An Actor Database System for Akka

Frederic Schneider, Sebastian Schmidl[1]

Abstract: Abstract goes here

Keywords: Actor Model; Actor Database System; Akka; Database; Distributed Computing; Parallelization

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 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

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

capabilities. Object-oriented Programming (OOP) 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 [SS17a] call for a new paradigm for designing a scalable data storage solution using the Actor model. 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. 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.

With this framework we make the following contributions: We introduce domain actors (Dactors) to model data-centric applications' data and act as an application-defined scaling unit. Dactors encapsulate application data and logic, therefore bridging the aforementioned impedance mismatch between data and the application logic tier. We implement a SQL-like data interface for defining internal Dactor logic, and provide Functors to allow users to define application logic relying on data contained within multiple Dactors. We also present experimental results on the memory overhead introduced by using Akka actors for data management in this manner to demonstrate the feasibility of this approach.

To our knowledge, we present the first 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 differences between the classical notion of actors and the virtual actor concept in Sect. 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 Sect. 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 Sect. 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 Sect. 5, and offer a concluding statement about this and future work in Sect. 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 been 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 [SS17a] outline four tenets that identify an *Actor Database System* and specify needed features in their manifesto:

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 [SS17b] 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. 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 [Bi18] 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 be 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 [Ca17] constitutes a key-value store implementation

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

utilizing Akka actors for horizontal scaling and partitioning. 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.

3 Domain Actor Database Concept

Our concept is based on the idea of actor database systems as introduced by Shah; Salles [SS17a]. The combination of application and data tier provides considerable advantages for data-centric systems. This architecture allows leveraging domain and application knowledge to dynamically model the data layout, especially concerning data partitioning and replication schemes. This makes the database system modular, cloud-ready and scalable.

3.1 Domain Actors

Similar to Shah; Salles [SS17b], we introduce a special type of actor, called Dactor, that acts as an application-defined scaling unit. Dactors can be used to model application-domain objects and encapsulate the object's state and application logic in an actor. Using actors for this enforces technical encapsulation of state access due to the purely private state in actors and the need of explicit asynchronous messaging between the actors. This makes it easier to reason about state changes, bugs and other failures, as only code within the Dactor can change the corresponding state.

In-memory data contained within a Dactor instance is managed in a data structure called relation. A relation, similarly to a table in the relational database model, is defined as a multiset of tuples following a predefined schema specifying the metadata. 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.

We consider the following example of a web application with information on films and (human) 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. In contrast to the relational model, our model, shown in Fig. 1, is denormalized and distributed across two different Dactor types. Both, `Film` and `Actor` Dactor instances contain two relations and store information about a specific film or actor, respectively. In addition, `Actor` instances keep some information on related `Films`, i.e. those the `Actor` has starred in, including the `Film`'s Dactor id, and vice-versa. This realises a many-to-many relationship between both Dactor types.

Using Dactors to represent discrete domain-level objects, a single Dactor instance of the type film represents a single, specific film and encapsulates all corresponding data. This approach to layout an application's data results in much smaller relation sizes than typical table sizes, since each Dactor instance only stores data related to one specific domain entity, and enables many business-logic-driven approaches to scaling, data partitioning, and caching. The tradeoff to take into account, however, is a large number of Dactor instances encapsulating the application's data and a denormalized schema.

