HOW TO CONNECT SPARK TO YOUR OWN DATASOURCE

Ross Lawley
MongoDB


```
{ name: "Ross Lawley", role: "Senior Software Engineer", twitter: "@RossC0" }
```

MongoDB Spark connector timeline

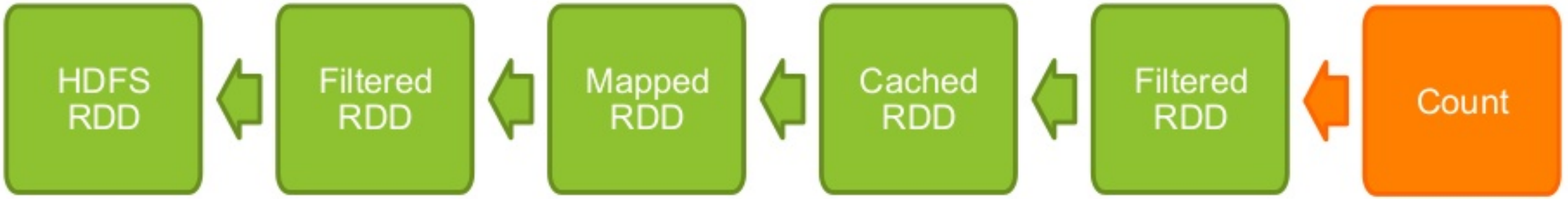
- Initial interest
  - Skunkworks project in March 2015
  - Introduction to Big Data with Apache Spark
  - Intern project in Summer 2015

- Official Project started in Jan 2016
  - Written in Scala with very little Java needed
  - Python and R support via SparkSQL
  - Much interest internally and externally

