DOI:10.1145/3181853

Deterministic database systems show great promise, but their deployment may require changes in the way developers interact with the database.

BY DANIEL J. ABADI AND JOSE M. FALEIRO

An Overview of Deterministic Database Systems

FOR DECADES, THE strongest guarantee that database systems users could expect from their system was "serializability." This guarantee ensured that even though the system would process many transactions concurrently, the final state of the database system would be equivalent to as if it had processed all transactions serially, one after another. However, the database system would make no guarantee about which serial order that processed transactions would be equivalent to—arbitrary nondeterministic events such as operating system thread scheduling, deadlock, or server failure could change this equivalent serial order.

Research from the past decade has discovered a number of advantages to architecting database systems with a stronger set of guarantees. Instead of promising equivalence to any arbitrary serial order, the system instead guarantees equivalence to processing transactions in a single predetermined serial order. Furthermore, there is only one possible state the system may end up in, despite the presence of potential nondeterministic code in transaction logic. Research has shown many benefits from this stronger set of guarantees, from simpler and higher performance database replication, to improved system scalability, to removing distributed commit protocols. Furthermore, while the early work assumed the increased set of guarantees would decrease the ability of the system to process transactions concurrently, more recent research has shown the total opposite result: transaction concurrency has increased. This article describes these benefits of deterministic database systems in more detail, along with a discussion of the primary disadvantages: the lack of support of interactive transactions in the system, and the need for transaction preprocessing prior to execution.

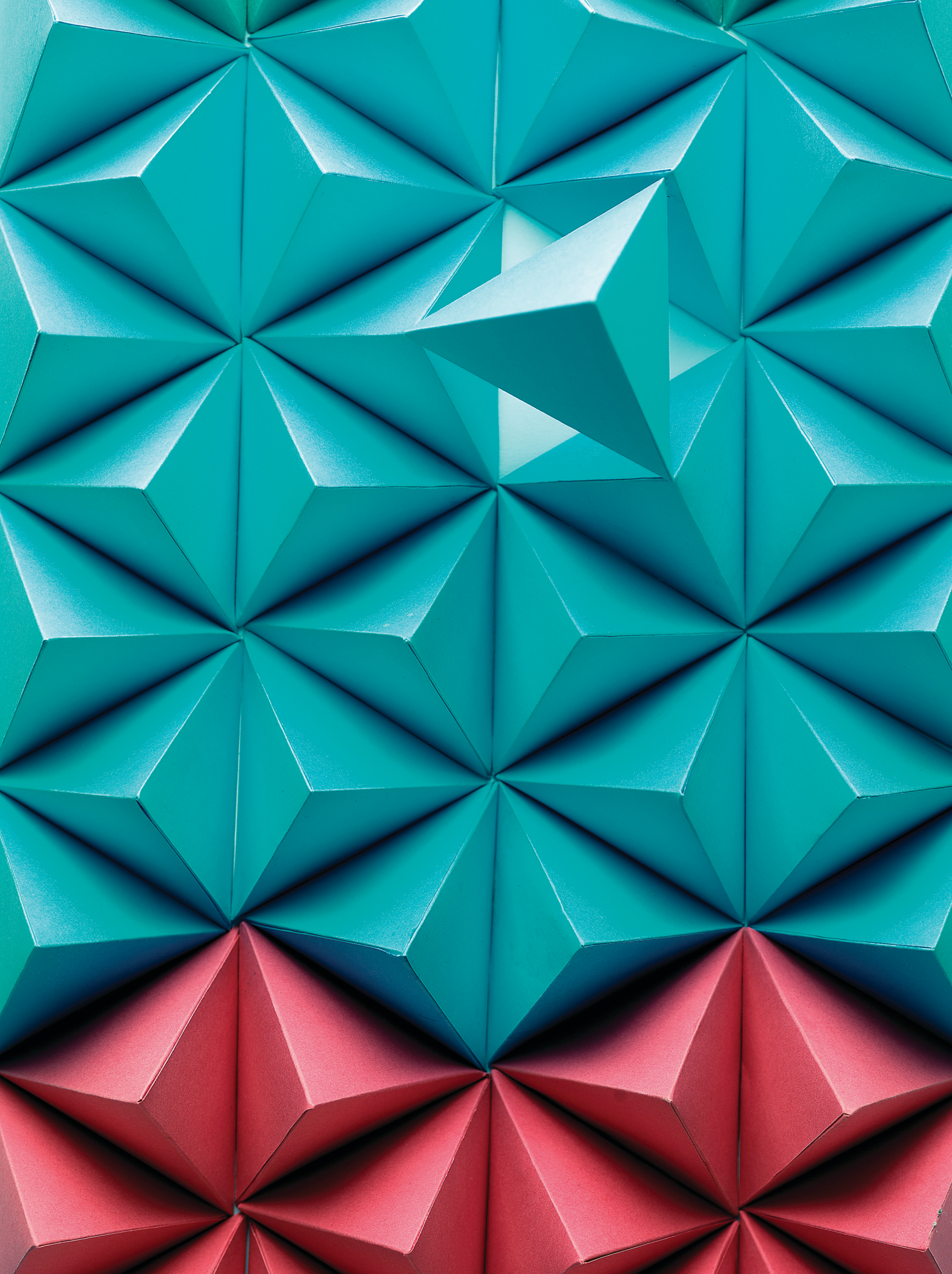
Principles and Properties

Modern deterministic database systems are built on top of nondeterministic operating systems, nondeterministic networking, and machines that may fail in arbitrary ways. Nonetheless,

» key insights

- **Every major commercial database system available today is nondeterministic. This has led to headaches for practitioners concerning database replication, scale-out, and concurrency.**
- **Deterministic database systems make database replication trivial—the same input is sent to every replica and they are guaranteed not to diverge. By reducing the cost of replication, they facilitate higher replica consistency levels (albeit constrained by CAP trade-offs) and provide a high-performance, scalable, but strongly consistent alternative to NoSQL systems.**
- **Deterministic database systems have shown promise to remove expensive commit protocols in scalable distributed deployments, and enable higher amounts of transactional throughput and concurrency.**

IMAGE BY DAN COMANICIU



given an initial state and a defined input to the system, they must end up in only one possible final state. They therefore must be architected according to the following principles:

▸ *Input preprocessing.* Existing nondeterministic database systems are typically architected such that a client communications layer receives transactional input from clients and hands them directly to the database system execution layer for processing. Usually, different communications threads will work independently from each other, receiving transactions from different clients and passing them to execution threads. This architecture is not viable for deterministic database systems because deterministic guarantees can only be made when there is a clear, universally agreed-upon input. When multiple threads are receiving input without any coordination with each other, there is no systemwide agreement on input.

