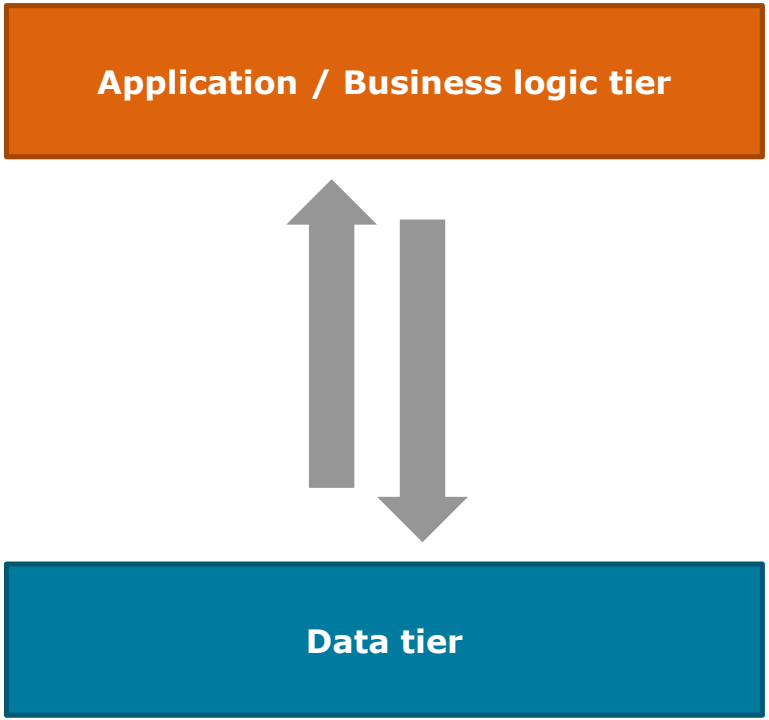
Hasso Plattner Institut
Digital Engineering · Universität Potsdam
18. BTW 2019 Uni Rostock 600 Jahre

An Actor Database System for Akka

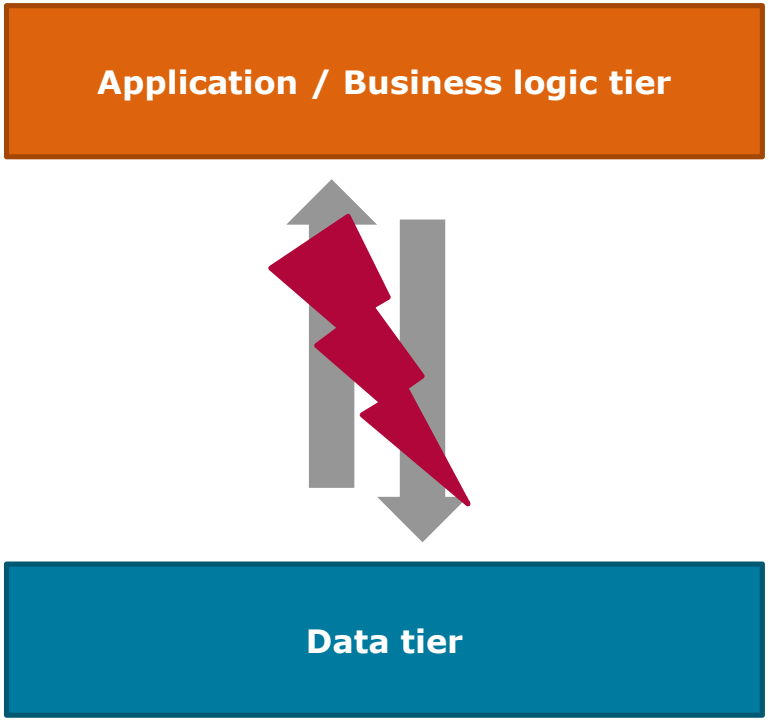
Frederic Schneider, Sebastian Schmidl
{frederic.schneider, sebastian.schmidl}@student.hpi.de