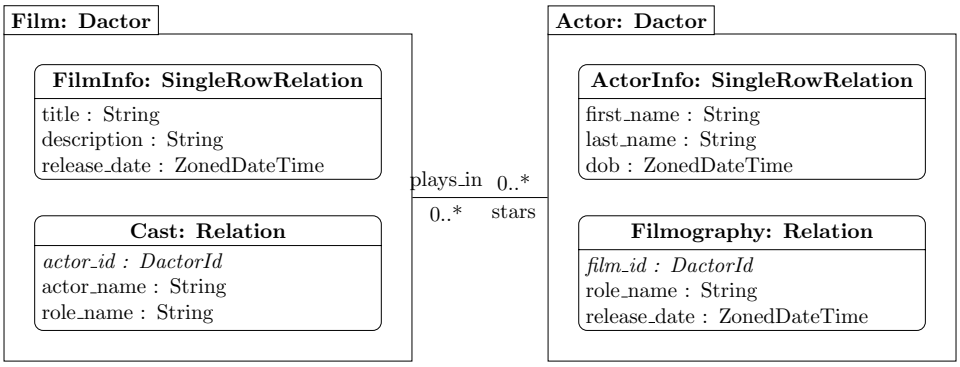


Fig. 1: Class diagram of the `Film` and `Actor` Dactor types defined in the example application. Both Dactor types contain two relations. Dactor id attributes in the relations are italised and realise a many-to-many relationship between the two Dactor types.

As Dactors not only contain data, but also the corresponding domain logic, computation is executed concurrently. This supports designing a modular and extensible database system and improves scalability. Actors provide single-threaded semantics, which makes enforcing constraints on data tied to one domain object stored inside Dactors no issue. Relations provide an SQL-like interface to query and manipulate the contained data, so known and proven syntax and semantics can be used to define Dactor-behavior. While state querying and modification within Dactors is possible in a declarative way, communication across all kind of actors is explicitly defined via asynchronous messages and how the Dactors handle the said messages.

