Broadcast Variables

Wednesday, May 10, 2017 4:49 PM

- Spark is fast, but we can do few things to make if faster.
- One of the optimization is to use shared variables.

Shared Variables in Spark

Used to optimize your code performance

- Broadcast variables
- Accumulator variables

ⓘ Shared variables are purely for optimization and efficiency.
You only need to use shared variables explicitly when a use case is right.

- There are two types of shared variables
  - Broadcast
  - Accumulator.

Broadcast Variables

Definition and use cases

Definition

```
Broadcast variables are read-only variables whose values are
broadcasted and cached on each node in a cluster and are
visible to all tasks of your application
```

Main use cases

- Application tasks across multiple stages need the same, relatively large and immutable dataset
- Application tasks need the same, relatively large and immutable dataset cached in deserialized form

- The broadcast variables are **readonly** with respect to the computation going on (task) and not with respect to the driver, driver can always change the value of broadcast variables.

Broadcast Variables

API

Statement	Description
val *broadcastVar* = *sc.broadcast(object)*	Declare and initialize broadcast variable *broadcastVar* with the *object* value of any type using method *broadcast* of the Spark Context object *sc*.
broadcastVar.value	Access *value* of broadcast variable *broadcastVar* in a Spark operation, such as *map*, *filter*, or *reduce*.

Example

DATASTAX ACADEMY

Not using broadcast variables

```
val popularTitles = Set("Alice in Wonderland",
                        "Alice Through the Looking Glass", "...")

val movies = sc.cassandraTable("killr_video","movies")
               .select("title","release_year","rating","genres")
               .cache

movies.filter(row => popularTitles contains row.getString("title"))
      .saveToCassandra("killr_video", "favorite_movies",
                       SomeColumns("title","release_year","rating","genres"))

movies.filter(row => !(popularTitles contains row.getString("title")))
      .saveToCassandra("killr_video", "other_movies",
                       SomeColumns("title","release_year","rating","genres"))
```

movies	
movie_id	UUID K
title	TEXT
release_year	INT
{genres}	SET<TEXT>
rating	FLOAT

- Here we are usnig **popularTitles** variables and checking if each 'title' element ins present in **popularTitles**, here we are making **popularTitles** at each node.
- We can do better by broadcasting **popularTitles** to all nodes so we do not need to make it at many places(as it is readonly we need not to worry about consistency.

Example

Using broadcast variables

```
val popularTitles = sc.broadcast(Set("Alice in Wonderland",
                                     "Alice Through the Looking Glass", "..."))

val movies = sc.cassandraTable("killr_video","movies")
               .select("title","release_year","rating","genres")
               .cache

movies.filter(row => popularTitles.value contains row.getString("title"))
      .saveToCassandra("killr_video", "favorite_movies",
                       SomeColumns("title","release_year","rating","genres"))

movies.filter(row => !(popularTitles.value contains row.getString("title")))
      .saveToCassandra("killr_video", "other_movies",
                       SomeColumns("title","release_year","rating","genres"))
```

- We just need to broadcast it while creating, it will automatically be available to all nodes.(no other change)

Accumulator Variables

Wednesday, May 10, 2017 5:33 PM

- Accumulator variables are way of taking a **mutable** value and allow many workers to work on that at a time and bring it as a single value by accumulating all results.
- The data will be accumulated back to the drivers.

- API for **Accumulator** is exactly same as the **Broadcast** variables.

Accumulator Variables

DATASTAX ACADEMY

API

Statement	Description
val *accumulatorVar* = *sc.accumulator*(*initialValue*)	Declare and initialize accumulator variable *accumulatorVar* with *initialValue* of a numeric type using method *accumulator* of the Spark Context object *sc*.
accumulatorVar += 1	Update *accumulatorVar* in a Spark action, such as *foreach*.
accumulatorVar.value	Access final aggregated *value* of accumulator variable *accumulatorVar* in a client program.

Example

DATASTAX ACADEMY

Not using accumulator variables

```
val movieRatings = sc.cassandraTable("killr_video","favorite_movies")
                     .select("rating")
                     .filter(row => row.getFloatOption("rating").isDefined)
                     .map(row => row.getFloat("rating"))
                     .cache

val numRatings   = movieRatings.count
val sumRatings   = movieRatings.sum
val avgRating    = sumRatings / numRatings

println(f"$avgRating%1.1f")
```

favorite_movies		
title	TEXT	K
release_year	INT	K
rating	FLOAT	
{genres}	SET<TEXT>	
details	details_type	

details_type	
country	TEXT
language	TEXT
runtime	INT

- By using **cache()** method we are going to persist the RDD in memory(learn in RDD persistent)
-

Using accumulator variables

```
val numRatings = sc.accumulator(0)
val sumRatings = sc.accumulator(0.0)

sc.cassandraTable("killr_video","favorite_movies")
  .select("rating")
  .filter(row => row.getFloatOption("rating").isDefined)
  .foreach{row => numRatings += 1; sumRatings += row.getFloat("rating")}

val avgRating = sumRatings.value / numRatings.value

println(f"$avgRating%1.1f")
```

favorite_movies		
title	TEXT	K
release_year	INT	K
rating	FLOAT	
{genres}	SET<TEXT>	
details	details_type	

details_type	
country	TEXT
language	TEXT
runtime	INT

RDD persistent

Wednesday, May 10, 2017 6:07 PM

- This is the third optimization technique.

The Challenge: Suboptimal Code

DATASTAX ACADEMY

Computing percentages of comedy movies released in 2014 and 2013

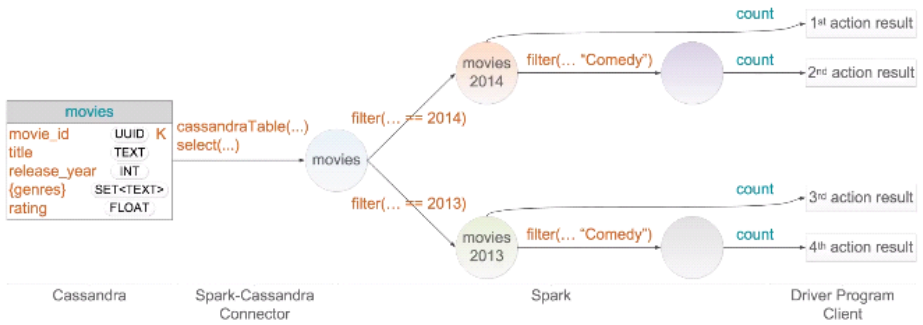
```
val movies = sc.cassandraTable("killr_video","movies")
               .select("release_year","genres")

val movies2014 = movies.filter(row => row.getInt("release_year") == 2014)
val total2014  = movies2014.count
val comedy2014 = movies2014.filter(row => row.getSet[String]("genres")
                                          contains "Comedy").count
val percentage2014 = 100.0 * comedy2014 / total2014

val movies2013 = movies.filter(row => row.getInt("release_year") == 2013)
val total2013  = movies2013.count
val comedy2013 = movies2013.filter(row => row.getSet[String]("genres")
                                          contains "Comedy").count
val percentage2013 = 100.0 * comedy2013 / total2013
```

movies	
movie_id	UUID K
title	TEXT
release_year	INT
{genres}	SET<TEXT>
rating	FLOAT

- The code works but it is not very efficient, if we look at the DAG of the operations.