Motivation
Object-Relational Impedance Mismatch



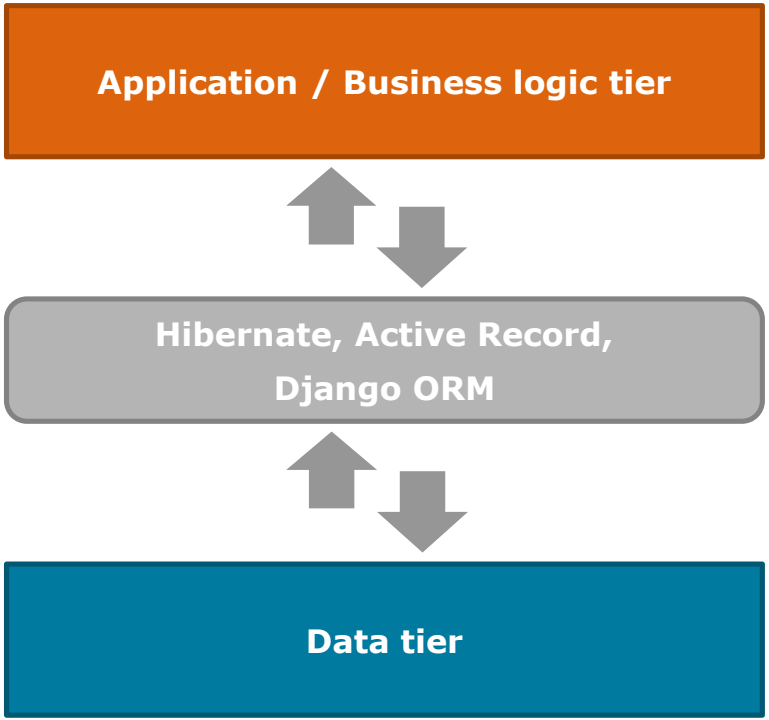
- Two-tiered system layout for data-centric applications

Motivation
Object-Relational Impedance Mismatch



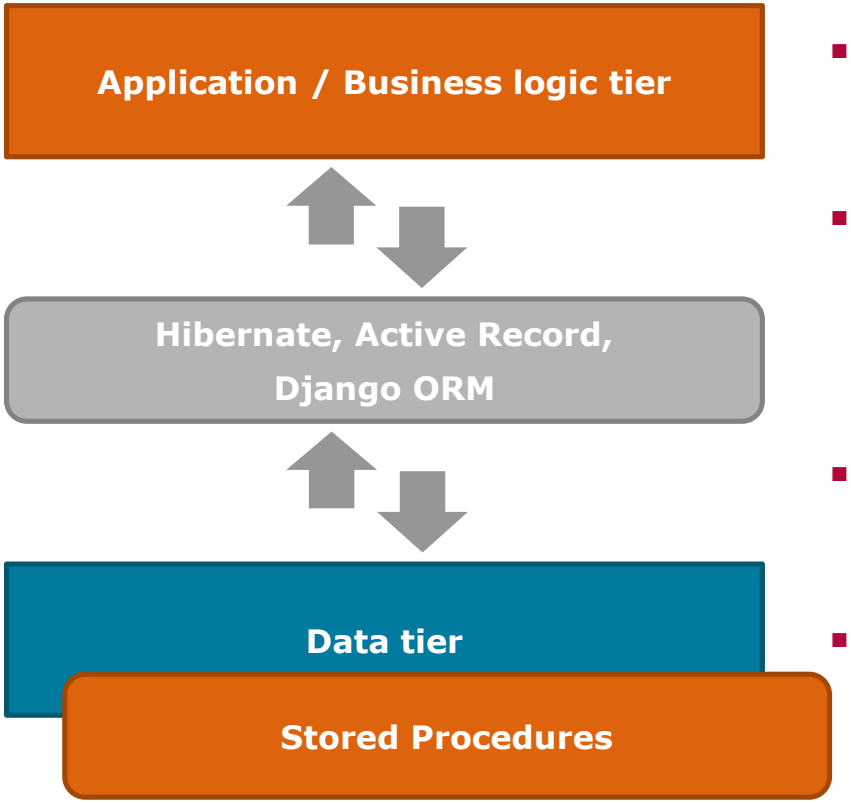
- Two-tiered system layout for data-centric applications
- Object-relational impedance mismatch of data representation in the respective tiers

