Indexing in an Actor-Oriented Database

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

Mohammad Dashti
EPFL
mohammad.dashti@epfl.ch

Tim Kiefer
TU Dresden
tim.kiefer@tu-dresden.de

David Maier
Portland State Univ.
maier@pdx.edu

ABSTRACT

Many of today's interactive server applications are implemented using actor-oriented programming frameworks. Such applications treat actors as a distributed in-memory object-oriented database. However, actor programming frameworks offer few if any database system features, leaving application developers to fend for themselves. It is challenging to add such features because the design space is different than traditional database systems. The system must be scalable to a large number of servers, it must work well with a variety of cloud storage services, and it must integrate smoothly with the actor programming model.

We present the vision of an actor-oriented database. We then describe one component of such a system, to support indexed actors, focusing especially on details of the fault tolerance design. We implemented the indexing component in the Orleans actor-oriented programming framework and present the result of initial performance measurements.

1. INTRODUCTION

1.1 Motivation

The classic architecture for on-line stateful services has three-tiers: a database tier accessed via queries and stored procedures; a middle tier that implements some application functions on a cache and passes others through to the database; and a stateless client tier that interacts with end-users. In this architecture, the database is the center of attention. Users submit requests that invoke functions that do some local processing and read and write the database. There is non-trivial logic in many of those functions. Often, the database is the primary bottleneck. Therefore, one purpose of the middle and client tiers is to offload the database, so the system can scale elastically by adding or removing inexpensive servers that run the middle-tier and client-tier.

Many interactive applications developed today do much more than simply read and write the database. For example, they 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 applications perform heavy computation, such as rendering images or computing over large graphs. They monitor streams in near real-time to detect intrusion attempts or outlier measurements. To handle this memory and processor load, they need a large number of middle-tier servers.

To support this interactive middle-tier functionality, applications manage what amounts to a distributed, in-memory object-oriented database. An application is distributed for scalability and geo-distributed for low-latency access by users world-wide. It stores most data in-memory for fast response time. It encapsulates the data as objects, to share common functionality and ensure data integrity. Modeling the data as objects is quite natural, because the data often represents physical real-world objects. The objects comprise a database because many objects are **standing objects**, that is, they need to live beyond the lifetimes of the procedures or processes that created them. Some of them need to be **persistent**, that is, resilient in the face of failures. These applications often support a large community of users with intensive interaction and computation, hence use a lot of processor and memory resources. Thus, a deployment needs to scale out to a large number of compute servers independent of storage servers.

For example, a large-scale multi-player game represents players as objects. Players interact with each other, and with abstract objects such as games, grid positions, lobbies, player profiles, leaderboards, in-game money, and weapon caches. In a real-time social app, people are objects that interact via chat rooms, messages, photos, and news items. An IoT application for buildings represents sensors as objects, which detect the state of a room: temperature, motion, light, moisture, and sound. More abstract objects infer when to start or stop the air conditioning, when to alert a security guard of a break-in, or when to shut off electricity and water due to a flood. Applications with similar characteristics can be found in the telemetry, mobile computing, and communications domains, among others. Together, these types of applications represent a large fraction of new application development.

All of the above-mentioned applications have several aspects in common. First, since the real-world objects are independent, the software objects that model them do not share state. To communicate, they exchange messages asynchronously. If they need common access to state, then that state is modeled as other objects that they reference. To simplify the programming model, such objects are often restricted to be single-threaded. Objects with these characteristics are called **actors** [8].

Second, many, if not most actors that are in main memory represent the latest state of the modeled entity, not a stale, cached version whose freshest state is in persistent storage. Therefore, the application runs most requests using these "active" actors, not by executing stored procedures on persistent storage.

Third, as a corollary to the previous point, these applications use storage more for persistence than querying. Hence, they typically use document stores, storing the state of an actor as a record, JSON document, or BLOB.

Fourth, many actors are not stored persistently. For example, the state of a lobby actor in a game is the set of users connected to it. After a lobby recovers from a crash, different users may connect to it, so it is pointless to recover its state from storage. Some actors have read-only state, e.g., a price list of weapons in a game.

Together, these four points suggest that an actor-oriented application treats its actors as a database (DB). However, developers of such applications do not make heavy use of a database management system (DBMS). Rather, they typically use an actor

 8th Biennial Conference on Innovative Data Systems Research (CIDR '17). January 8-11, 2017, Chaminade, California, USA.

programming framework, such as Erlang [3], Akka [1], or Orleans [4]. These systems provide functions to activate, invoke, store, and recover actors. They often include a plug-in architecture to map actor state to storage, and publish-subscribe plumbing to stream events from external sources to actors and between actors. They rarely include more advanced DB functionality, such as queries, authorization, constraints, stream processing, and transactions—and when they do, the functionality is often quite limited. We view this omission as an opportunity.

1.2 Actor-Oriented Databases

We propose enriching a distributed actor framework so it becomes a full-function actor-oriented database (AODB) as shown in Figure 1. This extension entails adding plug-ins for transactions, indexing, queries, views, triggers, geo-distribution, and replication. Transactions would enable a set of actor invocations to be wrapped in an ACID transaction. Indexing would enable retrieving the set of active or persisted actors based on secondary key values. As in object-oriented databases (OODBs), queries could be executed over actors that are grouped into a collection [6]. Queries could be reused as view definitions or materialized.

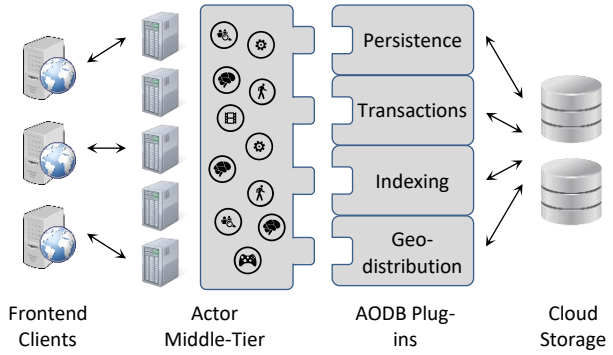

Figure 1. Actor-Oriented Database System

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*.

Scale-out is best satisfied by inexpensive servers that cloud vendors offer as virtual machines (VMs). An AODB cannot count on accessing a VM's storage devices after the VM recovers from a failure, so it is driven to using cloud storage.

Application developers want freedom of choice and to avoid being locked into a specific storage service. Therefore, an AODB's storage must be able to reside on a wide variety of storage systems, such as page servers, BLOB servers, document stores, and SQL databases. They may be storage services managed by the cloud provider or open-source versions of these components running on VMs managed by the application user.

An actor framework usually dictates the programming model for actor invocation, lifecycle, threading, communication, and exception handling. It may also control load balancing and caching. Developers want to work with one model, with no impedance mismatch to database functionality. Moreover, they value a framework with low cost of entry. Hence it must be possible to start developing applications in the framework without mastering all the database aspects first.