Therefore, some component of the system must create a canonical record of the input to the system. On a single-machine architecture, this is often done by feeding all transactions through a single thread that records the input transactions in the order it observes them. This thread sits between the client communications threads and the database execution threads. At the other extreme, geo-replicated, highly scalable, shared-nothing systems such as Calvin[33,34] implement a distributed, replicated append-only log via Paxos, and all transactions are fed through this log before processing.

The preprocessing layer also replaces nondeterministic code inside transaction logic with deterministic code. For example, code that makes system calls to get the current time or to generate a random number must be executed by the preprocessor and be replaced by a fixed value. Note that this recording of input and pre-execution of nondeterministic code assumes that the preprocessing layer has access to the entire transaction logic prior to execution. We will discuss this assumption and corresponding limitation of deterministic database systems later.

▸ *Nondeterministic failures do not cause transaction failure.* Widely used database system recovery protocols such as ARIES[20] abort all in-process transactions upon a server failure. Since server failures are fundamentally nondeterministic events, deterministic database systems cannot allow the commit status of transactions (and their effect on the final state of the database system) to be affected by such nondeterministic events. Thus, deterministic database systems typically recover by recreating state at the time of a nondeterministic failure, and continuing all in-process transactions from that point instead of aborting them and restarting them later.

▸ *Thread race conditions cannot affect database state.* Concurrent transactions in a database system typically run in different threads or processes that are scheduled and managed by the operating system. Since the operating system schedules threads in a fundamentally nondeterministic way, many types of race conditions that affect database state are present in traditional database systems. For example, in a retail application where two transactions simultaneously attempt to purchase the same last item in an inventory, only one of them can succeed. In traditional pessimistic, locking-based systems, if these two transactions are running in different threads, whichever thread requests the lock on the inventory item first will be the one to succeed. However, this is entirely dependent on how the OS schedules the competing threads. In traditional optimistic systems, it is the thread that starts the validation phase first that will succeed. This again is entirely dependent on OS thread scheduling. Such race conditions cannot be present in deterministic database systems.

▸ *Deadlocks cannot occur in the system.* Deadlocks are typically the outcome of nondeterministic race conditions, and are resolved through aborting at least one of the deadlocked transactions. As noted, deterministic databases systems cannot allow nondeterministic events to lead to transaction failure. Therefore, either the deterministic database system must use non-locking concurrency control protocols, or they must use deadlock avoidance techniques.

▸ *Recovery from the input log.* Database state recovery is typically much simpler in deterministic database systems relative to traditional nondeterministic database systems. Deterministic database systems can simply load a checkpoint[23] and play the input log forward from there in order to recover state at the time of the failure (and continue from there).[19,26,31] In contrast, the input request log is not sufficient to recover state in non-deterministic systems. Instead, they have to replay history, which typically involves physically reloading every page that was modified from a "redo log."[20]

▸ *Do not rely on the OS for help enforcing determinism.* Enforcing determinism at the level of abstraction of the database system avoids overheads associated with determinism at the level of an operating system or language runtime. Lower levels of abstraction cannot exploit application-specific semantics to enforce determinism, and must therefore impose more severe restrictions during processing (such as deterministic mutex acquisition) that are expensive and unnecessary. In contrast, deterministic database systems can exploit the application-specific fact that the order of non-conflicting transactions does not need to be constrained, even if they acquire the same mutex(es) internally, because reordering them does not affect externally visible state.

Implementation Techniques

Despite the complexity of building a deterministic system on top of nondeterministic components, there are many different ways to implement deterministic database systems. The easiest solution is to not support concurrent transaction execution.[28] This eliminates any nondeterminism arising from OS thread scheduling, and other sources of nondeterminism can be side-stepped using the techniques described earlier (for example, replace nondeterministic code in the preprocessor or recreate transactional state after a hardware failure). Unfortunately, not supporting concurrency would result in extremely poor transactional throughput and scalability. Therefore, this section discusses three approaches to supporting concurrent transac-

tion execution while still ensuring deterministic guarantees.

Partitioning. This simplest approach to supporting concurrent transaction execution despite the nondeterministic race conditions associated with multi-threaded execution on modern operating systems is to ensure that each thread is completely independent from each other. Such threads can be scheduled arbitrarily without affecting the state of the database system. This approach is taken by H-Store.[30]

H-Store partitions the database across the number of cores in a distributed set of servers. A single thread runs on each core, processing all transactions that access data stored on that core's partition. As long as all transactions only access data on a single partition, then each thread is completely independent from every other thread. However, transactions that access data from multiple partitions require expensive coordination logic, which in some implementations results in global serialization of these transactions. Thus, H-Store supports high amounts of concurrency for partitionable workloads, but limited concurrency for workloads with large numbers of multi-partition transactions.

Ordered locking. If the database system is not partitioned, different threads are capable of accessing the same data. Therefore, race conditions that could affect database state exist, and arise from OS thread scheduling. However, even in the absence of needing to implement deterministic guarantees, database systems have always had to implement other guarantees that are endangered by race conditions arising from OS thread scheduling. Therefore, database systems have long implemented defensive mechanisms that prevent these race conditions, the most prevalent of which involve locking data.

