Cloud Actor-Oriented Database Transactions in Orleans

Tamer Eldeeb
Columbia University
tamer.eldeeb@columbia.edu

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Reuben Bond
Microsoft
reuben.bond@microsoft.com

Asaf Cidon
Columbia University
asaf.cidon@columbia.edu

Junfeng Yang
Columbia University
junfeng@cs.columbia.edu

Philip A. Bernstein
Microsoft Research
phil.bernstein@microsoft.com

ABSTRACT

Microsoft Orleans is a popular open source distributed programming framework and platform which invented the virtual actor model, and has since evolved into an actor-oriented database system with the addition of database abstractions such as ACID transactions. Properties of Orleans' virtual actor model imply that any ACID transaction mechanism for operations spanning multiple actors must support distributed transactions on top of pluggable cloud storage drivers. Unfortunately, distributed transactions usually perform poorly in this environment, partly because of the high performance and contention overhead of performing two-phase commit (2PC) on slow cloud storage systems.

In this paper we describe the design and implementation of ACID transactions in Orleans. The system uses two primary techniques to mask the high latency of cloud storage and enable high transaction throughput. First, Orleans pioneered the use of a distributed form of *early lock release* by releasing all of a transaction's locks during phase one of 2PC, and by tracking commit dependencies to implement cascading abort. This avoids blocking transactions while running 2PC and enables a distributed form of *group commit*. Second, Orleans leverages *reconnaissance queries* to prefetch the state of all actors involved in a transaction from cloud storage prior to running the transaction and acquiring any locks, thus ensuring no locks are held while blocking on high latency cloud storage in most cases.

PVLDB Reference Format:
Tamer Eldeeb, Sebastian Burckhardt, Reuben Bond, Asaf Cidon, Junfeng Yang, and Philip A. Bernstein. Cloud Actor-Oriented Database Transactions in Orleans. PVLDB, 17(12): 3720 - 3730, 2024.
doi:10.14778/3685800.3685801

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dotnet/orleans.

1 INTRODUCTION

Many modern cloud services have a 3-tier architecture with a stateless front-end, a stateful middle-tier that implements business logic, and a storage layer. The stateful middle-tier is needed due to heavy CPU and memory requirements to execute the business logic, which does much more than simply read or write the database. For example, apps often manage a lot of state in the middle-tier, such as a knowledge base or image cache. Some of this state needs to be read and written at high rates. These apps may also perform heavy computation, such as rendering images or computing over large graphs. Such requirements make it infeasible to embed the application logic as stored procedures in the storage layer [16]. The architecture also allows computation and storage to scale independently.

The actor model [6] has become a popular choice for building stateful middle-tier applications in the modern cloud, especially interactive ones such as games, social networks, Internet of Things, and telemetry [8, 13]. Actors are single-threaded objects that do not share memory and interact only via asynchronous message passing. The single-threaded nature of actors simplifies their implementation, and applications are typically made of many actors, which are natural units of scaling that are spread over many servers for scalability. Actors allow building a stateful middle tier with data locality and the semantic and consistency benefits of encapsulated entities via application-specific operations [13].

Orleans [5, 17] is an actor-based platform targeting stateful middle-tier applications with a primary focus on scalability and programmability. It simplifies the process of writing .NET scalable stateful middle-tier services, making it accessible to developers who are not necessarily distributed system experts. Orleans invented the abstraction of virtual actors [13], where actors are transparently loaded on demand, like pages in a virtual memory system. This solves a number of the complex distributed systems problems, such as reliability and distributed resource management, liberating the developers from dealing with those concerns. The Orleans runtime implements the virtual actor model, enabling applications to attain high performance, reliability and scalability. Over time, Orleans added support for automating the process of storing actor state durably to the programmer's choice of cloud storage systems.

By viewing an Orleans application as a collection of stateful actors, one can think of it as an actor-oriented database (AODB) [11, 14]. The distinguishing features of an AODB are that it scales out elastically to hundreds of servers, can use a variety of cloud storage services, and is compatible with the actor framework's programming model. Since application developers want to avoid being locked into a specific storage service, such a database must be able to use a wide variety of storage systems, such as page servers, BLOB servers, key-value stores, JSON stores, and SQL databases. An actor-oriented database needs to implement its own database abstractions, to compensate for the lack of such abstractions in the storage system and to ensure it integrates smoothly with the actor

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685801

programming model. This perspective has led to the evolution of the system to add support for database abstractions as first class concepts, such as geo-replication [12] and indexing [14], as depicted in Figure 1. Prior work also investigated using the actor model for query serving [30].

Applications often need to perform logically atomic operations that span multiple actors. Because actors do not share memory, adding general support for such multi-actor operations with isolation and fault tolerance guarantees requires an ACID transaction mechanism [15]. Since actors in Orleans are randomly distributed across many servers for scalability and availability, most transactions that access multiple actors will access multiple servers, and hence must be distributed. The well-known challenges of scaling distributed transactions [23, 29, 42, 45] have led many cloud systems to offer limited transaction support (e.g., within a shard [18, 36] or with weak semantics [9]) or no support at all [20]. On top of these standard challenges, additional difficulties arise from implications of the virtual actor model and requirements from Orleans users (which we describe in detail in §3). However, opting not to offer distributed transactions would put the burden on developers to use ad-hoc methods to obtain cross-actor consistency, which is hard to do well. ACID transactions are a key database abstraction, and supporting them as a first-class concept in Orleans has been a major step in its evolution into a fully-fledged actor-oriented database system.

In this paper we describe the design and implementation of ACID transactions in Orleans. We utilize a classic design that combines two-phase locking [24] (2PL) for serializable isolation, with two-phase commit [32] (2PC) for atomicity, to implement distributed transactions, but with novel and unique twists. Our design does not introduce any centralized components to Orleans such as a shared log or an independent transaction manager system. Instead, all durable transaction state is maintained in decentralized, per-actor cloud storage accessible only via transactional storage drivers, allowing developers to pick any cloud storage they prefer.

The high latency of cloud storage presents many performance challenges and would typically limit transaction throughput. We introduce two main techniques to mask the latency of cloud storage. First, we apply a distributed form of *early lock release* [25, 38] to 2PC by allowing a transaction to release locks at the start of phase-one of the 2PC protocol. After a transaction T finishes executing, it will not acquire more locks, so holding locks after this point has no value from a 2PL perspective. By releasing its locks at the start of 2PC, T avoids blocking other transactions that access T's writeset while T is executing its high-latency commit process. However, since T releases write locks before it commits, subsequent transactions can read "dirty" (i.e., uncommitted) data that will be invalid if T aborts. To avoid this inconsistency, a transaction keeps track of its dependencies on uncommitted transactions, and can only commit if and when all its dependencies commit.

