38]. This happens for two reasons. First and foremost, long network delays lead to long execution time of distributed transactions, making them slower than single-partition transactions. Second, long execution time increases the likelihood of contention among transactions, leading to more aborts and performance degradation.

Recent work on improving distributed concurrency control has focused on protocol-level and hardware-level improvements. Protocol-level improvements can reduce synchronization overhead among transactions [35, 46, 22, 36] and hence the abort rates. Existing approaches, however, still limit performance due to high coordination overhead (e.g., lock blocking). Hardware-level improvements include using special hardware like optimized networks that enable low-latency remote memory accesses [20, 48, 55]. Such hardware, however, increases the overall cost of the system. Ideally, concurrency control protocols should achieve good performance on low-cost commodity hardware.

To address these issues, we introduce Sundial, an in-memory distributed concurrency control protocol that outperforms existing approaches. To address the long network latency problem, Sundial natively supports data caching in a server’s local main memory without requiring an additional protocol or sacrificing serializability, and uses cached data only when beneficial. To address the high overhead of coordinating distributed transactions, Sundial employs a hybrid pessimistic/optimistic concurrency control protocol that reduces aborts by dynamically reordering transactions that experience read-write conflicts.

The enabling technique behind Sundial is a concept called logical leases, which allow the DBMS to dynamically determine the logical order among transactions while enforcing serializability. A logical lease comprises two logical timestamps indicating the range of logical time wherein the tuple is valid. The DBMS dynamically computes a transaction’s commit timestamp, which must overlap with the leases of all tuples accessed by the transaction. Prior work has shown that logical leases are effective in improving performance and concurrency for both hardware cache coherence [52] and multicore concurrency control protocols [53]. To the best of our knowledge, this paper is the first to apply logical leases to a distributed system, and the first to seamlessly combine concurrency control and caching in a shared-nothing database.

To evaluate Sundial, we implemented it in a distributed DBMS...
writes. We consider serializability in this paper: concurrent transac-
tions have been intensively studied [28, 32, 37, 47, 49, 50, 54, 53]. Some of
these techniques can be applied to a distributed setting as well; we
incorporate some of them into our distributed concurrency control
protocols. Other techniques developed for multicore processors,
however, may not be readily used in a distributed setting due to
different system properties, such as high network latency.

In this paper, we focus on the performance of distributed concur-
rency control protocols and therefore do not compare to protocols
that are designed for single-server multicore systems.

2. Background and Related Work

Concurrency control provides two critical properties for database
transactions: atomicity and isolation. Atomicity guarantees that ei-
ther all or none of a transaction’s changes are applied to the database.
Isolation specifies when a transaction can see other transactions’
writes. We consider serializability in this paper: concurrent transac-
tions produce the same results as if they are sequentially executed.

While serializability is a robust and easy-to-understand isolation
level, concurrency control protocols that provide serializability incur
large synchronization overhead in a DBMS. This has been shown
in scalability studies of concurrency control in both multicore [51]
and distributed [25] settings. The problem is worse in a distributed
DBMS due to high network latencies between database servers.

Two classes of concurrency control protocols [9] are commonly
used: two-phase locking (2PL) and timestamp ordering (T/O).
2PL [11, 21] protocols are pessimistic: a transaction accesses a
tuple only after the transaction acquires a lock with the proper per-
mission (e.g., read or write). In contrast, T/O protocols use logical
timestamps to determine the logical commit order of transactions.
Variants of T/O protocols include Multi-Version Concurrency
Control (MVCC), which maintains multiple versions of each tuple to
reduce conflicts, and Optimistic Concurrency Control (OCC), where
transactions are executed assuming no conflicts, and conflict checks
happen after execution. OCC may incur more aborts than 2PL [51],
but has the advantage of non-blocking execution. Sundial combines
the benefits of 2PL and OCC by using them for write-write and
read-write conflicts, respectively. Sundial further reduces aborts due
to read-write conflicts through logical leases (cf. Section 3).