Motivation
Object-Relational Impedance Mismatch



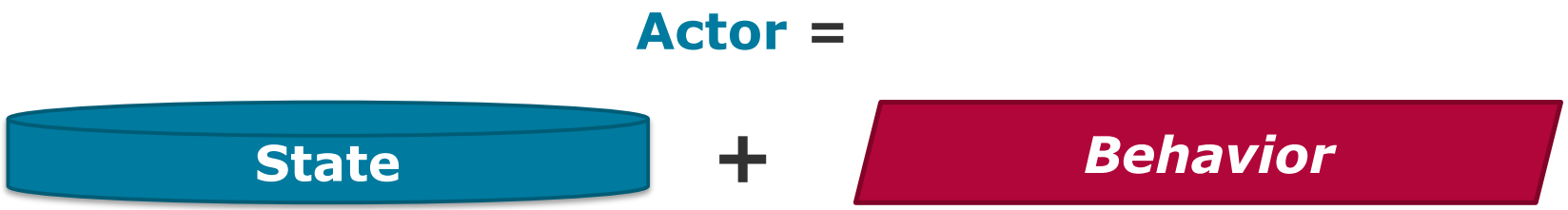
- Two-tiered system layout for data-centric applications
- Object-relational impedance mismatch of data representation in the respective tiers
- ORM tools are an attempt to solve this challenge…

