

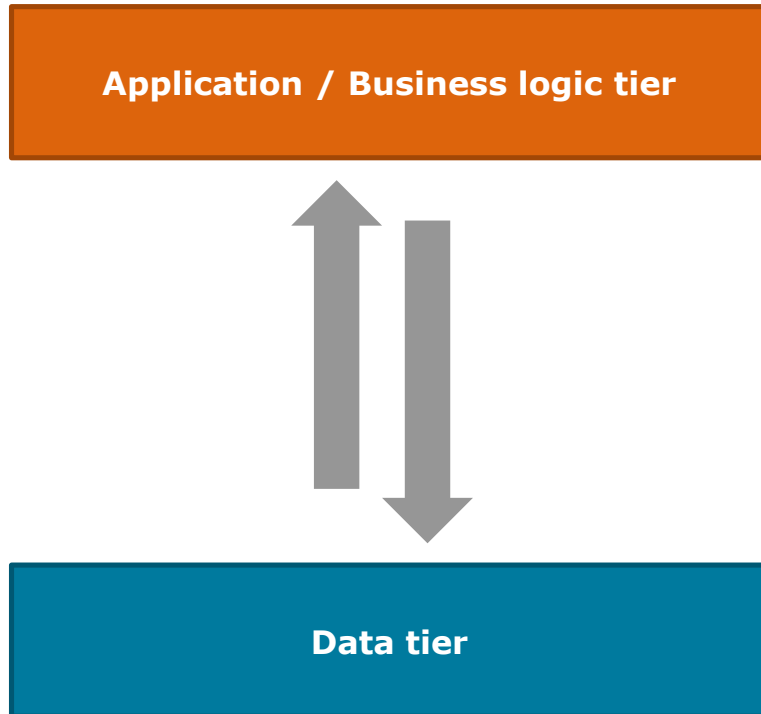
# An Actor Database System for Akka

Frederic Schneider, Sebastian Schmidl

{frederic.schneider, sebastian.schmidl}@student.hpi.de

# Motivation

## Object-Relational Impedance Mismatch



- Two-tiered system layout for data-centric applications

### **An Actor Database System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 2

# Motivation

## Object-Relational Impedance Mismatch

Application / Business logic tier



Data tier

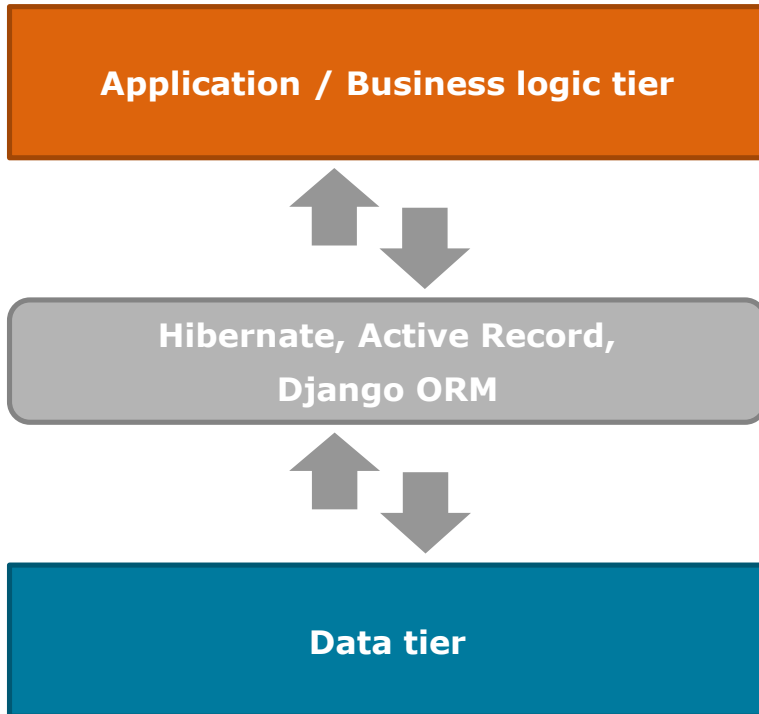
- Two-tiered system layout for data-centric applications
- Object-relational impedance mismatch of data representation in the respective tiers

### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 3

# Motivation

## Object-Relational Impedance Mismatch



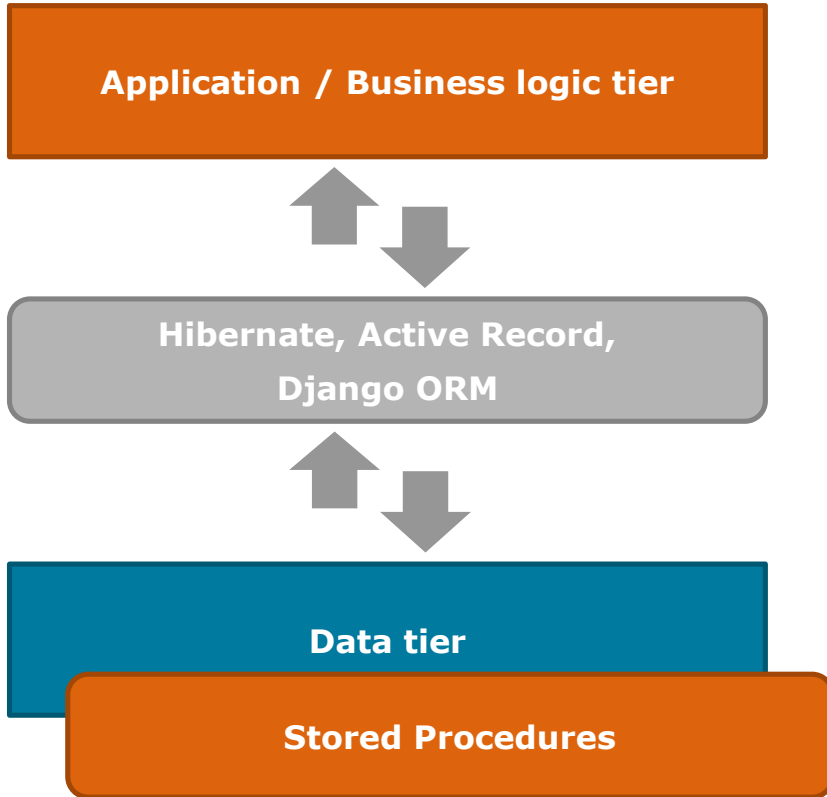
- Two-tiered system layout for data-centric applications
- Object-relational impedance mismatch of data representation in the respective tiers
- ORM tools are an attempt to solve this challenge...

### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 4

# Motivation

## Object-Relational Impedance Mismatch



- Two-tiered system layout for data-centric applications
- Object-relational impedance mismatch of data representation in the respective tiers
- ORM tools are an attempt to solve this challenge...
- ...but they don't avoid co-locating logic of expensive computations in the data tier

### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 5

**Actor =**



+



**An Actor Database  
System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart **6**

**Dactor =**



+




→ **Dissipates object-relational impedance mismatch**

**An Actor Database  
System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 7

# Example Use-Case

Part of the Collection: *Predator* [View Collection](#)



### THE PREDATOR

**Critics Consensus**  
The *Predator* has violence and quips to spare, but its chaotically hollow action adds up to another missed opportunity for a franchise increasingly defined by disappointment.

**32%** TOMATOMETER  
Reviews Counted: 262

**36%** AUDIENCE SCORE  
User Ratings: 6.172

[More Info](#)

## MOVIE INFO

From the outer reaches of space to the small-town streets of suburbia, the hunt comes home in Shane Black's explosive reinvention of the *Predator* series. Now, the universe's most lethal hunters are stronger, smarter and deadlier than ever before, having genetically upgraded themselves with DNA from other species. When a young boy accidentally triggers their return to Earth, only a ragtag crew of ex-soldiers and a disgruntled science teacher can prevent the end of the human race.