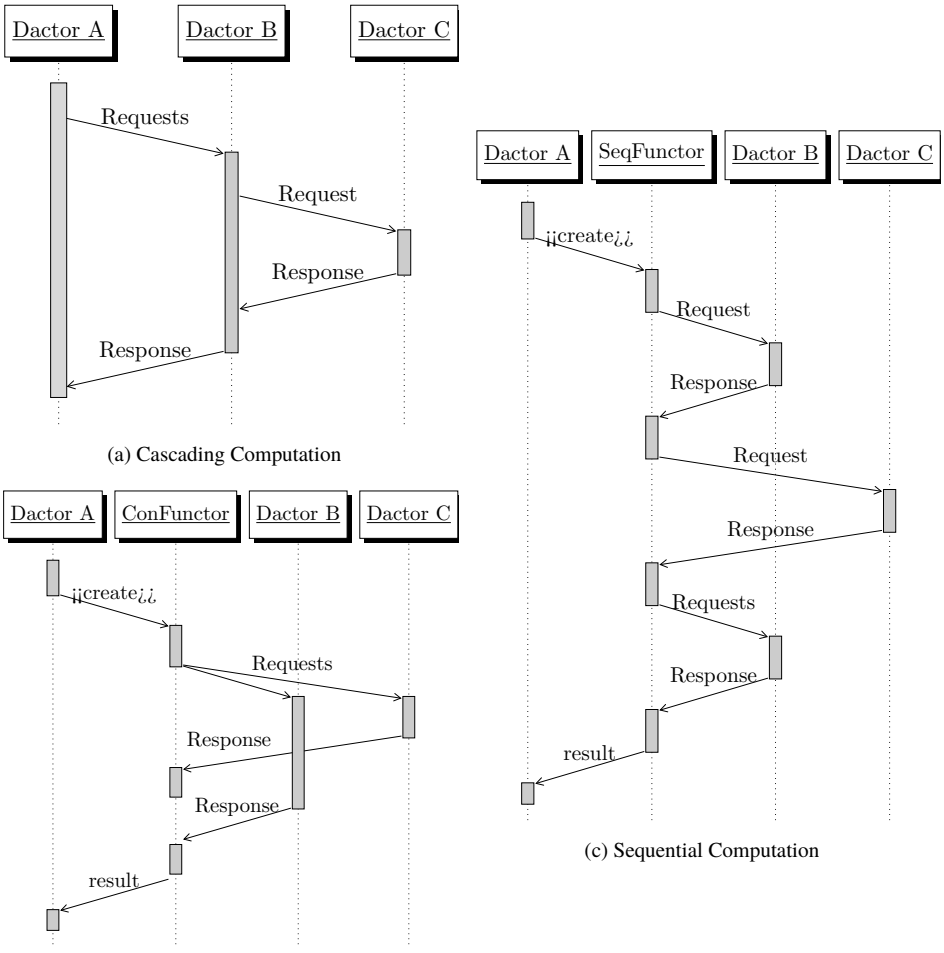
3.2 Communication between Domain Actors

Not all computation can be done with the information residing in a single Dactor. Communication between Dactors is required. As already mentioned, inter-Dactor communication is realized via explicit asynchronous message passing. This allows application developers to chose the right messaging pattern for their use case. The choice is heavily influenced by the data layout used for the Dactors and sets the level of parallelization and the network-load for the computation.

We consider three main messaging patterns for inter-Dactor communication, which are shown in Fig. 2: Cascading Computation, Sequential Computation, Concurrent Computation. They are messaging primitives and can be combined to create more complex message flows and computation models. All three patterns assume a request-response messaging schema and take a high-level message to a Dactor, lets call it Dactor A, as starting point.

1. **Cascading Computation** A high-level message to Dactor A, will trigger successive messages to other Dactors, which are hidden from the original requester. Depending on the computation performed in Dactor A, it can send requests to another Dactor (e. g. Dactor B) and handle the result itself before returning the response to the caller. Dactor B can itself send messages to other Dactors as well.
   As one can see in Fig. 2a, this pattern is comparable to function calls in OOP. But contrary to simple function calls, messages in this pattern are sent asynchronously. This means that the requesting Dactor has to manage the state of pending responses, such as actor references and intermediate results, or failure cases. This clutters the domain logic in the Dactor and leads to complex and error-prone code. In addition, each message introduces network latency and can therefore increase the overall computation time compared to a single Dactor computing the response. If used sparsely, this pattern supports separation of concerns and the tell-don't-ask paradigm.
2. **Sequential Computation** Fig. 2c shows the sequential computation pattern. It is used for computations that consist of consecutive steps, where each step depends on the previous step's result, such as filter chains.
   For this and the next messaging pattern we introduce the concept of actor-level Functors. Functors encapsulate the processing of a high-level request in a new short-living actor. They are able to communicate with other Dactors, track the state of pending requests and handle failure cases. Every actor can create a new Functor to encapsulate multiple depending requests to Dactors. The Functor handles the message processing and sends the final result or a failure message back to its parent actor before stopping itself. This reduces the code complexity in Dactors for handling such messages. This message processing abstraction is also called cameo pattern[4].
   Using a Functor to process the consecutive steps of the computational chain relieves Dactor A from dealing with intermediate state, because it is managed by the Functor. Each Functor only has to deal with one request-response pair at a time, which leads to a simple state and processing logic for the Functor itself.
3. **Concurrent Computation** This messaging pattern also makes use of Functors to encapsulate the processing of multiple request-response pairs. The concurrent Functor sends messages to several Dactors in parallel and collect the results when they are finished. It allows for highly parallelized computation as all involved Dactors are messaged at the same time and calculate their responses concurrently. The Functor collects each individual response and returns the aggregated result to it's creator.

[4] `http://blog.simplex.software/the-cameo-pattern`



(a) Cascading Computation

(c) Sequential Computation

(b) Concurrent Computation

Fig. 2: Inter-Dactor-communication patterns. Dotted lines show actor lifetime and gray bars indicate that an actor performs computation or holds state that is related to the showed message flow.

