An Actor Database System for Akka

Sebastian Schmidl, Frederic Schneider, Thorsten Papenbrock[1]

Abstract: System architectures for data-centric applications are commonly comprised of two tiers: An application tier and a data tier. The fact that these tiers do not typically share a common format for data is referred to as *object-relational impedance mismatch*. To mitigate this, we implement a proof-of-concept actor database framework, which enables the definition of application logic as part of the data storage runtime. This approach is based on the concept of actor database systems introduced by Shah; Salles [SS17a], who propose using the actor programming model to build a distributed database runtime. Our framework's domain actors provide a typesafe, SQL-like interface to develop actor-internal functionality. Additionally, we present the Functor concept to enable queries spanning data contained in multiple actor instances. Our experiments demonstrate the feasibility of domain actors by evaluating their memory overhead and performance. We also discuss challenges that arise from the distributed database concept and how the actor model lends itself to implement solutions for data partitioning and failure handling for distributed, concurrent query processing.

Keywords: Actor Model; Database System; Akka; 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, system architectures for these types of software are comprised in two tiers: An application tier containing the bulk of the business logic, as well as a data tier, storing the application's data. The separation of concerns within this design 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 a complex, declarative query language. However, the monolithic design of conventional RDBMSs proves a hurdle for scalability and reduces modularity, which has a negative

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

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 (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. Even data storage solutions with a less rigid schema, such as document-oriented databases, 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 behavior 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. 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 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. With this framework, we make the following contributions: We introduce domain actors (Dactors) to model application data in an Akka-based actor database system. Similarly to the work of Shah; Salles on reactors [SS17b], Dactors encapsulate application data and logic. Since these actors are not part of a dedicated database runtime, but are defined using the application framework, data objects share the same representation throughout business logic and data storage. This approach bridges the aforementioned impedance mismatch between data and the application logic tier. In contrast to reactors, Dactors are not relational entities, but employ relational structures internally. Dactor state can be manipulated via an SQL-like interface. To define application logic relying on data contained within multiple Dactors, we provide the concept of Functors. Functors make the usage of asynchronous and concurrent computations explicit.

To our knowledge, we present the first implementation of this concept using the Akka framework. Comparable approaches are discussed in Sect. 2. In Sect. 3, we outline our concept for an actor database system in more detail, before explicitly describing the implementation of our prototype in Sect. 4. We present the results of our experimental evaluation using our framework in Sect. 5 and offer a concluding statement about this and future work in Sect. 6.

2 Related Work

In this section, we give an outline of recent actor model implementations and elaborate on the notion of *Actor Database Systems* and their requirements.

Various programming languages and libraries [Ak17; Ar07; Ha12; Li18a] implement the actor model that we introduced in the previous section. In 2004, Bernstein et al. [Be14] raised the abstraction level of actor objects even further by introducing the virtual actor programming model within the Orleans framework for .NET. 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. The framework Orbit [El18] provides virtual actors for the Java Virtual Machine (JVM).

The actor and virtual actor programming model has 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] specify needed features in their manifesto and outline four tenets that identify an *Actor Database System*:

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

Most practical research in the field of actor database systems has been presented in conjunction with the Orleans framework and the Erlang programming language [Be17; EB16; SS17b]. Biokoda [Bi18] takes another approach and encapsulates a full relational SQL database inside an actor. Cardin [Ca17] uses actors to build a scalable key-value store.

In contrast to the mentioned actor database systems, we developed our prototype using the Scala programming language and the Akka framework. 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. This allows close configuration of the in-memory state of the actor 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. This makes our prototype on Akka different from the related implementations in Orleans.

The features of our prototype relate to three of the four 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. Our concept of Functors allows for the definition of functionality using multiple actors' data. 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.