DAG of Operations

DATASTAX ACADEMY



- this is the ideal graph, which do not happen in real life, in Real life the **movie** get computed 4 times, **movie2014** get computed twice.
- This is because the movies, get deleted after it has done its work , and again we have to calculate that.

Stages of Computation

DATASTAX ACADEMY

Stages 1, 2, 3, and 4

The Challenge: Summary

Efficiency issues:

- Reading the same data from Cassandra into RDD *movies* four times
- Recomputing RDD *movies2014* twice
- Recomputing RDD *movies2013* twice

Take aways:

- Recomputing an RDD multiple times due to multiple actions on the RDD or its derivatives
- Need a way to materialize and reuse an RDD after it is computed for the first time

- We can persist a **RDD .** so it will be automatically saved into cached in memory when computed, and as we know if memory becomes full it will be kept in disk by the spark, but spark will always try to keep it inside memory.
- After persisting the RDD can be used by others if they wanted that.

RDD Persistence

One of the most important optimizations in Spark for iterative algorithms and interactive computation!

- You can instruct Spark to cache or persist any RDD in your program
- Persisted RDD is kept in memory (by default) once it is computed for the first time
- Persisted RDD is reused by other operations that require the same dataset

The persistence mechanism should be used to avoid recomputing the same dataset multiple times. With respect to a DAG of operations, any RDD that is a common ancestor of two or more leaf nodes resulting from actions is a good candidate for the persistence optimization.

RDD Persistence API

Any RDD, including a Cassandra RDD, can be cached or persisted

Transformation	Description
persist([*storageLevel*])	Persists the source RDD according a storage level specified by the optional parameter. The default storage level is *org.apache.spark.storage.StorageLevel.MEMORY_ONLY*, which prescribes persisting RDD elements as deserialized Java objects in the JVM.
cache()	Same as *persist*() or *persist*(*StorageLevel.MEMORY_ONLY*).
unpersist()	Unpersists the source RDD (manually). It is safe to not use this transformation because Spark automatically monitors and unpersists least-recently-used RDD partitions.

When using *cache*() or *persist*(), if an RDD does not fit into memory, some partitions will not be cached and will be recomputed on the fly when needed.

04:24

Storage Levels

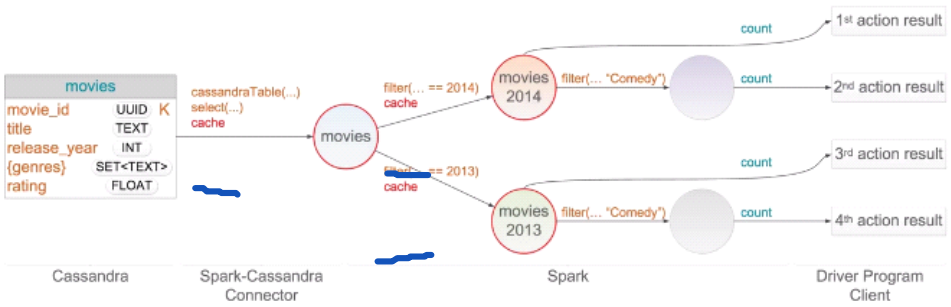
Storage Level	Description
MEMORY_ONLY *MEMORY_ONLY_SER*	Persisting RDD elements as *deserialized* or_serialized_ Java objects in the JVM, respectively. Partitions that do not fit into memory are not cached and recomputed when needed. *MEMORY_ONLY_SER* is more space-efficient but more CPU-intensive than *MEMORY_ONLY*.
MEMORY_AND_DISK *MEMORY_AND_DISK_SER*	Persisting RDD elements as *deserialized* or *serialized* Java objects in the JVM, respectively. Partitions that do not fit into memory are spilled to disk.
DISK_ONLY	Persisting RDD partitions on disk.
MEMORY_ONLY_2, *MEMORY_AND_DISK_2*, …	Same as the respective storage levels above, but with replication on two nodes in a cluster.
OFF_HEAP (experimental)	Persisting the source RDD in *serialized* format in *Tachyon* (a memory-centric distributed storage system)

04:42 06:40

- We just need to add **cache()** method, to make it persistent.

Solving the Challenge

Caching RDDs *movies2014* and *movies2013*



Final Solution

```
val movies = sc.cassandraTable("killr_video","movies")
               .select("release_year","genres")
               .cache

val movies2014 = movies.filter(row => row.getInt("release_year") == 2014)
                       .cache
val total2014  = movies2014.count
val comedy2014 = movies2014.filter(row => row.getSet[String]("genres")
                                          contains "Comedy").count
val percentage2014 = 100.0 * comedy2014 / total2014

val movies2013 = movies.filter(row => row.getInt("release_year") == 2013)
                       .cache
val total2013  = movies2013.count
val comedy2013 = movies2013.filter(row => row.getSet[String]("genres")
                                          contains "Comedy").count
val percentage2013 = 100.0 * comedy2013 / total2013
```

movies	
movie_id	UUID K
title	TEXT
release_year	INT
{genres}	SET<TEXT>
rating	FLOAT

Introduction to Pair RDDs

Wednesday, May 10, 2017 6:37 PM

- There may be many formate of Data in row, but he key-value records seems to be most important.
- Key/value pair RDD emerges as common pattern,
  so Special API has been given for dealing with Key/Value RDDs.

What is a Key-Value Pair RDD?

- Any RDD whose elements are key-value pairs
  - Key-value pair is a tuple with two components: (*key*, *value*)
  - Different pairs may have the same keys
  - Both keys and values can be of primitive or complex data types
- Examples:

```
val users = sc.parallelize(List( ("Alice",21), ("Bob",12), ("Bob",18) ))
// users: org.apache.spark.rdd.RDD[(String, Int)]

val movies = sc.cassandraTable("killr_video","movies")
               .keyBy( row => (row.getString("title"), row.getInt("release_year")) )
// movies: org.apache.spark.rdd.RDD[((String, Int),
com.datastax.spark.connector.CassandraRow)]
```

Operations on Key-Value Pair RDDs

Key-value pair semantics enables a number of important operations

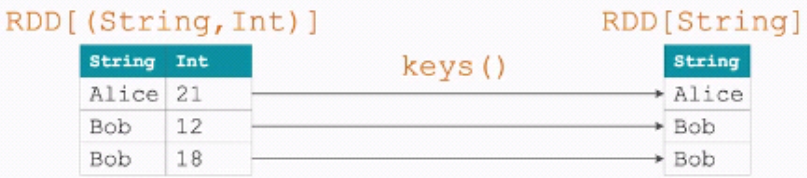
- Key-based operations
  - Aggregation
  - Grouping and sorting
  - Inner and outer joins
  - Union, intersection, difference
  - Other "supporting" operations

ⓘ Pair RDDs also support operations defined for generic RDDs, such as *filter*, *map*, *count*, and so forth.

"Supporting" Transformations for Pair RDDs

Transformation	Description
keys()	A new RDD is formed by keys of the source RDD.
values()	A new RDD is formed by values of the source RDD.

ⓘ Both *keys*() and *values*() can be easily implemented using *map*(*f*).