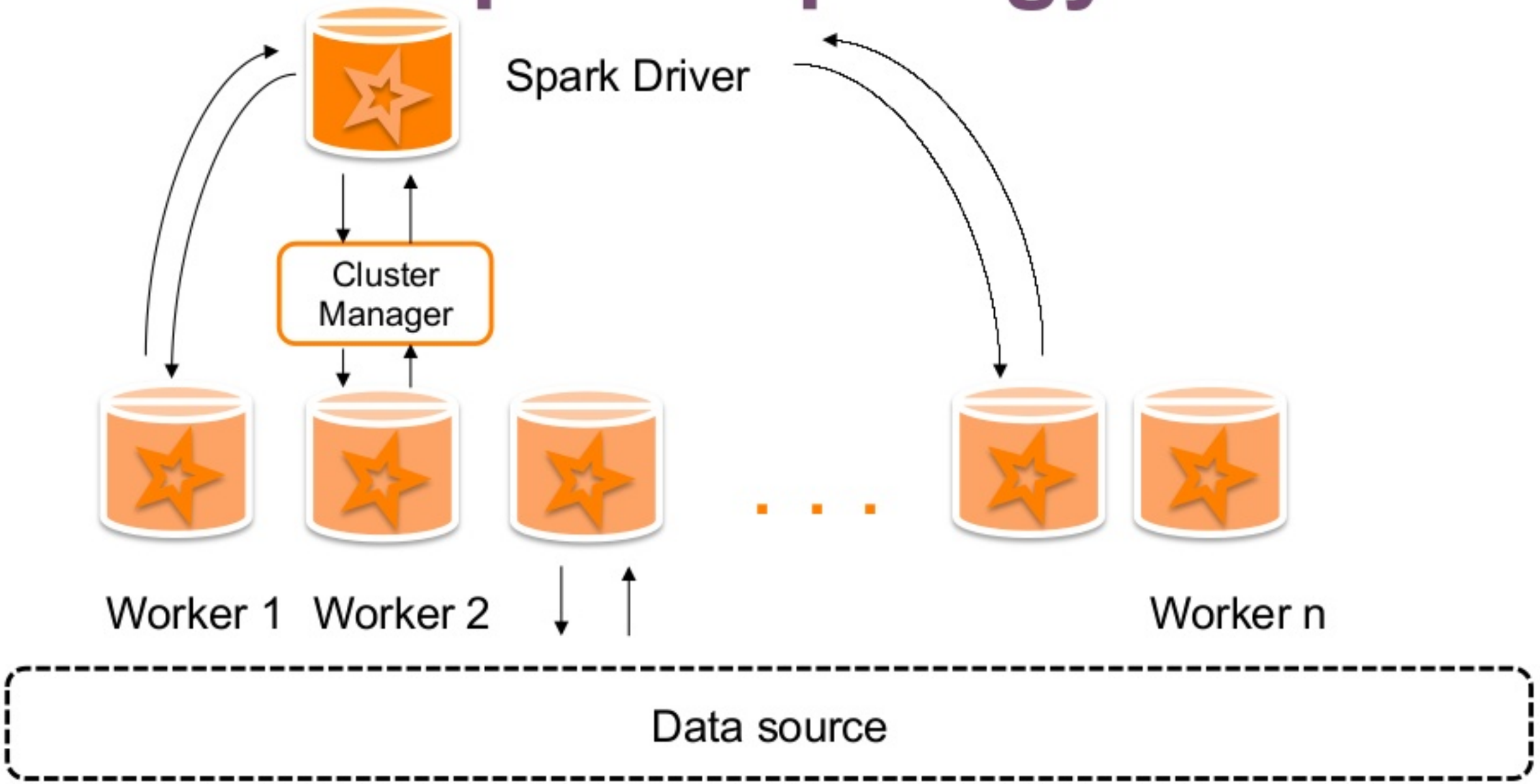
Spark 101 - the core

RDDs maintain *lineage* information that can be used to reconstruct lost partitions

```
val searches = spark.textFile("hdfs://...")
                    .filter(_.contains("Search"))
                    .map(_.split("\t")(2)).cache()
                    .filter(_.contains("MongoDB"))
                    .count()
```