Locking can also be used to implement deterministic guarantees, but more restrictive locking algorithms are necessary relative to nondeterministic locking. Several different deterministic locking algorithms have been proposed,[25,27,31] but the high-level idea is that locks must be requested in the order that transactions appear in the input log. If transaction X appears before transaction Y in the input log, then all of transaction X's locks must be requested before the first lock from transaction Y is requested. Furthermore, locks must be granted in the order that they are requested.

Requesting (and granting) locks in the order they appear in the input log is clearly deadlock-free since it will be impossible for two (or more) different transactions to hold locks that the other one needs. Furthermore, it is clear the final database state after concurrently executing transactions in the input log will be equivalent to what the state would have been if it had serially executed the transactions in the input log—one of the fundamental principles of determinism we discussed.

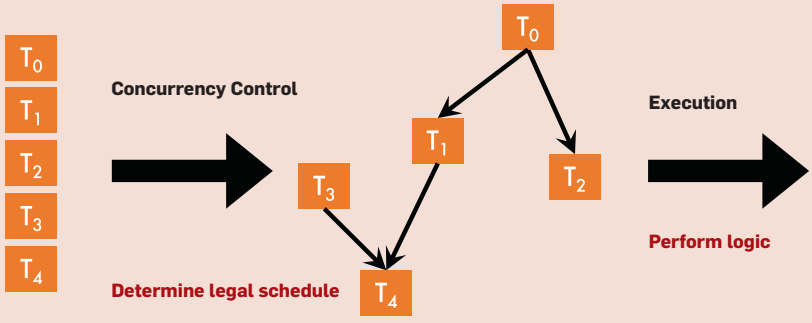
Unfortunately, there are situations where this approach also can lead to limited concurrency. In particular, if it is unknown at the beginning of a transaction which data it will need to access (and therefore need to lock), then any subsequent transaction cannot begin until the access-set of the previous transaction is determined, because it cannot request any locks until the previous transaction is finished with all of its requests. Thus, if all transactions do not know their access-sets in advance, they must run approximately serially with no concurrency whatsoever.

In practice, deterministic database systems that use ordered locking do not wait until runtime for transactions to determine their access-sets. Instead, they use a technique called OLLP[33] where if a transaction does not know its access-sets in advance, it is not inserted into the input log. Instead, it is run in a trial mode that does not write to the database state, but determines what it would have read or written to if it was actually being processed. It is then annotated with the access-sets determined during the trial run, and submitted to the input log for actual processing. In the actual run, every replica processes the transaction deterministically, acquiring locks for the transaction based on the estimate from the trial run. In some cases, database state may have changed in a way that the access sets estimates are now incorrect. Since a transaction cannot read or write data for which it does not have a lock, it must abort as soon as it realizes that it acquired the wrong set of locks. But since the transaction is being processed deterministically at this point, every replica will independently come to the same conclusion that the wrong set of locks were acquired, and will all independently decide to abort the transaction. The transaction then gets resubmitted to the input log with the new access-set estimates annotated.

Dependency graphs. Recent deterministic system implementations neither use partitioning nor locking. Instead, a dependency graph is generated from the transactional input log. Each node in the graph is a transaction, and edges in the graph correspond to conflicts between transactions.[8–10] The direction of each edge

Figure 1. Dependency graph scheduling.

Transactions are totally ordered by a preprocessor. This total order is then relaxed into a partial order based on the conflicts between transactions. Transactions are executed in an order consistent with the dependency graph.



is determined by the order that the two transactions being connected by the edge appear in the input log (see Figure 1). Once the graph is generated, it is used to manage execution of transactions. In particular, transactions that are not connected to each other in the dependency graph can be processed by independent execution threads without concern for race conditions between them.

Dependency graphs avoid the need for any centralized processing or data-structures during both graph construction and execution.[8] For graph construction, the set of database keys can be partitioned across a set of graph construction threads. Execution threads can independently crawl the graph to find independent transactions to process.

As we will describe, multi-versioning can be used to take this idea a step further and enable concurrent execution of transactions even if they are connected in the dependency graph.[8] Each write creates a new version of a data item, and reads are directed to the correct version based on where the transaction that is doing the read appears in the dependency graph. Therefore, two transactions that write the same data item can be run concurrently, and a transaction that reads a data item can be run concurrently with another transaction that writes the same data item if the read-transaction appears earlier in the input log.

In order to create the dependency graph, the set of data accessed by a transaction must be known prior to processing it. The OLLP techniques discussed earlier are thus also applicable for dependency-graph based systems.

Advantages of Determinism

The most straightforward and well-understood advantage of deterministic database systems is the benefit to database replication—as long as all replicas receive the same input, they are guaranteed not to diverge. Indeed, replication was the primary motivator behind the early deterministic database systems.[14,15,28,31] However, recent work has shown many other advantages to the deterministic architecture, from scalability, to modularity, to concurrency.

The only coordination that needs to happen in a deterministic database system is the communication required to agree on the input to the system.

Replication. All modern highly available OLTP database systems replicate database state in order to be robust to various failure scenarios. Furthermore, replication can improve the performance of read-only queries by serving them from the closest replica to the client (or from the least overloaded replica).

The consequence of using nondeterministic concurrency control protocols is that two servers running exactly the same database software with the same initial state and receiving identical sequences of transaction requests may nonetheless yield completely divergent final database states. This is because the strongest isolation guarantee available in traditional database systems is serializability that, as noted at the outset of this article, allows multiple transactions to execute in parallel, interleaving their database reads and writes, while guaranteeing equivalence between the final database state and the state that would have resulted had transactions been executed in some serial order. The key modifier here is "some." The agnosticism of serialization guarantees to which serial order is emulated generally means this order is never determined in advance; rather it is dependent on a vast array of factors entirely orthogonal to the order in which transactions may have entered the system, including thread and process scheduling, buffer and cache management, hardware failures, variable network latency, and deadlock resolution schemes.

