Leveraging Scala and Akka to build NSDb,

a distributed time-series database

Saverio Veltri @save_veltri
Paolo Mascetti @mascettipaolo

Firenze
14th September

Who we are

Saverio Veltri
Solution Architect

Paolo Mascetti
Data Engineer

RADICALBIT

- **Based in Milan since 2015**
- Event Stream Processing products and solutions

We are a specialized software firm, born in Milan on 2015

RADICALBIT

- Based in Milan since 2015
- Event Stream Processing products and solutions

We are focussed on the design and development of **Event Stream Processing products and solutions**, combining **streaming technologies** with **Machine Learning and A.I.**

Agenda

Introduction

NSDb Main Features

Single Node Design

Akka Cluster Overview

Distributed Design

Roadmap & Licensing

Contribution

Introduction

Motivations

Connotations

Time Series Model

Consistency Model

NSDb in Data Intensive Architectures

- NSDb in CQRS Pattern

Motivations

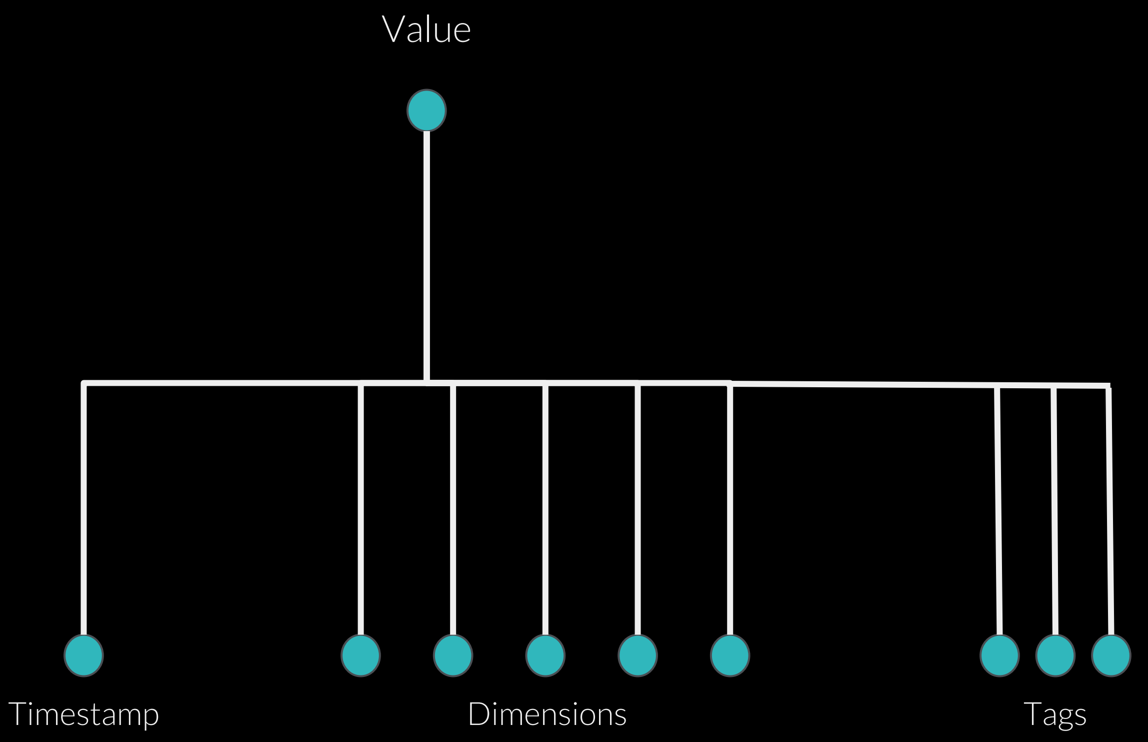
- Have a deep technical ownership of the solution
- Too many licensing and pricing issues exploring third-party OEM solutions
- Third-party solutions don’t completely fit our requirements

Connotations

- Distributed
  - Allows cluster deploy of p2p nodes
  - Based on Akka Cluster
- TimeSeries
  - Optimized time series management
- Streaming oriented
  - Maintain real-time capability in streaming architectures

Time Series Model (I)

Bit: a MultiDimensional Time Series value



Timestamp: the record time

Value: the numerical value being measured

Dimensions: a dynamic list of queryable String -> Value pairs

Tags: special dimensions user can apply aggregations on

Time Series Model (II)

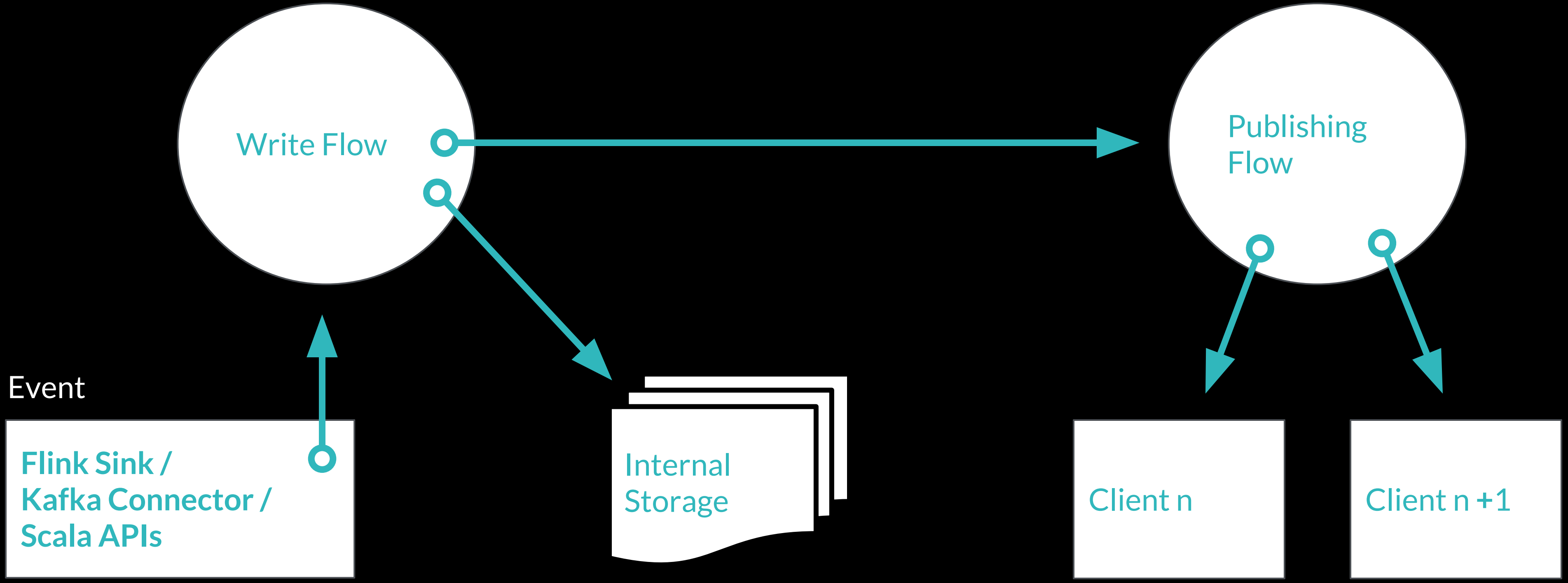
- NSDB’s Bits are **immutable**. New data continuously arrives, and will be always inserted and never updated.
- Bit schema is monotonic

Bit organization:

- Metric: a series of Bit (Records)
- Namespace: high level structure grouping metrics
- Database: logical container grouping namespaces

NSDb - Consistency Model

- Eventual consistency
- Real time delivery for subscribed client

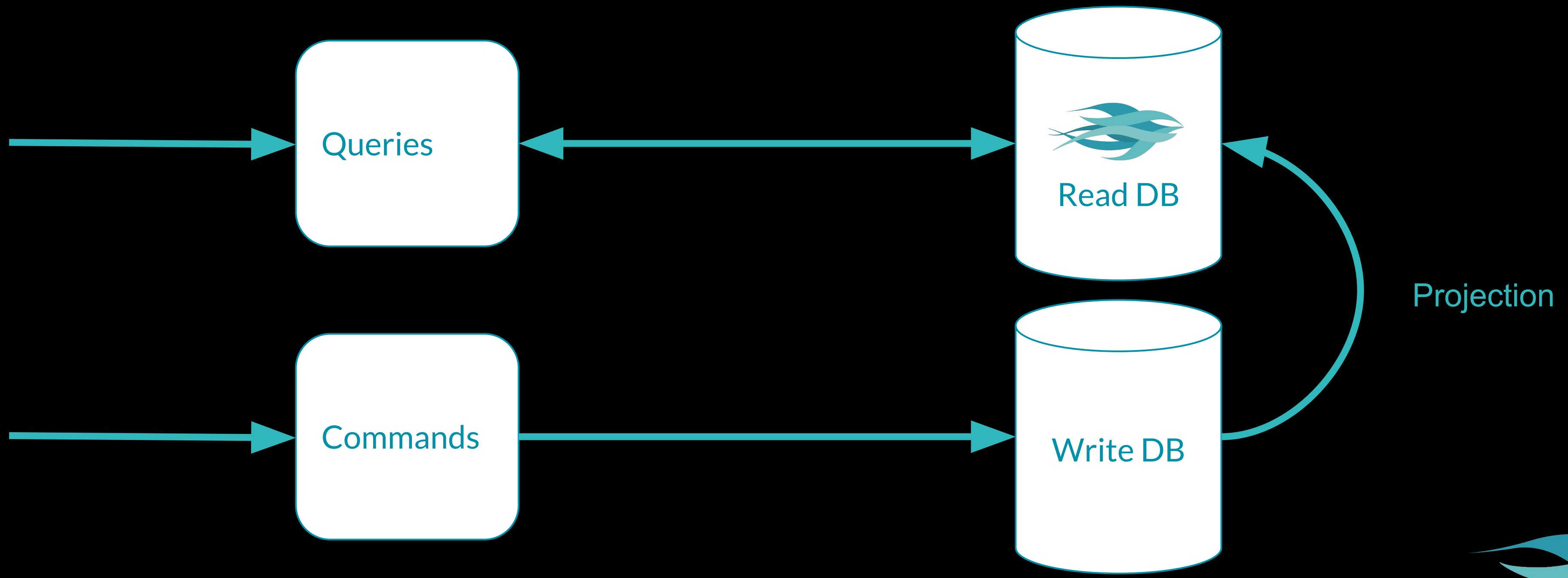


NSDb in data intensive architectures

- Eventual Consistency narrows down the points of applicability of NSDb
- Real time streaming and Push features perfectly fit the serving layer (e.g. Kappa architecture and CQRS)