Second, we utilize *reconnaissance queries* [42], which run transaction logic in a low isolation, dry run mode to prefetch all of the actor state in main memory prior to the actual transaction, so that transaction execution does not block waiting for slow reads from cloud storage while holding locks. Similar to prior work [23], the reconnaissance query also collects the identities of all the actors involved in a transaction so that lock acquisition requests can be ordered to avoid deadlocks.

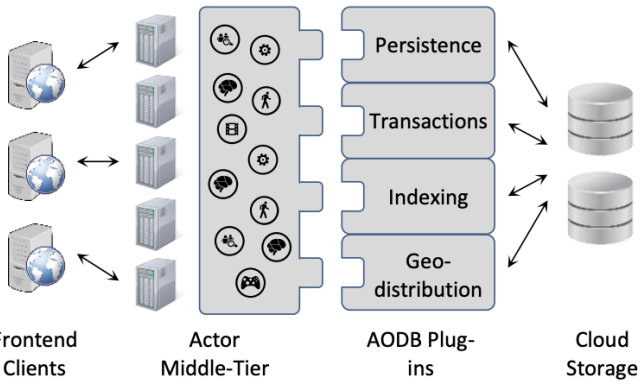


Figure 1: Actor-Oriented Database System [14].

While this paper focuses on transactions in an actor-oriented database, we think these techniques are generally applicable in any situation where 2PC commit is high. A summary of the techniques described in this paper:

(1) Transactional extensions to the Orleans programming model (§4).
(2) Transactional extensions to the Orleans runtime (§5).
(3) Pipelined commit protocol and distributed early lock release (§6).
(4) Reconnaissance queries for prefetching and deadlock avoidance (§7).

We also describe our experience and lessons learned from developing a prototype solution and taking it all the way to production in §3, including how feedback from users influenced the design and led to revisiting important aspects of the system. Additionally, as part of this paper, we are going to open source an Azure CosmosDB [3] implementation of the transactional storage interface described in §4.

2 BACKGROUND

2.1 Orleans

Orleans is a framework for writing actors, as well as a platform that provides a set of runtime services to execute that actor code. In this section we describe the parts of Orleans that are necessary to explain how we added ACID transactions to it. More details can be found in the Orleans documentation [5].

2.1.1 Runtime. An Orleans cluster is a set of servers running an identical software service called a *silo*. The Orleans Runtime is a set of subsystems that run on each silo of a cluster.

2.1.2 Grains. Actors in Orleans are known as *grains*. Each grain has a location-transparent identity, called its *key*, which is the only way to reference it. Grains cannot share state.

Like regular .NET types, grain types are defined using interfaces and classes. A grain's public interface can have only async methods, and grains only communicate via these asynchronous method calls; a grain can perform a system call to the Orleans runtime to obtain a

reference to another grain using the target grain's *key*, which is one of its member attributes. It then can use the reference to call any of the async methods on the target grain's interface. The reference returned by the runtime is in effect a *proxy* for the called grain, and allows the runtime to intercept all communication between grains in the system.

A method call immediately returns a promise, after which the caller can continue executing. It can also choose to wait for fulfillment of the promise (i.e., wait for the method call to finish executing and return) using the standard .NET *await* mechanism. Under the covers, this interaction is realized by messages in each direction.

Grains are single-threaded, and normally are non-reentrant. That is, a method call must execute to completion before the next call is processed. Optionally, a grain can be reentrant. In this case, the steps of method calls can be interleaved. However, even in this case, only one method call is allowed to be actively executing inside the grain at any given time.

2.1.3 Activations. Since grains cannot share state and can only be referenced via the location transparent key, the Orleans runtime is able to place any grain on any server in the cluster. Typically, it distributes grains randomly across servers to minimize the chance that any server is a bottleneck, though users can customize grain placement policies using plug-ins.

If a grain is not currently running when one of its methods is invoked, the Orleans runtime *activates* the grain, which involves choosing a server on which to execute the grain and executing the grain's constructor. It then performs the method call. It retains a reference to the grain in its distributed fault-tolerant grain directory so that future invocations can be directed to it. If a grain is idle for too long, the Orleans runtime *deactivates* it by calling the grain's destructor and releasing its resources. Since, this model of activate-on-demand is very similar to the demand-paging model of virtual memory, Orleans calls it the Virtual Actor Model [13].

Notice that there is no notion of *creating* a grain in the Orleans programming model. Grains are assumed to always exist, and are instantiated only when referenced.

The mapping of grains to servers is dynamic. Each time a grain is activated, it may (and often does) execute on a different server than its previous activation.

Grains are fault tolerant. If a server fails, Orleans detects the failure and updates its grain directory accordingly. The next invocation of a grain that died on the failed server causes that grain to be re-activated on another server, just like any invocation of an inactive grain.

The grain directory is implemented as a decentralized, distributed hash table where each server in the cluster is responsible for a portion of the directory. The system strives to ensure there is at most one active instance of a grain at any point in time. However, this can be violated during periods of server failures, cluster reconfiguration, or network partitions, which has implications for the correctness of our transaction implementation, which we discuss in §6.

2.1.4 Persistence. Orleans offers a simple declarative model of persistence, where a grain type identifies its persistent properties. Orleans maps those properties to persistent storage via a **storage provider** plug-in. The app specifies the storage provider (and hence the storage system) to use via a configuration attribute. Orleans uses the storage provider to populate a grain's state when the grain is activated. A grain can call WriteStateAsync to save its state at any time, e.g., just before returning from a method call that modifies its state or just before it is deactivated.