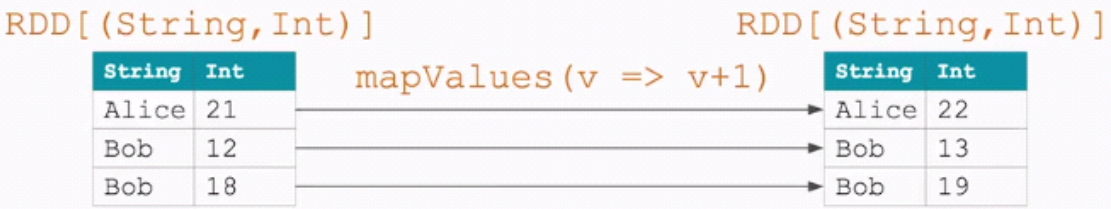


NOTE - We can implement **key/value** pair pattern with **map** also.

"Supporting" Transformations for Pair RDDs

Transformation	Description
mapValues(*f*)	A new RDD is formed by applying a function *f* on each value of the source RDD. Keys are retained without changes, which implies that any key-based partitioning of the source is also retained. There is a *one-to-one* correspondence between input and output elements.
flatMapValues(*f*)	Same as above except there is a *one-to-many* correspondence between input and output elements if *f* returns a *Seq* with more than one element.

⚠ It is a bad idea to implement these transformations using *map*(*f*).



"Supporting" Actions for Pair RDDs

Transformation	Description
lookup(*key*)	Returns a *Seq* of values in the source RDD for a given key.
collectAsMap()	Returns a *Map* of key-value pairs in the source RDD.

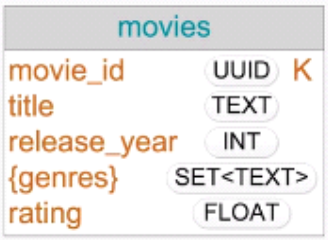
Only use *collectAsMap()* when an RDD does not contain multiple pairs with the same key.

- **collectAsMap()** is same as **collect()** we have used with simple RDD's , the only difference is that, this method returns the values as map between key and value as the RDD is key/value pair RDD.

Example

Find ratings of movies released in 2014

```
sc.cassandraTable[(Int,Option[Float])]("killr_video","movies")
  .select("release_year","rating")
  .mapValues(v => v.getOrElse(0.0))
  .lookup(2014)
  .foreach(println)

// Sample output:
// 6.3
// 6.0
// 5.4
```



- Here we have fetched Pair RDD, from the table

Operations: Aggregation

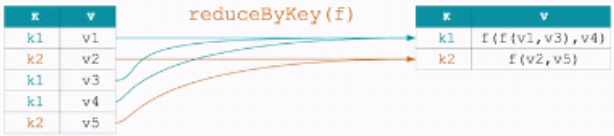
Thursday, May 11, 2017 10:25 AM

Computing Per-Key Aggregates

- Aggregating values with the same key is a common task
  - Statistical analysis, summarization
  - The *WordCount* problem is classic
- *KillrVideo* challenges for this presentation
  - Count how many movies featuring Johnny Depp were released per year
  - Find the highest rated movie featuring Johnny Depp for each year
  - Compute an average rating of movies featuring Johnny Depp for every year

Transformation *reduceByKey*

Transformation	Description
reduceByKey(*f*, [*numTasks*])	A new RDD of (*K*,*V*) pairs is formed by aggregating values for each key in the source RDD of (*K*,*V*) pairs. The reduce function *f*: *V* x *V* → *V* takes two values of type *V* and returns a new value of type *V*. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.



- Reduce by key just aggregates all the values according to the keys, so we will get unique keys after this.
- We can pass the function, which will tell what to do with the values which are of same key.

Count how many movies featuring Johnny Depp were released per year

```
sc.cassandraTable("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp'")
  .select("release_year")
  .as( (year:Int) => (year,1) )
  .reduceByKey(_ + _)
  .collect
  .foreach(println)

// Sample output:
// (2010,2)
// (2000,3)
// (2014,3)
```

movies_by_actor

actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

- FoldByKey is same as simple as simple 'fold' method only it takes initial value **zeroValue,** and then aggregate.

Transformation *foldByKey*

Transformation	Description
foldByKey(*zeroValue*, [*numTasks*])(*f*)	A new RDD of (*K*,*V*) pairs is formed by aggregating values for each key in the source RDD of (*K*,*V*) pairs. The *zeroValue* parameter is a neutral value, which can be "added" to the result an arbitrary number of times without affecting it (e.g., *Nil* for list concatenation, *0* for addition, or *1* for multiplication). The associative function *f*: *V x V* → *V* takes a partially aggregated result and a value from the source RDD and returns a new result. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.

Find the highest rated movie featuring Johnny Depp for each year

```
sc.cassandraTable("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp'")
  .select("release_year","title","rating")
  .as( (y:Int,t:String,r:Option[Float]) => (y,(t,r)) )
  .filter{case (y,(t,r)) => r.isDefined}
  .mapValues{case (t,r) => (t,r.get)}
  .foldByKey( ("",0.0f) ){ case ((maxT,maxR),(t,r)) =>
                             if (maxR < r) (t,r)
                             else (maxT,maxR) }
  .collect.foreach(println)

// Sample output:
// (2010,(Alice in Wonderland,6.5))
// (2000,(Before Night Falls,7.3))
// (2014,(Transcendence,6.3))
```

movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

05:03

- Here the zero value is ("",0.0f) because our values get (t,r) which are 'title' and 'rating' respectively.
  In the accumulator we have defined **maxT** and **maxR** which will accumulate the values.
- Here if **maxR<r** then we are going to update **(maxT,maxR)** by **(t,v)** and in this way we get our final values.

Compute an average rating of movies featuring Johnny Depp for every year

```
sc.cassandraTable[(Int,Option[Float])]("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp'").select("release_year","rating")
  .filter(_._2.isDefined).mapValues(r => r.get)
  .combineByKey(
   (rating:Float)                      =>(rating, 1),
   (res:(Float,Int),rating:Float)      =>(res._1 + rating, res._2 + 1),
   (res1:(Float,Int),res2:(Float,Int))=>(res1._1 + res2._1, res1._2 + res2._2)
  )
  .mapValues{case (sum,count) => val avg = sum / count; f"$avg%1.1f"}
  .collect.foreach(println)

// Sample output:
// (2010,6.3)
// (2000,6.9)
// (2014,5.9)
```

movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

07:56

Action *countByKey*

Action *countByKey*

Action	Description
countByKey([*numTasks*])	Returns a *Map* of (*K*,*N*) pairs, where *N* is the number of elements for each key in the source RDD of (*K*,*V*) pairs.