Therefore, traditional replication schemes must take precautions to prevent or limit such divergence. Commonly used replication schemes generally fall into one of three families, each with its own subtleties, variations, and costs:

Post-write replication. Writes are performed by a single replica first, and the replication occurs after the write is completed. This category includes traditional master-slave replication, where all transactions are executed by a primary "master" system, whose write sets are then propagated to all other "slave" replica systems, which update data in the same order so as to guarantee convergence of their final states with that of the master.

This is typically implemented via log shipping[16,22]—the master sends out the transaction log to be replayed at each replica.

This category also includes schemes where different data items have different masters, and variations on this theme where different nodes can obtain "leases" to become the master for a particular data item. In these cases, transactions that touch data spanning more than one master require a network communication protocol such as two-phase commit to ensure consistency across replicas. Distributed deadlock must also be detected if locking-based concurrency control protocols are used.

For both the traditional master-slave, and variations with different data being mastered at different nodes, writes occur at the master node first, and data is replicated after the write has completed. In order to guarantee availability and durability, an acknowledgment from a replica must be received by the master before the transaction can commit. During this waiting period, no conflicting transaction can run, because until a transaction commits, it still has the possibility of aborting, and the isolation guarantee of database systems require that concurrent transactions do not see writes of aborted transactions. Thus, in addition to the fundamental latency cost of replication, post-write replication also incurs a concurrency/throughput cost.

Active replication with synchronized locking. A quorum of replicas have to agree on write locks granted to data items.[3] Since writes can only proceed with an agreed upon exclusive lock, all replicas will perform updates in a manner equivalent to the same serial order, guaranteeing consistency. The disadvantage of this scheme is the additional latency due to the network communication for the lock synchronization. For this reason, it is used much less frequently in practice than post-write replication schemes.

Replication with lazy synchronization. Multiple active replicas execute transactions independently—possibly diverging temporarily—and reconcile their states at a later time.[5,11,21] Lazy synchronization schemes enjoy good performance and CAP-level availability (availability of minority partitions during a network partition) at the cost of consistency.

Deterministic database systems are able to achieve the consistency and availability of post-write replication without paying the concurrency and throughput costs. As long as all replicas agree on the input to the database system (for example, via the preprocessing layer), each replica independently reaches a final state consistent with that of every other replica while incurring no further agreement or synchronization overhead.[a] Thus, the only coordination that needs to happen in a deterministic database system is the communication required to agree on the input to the system. This coordination happens entirely prior to transaction execution, and thus does not increase the window for which conflicting transactions cannot run.

Scalability. It is well known that single-server database systems will always have limited scalability. Highly scalable database systems must "scale-out"—partitioning the data across a distributed set of servers, and coordinating transaction processing among them. However, distributed servers may fail independently from each other, which risks "atomicity" properties of transactions (where either the entire transaction is processed or none of it is, but nothing in between). Therefore, traditional distributed database systems typically run distributed commit protocols such as "two-phase commit" that guarantee atomicity by ensuring all nodes involved in processing a transaction have not failed and are prepared to commit, and guarantee durability by ensuring the results of a transaction have reached stable storage and that a failure of a node during the protocol will not prevent its ability to commit the transaction upon recovery.

Due to the differences in the way failures are handled in deterministic systems, much of the effort of traditional commit protocols is unnecessary. As noted, while traditional systems abort all in-process transactions on a failed node, deterministic systems simply delay the completion of in-process transactions until the failed node recovers.[b]

Nondeterministic failure (no matter the reason for the failure, for example, a failed node, corrupt memory, or out-of-memory/disk) will not result in a transaction being aborted, since the database can always recover its state at the time of the crash by loading a check-pointed snapshot of database state, and replaying the input transaction log deterministically from that point.[19,23,31,33] Since the failure was nondeterministic, the transaction will eventually succeed.[c] Therefore, a distributed commit protocol does not need to worry about ensuring that no node fails during the commit protocol, and it does not need to collect votes from nodes involved in the transaction if the only reason why they would vote against a transaction committing is due to node (or any other type of nondeterministic) failure. Put a different way: the only thing a commit protocol must check is whether there was any node that executed code that could deterministically cause an abort (for example, an integrity constraint being violated).

For transactions that do not contain code that could cause a transaction to deterministically abort, no commit protocol whatsoever is required in deterministic database systems. For transactions that do contain code that could result in a deterministic abort, nodes involved in those transactions can vote 'yes' as soon as they can be sure they will not deterministically

a One downside of this approach is it requires full processing of every transaction on every replica, which can be more compute intensive than just replaying a log. Therefore, some deterministic database systems lazily process transactions,[10] opening up the possibility of copying the values of writes from a replica instead of calculating them locally.

b Another benefit of determinism is helpful here: since replicas are (in parallel) progressing through the same database states in the same order, then if replicas of this failed node remain active, then the rest of the database nodes do not need to wait for the failed node to recover. They can proceed with transaction processing and if they need data stored on the failed node as part of a distributed transaction, they can reroute that request to live replicas of the failed node.

c In the case of out-of-memory/disk, it may need to replay this log on a new/larger database server node.

abort the transaction. Therefore, transactions do not need to wait until the end of processing before initiating the commit protocol.

Deterministic database systems thus dramatically reduce the latency of the commit protocol. Instead of taking two or three rounds of communication in traditional nondeterministic systems, they take at most one round of communication in deterministic systems, and sometimes no communication is required at all. Furthermore, they enable the overlap of the commit protocol with transactional processing, thereby further reducing the latency of the protocol.

This advantage of shortening the commit protocol is far more significant than the obvious latency advantage discussed thus far. In general, if a database system is processing transaction *X*, the system prevents all transactions that conflict with *X* (transactions that write and in some cases even read the same data as *X*) from making progress not only while *X* is being processed, but also during *X*'s commit protocol. In traditional nondeterministic database systems, the commit protocol can be a large percentage of overall transaction process time. Thus, by reducing the length of the protocol, deterministic systems reduce the time period for which conflicting transactions cannot run. This increases the concurrency of the system under high-conflict workloads, thereby improving both throughput and scalability.