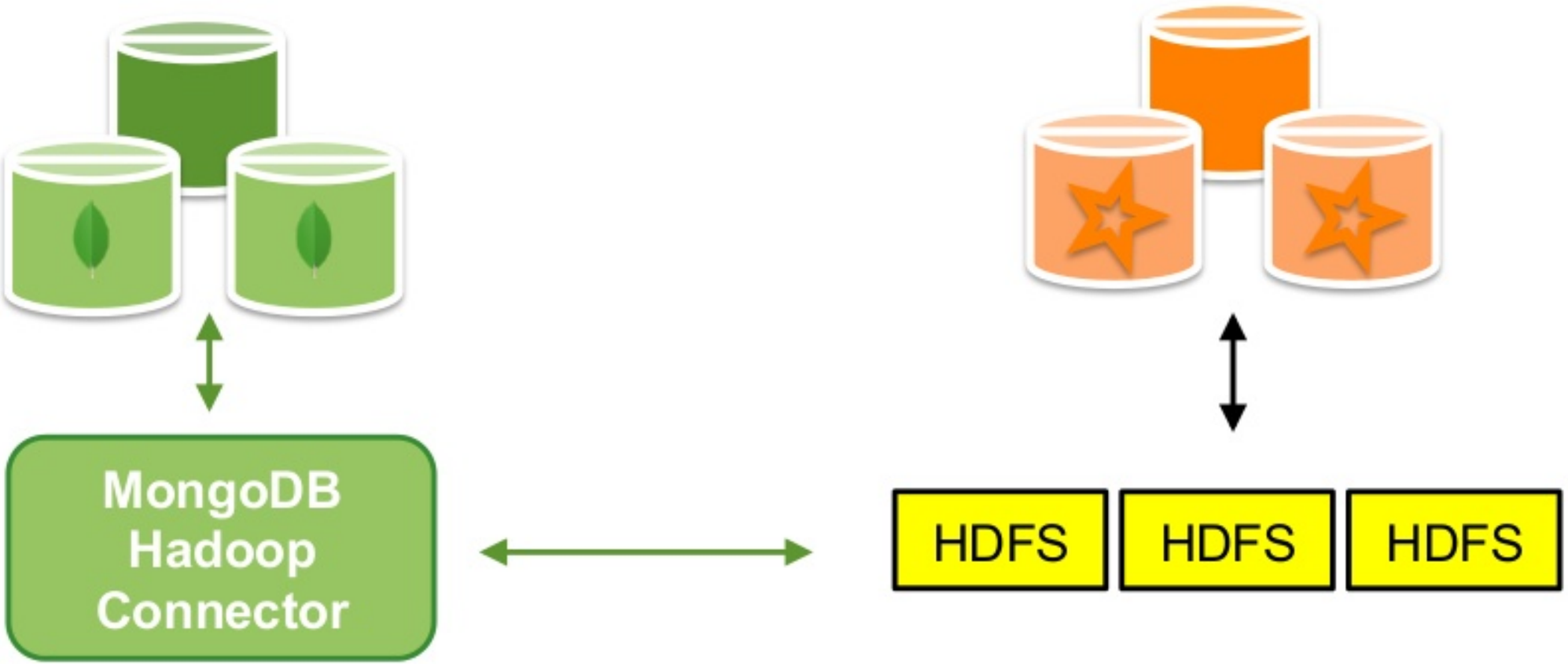


Spark topology




Spark
&
mongoDB.
FOR GIANT IDEAS
SPARK SUMMIT
EUROPE 2016

Prior to the Spark Connector



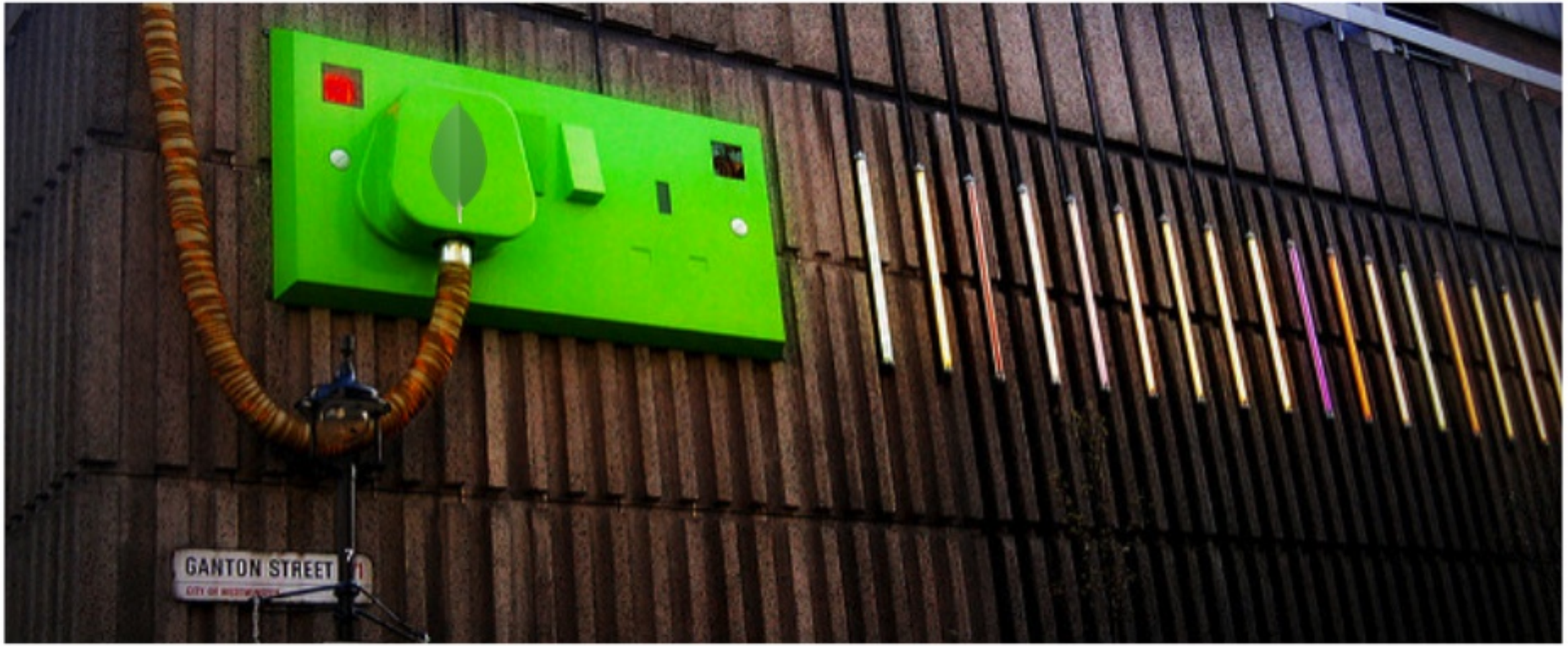
The MongoDB Spark Connector

Roll your own connector

The Golden rule

Learn from other connectors

1. Connecting to your data source



https://www.flickr.com/photos/memake/5506361372

Making a connection

- Has a cost
  - The Mongo Java Driver runs a connection pool
    Authenticates connections, replica set discovery etc..
- There are two modes to support
  - Reading
  - Writing

Connections need configuration

- **Read Configuration**
  - URI, database name and collection name
  - Partitioner
  - Sample Size (for inferring the schema)
  - Read Preference, Read Concern
  - Local threshold (for choosing the MongoS)
- **Write Configuration**
  - URI, database name and collection name
  - Write concern
  - Local threshold (for choosing the MongoS)