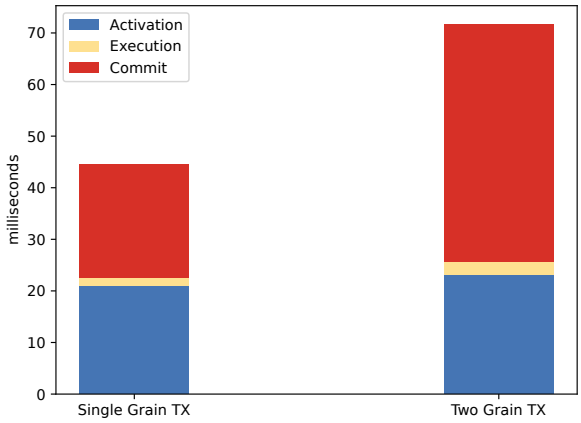


Figure 2: Breakdown of time spent in transactions

This approach to persistence decouples actor implementation from its storage. Developers can override this declarative persistence model with their own mechanism. For example, the developer can write custom code in the grain's constructor to initialize the grain state from any source, and can include code to save the grain's state in any method.

2.2 The Cost of Distributed Transactions

To understand the performance challenges of distributed transactions, consider the classic distributed transaction protocol: 2PL for isolation with 2PC for atomicity. To ensure isolation, a transaction holds locks until it finishes executing. Each object in its readset can release read-locks when it receives a prepare-request in phase-one of 2PC. However, each object in its writeset must hold its write locks until it receives a commit request in phase-two of 2PC.

2PC incurs significant performance overhead for two main reasons. First, it requires at least two network round trips and two synchronous log writes to persistent storage per transaction [28], which incurs network and storage I/O overhead, as well as CPU usage by the TCP/IP stack [45]. A typical cloud storage system has high latency due to networking, disk, and replication overhead. In our runs, a write to cloud storage takes on average around 20 milliseconds (ms) within a single region. Thus, a transaction running 2PC needs to hold locks for an additional ~40 ms. This limits throughput to 25 transactions/second (tps) on write-hot data. SSD-based cloud storage is faster [2, 4], but it still incurs double-digit millisecond 2PC latency, plus higher price.

Second, the coordination necessary to guarantee isolation can significantly decrease concurrency, which leads to performance degradation as well as high abort rates [9]. This increased contention due to 2PC is particularly harmful for short read-write transactions, because of the high latency of the commit protocol

relative to the time it takes to execute the transaction logic [42]. The impact of contention is evident in 2PL, but optimistic concurrency control (OCC) schemes are also not immune, and can in fact perform worse under high contention [28, 33, 46].

Figure 2 shows the breakdown of where the time is spent in the execution of a simple transaction that writes one or two randomly selected grains, whose storage is backed by Azure storage. As shown on the figure, the time to load the state and execute the commit protocol is much larger than the time taken to execute the transaction logic itself.

3 REQUIREMENTS

Here we discuss a set of requirements that we collected from Orleans users within and outside Microsoft.

First, to ensure the programming model for transactions is natural to Orleans users, transactions must be opt-in. They should affect only the programming model and performance of applications that use them.

Second, users want the ability to choose from a wide variety of cloud storage solutions. Hence all transaction storage must be external and pluggable.

Third, as reported in prior work [23, 42] and by our users, most workloads have low contention most of the time, but many workloads sometimes have a few very hot grains. Hence the transactions design needs to handle both high and low contention cases well.

We built an initial prototype satisfying these requirements [22]. However, when the Orleans team embarked on incorporating the prototype into their product, they identified additional requirements. First, they wanted an application opt into transaction functionality by using composition and dependency injection, rather than by extending a base class, as was required in the prototype. This allows better composability with other Orleans features that is difficult to achieve with an inheritance-based model.

Second, the prototype had a disadvantage where even single-grain transactions have to go through a two-phase commit process, and they were keen to eliminate this overhead both for performance reasons, and for integration with external one-phase systems that do not support 2PC.

Third, transaction aborts due to deadlock timeouts were identified as a major source of performance problems. The possibility of them occurring was often a surprise for users since locking is not explicit in the programming model. Users were keen for a way to avoid or reduce deadlocks.

Finally, they wanted there to be no additional components to deploy beyond that of the existing Orleans setup, which deploys one service (the Silo) per server. Users were quite unwilling to take on the many complications of the deployment, versioning and rollout story that would be necessary due to additional components. This rules out architectures that include a dedicated transaction manager service or a centralized sequencer [10, 23].

4 PROGRAMMING MODEL

In this section we describe the extensions we added to the Orleans programming model to support transactions. We include a short code sample in Appendix A to demonstrate how everything fits together.

```
public interface ITransactionalState<TState>
    where TState : class, new()
{
    Task<TResult> PerformRead<TResult>(
        Func<TState, TResult> readFunction);

    Task<TResult> PerformUpdate<TResult>(
        Func<TState, TResult> updateFunction);
}
```

Figure 3: Transactional Grain State Interface

4.1 Transactional Grain

A transactional grain is a stateful grain whose state is protected by ACID transactions. Any grain type can become transactional by declaring a field of type **ITransactionalState** in the class implementing the grain, which is a wrapper providing transactional read and write access to the grain state. The interface of ITransactionalState is shown in Figure 3. In Appendix A, the **AccountGrain** class is an example of a transactional grain.

4.2 Transactional Methods

Transactions in Orleans are bracketed by method tags, similar to the programming model of Java EE or .NET's COM+. A method on any grain interface can be declared as transactional by annotating it with the **Transaction** attribute and specifying a **TransactionOption** value indicating how this method behaves within a transaction. We list the most common TransactionOptions here. The Orleans documentation [5] contains a comprehensive list.

- **Create**. Every call to the method starts a new transaction, T, and completes T on exit.
- **Join**. The method can be called only within an already executing transaction.
- **CreateOrJoin**. If the method's caller is executing within a transaction, T, then it becomes part of T. If not, then the call starts a new transaction, T′, and completes T′ on exit.

Once a method that starts a transaction completes without throwing any exceptions, the Orleans runtime will attempt to commit the transaction. If it succeeds, the method returns normally. Otherwise, an exception will be thrown to the caller. Transactional methods should not have any side effects beyond changing the state of transactional grains, so that if they need to be aborted they can be rolled back cleanly. This is not enforced by Orleans, and left to programmer discipline.

All methods accessing transactional grain state must be transactional, but transactional methods can exist on non-transactional grains as well. In Appendix A, the **Transfer** method of the **ATMGrain** is an example of a transactional method on a non-transactional grain.