NSDb in CQRS Pattern

- Clear separation of Commands and Queries
- Scalability guaranteed by using 2 different databases



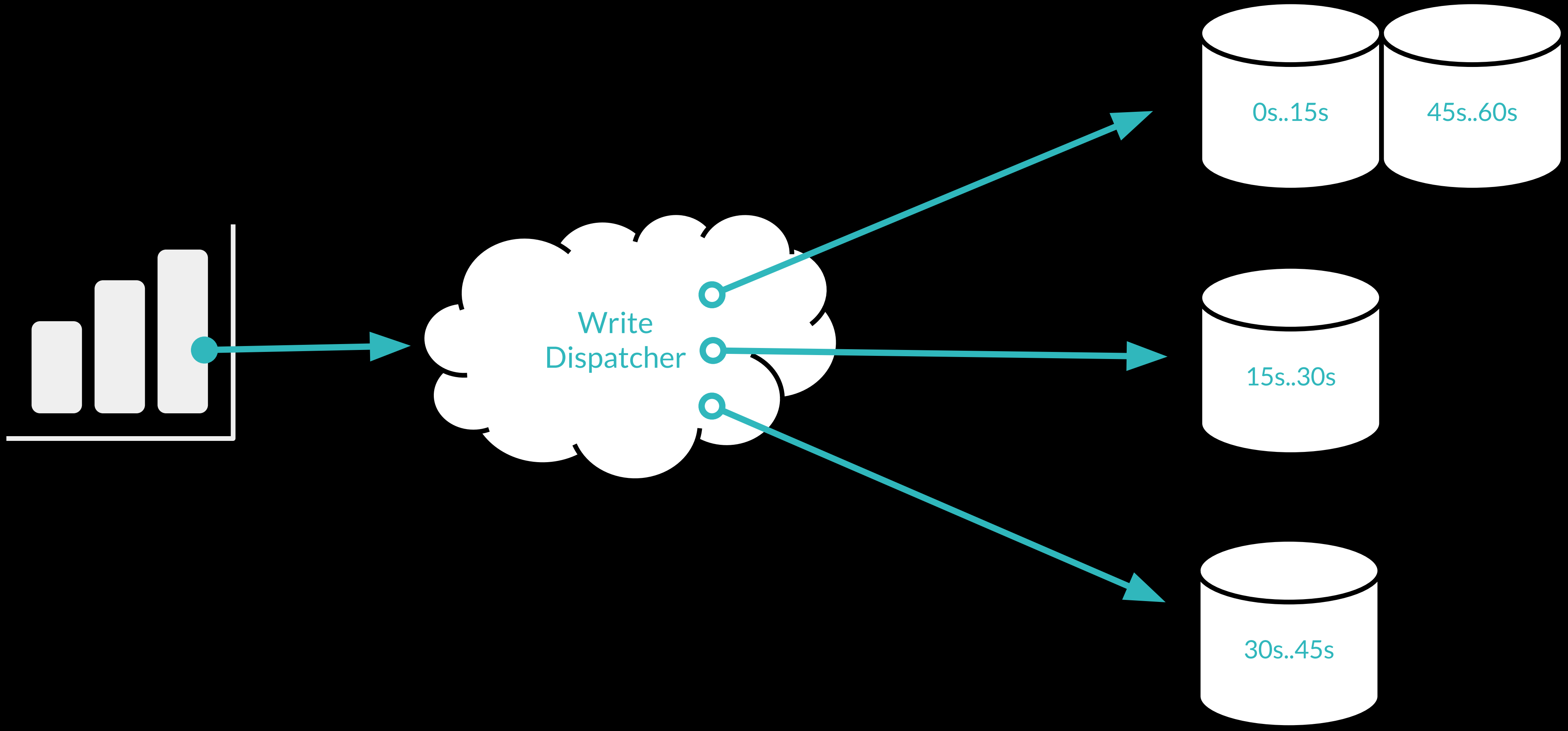
NSDb Main Features

- NSDb Sharding
  - Natural Time Sharding
  - Data Partitioning
- APIs & Connectors
- Publish Subscribe

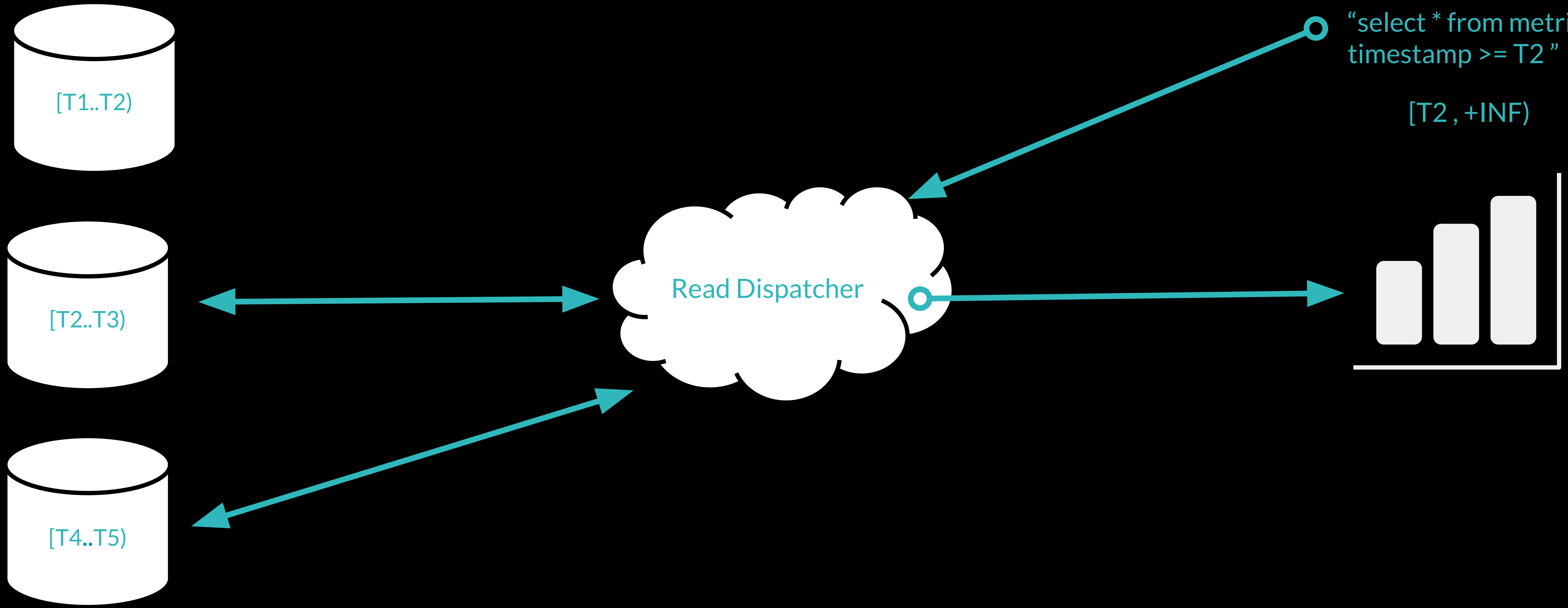
Natural Time Sharding

- Time Series points are gathered into Shards based on “event time”
- Any other partitioning will be demanded to Lucene indices
- This concept optimizes some time related frequent access patterns
- Data chunks are concatenated (and in case ordered) and not merged

Data Partitioning - Write



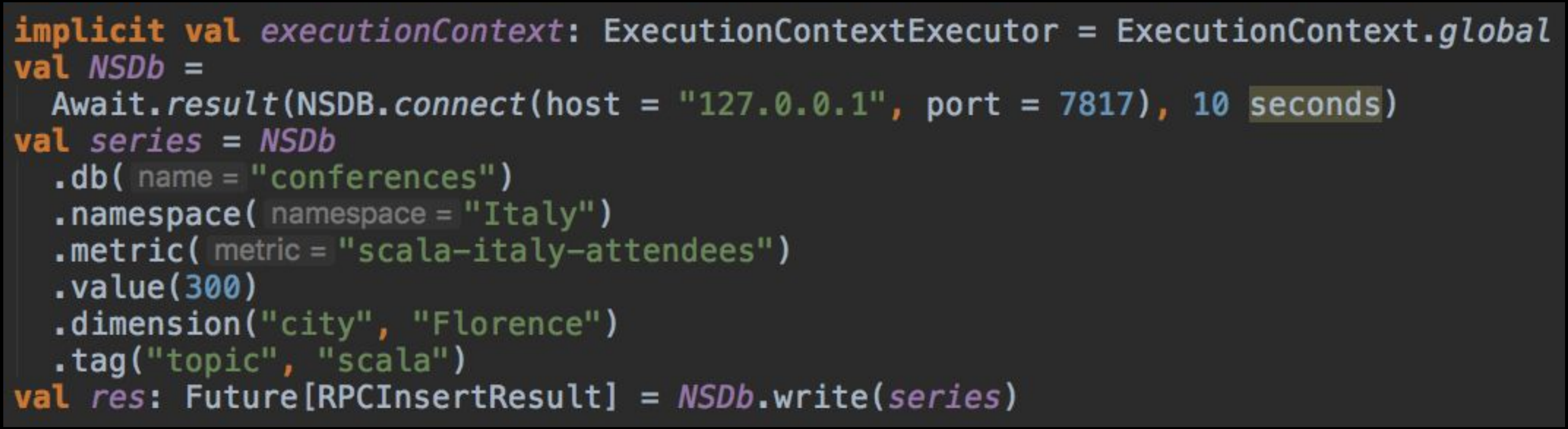
Data Partitioning - Read



APIs & Connectors

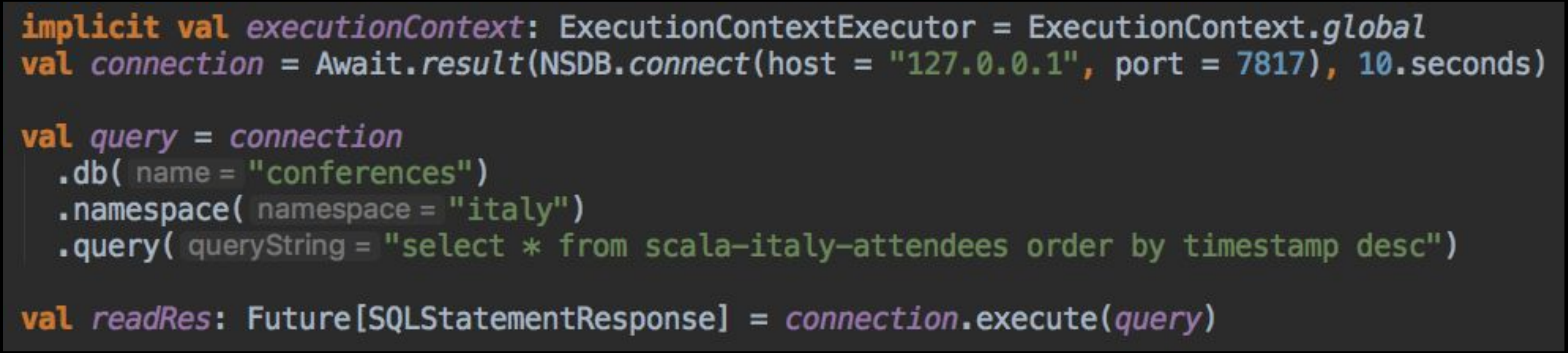
- Scala & Java APIs
- HTTP(S) APIs implemented using Akka HTTP
- WS APIs
- Flink Sink
- Kafka Connector