**Rating:** R (for strong bloody violence, language throughout, and crude sexual references)

**Genre:** Action & Adventure, Horror, Science Fiction & Fantasy

**Directed By:** Shane Black

**Written By:** Shane Black, Fred Dekker

**In Theaters:** Sep 14, 2018 Wide

**On Disc/Streaming:** Dec 18, 2018

**Studio:** 20th Century Fox

## CAST



Boyd Holbrook  
as Quinn McKenna



Trevante Rhodes  
as Williams



Jacob Tremblay  
as Rory McKenna



Keegan-Michael Key

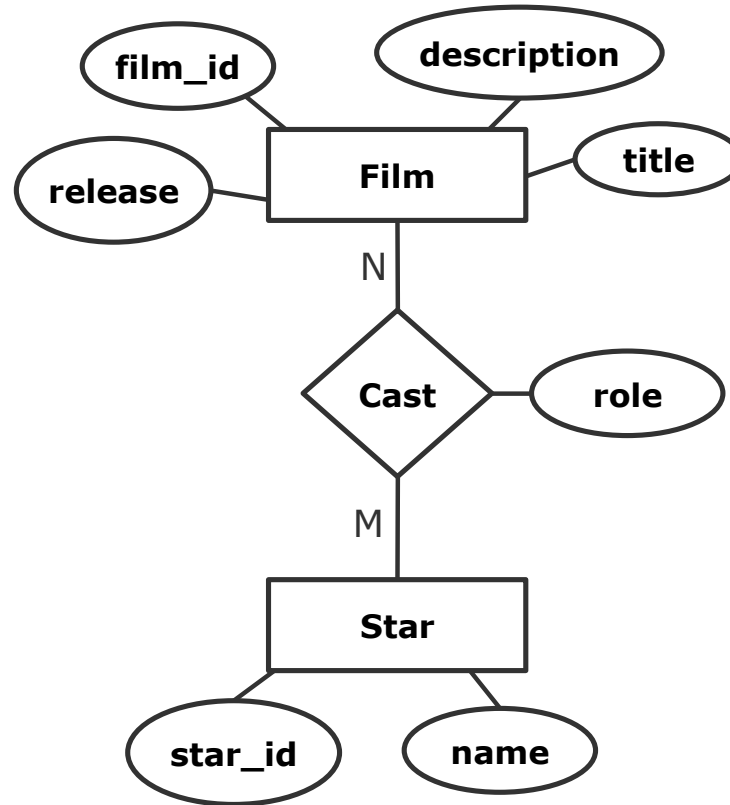


Olivia Munn  
as Casey Bracket



Sterling K. Brown  
as Government Agent

[View All](#)