Connector

```
case class MongoConnector(mongoClientFactory: MongoClientFactory) extends
    Logging with Serializable with Closeable {

  def withMongoClientDo[T](code: MongoClient => T): T = {
    val client = acquireClient()
    try {
      code(client)
    } finally {
      releaseClient(client)
    }
  }


  def withDatabaseDo[T](config: MongoCollectionConfig, code: MongoDatabase => T): T =
    withMongoClientDo({ client => code(client.getDatabase(config.databaseName)) })

  def withCollectionDo[D, T](config: MongoCollectionConfig, code: MongoCollection[D] => T) …

  private[spark] def acquireClient(): MongoClient = mongoClientCache.acquire(mongoClientFactory)
  private[spark] def releaseClient(client: MongoClient): Unit = mongoClientCache.release(client)
}
```

Connection Optimization

- MongoClientCache
  - A lockless cache for MongoClients.
  - Allowing multiple tasks access to a MongoClient.
  - Keeps clients alive for a period of time.
  - Timeouts and closes old.

Making a connection

- Should be cheap as possible
  - Broadcast it so it can be reused.
  - Use a timed cache to promote reuse and ensure closure of resources.
- Configuration should be flexible
  - Spark Configuration
  - Options – Map[String, String]
  - ReadConfig / WriteConfig instances

2. Read data

https://www.flickr.com/photos/ericejohnson/4408073661/

Implement RDD[T] class

- Partition the collection
- Optionally provide preferred locations of a partition
- Compute the partition into an Iterator[T]

Partition

```scala
/**
 * An identifier for a partition in an RDD.
 */
trait Partition extends Serializable {
  /**
   * Get the partition's index within its parent RDD
   */
  def index: Int

  // A better default implementation of HashCode
  override def hashCode(): Int = index

  override def equals(other: Any): Boolean = super.equals(other)
}
```

MongoPartition

Very simple, stores the information on how to get the data.

```
case class MongoPartition(
    index: Int,
    queryBounds: BsonDocument,
    locations: Seq[String]) extends Partition
```

MongoPartitioner

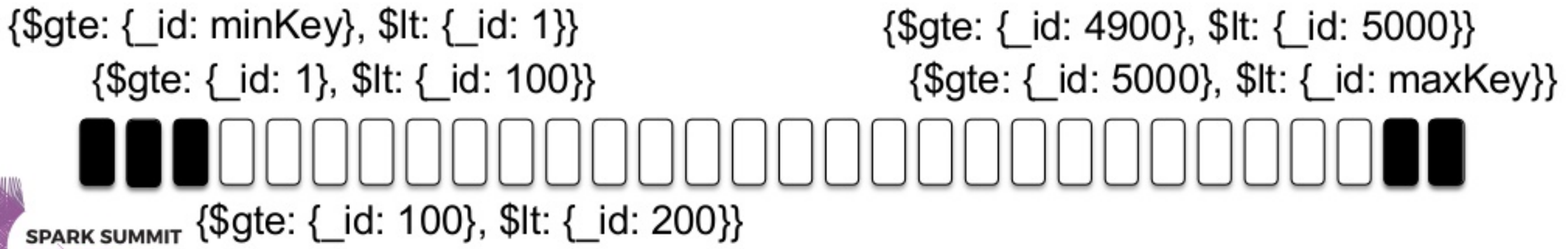
```
/**
 * The MongoPartitioner provides the partitions of a collection
 */
trait MongoPartitioner extends Logging with Serializable {

  /**
   * Calculate the Partitions
   *
   * @param connector the MongoConnector
   * @param readConfig the [[com.mongodb.spark.config.ReadConfig]]
   * @return the partitions
   */
  def partitions(connector: MongoConnector,
                 readConfig: ReadConfig,
                 pipeline: Array[BsonDocument]): Array[MongoPartition]

}
```

MongoSamplePartitioner

- Over samples the collection
  - Calculate the number of partitions.
    Uses the average document size and the configured partition size.
  - Samples the collection, sampling *n* number of documents per partition
  - Sorts the data by partition key
  - Takes each *n* partition
  - Adds a min and max key partition split at the start and end of the collection



MongoShardedPartitioner

- Examines the shard config database
  - Creates partitions based on the shard chunk min and max ranges
  - Stores the Shard location data for the chunk, to help promote locality
  - Adds a min and max key partition split at the start and end of the collection

{$gte: {_id: minKey}, $lt: {_id: 1}}

{$gte: {_id: 1000}, $lt: {_id: maxKey}}

Alternative Partitioners

- **MongoSplitVectorPartitioner**
  A partitioner for standalone or replicaSets. Command requires special privileges.