Scala Write APIs



```
implicit val executionContext: ExecutionContextExecutor = ExecutionContext.global
val NSDb =
  Await.result(NSDB.connect(host = "127.0.0.1", port = 7817), 10 seconds)
val series = NSDb
  .db( name = "conferences")
  .namespace( namespace = "Italy")
  .metric( metric = "scala-italy-attendees")
  .value(300)
  .dimension("city", "Florence")
  .tag("topic", "scala")
val res: Future[RPCInsertResult] = NSDb.write(series)
```

Scala Read APIs



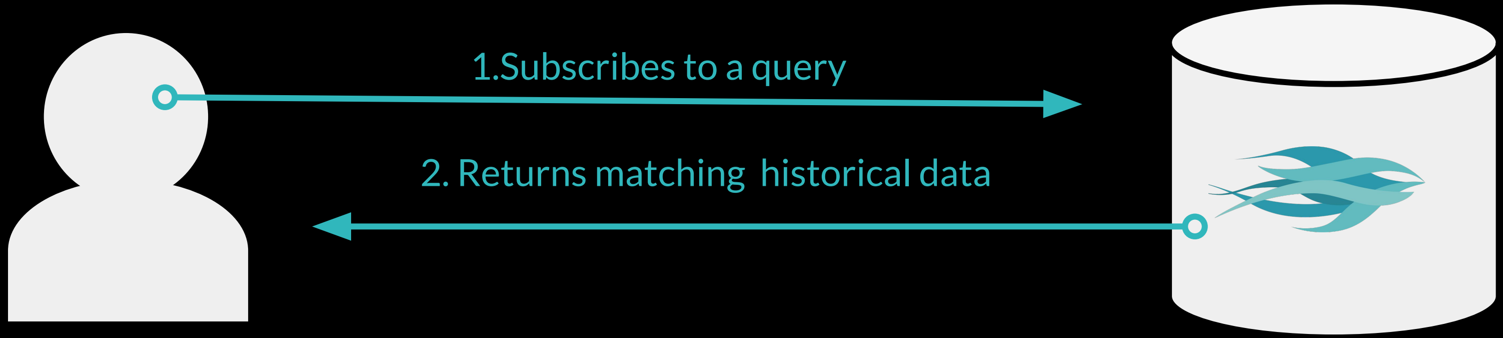
```scala
implicit val executionContext: ExecutionContextExecutor = ExecutionContext.global
val connection = Await.result(NSDB.connect(host = "127.0.0.1", port = 7817), 10.seconds)

val query = connection
  .db( name = "conferences")
  .namespace( namespace = "italy")
  .query( queryString = "select * from scala-italy-attendees order by timestamp desc")

val readRes: Future[SQLStatementResponse] = connection.execute(query)
```

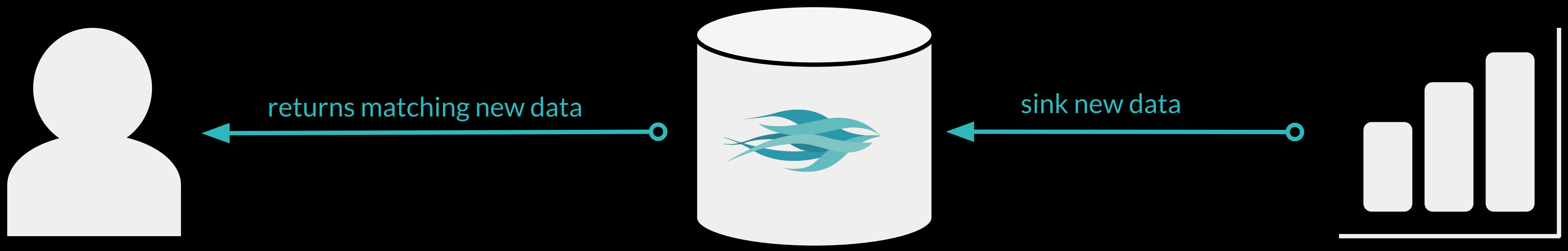
Publish-Subscribe (I)

1. User subscribes a query using WebSocket APIs
2. Historical data matching the query is returned



Publish-Subscribe (II)