These AODB features affect the traditional design space of many database mechanisms. For example, you cannot assume that only objects that have been persisted need to be indexed. Nor can you assume the backing store supports indexing; most BLOB stores and some table stores do not. Thus, you cannot always rely on the indexing capabilities of the backing store. For transactions, there may not be a shared log, so updates to two actors on the same server may still require two-phase commit for atomicity. On the other hand, the actor environment may have features that help support DB features. For example, in an actor model where objects are not explicitly created and destroyed, referential integrity comes for free if based on object references.

1.3 Comparison to Existing Systems

AODBs are similar to OODBs in that programming language types and operations define the database interface, rather than embedding database types and operations into an independently defined language. However, an AODB differs from OODBs in its architecture and implementation. Most OODBs were targeted for design applications, with server-attached storage and function shipping to an object server (though a few did use data shipping).

Distributed object systems from the 1980's and 1990's are similar to AODBs in their ability to scale-out. However, hardly any focused on database applications. One exception is Thor [11][12], but it used a custom object-server with server-attached storage.

Some of today's DBMSs offer high-performance transactions over a main memory database, often with stream processing. This targets similar workloads as an AODB. Like an AODB, the data in main memory is the latest state, while the data in storage may be stale—a reverse-cache architecture. But unlike an AODB, the programming model is stored procedures and SQL dialects with a fixed DBMS. Also, most such systems can scale out only if the data is fully partitionable, which many actor applications are not.

AODBs are also similar to enterprise computing platforms, such as Java EE [9], in that they manage a middle-tier of application functionality that communicates with databases. However, they are not actor-oriented, in that their components are multithreaded and communicate via synchronous RPC. Applications sometimes use object-to-relational mappers, such as Hibernate or .NET Entity Framework, in which case the database functionality is in the database server, not the middle-tier.

To enable scalability, middle-tier applications often use cache managers, such as memcached and Redis. However, these systems cache records or structures, not objects, and hence do not support actor-like functionality.

An AODB is similar to graph databases that add query functionality to an object-oriented programming language, such as TinkerPop and Gremlin [14]. They could be integrated with an actor-oriented language, though we do not know of any that do. Often, they implement the graph with their own representation, rather than interpreting application objects and object-valued properties as a graph, which would be the natural approach for an AODB.

1.4 Contributions

We are participating in a project to develop an AODB added to Orleans. Orleans has a plug-in architecture for durable storage, which enables actors to be read and written to different storage systems. A recent project has added geo-distribution [5], and a project to add distributed transactions is nearing completion [7].

In this paper we describe our work on a third project, to enable actors to be indexed on secondary keys. Although indexing is well known database technology, we will see that the distinguishing

features of an AODB lead to different problems than those faced in building an indexing subsystem for a classical DBMS.

The main contributions of the paper are as follows:

- We introduce a new type of database system, AODB, to support scalable, fault-tolerant, distributed, actor-oriented applications.
- We define the challenges of indexing in an AODB and list design requirements for an exemplary AODB, Orleans.
- We describe a novel, extensible indexing architecture for AODBs that works with different types of storage systems.
- We present an algorithm for fault-tolerant, eventually-consistent and causally-consistent indexing in AODBs.
- We describe our implementation of an AODB indexing system and present the results of experiments that show the relative performance of different features of that system.

The rest of the paper focuses on adding indexing to an AODB. Section 2.1 gives background on Orleans, the actor framework in which we embed our solution. Section 2.2 discusses implications of Orleans features on the design of an AODB. Section 3 lays out requirements for an AODB indexing system. Section 4 describes the overall architecture of our indexing solution and the algorithm for maintaining reliable consistent indexes without the use of transactions. Additional implementation details are in Section 5. Section 6 gives some preliminary performance results. Related Work and the Conclusion are in Sections 7 and 8, respectively.

2. ORLEANS

The design space for an AODB component unavoidably depends on details of the actor framework in which it is embedded. Our indexing subsystem is embedded in Orleans, an open-source actor-oriented programming framework that extends the .NET Framework. Its main goal is to simplify the development of scalable, fault-tolerant, distributed applications. It is widely used by Microsoft (e.g., for the Xbox games Halo and Gears of War), is available as open source [15], and is used by many third parties. We give a brief introduction to Orleans, just enough to understand how we added indexing to it. A complete description is in [15].

2.1 Actors in Orleans

Actors in Orleans cannot share state. Each actor has a location-transparent identity, called its **key**, which is the only way to reference it. These two characteristics of actors enable the Orleans runtime to place each actor on any server. Typically, it distributes actors randomly across servers of a deployment, to minimize the chance that any server is a bottleneck; users can customize actor placement using plug-ins.

Actors communicate asynchronously only. A method call immediately returns a **promise**, after which the caller can continue executing. It can later synchronously **await** for fulfillment of the promise (i.e., wait for the method call to finish executing and return). Under the covers, this interaction is realized by messages in each direction.

If an actor is not currently running when one of its methods is invoked, the Orleans runtime chooses a server on which to activate the actor, executes the actor's constructor on that server, and then performs the method call. It retains a reference to the actor in its distributed fault-tolerant actor directory so that future invocations can be directed to it. If an actor is idle for too long, the Orleans runtime calls the actor's destructor and releases 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.

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

Actors are fault tolerant. If a server fails, Orleans detects the failure and updates its actor directory accordingly [4]. The next invocation of an actor that died on the failed server causes that actor to be re-activated on another server, just like any invocation of an inactive actor.

Actors are single-threaded, and normally are non-reentrant. That is, a method call must execute to completion before the next call is processed. Optionally, an actor 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 actor at any given time.

Orleans offers a simple declarative model of actor persistence, where an actor 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 an actor's state when the actor is activated. An actor 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. This approach to persistence decouples actor implementation from its storage. Developers often override this declarative persistence model with their own mechanism. For example, the developer can write custom code in the actor's constructor to initialize the actor state from any source, and can include code to save the actor's state in any method.

2.2 Programming and Design Implications

Orleans takes a middleware-centric approach rather than the database-centric one of standard 3-tier application architectures. We discuss here some of the implications of that difference for the programming model and design space for AODB features.

An actor type provides a naming scope. For a given key, a factory associated with the type always returns a handle to the same virtual actor. However, there is not an explicit type extent, that is, an enumeration of the values of the type that are currently being used. This is unlike a DBMS where each database table has an explicit extent, namely its rows, but it is similar to most programming languages. Thus, an indexing mechanism must not rely on an explicit type extent. While it is certainly possible for an application to maintain explicit collections of actors as a basis for indexing, it is important that an indexing scheme does not impose overhead on actors that do not participate in the collection, and hence are not indexed. Thus, it could be useful for an actor to know whether it participates in an index, and, if so, which one.