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

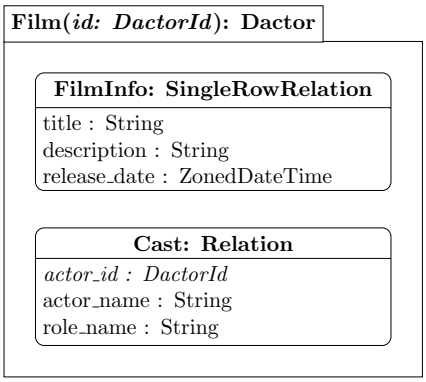
Similar to Shah; Salles [SS17b], we introduce a special type of actor, called Dactor, that acts as an application-defined scaling unit. Dactors can be used to model application-domain objects and encapsulate the object's state and application logic in an actor. Using actors for this enforces technical encapsulation of state access due to the purely private state in actors and the need of explicit asynchronous messaging between the actors. The encapsulation also 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. One Dactor can contain multiple relations. A relation is, similarly to a table in the relational database model, defined as a multiset of tuples following a predefined schema. 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. Relations form a typed, Dactor-internal data model. Using Dactors to implement a database leads to a modeling approach that is different to Entity-relationship modeling. We want to discuss the conceptual differences between the two with the following example.

We consider the example of a web application with information on movies similar to the imdb.com or rottentomatoes.com websites. A standard query for those websites is to display a film with its description and cast. A traditional data layout might be comprised of two entities: *Film* containing the film's ID, title, description and release date and *Actor* containing the actor's ID and name. Those two entities might be in a N-to-M relation (*Cast*) with an attribute showing the actor's role in the film. In contrast to the relational model, our model, shown in Fig. 1a, consists of one Dactor type and two relations and is denormalized. The information contained in the *Actor* entity is distributed across the `Cast` relations. This allows us to answer the standard queries from one single actor instance instead of joining the result together from different Dactors instances, which could be distributed across a network.

This approach to layout an application's data results in much smaller data sizes per Dactor compared to typical database tables and enables many business-logic-driven approaches to scaling, data partitioning, and caching. The trade-off to take into account, however, is a large number of Dactor instances and a denormalized schema.

As Dactors not only contain data, but also the corresponding domain logic, computation is executed concurrently. Actors provide single-threaded semantics, which makes enforcing constraints on data tied to one domain object stored inside a Dactor easy. While state querying and modification within Dactors is possible in a declarative way, the application developer can explicitly define the communication across all kinds of actors via asynchronous messages and how the Dactors handle the said messages. The explicit message definition and handling differentiates Dactors from Shah; Salles [SS17b]'s reactors, as reactors can be used as relational entities and hide the message passing from the developer.



(a) Graphical representation of the `Film` Dactor type definition.

```
class Film(id: Id) extends Dactor(id) {
  override protected val relations = {
    Film.Info -> SingleRowRelation(Film.Info),
    Film.Cast -> RowRelation(Film.Cast)
  }
  // define Dactor behavior
  override def receive: Receive = ???
}
object Film {
  object FilmInfo extends RelationDef {
    val title = ColumnDef[String]("title")
    val description = ColumnDef[String]("description")
    val release = ColumnDef[ZonedDateTime]("release")
  }
  object Cast extends RelationDef {
    val actorId = ColumnDef[Id]("actor_id")
    val name = ColumnDef[String]("actor_name")
    val rolename = ColumnDef[String]("role_name")
  }
}
```

(b) Example code using our framework.

Fig. 1: `Film` Dactor type definition with two relations of the example application.