4.3 Transactional Storage

As we discussed in §3, Orleans users require the flexibility to use a cloud storage solution of their choice to store durable grain state,

```
public interface ITransactionalStateStorage<TState>
      where TState : class, new()
   {
      Task<TransactionalStorageLoadResponse<TState>>
           Load();

      Task<string> Store(
          string expectedETag,
          TransactionalStateMetaData metadata,
          List<PendingTransactionState<TState>>
               statesToPrepare,
          long? commitUpTo,
          long? abortAfter
      );
   }
```

Figure 4: Transactional State Storage Plugin Interface

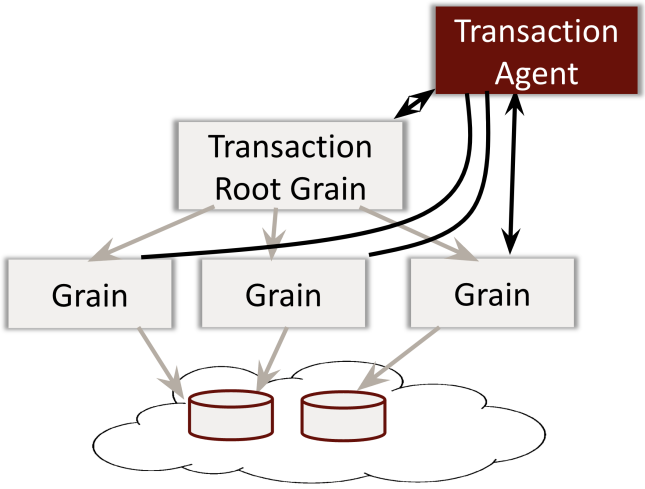


Figure 5: Transaction Execution

including transactional grains. To this end, we augment the Orleans framework with a pluggable transactional storage interface **ITransactionalStateStorage** that users can implement, see Figure 4. The interface requirements are quite minimal; it accommodates any highly available cloud storage solution that supports conditional writes based on an ETag check (which in practice means any cloud storage can be used).

5 TRANSACTION EXECUTION

Orleans uses dependency injection to populate the transactional state field of a transactional grain. In addition to the methods on ITransactionalState's public API to read and write the transactional state (Figure 3), the injected object also has methods to *prepare*, *commit* and *abort*, which are required for the 2PC protocol. In effect, a transactional grain acts as a mini-database. It is the unit of access, meaning that a transaction that reads or writes any part of the grain's state is considered to have read or written its entire state. This is to simplify the bookkeeping required during transaction execution; while it is conceptually possible that a transaction only needs to access a part of the grain's transactional state, grains are meant to be small and developers naturally divide large state across many grains. Orleans serialization features are used to generate a deep copy of the state of arbitrary type TState so that working copies can be created for transactions that can later be rolled back if the transaction aborts.

Each silo in the Orleans cluster has a component called the **Transaction Agent** (TA), which provides transaction functionality within the Orleans runtime. It has APIs to start, commit, and abort a transaction. The TA assigns a globally unique transaction ID to each transaction.

Recall from §2.1.2 that the Orleans runtime intercepts all grain method calls via grain reference objects. When a transactional method M is invoked, the runtime uses M's transaction tag to determine whether to start a transaction or propagate the caller's transaction to M. Figure 5 illustrates how a transaction starts and propagates. To start a transaction, the runtime calls the local TA on the server running M. Once M completes, the runtime calls its local TA to coordinate running the commit protocol to commit the transaction, which we describe in §6. Optionally, the runtime might also run M in reconnaissance mode prior to starting the transaction, as explained in §7.

5.1 Transaction Context

Each transactional method call carries a hidden parameter called the *transaction context* to record information about the transaction's execution. The transaction context includes the identity of the caller's transaction, T, and of the grains and versions accessed by T. A transaction context is created when the runtime starts a new transaction.

The transaction context is passed back and forth between the grain that started T and other grains T accesses (which all happen via grain async method calls). The transaction context supports a Union method, which accepts another transaction context with the same transaction id and unions its readset, writeset. After a method call is completed, the callee returns an updated transaction context which the caller unions with its own. If a method M1 running within a transaction T calls another method M2 within T, M1 must await M2's return so that it can union the updated context down M2's path. Otherwise, the call down M2 could have made some changes that are not recorded and updates by the transaction can be lost, breaking atomicity. If M1 fails to await M2, we call this an *orphaned call*. The system is able to detect such calls and abort the transaction if they occur.

5.2 Concurrency Control

We use 2PL with grain-granularity for concurrency control. When the runtime propagates a transaction, T, to a transactional grain, G, it locks G on behalf of T. While T holds the lock, only method calls that are part of T may execute. Other calls are queued. This is susceptible to deadlocks. Our reconnaissance queries (§7) allow us to avoid deadlocks in most cases. When this fails, we use timeouts as the fallback mechanism.

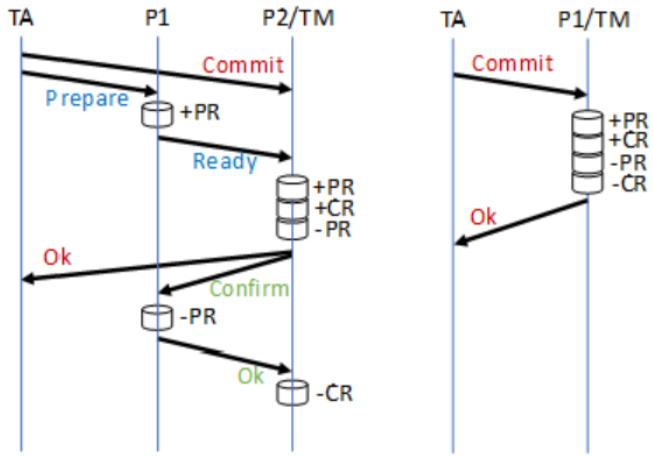


Figure 6: Pipelined 2PC with two participants (left) and one participant (right). Showing transfer of coordination from TA to TM. PR is PrepareRecord and CR is CommitRecord.

Since locks are not persisted, if a server hosting a transactional grain G fails, a transaction T holding the lock on G will lose the lock and T's in-flight updates of G. If G recovers before T starts 2PC, then T must avoid being fooled into committing and thereby breaking atomicity. As we explain in §6, the prepare-phase ensures this by checking whether T still holds its locks; if not, it aborts T. It is also possible for T to lose a lock due to a failure and then re-acquire the lock by accessing G again before it finishes, perhaps via a different callpath. In this case, T's context will refer to two versions of G. To handle this, if the union operation on T's transaction context has seen more than one version of any grain, it will abort T. This also handles the rare cases mentioned in §2.1.3, where there are multiple activations of G simultaneously.