3. Everytime new bits are written into NSDb, if they match user registered queries, are published on WebSocket channel



Single Node Design

Akka Recap

Overall Node Architecture

Lucene as Storage Layer

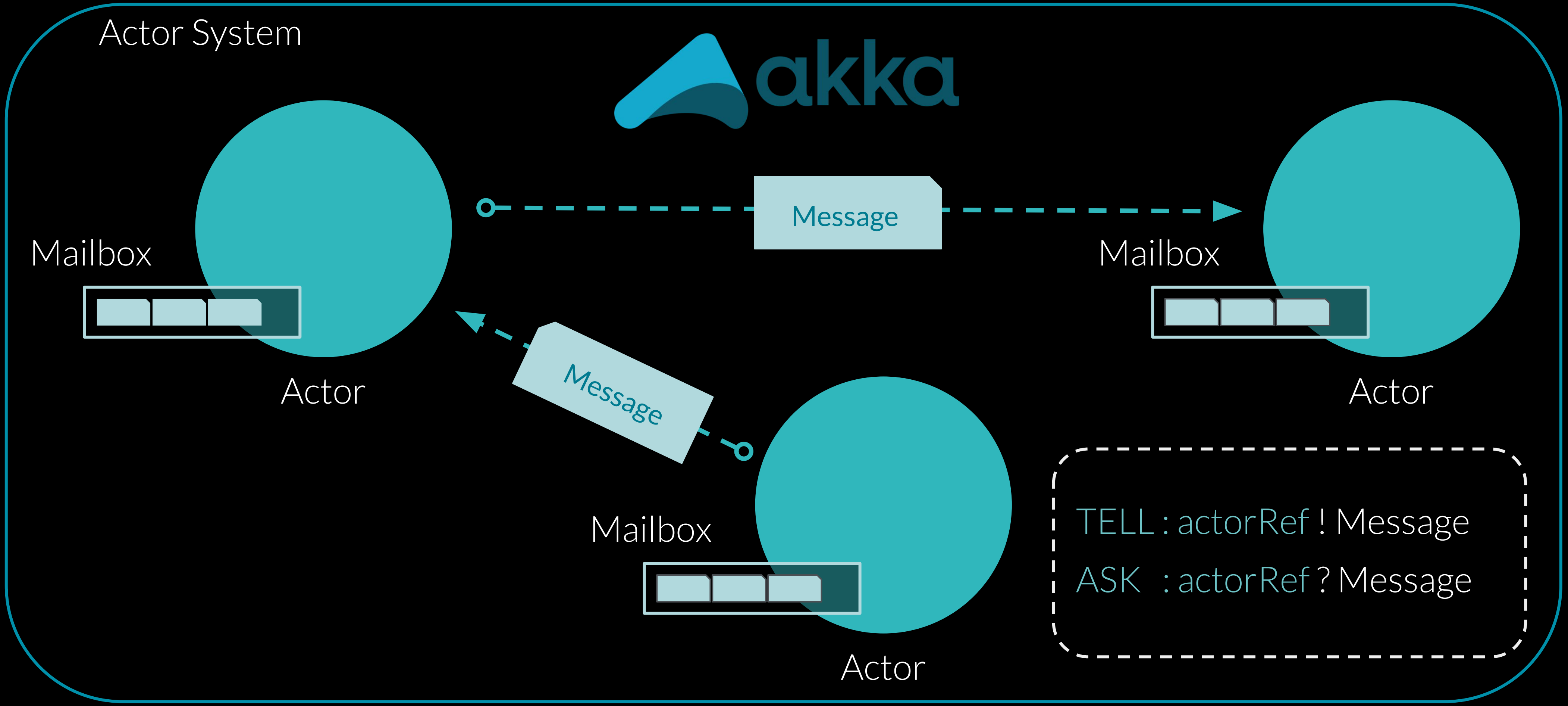
SQL Like Support

Handling mutable Lucene indices with Akka

Node actors hierarchy

Data Streaming

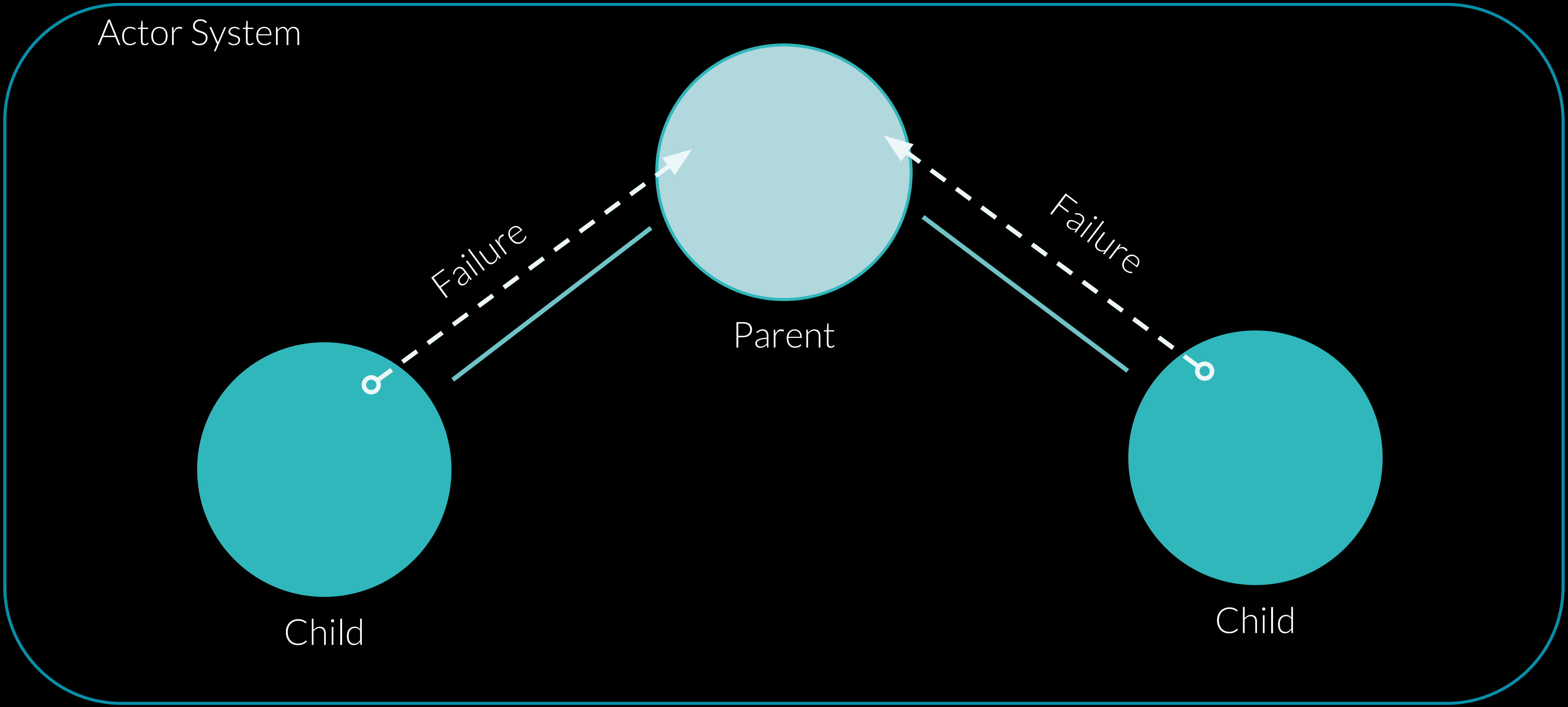
Akka Recap (I)



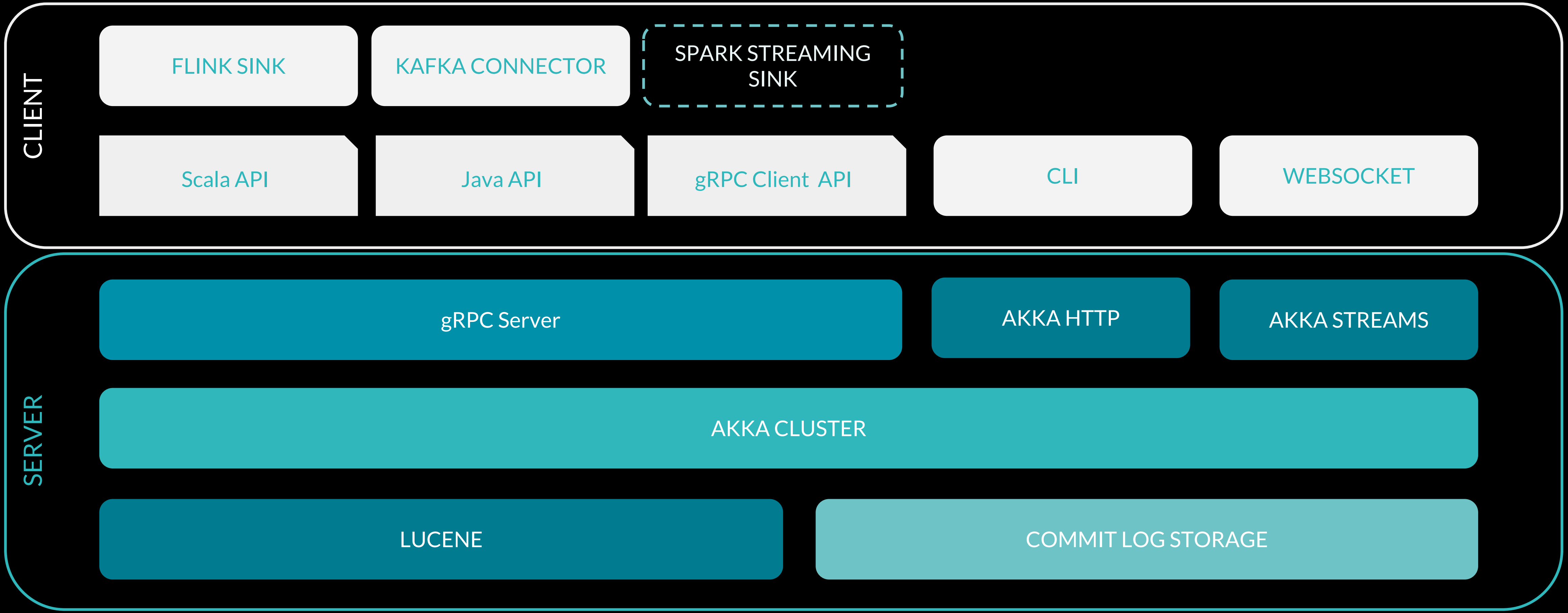
TELL : actorRef ! Message

ASK : actorRef ? Message

Akka Recap (II)



Overall Node Architecture



Lucene as Storage Layer (I)

"Apache Lucene is an open source project implementing full-featured text search engine library written entirely in Java."

- Ad Hoc indices management according to time-series handling

Lucene as Storage Layer (II)

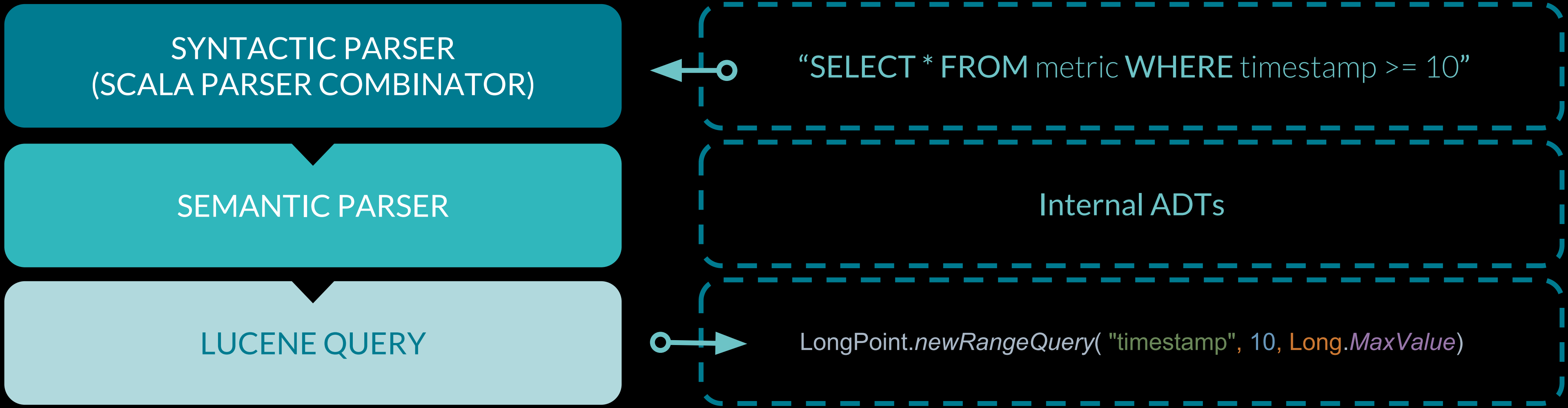
PROs:

- Stable and continuously improved project
- Scalable, High-Performance Indexing
- Very common choice in database field
- Powerful query optimization
- Java implementation

CONs:

- Lack of documentation
- Java implementation

SQL Like Support



SYNTACTIC PARSER
(SCALA PARSER COMBINATOR)

SEMANTIC PARSER

LUCENE QUERY

“SELECT * FROM metric WHERE timestamp >= 10”

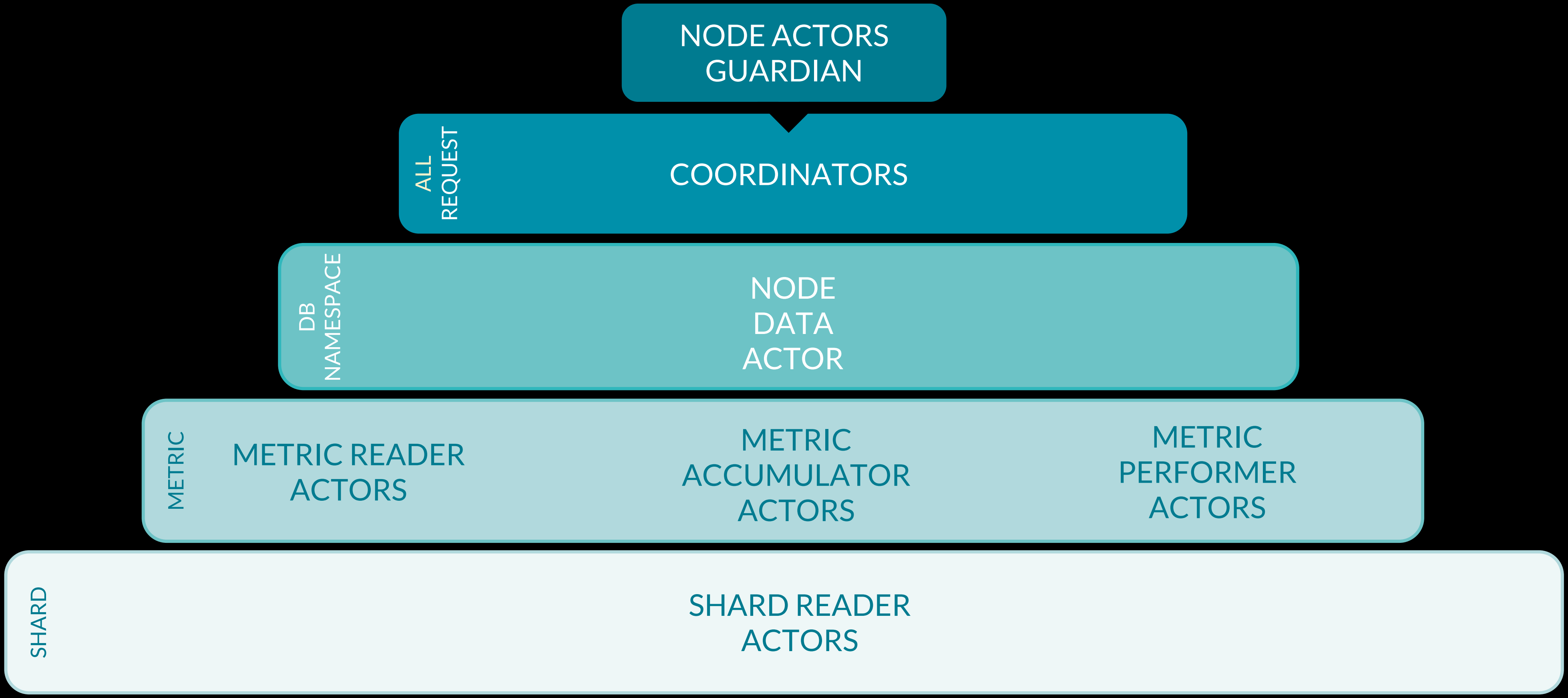
Internal ADTs

```
LongPoint.newRangeQuery( "timestamp", 10, Long.MaxValue)
```