Figure 2. Throughput of BOHM's deterministic MVCC protocol vs. state-of-the-art nondeterministic single-version (which uses pessimistic locking) and multi-version protocols.

Each transaction performs two updates and eight reads under high contention. Records accessed come from a set of 1,000,000 records, chosen according to a zipfian distribution with theta of 0.9.

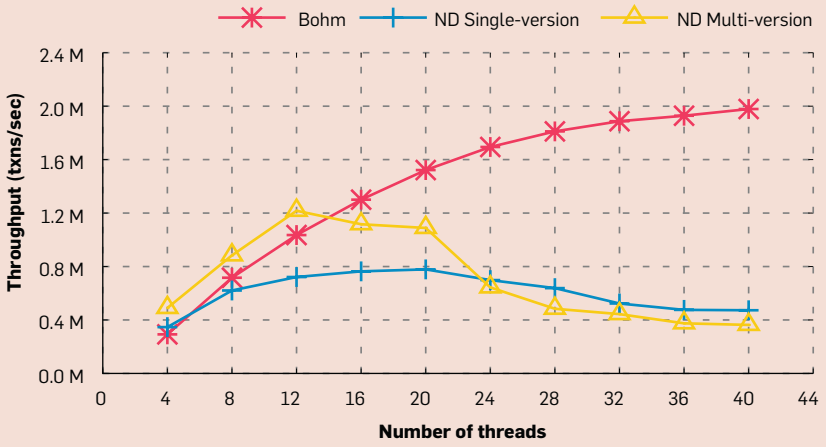


Figure 3. Transaction decomposition example.

(a) Coarse-grained transaction level scheduling prevents updates from being read until the end of a transaction's execution.
(b) Fine-grained piece level scheduling allows a piece's updates to be read even if one or more pieces remain to be executed.

```
Purchase(tx_id, item_id,
cust_id):
item = items_tbl[item_id]
if item.count == 0:
     Abort()

item.count -= 1

bills.insert(cust_id,
             item_id,
              item.price)

history.insert(tx_id,
               cust_id,
               item_id)
```

(a) Updates can only be read at the end of transaction execution

```
Purchase(tx_id, item_id,
cust_id):
item = items_tbl[item_id]
if item.count == 0:
     Abort()

item.count -= 1

bills.insert(cust_id,
             item_id,
              item.price)

history.insert(tx_id,
               cust_id,
               item_id)
```

(b) Each update can be read immediately

Concurrency. Deterministic execution requires that transactions are executed according to a predefined serial order. This requirement is stricter than that required for an execution to be serializable, which only requires that transactions execute according to some serial order. Surprisingly, this more restrictive requirement permits more concurrency among conflicting transactions at runtime.

Multi-version concurrency control. Modern database systems increasingly store data in a multi-versioned format. Each update to a record is associated with a unique version. An update creates a new version of the record and prior values of the record are preserved in old versions. Multi-versioning is attractive because, in principle, reads and writes to the same data item can be decoupled; reads can be satisfied by old versions while writes create new versions. Unfortunately, while this decoupling of reads and writes can be exploited by weaker consistency levels, such as snapshot isolation, it is insufficient to guarantee serializable execution. Serializable multi-version concurrency control (MVCC) protocols restrict concurrency between conflicting reads and writes, and are consequently unable to effectively exploit the presence of multiple record versions. Indeed, recently proposed serializable MVCC protocols[17] bear significant resemblance to single-version protocols.[35]

As mentioned previously, deterministic database systems create a

global log containing all transactions that have been input to the system. The system then guarantees concurrent execution in a fashion that is equivalent to processing all transactions serially in the order that they appear in this log. We described how a dependency graph can be generated from this log that explicitly tracks the read/write dependencies across transactions. This dependency graph can be used to precisely determine which versions must be read and written by transactions. Conflicting writes—and by extension the versions corresponding to those writes—are resolved according to the direction of the edges in this graph. The graph is also used to determine the correct version of a data record to read.

This technique allows for the following increases in concurrency in serializable MVCC implementations:

Reads never block writes. The version of each record that must be read by a transaction is determined by its position in the dependency graph, prior to transaction execution. Therefore, in order to satisfy a read, a transaction must simply wait until the version has been created by the corresponding writing transaction. As a consequence, a reading transaction does not need to block the execution of any writing transactions. Note, however, that reads may still have to block for the appropriate version to be produced by a corresponding write.

Writes do not conflict with each other. In several serializable and (non-serializable) snapshot isolation MVCC protocols, if two concurrent transactions attempt to perform conflicting writes to the same record, then one of the transactions is aborted.[2,17] These write-write conflicts are disallowed to prevent *lost updates*: a concurrency anomaly in which one transaction's writes are superseded by a later transaction, without the later transaction being aware of the former's write.[2] While aborting transactions on encountering write-write conflicts is sufficient to prevent lost updates, it is not necessary. For example, if the later transaction updates the record without reading it, then the later transaction's outcome is unaffected by the former write. In a deterministic database system implemented via dependency graphs, write-write conflicts are resolved according to the order in which they appear in the graph. Neither transaction is aborted, and lost updates are eliminated by the waiting necessitated by write-read conflicts.