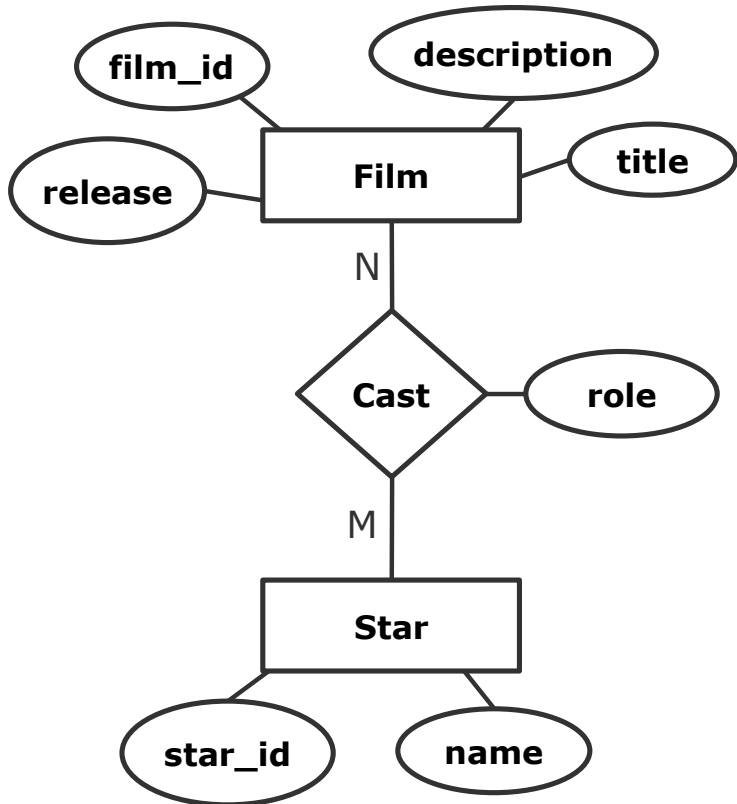


## An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 8



# Example Use-Case



## Film (film\_id)

FilmInfo		
title	description	release

Cast		
star_id	name	role

prepareDisplayInfo()

**An Actor Database System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 9

# Implementation in our Framework

## Film (film\_id)

### FilmInfo

title	description	release
-------	-------------	---------

### Cast

star_id	name	role
---------	------	------

**prepareDisplayInfo()**

```
object FilmInfo extends RelationDef("film_info") {
  val title: ColumnDef[String] = ColumnDef("title")
  val description: ColumnDef[String] = ColumnDef("description")
  val release: ColumnDef[ZonedDateTime] =
    ColumnDef("release", ZonedDateTime.EPOCH)
}

object Cast extends RelationDef("cast") {
  val actorId: ColumnDef[Int] = ColumnDef("actor_id")
  val name: ColumnDef[String] = ColumnDef("name")
  val role: ColumnDef[String] = ColumnDef("role")
}

class Film(id: Int) extends Dactor(id) {
  override protected val relations: Map[RelationDef, MutableRelation] =
    Dactor.createAsRowRelations(Seq(FilmInfo, Cast))

  override def receive: Receive = {
    case PrepareDisplayInfo.Request() => prepareDisplayInfo()
  }
}
```

# Implementation in our Framework

## Film (film\_id)

### FilmInfo

title	description	release

### Cast

star_id	name	role

prepareDisplayInfo()

```

val results: Seq[Record] =
  relation(Cast)
    .where(Cast.role ~> {
      _ equals "Casey Bracket"
    })
    .project(Set(Cast.name))
    .records
  
```

```

results.foreach(println)
// ["Olivia Munn"]
  
```

```

object FilmInfo extends RelationDef("film_info") {
  val title: ColumnDef[String] = ColumnDef("title")
  val description: ColumnDef[String] = ColumnDef("description")
  
```

```

    date: ColumnDef[Date] = ColumnDef("release_date")
    dateEpoch: ColumnDef[Int] = ColumnDef("release_date_epoch")
  
```

```

  ("cast") {
    actor: ColumnDef[Int] = ColumnDef("actor_id")
    name: ColumnDef[String] = ColumnDef("name")
    role: ColumnDef[String] = ColumnDef("role")
  }
  
```

```

  actor(id) {
    relations: Map[RelationDef, MutableRelation] =
      Map(FilmInfo, Cast)
  }
  
```

```

  override def receive: Receive = {
    case PrepareDisplayInfo.Request() => prepareDisplayInfo()
  }
  
```

```

}
  
```

# Distributed Actor Database System

- **Dactors** are **strongly encapsulated**
- **Dactors** can be distributed across **multiple runtimes**  
**and / or physical nodes**



**Dactors are an application-defined scaling unit**

**An Actor Database  
System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart **12**

# Distributed Actor Database System Partitioning

## Film

film_id	title	descr.	release
1	The Predator	...	14.09.2018
2	Mandy	...	14.09.2018
3	The Nun	...	14.09.2018

## Cast

film_id	cast_id	role
1	13	Quinn McKenna
1	75	Casey Bracket
...	...	...