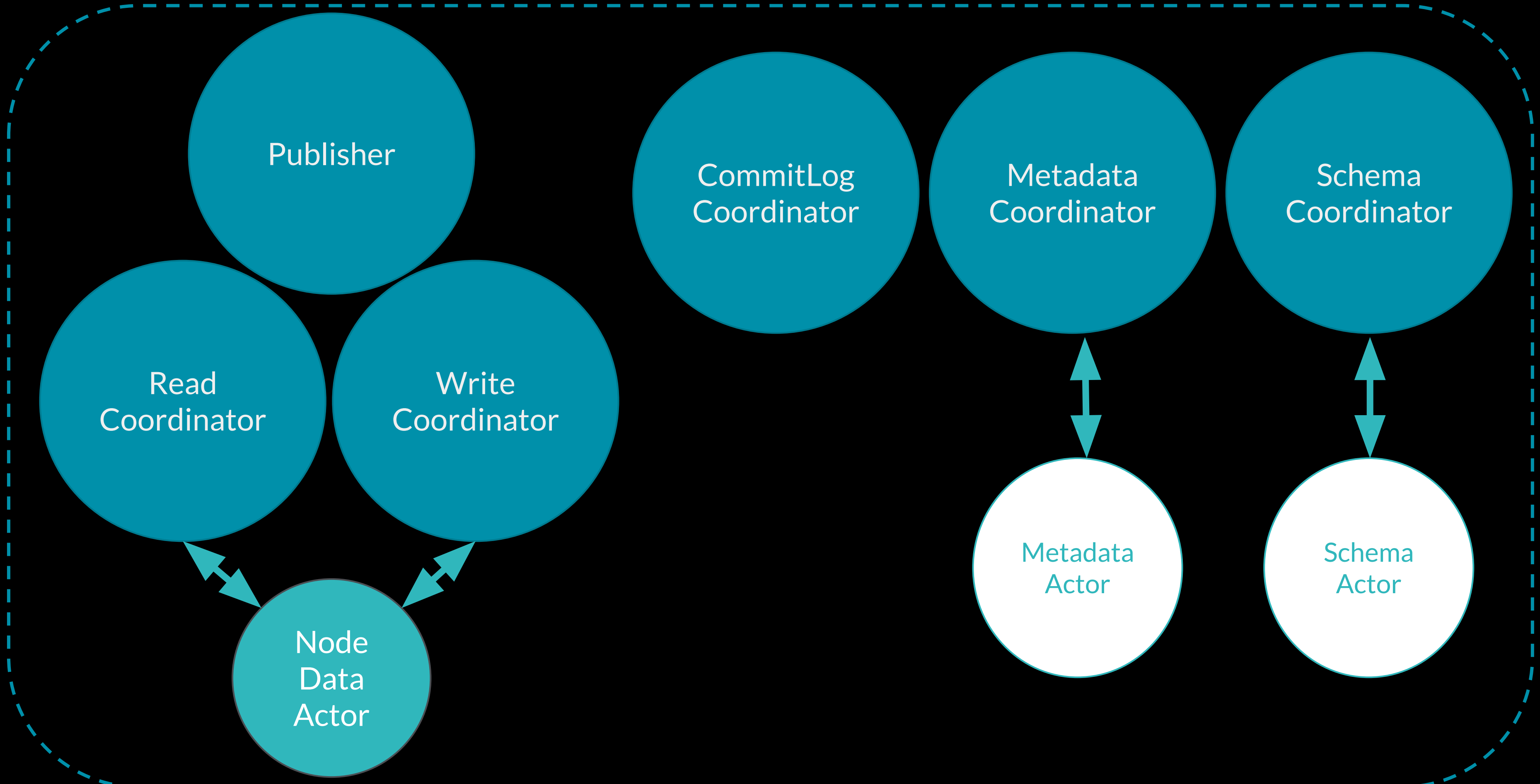
Handling mutable Lucene indices with Akka

- Usage of message passing avoids locking and blocking
- Akka Actors wraps our own Lucene access layer
- Each Actor handles a single kind of operation (read or write) on a specific index
- Scale up on single node