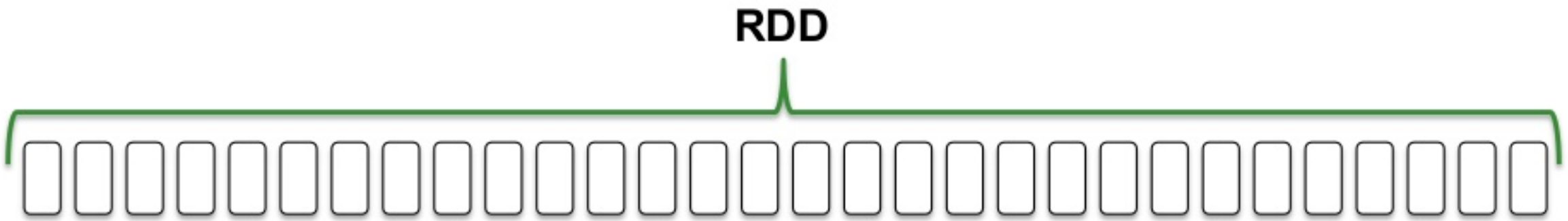
- **MongoPaginateByCountPartitioner**
  Creates a maximum number of partitions
  Costs a query to calculate each partition

- **MongoPaginateBySizePartitioner**
  As above but using average document size to determine the partitions.

- **Create your own**
  Just implement the MongoPartitioner trait and add the full path to the config

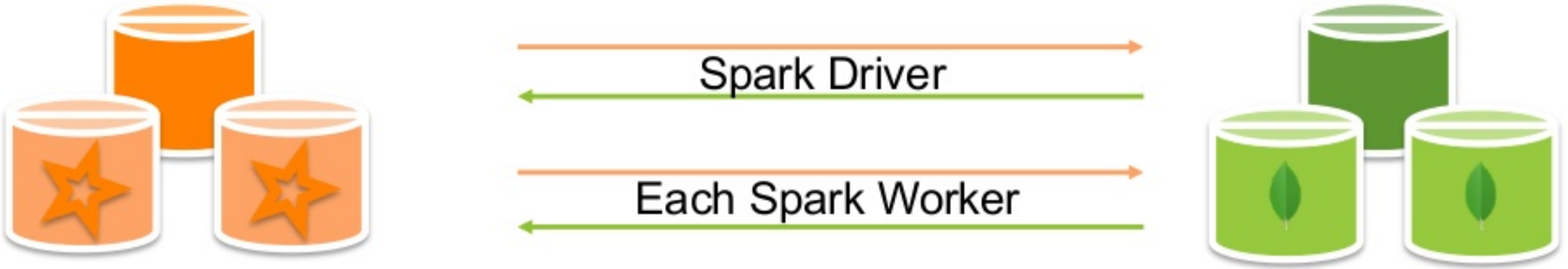
Partitions

- They are the foundations for RDD's
- Super simple concept
- Challenges for mutable data sources as not a snapshot in time



Reads under the hood

```
MongoSpark.load(sparkSession).count()
```



1. Create a MongoRDD[Document]
2. Partition the data
3. Calculate the Partitions
4. Get the preferred locations and allocate workers
5. For each partition:
   i. Queries and returns the cursor
   ii. Iterates the cursor and sums up the data
6. Finally, the Spark application returns the sum of the sums.

Reads

- Data **must** be serializable
- Partitions provide parallelism
- Partitioners should be configurable