There are implicit extents in Orleans that could be targets for indexing, such as all currently running actors or all actors that have ever been initialized. While it might be possible to determine membership in such extents by other means, an index is an explicit representation of such extents, which can simplify applications and avoid excessive interaction with the underlying run time.

In an AODB, such as one based on Orleans, the unit of consistency is a single actor rather than the database as-a-whole. Since other actors can only discover state changes of a given actor through calls to that actor, an actor knows when any state changes

are visible to the "outside world". Moreover, an actor is the authority on its state. This is unlike an object in a 3-tier architecture, which might have an out-of-date cache of the authoritative state in a database. Thus, an actor can be relied upon to know an appropriate point at which to update an index in which it participates.

One consequence of actor-centric consistency is that an actor, absent a multi-actor transaction capability, is never certain about the state of another actor. The other actor's state could have changed since the last communication with it. Thus, the only guarantee is that any method result reflects some internally consistent state of the actor. Any indexing mechanism by itself can offer no better guarantee than that a returned object at some point satisfied the search key. Experience with Orleans shows that developers can write effective systems with these soft guarantees. Also, separately, we are working on an optional transaction mechanism to provide hard guarantees.

Fulfillment of promises is not guaranteed to be in the order of method calls in Orleans. If the sender needs ordering guarantees, it can hold off issuing a new request until it receives the result of the previous request. Thus, an indexing mechanism should not rely on implied order of method calls. On the other hand, it need not be concerned that an actor invoking it is blocked on its response, since the invocation is asynchronous.

With Orleans' virtual actor model, the activation and deactivation of actors is managed by the system rather than by applications. A method call on an actor will activate it if it is not already running. Thus, an indexing mechanism should avoid issuing additional method calls on the actors it indexes. For example, a hash index bucket might hold references to actors with different search keys. While it could consult each actor in the bucket to determine if it matches the current search key, that would activate inactive actors. It would be better to store the search-key value with each reference in the hash table.

3. THE INDEXING PROBLEM

3.1 Motivation

The state of an actor type is defined by member variables called **properties** (a.k.a. attributes in some data models). Actor instances can be gathered into explicit collections. If an application needs access to a particular collection of actors, such as those with the same value of a property, the app needs to create and maintain the collection, i.e., a secondary index. This approach duplicates functionality for each such scenario in the application and hence complicates the code. Moreover, it is difficult to work out all possible failure cases in a distributed system, so applications are likely to miss some cases and end-up with inconsistent indexes. For these reasons, it is beneficial to add generic functionality to the framework that an application can use to index any actor if needed.

In a conventional database system, an index serves only to accelerate existing functionality. In an actor app, by contrast, the index enables the functionality (e.g., find all players at a given location). In this sense, it is like a key-value store, where records can be accessed by a key only if an index was previously defined.

3.2 Requirements

Based on conversations with developers of actor applications, we identified the following requirements. The first two are familiar database features: access actors based on a property value or a range of property values; and optionally ensure uniqueness, i.e., ensure no two actors have the same value of the property.

The remaining requirements differ from those found in classical database systems. First, in an actor system, it is usually sufficient for each index to be causally consistent with respect to its base actor type. By this, we mean the index may be updated after the actor update commits, rather than bracketing both updates in a transaction. However, the index update cannot be postponed indefinitely. That is, each index must be eventually-consistent with respect to its base type. Thus, after an indexed property is updated, the corresponding index must eventually be updated—ideally, shortly after the actor update. This implies that the indexing solution needs to be fault tolerant to ensure that an index update is never lost.

Second, to make the indexing feature appealing to application developers, the API for accessing indexes should be tightly integrated into the actor programming language. For example, it should look very similar to the way actors are accessed based on their identity.

Third, it must be possible to have an index only for **active actors**, that is, that are currently running. Actors are added to such indexes when they are activated and removed later when they are deactivated. This feature is independent of whether the actor's state is saved in persistent storage. For example, a gaming application might want to access all active players at a given skill level, to offer them an *ad hoc* tournament. An actor's active-status approximates recent use. An application could explicitly track recently-used actors in a collection and then index such actors by skill level. Supporting indexes on active actors allows a convenient substitute for such explicit tracking, and hence was requested by users we consulted.

With the virtual actor abstraction, an actor's activation status is transparent to the application, in that all actors can be accessed, whether or not they are currently active. By indexing active actors, we expose actors' activation status and thus break the virtual actor abstraction. Therefore, the indexing system must be sensitive to activation status by not activating inactive actors. It is not just an efficiency issue, as described at the end of Section 2.2. It is a semantic issue in that it will increase the set of active actors due to system behavior, rather than application behavior.

Fourth, the index implementation should work with any persistent storage system, with only minor customization. In particular, it should work with storage that does not support indexing, such as BLOB stores or key-value stores. This entails explicitly storing and maintaining the consistency of the index as another storage object. We call this an **AODB-managed index**. On the other hand, the indexing system should use the indexing functionality of the storage system, if it exists. We call this a **storage-managed index**.

Indexing should be optional. It should impose no overhead to an application that does not use it. It should also be orthogonal to other features, meaning that it can be used with any combination of other AODB features.

4. INDEXING ARCHITECTURE

The high-level design of our indexing system is shown in Figure 2**.** The generic indexing functionality of an actor is encapsulated in an abstract actor type IndexableActor. An actor type *C* must inherit from `IndexableActor` to enable any of its properties to be indexed. For each property *p* of *C* that is indexed, the `IndexableActor` actor maintains two copies, a before-value and after-value, which are the values of *p* before and after its last update. These values are needed to do the corresponding update to *p*'s index.

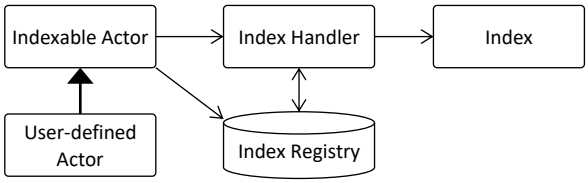

Figure 2. High-level index-system architecture

A method that updates *C*'s state *S* must also update indexes that are affected by updates to *S*. To do this, it calls the **index handler** associated with *C*, passing it the before-value and after-value of each indexed property. An empty before-value or after-value indicates an insert or delete, respectively. The index handler accesses the **index registry** to find information about the indexes defined on *C*. It includes the list of indexed properties of C and the type of index defined on each property, e.g., point vs. range, or centralized vs. distributed index. This information is gathered from the index annotations defined on C and enables the index handler to invoke the appropriate index actor for each of *C*'s indexes. There is a singleton instance of the index handler for *C* on each server, which is shared among all actors of type *C*. The actors of type *C* discover their corresponding index handler as a part of their initialization.

We now refine this high-level view and show how indexing requirements and aspects of the actor framework and cloud storage architecture influence its design.