2.1 Distributed Concurrency Control

Recent research has proposed many distributed concurrency con-
trol protocols [15, 20, 35, 41, 48]; they are all based on variants of
2PL (such as Spanner [15] and F1 [41]) or T/O (such as MaaT [35]
and Lomet et al. [34]). Among these concurrency control protocols,
MaaT [35] is the closest to Sundial. Both MaaT and Sundial use
logical timestamps to dynamically adjust the commit order among
transactions, but each protocol uses different techniques. MaaT re-
quires all conflicting transactions to explicitly coordinate with each
other to adjust their timestamp intervals, which incurs higher per-
formance cost and more aborts. By contrast, conflicting transactions
do not coordinate in Sundial. Furthermore, Sundial uses caching to
reduce network latency, whereas MaaT does not support caching.

Lomet et al. [34] proposed a multi-version concurrency control
protocol that lazily determines a transaction’s commit timestamp
using timestamp ranges. The protocol works in both multicore and
distributed settings. Upon a conflict, the protocol requires all in-
volved transactions to shrink their timestamp ranges—an expensive
operation that Sundial avoids through logical leases. Furthermore,
their protocol requires a multi-version database while Sundial works
in a single-version database. The Appendix compares the perfor-
mance of Lomet et al.’s protocol and Sundial.

2.2 Multicore Concurrency Control

Fast transaction processing on single-server multicore systems
has been intensively studied [28, 32, 37, 47, 49, 50, 54, 53]. Some of
these techniques can be applied to a distributed setting as well; we
incorporate some of them into our distributed concurrency control
protocols. Other techniques developed for multicore processors,
however, may not be readily used in a distributed setting due to
different system properties, such as high network latency.

In this paper, we focus on the performance of distributed concur-
rency control protocols and therefore do not compare to protocols
that are designed for single-server multicore systems.

2.3 Data Replication and Cache Coherence

Data replication [16, 38] has been previously proposed to reduce
the overhead of distributed transactions. Replication is particularly
effective for hot (i.e., frequently accessed) read-only data. When
replicated across servers, hot data can serve read requests locally,
significantly reducing remote requests and their network overheads.

Data replication, however, has several drawbacks. First, the user
doing high-stake tasks needs to either profile the workloads or manually
specify what data to replicate—a daunting task in a rapidly evolving
database. Second, when the DBMS updates data that is replicated,
all servers holding replicas need to be notified of the update to
maintain consistency, which adds complexity and overhead. Third,
full-table replication increases memory footprint, which is problem-
atic if the replicated table is large.

Caching is a more flexible solution to leverage hot data. Specif-
ically, the DBMS decides to cache remote data in a server’s lo-
cal memory without user intervention. A query that reads data
from its local cache does not contact the remote server, thus
reducing both network latency and traffic. The key challenge in a
caching protocol is to maintain cache coherence (i.e., to avoid read-
ing stale cached data). Cache coherence is a classic problem in
parallel computing and much prior work exists in different research
fields including multicore and multi-socket processors [6, 42, 56],
distributed shared memory systems [27, 31], distributed file sys-
tems [24], and databases [5, 23, 39]. In traditional DBMSs, caching
and concurrency control are two separate layers; and both of them
are notoriously hard to design and to verify [43]. In Section 4, we

demonstrate how Sundial merges concurrency control and cache
coherence into a single lightweight protocol with the help of logical
leases.

2.4 Integrating Concurrency Control and
Cache Coherence

The integration of concurrency control and cache coherence have
also been studied in the context of data sharing systems [40]. Ex-
amples include IBM DB2 [26], Oracle RAC [13], Oracle Rdb [33],
and the more recent Microsoft Hydra [10]. In these systems, servers
in the cluster share the storage (i.e., shared disk architecture). Each
server can access all the data in the database, and may cache recently
accessed data in a local buffer. A cache coherence protocol ensures
the freshness of data in caches.

Sundial is different from data sharing systems in two aspects.
First, Sundial is built on top of a shared-nothing architecture, and
thus scales better than a data sharing system, which uses a shared-
disk architecture. Second, existing data sharing systems typically
use 2PL for concurrency control; the coherence protocol is also
based on locking. By contrast, Sundial uses logical leases for both
concurrency control and cache coherence, which reduces protocol
complexity and increases concurrency (Section 4).

Another related system is G-Store [17], which supports transac-
tional multi-key access over a key-value store. Before a transaction
starts, the server initiates a grouping protocol to collect the exclusive
ownership of keys that the transaction will access; after the trans-
action completes, the ownership is returned back. The grouping
protocol has two downsides: (1) it requires the DBMS to know what keys a transaction will access before executing the transaction, and (2) it does not allow multiple transactions to read the same key concurrently. These problems do not exist in Sundial, which supports dynamic working sets and concurrent reads.