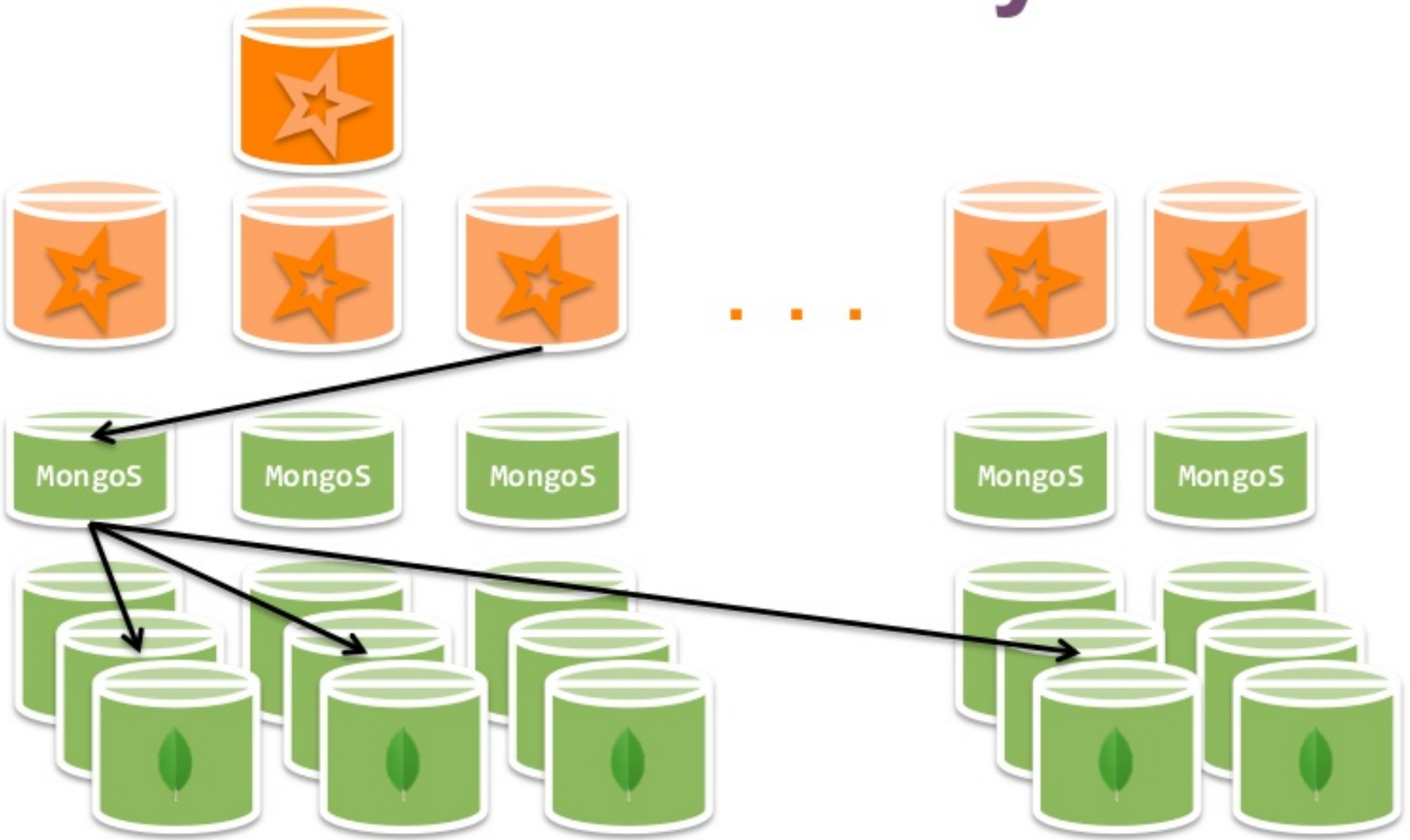
  No one size fits all
- Partitioning strategy may be non obvious
  - Allow users to solve partitioning in their own way

