An Actor Database System for Akka

Frederic Schneider[1], Sebastian Schmidl[2]

Abstract: Abstract goes here

Keywords: Actor Database System; Akka; Scala; Domain Driven Design

Paper Goals (for us):

- Concept / Approach: Domain design with actors (How does a Actor Database System look like in our view?)
- Memory overhead of actor-design? feasible
- Queries between actors and optimization through concurrency

1 Introduction

Most web-based applications process data at ever growing rates. Whether user logins, the processing and storage of sensor data, or dynamic information provisioning, data is at the center of online applications. Cluster or cloud deployments and multi-core hardware architectures allow scaling application logic in terms of computational power. Data management systems, however, are at risk of becoming the bottleneck in data-centric software systems. Such applications create an increasing demand for scalable data storage solutions, that are capable of fulfilling real-time requirements for OLTP workloads at low latency.

Traditionally, systems' architectures for these types of software is comprised in two tiers: An application logic tier containing the bulk of the business logic, as well as a data tier, which stores the application's data. The separation of concerns within this design in terms of concrete functionality depends on the choice of data storage solution. A relational database management system (RDBMS) enables data manipulation and the execution of most data-related computation in the data tier, by using e.g. stored procedures and a complex, declarative query language. However, the monolithic design of conventional RDBMSs

[1] Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, frederic.schneider@student.hpi.de
[2] Hasso-Plattner-Institut, University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, sebastian.schmidl@student.hpi.de

proves a hurdle for scalability and reduces modularity, which has a negative impact on code maintainability. More recently key-value storage solutions have gained popularity especially for highly scalable online applications. This is due to the fact that their less rigid schema guarantees allow for improved scalability. The trade-off of this solution is having to offload much of the data-handling logic into the application tier, thereby increasing load on the application as well as not keeping true to the separation of concerns.

In either case, the data and application tier do not necessarily share a format for data objects. Application logic and database exhibit differences in available data types and modelling capabilities. Object-oriented programming languages provide concepts, such as inheritance, polymorphism, and nested objects. They enforce strict object encapsulation. RDBMSs are not able to express all of the same concepts due to the representation of data as relations in tables. While data storage solutions with a less rigid schema, such as document-oriented databases, are able to represent e.g. nested objects, they still face the same problems as RDBMSs with regard to data formats, inheritance, and encapsulation. The described set of difficulties when persisting application state and data objects in a database is commonly referred to as *object-relational impedance mismatch*. The use of object-relational mapping (ORM) tools, such as Hibernate for Java or Active Record for Ruby on Rails, is one approach to provide a middle tier for mapping between the data formats of the application and data tier.

Shah; Salles call for a new paradigm for designing a scalable data storage solution using the Actor model [SS17a]. The core primitive in this model are actors, which are objects comprised of state and behaviour that execute computational tasks concurrently. Individual actors communicate with each other exclusively via asynchronous message passing. Incoming messages are stored in a mailbox allowing for the separate and independent processing of each message. When receiving a message an actor can modify its state, send a finite amount of messages to other actors, and create new actors. An actor's internal state is only available to said actor which encourages a shared-nothing system architecture. The self-contained nature of actors and the fact that, through asynchronous message passing, actors provide a lock-free concurrency model, allows for naturally scaling out applications and systems [Ve15]. Shah; Salles propose an implementation of the data tier using the actor model as a logically distributed runtime using the actor model. They predict that this programming abstraction will allow for a modular, scalable application design.

We build on this idea and present an application development framework for actor-based data-centric applications. In this framework actors employ relational structures internally to manage application state and data and provide functionality based on this data in the form of actor behavior. Since these actors are not part of a dedicated database runtime, but are defined and instantiated using the application framework, this approach dissipates the strict line between data and application tier. It allows for utilizing and interacting with data objects in the same representation throughout business logic and data storage.