3. SUNDIAL CONCURRENCY CONTROL

This section presents the Sundial distributed concurrency control protocol in detail. Our discussion assumes that the database is deployed over a homogeneous cluster of servers connected through a Local Area Network (LAN). Data is partitioned across servers. Each tuple is mapped to a home server that can initiate a transaction on behalf of an external user, as well as process remote queries on behalf of peer servers. The server initiating the transaction is called the coordinator of the transaction; other servers involved in a transaction are called the participants of the transaction.

3.1 Logical Leases

Logical leases are based on logical time, which specifies a partial logical order among concurrent operations in a DBMS. A logical lease is associated with each data element. It is represented using two 64-bit timestamps, wts and rts. wts is the logical time when the tuple was last written; rts is the end of the lease, meaning that the tuple can be read at any logical time ts such that wts ≤ ts ≤ rts. A transaction can write to a tuple only after the current lease expires, namely, at a timestamp no less than rts + 1. Since leases are logical in Sundial, the writing transaction does not need to ‘wait’ for the lease to expire in physical time—it simply jumps ahead in logical time to the end of the lease.

To achieve serializability, the DBMS computes a single commit timestamp for a transaction T, which is a logical time that falls within the leases of all tuples the transaction accesses. Namely, for all tuples accessed by T, tuple.wts ≤ T.commit_ts ≤ tuple.rts. From the perspective of logical time, T’s operations are atomically performed at the commit timestamp. The DBMS calculates this commit timestamp using only the leases of tuples accessed by the transaction—no inter-transaction coordination is required.

Figure 1 shows an example illustrating the high-level idea of Sundial concurrency control. The DBMS executes two transactions, T1 and T2. T1 reads tuples A and B with logical leases [0, 1] and [1, 2], respectively; T1 also writes to B and creates a new lease of [1, 1] for the new data. T1 commits at timestamp 1 as it overlaps with all the leases that T1 accesses.

When T2 writes to A, it creates a new lease for A of [2, 2]. T2 also reads C at [3, 3]. These two leases, however, do not overlap. In this case, the DBMS extends the end of the lease on A from 2 to 3 such that both operations performed by T2 are valid at timestamp 3. If the lease extension succeeds, namely, no other transaction in the system has written to A at timestamp 3, the DBMS commits T2 at timestamp 3.

Note that in this example, when T1 commits, T2 has already modified A, which was read by T1. However, this does not cause T1 to abort as in traditional OCC protocols. This is because Sundial serializes transactions in logical rather than physical time order, so the physical and logical commit orders can differ, as shown in the example—although T1 commits after T2 in physical time order, T1 commits with a smaller logical timestamp. This time traveling feature of Sundial allows the DBMS to dynamically determine a transaction’s commit timestamp, which avoids some unnecessary aborts due to read-write conflicts and results in lower abort rate.

3.2 Conflict Handling

Sundial uses different methods to resolve different types of conflicts between transactions. Such hybrid schemes have also been proposed before in both databases [18, 44] and software transactional memory [19].

Sundial handles write-write conflicts using pessimistic 2PL. We made this choice because many writes are read-modify-writes, and two such updates to the same tuple always conflict (unless we exploit semantic information like commutativity); handling such conflicts using OCC causes all but one of the conflicting transactions to abort.

In contrast, Sundial handles read-write conflicts using OCC instead of 2PL to achieve higher concurrency. This prevents transactions from waiting for locks during execution. More importantly, this allows the DBMS to dynamically adjust the commit order of transactions to reduce aborts, using the technique of logical leases discussed in Section 3.1.

Figure 2 shows an example to illustrate the difference between 2PL, traditional OCC, and Sundial in handling read-write conflicts. In 2PL, the DBMS locks a tuple before transaction T1 reads it. This lock blocks transaction T2 from writing to the same tuple, which increases the execution time of T2.

In a traditional OCC protocol, a transaction does not hold locks during its normal execution. Therefore, T2 commits before T1 and updates tuple A in the database. When T1 tries to commit, A has been changed since the last time T1 reads it. The DBMS then aborts T1 due to this data conflict.