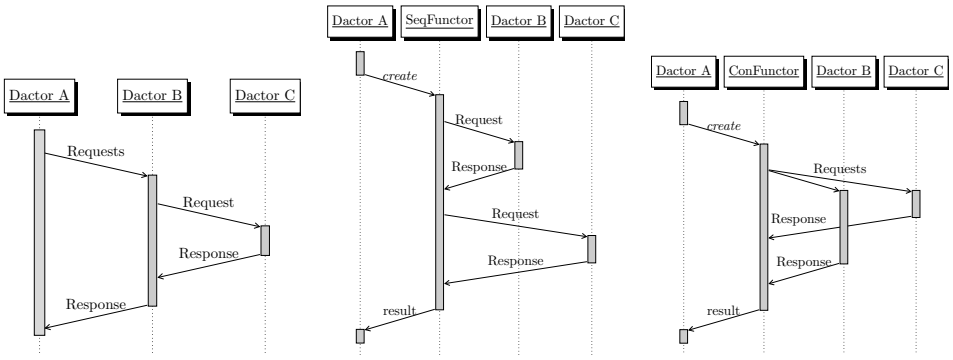
3.2 Communication between Domain Actors

Not all computation can be done with the information residing in a single Dactor. Hence, communication between Dactors is required. Explicit, asynchronous message passing 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. Those messaging primitives are provided in our framework and can be combined to create more complex message flows and computational models.

The first messaging pattern is **Cascading Computation**. A high-level message to an initial Dactor (Dactor A), will trigger successive messages to other Dactors, which are hidden from the original requester. Following Dactors can also trigger further messages themselves. 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. This clutters the domain logic in the Dactor and leads to complex and error-prone code. If used sparsely, this pattern supports separation of concerns and the tell-don't-ask paradigm.

Fig. 2b 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. This pattern can be implemented via Functors, which are Dactor-proxies provided by our framework. We introduce the concept of Functors in Sect. 4.2. Using a Functor to process the consecutive steps of the computational chain relieves Dactor A from dealing



(a) Cascading Computation (b) Sequential Computation (c) Concurrent Computation

Fig. 2: Inter-Dactor communication patterns. Gray bars indicate that an actor holds state that is related to the showed message flow.

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.

The **Concurrent Computation** 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 collects the results when they are finished to forward them to its creator. It allows for highly parallelized computations as all involved Dactors are messaged at the same time and calculate their responses concurrently.

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

4.1 Domain Actor Design

To illustrate the definition of `Dactors`, we use the example introduced in Sect. 3.1, which is schematically depicted in Fig. 1a. The corresponding definition of the `Film`'s data model is shown in Fig. 1b.

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 type's internal data layout, i.e. relations, and its functionality to process said data. In detail, a Dactor type's 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.

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 an operation that involves a multitude of Dactor instances. Implementation examples for this functionality and the trade-off are discussed in the following Section 4.2.

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. 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. While the basic functionality of the two Functor types for sequential and concurrent computation is discussed in Sect. 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 the concurrent and sequential computation patterns, introduced in Sect. 3.2.

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 two 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 insertion 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 its functionality in detail. In particular, the described feature set 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:

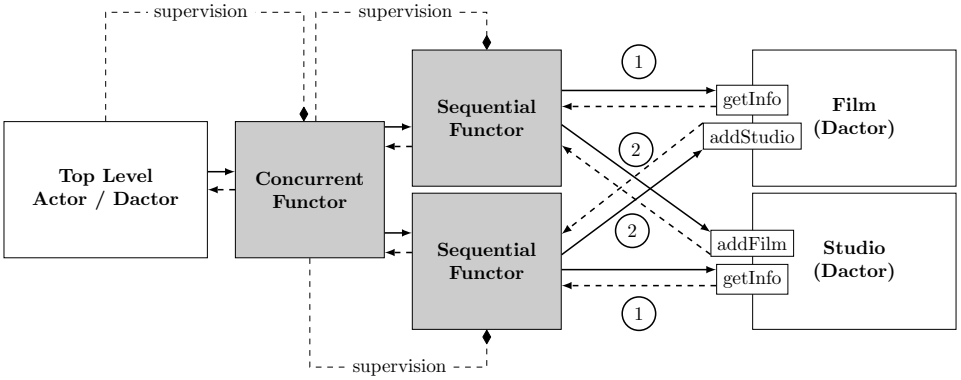


Fig. 3: Component diagram indicating the message flow through Functor objects and their supervision by the calling actor or Dactor. Arrow and dashed arrow pairs indicate corresponding request and response messages. The outgoing requests of each sequential functor are numbered to indicate their order.

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

List. 1: Lookup of and querying from a Film dactor instance.