Read Performance

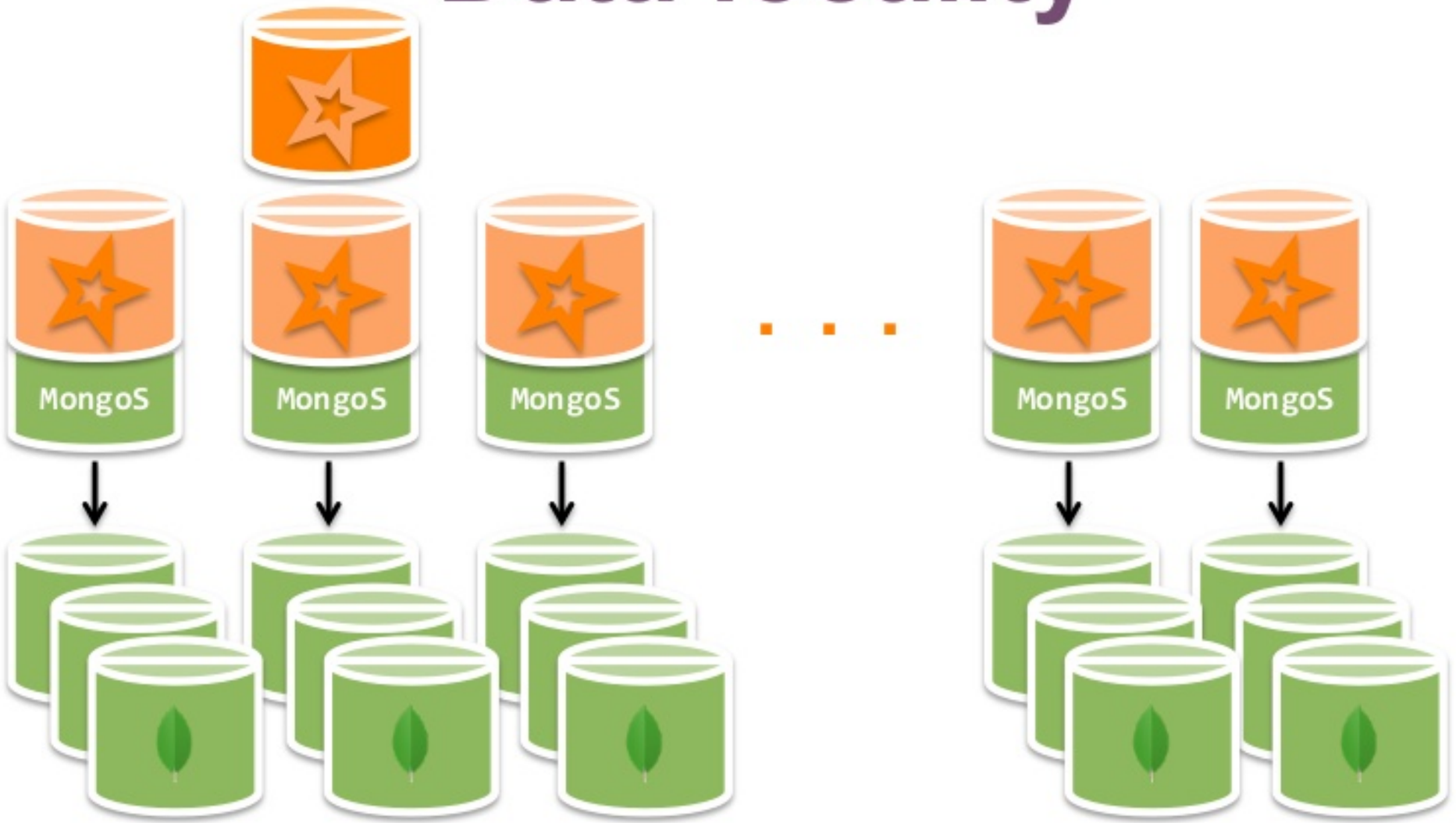
- MongoDB Usual Suspects
  - Document design
  - Indexes
  - Read Concern
- Spark Specifics
  - Partitioning Strategy
  - Data Locality

Data locality

Data locality



Data locality



Configure: LocalThreshold, MongoShardedPartitioner

Data locality

Configure: ReadPreference, LocalThreshold, MongoShardedPartitioner

3. Writing data

https://www.flickr.com/photos/knightjh/3415926549

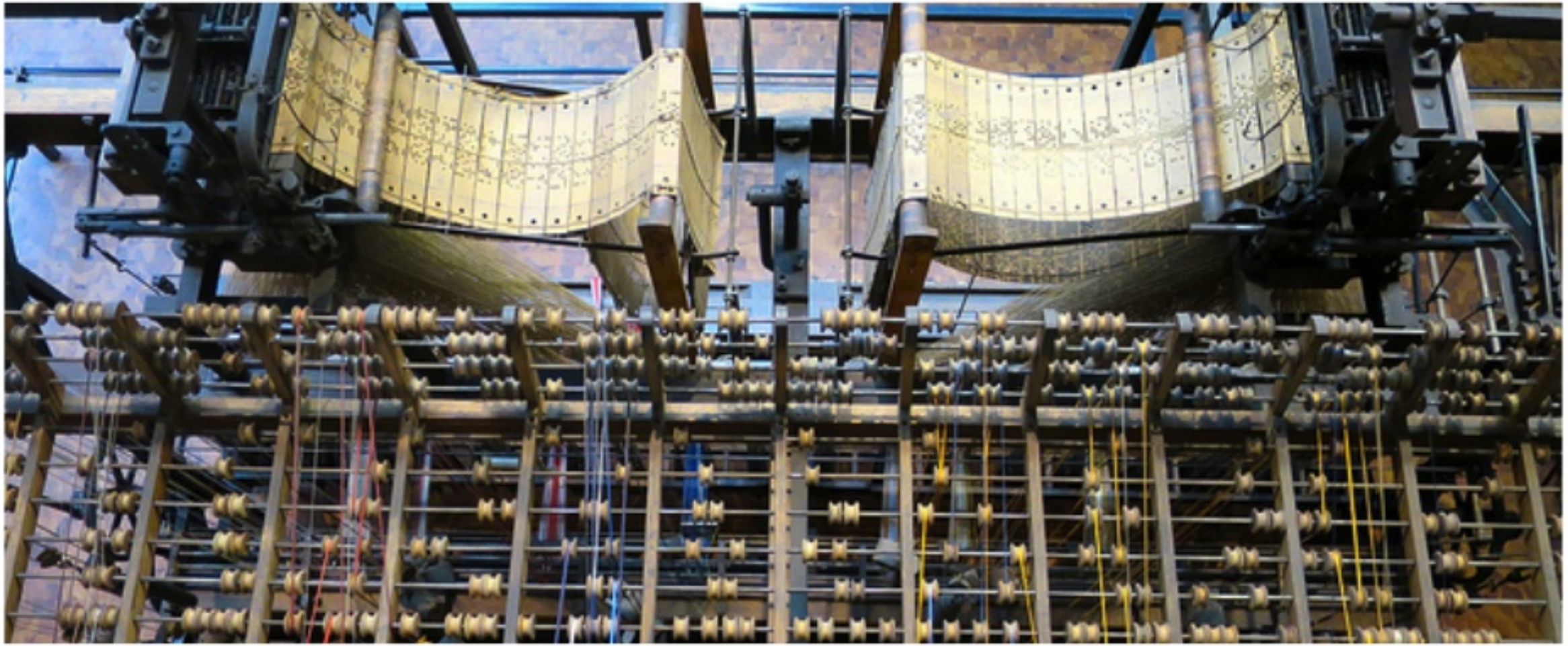
Writes under the hood

```
MongoSpark.save(dataFrame)
```