In this example, aborting T1 is unnecessary because there exists a logical commit order (i.e., T1 commits before T2) between the two transactions that maintains serializability. Sundial is able to identify this order using logical leases. Specifically, T1 reads a version of A with a lease [0, 18], it commits at a timestamp within that lease (e.g., timestamp 5). T2 writes to A with a new lease of [11, 11], and therefore commits at timestamp 11. Although T1 made the commit
3.3 Protocol Phases

The lifecycle of a distributed transaction in Sundial is shown in Figure 3, which consists of three phases: (1) execution, (2) prepare, and (3) commit. The coordinator server initiates a new transaction after receiving a request from a user application. In the first phase (execution), the coordinator executes the transaction’s logic and sends requests to participants of the transaction. When the transaction completes the execution phase, the coordinator begins two-phase commit (2PC). In the prepare phase, the first phase in 2PC, the coordinator sends a prepare request to each participant involved in the transaction (i.e., the servers that were accessed during the execution phase). Each participant responds with either OK or ABORT. If all participants (including the coordinator) agree that the transaction can commit, then the transaction enters the final (commit) phase, where the coordinator sends a message to each participant to complete the transaction. Otherwise, the DBMS aborts the transaction and rolls back its modifications. To ensure that a transaction’s modifications persist after a server/network failure, the DBMS performs logging at each server at different points during the 2PC protocol (shown as * in Figure 3).

In Sundial, the wts of a lease changes only when the tuple is updated during the commit phase of a transaction; the rts of a lease changes only in the prepare or commit phases. Both the wts and rts of a tuple only increase but never decrease. During the three phases, the DBMS may coordinate the participants to calculate a transaction’s commit timestamp or to extend logical leases. The DBMS piggybacks timestamp-related information in its normal runtime messages. We now describe the logic of each phase in more detail.

3.3.1 Execution Phase

During the execution phase, a transaction reads and writes tuples in the database and performs computation. Each tuple contains a logical lease (i.e., wts and rts). Sundial handles write-write conflicts using the 2PL Wait-Die algorithm [9]. Therefore, the DBMS also maintains the current lock owner and a waitlist of transactions waiting for the lock. Each tuple has the following format, where DB is the database and DB[key] represents the tuple that has key as its primary key:

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

Sundial handles read-write conflicts using OCC, and thereby the DBMS maintains the read set (RS) and the write set (WS) of each transaction. The data structure of the two sets are shown below. The data field contains a local copy of the database tuple that the transaction reads (in RS or WS) or writes (in WS).

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

Algorithm 1 shows the logic for read and write requests. The code shadowed in gray is related to caching and will be explained in Section 4.

For a read request, if the requested tuple is already in transaction T’s read or write set, the DBMS simply returns the data (lines 2–5). Otherwise, the DBMS sends a remote procedure call (RPCn) to the home server n of the tuple, which atomically reads the wts, rts, and data of the tuple and returns them to the coordinator’s read set (lines 10–12). The coordinator then updates T.commit_ts to be at least the wts of the tuple (line 14), reflecting that T’s logical commit timestamp must be no earlier than the logical time when the tuple was last written. Finally, the data is returned (line 15).

For a write operation, if the tuple is already in T’s write set, the DBMS simply updates the local data in the write set (line 17). Otherwise, the DBMS locks the tuple by issuing an RPC to its home server (lines 18–20). The RPC returns whether the lock acquisition succeeds, and if so, the wts and rts of the locked tuple (success, wts, rts) = RPCn.lock(key) (lines 21–22). If the lock fails, or if the locked key exists in the read set but the returned wts does not match the wts in the read set, then the transaction aborts (line 22). Otherwise, the DBMS advances the transaction’s commit_ts to rts + 1 (line 24), and updates the data in the write set (line 25). Note that during the execution phase of T, other transactions cannot read tuples in T’s write set—they become visible only after T commits (Section 3.3.3).

An important feature of Sundial is that a transaction reading a tuple does not block another transaction writing the same tuple (and
vice versa). If a tuple is locked, a reading transaction still reads the data of the tuple, with its associated logical lease. The commit timestamp of the reading transaction will be smaller than the commit timestamp of the transaction holding the lock. But both transactions are able to commit as long as they both find commit timestamps satisfying their leases.