## Star

star_id	name
13	Boyd Holbrook
75	Olivia Munn
...	...

## Film (3)

## Film (2)

## Film (1)

### FilmInfo

title	Descr.	Release
The Predator	...	14.09.2018

### Cast

star_id	name	Role
13	Boyd Holbrook	Quinn McKenna
75	Olivia Munn	Casey Bracket

prepareDisplayInfo()

# Evaluation

## Memory Overhead Experiments

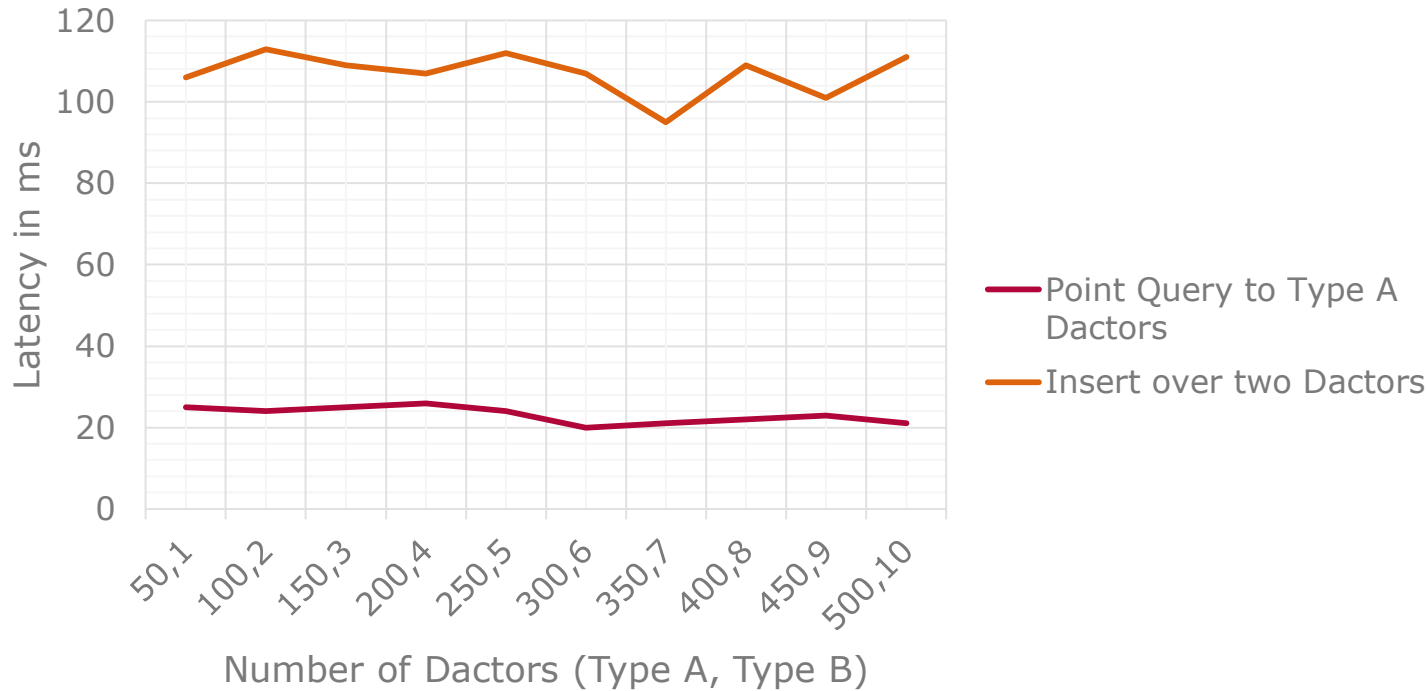


### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 14

# Evaluation

## Query Performance



### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 15

# Conclusion

---

- ✓ **Shared runtime** for storage and application logic
- ✓ Large Number of **Dactors** for data storage feasible