6 COMMIT PROTOCOL

After a method that starts a transaction T completes without throwing any exceptions, the runtime initiates the commit process by invoking the local Transaction Agent (TA) and passing it the complete transaction context, which contains the full readset and writeset (i.e., participant transactional grains) of T. Figure 6 illustrates the full commit protocol.

Recall that the transactional grain state object has methods to implement 2PC. The TA designates one of the participant grains as the *transaction manager* (TM) of the transaction, which will be the authority on T's outcome. The TA then issues *prepare* RPCs to all participants, except the TM. The RPC includes the identity of the TM as one of its parameters. Each participant grain G that receives a prepare RPC first validates that T still holds its lock, and if it is part of the writeset, validates that it still has all the writes made by T. If this validation passes, G immediately releases the transaction's lock on it, which allows other transactions to access G and take a dependency on T (since T is not yet committed). T is now said to be *pending* at G. Asynchronously, G will submit a write to its transactional storage to record a *prepare record* for T, which persists T's writes as well as the identity of the TM to durable storage. If and when that write returns successfully, and G has committed all its pending transactions prior to T, G notifies the TM that it is done with its prepare phase. Alternatively, if the validation fails, G rolls back the writes of T and any subsequent pending transaction.

In parallel to the prepare RPCs, the TA also sends a *commit* RPC to the TM. The TM handles that call similarly to how other participants handle the prepare RPC, except it also has the additional responsibility of deciding T's outcome. The TM waits for all the other participants to successfully finish their prepare phase. After the TM receives notifications from all participants that they have successfully prepared, it asynchronously submits a write to its transactional storage to store the *commit record* for T. After that write is successful, the TM replies to the TA so it can notify the client that the transaction completed. Additionally, the TM notifies all the participants of T's outcome so they can clean up their logs and mark T as no longer pending. The persisted commit record includes the identity of all the transaction participants that need to be notified, so that the TM grain can perform this duty despite failures. Once all participants have been notified of the transaction commit, the TM can remove the commit record from its storage to reclaim space.

As illustrated in Figure 6, transactions that only touch a single grain only need one RPC to commit, instead of the usual two rounds of 2PC.

One subtle issue in Orleans' commit protocol is that even read-only participants have to go through the process of persisting a prepare record. Recall from §2.1.3 that multiple instances of the same grain might be active simultaneously in some failure cases. It is therefore necessary to do a write to transaction storage to perform an ETag check and ensure that the grain version read by the transaction is indeed the latest and not a stale version.

Apart from the above considerations that arise from the early release of locks, the recovery actions required when participant fails, a storage write fails, or a message is lost are the same as in other 2PC protocols. A detailed description appears in Chapter 7 of Bernstein et al. [15].

6.1 Discussion

Orleans' pipelined 2PC variant enables two major benefits. First, transaction locks need not be held during the high latency 2PC. Second, since persisting prepare and commit records to storage is asynchronous, writes can be efficiently batched, enabling a distributed form of the *group commit* optimization. Together, these reduce the contention of transactions dramatically and allow for much higher transaction throughput despite the high latency of 2PC over cloud storage compared to a standard 2PL/2PC implementation.

On the other hand, a potential drawback of this design is that it allows cascading aborts. However, since a transaction only releases its locks after it has already finished executing, transaction program failures and deadlocks are no longer possible. Hence, cascading aborts only happen only due to server failures, e.g., a hardware or operating system failure, which are relatively rare. Additionally,

when a server S fails, there is significant delay in cluster reconfiguration, since other servers need to wait long enough to be sure that S failed and is not simply slow. Then they must work around S's failure until S recovers. Thus, independent of the existence of transactions, application execution will be disrupted. Cascading aborts will add to the period of unavailability, but we argue that the effect is incremental, not a fundamentally new effect to be coped with.

7 RECONNAISSANCE QUERIES

A guiding principle in the design of Orleans transactions is to avoid holding locks while waiting on high latency cloud storage access. Orleans' pipelined 2PC removes the latency of the commit protocol from a transaction contention period. The other cause of cloud storage access is reading grain state from storage when activating a grain. Hot grains will typically have been activated by the system and hence have in-memory instances. However, this does not fully address the issue because a transaction might access hot grains along with other cold grains that have not been recently activated. There are several well-known techniques to work around this problem [15]. For example, the programmer could manually prefetch grains before executing the transaction. Another technique is to ensure hot grains are the last to be accessed. This is beneficial because it minimizes the execution time during which access to the hot grains causes a conflict. Unfortunately, these are are not always applicable, and they push a lot of complexity to programmers.

To deal with this problem in a general way, we added reconnaissance queries [23, 42], which is currently an experimental feature. When a grain method is supposed to start a new transaction T, the system can first start T in *reconnaissance mode*. In this mode, T executes in lock-free *repeatable read* isolation. Transactional grains, which are multi-versioned and keep track of stable, committed versions, return a known committed value to serve the read operations without waiting on any locks. Any writes made by T in reconnaissance mode are staged at the grain state, and discarded at the end of execution. After the reconnaissance phase, the system then starts T normally in lock-acquiring mode. Note that this does not require any changes to the application code, and is done completely transparently.

As shown in prior work [23, 42], it is rare for the readset of a transaction to change between the reconnaissance query and the actual transaction. As a result, running the reconnaissance query serves the important purpose of activating the grains in the transaction's readset which ensures their state is loaded from cloud storage prior to acquiring any locks. In the uncommon case where the readset does change, transactions will potentially have to hold locks while reading from slow cloud storage, but the system does not face any correctness problems.

Reconnaissance queries have two main disadvantages. First, they add to the query latency, although this additional latency does not lead to increased the contention. Second, they require executing the transaction logic twice before committing. While transactions tend to be short, this could still be wasteful if the transaction is compute-intensive, particularly in low contention cases. Hence, we allow users to opt-out of reconnaissance queries on a per transactional method basis.

Table 1: Single Silo Write Throughput.

	Single Grain	Two Grains
Writes (KTPS)	430	212
Persistent Writes (KTPS)	126	61
Transactions (KTPS)	46	11

7.1 Deadlock Avoidance

Since transactions in Orleans use locking for concurrency control, the system must deal with the potential for deadlocks. Deadlocks in Orleans transactions have been identified as a major source of performance issues by prior work [34], and by our users. Furthermore, that they can occur at all is often a surprise for Orleans users since locking is not explicit in the programming model.