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 and partitioning. Dactors can be provisioned across multiple virtual runtimes and physical machines, because every Dactor instance is independent of the others and the only mean of communication is message passing. As such, they provide flexible, fine-grained data partitioning based on application needs.

From the distributed setting of the database system the new problem of partition or **actor discovery** arises. 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. 1. In the case of a multi-node deployment, this is complemented by the functionality of Akka Cluster Sharding [Li18b], which routes the messages to the right physical host.

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

We measured the time a complex query consisting of multiple Functors takes to return the final result in our framework. It only took x ms, which is super fast. Our concept is based on the usage of hundreds of thousands of Dactors, which all store only a small amount of data. Therefore, the question of how much memory overhead is introduced for storing data split across a multitude of actors is more interesting than the computation time.

We performed experiments using an exemplary actor database system, which is modeled based on a real-world scenario. It consists of four different Dactor types, each containing one to three relations. The data stored in one Dactor ranges from seven to about 700 kilobytes depending on its type. This small data size makes the relative memory overhead more visible as we need thousands of Dactors and relations to store our datasets in the heap. We used a script to generate four different datasets emulating the scaling of the system by increasing the number of Dactor-instances 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 dataset sizes are reported in Tab. 1.

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 tests.

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[2] 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 Tab. 1.

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

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

Dataset	Disk size	# Dactors	Heap size *Single string*	Heap size *Relations*	Heap size *Framework*	Overhead / Dactor
D_1	10 MB	829	11 MB	28 MB	29 MB	526 B
D_2	25 MB	2 578	18 MB	43 MB	44 MB	539 B
D_3	50 MB	4 373	47 MB	116 MB	119 MB	532 B
D_4	100 MB	8 618	101 MB	233 MB	237 MB	534 B

Tab. 1: Used heap size of our three different methods to load data into memory and the memory overhead of Dactors compared across the four datasets.

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 Tab. 1, using the full framework with Dactors does not introduce much additional memory overhead. On average, using a Dactor only needs an additional 533 B more.

Let us assume that 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%.

6 Conclusion

This work presents the proof-of-concept implementation of an actor database framework, which enables developers to declaratively define a data model using Dactors. Dactors model application-domain objects and encapsulate the object's sate and application logic in an actor. The framework provides a shared, distributed runtime for database functionality and application logic, mitigating the *object-relational impedance mismatch* between data and business logic tier.

We discuss challenges specific to a database system in a distributed runtime and how the proposed actor database framework solves these challenges. In particular, Dactors constitute independent units, which allow for flexible partitioning of application data based on application domain logic and concepts. The use of the Akka framework enables efficient implementation of partition discovery and failure handling in this distributed setup. The introduced Functor concept, i.e. temporary actors managing computations that involve data originating from multiple Dactor instances, provide a transparent computation model and failure handling capabilities.

Finally, we present an experimental evaluation of the memory overhead introduced by using actors for data management. The results show that, due to the constant and low memory footprint of actors, the actor database system approach is feasible and lends

itself especially when presented with large amounts of data and highly concurrent data manipulation workload.

The research implementation of the framework provides a platform for the integration of traditional database functionalities. Future work will comprise the implementation of inter-Dactor consistency guarantees by extending Functors with rollback and, e.g., a *two-phase commit* protocol implementation. In addition, further testing of this research implementation's query execution latency and throughput performance remains a topic of interest.

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. Pp. 6-1–6-26, 2007.

[Be14] Bernstein, P. A.; Bykov, S.; Geller, A.; Kliot, G.; Thelin, J.: Orleans: Distributed Virtual Actors for Programmability and Scalability, tech. rep., Microsoft Research, 2014.

[Be17] Bernstein, P. A.; Dashti, M.; Kiefer, T.; Maier, D.: Indexing in an Actor-Oriented Database. In: Proceedings of the Conference on Innovative Data Systems Research (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.: Transactions for Distributed Actors in the Cloud, tech. rep., Microsoft Research, 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. Pp. 1–6, 2012.

[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.

[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 Actor Database Systems. In: Proceedings of the International Conference on Management of Data (COMAD). Pp. 259–274, 2017.

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