4.1 Programming Interface

For many real-world user scenarios, indexes in an actor program not only optimize the program's performance, but also enable the core functionality of accessing a collection of actors. The explicit definition of indexes is part of the application program itself. Application programmers want the definition to be succinct, intuitive, and customizable, and to integrate smoothly with the programming language.

The generic actor type `IndexableActor` needs to know which properties are indexed and needs to be notified about each update to an indexed property. The syntax should allow for compile-time type checking, to ensure that the properties being indexed are indeed part of the actor's state. To do this, a certain amount of compiler magic and careful interface design is needed. To simplify the explanation and avoid boilerplate, we describe it in terms of class definitions. However, in fact, interface definitions are also involved. The interface of an actor is separate from the class that implements it and user programs only use the interface to interact with the actor.

The `IndexableActor` actor type is the super-type of all indexable actors and is defined to be generic, parameterized by an ordinary class that contains all the properties of the actor including the ones being indexed. For example, for the indexable actor type Player, an ordinary class `PlayerProperties` is defined. Each instance α of Player includes an instance of `PlayerProperties`, which contains all of α's properties. The indexed properties in `PlayerProperties` are annotated with an attribute "[`indexed`]", which the indexing system can find using reflection. To pick up indexing functionality, Player inherits from `IndexableActor<PlayerProperties>`.

Our use of inheritance to enable indexing functionality was a compromise between programmability and ease of implementation. The main disadvantage is that it is not orthogonal to other features, when used in a language that supports only single inheritance. For example, in Eldeeb and Bernstein [7], an actor enables transaction functionality by inheriting from `TransactionalActor`. For an implementation to offer both indexing and transactions to an application, it would also need to allow an actor to inherit from `IndexableTransactionalActor`. Clearly, the number of combinations of features grows geometrically with number of AODB features, and is hence undesirable from both a programmability and implementation perspective. Replacing inheritance with a more suitable mechanism is a high-priority item for future work.

Ordinarily, a client accesses an actor by feeding its key *k* to the actor type's factory, like this:

```
Player p = ActorFactory.GetActor<Player>(k)
```

Suppose `Location` is in `PlayerProperties` and has been tagged as [`indexed`]. To retrieve players based on an index, one can invoke a direct lookup method on the class, or use the Language-Integrated Query (LINQ) facility of .NET to write:

```
IQueryable<Player> result =
          from p in ActorFactory.GetActors
                      <Player, PlayerProperties>()
          where p.Location == "Redmond"
          select p;
```

Passing the `PlayerProperties` type to the `GetActors` method in the above LINQ query looks redundant, because `PlayerProperties` was passed as a type parameter to the `IndexableActor` type that Player indexes. However, this deep access to the generic type parameters is not supported in many languages, such as C# and Java.

The decision to have an index is made at class-definition time, not dynamically at any time via a CreateIndex operation. In an AODB, indexes are an integral part of the application, not just an optimization for query processing. If any change to the choice of indexes is required, application code will change. These modification will be applied the next time the application is deployed.

4.2 Capturing Updates to Indexed Properties

Any indexing mechanism on actors requires a way to capture changes in the target actors. `IndexableActor` plays the role of an interceptor that automatically captures the changes from actors. This interception happens by calling the `UpdateIndexes` method of the parent `IndexableActor` type. This method is also implicitly called whenever actor tries to persist its state. By default, before-values and after-values of the properties of the actor are used for denoting the modifications. However, this is not always the most efficient way to capture an update to an indexed property. For example, the property could be a compound structure that is indexed as a whole, but whose components are independently updated.

To give the app developer flexibility in representing updated properties, we define a class `MemberUpdate` that encapsulates an update to an indexed property and offers methods to get the property's before-value and after-value of the property. The instances of `MemberUpdate` are produced by an implementation of the `UpdateGenerator` interface that are employed in the `UpdateIndexes` method. An UpdateGenerator takes the current properties of the actor and a previously generated MemberUpdate if one is already stored inside the actor left over `from` a previous index update. Then, it produces a new instance of `MemberUpdate` that represent the change. This new instance of `MemberUpdate` is stored inside the actor to be used in the next index update.

The default implementation of `UpdateGenerator` captures both before and after values of an indexed property inside an instance

of `MemberUpdate`, but programmers can provide their own `UpdateGenerator` implementations. Using a user-defined `UpdateGenerator` in the example of a compound property above, the `MemberUpdate` might include the full before-value but the after-value of only components that changed, thus avoiding redundant storage of identical before and after portions. Programmers can determine the special `UpdateGenerator` for the specific indexed properties using an additional parameter to the [`indexed`] annotation.

Each call to `UpdateIndexes` of an actor creates a dictionary of property names to `MemberUpdates`. This dictionary is passed to the index handler for processing the updates with respect to the indexes defined on each property of the actor type.

4.3 Processing Updates to Indexed Properties

The simple design of Figure 2 ignores the possibility of failures during actor and index updates. This possibility is especially troublesome for AODB-managed indexes, where the indexing system has to write into both actor storage and index storage.

It is tempting to use transactions to simplify fault tolerance. This is problematic for two reasons. First, most actor systems do not support distributed transactions. Second, transactions add cost due to concurrency control and two-phase commit, which is annoying given that most users only require eventual and causal consistency. We therefore consider designs both with and without transactions.

If transactions are not used, then the process of updating an indexed actor must execute as a multi-step workflow. A correct implementation of actors and storage must satisfy the following invariants:

P1. (Consistency of actor and indexes) For each update request R to an indexed actor α, if R's update to α succeeds (that is, α's updated state can be read), then eventually all of α's indexes are updated to be consistent with α's updated state.

P2. (Causality) An index state is never ahead of actor states it refers to. That is, if an index for value v of property P refers to actor α, then α.P=v or α.P=v' where v' was written to α.P after v.

To design a workflow that satisfies these invariants, we require index updates and their inverses (i.e., undo operations) to be idempotent. Most index updates are naturally so. For example, to change the value of an actor α's indexed property P from u to v, we delete a reference to α from u's index entry and add it to v's index entry. When re-executing the update, if α is not found in u's index entry then no action is needed; if it is found in v's index entry then it need not be added again. In ambiguous cases, each update can be identified by a version number or similar state variable, which is written to the index along with the updated value or tombstone to ensure idempotence.

In the following description of the workflow, most writes to storage are lazy. These writes execute periodically but frequently, so that under high load they persist the result of many update operations to the relevant actor. Batching writes improves throughput under high load at the cost of some latency in replying to the relevant request.

Figure 3 shows a workflow that satisfies the above invariants. It starts with an update request R from some caller K to an indexed actor α. Downstream, for each actor, such as α, there is a Workflow Queue actor WQ that manages requests to process index updates on behalf of α. There is also an Index actor that processes lookups and writes to an index over α. We assume each actor is single-threaded and non-reentrant. The steps are as follows:

1. When α has finished processing R and before it writes α's updated state to storage, it creates a workflow record w_R and calls WQ to append w_R to the queue.