### 3.3.2 Prepare Phase

The goal of the prepare phase is for the DBMS to determine whether all the reads and writes of a transaction are valid at the calculated commit_ts. Algorithm 2 shows the logic that the DBMS performs during the prepare phase; validate_read_set() is executed at the coordinator, which calls renew_lease() at the participants.

Since all tuples in the write set are already locked, the DBMS only validates tuples that are read but not written. For each key in transaction T’s read set but not the write set (line 2), if T.commit_ts is within the lease of the tuple (i.e., tuple.ws ≤ T.commit_ts ≤ tuple.rts), then the read is already valid and nothing needs to be done (line 3). Otherwise, an RPC is sent to the tuple’s home server to renew its lease (lines 4–6). The transaction aborts if the lease renewal fails (line 9).

A participant executes renew_lease() to perform lease renewal. If the current ws of the tuple is different from the ws observed by the transaction during the execution phase, or if the extension is required but the tuple is locked by a different transaction, then the DBMS cannot extend the lease and ABORT is returned to the coordinator (lines 13–14). Otherwise, the DBMS extends the lease by updating the tuple’s rts to at least commit_ts (line 16). In Sundial, lease extension is the only way to change a tuple’s rts. Both ws and rts of each tuple can only increase but never decrease.

### 3.3.3 Commit Phase

During the commit phase, the DBMS either commits the transaction by copying all the writes from the write set to the database and releasing the locks, or aborts the transaction by simply releasing the locks. Algorithm 3 shows the functions that the coordinator executes. If a lock acquisition fails in the execution phase, or if the

1 Note that tuple.ws ≤ T.commit_ts must be true due to the way T.commit_ts was updated during the execution phase (see Algorithm 1, lines 14 and 24).
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 protocol (2PC), which requires the coordinator and the participants to log to persistent storage before sending out certain messages (Figure 4). 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 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 1: Serializability violation when logical leases are not logged – Tuples A and B are stored in servers 1 and 2, respectively

\[ T1 \text{ R}(A) \text{ lease } \[0, 9] \]
\[ T2 \text{ W}(A), W(B), \text{ commit } @ \text{ TS} = 10 \]
\[ \text{Server 2 crashes} \]
\[ \text{Server 2 recovers} \]
\[ T1 \text{ R}(B) \text{ 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 time 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 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 it is logged only when the maximum lease exceeds the last logged UT.

### 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 [7]. 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.

Figure 5 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 this decision depends on the workloads.

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.

#### 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

---

**Figure 5: Caching Architecture** – The cache is at the network side, containing multiple banks with LRU replacement.
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, but 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 speculate on 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.

### 5. DISCUSSION

We now discuss some other aspects of Sundial’s concurrency control and caching protocols described in the previous sections.

#### 5.1 Transaction Aborts

As described in Section 3.2, it is difficult for a DBMS to avoid transaction aborts due to write-write conflicts. Therefore, we focus our discussion on aborts caused by read-write conflicts.

In Sundial, a transaction T must satisfy the following two conditions in order to commit:

\[
T\text{.commit}_{\text{ts}} \leq \text{tuple.wts}, \quad \forall \text{tuple} \in T.RS \cup T.WS \quad (1)
\]

\[
T\text{.commit}_{\text{ts}} \geq \text{tuple.rts}, \quad \forall \text{tuple} \in T.RS \quad (2)
\]

Condition (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. Therefore, during the prepare phase, the DBMS can abort a transaction only if it fails Condition (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 (2), which Figure 6 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[tabkey].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 [53].

**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. We defer the exploration of these optimizations in Sundial to future work.

5.2 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 [8] and OCC protocols [12, 29, 30]. More recently, similar techniques have been applied to multi-version concurrency control protocols [34], as well as to MaaT, a single-version distributed concurrency control protocol [35].

In all these protocols, the DBMS assigns each transaction a timestamp range (e.g., 0 to \( +\infty \)) 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 [35], 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. This problem also exists in other timestamp-range-based protocols. Section 6 shows that these aborts degrade the performance of MaaT.

5.3 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 paper, 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 \((\text{cold}_wts, \text{cold}_rts)\). When a transaction accesses a cold tuple, the DBMS inserts an entry for it to the lease table with its lease assigned as \((\text{cold}_wts, \text{cold}_rts)\). When a tuple with \((wts, rts)\) is deleted from the lease table (e.g., due to insufficient space), the DBMS updates \(\text{cold}_wts\) and \(\text{cold}_rts\) to \(\max(\text{cold}_wts, wts)\) and \(\max(\text{cold}_rts, rts)\), respectively.