To our knowledge, we present the first partial implementation of this concept using the

Akka framework. Comparable, existing work has been presented using the virtual actor concept provided by the Orleans framework for Microsoft .NET. Therefore, we discuss the differences of the two concepts, such as the need for explicit actor creation, and manual actor lifecycle management in section 2. In this section, we also give an outline of the history of Actor model implementations as well as the characteristics of *Actor Database Systems* as proposed by Shah; Salles. In section 3 we outline our approach conceptually in more detail, before explicitly describing the implementation of our proof-of-concept framework, as well as some examples for usage patterns in section 4. We present setup and evaluation of experiments on the memory overhead introduced by the use of actors for data storage using our framework in section 5, and offer a concluding statement about this and future work in 6.

2 Related Work

The Actor model, as described in the previous section, denotes a mathematical concept for concurrent computation which dates back to the early 1970's. It has gained popularity and been implemented anew as language extensions or frameworks for existing object-oriented and functional languages, such as C#, Java, and Scala, in the mid-2000's. In this sections we give an outline of the recent history of Actor model implementations, discuss differences of the conventional actor model and the virtual actor model used in the Orleans framework, and elaborate on the notion of *Actor Database Systems* and their requirements.

2.1 Actor Model

Various programming languages and libraries implement the actor model and provide the corresponding functionality: The Erlang programming language supports concurrency using the actor programming model and is designed for distributed and highly available systems in the telecommunication sector [Ar07]. The Scala programming language's standard library included an actor model implementation for concurrent computation starting in 2006, which has now been deprecated for the Akka library [Ha12]. Akka implements the actor model for the Java Virtual Machine (JVM) and can be used with the languages Java and Scala [Li18a]. Akka.Net is a community-driven port of the Akka toolkit for the .NET platform in the languages C# and F# [Ak17].

2.2 Virtual Actor Model

While the actor programming model allows for low-level interaction with the actor lifecycle management, handling race conditions, physically distributing actors, and failure handling, the virtual actor programming model raises the abstraction level of actor objects. It still

adheres to the plain actor model, but treats actors as virtual entities. In this model actor lifecycle management and distribution is managed by the framework or runtime and not by the application developer. The physical location of the actor is transparent to the application. This model is comparable to virtual memory, which allows processes to access a memory address whether it is currently physically available or not.

The virtual actor model was introduced with the Orleans framework for .NET by Microsoft Research [Be14]. In Orleans, actors (virtually) exist at any time. There is no possibility to create an actor explicitly. Instead, this task is handled by the language runtime, which instantiates actors on demand. The runtime's resource management deals with unused actors and reclaims their space automatically. The Orleans runtime guarantees that actors are always available. This means, that in the view of an application developer an actor can never fail. The runtime will deal with crashing actors and servers and recreate missing actor incarnations accordingly.

The corresponding framework for the JVM is Orbit [El18]. In Orbit, a virtual actor can be active or inactive. But those two states are transparent to the application developer. Similar to Orleans, messages to an inactive actor in Orbit will activate it, so the actor can receive the sent message. Actor state in Orbit is usually stored in a database, which allows for activation and deactivation of actors at runtime. During activation of an actor, the state is recovered from persistent database storage.. Before the actor is deactivated, its state is written back to the database.

Akka provides similar functionality with Akka Cluster Sharding [Li18b]. It introduces the separation of virtual and physical Akka actors. Actors are first grouped into shards, which can then be located at different physical locations. The distribution of the shards and their actors is managed by the framework and it also deals with the routing of the messages and re-balancing the shards to different nodes based on different strategies. A developer must not know, where a specific actor is physically located to send messages to it. Akka Cluster Sharding also supports passivation of actors. This means that unused persistent actors can be unloaded to free up memory (passivation). A message to passive actors will instruct the framework to load it back into memory.

In this paper we denote virtual actors as vactors to clearly distinguish actors of the actor model (actor) from those of the virtual actor model (vactor).

2.3 Actor Database Systems

The actor and virtual actor programming model is increasingly used to build soft caching layers and cloud applications for various purposes [Er18; Li18c; NE18]. This shows the appeal of the programming model that allows developers to easily develop modular and scalable software, which can be deployed in the heterogenous parallel cloud computing infrastructure.