3.3 Domain Actor Database System

A domain actor database system is the combination of Dactors and other types of actors, which together offer an interface to other applications or clients. The domain actor database system is a distributed runtime for persisting data and performing computations on it. Each Dactor must incarnate a specific Dactor type. This type is defined by application developers and determines the data schema encapsulated by the Dactor and the messages it can handle (its behavior). Dactors are the only place in the system that actually have persistent state. Other actors are stateless or have non-durable state, such as intermediate results or actor references. These actors can interact with the Dactors via the defined messages and represent application services or components of the infrastructure layer.

To develop such a system, we can decompose our application domain into encapsulated entity types. Those types can be modeled as Dactors and consist of a data schema and behavior. During this process we have to keep our requirements regarding query patterns and scaling in mind. Those requirements will influence the domain decomposition and the messaging patterns used. The entry points to our application are stateless actors on top of the Dactors, which provide high throughput and can be scaled out easily. They act as routers for incoming requests and orchestrate computations.

4 Domain Actor Database Framework

After describing the general concept of an *Actor Database System*, we present the implementation of a proof-of-concept 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 framework provides the aforementioned features for developers:

- Declarative data model definition using Dactors
- SQL-like interface for accessing and manipulating internal data within Dactors
- Functor objects for procedures, which rely on data contained in multiple Dactors

These features relate 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 application data. They satisfy the actor definition and support high modularity. Furthermore, Dactors provide a model for concurrent computation of predefined functionality that increases locality of data accesses. This is part of the requirements described in **Tenet 2**. All communication between Dactors is asynchronous to leverage the advantages of increasingly parallel hardware. Functors allow for the definition of

functionality using multiple actors' data and are executed asynchronously as well. The possibility to define these Functors as well as Dactor behavior in a declarative way 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 inter-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 suffices the investigation of the practical feasibility and implications of the actor database concept.

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

The direct control of the Dactors lifecycles, in principle, enables close configuration of the in-memory state of the database system. Frequently accessed Dactors can be kept in memory while ones, which haven't been involved in recent computational effort, could be temporarily persisted to disk. The virtual actor system implemented in the Orleans framework withdraws such control, making custom, database specific logic for configuring Dactor availability impossible.

4.1 Domain Actor Design

Developers can model the application's domain objects by defining Dactor types as subclasses of the framework-provided `Dactor` class. 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 layout, i.e. relations, and its functionality to process said data. In detail, a Dactor types internal data model definition consists of relations with unique schemas. A relation's schema is comprised of column definitions, whereby every column is defined by a name and a data type. Using the column's predefined data types, all functions support compile-time type-safety. 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 helps eliminate the impedance mismatch between application logic and data tier with regard to handled data types and object (de-)serialization. Note that, since Dactors can logically encapsulate multiple data sets and state information, they can and often do comprise multiple relations.