1. Create a connector
2. For each partition:
    1. Group the data in batches
    2. Insert into the collection
        * DataFrames / Datasets will upsert if there is an `_id`


LEVEL 1 COMPLETE!
GET READY FOR THE NEXT LEVEL

4. Structured data



https://www.flickr.com/photos/fabola/15524427452

Why support structured data?

- RDD's are the core to Spark
  You can convert RDDs to Datasets – but the API can be painful for users
- Fastest growing area of Spark
  40% of users use DataFrames in production & 67% in prototypes*
- You can provide Python and R support
  62% of users use Spark with Python, behind Scala but gaining fast*
- Performance improvements
  Can passing filters and projections down to the data layer

* Figures from the Spark 2016 survey

Structured data in Spark

- DataFrame == Dataset[Row]
  - RDD[Row]
    - Represents a row of data
    - Optionally define the schema of the data
- Dataset
  - Efficiently decode and encode data

Create a DefaultSource

- DataSourceRegister
  Provide a shortname for the datasource (broken? not sure how its used)
- RelationProvider
  Produce relations for your data source – inferring the schema (read)
- SchemaRelationProvider
  Produce relations for your data source using the provided schema (read)
- CreatableRelationProvider
  Creates a relation based on the contents of the given DataFrame (write)

DefaultSource continued…

- StreamSourceProvider (experimental)
  Produce a streaming source of data
  Not implemented - In theory could tail the OpLog or a capped collection as a source of data.
- StreamSinkProvider (experimental)
  Produce a streaming sink for data

Inferring Schema

- Provide it via a Case Class or Java bean
  Uses Sparks reflection to provide the Schema
- Alternatively Sample the collection
  - Sample 1000 documents by default
  - Convert each document to a StructType
    For unsupported Bson Types we use extended json format
    Would like support for User Defined Types
  - Use treeAggregate to find the compatible types
    Uses TypeCoercion.*findTightestCommonTypeOfTwo* to coerce types
    Conflicting types log a warning and downgrade to StringType

Create a BaseRelation

- TableScan
  Return all the data
- PrunedScan
  Return all the data but only the selected columns
- PrunedFilteredScan
  Returns filtered data and the selected columns
- CatalystScan
  Experimental access to logical execution plan
- InsertableRelation
  Allows data to be inserted via the INSERT INTO

Multi-language support!

```
// Scala
sparkSession.read.format("com.mongodb.spark.sql").load()

// Python
sqlContext.read.format("com.mongodb.spark.sql").load()

// R
read.df(sqlContext, source = "com.mongodb.spark.sql")
```

Rolling your own Connector

- Review existing connectors https://github.com/mongodb/mongo-spark
- Partitioning maybe tricky but is core for parallelism.
- `DefaultSource` opens the door to other languages
- Schema inference is painful for Document databases and users
- Structured data is the future for Speedy Spark

Free, online training

http://university.mongodb.com

THANK YOU.

Any questions come see us at the booth!