2. Next, WQ eagerly writes w_R to WQ's storage and sends an acknowledgement to α when it is done. If there is already an in-progress storage write to the queue, then WQ waits for the write to complete before issuing another, which includes other workflow records that have been added to the queue in the meanwhile. This batching improves throughput under high load.

If α times out waiting for a reply from WQ, then it assumes (possibly incorrectly) that w_R was not written to storage. So α undoes the update it performed for R and throws an exception to K.

3. This step applies if and only if α updated a property that has a unique index. In this case for each unique index that is affected by the update to α, α tries to insert the new value into the index. If it succeeds (i.e., there is no duplicate), then it marks the new index entry as *tentative*, and marks the index entry for α's previous value as *tentatively deleted*. After all tentative updates are written to storage, α continues with step 4. If any tentative insertion fails, then α undoes any previous tentative updates, undoes its update to α's state, and throws an exception to K.

Subsequent lookup operations should ignore entries flagged as tentative and should ignore a tentatively-deleted flag by treating the index entry as present, for two reasons. First, R might fail and the updates will be undone. And second, even if R succeeds, since α has not yet been written to storage, reading the corresponding index updates would violate causality.

4. Actor α adds the identity of the workflow record w_R to its state and writes its state to storage. After the storage update completes, it replies to K, thereby completing the call.

5. There is a dispatcher D_{WQ} associated with WQ that executes workflows in batches. In each batch, for a workflow record w_R on actor α, in step 5.1 D_{WQ} ensures that the update to α occurs before updates to any of α's indexes by first calling α to check that α still has a reference to w_R and wrote it to storage. It might not, because the update that generated w_R did not complete, or because w_R was processed by an earlier execution of D_{WQ}, but there was a failure before it deleted w_R from storage. If it does not hold, then D_{WQ} undoes any updates it did to unique indexes and writes them lazily to storage. Then it deletes w_R from the queue in storage.

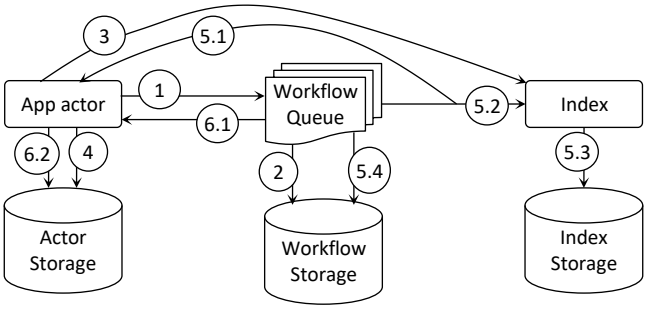

Figure 3. Workflow to propagate updates to an index

If α still has a reference to w_R, then in step 5.2 the dispatcher updates the relevant index entries. This includes changing tentative updates to unique indexes into permanent ones. It does these index updates in batches, per index. Next, if the index is

durable, in step 5.3 it lazily writes the index to storage. After the write completes, in step 5.4 it deletes w_R from the queue and lazily writes the updated queue to storage.

6. Finally, in step 6.1 the dispatcher calls α one last time to delete the pointer to w_R from α's state. Since the state of α has changed, in 6.2 α lazily writes its state to storage.

4.4 Correctness

We need to show that the protocol for actor updates is correct, despite any failures that might occur. In Orleans, there are two main categories of failures: server failure and message timeout. If a server fails, all actors on that server lose their memory state and disappear. A message timeout occurs when a method call does not return to the caller within its timeout period. An individual actor never fails. It can catch exceptions, but it cannot silently die.

To explain why the workflow in Section 4.3 is correct, we need to show that properties P1 (consistency of actor and indexes) and P2 (causality) always hold, despite the failure of any step in the workflow. We argue correctness by analyzing each workflow step, in turn.

Step 1: Actor α updates its memory state and sends w_R to *WQ*. Since α is non-reentrant, its state cannot be read until it returns to *K* in step 4. Therefore, P1 and P2 are trivially satisfied.

If α's server fails, then α's state is lost and its caller *K* will time out waiting for a reply to *R*. When α is activated again, it has no information about w_R, since w_R is not written to α's storage until step 4. Possibly, α appended w_R to *WQ* before α failed. We will show in step 5 that *WQ* will remove any downstream effects of w_R in this case. Hence, w_R has no effect, as required by P1 and P2.

Step 2: Actor α asks *WQ* to write workflow record w_R to storage. This has no effect on the state of α or its indexes, and hence has no effect on P1 and P2.

Step 3: If there are no uniqueness violations or failures while updating the unique indexes, then like step 1, step 3 finishes in a state where α's state cannot be read and tentative updates in unique indexes will be ignored. Hence, P1 and P2 are trivially satisfied.

If any of α's updates to unique indexes fails, it undoes any previous tentative updates to the indexes and to α's state. Thus, α and its indexes revert to their original states, so P1 and P2 are trivially satisfied. If there is a failure before this undo activity finishes, then it will be repeated when w_R is processed in step 5.

Step 4: After this step, α's state can be read. Thus, P1 requires that later steps ensure that indexes are eventually updated.

Step 5: This step updates α's indexes. Dispatcher D_{WQ} first checks that α has a reference to w_R. If so, then since α is non-reentrant, α must have completed step 4. Hence, updating the indexes will satisfy P2. If not, then D_{WQ} undoes tentative updates to unique indexes, thereby deleting all effects of *R*, and trivially satisfying P1.

If α did complete step 4, then D_{WQ} makes tentative updates permanent, and after they are in storage, it deletes w_R from *WQ*. If *WQ* fails before finishing this work, it is reactivated by a reliable reminder service [4]. On recovery, it will process w_R again, which is safe since index updates are idempotent. Thus, index updates are eventually processed, satisfying P1.

4.5 Transactions

Consider how we could use transactions to simplify recovery. One possibility is to group steps 1-4 into a transaction. This requires updating at least two actors (the queue and α) and possibly unique indexes. Since these actors are almost certainly not co-located in storage, this will require independent writes to storage and hence two-phase commit (2PC). They can do the writes in parallel, rather than sequentially, as in Section 4.3. However, there is an extra write in 2PC, which partly neutralizes this advantage.

Another possibility is to avoid the workflow entirely and do all the updates in a transaction. This too requires two rounds of writes for 2PC, so there is no improvement in latency. Depending on the concurrency control protocol used to update the indexes, it may entail more delay and/or aborts. It is not obvious to us which of the strategies will have the best throughput and latency. Exploring these alternatives is a topic for future work.

5. IMPLEMENTATION DETAILS

This section discusses more detailed aspects of the types of indexes we implemented.