Despite its popularity for application development, the actor programming model and actor programming frameworks lack state management capabilities, specifically for data-centric applications. The developer has to decide how to handle state persistence and how to satisfy the failure and consistency requirements of the application, because actor model implementations do no provide atomicity or consistency guarantees for state across actors. Shah; Salles [SS17a] find a need for state management guarantees in actor systems and propose to integrate actor-based programming models in database management systems. The authors postulate that *Actor Database Systems* should be designed as a logical distributed runtime with state management guarantees. They should allow for and encourage the design of modular, scalable and cloud-ready applications. Shah; Salles outline four tenets that identify an *Actor Database System* and specify needed features in their manifesto [SS17a]:

Tenet 1 Modularity and encapsulation by a logical actor construct

Tenet 2 Asynchronous, nested function shipping

Tenet 3 Transaction and declarative querying functionality

Tenet 4 Security, monitoring, administration and auditability

Practical work on actor database systems has been done using the Orleans framework. Shah; Salles tightly follow the propositions of the manifesto and present ReactDB, an in-memory OLTP database system that is programmed via application-defined actors with relational semantics, called reactors [SS17b]. A reactor is a logical entity encapsulating state as relations, with asynchronous function processing capability, which typically represents a application-level scaling units. Within individual reactors classic declarative querying can be used, but across actors explicit asynchronous function calls must be defined. Shah; Salles show that transactions across reactors can have serializability guarantees. Eldeeb; Bernstein [EB16] introduce a new transaction protocol for distributed actors improving the throughput compared to the two-phase commit protocol. Bernstein et al. [Be17] explore indexing mechanisms in an actor database system. They propose a new indexing architecture based on the Orleans framework that is fault-tolerant and eventually-consistent.

Biokoda takes a different approach to combine actor programming models and relational database systems. In their development actorDB, an actor encapsulates a full relational SQL database using the SQLite database engine[3]. Actors can be deployed in an distributed cloud environment and communicate asynchronously. The database is ACID-conform and enforces consistency across actors with the Raft consensus protocol [OO14]. The database can used via the MySQL-protocol

Most practical research in the field of actor database systems has been presented in conjunction with the Orleans framework and the Erlang programming language. The open-source project *Actorbase* by Cardin constitutes a key-value store implementation utilizing

[3] `https://www.sqlite.org/about.html`

Akka actors for horizontal scaling and partitioning [Ca17]. In this research database system, partitions are managed by storekeeper actors while inter-partition lookup and indexing is managed by so-called storefinder actors. This approach is limited to a key-value store function set and does not provide functionality corresponding to the relational features discussed earlier.

We base our work on the concept of actor database systems as introduced by Shah; Salles [SS17a]. Similar as in [SS17b], actors in our approach consist of state abstracted as relations and asynchronous function processing logic. The application developer defines actors and the asynchronous functions. Different to previous approaches, we use Scala as programming language and Akka as the reference actor model to implement an framework for actor database systems, which is explained in section 4.

3 Domain Actor Database Concept

3.1 Domain Actors

3.2 Actor Database Systems

4 Domain Actor Database Framework

After describing the general concept of an *Actor Database System*, we present the implementation of an application development framework which allows for the definition of an application's data model and the provisioning of corresponding Dactors as part of the application runtime.

The goal for this research implementation is a proof-of-concept framework which allows developers to declaratively define their application's data model and which automatically instantiates special-type actors, i.e. Dactors, corresponding to defined domain-object entities. These Dactors encapsulate the respective application data and provide a SQL-like interface for data access and manipulation internally. This interface is used to define actor behaviour and functionality. Functionality that relies on data contained within multiple Dactor objects is defined explicitely using Functor objects which have been described conceptually in the former section 3. These Functors allow to define inter-Dactor functionality without breaking the strictly asynchronous communication between actors or halting an actors single-threaded computation process.