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

Actor Programming

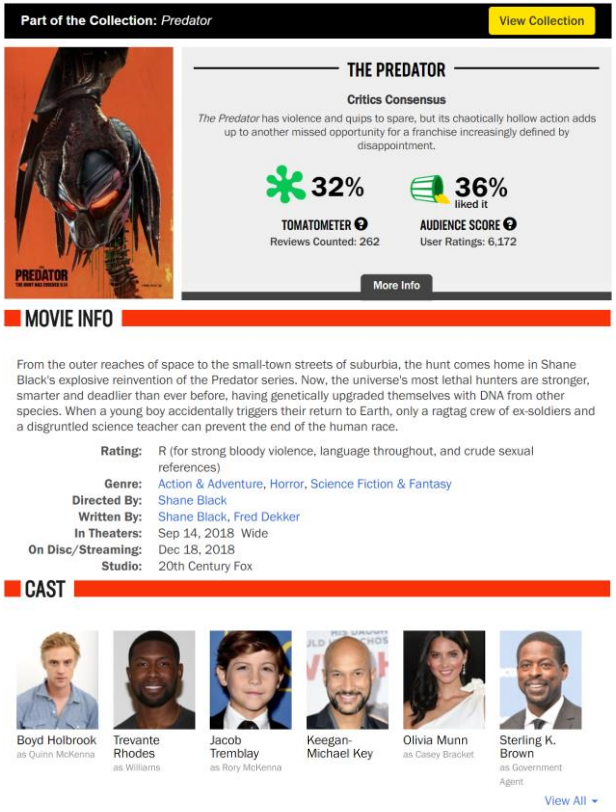


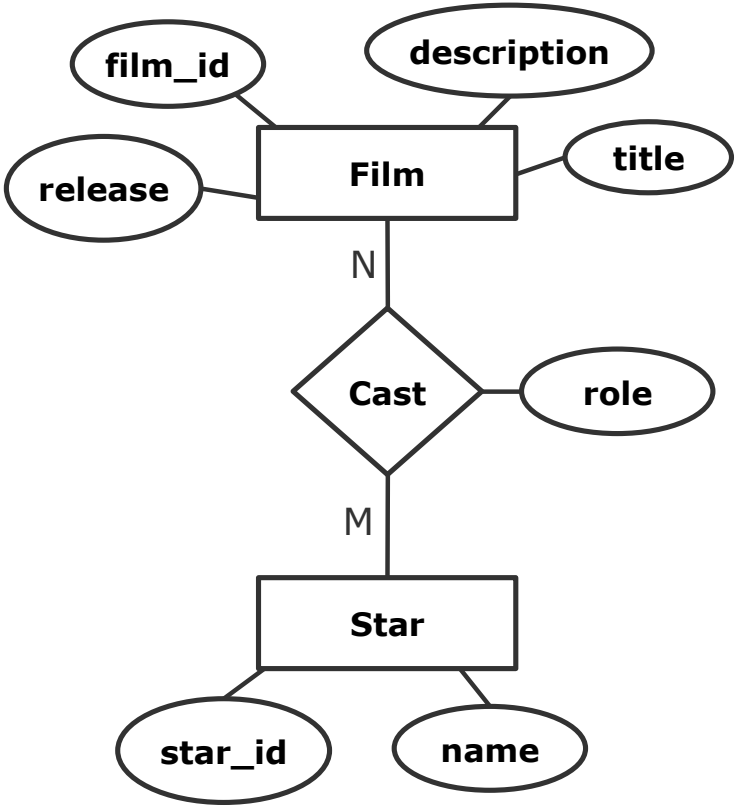
Domain Actors

Dactor =

→ **Dissipates object-relational impedance mismatch**

Example Use-Case

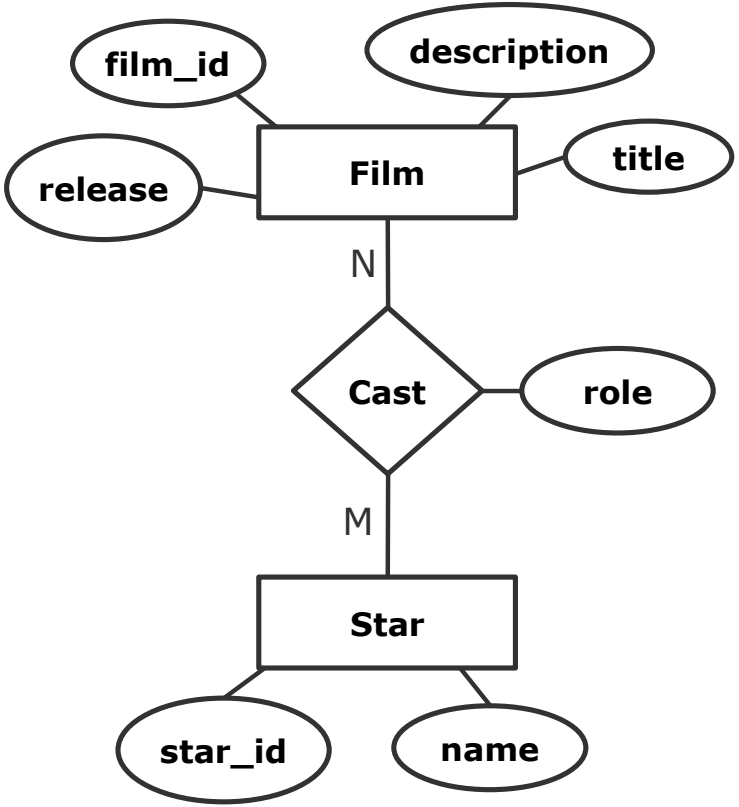


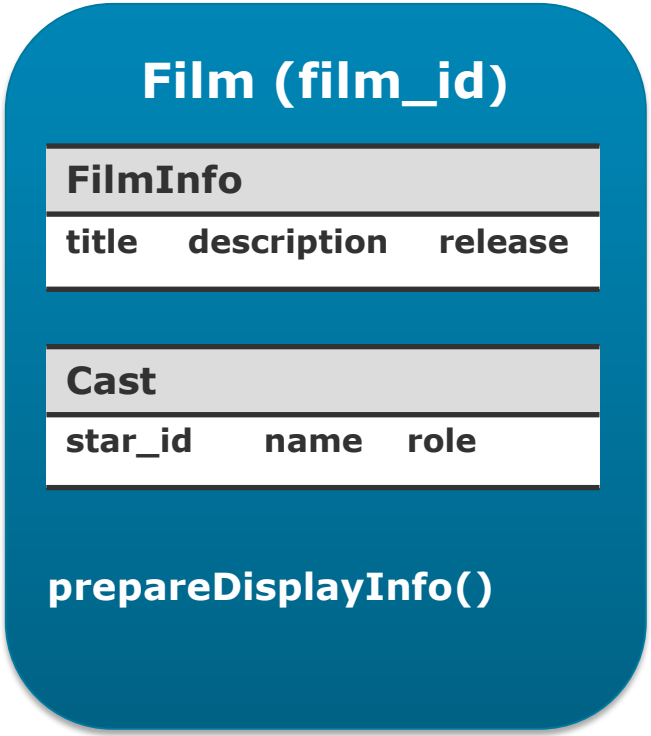


An Actor Database System for Akka

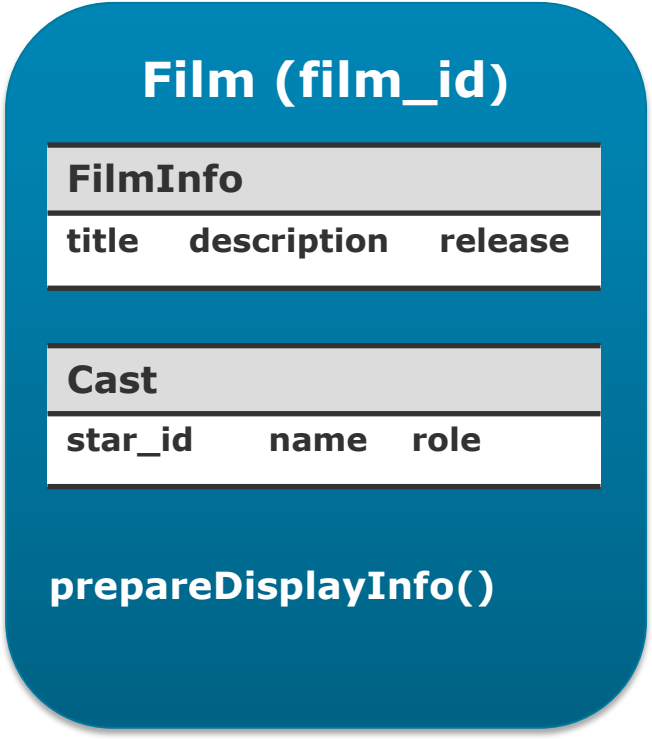
Frederic Schneider, Sebastian Schmidl, 2019-03-05
Chart **8**

Example Use-Case





Implementation in our Framework



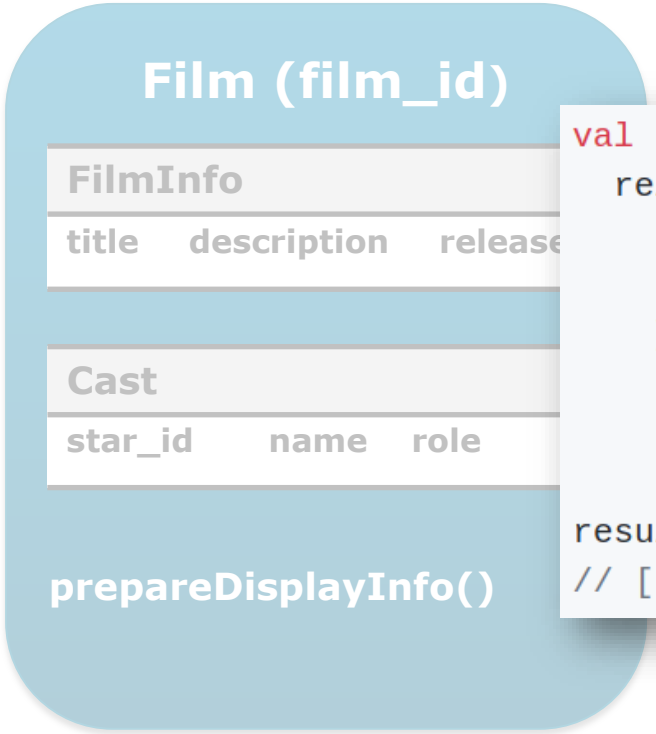

```
object FilmInfo extends RelationDef("film_info") {
  val title: ColumnDef[String] = ColumnDef("title")
  val description: ColumnDef[String] = ColumnDef("description")
  val release: ColumnDef[ZonedDateTime] =
    ColumnDef("release", ZonedDateTime.EPOCH)
}

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




```
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")
  ateTime] =
  ateTime.EPOCH)

("cast") {
 ColumnDef("actor_id")
 ColumnDef("name")
 ColumnDef("role")

tor(id) {
ons: Map[RelationDef, MutableRelation] =
(Seq(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

