An Actor Database System for Akka

Frederic Schneider, Sebastian Schmidl[1]

Abstract: Abstract goes here

Keywords: Actor Database System; Akka; Scala

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 proves a hurdle for scalability and reduces modularity, which has a negative impact on code

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

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. 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

Our concept is based on the idea of actor database systems as introduced by Shah; Salles [SS17a]. Database systems should be a distributed runtime and provide the appropriate programming abstraction. This is solved by actor database systems. 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 Domain Actor (Dactor for short), 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.

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. The system grows naturally based on the load. Take an e-commerce application as an example for that: If we have a customer entity for every customer in our system, then we can model each entity as a Dactor. This allows our system to scale linearly with the number of customers using it, because each customer is represented with their own Dactor, which holds their information and performs computations. This also illustrates that such a system tends to be more robust compared to monolithic systems, because if customer A's Dactor crashes for an unknown reason, all other customer Dactors are unaffected.

Actors provide single-threaded semantics, which makes enforcing constraints on data tied to one domain object stored inside Dactors no issue. It is implemented via application logic within the Dactor. State querying and modification within Dactors is possible in a declarative way, but 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 figure 1: Cascading Computation, Sequential Computation, Concurrent Computation. They all 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 figure 1a, this pattern is comparable to function calls in OOP, but can lead to a growing cascade of messages sent and received. As each message can introduce network latency, it can take significantly more time to receive the top-level response compared to the situation when all computation is done in Dactor A. In addition to that, the requesting Dactor has to deal with the state for pending responses, such as actor references, intermediate results, or failure cases. 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 idea.
2. **Sequential Computation** Figure 1c 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 or whatelse?.
   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. One drawback of this design is the need to create an extra actor for every top-level request that should be processed.

3. **Concurrent Computation** This messaging pattern also makes use of Functors to encapsulate the processing of multiple request-response pairs. This time, we want to send messages to several Dactors concurrently and collect the results when they are finished. An exemplary message flow is shown in figure 1b. As this messaging pattern also relies on Functors, it has similar advantages and drawbacks than 2. It allows for highly parallelized computation as all subtasks are started asynchronously at the beginning and the result is sent back as soon as all responses were received.

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 perform computations on a higher level.

- Consistency?
- Transactional guarantees?

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

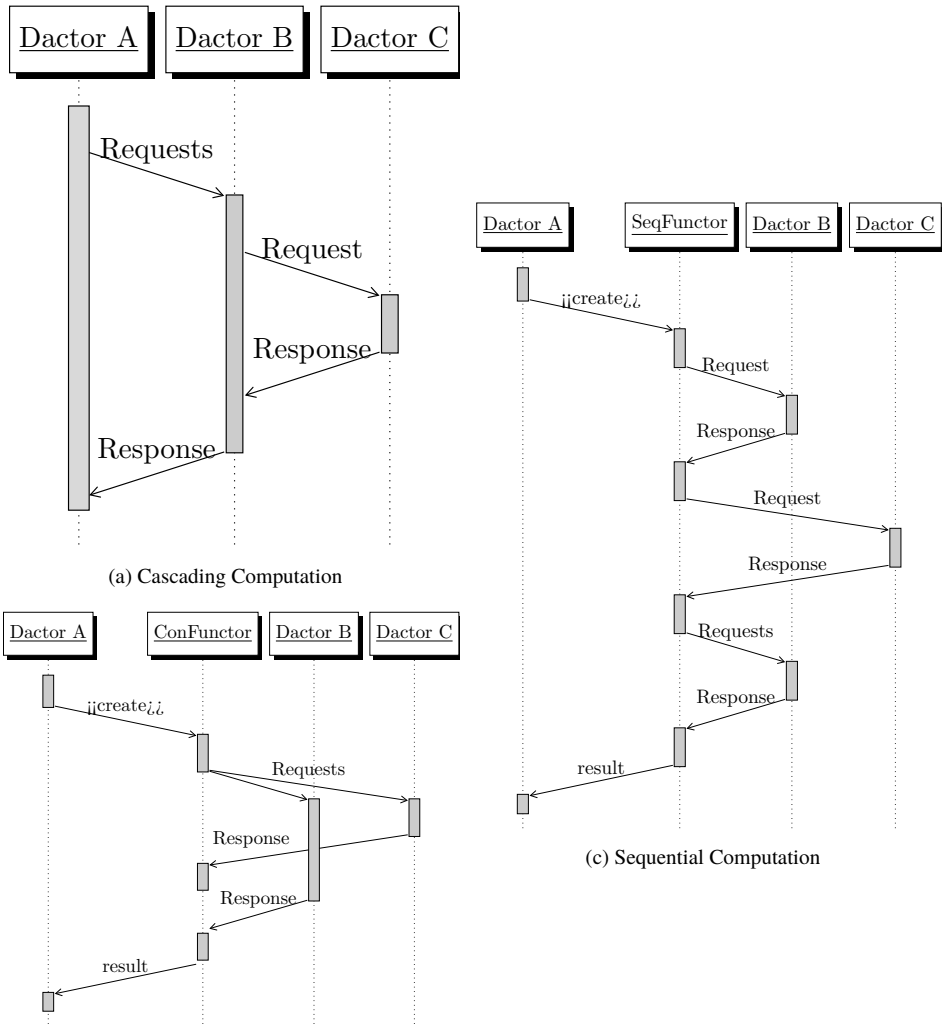

Fig. 1: Inter-Dactor-communication patterns. Dotted lines show actor lifetime and gray bars indicate that an actor performs computation or holds state that is related to the showed message flow.