This framework's feature set relates to three of the four core tenets that define an *Actor Database System* according to Shah; Salles's manifesto [SS17a]: **Tenet 1** calls for the use of actors to achieve a modular logical model for data encapsulation. Dactor instances are in-memory storekeepers for an application's data and provide an actor construct which encapsulates data. Furthermore, Dactors provide a model for concurrent computation of

predefined functionality that processes a Dactor's data. This is part of the requirements described in **Tenet 2**. Functors allow for the definition of functionality using multiple actors's data and are excuted asynchronously. The possibility to predefine these Functors as well as Dactor behaviour relates to the second part of **Tenet 3**. Due to their single-threaded computation model, Dactors can be argued to enforce internal consistency by default. In principle, Functors allow for the implementation of further transaction protocols to ensure cross-Dactor consistency guarantees. Since the requirements listed under **Tenet 4** relate mostly to production-ready database products, we do not implement or investigate them in this research project. The functionality described by Shah; Salles in tenets one through three allows for investigation of the practical feasibility and implications of the actor database concept.

The proof-of-concept framework has been developed using the Scala programming language and the Akka framework for actor programming. This implementation marks, to the best of our knowledge, the first practical examination of the actor database concept using Akka and the classical actor concept. While the Orleans framework, having been used in earlier related work, provides virtual actors with automatic lifecycle management, Akka allows for and requires direct control over an actors lifecycle. These considerations directly apply to Dactors as they are built on top of the Akka actor implementation.

4.1 Internal Data Management

In-memory data that is contained within a Dactor instance is managed in a data structure called relation. Relations contain a multiset of records or tuples which follow a pre-defined schema. They provide a type-safe, SQL-like interface for querying and manipulating contained data. The features and interface they provide is inspired by SQL to allow for simple data manipulation using known and proven syntax and semantics when defining a Dactors behavior.

A relation, similarly to a table in the relational database model, is defined as a multiset of tuples following a predefined schema. This schema is comprised of column definitions whereby every column is defined by a name and data type. All manipulation and query functionality on a relation's data provide compile-time type safety based on these column's predefined data types. Due to the Dactor system sharing the application's runtime and programming environment, these data or object types are equal to the types handled in any application logic. Thus, this approach eliminates the impedance mismatch between application logic and data tier with regard to handled data types and object (de-)serialization.

Due to the similarities between relations and tables known from the relational database model, we want to discuss the conceptual differences between the two. While relations can be used to model much of the same information as is commonly represented using tables, the *Actor Database System* and specifically the concept and framework we present relies on Dactors as models for application domain objects.

For clarification consider the following example of a web application with information on films and (real-life) actors similar to the imdb.com or rottentomatoes.com websites, a traditional data layout might be comprised of two tables containing information on films and actors, respectively. Using Dactors to represent discrete domain-level objects, a single Dactor of the type film represents a single, specific film and encapsulates all corresponding data. This approach to layouting an application's data results in much smaller relation sizes than typical table sizes and enables many business-logic–driven approaches to scaling, data partitioning, and caching, which are discussed in section 4.4.

4.2 Domain Actor Design

The presented framework provides the abstract Dactor class which is extended by developers to model their application's domain object classes which encapsulate application data. Instances of such user-defined Dactor types are managed by the framework and are available for messaging in a consistent namespace. A Dactor type definition is comprised of the types internal data schema, i.e. relations, and it's functionality to process on said data. Note that, since Dactors can logically encapsulate multiple data sets and state information, they can and often do comprise multiple relations.

To illustrate, we present the Dactor definition in terms of data model and functionality using the presented framework with regard to the formerly considered example. This application in it's simplest form comprises two Dactor types modelling the concerns of the domain-specific object types of Film and Actor. We capitalize these Dactor types to eliminate the ambiguity of the Dactor type Actor and the general actor concept within the actor model.