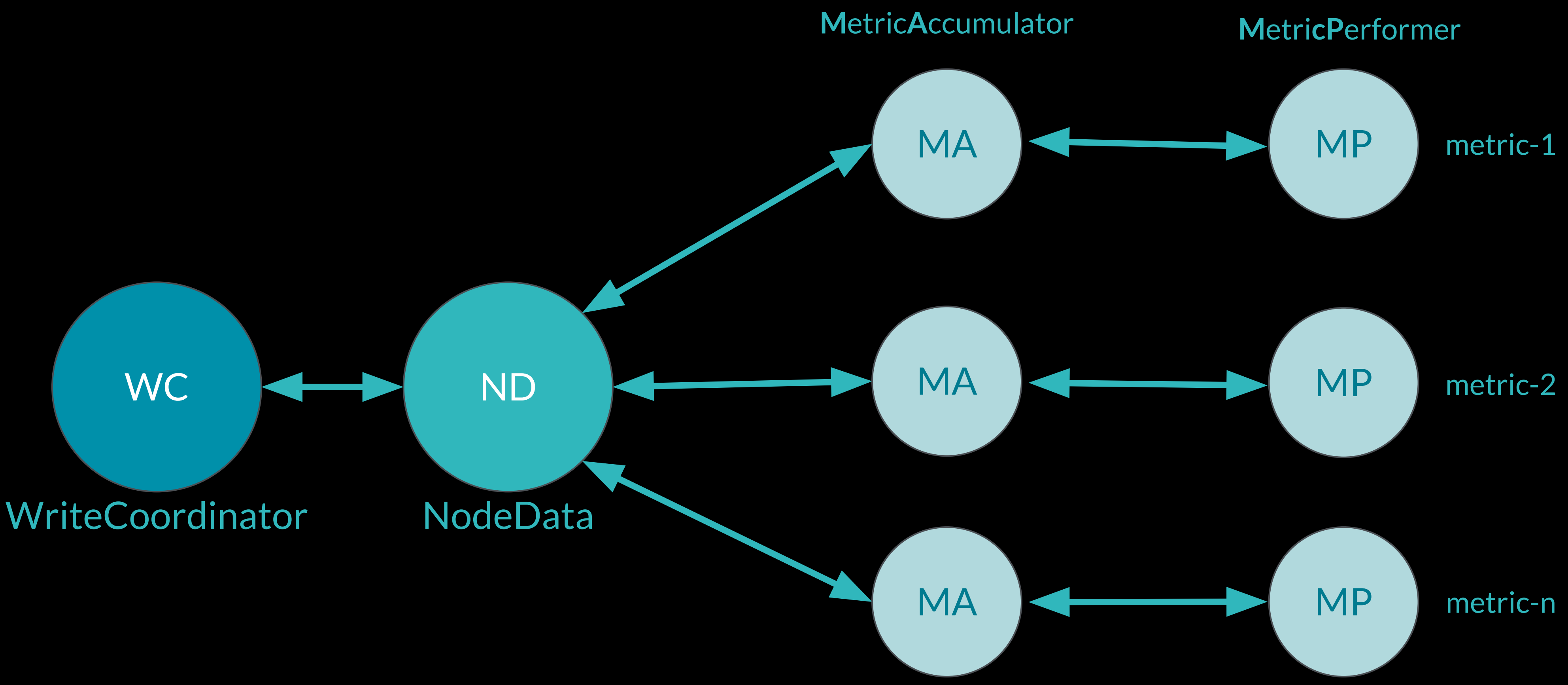
Node Actors Hierarchy



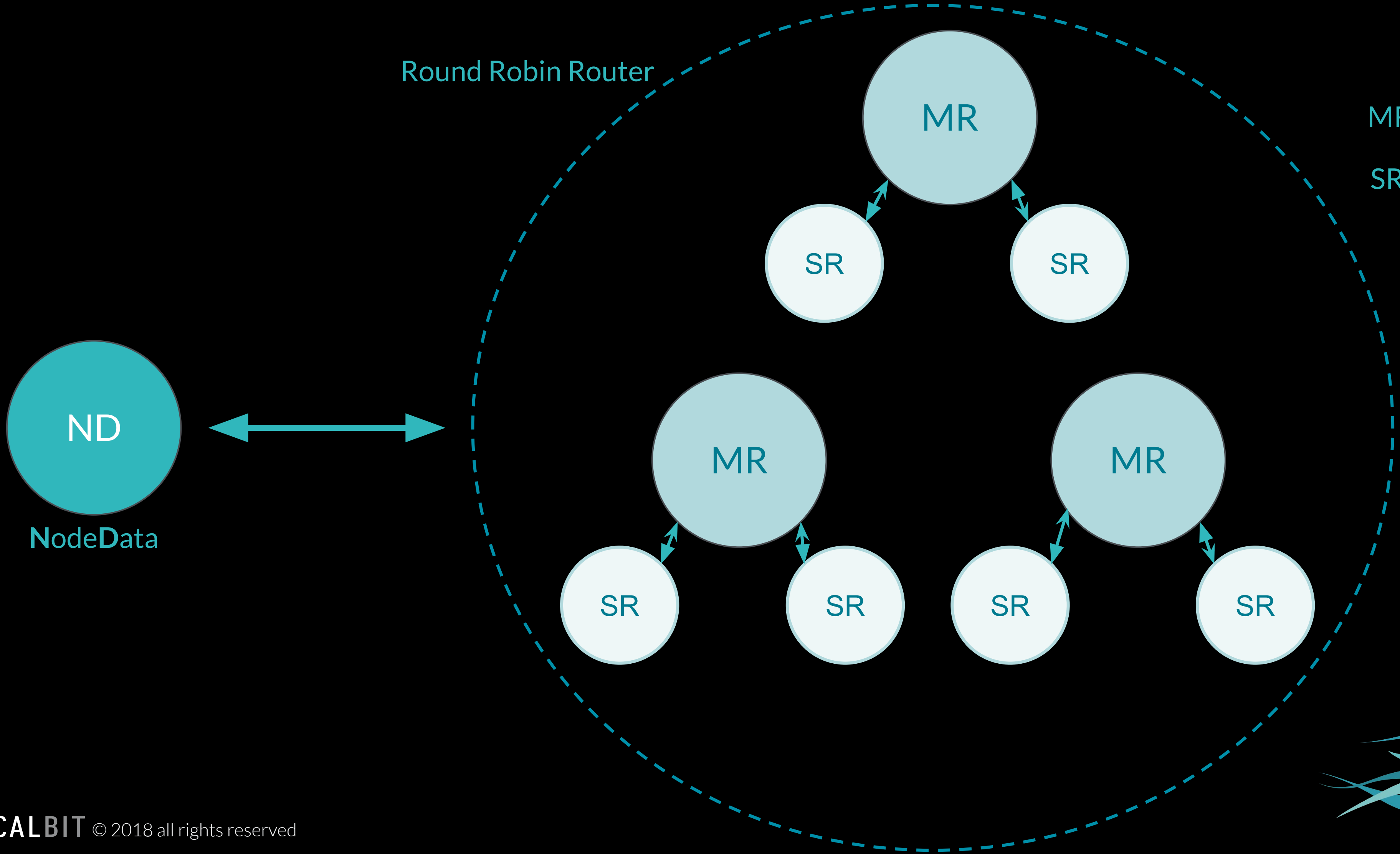
Node Actors Hierarchy - Coordinators



Node Actors Hierarchy - Write Flow



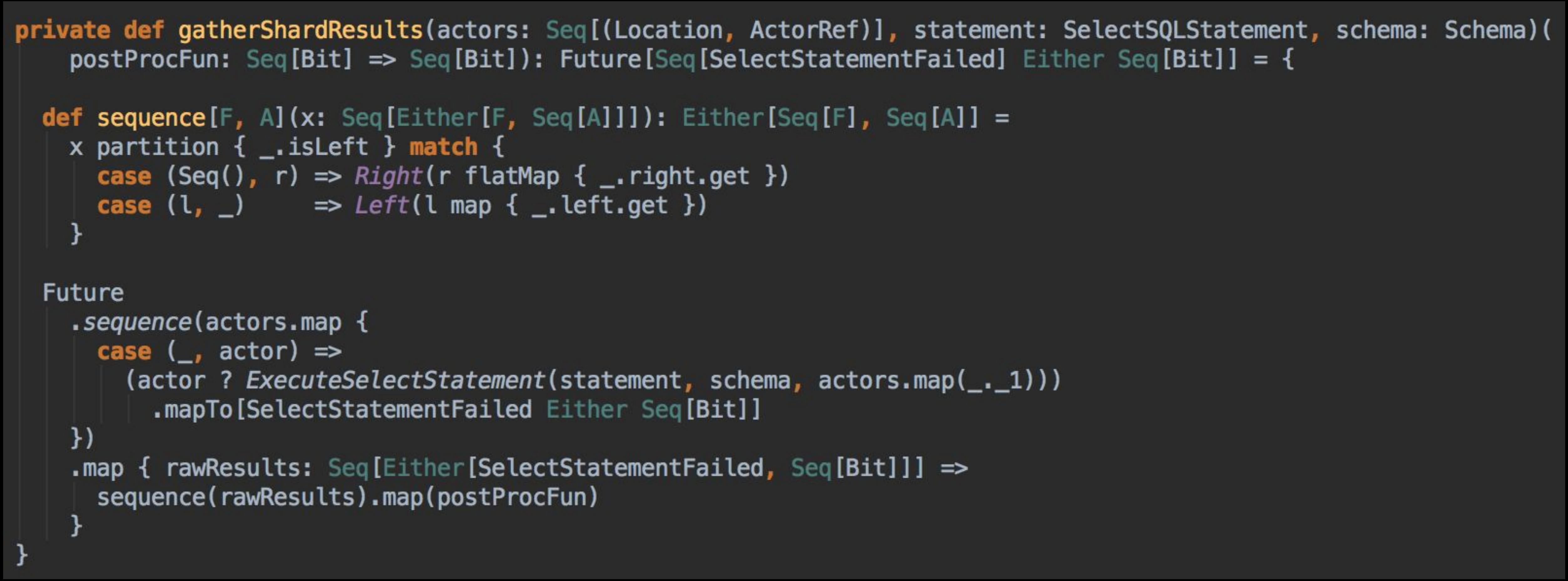
Node Actors Hierarchy - Read Flow (I)



MR = **M**etric**R**eader

SR = **S**hard**R**eader

Node Actors Hierarchy - Read Flow (II)



```scala
private def gatherShardResults(actors: Seq[(Location, ActorRef)], statement: SelectSQLStatement, schema: Schema)(
    postProcFun: Seq[Bit] => Seq[Bit]): Future[Seq[SelectStatementFailed] Either Seq[Bit]] = {

  def sequence[F, A](x: Seq[Either[F, Seq[A]]]): Either[Seq[F], Seq[A]] =
    x partition { _.isLeft } match {
      case (Seq(), r) => Right(r flatMap { _.right.get })
      case (l, _)     => Left(l map { _.left.get })
    }

  Future
    .sequence(actors.map {
      case (_, actor) =>
        (actor ? ExecuteSelectStatement(statement, schema, actors.map(_._1)))
          .mapTo[SelectStatementFailed Either Seq[Bit]]
    })
    .map { rawResults: Seq[Either[SelectStatementFailed, Seq[Bit]]] =>
      sequence(rawResults).map(postProcFun)
    }
}
```

Data Streaming

- Once a new bit is received, it's being sent to *PublisherActor*.
- If the bit matches a registered query it's sent on the corresponding WebSocket via Akka Stream flow.

Problem: unbalance in term of number and frequency between subscription commands and published bits received by *PublisherActor*.

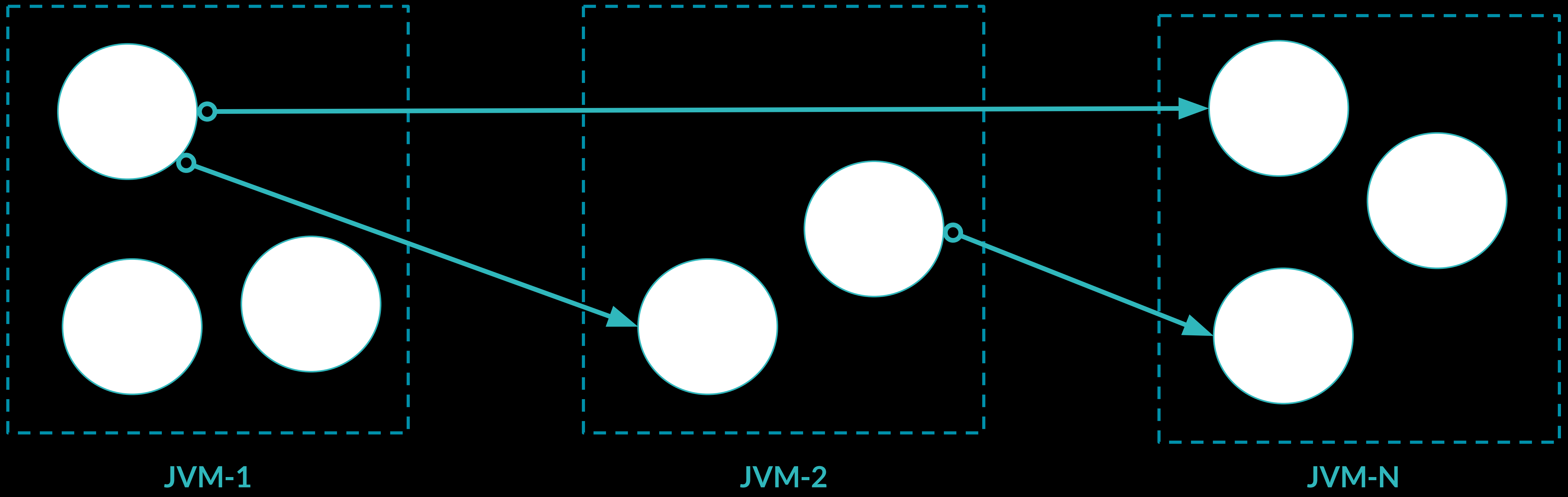
Solution: Akka *UnboundedControlAwareMailbox* implementing a priority queue for command messages.

Akka Cluster Overview

- Akka Cluster
- Akka Cluster extensions
    - Akka Distributed Data
    - Akka Distributed Publish Subscribe

Akka Cluster (I)

"A set of nodes joined together through a membership service"



Akka Cluster (II)

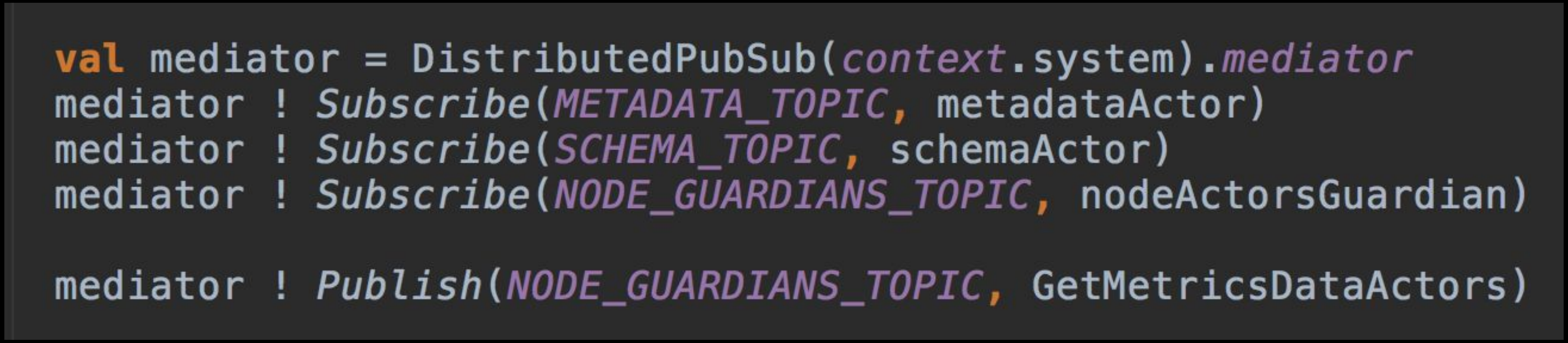
- P2P
- Gossip protocol and failure detection
- Event based notification
- Metrics Collector
- Useful Extensions