The most general mechanism used in the system to handle deadlocks is transaction timeouts, since these are required to handle other possible failures. However, requiring transactions to wait for the entire timeout duration to resolve deadlocks can be too slow and in practice leads users to set their transaction timeouts to conservatively short values, resulting in many transaction aborts and restarts.

By making all transactions acquire their locks in the same order, we can prevent deadlocks. The system leverages the fact that the readsets and writesets of the transaction are (approximately) computed by the reconnaissance queries, prior to acquiring any locks. After the reconnaissance query, the runtime will issue RPCs to acquire the locks on the participant grains in a defined order before executing the transaction. A naive implementation of this idea would require adding $|\text{readset} \cup \text{writeset}|$ round-trips to time under locks, which increases contention. Instead, as in prior work [23], the Orleans runtime uses an RPC Chains [40] style approach, which cuts the round-trips required roughly in half compared to the naive approach. The way this works is that the lock acquisition RPC to a grain contains an ordered list of subsequent grains that need to be locked. When the silo locks a grain, it forwards the remainder of the list to the next grain and so on until the last grain is locked, which will then notify the first grain that all the locks have been acquired, at which point the transaction execution can start.

It is again possible that the transaction's readset or writeset changes between the reconnaissance query and actual transaction. We fall back to timeouts to resolve any potential deadlocks that might arise as a result.

This ordered lock acquisition scheme can actually increase a transaction's contention period, because lock acquisition has to be serialized. We allow the programmer to disable the scheme on a per transaction basis. Note that there is no overhead for the common case of transactions accessing a single grain.

8 EXPERIMENTS

In this section we study the overhead of transactions by comparing transactional operations to regular persistent non-transactional grains (§8.1) and the effectiveness of the pipelined commit protocol (§8.2) using micro-benchmarks. We also evaluate effectiveness of reconnaissance queries in Orleans using the Smallbank benchmark (§8.3).

All Silos and clients are running on Azure Standard D8as v5 VMs, which promise 8 cores and 32GB of RAM. We use Orleans 7. Unless otherwise stated, we use Azure Storage for grain persistent storage. Throughput numbers are computed by running the workload for 5 minutes and taking the average.

8.1 Transaction Overhead

Transactions incur overhead since transactional state has to create copies of itself to support rollbacks and multi-versioning, in addition to the RPCs and write needed for the commit protocol. To measure that overhead, we devise a simple micro-benchmark. Grains in this workload have a very simple 64-bit integer state. Each operation can either write the state of one or two grains, selected at random from a large universe of grains. In this experiment, storage for grain state is in-memory, since we want to measure the overhead of Orleans' transactions operations, not the cost of IO.

The results are shown in Table 1. The first row shows the total throughput for regular non-persistent grain operations, the second row shows the throughput for the same operations when performed with persistence, and the third row shows the throughput when performed within a transaction.Transactions have significant overhead compared to plain grain operations and even persistent grains. One notable thing about the results is that the throughput of single grain workload is significantly more than double the throughput of the two-grain transaction workload. This shows that single-grain transactions are significantly more efficient than multi-grain ones, which is due to the transfer of coordination from TA to TM in the single-grain case, as described in §6.

8.2 Single Grain Microbenchmarks

Hot data is a worst case for transaction performance and often arises in practice. Orleans' commit protocol is designed to support high throughput for hot data. In this experiment we evaluate its performance for a single hot grain and compare it to a standard 2PL/2PC baseline as well as non-transactional persistent grains. The workload involves writing a single grain which has 1KB state. We plot the results in Figure 7.

With its early lock release and pipelined commit, Orleans is able to sustain much higher throughput than a baseline 2PL/2PC implementation. Furthermore, it also greatly outperforms even the throughput of non-transactional grain writes, which still have to lock the grain for every individual write during the entire duration of performing the write to cloud storage.

8.3 Smallbank Multi-Transfer

Here we use the Smallbank [7] benchmark to evaluate the effectiveness of reconnaissance queries in addressing high contention and deadlock aborts, by comparing the performance when configured to use reconnaissance queries vs. a baseline where they are disabled. SmallBank is an OLTP benchmark simulating basic operations on bank accounts, and is a good fit for simulating actor workloads which are write-intensive and interactive [34]. We use a similar setup to prior work [34]; in particular, we also use a MultiTransfer transaction that withdraws money from one account and deposits money to multiple other accounts in parallel. In this workload, each account is modeled as a separate grain, and there

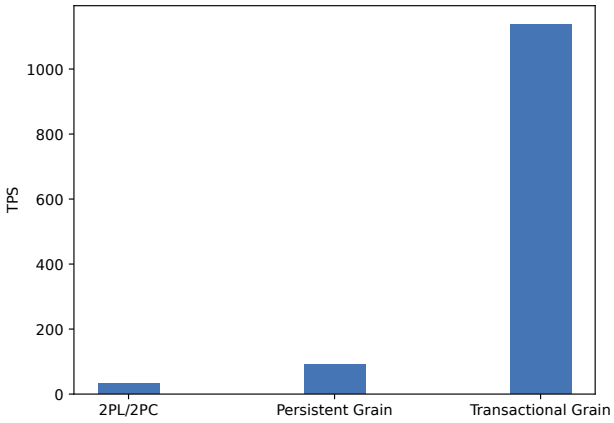


Figure 7: Single Grain Throughput

are 10k accounts in total. The grains accessed by each transaction are selected randomly using zipfian distribution. We vary the zipf parameter to generate different levels of *skewness* to measure the effect of contention. In a highly skewed workload, transactions access only a small set of grains, which causes them to be highly contended. We plot the throughput and abort rates under different workload skewness in Figure 8.