The second issue is that Sundial may not deliver the best performance for partitionable workloads. Sundial does not assume that the workload can be partitioned and thus does not have special optimizations for partitioning. Systems like H-Store [2] 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.

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 [51]. We have open-sourced Sundial and made it available at https://github.com/yxyitm/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 × 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.

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 [14] 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 \(\theta\). By default, we use \(\theta=0.9\), 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 [45]. It models a warehouse-centric order processing application that contains 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 [9]. 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 [15] used for read-write transactions.

Google F1: This is an OCC-based algorithm used in Google’s F1 DBMS [41]. 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 5.2 [35]. 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 Sundail’s caching optimizations from Section 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 Figures 7a and 8a 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.

Figures 7b and 7c 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 Figure 7b 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, which requires a network round trip.

The latency and traffic breakdown of TPC-C (Figures 8b and 8c) 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.

Although not shown due to limited space, 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 30% better than that of F1 and MaaT. 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 subsection.

#### 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 [38, 16]. 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 Figure 9, 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 Figure 9, 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_trs, 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 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 30% for all trials.

Figure 10 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 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

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Throughput (in Kilo Txns/s) with Different Cache Sizes in Sundial.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>0 MB</td>
</tr>
<tr>
<td>Throughput</td>
<td>129</td>
</tr>
</tbody>
</table>

We now evaluate how sensitive the performance is to different cache sizes. Table 2 shows the throughput of Sundial on the same social-network-like workload as used in Section 6.4.2 with a skew factor θ = 1.3.

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
This allows the DBMS to achieve the best throughput with any ratio – Performance of Sundial with and without caching on YCSB as the skew factor of read distribution changes.

The results in Figure 11 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 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.

#### 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 [1, 18, 32]. 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) [49]. This allows us to compare the amount of concurrency enabled by Sundial and MVCC. We use ptp [4] to synchronize the servers’ clocks and then adjust the drift from 1 to 10 ms.

In Figure 13, we observe 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.

### 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 instance 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 again and compared the different concurrency control protocols.

Figure 14 shows the throughput and per-transaction network traffic of the protocols in this environment. All of them suffer from much higher network latencies (≈100 ms per round trip) due to cross-continent communications, which leads to three orders of magnitude reduction in performance. 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.

### 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 data center. We ran the YCSB workload again and compared the different concurrency control protocols.

Figure 15 shows the throughput and per-transaction network traffic of the protocols in this environment. All of them suffer from much higher network latencies (≈100 ms per round trip) due to cross-continent communications, which leads to three orders of magnitude reduction in performance. 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.

### 7. 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.

### 8. ACKNOWLEDGEMENTS

This work was supported (in part) by the U.S. National Science Foundation (CCF-1438955, and CCF-1438967).

May his tortured soul rest in peace.

---

2North 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).
9. REFERENCES

High Contention

58.5
99.6
130.3

[54x232]
[55]
Z

UAN
[54x275]
[54]
Y

HOMSON
[54x544]
[46]
T

W

T

U

TU

Y

W

U

S

U


APPENDIX

Comparison to Dynamic Timestamp Range

In Section 2, we qualitatively discussed the difference between Sundial and Lomet et al [34]. In this Appendix, we quantitatively compare their performance and discuss why Sundial performs better. Our implementation of [34] strictly follows procedures 1 to 3 in Appendix A in the paper. To simplify our implementation, we made two changes of 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 [34] 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 [4] 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 2: Performance comparison between Sundial and Lomet et al [34]

<table>
<thead>
<tr>
<th></th>
<th>Low Contention</th>
<th>High Contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sundial</td>
<td>130.3</td>
<td>99.6</td>
</tr>
<tr>
<td>Lomet et al. [34]</td>
<td>110.9</td>
<td>58.5</td>
</tr>
</tbody>
</table>

Table 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 [34] becomes a more significant 70%. In this setting, Sundial has lower abort rate and thus performs better. One reason of this is that [34] 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, delay the decision to the end of the transaction in an optimistic way, thereby avoiding some unnecessary aborts.