Akka Distributed Data

- Akka Distributed Data is useful when you need to share data between nodes in an Akka Cluster.
- It is designed as a key-value store, where the values are Conflict Free Replicated Data Types (CRDTs).
- Supports many data types (Set, Map, Counter etc.)
- Supports different consistency levels for writes and reads
- It's not designed to handle big data

Akka Distributed Publish Subscribe

- Actors can subscribe to a named topic
- Messages are published to a named topic
- The message will be delivered to all subscribers of the topic
- Each node interact with the *DistributedPubSubMediator*
- *At most once* delivery guarantee



```
val mediator = DistributedPubSub(context.system).mediator
mediator ! Subscribe(METADATA_TOPIC, metadataActor)
mediator ! Subscribe(SCHEMA_TOPIC, schemaActor)
mediator ! Subscribe(NODE_GUARDIANS_TOPIC, nodeActorsGuardian)

mediator ! Publish(NODE_GUARDIANS_TOPIC, GetMetricsDataActors)
```

Distributed Design

Overall Architecture

State Replication

Data Replication

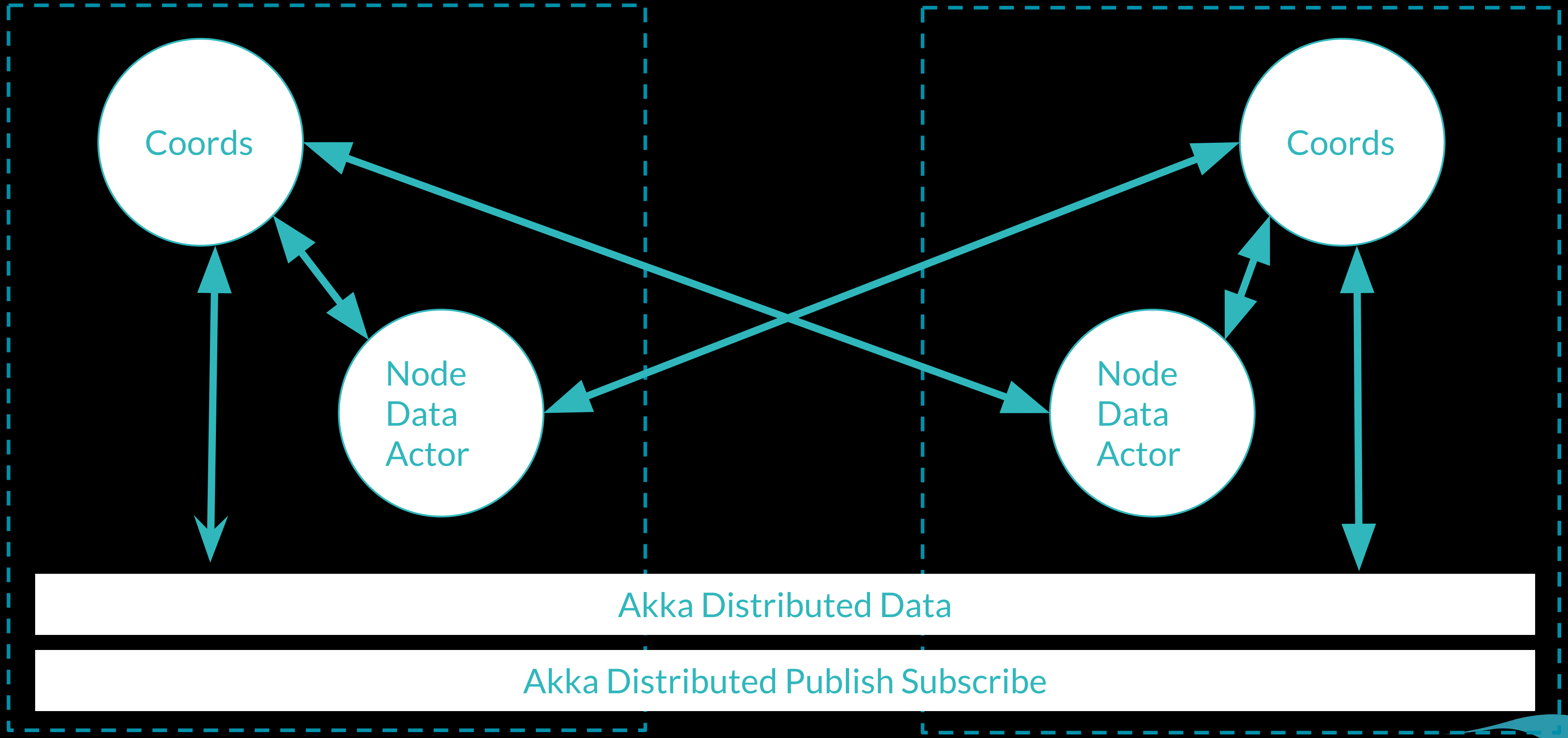
Distributed Write Model

Distributed Read Model

Error Management

Overall Architecture

- **Multimaster replication**, each node can read and write data

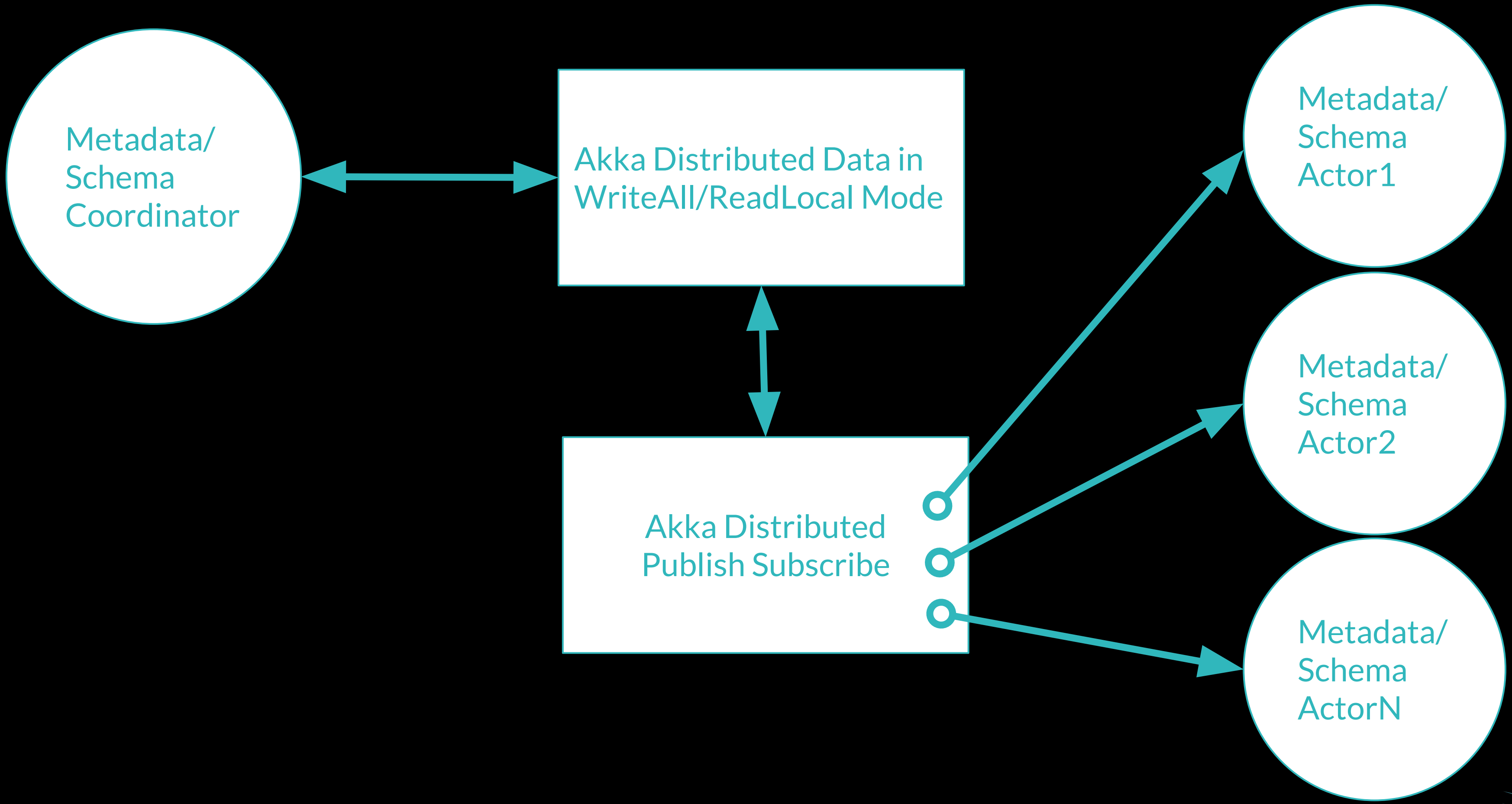


Heartbeat protocol

- Leverages Distributed Publish Subscribe
- Every Coordinator is subscribed to a dedicated topic as well as the guardians
- A cluster singleton actor periodically asks guardians to send their data actors reference.
- Cluster events trigger delta updates spread:
    - if a node joins, an add event is disseminated
    - if a node leaves, a remove event is disseminated