Action *countByKey*

Count how many movies featuring Johnny Depp were released per year

```
sc.cassandraTable("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp'")
  .select("release_year")
  .as( (year:Int) => (year,1) )
  .countByKey
  .foreach(println)

// Sample output:
// (2010,2)
// (2000,3)
// (2014,3)
```

movies_by_actor

actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

Grouping and Sorting

Thursday, May 11, 2017 10:25 AM

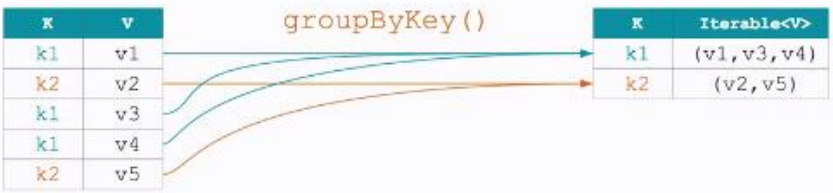
Key-Based Grouping and Sorting

Grouping values with the same key	Sorting values using keys
• Reorganizing data by a new key • Post-processing per-key groups	• Generating special-purpose datasets • Generating reports that require ordering

- *KillrVideo* challenges for this presentation
  - Output movies featuring Johnny Depp grouped by genre
  - Output movies with Johnny Depp and movies with Tom Hanks co-grouped by year
  - Output movies from 2010s featuring Johnny Depp ordered by rating

Transformation *groupByKey*

Transformation	Description
groupByKey([*numTasks*])	A new RDD of (*K*,*Iterable<V>*) pairs is formed by grouping values for each key in the source RDD of (*K*,*V*) pairs. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.

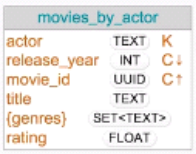


- We will get iterable sequence in output corresponding to each key.

Transformation *groupByKey*

Output movies featuring Johnny Depp grouped by genre

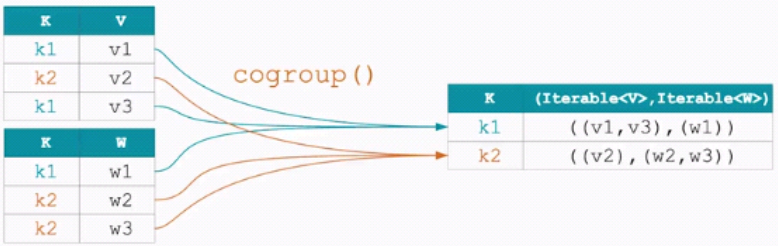
```
sc.cassandraTable[(String,Int,Set[String])]("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp'")
  .select("title","release_year","genres")
  .flatMap{case (t,y,gs) => gs.map( g =>(g, t + ", " + y) )}
  .groupByKey()
  .collect
  .foreach(println)

// Sample output for one group:
// (Family,CompactBuffer(
//         Alice Through the Looking Glass, 2016,
//         Alice in Wonderland, 2010,
//         Charlie and the Chocolate Factory, 2005,
//         Finding Neverland, 2004))
```



- here have made "genre" as key because we want to group by that genre.
- After using **flatMap** we get something like
  <genre>=>"<title>,<year>" as pair(string,string) RDD

Transformations *cogroup* and *groupWith*

Transformation	Description
cogroup(*otherRDD*, [*numTasks*]) or **groupWith**(*otherRDD*, [*numTasks*])	A new RDD of (*K*,(*Iterable<V>*, *Iterable<W>*)) pairs is formed by grouping values for each key from the source RDD of (*K*,*V*) pairs and the *otherRDD* of (*K*,*W*) pairs. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation. Both *cogroup* and *groupWith* refer to the same transformation.



- **Cogroup** combines two RDD according to the keys and form two iterables, 1 for each RDD.

Output movies with Johnny Depp and movies with Tom Hanks co-grouped by year

```
val johnnyMovies = sc.cassandraTable("killr_video","movies_by_actor")
                     .where("actor = 'Johnny Depp'")
                     .keyBy(row => row.getInt("release_year"))
val tomMovies = sc.cassandraTable("killr_video","movies_by_actor")
                     .where("actor = 'Tom Hanks'")
                     .keyBy(row => row.getInt("release_year"))
johnnyMovies.cogroup(tomMovies)
            .collect.foreach(println)

// Sample output for one group:
// (2010,CompactBuffer(
//         CassandraRow{actor: Johnny Depp, ..., title: The Tourist},
//         CassandraRow{actor: Johnny Depp, ..., title: Alice in Wonderland})
//       CompactBuffer(
//         CassandraRow{actor: Tom Hanks, ..., title: Toy Story 3}))
```

movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

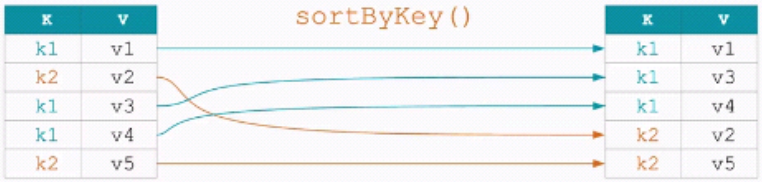
- **KeyBy** will convert the row into key,value pair.

keyBy[*KeyType*] (*columns*)	Optionally used with *cassandraTable*[*ValueType*](...) to convert Cassandra rows to pairs of objects, where a pair key is of type *KeyType* and a pair value is of type *ValueType*. *KeyType* and *ValueType* are usually defined as Scala tuples and/or case classes.

- Here we are converting the row into pair RDD ,by key **release_year.**

Transformation *sortByKey*

Transformation	Description
sortByKey([*ascending*], [*numTasks*])	A new RDD of (*K*,*V*) pairs is formed by sorting pairs in the source RDD of (*K*,*V*) pairs based on keys in ascending (default) or descending order. The *K* type must implement trait *Ordered*. The optional *ascending* parameter has the default value of *true*. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.



- **Ascending** paramter is by default **true,** if we wanted to make false we can pass **false.**

Output movies from 2010s featuring Johnny Depp ordered by rating

```
sc.cassandraTable("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp' AND release_year > 2010")
  .select("title","release_year","rating")
  .as( (t:String, y:Int, r:Option[Float]) => (r.getOrElse(0.0f),(t,y)) )
  .sortByKey(false)
  .collect
  .foreach(println)

// Sample output:
// (7.3,(Rango,2011))
// (6.7,(Pirates of the Caribbean: On Stranger Tides,2011))
// (6.5,(The Lone Ranger,2013))
// (6.3,(Transcendence,2014))
```

movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

- We are here sorting in descending order.

Grouping and Sorting Best Practices

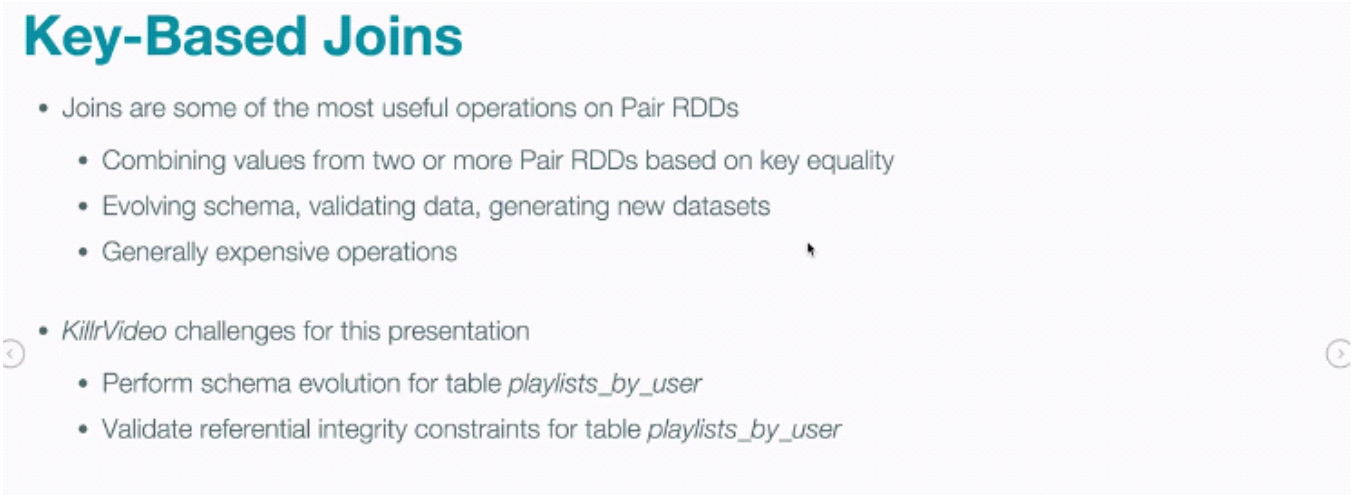
Grouping	Sorting
• Expensive on large datasets • Do not use grouping for aggregation or joins • Grouping transformations may result in large key-value pairs	• Expensive on large datasets • Prefer smaller datasets • Prefer Cassandra clustering column ordering to sorting

- Always aware of the fact that group and sorting on large dataset can fatal.