5.1 Index Variants

There are three dimensions for the possible index types in an AODB: the type of query that can be directly answered using the index lookup, i.e., equality, range, or spatial query; the target collection of actors that is being indexed, i.e., only active actors or all persisted actors; and the way an index is distributed among multiple servers, i.e., logically partitioned based on key, physically partitioned based on actor location, or directly managed by the underlying storage. Each point in this three-dimensional space represents a specific index type with special consistency properties and fault-tolerance requirements.

In this paper, we focus on equality queries using hash indexes, over all initialized actors or over only active actors. An **initialized actor** is one that was activated at least once and whose state is in persistent storage. Once initialized, it remains initialized forever, even if it is subsequently deactivated. An active actor can be indexed whether or not it has persistent state. An index over initialized actors is called an **I-index**, and an index over active actors is called an **A-index**.

5.2 Distribution and Partitioning

Next, we discuss possible distribution strategies. The simplest form of index is a single actor that maintains the whole index. Even though it is a single point of contention, it is a practical option for small indexes with a low access rate. In fact, it can handle a moderately high update rate by batching updates, as we will see in Section 6.1.

In many cases, distributed indexes are preferable because they scale out. An index can be divided into buckets, where each bucket is a distinct actor. In theory, buckets can overlap, but to simplify index update and fault-tolerance processes, we ensure they are disjoint. There are two major ways that index entries can be partitioned into buckets: logically based on key or physically based on location.

In logical partitioning, each bucket is assigned a set of indexed value(s) it can contain. For example, when indexing the Location of Players, one way to assign buckets is by hashing the Location of each player. For a given Location, its hash value plus Location's index identifier uniquely identifies the actor maintaining that bucket of the index.

Ordinarily, actors with the same value of the indexed property are randomly distributed across servers. Therefore, a disadvantage of logical partitioning is that to process an update to an actor's indexed property, the actor's accesses to the index buckets are likely to be remote.

Another problem is providing users with consistent results. Often, an update to an actor's indexed property requires deleting the actor from one index bucket and inserting it into another one. Suppose the delete precedes the insert, and in between them, a caller issues two read requests, for all actors with the old value and for all with the new one. The reader fails to read the actor with the in-flight update, with no obvious fallback to find it. Instead, we should insert into the new bucket before deleting from the old one. This might result in seeing the actor in both buckets. But the reader can recover by checking the indexed property when accessing the actor. Unfortunately, this insert-before-delete strategy cannot be guaranteed when updates to index buckets are batched. For example, if one player moves from Location X to Location Y, while another player moves from Y to X, and if X and Y hash to different buckets, then the first bucket to be written to storage will cause one of the deletes to precede its corresponding insert.

In physical partitioning, an index has one bucket on each AODB server. The actors that are instantiated on a server send their index updates to the local index bucket on the same server. One benefit of this approach is that index updates are always handled locally. Another is that index updates are atomic, because moving an actor reference from the old inverted list to the new one can be done as one operation on the bucket. A third benefit is fault tolerance, in that a server failure causes actors and their indexes to fail together. This leaves the other servers in a consistent state. This property makes physically partitioned indexes a natural choice for indexing active actors (i.e., A-indexes), as it is fault-tolerant by design if the unit of failure is a server.

A downside of physical partitioning is that each index lookup requires a fan-out to all buckets on all servers. However, these lookups happen in parallel and the first result returned from one of the servers can be streamed back to the user.

Physical partitioning of I-indexes has a serious drawback. If an index is physically partitioned and if an indexed actor α located on server S previously existed on another server S', then α might still have an index entry on S' (something that cannot happen with an A-index). Therefore, an update to an I-indexed attribute of α requires a fanout lookup to partitions on all servers, followed by removing the previous index entry if one exists, and then updating the index bucket on S. This is much more expensive than updating a logically partitioned I-index, which can usually be done with a blind write. Overall, we believe that physically-partitioned I-indexes are not worth offering.

5.3 Handling Data Skew

The values of the indexed attributes are not always uniformly distributed among the index buckets. When dealing with a very large number of actors, special techniques are needed to handle this skew.

The index skew problem happens when an index bucket overflows. Skew can be either real or artificial. Artificial skew happens when the buckets are too coarse-grained. It can be mitigated by having finer-grained buckets, e.g., more buckets for a hash-index distributed across many servers. This requires a repartitioning of buckets, which can be minimized by any form of dynamic hashing, such as consistent hashing [10].

Real skew arises when a large number of actors have the same value of an indexed property. In this case, finer-grained buckets do not help. A solution is to define a logical super-bucket comprised of multiple physical buckets. When an ordinary bucket reaches its configuration-defined maximum size, it is converted to a super-bucket. The physical buckets of a super-bucket can be chained in a list with the super-bucket as header. Or the super-bucket can contain a list of pointers to the physical buckets. The latter is preferable, since it enables parallelizing delete operations, and fast access to the last bucket for insertions. Also, for I-indexes, it enables parallel retrieval of physical buckets from storage.

Another skew problem arises when the access rate on an index exceeds the ability of a single actor to serve it. Since reads usually predominate, this problem can be addressed by enabling an actor to allow multiple concurrent read threads and one write thread. To avoid interference between readers and the writer, the actor can use a multi-version representation of its state. This is a general-purpose capability that can apply to any type of actor, not just index buckets.

5.4 Implementation Status

So far, we have implemented the following features: I-indexes and A-indexes; AODB-managed and storage-managed indexes; the fault-tolerant, multi-step workflow for index update; single-bucket indexes; distributed indexes partitioned by key-value; physically-partitioned A-indexes; indexes that have very large buckets due to data skew; a programming interface that is integrated into the Orleans actor framework; and a modularized way of capturing updates from actors. Features that are high on our list are range indexes, and using transactions as appropriate for reducing I/O and simplifying fault tolerance.

The implementation is approximately 6K lines of C#. Nearly all of it is at application level, with just a few changes to the Orleans runtime library for performance-sensitive operations. We plan to release it soon as open source.

6. PERFORMANCE

We evaluate our indexing system in the exemplary three-tier setup shown in Figure 4. Multiple clients concurrently invoke methods on actors that are hosted by an actor middle tier. Both front ends and middle tier servers are virtual machines hosted in Microsoft Azure. Persisted actors, indexes, workflow records, and auxiliary data are stored in Azure Table, one of Microsoft's hosted key-value stores.

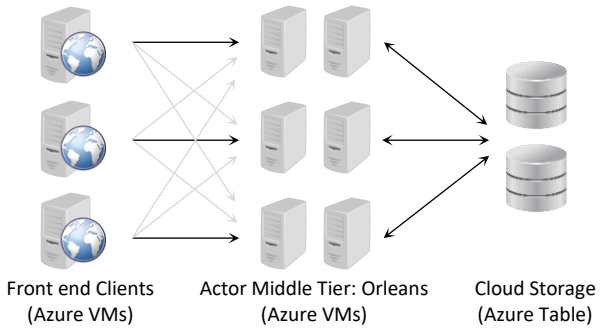

Figure 4. Performance evaluation configuration

We have experimented with moderately sized setups of up to 20 middle-tier servers. We overprovision the front ends to ensure they are not the bottleneck, typically with twice as many servers as the middle tier. Each client and server is an Azure "A4 worker

role," which is an 8-core 1.6GHz processor with 14GB of memory. The middle-tier servers run the latest Orleans release, version 1.3.1. The clients run the Orleans 1.3.1 client library, which enables calls to actors executing in the middle tier.