- ✗ **Consistency** and **transactions** missing
- ✗ Redundant storage due to **denormalization**
- ✗ Evaluation of multi-node deployment

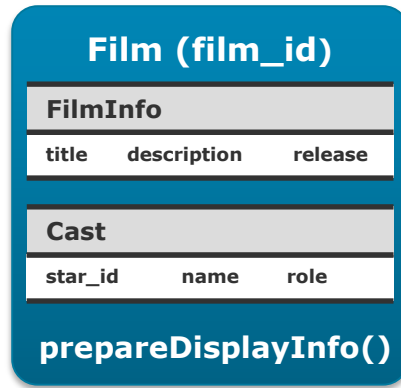
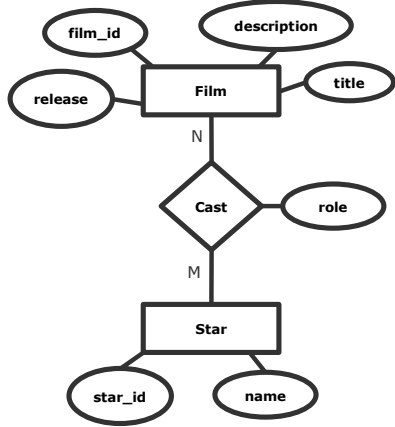
We also investigated...

- **Failure Handling**
- **Multi-Dactor Queries**

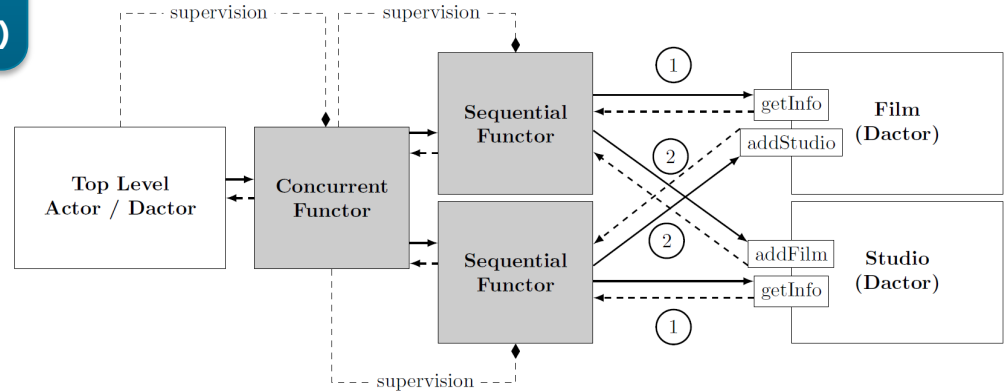
**An Actor Database System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart **16**






~ 500 Byte memory overhead per Dactor



Thank you!  
 Questions?

Frederic Schneider  
 Sebastian Schmidl

{frederic.schneider, sebastian.schmidl}@student.hpi.de

- Dactor Model eases failure handling 
  - What about **distributed system, messaging over the network, multiple processes and threads?**
- Single-threaded semantics inside Actors
- Explicit messaging: we expect failures
- **Akka: Actor Supervision**
  - Each Actor has parent, gets notified on any failure, can handle it
  - No need to think about each failure beforehand