To illustrate the definition of Dactors we use the example introduced in Sect. 3.1 which is schematically depicted in Fig. 1. The corresponding definition of the Film's data model is shown in List. 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)
  }
  override def receive: Receive = ??? // Dactor behavior
}
```

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

The partial denormalization of information, which relates to Actor instances, into the Film.Cast relation of Film type Dactors, enables the computation of and response to common queries, such as requesting info on a film and their cast list, from within a single Dactor instance. The trade-off for this is that inserting new information about an actor (person) starring in a film becomes a functionality involving two Dactor instances. This trade-off and implementation examples for this functionality are discussed in the following Section 4.2.

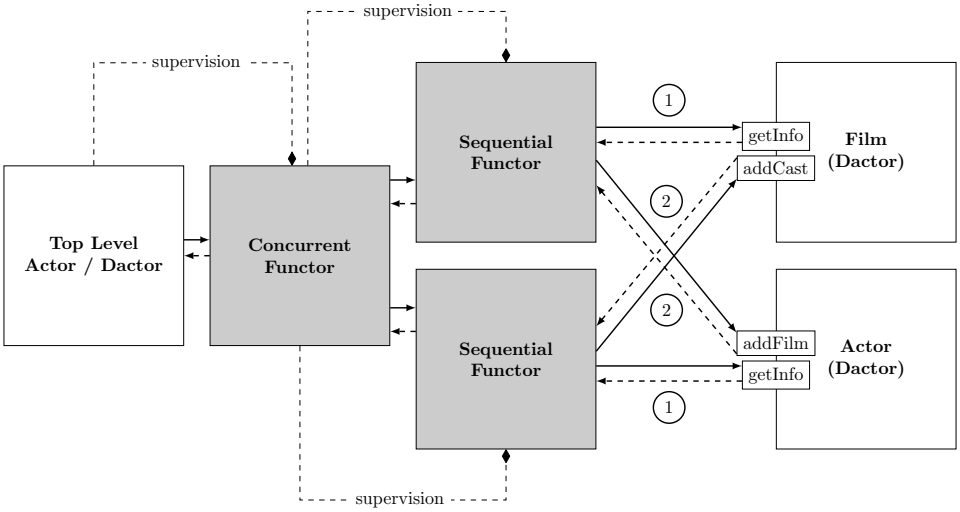


Fig. 3: Component diagram indicating the message flow through Functor objects and their supervision by the calling actor or Dactor. Arrow and dashed arrow pairs indicate corresponding request and response messages. The outgoing requests of each sequential functor are numbered to indicate their order.

4.2 Multi-Dactor Queries

Explicit messaging and complementary actor behavior are used to implement the cascading communication pattern described in Section 3. Besides this, Functors are the framework's system, which enable inter-Dactor communication and computations. While the basic functionality of the two Functor types for sequential and concurrent computation is discussed in Section 3, this section focuses on the definition of a concrete function integrating data from multiple Dactors.

Functors are actors with a limited lifetime. They are instantiated once a defined Functor is called by a Dactor or other actor and deleted once they return the final response or encounter a failure. Functors are always called from an Akka actor as a child actor. This Akka-specific hierarchical relationship enables notifying the calling actor even in case of an unforseen crash of the Functor themselves which in turn allows to trigger error handling, e.g. retrying the Functor execution.

Returning to the web application example, we can define simple functionality concerning instances of both Dactor types defined in Section 4.1. Adding a new character role to a film, played by a specific actor involves changes to `Film` and `Actor` instances. We describe the implementation of this functionality using all three of the introduced communication patterns.

Fig. 3 displays the message-flow between the involved Dactors and Functors, as well as their supervision relations. To provide a top-level function that takes only an `Actor` id, a `Film` id, as well as a rolename and adds the corresponding Actor information to the Film's cast list and vice versa, this example utilizes a concurrent Functor wrapping to sequential Functors. Both sequential Functors are comprised of two subsequent steps: The first sequential Functor queries information from the Film Dactor instance, which is needed to add the film to the Actor Dactor's filmography relation, i.e. the film's title. Then it sends a message to the Actor Dactor containing said data to trigger the insertion of the film. The second sequential Functor queries the Actor instance's data, which is necessary to add them to the Film's cast relation, i.e. their name. Using this data and their rolename the Functor then sends another message to the Film Dactor to trigger an insert into its cast relation. Both sequential Functors are executed in parallel using a concurrent Functor, which only sends a successful response to its caller after both sequential Functors have sent their responses to the concurrent Functor.

Using the parent-child relationship between calling actors and Functors enables a framework-level supervision of Functors, which are currently computing or waiting on results from down-stream Dactors. The supervision concept between parent and child actors in the Akka framework, termed parental supervision, enables passing back the control flow to the parent actor in case that a child actor, in this case a Functor, fails. The calling actor can then react to the failure by resuming or restarting the Functor, stopping this computation, or escalating the failure to its own parent. The correct course of action depends on the desired application behavior.

4.3 Framework Discussion

We presented an actor database framework and showcased some of it's functionality in detail. In particular, the described feature sets is aimed to alleviate some of the problems which arise from a distributed database system. In this section we discuss three core problems which are specific or pose additional challenges in a distributed setting:

Data partitioning in an actor database system differs fundamentally from common partitioning techniques used in relational databases. While large tables are typically partitioned based on a specific column's value or the hash thereof, our framework provides Dactors as entities for data encapsulation. Due to every Dactor being independent of other Dactor instances, only interacting with on another via messaging, Dactors can be provisioned accross multiple virtual runtimes or physical machines. As such, they provide flexible data partitioning based by application-domain concepts.

From the distributed setting of the database system arises the new problem of partition or **actor discovery**. The framework maintains a unified namespace, in which each Dactor instance is identified by its Dactor type and a unique id. In fact, querying a specific Dactor

just requires obtaining the messaging address from the nameservice and sending a message to it as shown in List. 2.