The performance numbers that we present here are preliminary and not comprehensive. There is a lot of room for optimizing the system before using it in production. These results are intended to give the reader intuition about the behavior of our indexing system, but not its absolute performance.

To emphasize the latter point: The latest release of Orleans runs at ~200K requests/second on their nightly load test on 25 servers, on a private cluster outside of Azure. In this load test, each client call has two hops, which implies 16K requests/second per server (i.e., 2*(200K/25)). Our baseline tests run at less than half that number. To have high confidence in our absolute numbers, we need to investigate the source of this discrepancy.

6.1 Scalability of A-Indexes

A major feature of Orleans is that it scales out to many servers. It is important that indexed actors scale out too. To test this, we ran a workload consisting entirely of updates to non-persistent actors. The middle-tier application consists of a single actor type with one indexed attribute. We compare three partitioning and distribution strategies for a hashed index:

- a single-bucket cluster-wide index
- an index that is partitioned logically based on key values, and
- an index that is partitioned physically, with one partition per server that indexes all actors on that server.

For a given number of servers, we did successive runs of two minutes, where each run offered more load (i.e., requests/second) than the previous run. To get stable numbers, we found it was important to gradually increase load in this way, rather than flooding the system with the maximum load it could handle and waiting for it to stabilize. For each run, we measured throughput, average latency, and 90th-percentile latency. Starting with a relatively low request rate as the offered load from the front ends, we gradually increased the request rate until either the average latency of completed requests reached one second or the 90th-percentile latency reached 3 seconds (whichever came first), and recorded that as the throughput for that number of servers. We did this for 5, 10, 15, and 20 servers, with a comparable number of servers doing the load generation. The variance across runs was typically 5-10% and never more than 15%. We omit error bars, since they would make the graph unreadable.

A quick look at the graph in Figure 5 shows that throughput grows as a function of the number of servers, validating the absence of any major bottlenecks. It also shows a similar difference between the throughput with no index and with each of the different index types. We found this to be surprising. Whereas an index update to a one-bucket or per-key index usually adds an RPC to the execution, an index update to a per-silo index is a local call, which should be much faster. After some investigation, we discovered that our technique for ensuring per-silo calls were local was effectively doing an RPC through the entire network stack and hence was much slower than it should be. Avoiding this overhead is a future-work item.

We define the scalability ratio as the fractional-increase in throughput divided by the fractional-increase in the number of servers. Ideally, we would like the ratio to be 1.0, but in fact it was significantly less. For example, with no index the throughput increases from 38K to 84K when going from 5 servers to 20 servers, which means the scalability ratio is 0.56, calculated as follows: ((84K/38K) / (20/5)) = .56. For the one-bucket, per-key, and per-silo cases, the scalability ratio is .61, .56, and .68, respectively.

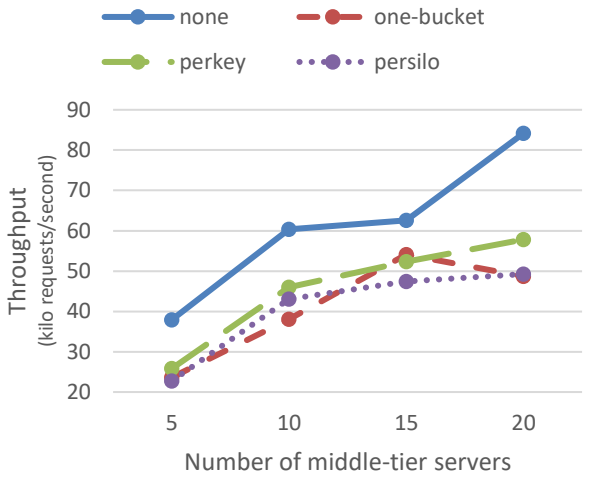

Figure 5. The performance scalability of different A-Indexes

Next, we study the effect of adding different numbers of A-indexes to a non-persistent actor type. Each index is over a string-valued property and is stored in one bucket. The configuration uses 5 servers. The results, in Figure 6**,** show that adding one index reduces peak throughput by 33%, due to the overhead of capturing updates and forwarding them to index buckets. The incremental cost of adding more indexes is lower, an additional 6%-13% per index.

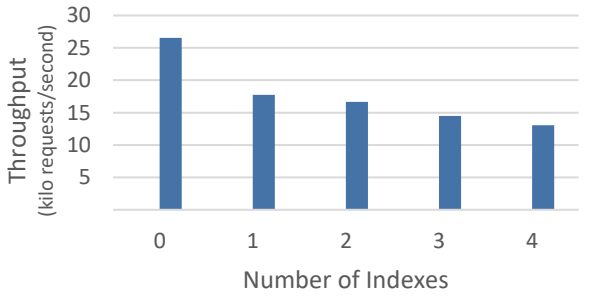

Figure 6. Effect of adding A-indexes for non-persistent actors

6.2 Effect of Actor Persistence on Indexing

Although some actors are only memory resident, many can also be persisted. We conducted an experiment to find out the effect of actor persistence on the performance of different index types.

In this experiment, we consider a single actor type *C* with a string property *p*. There are 10,000 instances of *C* initialized on 5 servers. The only operation done on the actors of *C* is *UpdateP*, which updates the value of *p* with a value selected randomly from 1000 predefined values and then persists the actor. We experimented with different index types: A-index, I-index and storage-managed index. The non-fault-tolerant and fault-tolerant variants of our AODB-managed I-index are represented as NFT I-index and FT I-index respectively, where the latter uses the workflow mechanism explained in Section 4.3. The storage-managed index (SM I-index) uses a key-value store as its

backend. The results are shown in Figure 7 and compared to a baseline that has no-index on *C*.

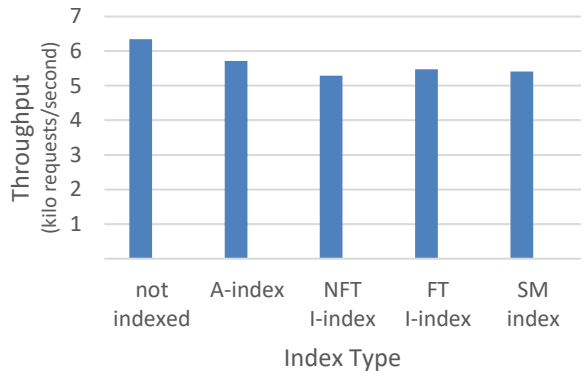

Figure 7 The effect of index type on update throughput

As you see in Figure 7, there is not a big difference between the performance of various indexing mechanisms if actors are persisted. The reason is that storage is network-attached and the indexing throughput is bounded by the throughput of network connections initiated from each server. In the default storage model supported by Orleans, each actor is persisted individually. The writes to storage are not batched, which puts an upper bound on the per-server throughput of operations on persistent actors. Extending the Orleans storage model to support batch writes is a subject for future work.