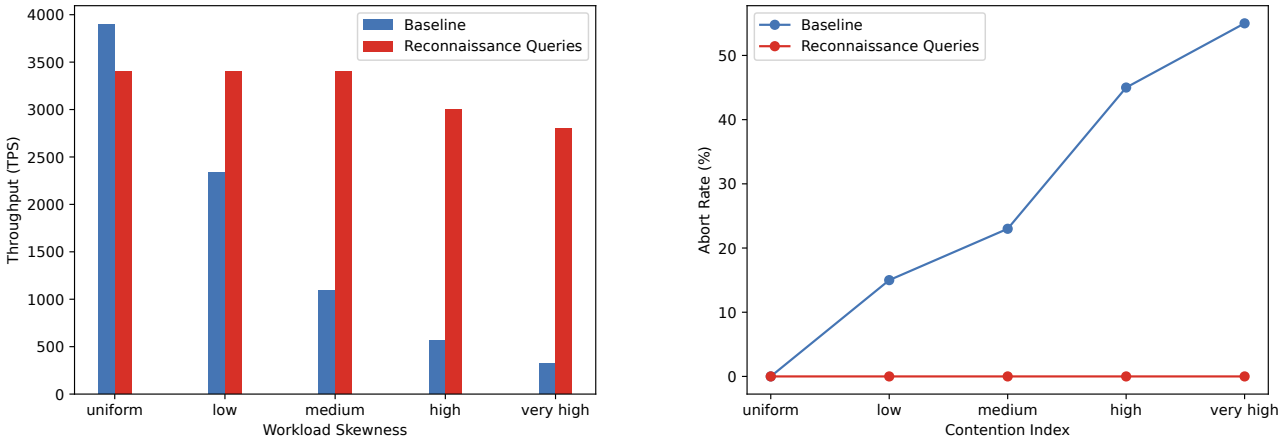
Analysis. As expected, under low contention, the reconnaissance queries in Orleans are mostly wasteful and consequently the baseline configuration has better throughput. As contention increases, however, they become crucial. Throughput of the baseline drops significantly and deadlock aborts become frequent, even with a modest increase in skewness, whereas the performance of the variant with reconnaissance queries holds steady and is able to avoid deadlock aborts completely.

9 RELATED WORK

Early Lock Release. Relaxing strict 2PL to increase concurrency by releasing locks at earlier stages [38, 39] is a technique that was applied to single-node databases with large buffer pools, since a transaction may run in less time than it takes to log the transaction's commit record on stable storage [25], and was later rigorously formalized and further developed in works like *controlled lock violation* [25] and Bamboo [27]. Distributed forms of this technique were recently proposed to optimize performance in distributed databases [26]. Orleans pioneered the use of a distributed form of early lock release by releasing all of a transaction's locks during phase one of 2PC, and by tracking commit dependencies to implement cascading abort. Conceptually this is also similar to MongoDB's speculative execution model [37].

Commit Dependencies. The notion of commit dependency was introduced in the ACTA framework [19]. We know of two prior works that use the concept.

In Speculative Locking (SL) [31], if a transaction T_1 updates x and a later transaction T_2 reads x, then T_2 speculates by having



(a) Throughput. **(b) Abort Rate.**

Figure 8: Smallbank Multi-Transfer Results

two incarnations, T_{21} that reads T_1's before-image of x and T_{22} that reads its after-image [42]. T_{21} and T_{22} both take a commit dependency on T_1. If T_1 commits, T_{22} is retained, else T_{21} is retained. The simulation study [31] of a distributed DBMS shows that SL gets better throughput than 2PL by overlapping speculative executions of T_2 with T_1, at the cost of more CPU load. By contrast, in Orleans, T_2 only takes a dependency on T_1 after T_1 terminates so it has no more CPU load.

Microsoft's Hekaton uses a more limited form of commit dependency [21]. It allows T_2 to take a commit dependency on T_1 if T_2 started after T_1 finished execution and entered the validation phase but has not yet committed. Thus, it benefits from overlapping T_2's execution with T_1's validation. Unlike Orleans, Hekaton is a DBMS, not a programming framework, and is not distributed.

Deterministic Execution. Deterministic execution has been explored as an alternative to distributed commit in systems such as Calvin [42] and Aria [35]. Systems based on deterministic execution typically have to restrict the programming model to one-shot stored procedures, and need to know the transaction's readset and writeset ahead of time. This makes it impractical to adopt in a stateful middle-tier system like Orleans.

Reconnaissance Queries. Reconnaissance queries were proposed in deterministic database systems [41, 42] as a mechanism to support *dependent queries*, which are queries whose read and write-sets cannot be determined before executing the query. The idea is to execute the query first in a low isolation mode to (approximately) collect the identities of records accessed. Chardonnay [23] uses a similar idea in order to prefetch the readset from disk into memory before executing the transaction, and to schedule lock acquisition to avoid deadlocks. We adopt this technique in Orleans.

Transactional Actor Frameworks. An earlier design of Orleans had a transaction mechanism based on multi-master replication [17]. It only provided snapshot isolation, not serializability. It was dropped before Orleans was released because it performed poorly and users found it too complex [13]. Akka is a Java-based actor framework that has transactions, but only on a single machine [1]. Orleans is compared to Akka in Bernstein et. al. [13]. Snapper [34] is a transaction library for single-node systems based on the Actor model, which enables deterministic execution for transactions that can be labeled with their readsets and writesets, while simultaneously supporting non-deterministic execution for transactions where this is not possible.

Co-designing commit and replication. A typical implementation of 2PC on top of a consensus-based replication protocol requires 4 round-trips to run the full 2PC protocol [47]. The additional RPCs come from having to go through a leader replica to perform replication. To eliminate these RPCs, TAPIR [47] introduces a leaderless, inconsistent replication protocol (IR) and a complex commit protocol that is tightly integrated with IR and uses Optimistic Concurrency Control. This work is complementary to ours, and in principle we could use TAPIR optimizations to reduce 2PC latency. Carousel [43] reduces latency by overlapping running portions of the consensus and commit protocols, but limits the programming model to only support two-round Fixed-set Interactive transactions. Natto [44] builds on Carousel by introducing prioritization and scheduling techniques to improve performance for high contention workloads.

10 CONCLUSIONS AND FUTURE WORK

We presented the design of ACID transactions mechanism in Orleans, an actor framework and platform that has evolved into an actor-oriented database system. Orleans transactions have to be distributed over external high-latency cloud storage, yet we have shown how to mask the high latency of cloud storage using early lock release, pipelined 2PC and reconnaissance queries to achieve high throughput and good performance. We shared many experiences from our journey to productionize transactions in Orleans,

including how much considerations like extensibility, ease of deployment and ease of integration with existing workflows often trump pure performance once the system achieves performance acceptable to customers.

There is much that can be done to extend this work. On the research side, one could experiment with variations of our technique to identify further optimizations. For example, one could try multi-version optimistic concurrency control, so transactions can read or overwrite data that was last written by a still active transaction. This should increase the maximum throughput when the transaction conflict rate is low. On the practical side, it would be beneficial to avoid deep copying the entire object state when a small update is made to a big structure, e.g., a dictionary. One way is to implement a custom transactional variation of the data structure that can log and undo incremental updates.

