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

Application / Business logic tier

Data tier

An Actor Database System for Akka
Frederic Schneider, Sebastian Schmidl, 2019-03-05
Chart 2
Motivation
Object-Relational Impedance Mismatch

- 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

Application / Business logic tier

| Hibernate, Active Record, Django ORM |

Data tier

| Stored Procedures |

An Actor Database System for Akka
Frederic Schneider, Sebastian Schmidl, 2019-03-05
Chart 5
Actor Programming

An Actor Database System for Akka
Frederic Schneider, Sebastian Schmidl, 2019-03-05
Chart 6

Actor = State + Behavior
Domain Actors

\[ \text{Dactor} = \text{Data Storage} + \text{Business Logic} \]

\( \rightarrow \text{Dissipates object-relational impedance mismatch} \)
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
Example Use-Case

An Actor Database System for Akka
Frederic Schneider, Sebastian Schmidl, 2019-03-05
Chart 9
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 Doctor(id) {
  override protected val relations: Map[RelationDef, MutableRelation] =
    Doctor.createAsRowRelations(Seq(FilmInfo, Cast))

  override def receive: Receive = {
    case PrepareDisplayInfo.Request() => prepareDisplayInfo()
  }
}
Implementation in our Framework

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

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

results.foreach(printLn)  
// ["Olivia Munn"]

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**

<table>
<thead>
<tr>
<th>film_id</th>
<th>title</th>
<th>descr.</th>
<th>release</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Predator</td>
<td>...</td>
<td>14.09.2018</td>
</tr>
<tr>
<td>2</td>
<td>Mandy</td>
<td>...</td>
<td>14.09.2018</td>
</tr>
<tr>
<td>3</td>
<td>The Nun</td>
<td>...</td>
<td>14.09.2018</td>
</tr>
</tbody>
</table>

**Cast**

<table>
<thead>
<tr>
<th>film_id</th>
<th>cast_id</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>Quinn McKenna</td>
</tr>
<tr>
<td>1</td>
<td>75</td>
<td>Casey Bracket</td>
</tr>
</tbody>
</table>

**Star**

<table>
<thead>
<tr>
<th>star_id</th>
<th>name</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Boyd Holbrook</td>
<td>Quinn McKenna</td>
</tr>
<tr>
<td>75</td>
<td>Olivia Munn</td>
<td>Casey Bracket</td>
</tr>
</tbody>
</table>

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 Actors** 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**
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
An Actor Database System for Akka

Frederic Schneider, Sebastian Schmidl, 2019-03-05
Chart 19
Appendix
Declaration of Sequential Functors

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

val addCastToFilm = SequentialFunction()
  .start(_: AddCastToFilm.Start) =>
    GetActorInfo.Request()
  }, Doctor.actorSelection(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())
    }
  }, Doctor.actorSelection(classOf[Film], filmId)

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

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

val functorRef = Doctor.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