Distributed Actor Database System Partitioning

Film

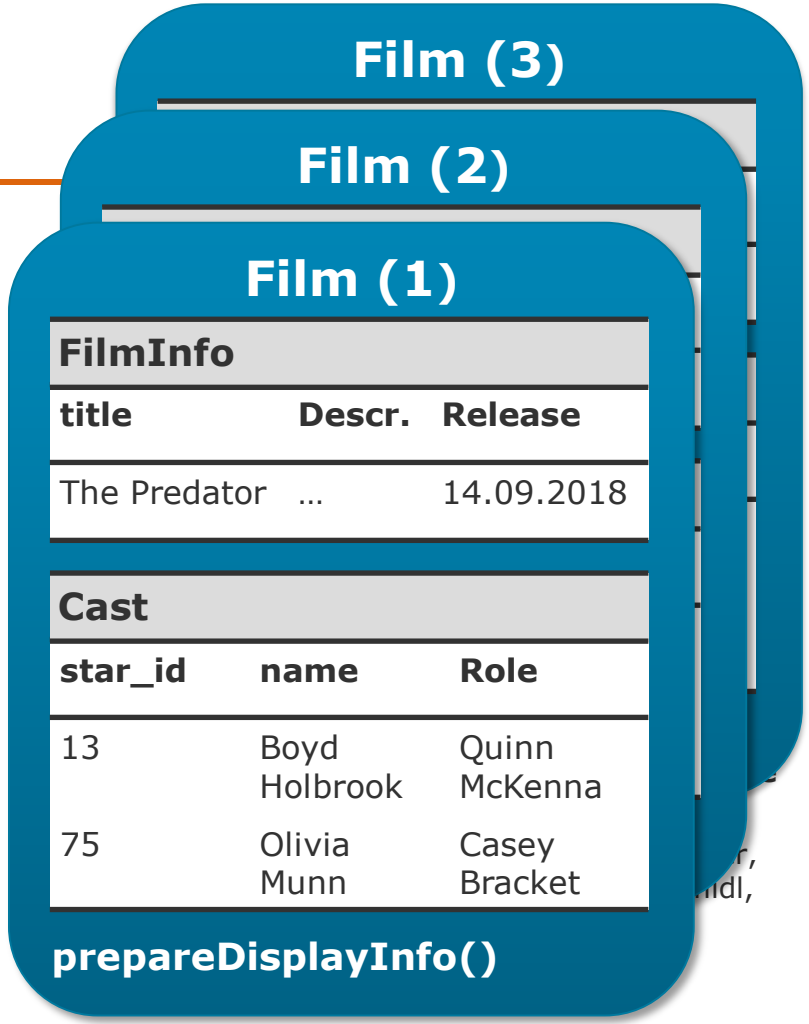
film_id	title	descr.	release
1	The Predator	…	14.09.2018
2	Mandy	…	14.09.2018
3	The Nun	…	14.09.2018