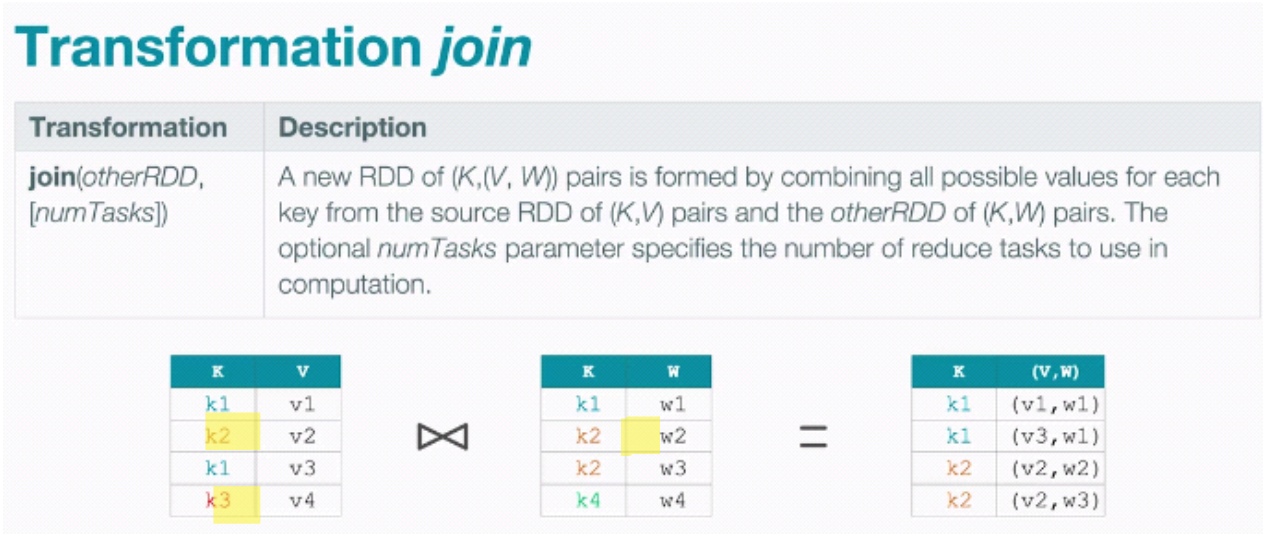
Joins

Thursday, May 11, 2017 12:51 PM

- Joins can be expensive operation but can be useful but keep aware always.



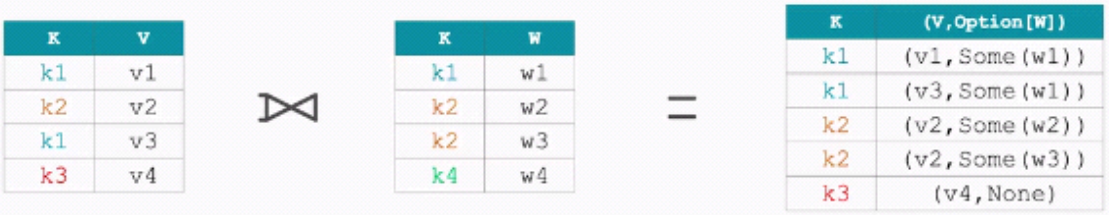
- **Join** is same as Relational DB which means Inner Join(normal join).



Transformation *leftOuterJoin*

Transformation	Description
leftOuterJoin(*otherRDD*, [*numTasks*])	A new RDD of (*K*,(*V*, *Option*[*W*])) pairs is formed by combining values for each key from the source RDD of (*K*,*V*) pairs and the *otherRDD* of (*K*,*W*) pairs as: • (*K*,(*V*, *Some*[*W*])) pairs if a key exists in both RDDs; • (*K*,(*V*, *None*)) pairs if a key only exists in the source RDD. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.

- Here not common come, so there may be possibility that right table do not have that key, in this case we will get **None**, otherwise we will get object **Some()** having all values.



Transformation *rightOuterJoin*

Transformation	Description
rightOuterJoin(*otherRDD*, [*numTasks*])	A new RDD of (*K*,(*Option*[*V*],*W*)) pairs is formed by combining values for each key from the source RDD of (*K*,*V*) pairs and the *otherRDD* of (*K*,*W*) pairs as: • (*K*,(*Some*[*V*], *W*)) pairs if a key exists in both RDDs; • (*K*,(*None*, *W*)) pairs if a key only exists in *otherRDD*. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.

Transformation *fullOuterJoin*

Transformation	Description
fullOuterJoin(*otherRDD*, [*numTasks*])	A new RDD of (*K*,(*Option*[*V*],*Option*[*W*])) pairs is formed by combining values for each key from the source RDD of (*K*,*V*) pairs and the *otherRDD* of (*K*,*W*) pairs as: • (*K*,(*Some*[*V*], *Some*[*W*])) pairs if a key exists in both RDDs; • (*K*,(*Some*[*V*], *None*)) pairs if a key only exists in the source RDD; • (*K*,(*None*, *Some*[*W*])) pairs if a key only exists in *otherRDD*. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.

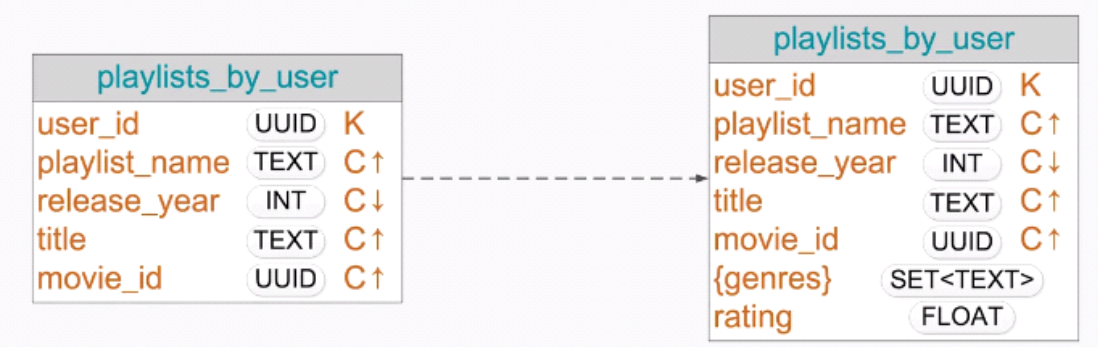
- Every join will always happen with the key.