Film Dactor instances comprise information on a specific stored film, such as its title, release date, a description, and a unique id, but also information about its starring actors: The actor's (person) name, the name of the role they portray, as well as the corresponding Actor instance's id. The Actor instances comprise additional information on the person such as the date of birth, but also a list of films they starred in, including the name of the role they portray, the film's title, and the Film's id. The definition of the Film Dactor's data model is shown in listing 1.

```
object Film {
  /**
   * Definitions of relations contained in Film type Dactors.
   */
  object Info extends RelationDef {
    // column with name ``title'' and data-type String:
    val title = ColumnDef[String](``title'')
    val description = ColumnDef[String](``description'')
    // rich datetime data-type column:
```

```
      val release = ColumnDef[ZonedDateTime](``release'')
    }

    object Cast extends RelationDef {
      // foreign key relating to an Actor type Dactor instance:
      val actorId = ColumnDef[Id](``actor_id'')
      val name = ColumnDef[String](``actor_name'')
      val rolename = ColumnDef[String](``role_name'')
    }
  }

  /**
   * Dactor type definition.
   */
  class Film(id: Id) extends Dactor(id) {
    override protected val relations = {
      // special relation type ensuring domain-logic constraint
      Film.Info -> SingleRowRelation(Film.Info),
      Film.Cast -> RowRelation(Film.Cast)
    }
  }
```

List. 1: Film Dactor type definition using the actordb framework

List. 2: Actor Dactor type definition using the actordb framework

Note the partial denormalization of data on the respective domain objects into two Dactors. This enables the computation and response to common queries, such as requesting info on a film and their cast list, from a singular Dactor instance. More complex queries, e.g. finding out which films star the same actors as a given film, involve data and computation on multiple Dactor instances of both Dactor types. The tradeoff for this is that inserting new information about an actor (person) starring in a film becomes a functionality involving two Dactor instances and types. This tradeoff and implementation examples for this functionality are discussed in the following section 4.3.

4.3 Multi-Dactor Queries

Explicit messaging and complementary actor behaviors are used to implement the cascading communication pattern described in section 3. Besides this, Functors are the framework's system which enables inter-Dactor communication and functionality. While the basic functionality of the two Functor types for sequential and concurrent computation is

discussed in 3, this section focuses on the definition of a concrete function integrating data from multiple Dactors. We also discuss the implications of using the respective messaging patterns corresponding to these types of Functors on computational scalability.

Returning to the web application example we can define simple functionality concerning instances of both Dactor types defined in section 4.2. Adding a new character role to a film, played by a specific actor involves changes to Dactor instances of the types Film and Actor. We describe the implementation of this functionality using all three of the introduced communication patterns.

4.4 Discussion

5 Experiments

5.1 Actor Memory Overhead

5.2 Query Performance

5.3 Query Optimization

5.4 Discussion

6 Conclusion

7 Example

This example is from the original LNI documentation and therefore in German:

Referenzen mit dem richtigen Namen (*Table*, *Figure*, ...): Sect. 1 zeigt Demonstrationen der Verbesserung von GitHub-LNI gegenüber der originalen Vorlage. Sect. 7 zeigt die Einhaltung der Richtlinien durch einfachen Text.

Referenzen sollten nicht direkt als Subjekt eingebunden werden, sondern immer nur durch Authorenanganben: Beispiel: Abel; Bibel [AB00] geben ein Beispiel, aber auch Azubi et al. [Az]. Hinweis: Großes C bei `Citet`, wenn es am Satzanfang steht. Analog zu `Cref`.

Fig. 1 zeigt eine Abbildung.

Hier sollte die Graphik mittels `includegraphics` eingebunden werden.

Fig. 1: Demographik

Tab. 1 zeigt eine Tabelle.