```
val ref = Dactor.dactorSelection(classOf[Film], filmId)
ref ! SelectAllFromRelation.Request("film_info")
```

List. 2: Lookup of and querying from a Film dactor instance.

Finally, **failure handling**, especially with regard to computations relying on multiple Dactors' data, requires careful monitoring due to Dactor distribution. Building on the Akka framework's parent-child supervision concept, our framework allows for transparent failure handling configurations. Failures can be handled within Dactors if appropriate. In case of multi-Dactor queries a fail-fast approach is chosen to allow calling actors or Dactors to react to exceptions in a timely manner.

5 Dactor Memory Overhead Experiments

Our concept is based on the usage of hundreds of thousands of Dactors, which all store only a small amount of data. This raises the question of how much memory overhead is introduced for storing data split across a multitude of actors. Therefore, we performed experiments comparing the memory usage of an exemplary actor database system with that of just loading the data into our data storage abstraction, called relations, or in a big string into memory.

5.1 Experimental Setup

The exemplary actor database system used for testing consists of four different Dactor types, each containing different number of relations and data sizes. We used a script to generate four different datasets emulating the scaling of the system by increasing the number of Dactor-instances in the system and keeping the data size stored in one Dactor nearly constant. The script creates various primitive data types, such as `String`, `Double`, and `Int`, as well as complex data types, such as `ZonedDateTime`, and distributes the data across Dactors and relations. The data distribution across the Dactor types is reported in Tab. 1a and the key figures of the datasets are reported in Tab. 1b. The data size stored in one Dactor and one relation is very small (in the kilobytes range), this will make the relative memory overhead more visible as we need thousands of Dactors and relations to store our datasets in the heap.

For each dataset we performed three different tests:

Single string Convert all data into its `String` representation and load it as a single big `String` into memory. This test serves as baseline for the other ones.

Dactor type	Data size	# Relations
X_1	7 KB	2
X_2	22 KB	3
X_3	171 KB	2
X_4	721 KB	1

(a) Dactor types and their corresponding data sizes

Dataset	Size on disk	# Dactors	# Relations
D_1	10 MB	829	1 714
D_2	25 MB	2 578	5 250
D_3	50 MB	4 373	8 935
D_4	100 MB	8 618	17 596

(b) Datasets sizes and key figures

Tab. 1: Datasets used for the memory overhead experiments

Relations Load the data into their respective `Relations`, preserving the type information and using the in-memory data storage objects from our framework.

Framework Use the full-fledged framework to load the data into memory. This approach stores the data distributed across Dactors in `Relation` objects.