Challenge 1: Schema Evolution

Step 1: Adding two new columns to *playlists_by_user*

```
ALTER TABLE playlists_by_user ADD genres SET<TEXT>;

ALTER TABLE playlists_by_user ADD rating FLOAT;
```

- Let we wanted to add two columns, genres and ratings.



- Let see code doing the schema evolution.
  (in created new columns it is just filling values from **movies** table.

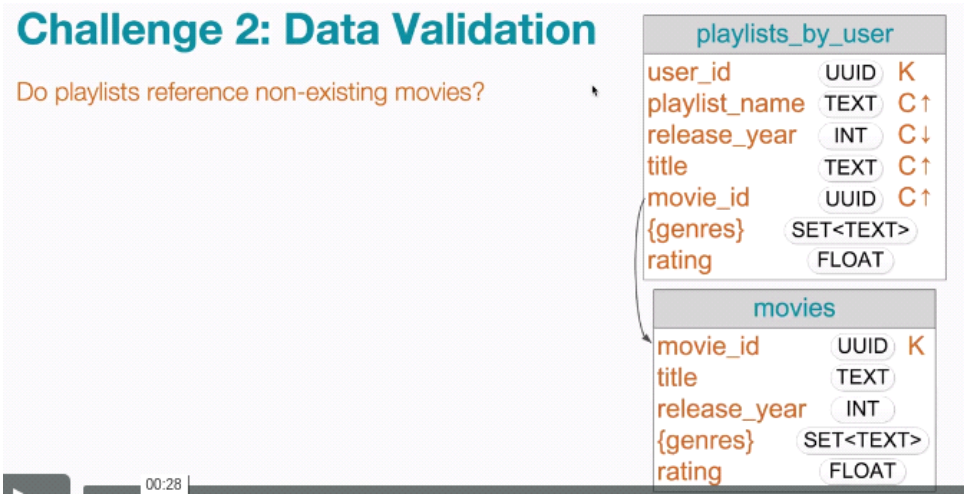
```
val playlists =
    sc.cassandraTable("killr_video","playlists_by_user")
      .select("user_id","playlist_name","release_year","title","movie_id")
      .as((u:java.util.UUID,p:String,y:Int,t:String,m:java.util.UUID) =>
          (m,(u,p,y,t)))

val movies =
    sc.cassandraTable("killr_video","movies")
      .select("movie_id","genres","rating")
      .as((m:java.util.UUID,g:Set[String],r:Option[Float]) =>
          (m,(g,r)))

playlists.join(movies)
         .map{case (m,((u,p,y,t),(g,r))) => (u,p,y,t,m,g,r)}
         .saveToCassandra("killr_video","playlists_by_user")
```

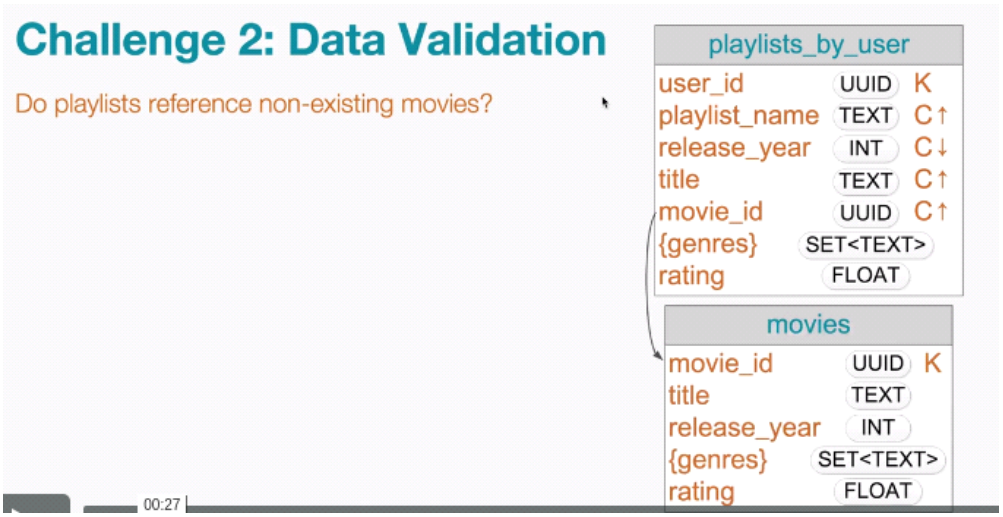
- Here we are taking data from both tables and then making movie as the key of both.
- Then join both tables, so that we can get **movie_id** and **genres** columns also.

Set operations

Thursday, May 11, 2017 1:18 PM

- Cassandra do not provide **refrencial integrity check** by default, so the database will not tell us, if we are pointing to wrong thing on other table and it will not cause error when the foreign key of some value get deleted.

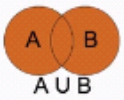
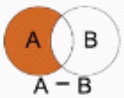




- Here movie_id of 'movies' table is foreign key to playlist_by_user table.
- Now we wanted to check on the movie we are referencing exist or not.

```
val playlists =
    sc.cassandraTable("killr_video","playlists_by_user")
      .keyBy(row => row.getUUID("movie_id"))

val movies =
    sc.cassandraTable("killr_video","movies")
      .select("movie_id")
      .keyBy(row => row.getUUID("movie_id"))

playlists.leftOuterJoin(movies)
         .filter{case (m,(rowP,rowM)) => !rowM.isDefined}
         .map{case (m,(rowP,rowM)) => rowP}
         .collect.foreach(println)

// Sample output:
// CassandraRow{user_id: 709e42f0-5f25-4551-9d85-6e3ad39d6cde,
//              playlist_name: Pirate Movies,
//              release_year: 2017,
//              title: Pirates of DataStax, ...}
```

Union, Intersection, and Difference

Operation	Venn Diagram	Generic RDD API	Key-Value Pair RDD API
Union		• *union(otherRDD)* • Duplicates: Yes	• N/A • Can be implemented
Intersection		• *intersection(otherRDD)* • Duplicates: No	• N/A • Can be implemented
Difference		• *subtract(otherRDD)* • Duplicates: Yes	• *subtractByKey(otherRDD)* • Duplicates: Yes

02:35

- Everything operation is done on the basis of key.
- As we can see key based union and intersection are not available, but we can implement them as follows.

Key-Based Union

Sample implementation for *union-compatible* Pair RDDs

```
val A = sc.parallelize(Array(("k1","v1"), ("k2","v2"), ("k1","v3"), ("k3","v4")))
val B = sc.parallelize(Array(("k1","w1"), ("k2","w2"), ("k2","w3"), ("k4","w4")))

A.union(B)
```

K	V
k1	v1
k2	v2
k1	v3
k3	v4

∪

K	V
k1	w1
k2	w2
k2	w3
k4	w4

=

K	V
k1	v1
k2	v2
k1	v3
k3	v4
k1	w1
k2	w2
k2	w3
k4	w4

Key-Based Intersection

Sample implementation for *union-compatible* Pair RDDs

```
val A = sc.parallelize(Array(("k1","v1"), ("k2","v2"), ("k1","v3"), ("k3","v4")))
val B = sc.parallelize(Array(("k1","w1"), ("k2","w2"), ("k2","w3"), ("k4","w4")))

A.groupByKey
 .join(B.groupByKey)
 .flatMapValues{case (aList,bList) => aList ++ bList}
```

K	V
k1	v1
k2	v2
k1	v3
k3	v4

∩

K	V
k1	w1
k2	w2
k2	w3
k4	w4

=

K	V
k1	v1
k1	v3
k1	w1
k2	v2
k2	w2
k2	w3

- For intersection, we have to do it manually.
- Here we continated the list using **++** operator , we can also use **:::** operator.

```
scala> List(1,2,3) ++ List(4,5)
res0: List[Int] = List(1, 2, 3, 4, 5)

scala> List(1,2,3) ::: List(4,5)
res1: List[Int] = List(1, 2, 3, 4, 5)
```

- As we know **substractByKey** is already available, so we can use it right away.