4 Domain Actor Database Framework

4.1 Actor Design

4.2 Memory Storage

4.3 Multi-Actor Queries

4.4 Discussion

5 Actor 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 table 1a and the key figures of the datasets are reported in table 1b.

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

For each dataset we performed three different tests:

SingleString 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.

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 and discuss the results of those experiments in the next section.

5.2 Results

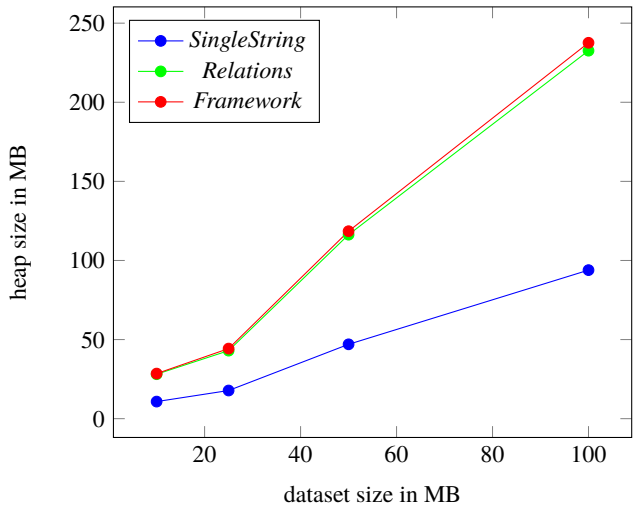

Fig. 2: Used heap size when loading the data into memory using the three different methods.

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

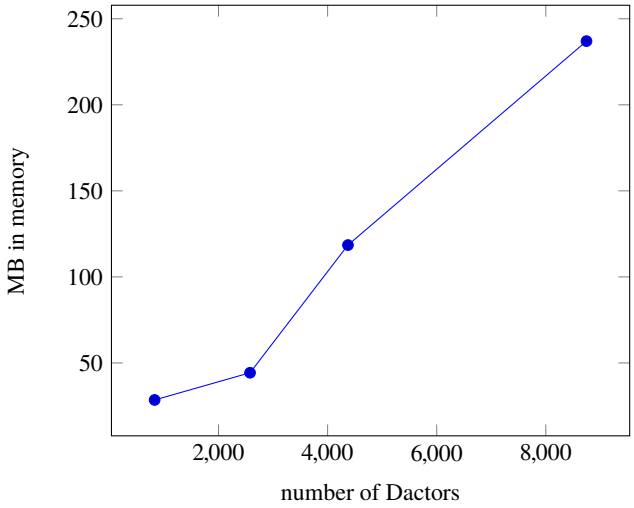

Fig. 3: Memory consumption as a function of the number of Dactors

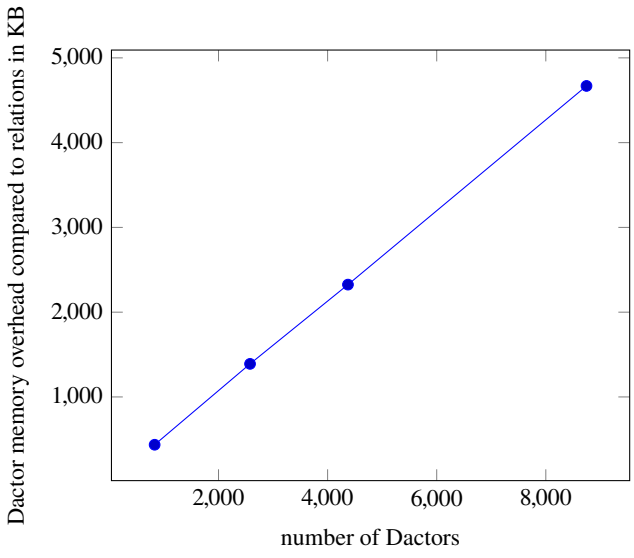

Fig. 4: Memory overhead of Dactors as a function of the number of Dactors

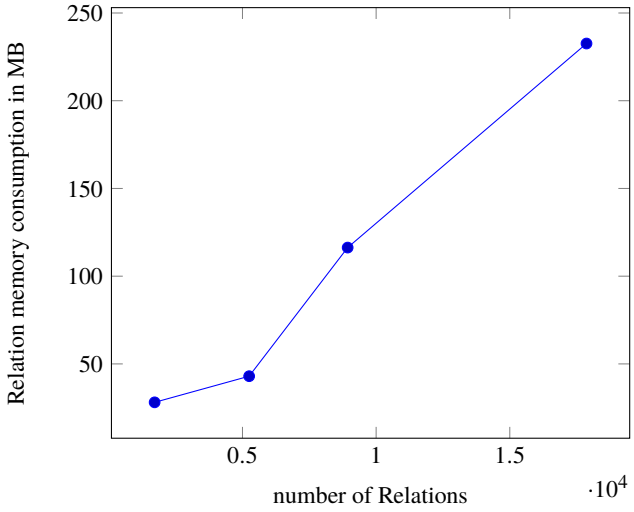

Fig. 5: Memory overhead of Relations as a function of the number of Relations

5.3 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. 6 zeigt eine Abbildung.

Hier sollte die Graphik mittels `includegraphics` eingebunden werden.

Fig. 6: Demographik

Tab. 2 zeigt eine Tabelle.

Die LNI-Formatvorlage verlangt die Einrückung von Listings vom linken Rand. List. 1 zeigt uns ein Beispiel, das mit Hilfe der `lstlisting`-Umgebung realisiert ist. Referenz auf `print("Hello World!")` in Line 6.

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. 2: Die Überschriftsarten

```
/**
  * 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. 1: 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.