Cast

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

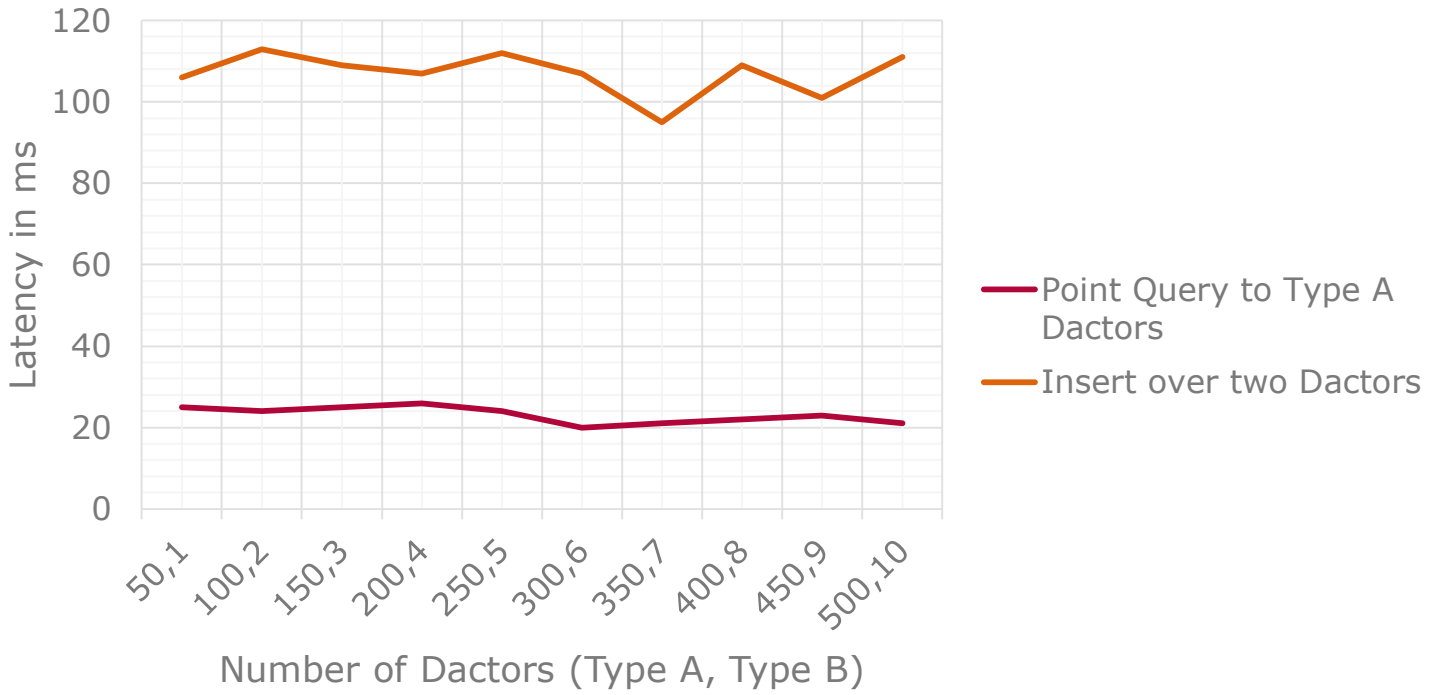
Star

star_id	name
13	Boyd Holbrook
75	Olivia Munn
…	…



Evaluation
Memory Overhead Experiments

Evaluation
Query Performance



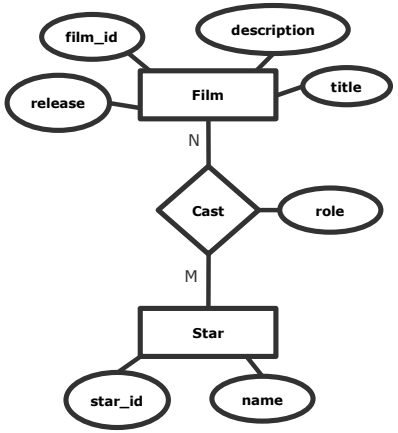
Conclusion

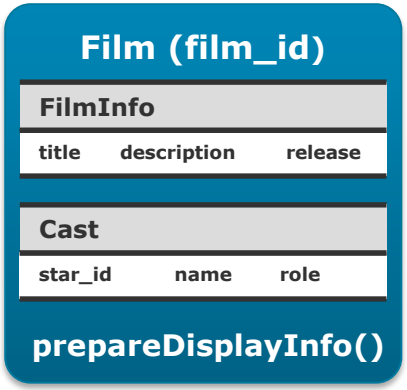
- ✓ **Shared runtime** for storage and application logic
- ✓ Large Number of Dactors for data storage feasible