Key-Based Difference

Transformation	Description
subtractByKey(*otherRDD*, [*numTasks*])	A new RDD of (*K*,*V*) pairs is formed by those pairs from the source RDD of (*K*,*V*) pairs whose keys are not present in the *otherRDD* of (*K*,*W*) pairs. The optional *numTasks* parameter specifies the number of reduce tasks to use in computation.

- As we know , everything is done according to **key ,** so the key which is in A but not in B will be there in output.

```
val playlists =
    sc.cassandraTable("killr_video","playlists_by_user")
      .keyBy(row => row.getUUID("movie_id"))

val movies =
    sc.cassandraTable("killr_video","movies")
      .select("movie_id")
      .keyBy(row => row.getUUID("movie_id"))

playlists.subtractByKey(movies)
         .collect.foreach(println)

// Sample output:
// CassandraRow{user_id: 709e42f0-5f25-4551-9d85-6e3ad39d6cde,
//              playlist_name: Pirate Movies,
//              release_year: 2017,
//              title: Pirates of DataStax, ...}
```

Understanding Partitioning

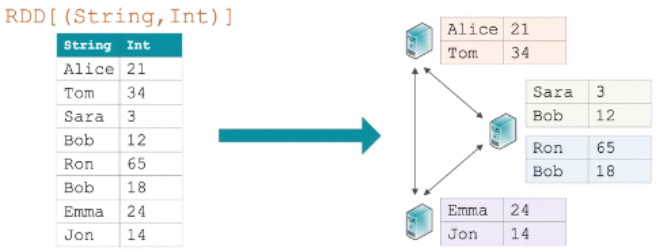
Thursday, May 11, 2017 2:19 PM

- Controlling Partitioning is one of the most important optimization in spark.

An RDD is a Distributed Collection of Partitions

A C A D E

- Spark automatically partitions RDDs
- Spark automatically distributes partitions among nodes



- Spark automatically partitions RDD on different nodes, but there is an API to expose the internal partioning scenarios(how they are actual partitioned).

RDD Partitioning Properties

A C A D E M Y

Number of partitions

Property	Description
partitions	Returns an *Array* with all partition references for the source RDD.
partitions.size	Returns a number of partitions in the source RDD.

```
val movies = sc.parallelize(List(("Alice in Wonderland",2016), ("Alice Through the Looking
Glass",2010), ...))
println(movies.partitions.size)
// Sample output: 3

val interactions = sc.cassandraTable("killr_video","video_interactions_by_user")
println(interactions.partitions.size)
// Sample output: 4
```

00:56

Partitioner

Property	Description
partitioner	Returns an *Option*[*Partitioner*] for the source RDD, where *Partitioner*, if any, can refer to *HashPartitioner*, *RangePartitioner*, or a custom partitioner.

```
val movies = sc.parallelize(List(("Alice in Wonderland",2016), ("Alice Through the Looking Glass",2010), ...))
println(movies.partitioner)
// Sample output: None

val moviesByYear = movies.map{case (t,y) => (y,t)}.groupByKey
println(moviesByYear.partitioner)
// Sample output: Some(org.apache.spark.HashPartitioner@3)
```

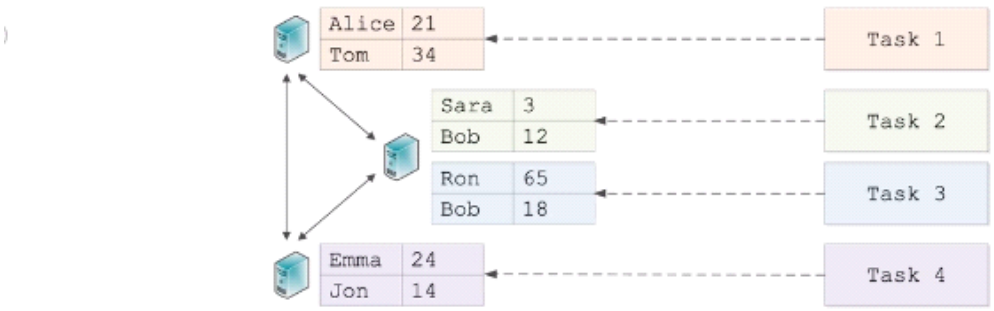
- HashPartitioner – used mostly for key based operation
- RangePartitioner – used when sorting by key required
- Custompartitioner - we can configure according to our need.

Factors That Affect Partitioning

- Resources available to an application
- External data sources, such as local collections
- Transformations used to derive an RDD
- Partitioning properties of parent RDD(s)
- Transformation from which that RDD is generated may also decide which partitioning to use.

Partitioning and Computation

- Partition is the smallest unit of data
- Task is the smallest unit of computation
- *Number of partitions = Number of tasks*



- We know **job** hasmany **Stages** , **stage** hasmany **tasks**.
- Inside a stage there will always be 1 partition per task.

Partitioning and Default Parallelism

Partitioning and Default Parallelism

- Default level of parallelism refers to a number of tasks that can be executed concurrently
  - Defined as a number of cores allocated to an application in a cluster
  - General recommendation: *Number of partitions >= Default Parallelism*

Property	Description
defaultParallelism	Returns a default level of parallelism of a *SparkContext* object.

```
println(sc.defaultParallelism)
// Sample output: 3

val movies = sc.parallelize(List(("Alice in Wonderland",2016), ("Alice Through the Looking Glass",2010), ...))
println(movies.partitions.size)
// Sample output: 3
```

Partitioning Rules

Thursday, May 11, 2017 2:35 PM

- Spark provides different type of default behaviour in different scenarios.