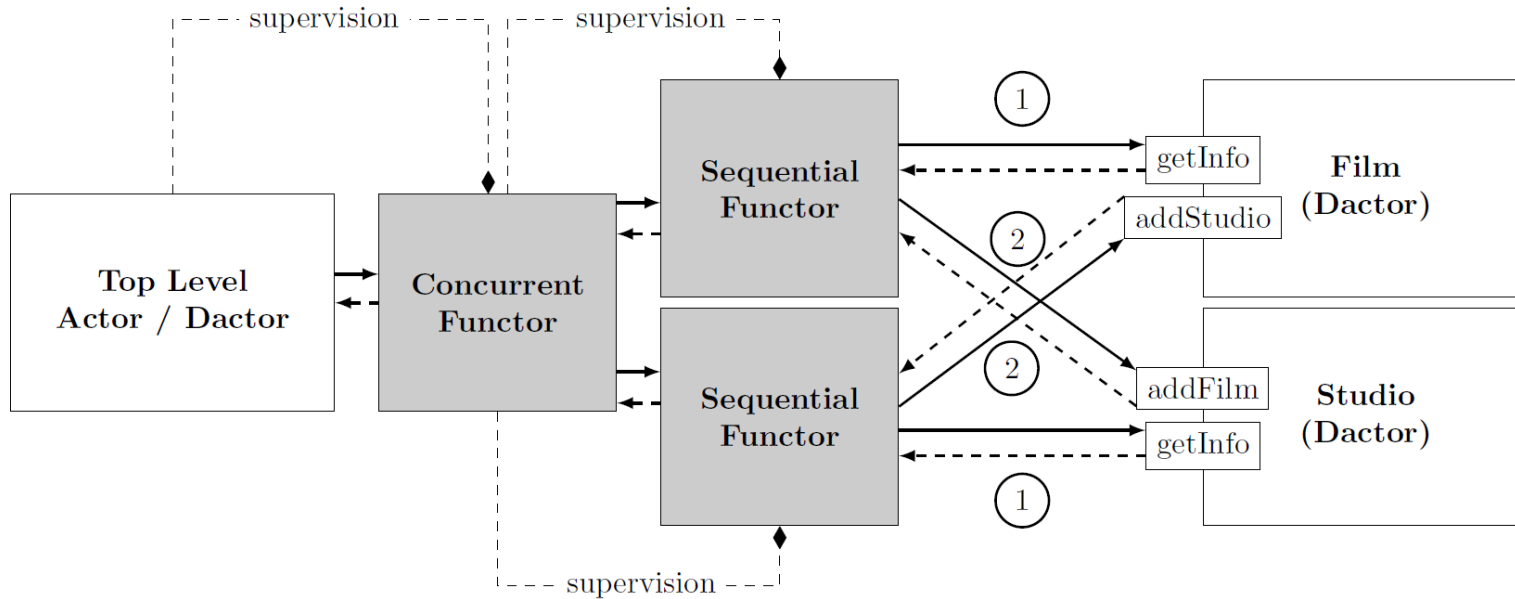
**Use Actor supervision for failure handling of queries**

**An Actor Database  
System for Akka**

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart **18**

# Appendix

## Multi-Dactor Queries → Functors



### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 19

# Appendix

## Declaration of Sequential Functors

```
val filmId = 1
val actorId= 13
val role = "Quinn McKenna"

val addCastToFilm = SequentialFunction()
  .start( (_: AddCastToFilm.Start) =>
    GetActorInfo.Request()
  ), Dactor.dactorSelection(classOf[Actor], actorId)

  .next(response => {
    response.records.headOption match {
      case Some(actorInfo: Record) =>
        val name= actorInfo(ActorInfo.name)
        AddCast.Request(actorId, name, role)
      case None =>
        fail(ActorInfoNotFoundException())
    }
  }, Dactor.dactorSelection(classOf[Film], filmId)

//.next(response => new request, receiver)

.end(identity) // sends last response or failure message to caller
```

```
val functorRef = Dactor.startFunctor(
  // functor, akka-context, reference to Actor receiving response
  addCastToFilm, context, self
)(
  // start-message
  AddCastToFilm.Start
)
```

### An Actor Database System for Akka

Frederic Schneider,  
Sebastian Schmidl,  
2019-03-05  
Chart 20