11 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their useful comments and suggestions. Orleans Transactions was a joint effort by Phil Bernstein, Reuben Bond, Jason Bragg, Sebastian Burckhardt, Sergey Bykov, Tamer Eldeeb, Christopher Meiklejohn, Alejandro Tomsic, and Xiao Zeng. We also benefited greatly from feedback from our users: Josh Collins, Maryam Mohammadzadeh, and Allen Xiao.

A ATM APP EXAMPLE

```
public interface IAccountGrain {
    [Transaction(TransactionOption.Join)]
    Task Increment(int amount);

    [Transaction(TransactionOption.CreateOrJoin)]
    Task<uint> GetBalance();
}
public class AccountGrain : Grain, IAccountGrain {
    private readonly ITransactionalState<Balance> _balance;

    Task Increment(int amount) {
        return _balance.PerformUpdate(x => x.Value +=
             amount);
    }
    Task<uint> GetBalance() {
        return _balance.PerformRead(x => x.Value);
    }
}
public interface IATMGrain {
    [Transaction(TransactionOption.Create)]
    Task Transfer(Guid from, Guid to, uint
         amountToTransfer);
}
public class ATMGrain : Grain, IATMGrain {
    async void Transfer(Guid from, Guid to, uint amount) {
        var sender = client.GetGrain<IAccountGrain>(from);
        var receiver = client.GetGrain<IAccountGrain>(to);
        await sender.Increment(-amount);
        await receiver.Increment(amount);
    }
}
```

REFERENCES

[1] 2023. Akka documentation. http://akka.io/docs.
[2] 2023. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.
[3] 2023. Azure CosmosDB. https://azure.microsoft.com/en-us/products/cosmos-db.
[4] 2023. Google Cloud BigTable. https://cloud.google.com/bigtable.
[5] 2023. Microsoft Orleans. docs.microsoft.com/dotnet/orleans.
[6] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[7] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering*. 576–585. https://doi.org/10.1109/ICDE.2008.4497466
[8] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (sep 2010), 68–75. https://doi.org/10.1145/1810891.1810910
[9] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. HAT, Not CAP: Towards Highly Available Transactions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX Association, Santa Ana Pueblo, NM. https://www.usenix.org/conference/hotos13/session/bailis
[10] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. https://doi.org/10.1145/2517349.2522732
[11] Philip A. Bernstein. 2018. Actor-Oriented Database Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 13–14. https://doi.org/10.1109/ICDE.2018.00010
[12] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. 2017. Geo-Distribution of Actor-Based Services. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 107 (oct 2017), 26 pages. https://doi.org/10.1145/3133931
[13] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/
[14] Philip A. Bernstein, Mohammad Dashti, Tim Kiefer, and David Maier. 2017. Indexing in an Actor-Oriented Database. In *CIDR*.
[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.
[16] Philip B Bernstein. 2019. Resurrecting Middle-Tier Distributed Transactions. *IEEE Data Eng. Bull.* 42 (2019), 3–6.
[17] Sergey Bykov, Alan Geller, Gabriel Kliot, Jim Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *ACM Symposium on Cloud Computing (SOCC 2011)* (acm symposium on cloud computing (socc 2011) ed.). ACM. https://www.microsoft.com/en-us/research/publication/orleans-cloud-computing-for-everyone/
[18] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 143–157. https://doi.org/10.1145/2043556.2043571
[19] Panayiotis K. Chrysanthis and Krithi Ramamritham. 1990. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. *SIGMOD Rec.* 19, 2 (may 1990), 194–203. https://doi.org/10.1145/93605.98729
[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281
[21] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710
[22] Tamer Eldeeb and Philip A. Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report MSR-TR-2016-1001. https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2/

[23] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi23/presentation/eldeeb

[24] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (nov 1976), 624–633. https://doi.org/10.1145/360363.360369

[25] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled Lock Violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2463676.2465325

[26] Hua Guo, Xuan Zhou, and Le Cai. 2021. Lock Violation for Fault-tolerant Distributed Database System. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1416–1427. https://doi.org/10.1109/ICDE51399.2021.00126

[27] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 658–670. https://doi.org/10.1145/3448016.3457294

[28] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (jan 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[29] Pat Helland. 2007. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*.

[30] Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. 2022. Data-Parallel Actors: A Programming Model for Scalable Query Serving Systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1059–1074. https://www.usenix.org/conference/nsdi22/presentation/kraft

[31] P. Krishna Reddy and M. Kitsuregawa. 2004. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering* 16, 2 (2004), 154–169. https://doi.org/10.1109/TKDE.2004.1269595

[32] Butler W. Lampson and David B. Lomet. 1993. A New Presumed Commit Optimization for Two Phase Commit. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB '93)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 630–640.

[33] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 21–35. https://doi.org/10.1145/3035918.3064015

[34] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2022. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3514221.3526172

[35] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[36] Jun Rao, Eugene J. Shekita, and Sandeep Tata. 2011. Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore. *Proc. VLDB Endow.* 4, 4 (jan 2011), 243–254. https://doi.org/10.14778/1938545.1938549

[37] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable consistency in MongoDB. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2071–2081. https://doi.org/10.14778/3352063.3352125

[38] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial Strictness in Two-Phase Locking. In *Proceedings of the 5th International Conference on Database Theory (ICDT '95)*. Springer-Verlag, Berlin, Heidelberg, 139–147.

[39] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial Strictness in Two-Phase Locking. In *Proceedings of the 5th International Conference on Database Theory (ICDT '95)*. Springer-Verlag, Berlin, Heidelberg, 139–147.

[40] Yee Jiun Song, Marcos K. Aguilera, Ramakrishna Kotla, and Dahlia Malkhi. 2009. RPC Chains: Efficient Client-Server Communication in Geodistributed Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (6th usenix symposium on networked systems design and implementation (nsdi '09) ed.). USENIX.

[41] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 70–80. https://doi.org/10.14778/1920841.1920855

[42] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[43] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 231–243. https://doi.org/10.1145/3183713.3196912

[44] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 715–729. https://doi.org/10.1145/3514221.3526161

[45] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[46] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-Centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 511–526. https://doi.org/10.1145/3318464.3389724

[47] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404