Überschriftsebenen	Beispiel	Schriftgröße und -art
Titel (linksbündig)	Der Titel ...	14 pt, Fett
Überschrift 1	1 Einleitung	12 pt, Fett
Überschrift 2	2.1 Titel	10 pt, Fett

Tab. 1: Die Überschriftsarten

Die LNI-Formatvorlage verlangt die Einrückung von Listings vom linken Rand. List. 3 zeigt uns ein Beispiel, das mit Hilfe der `lstlisting`-Umgebung realisiert ist. Referenz auf `print("Hello World!")` in Line 6.

```
/**
  * Hello World application!
  */
object Hello {
  def main(args: Seq[String]): Unit = {
    println("Hello␣World!")
  }
  def math: Unit = $\frac{\sqrt{1-x^2}}{x_i+\lambda} - \infty$
}
```

List. 3: Beschreibung

Die korrekte Einrückung und Nummerierung für Formeln ist bei den Umgebungen `equation` und `align` gewährleistet.

$$1 = 4 - 3 + x \tag{1}$$

und

$$2 = 7 - 5 \tag{2}$$

$$3 = 2 - 1 \tag{3}$$

References

[AB00] Abel, K.; Bibel, U.: Formatierungsrichtlinien für Tagungsbände. Format-Verlag, Bonn, 2000.

[Ak17] Akka.NET project: Akka.NET, 2017, URL: https://getakka.net/, visited on: 08/15/2018.

[Ar07] Armstrong, J.: A History of Erlang. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages. HOPL III, ACM, San Diego, California, pp. 6-1–6-26, 2007, ISBN: 978-1-59593-766-7, URL: http://doi.acm.org/10.1145/1238844.1238850.

[Az] Azubi, L. et al.: Die Fußnote in LNI-Bänden. In. Pp. 135–162.

[Be14] Bernstein, P. A.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability. MSR-TR-2014–41/, 2014.

[Be17] Bernstein, P. A.; Dashti, M.; Kiefer, T.; Maier, D.: Indexing in an Actor-Oriented Database. In: CIDR. 2017.

[Bi18] Biokoda: ActorDB – 1. About, 2018, URL: http://www.actordb.com/docs-about.html, visited on: 08/16/2018.

[Ca17] Cardin, R.: Actorbase, or "the Persistence Chaos", 2017, URL: https://dzone.com/articles/actorbase-or-quotthe-persistence-chaosquot, visited on: 08/18/2018.

[EB16] Eldeeb, T.; Bernstein, P. A.: Transactions for Distributed Actors in the Cloud. In. 2016.

[El18] Electronic Arts Inc.: What is Orbit?, 2018, URL: https://github.com/orbit/orbit/wiki, visited on: 08/15/2018.

[Er18] Erlang Solutions Ltd: Case Studies & Insights, 2018, URL: https://www.erlang-solutions.com/resources/case-studies.html, visited on: 08/17/2018.

[Ha12] Haller, P.: On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. In: Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions. AGERE! 2012, ACM, Tucson, Arizona, USA, pp. 1–6, 2012, ISBN: 978-1-4503-1630-9, URL: http://doi.acm.org/10.1145/2414639.2414641.

[Li18a] Lightbend, Inc.: Akka, 2018, URL: https://akka.io/, visited on: 08/15/2018.

[Li18b] Lightbend, Inc.: Akka Documentation – Cluster Sharding, 2018, URL: https://doc.akka.io/docs/akka/current/cluster-sharding.html?language=scala#cluster-sharding, visited on: 08/16/2018.

[Li18c] Lightbend, Inc.: Lightbend Case Studies, 2018, URL: https://www.lightbend.com/case-studies#filter:akka, visited on: 08/17/2018.

[NE18] .NET Foundation: Who Is Using Orleans?, 2018, URL: https://dotnet.github.io/orleans/Community/Who-Is-Using-Orleans.html, visited on: 08/17/2018.

[OO14] Ongaro, D.; Ousterhout, J. K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference. Pp. 305–319, 2014.

[SS17a] Shah, V.; Salles, M. V.: Actor Database Systems: A Manifesto. CoRR abs/1707.06507/, 2017.

[SS17b] Shah, V.; Salles, M. V.: Reactors: A Case for Predictable, Virtualized OLTP Actor Database Systems. CoRR abs/1701.05397/, 2017.

[Ve15] Vernon, V.: Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka. Pearson Education, 2015.