In single-versioned concurrency protocols and recently proposed high performance (non-deterministic) serializable MVCC protocols, neither of these concurrency guarantees are possible.[8] Thus, this deterministic dependency graph approach yields a fundamental improvement in concurrency relative to these other approaches. The higher the overlap of the read and write sets across transactions, the higher the improvement in concurrency. For example, BOHM is an implementation of this approach.[8] Figure 2 shows an example of how this increase in concurrency leads to an increase in throughput relative to state-of-the-art single-version and multi-version concurrency control protocols for a high conflict workload. As the number of threads attempting to execute concurrently increases along the *x*-axis, the more clogged the system becomes with transactions unable to make progress due to conflicting concurrent transactions. However, BOHM becomes far less clogged due to its ability to decouple conflicting reads and writes. The experiment is described in more detail in the BOHM paper.[8]

Reducing the cost of strong isolation. Database systems execute transactions as indivisible units. As a result, a transaction prevents the processing of concurrent conflicting transactions until its logic has been executed in its entirety. This execution strategy is inherent to mechanisms such as strict two-phase locking and optimistic concurrency control, which are the basis of transaction processing mechanisms in most modern database systems. Under strict two-phase locking, transactions hold *long-duration* write locks on records; any locks acquired by a transaction are only released at the end of its execution. Under optimistic concurrency control, transactions perform writes in a local buffer, and only copy these writes to the active database after a validation step which determines that no conflicting transactions were running concurrently.

Executing a transaction's logic as a single unit fundamentally limits the performance of serializability as compared to weak isolation levels, such as read committed. Weak isolation levels allow applications to trade off consistency for performance by permitting more interleavings between conflicting transactions. As a rule of thumb, serializability requires that transactions generally read the most up-to-date value of each record at the point at which they are serialized. In contrast, read committed only requires that a transaction read a committed record value; record values can be arbitrarily stale. The combination of exposing a transaction's writes at the end of its execution (a consequence of executing its logic as a single unit) and serializability's requirement that transactions generally observe the latest value of a each record means that serializable implementations have far less room to interleave conflicting transactions. In order to circumvent this limitation, the system can decompose a transaction into sub-transactions or pieces, and then execute pieces as indivisible units. Instead of waiting for a transaction to finish executing in its entirety before exposing its writes, a piece's writes can be exposed as soon as the piece finishes executing, even if one or more pieces remain to be executed. Consider the example in Figure 3; conventional serializable protocols will only allow later transactions to observe the `item.count` update after the insertions into the `bills_tbl` and `history_tbl` have finished (3a), forcing later transactions that purchase the same item to wait. In contrast, transaction decomposition can allow the item's count update to be visible immediately (3b), which reduces waiting due to conflicts to the bare minimum.

While attractive in theory, transaction decomposition complicates the mechanisms that the system can use to guarantee serializability, atomicity, and recoverability, which every serializable protocol must provide:

▸ *Serializability.* Given that transactions can be decomposed into multiple pieces, how should pieces be scheduled such that transactions execute in a serializable order? A serializable ordering of pieces is insufficient because it does not ensure that transactions, each of which can be composed of several pieces, execute in a serializable order.

▸ *Atomicity.* Database systems employ well-established techniques to guarantee atomicity, the all-or-nothing processing of a transaction's updates, but it is unclear how to achieve atomicity when a transaction's updates are divided across several pieces. The fundamental issue is that a transaction can commit only if all of its pieces can commit, otherwise all of its pieces must abort.

▸ *Recoverability.* Database systems must ensure that committed transactions read committed data, a property known as recoverability.[4] Like atomicity, this guarantee is complicated by the fact that a transaction's writes may be spread across multiple pieces, and that an abort of even a single piece must cause all other pieces to abort as well.

While guaranteeing serializability is challenging because of the granularity of isolation (fine-grained piece level isolation versus coarse-grained transaction level isolation), guaranteeing atomicity and recoverability is complicated because non-deterministic database systems reserve the right to abort a piece at any point during its execution. If a subset of a transaction's pieces has finished executing and a later piece aborts, it may be unacceptable to commit the previously executed pieces (a potential atomicity violation). At the same time, it may also be unacceptable to abort the previously executed pieces' if their writes were observed by another transaction's pieces (a potential recoverability violation).

Transaction aborts can broadly be classified into state-based and system-induced aborts. State-based aborts arise from transaction/application logic choosing to abort a transaction based current database state. For example, a transaction may include an explicit abort statement that is conditionally triggered after reading a database record, or the transaction may be aborted if its updates cause a constraint violation. System-induced aborts are triggered by the database system, and are not strictly the result of database state. Examples of system-induced aborts include aborts due to deadlock handling logic, failures, and validation errors in optimistic protocols.

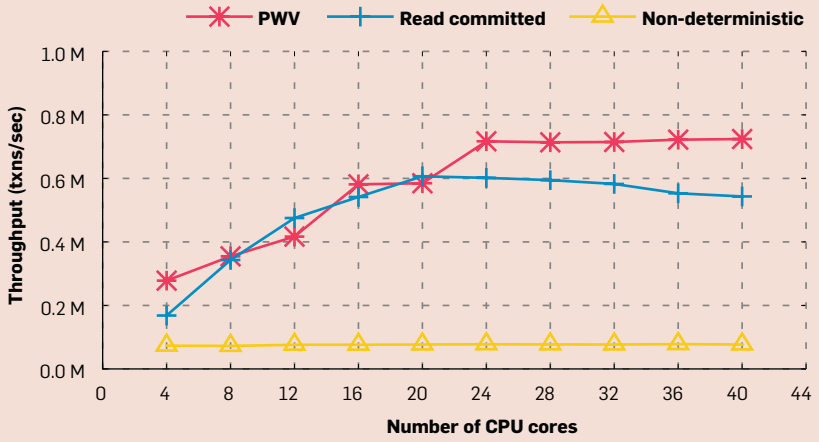
As described earlier, deterministic database systems eliminate any aborts that are not strictly determined by database state. Therefore, in a deterministic database system, only the subset of pieces that might experience state-based aborts are capable of causing a transaction to abort. A transaction is thus guaranteed to commit as soon as all such "abortable" pieces can commit, even if one or more "non-abortable" pieces remain to be executed. This guarantee yields a straightforward discipline to ensure atomicity and recoverability in deterministic database systems; a piece can commit and expose its writes after every abortable piece from the same transaction can also commit.