For persisted actors, some of the overhead related to indexing is hidden by the necessary storage access. Hence, in this scenario, the relative update throughput of A-Index is as high as 90% of the update throughput for actors without an index.

To quantify this effect in isolation, we ran a simple experiment to measure the combined effect on update throughput of persisted vs. non-persisted actors and of indexed vs. non-indexed actors using an A-index. In this experiment, updates to the index are propagated lazily. The results are summarized in Figure 8.

By persisting actors, throughput drops by 82%, from 35K to 6K. We ran 4K actors/server with 5 servers and overprovisioned storage bandwidth, to ensure that this performance drop is entirely due to processor overhead. By adding a non-persistent A-index to non-persistent actors, throughput drops by 34**%.** When combining both features, with an A-index over persistent actors, throughput drops by 85%, which is only slightly more than adding persistence alone. Since indexes are updated lazily, most of the overhead of updating the A-index overlaps with the update I/O.

	Persisted Actor	Non-persisted Actor
Indexed	5,400	23,600
Not Indexed	6,200	34,800

Figure 8. Comparing the effect of indexing and persistence on update throughput (expressed in updates/second)

7. RELATED WORK

We compared AODBs to existing systems in the introduction. Although we do not know of work on indexing that closely relates to this paper, our indexing work is somewhat similar to approaches to supporting a database cache in middle-tier servers. For instance, MTCache by Larson et al. [11] caches relations in the middle tier as materialized views. However, the cached data is read-only and updates are always performed on the base data. It uses SQL Server replication features to propagate changes in the base data one complete transaction at a time in commit order (i.e., the caches are transactionally consistent but may not reflect the latest state). A similar approach is infeasible in actor-oriented databases because it relies on a (not generally available) mechanism to trigger index updates from inside the storage system.

A lot of work has been done on maintaining materialized views. The special case of lazily maintaining views is related to our work as the same technique can be used to lazily update indexes. However, systems like the one proposed by Zhou et al. [18] rely on capabilities of the relational database system, namely a version store to access before images of rows and transactions to update the base table and create delta information atomically.

A recent work by Tang et al. [16] proposes a transparent indexing middleware component between applications and log-structured key-value stores, such as BigTable, HBase, or Cassandra. The approach is to store an inverted index as a table alongside the base data and to enrich the API with a GetValue method. Since this index middleware is oblivious to the programming framework, it can be used in an AODB as a storage-managed index.

Tai et al. describe an optimized replication strategy for indexing when many indexes are supported on the same table [16]. Using their optimized strategy, the number of replicas is less than twice the number of indexes. This strategy could be applied to the storage for our AODB indexes, thereby offering the benefit of multiple indexes along with the fault tolerance benefit of replication.

We note that ActorDB [1] also casts itself as a database inspired by the actor model. However, there are some significant differences between that system and our AODB approach. An actor in ActorDB is essentially a shard of a larger database, running as an independent DBMS instance. For example, in an ActorDB database that supports blog posts, each actor might contain tables for the posts and comments for a single user. As in AODBs, actors in this approach do not share state, but unlike AODBs, that state in ActorDB is always persistent. Further, there is no direct actor-to-actor messaging in ActorDB. Rather, applications interact with one or more actors using SQL inside transactions. Relative to Orleans, another difference is the actor life cycle: ActorDB actors are explicitly created and destroyed.

8. CONCLUSION

We presented our vision of an actor-oriented database system and the distinctive challenges in building one. We described an indexing component for an AODB, which exemplifies many of these challenges. We identified requirements for indexing actors in an AODB and a system design that satisfies them. In particular, we explained how to make indexes fault-tolerant and provided details of our implementation. Our preliminary measurements suggest that our implementation has good performance.

Achieving the vision of a fully functional AODB requires further research on the other database features that are missing from actor frameworks, including transactions, queries, views, stream processing, triggers, and replication. This calls for a tailored design of these components and their underlying algorithms to keep up with scalability, fault tolerance, and programmer friendliness of actor frameworks.

Acknowledgments: We are grateful for help with many aspects of this work from Sebastian Burckhardt, Sergey Bykov, Julian Dominguez, Tova Milo, and Jorgen Thelin. We also thank

engineers in Microsoft Studios and the Orleans community for suggesting and validating requirements for indexing actors.

9. REFERENCES

[1] ActorDB, actordb.com

[2] Akka documentation, http://akka.io/docs/

[3] Armstrong, J., "Erlang," *CACM* 53, 9 (2010), pp. 68–75.

[4] Bernstein, P.A. et al., *Orleans: Distributed Virtual Actors for Programmability and Scalability*, MSR-TR-2014-14, http://research.microsoft.com/apps/pubs?id=210931.

[5] Burckhardt, S. and P.A. Bernstein, "Geo-Distribution for Orleans," https://www.youtube.com/watch?v=fOl8ophHtug.

[6] Cattell, R. et al., *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, 2000.

[7] Eldeeb, T. and P.A. Bernstein, "Transactions for Distributed Actors," MSR Technical Report, https://www.microsoft.com/en-us/research/publication/transactions-distributed-actors-cloud-2, October 2016.

[8] Hewitt, C., P. Bishop, and R. Steiger, "A Universal Modular Actor Formalism for Artificial Intelligence," IJCAI, 1973.

[9] Java EE documentation, http://www.oracle.com/technetwork/java/javaee/documentation/index.html

[10] Karger D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. STOC '97.

[11] Larson P.-Å., J. Goldstein, J. Zhou, "MTCache: Transparent Mid-Tier Database Caching in SQL Server", ICDE, 2004.

[12] Liskov, B., M. Castro, L. Shrira, A. Adya: Providing Persistent Objects in Distributed Systems. ECOOP 1999: 230-257.

[13] Powers, D. M. W.: Applications and explanations of Zipf's law. NeMLaP3/CoNLL '98: 151-160.

[14] http://tinkerpop.apache.org/

[15] Orleans, http://dotnet.github.io/orleans

[16] Tai, A., M. Wei, M.J. Freedman, I. Abraham, D. Malkhi, "Replex: A Scalable, Highly-Available, Multi-Index Data Store," 2016 USENIX Annual Technical Conf., pp. 337-350.

[17] Tang, Y. et al., "Deferred lightweight indexing for log-structured key-value stores.", CCGrid 2015.

[18] Zhou, J., P.-Å. Larson, and H.G. Elmongui, "Lazy Maintenance of Materialized Views", VLDB, 2007