To obtain the used memory of the objects in our test approaches, we used VisualVM[5] to create heap dumps. After the data was completely loaded into memory, we triggered a garbage collection run and created a heap dump. VisualVM is able to compute the retained sizes of object hierarchies in those dumps. This allowed us to investigate the memory usage of selected objects and their members in detail. We report the results of those experiments in the next section and discuss them in Sect. 5.3.

5.2 Dactor Memory Overhead

Fig. 4 shows an overview about the measured heap size of the different tests over the four datasets. If the data is loaded into memory as a single big `String` object, it takes up around the same amount of heap as the dataset is big. Storing the data in *Relations* introduces a overhead of about 150%. Even for the smallest dataset the data is split up across thousands of relations, which each use a two dimensional `Array` to store the individual values in a non-optimized way. In addition to that, relations also store metadata about the contained data, such as column names and data types.

As we can see in Fig. 4, using the full framework with Dactors does not introduce much additional memory overhead. Tab. 2 compares the sizes of the *Relations* and *Framework* tests and lists the Dactor overhead for the datasets D_1 through D_4. The memory overhead per Dactor is computed with the following equation:

$$\text{Overhead / Dactor} = \frac{\text{Size } \textit{Framework} - \text{Size } \textit{Relations}}{\text{\# Dactors}} \tag{1}$$

In average, using a Dactor only needs an additional 533 B more.

[5] `https://visualvm.github.io/`

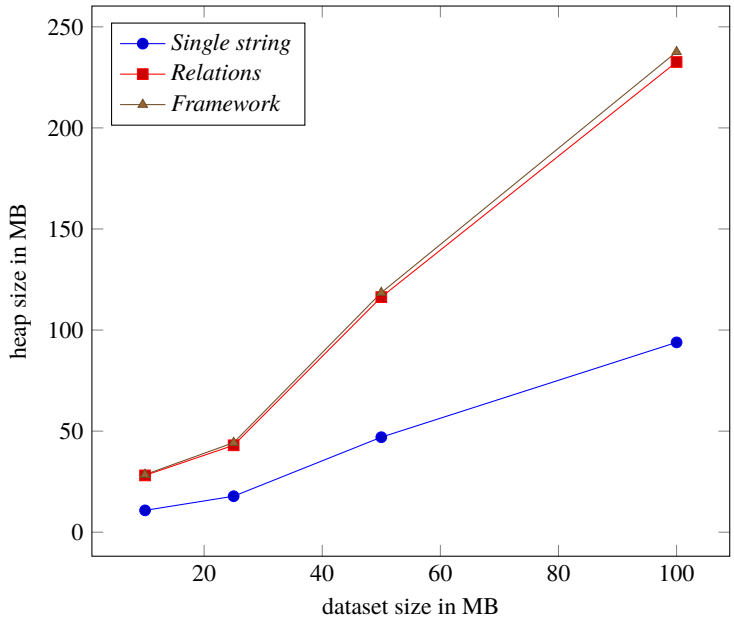


Fig. 4: Used heap size when loading the data into memory using the three different methods.

Dataset	**# Dactors**	**Size *Relations***	**Size *Framework***	**Overhead / Dactor**
D_1	829	28 788 KB	29 213 KB	526 B
D_2	2 578	44 034 KB	45 392 KB	539 B
D_3	4 373	119 084 KB	121 355 KB	532 B
D_4	8 618	238 169 KB	242 728 KB	534 B

Tab. 2: Memory overhead of Dactors

5.3 Discussion

Tab. 2 clearly shows that using Dactors only introduces a very small memory overhead. Dactors have a constant overhead of about 550 B per instance. Our experiments show a worst case scenario, where each Dactor only stores 230 KB on average.

Lets assume, we have a 1 TB database and we chose to store 1 MB per Dactor. This requires the actor database to instantiate about one million Dactors. Using the average overhead of 550 B per Dactor, this would yield a relative memory overhead of only 0.05 %. Doing the same thought experiment with storing 10 MB per Dactor, results in about 100 000 Dactors and reduces the relative memory overhead to 0.005%.

During execution of the experiments, we noticed that the load times for the *Framework* test were multiple times faster than the other testing scenarios. The data loading mechanism in the *Framework* test is highly parallel, because each Dactor is responsible for loading its data. They are triggered by a initial message telling them, where to find the data.

6 Conclusion

References

[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.

[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.