To guarantee serializability of decomposed transactions, deterministic database systems can extend the dependency graph technique to allow nodes in the graph to correspond to pieces instead of entire transactions. Edges are used to both represent conflicts between pieces and also commit dependencies amongst pieces within a transaction.[9]

Instead of using dependency graphs, nondeterministic transaction decomposition mechanisms must explicitly track a transaction's preceding conflicts as its pieces are executed, and enforce this order across all future pieces. Such a mechanism adds non-trivial runtime overhead and, in order to remain lightweight, requires approximations that reduce concurrency among pieces.

Figure 4 shows the performance of an implementation of a deterministic transaction decomposition protocol, piecewise visibility (or PWV).[9] PWV outperforms the non-deterministic transaction decomposition protocol by more than a factor of 5. Furthermore, it performs comparably to a weak isolation protocol (read-committed) despite guaranteeing full serializability.

Figure 4. Multi-core scalability of serializable transaction decomposition and read committed on the TPC-C benchmark configured with a single warehouse.



Logging overhead. In nondeterministic systems, the final state of the database is not known until after transaction processing. Therefore, they need to log all changes to database state as they happen and force all log records to stable storage prior to committing a transaction in order to ensure all state-changes made by committed transactions are durable under potential node failure. In addition to the additional latency incurred by the write to stable storage at the end of a transaction, past studies have indicated that generation of log records takes approximately 11% of all CPU cycles involved transaction processing.[12] In contrast, in deterministic systems, the final state is determined only by the input log. Therefore, no addition-

al logging is necessary in deterministic systems aside from this input log. Therefore, the log in deterministic systems is much smaller, much lighter-weight to generate, and is flushed at the beginning of the transaction instead of the end (and can be overlapped with transaction processing).

System modularity. Database management systems are notoriously monolithic pieces of software.[13] Many attempts have been made—with varying success—to build clean interfaces between various components, decoupling transaction coordination, buffer pool management, logging/recovery mechanisms, data storage structures, replication coordination, query optimization, and other processes from one another.[1,6,7,18,29]

One major fundamental difficulty in unbundling database components lies in the way concurrency control protocols are traditionally described.[32] Besides being highly nondeterministic, concurrency control algorithms are usually framed (and specified and implemented) in a very procedural way. This means that system components must often explicitly observe internal state of the concurrency control module to interact with it correctly. These internal dependencies (particularly for logging and recovery) become extremely apparent in modular systems that are otherwise successful at separating database system components.[18,29]

Deterministic systems create a log of all input to the system. Aside from the uses of this log described already, it also serves as a declarative specification of concurrency control behavior. Database system components that traditionally interact closely with the concurrency control manager can instead gain the same information simply by reading from the (immutable) transaction request log. This enables clean interfaces for normally entangled system components.

For example, the ordered locking mechanisms described earlier typically have a single concurrency control component that reads this input log and requests locks on behalf of transactions in the order that they appear in the log. Once a transaction has acquired all its needed locks, the transaction is handed over to execution threads for processing. These execution threads can process the transaction with no further communication with the concurrency control component since they already have acquired all of their locks before they begin. Similarly, the dependency graph mechanism we described creates a dependency graph based entirely on the information contained in the input log, and only hands over transactions to execution threads that are known to be safe to run without conflicting with concurrently running transactions. Once again, once an execution thread starts processing a transaction, no additional oversight from the concurrency control module is necessary. The execution module therefore does not need to have any knowledge of the concurrency control mechanism or implementation. Many other deterministic database systems also completely separate concurrency control from transaction execution.[8–10,24,30,33,36]

The recovery manager in a deterministic database system is entirely agnostic to implementation details of the log, scheduler, and storage backend—so long as they respect the determinism invariant.

The other major source of monolithicity in traditional nondeterministic systems is the logging and recovery manager that are notoriously cross-dependent with concurrency control managers and data storage backends. For example, recovery managers commonly rely on direct knowledge of record and page identifiers in the storage layer in order to generate log records, and may store their own data structures (for example, LSNs) inside the data pages themselves.

Deterministic database systems perform recovery by loading state from a recent checkpoint, and then deterministically replaying all transactions in the log after this point, which will bring the recovering machine to the same state as any non-crashed replica. Therefore, the recovery manager in a deterministic database system is entirely agnostic to implementation details of the log, scheduler, and storage backend—so long as they respect the determinism invariant.

Downsides to Determinism

Input preprocessing. At the outset of this article, we described the requirement of preprocessing transactions that modify database state in order to create a canonical log of input to the system. Scalable implementations of the preprocessing layer require

distributed coordination across multiple servers that necessarily increases latency of all transactions fed through this layer. Thus deterministic database systems may experience higher latency than nondeterministic systems. However, recall that deterministic systems shorten the commit protocol, and that they can commit trans actions after only partial execution. Thus, the latency disadvantage of preprocessing is often counterbalanced (and more) by these latency-saving techniques.

Information versus performance trade-off. The easiest way to avoid non-determinism arising from OS thread scheduling to is to disallow concurrency. This obviously would result in poor performance. Each of the deterministic database implementation techniques we described earlier in this article (for example, partitioning, ordered locking, and dependency graphs) improves performance by enabling concurrency at the cost of requiring information about transactions before they begin executing: either the partitions that they will access, or the actual records they will access. Although the OLLP technique can be used to eliminate the burden on the user to either provide this information directly or to submit transactions where it can be derived from inspection of the transaction, OLLP adds latency and increases the cost of processing the transaction.[d] Furthermore, the OLLP technique can only be used if the entire transaction is submitted to the system at once, so the "trial run" can complete. Therefore, OLLP cannot be used in conjunction with "interactive transactions," in which a client communicates with the system over multiple round-trips. Thus, for interactive transactions, there is an information vs. performance tradeoff: either the client must declare the access set of transactions (either in terms of partitions or records) when they are submitted to the system, or otherwise the system will default to (slow) serial execution.

d This cost increase is usually much less than doubling the cost of the transaction, since the trial mode can take several short-cuts not possible during runtime processing.[26]

Conclusion

Deterministic database systems have shown to be a promising direction to improving transactional database system scalability, modularity, throughput, and replication. However, all recent implementations have limited or no support for interactive transactions, thereby preventing their use in many existing deployments. If the advantages of deterministic database systems will be realized in the coming years, one of two things must occur: either database users must accept a stored procedure interface to the system, or additional research must be performed in order to enable improved support for interactive transactions.