State Replication

State = shards locations + schemas

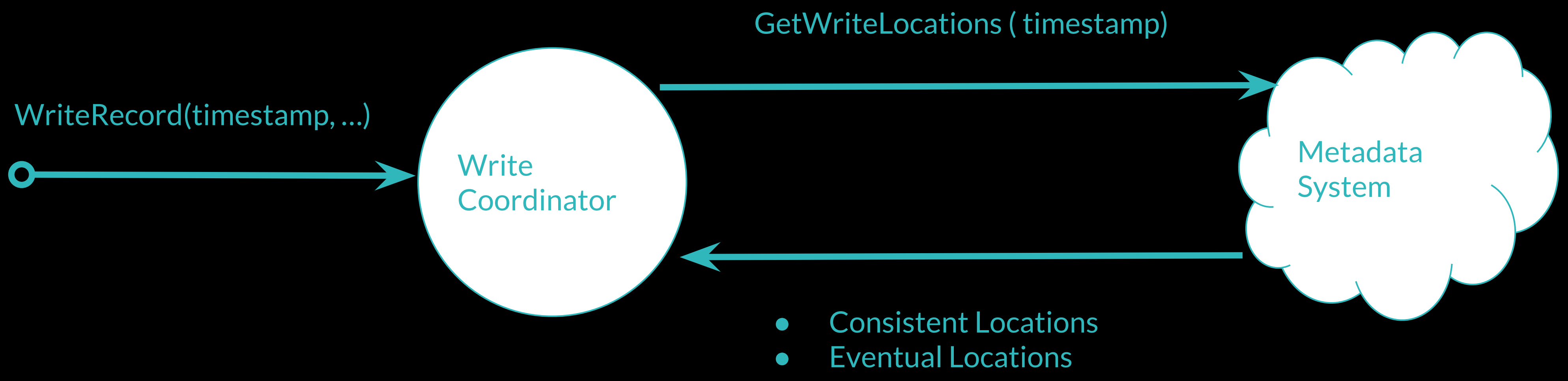


Data Replication

- **Active-active replication** approach
- NSDb implements two levels of replicas in terms of consistency
  - **Consistent replicas**: A record must be correctly acknowledge to all those nodes before the ack can be returned to the caller
  - **Eventual replicas**: the records will be written asynchronously (it fails silently)

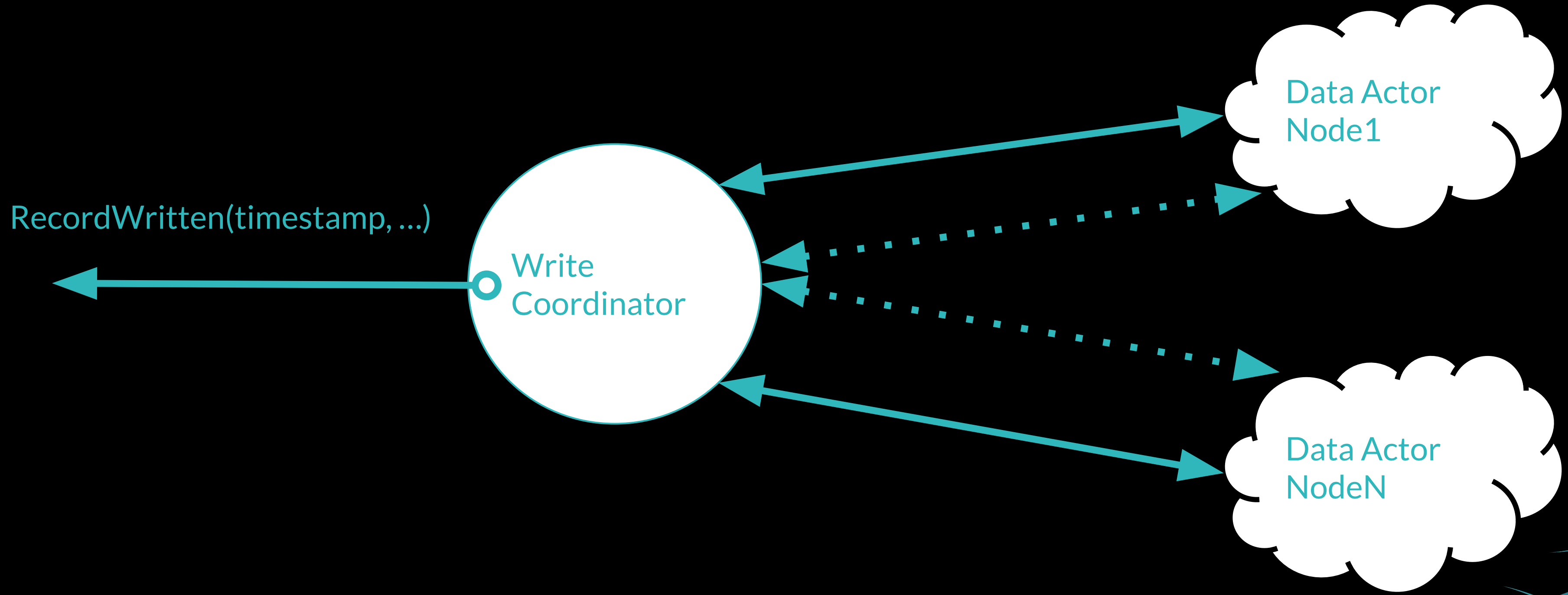
Distributed Write Model (I)

1. Record validation
2. Consistent and eventual write locations gathering



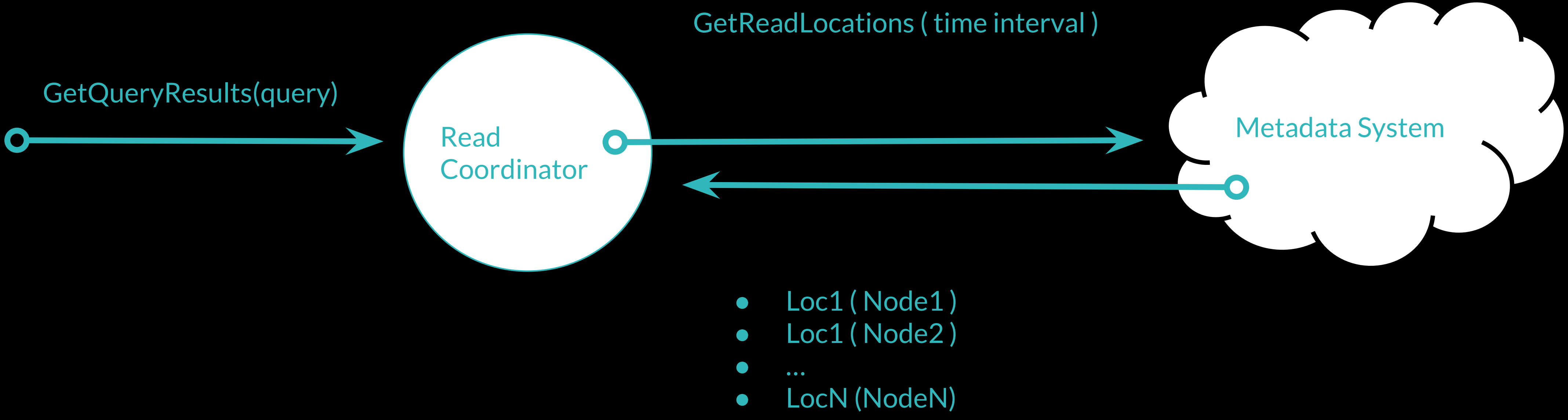
Distributed Write Model (II)

3. Data on Consistent locations written and acknowledge returned to the caller
4. Silently, writes on eventual locations performed



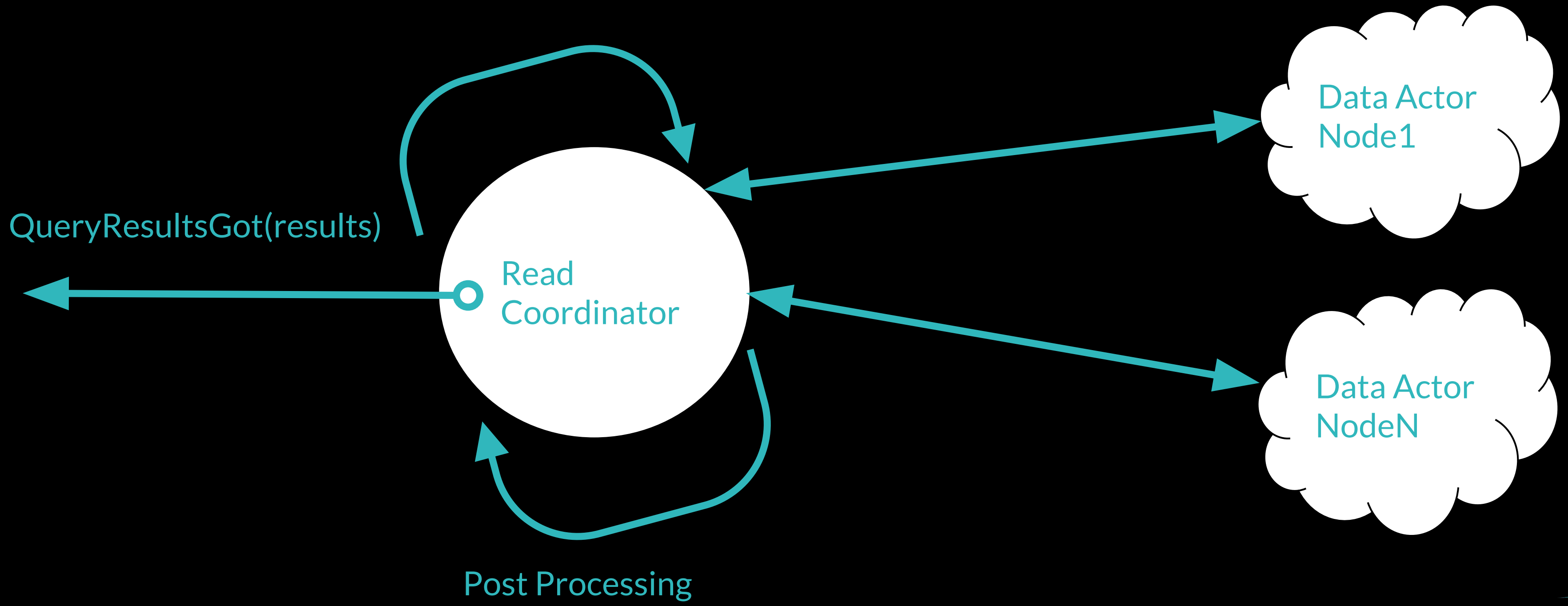
Distributed Read Model (I)

1. Extract time interval from input query where condition (if present)
2. Get locations from metadata system



Distributed Read Model (II)

3. Reduce location lists to one per location
4. Nodes results retrieving (parallel requests to every Node)
5. Post Processing and return result



Error Management (I)

- Write to a set of replicas == distributed transaction
- No isolation
- Saga pattern is applied

Error Management (II)

credits: @victorklang

Roadmap

- Enhance location selection algorithm
- Cluster Monitoring
- Container Orchestration System Support
- Bit TTL
- SQL Engine improvements

Community Edition

NSDb is released under :

Apache 2 License

Reach us on :

https://github.com/radicalbit/NSDb

Enterprise Edition

- Support
- Security
  - OpenID and OAuth support
  - Kerberos Support
- Metric Versioning

Q&A

GRAZIE!

<radicalbit.team/>

info@radicalbit.io