Indexing in Distributed Actor Systems

Philip Bernstein  
Microsoft

Mohammad Dashti  
EPFL

Tim Kiefer  
TU Dresden

David Maier  
Portland State Univ

8th CIDR  
January 9, 2017
Today’s interactive apps are built around a stateful, object-oriented middle tier

- Multi-player games, IoT, social networking, mobile, telemetry
- They comprise a large fraction of new app development
- Naturally object-oriented, modeling real-world objects

Examples of objects

- Gaming: players, games, grid positions, lobbies, player profiles, leaderboards, in-game money, and weapon caches
- Social: chat rooms, messages, photos, and news items
- IoT: sensors, virtual sensors (flood, break-in), buildings, vehicles, locations
Application Properties

Properties of these apps
- Objects are active for minutes to days, sometimes forever
- App manages a lot of state: millions of objects, knowledge graphs, images, videos
- App does heavy computation: complex actions, render images, compute over graphs, ...

Properties of the system
- Scale out to large number of servers
- Compute servers must scale out independently of storage servers
- Geo-distributed for worldwide low-latency access
Middle-tier Objects Comprise a Distributed DB

- Many objects outlive the processes that created them
- Many (but not all) objects are persistent
- Latest state is in main memory. Storage might be stale
- Active objects are in-memory for fast response
Actor Systems

- Many of these apps are implemented using **actor systems**
  - Simplifies distributed programming

- Actors are objects that ...

- Communicate only via asynchronous message-passing
  - Messages are queued in the recipient's mailbox
  - No shared-memory state between actors

- Process one message at a time
  - No multi-threaded execution inside an actor
Orleans is an open-source actor framework built on C#
- Ensures apps are fault tolerant and scalable
- https://dotnet.github.io/orleans/

Virtual actor model
- Each actor has a unique location-independent ID, always valid
- Actors are transparently activated on invocation
- On activation, actor invokes its constructor to initialize its state (e.g., read from storage)
- Actor can save state at any time (e.g., to storage)
- Runtime automates fault-tolerance, load balancing, actor lifecycle, ...
Actor-Oriented Database System (AODB)

- Current distributed actor systems lack DB functionality
  - But users frequently ask for it (and hack it)

- Vision: Actor-Oriented DB System
  - Indexes, queries, streams, transactions, replication, geo-distribution, views, triggers

- AODB’s main distinguishing features
  - Compatible with actor framework’s programming model (developer friendly)
  - In-memory and elastically scales out to hundreds of servers
  - Agnostic to the storage system, e.g., cloud storage services
Scalable and Storage-Agnostic

- Elastic scalability implies
  - Limited ability to co-locate functionality
  - Functionality must be parallelizable
  - Scale-out is more important than a fast path

- Storage agnostic implies each DB feature
  - Must work for persisted and non-persisted objects
  - Must not require the storage system to support it
  - Should benefit from a storage system that does support it
  - Must cope with storage latency of cloud storage
Requirements for AODB Indexes

- Statically choose indexed fields
- Optional uniqueness constraints (e.g., ensure Player.Email is unique)
- Index is eventually-consistent with actor and fault tolerant
- Can index active actors only (e.g., offer a tournament to certain on-line players)
- Can index persistent and non-persistent actors
- Leverage actor storage that supports indexing
- Support actor storage that does not support indexing
Challenges

- Lookup should avoid activating actors
- No type extents
- No multi-actor transactions
Fault Tolerance

- Index is comprised of actors, to gain benefits of Orleans
- Suppose we have an index on Player.Location

1. Update storage
2. Update index
3. Update index storage

- Ensure recoverability after each write to storage
Our solution: Multi-step Fault-tolerant Workflow

Player_A

Player Storage

Workflow queue

Storage

Local workflow queue

HashIndex on Player.Location

HashIndex on Player.Location in Storage
Our solution: Multi-step Fault-tolerant Workflow

1. Add update to queue
2. Batch write to Storage
3. Update storage including workflow record ID
4. Batch update the index
5. Check if Player has the workflow record, too
6. Remove workflow record ID

Player A

Workflow queue Storage

HashIndex on Player.Location

HashIndex on Player.Location in Storage
Index Physical Representation

Entire index in one actor

One index-actor per index bucket

One index-actor per server

HashIndex on Player.Location for actors on Server 1

HashIndex on Player.Location for actors on Server 2
public class PlayerProperties
{
    public int Rank { get; set; }
    [Index]
    public string Location { get; set; }
}

public class PlayerState
{
    public string Name { get; set; }
    public int Rank { get; set; }
    public string Location { get; set; }
}

public interface IPlayer : IIndexableGrain<PlayerProperties>
{
    public Task Move(Direction d);
    public Task<string> GetLocation();
}

public class Player : IndexableGrain<PlayerState, PlayerProperties>, IPlayer
{
    public Task Move(Direction d)
    {
        State.Location = d.GetDestination(State.Location);
        return WriteStateAsync();
    }
    public Task<string> GetLocation()
    {
        return Task.FromResult(State.Location);
    }
}
IOrleansQueryable<IPlayer> activePlayersInRedmond =
    from player in GrainFactory.GetActiveGrains<IPlayer, PlayerProperties>()
    where player.Location == "Redmond"
    select player;

//IOrleansQueryable extends IQueryable interface
foreach(IPlayer player in activePlayersInRedmond)
{
    Console.WriteLine(player.GetPrimaryKeyLong());
}
Future Work on Indexing

- Transactionally update actor and index
- Range indexes
- Richer materialized views
- Offer indexing with other AODB features, e.g., transactions, queries, geo-dist’n
Status of Orleans’ AODB Features

- Stream processing (January 2015)
- Geo-distribution and multi-master replication (January 2016)
- Distributed transactions (preview, this month) [MSR Technical Report]
- Indexing (prototype, August 2016)
Acknowledgments

- Sebastian Burckhardt, Sergey Bykov, Julian Dominguez, Tova Milo, Jorgen Thelin, Microsoft Studios and the Orleans community.

- More at https://dotnet.github.io/orleans/
Thank you!