Acknowledgments. This work was sponsored by the NSF under grants IIS-1763797 and IIS-1718581. We thank Alexander Thomson and Kun Ren for their contributions to the research described in this article.

References

1. Batoory, D., Barnett, J., Garza, J., Smith, K., Tsukuda, K., Twichell, B. and Wise, T. Genesis: An extensible database management system. *IEEE Trans. Software Engineering*, 1988.
2. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P. A critique of ANSI SQL isolation levels. In *Proc. of SIGMOD*, 1995, 1–10.
3. Bernstein, P.A. and Goodman, N. Concurrency control in distributed database systems. *ACM Comput. Surv. 13*, 3 (1981), 185–221.
4. Bernstein, P.A., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
5. Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S. and Silberschatz, A. Update propagation protocols for replicated databates. In *Proc. of SIGMOD*, 1999.
6. Carey, M.J., Dewitt, D.J., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J. and Vandenberg, S.L. The EXODUS extensible DBMS project: An overview. In *Readings in Object-Oriented Database Systems*, 1990.
7. Chaudhuri, S. and Weikum, G. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proc. of VLDB*, 2000.
8. Faleiro, J.M. and Abadi, D.J. Rethinking serializable multiversion concurrency control. *PVLDB 8*, 11 (2015).
9. Faleiro, J.M., Abadi, D.J., and Hellerstein, J.M. High performance transactions via early write visiblity. *PVLDB 10*, 5 (2017).
10. Faleiro, J.M., Thomson, A. and Abadi, D.J. Lazy evaluation of transactions in database systems. In *Proc. of SIGMOD*, 2014, 15–26.
11. Gray, J., Helland, P., O'Neil, P. and Shasha, D. The dangers of replication and a solution. In *Proc. of SIGMOD*, 1996.
12. Harizopoulos, S., Abadi, D.J., Madden, S.R. and Stonebraker, M. OLTP through the looking glass, and what we found there. In *Proc. of SIGMOD*, 2008.
13. Hellerstein, J.M., Stonebraker, M. and Hamilton, J. *Architecture of a Database System*, 2007.
14. Jimenez-Peris, R., Patino-Martinez, M. and Arevalo, S. Deterministic scheduling for transactional multithreaded replicas. In *Proc. of SRDS*, 2000.
15. Kemme, B. and Alonso, G. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proc. of VLDB*, 2000, 134–143.
16. King, R.P., Halim, N., Garcia-Molina, H. and Polyzois, C.A. Management of a remote backup copy for disaster recovery. *ACM Trans. Database Syst. 16*, 2 (1991), 338–368.
17. Larson, P.-A., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., and Zwilling, M. High-performance concurrency control mechanisms for main-memory databases. *PVLDB 5*, 4 (Dec. 2011), 298–309.
18. Lomet, D., Fekete, A., Weikum, G. and Zwilling, M. Unbundling transaction services in the cloud. In *CIDR*, 2009.
19. Malviya, N., Weisberg, A., Madden, S., and Stonebraker, M. Rethinking main memory OLTP recovery. In *Proc. of ICDE*, 2014, 604–615.
20. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.
21. Pacitti, E., Minet, P., and Simon, E. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. of VLDB*, 1999, 126–137.
22. Polyzois, C.A. and Garcia-Molina, H. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Trans. Database Syst. 19*, 3 (1994), 423–449.
23. Ren, K., Diamond, T., Abadi, D.J. and Thomson, A. Low-overhead asynchronous checkpointing in main-memory database systems. In *SIGMOD*, 2016, 1539–1551.
24. Ren, K., Faleiro, J. and Abadi, D.J. Design principles for scaling multi-core OLTP under high contention. In *Proc. of SIGMOD*, 2016.
25. Ren, K., Thomson, A. and Abadi, D.J. Lightweight locking for main memory database systems. *PVLDB 6*, 2 (2012), 145–156.
26. Ren, K., Thomson, A. and Abadi, D.J. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB 7*, 10 (2014), 821–832.
27. Ren, K., Thomson, A. and Abadi, D.J. Vll: A lock manager redesign for main memory database systems. *VLDB J. 24*, 5 (Oct. 2015), 681–705.
28. Schneider, F. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (1990).
29. Sears, R.C. *Stasis: Flexible Transactional Storage.* Ph.D. thesis, EECS Department, UC Berkeley, 2010.
30. Stonebraker, M., Madden, S., Abadi, D., Harizopoulos, S., Hachem, N. and Helland, P. The end of an architectural era (it's time for a complete rewrite). In *Proc. of VLDB*, 2007.
31. Thomson, A. and Abadi, D.J. The case for determinism in database systems. In *Proc. of VLDB*, 2010.
32. Thomson, A. and Abadi, D.J. Modularity and scalability in Calvin. *IEEE Data Engineering Bulletin 36*, 2 (2013), 48–55.
33. 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.
34. Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D.J. Fast distributed transactions and strongly consistent replication for OLTP database systems. *ACM Trans. Database Syst. 39*, 2 (May 2014), 11:1–11:39.
35. Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S. Speedy transactions in multicore in-memory databases. In *Proc. of SOSP*, 2013.
36. Wu, S.-H., Feng, T.-Y., Liao, M.-K., Pi, S.-K. and Lin, Y.-S. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In *Proc. of SIGMOD*, 2016.

Daniel J. Abadi (abadi@umd.edu) is the Darnell-Kanal Professor of Computer Science at the University of Maryland, College Park, MD, USA.

Jose M. Faleiro (jose.faleiro@yale.edu) is a Ph.D. student at Yale University, New Haven, CT, USA.