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

We also investigated…

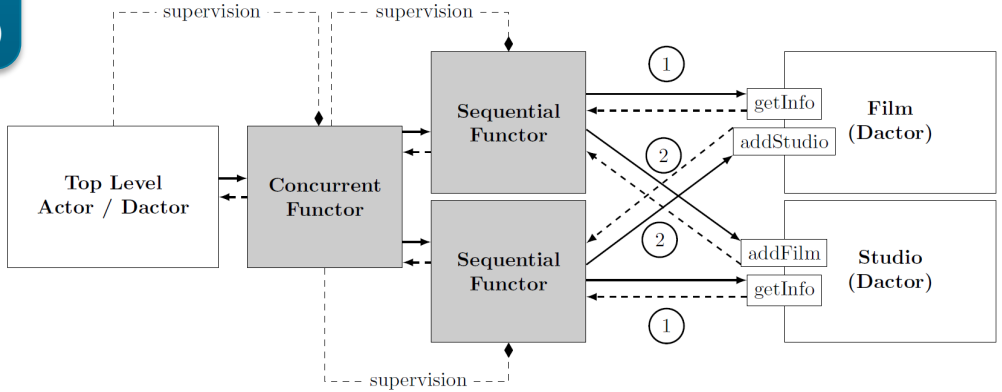
- **Failure Handling**
- **Multi-Dactor Queries**





Hasso Plattner Institut
Digital Engineering · Universität Potsdam
18. BTW 2019 Uni Rostock 600 Jahre

~ 500 Byte memory overhead per Dactor



Thank you!
Questions?

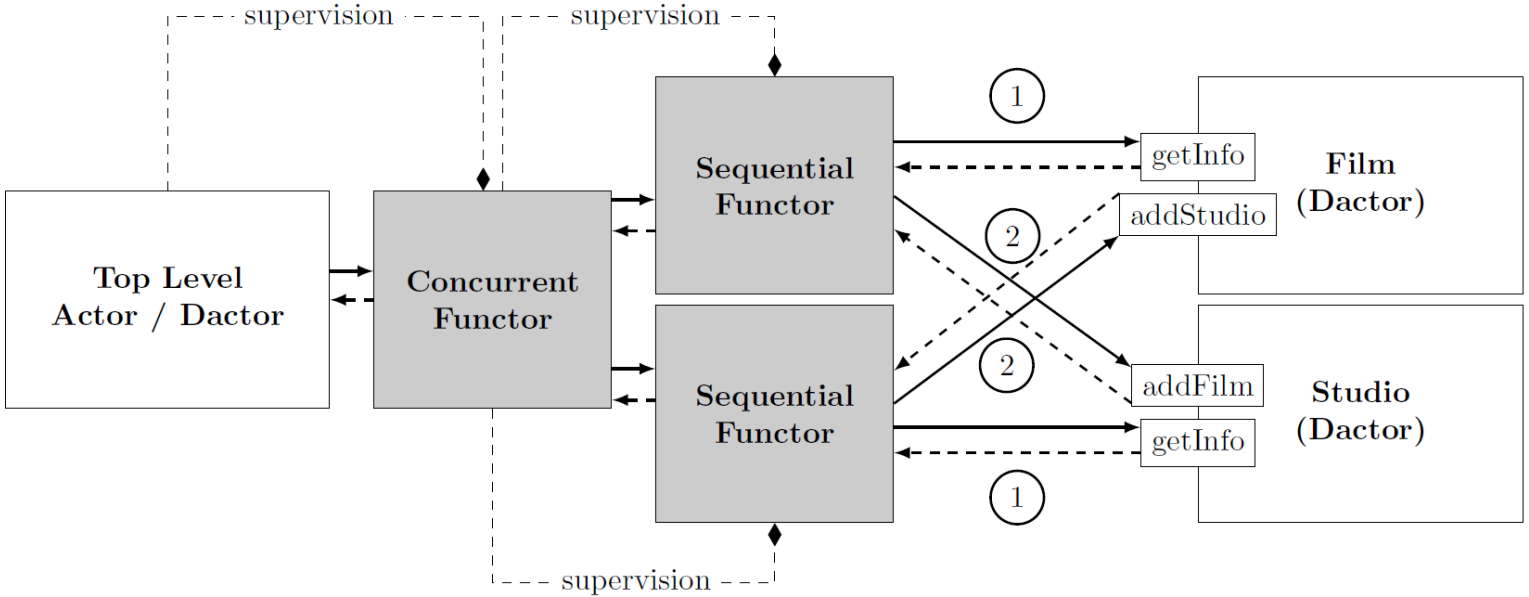
Frederic Schneider
Sebastian Schmidl
{frederic.schneider, sebastian.schmidl}@student.hpi.de

Appendix
Failure Handling

- 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**

Appendix
Multi-Dactor Queries → Functors



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