Default Partitioning Behavior

Scenarios we will talk about

- Creating an RDD from an external data source
  - Local Scala collection
  - Cassandra table
  - HDFS/CFS file
- Transforming an RDD into a new RDD
  - Generic transformations
  - Key-based transformations

Parallelizing a Scala Collection

A C A D E M Y

Default partitioning properties

API Call	Resulting RDD Partitioning Properties	
	partitions.size	**partitioner**
sc.parallelize(...)	*sc.defaultParallelism*	*None*

```
println(sc.defaultParallelism)
// Sample output: 3

val movies = sc.parallelize(List(("Alice in Wonderland",2016), ("Alice Through the Looking
Glass",2010), ...))
println(movies.partitions.size)
// Sample output: 3

println(movies.partitioner)
// Sample output: None
```

Retrieving Data From Cassandra

ACADEMY

Default partitioning properties

API Call	Resulting RDD Partitioning Properties	
	partitions.size	partitioner
sc.cassandraTable(...)	*sc.defaultParallelism* or approximate-data-size / 64MBs, whichever is greater	*None*

```
println(sc.defaultParallelism)
// Sample output: 3

val interactions = sc.cassandraTable("killr_video","video_interactions_by_user")
println(interactions.partitions.size)
// Sample output: 4

println(interactions.partitioner)
// Sample output: None
```

Reading Data From an HDFS/CFS File

ACADEMY

Default partitioning properties

API Call	Resulting RDD Partitioning Properties	
	partitions.size	partitioner
sc.textFile(...)	*sc.defaultParallelism* or a number of file blocks, whichever is greater	*None*

```
println(sc.defaultParallelism)
// Sample output: 3

val records = sc.textFile("cfs:///tmp/videos.csv")
println(records.partitions.size)
// Sample output: 3

println(records.partitioner)
// Sample output: None
```

Generic Transformations

Default partitioning properties

Transformation	Resulting RDD Partitioning Properties	
	partitions.size	**partitioner**
filter(…), *map*(…), *flatMap*(…), *distinct*(), …	The same number of partitions as in the parent RDD	*None*, except *filter* preserves parent RDD's *partitioner*, if any
rdd.union(*otherRDD*)	*rdd.partitions.size* + *otherRDD.partitions.size*	
rdd.intersection(*otherRDD*)	*max*(*rdd.partitions.size*, *otherRDD.partitions.size*)	
rdd.subtract(*otherRDD*)	*rdd.partitions.size*	
rdd.cartesian(*otherRDD*)	*rdd.partitions.size * otherRDD.partitions.size*	

Generic Transformations

Example

```
val movies = sc.cassandraTable("killr_video","movies")
// Partitioning properties: 3, None

val favoriteMovies = sc.cassandraTable("killr_video","favorite_movies")
// Partitioning properties: 3, None

val allMovies = movies.union(favoriteMovies)
// Partitioning properties: 6, None
```

Key-Based Transformations

A C A D E M Y

Default partitioning properties

Transformation	Resulting RDD Partitioning Properties	
	partitions.size	**partitioner**
reduceByKey(…), *foldByKey*(…), *combineByKey*(…), *groupByKey*(), …	The same number of partitions as in the parent RDD	*HashPartitioner*

sortByKey(…)		*RangePartitioner*
mapValues(…), *flatMapValues*(…)		Parent RDD's *partitioner*
cogroup(…), *join*(…), *leftOuterJoin*(…), *rightOuterJoin*(…), …	The same number of partitions as in the source RDD or the other RDD, depending on partitioning properties of the inputs.	*HashPartitioner*

- The number of partiions(partitions.size), do not change when doing all operation up to flatMapValues().

Key-Based Transformations

Example - unary transformations

```
val moviesByYear = sc.cassandraTable("killr_video","movies")
                     .keyBy(row => row.getInt("release_year"))
                     .groupByKey
// Partitioning properties: HashPartitioner@3

val moviesByYear2010 = moviesByYear.filter{case (y,rows) => y == 2010}
// Partitioning properties: HashPartitioner@3

val movies2010 = moviesByYear2010.flatMapValues(rows => rows)
// Partitioning properties: HashPartitioner@3

val capitalizedMovies2010 = movies2010.map{case (y,row) =>
(y,row.getString("title").capitalize)}
// Partitioning properties: 3, None
```

Default number of partitions

rdd.join(otherRDD), rdd.cogroup(otherRDD), …	**Partionner Can Be Reused**		**Resulting RDD partitions.size**
	rdd	**otherRDD**	
Case 1	No	No	*max(rdd.partitions.size, otherRDD.partitions.size)*
Case 2	Yes	No	*rdd.partitions.size*
Case 3	No	Yes	*otherRDD.partitions.size*
Case 4	Yes	Yes	*max(rdd.partitions.size, otherRDD.partitions.size)*

Binary Key-Based Transformations

Example - binary transformations

```
val users = sc.cassandraTable("killr_video","users")
              .keyBy(row => row.getInt("user_id"))
// Partitioning properties: 3, None

val interactions = sc.cassandraTable("killr_video","video_interactions_by_user")
                     .keyBy(row => row.getInt("user_id"))
                     .groupByKey
// Partitioning properties: HashPartitioner@4

val usersWithInteractions = users.join(interactions)
// Partitioning properties: HashPartitioner@4
```

Controlling Partitioning

Thursday, May 11, 2017 2:47 PM

Controlling Partitioning

One of the most important performance optimizations in Spark

- Number of partitions
  - Affects a number of tasks and the level of parallelism
  - *Goal*: balancing task execution and scheduling times
- Partitioner
  - Affects key-based operations
  - *Goal*: Avoiding shuffling the same dataset multiple times

How Many Partitions is Good?

General insights

- Too few partitions can be a problem
  - Less concurrency
  - Possible data skew
  - Increased memory pressure
  - Longer recovery from a failure
- Too many partitions can be a problem
  - Task scheduling may take longer than task execution
  - More lineage information to maintain

How Many Partitions is Good?

General recommendations

- Usually between 100 and 10,000 partitions depending on data and cluster size
- Lower bound – 2x number of cores in a cluster available to an application
- Upper bound – tasks should take 100+ ms to execute

ⓘ You will often need to change the default number of partitions to optimize your application performance! The best results are frequently achieved by experimenting with different partitioning settings and monitoring the metrics in *Spark Application Web UI*.

Summary Metrics for 6 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2 s	2 s	3 s	3 s	3 s
Scheduler Delay	44 ms	50 ms	52 ms	54 ms	84 ms

Summary Metrics for 6 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2 s	2 s	3 s	3 s	3 s
Scheduler Delay	44 ms	50 ms	52 ms	54 ms	84 ms

Other parts can be study from
https://academy.datastax.com/resources/getting-started-apache-spark

Data Shuffling

Thursday, May 11, 2017 4:30 PM

DATASTAX ACADEMY

Data Shuffling

Definition and use cases

Definition

```
Data shuffling is the process of reorganizing and transferring data from
existing partitions
into new partitions to achieve one or both properties for the resulting
partitions:
1) having a desired number of partitions
2) having pairs with the same key in the same partition
```

Main use cases

- Controlling the level of parallelism
- Supporting some key-based operations

DATASTA ACADEM

Which Operations May Trigger Shuffling?

- Repartitioning transformations
  - *repartition*, *coalesce*, *partitionBy*
- Many key-based operations
  - *reduceByKey*, *foldByKey*, *combineByKey*
  - *groupByKey*, *cogroup*
  - *join*, *leftOuterJoin*, *rightOuterJoin*, *fullOuterJoin*
  - *sortByKey*
  - *lookup*

Count

Thursday, May 11, 2017 3:02 PM

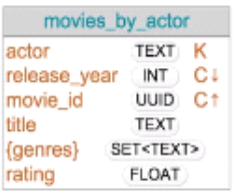
- We have already seen how we can get data from cassandra table, Transforming it and saving it back to the cassendra table.
- Here we will look on how we can optimize code which we have written earlier.

The Challenge: Suboptimal Code

DATASTAX ACADEMY

Counting movies with Johnny Depp that were released before 2015

```
sc.cassandraTable("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp' AND release_year < 2015")
  .count

// Sample output: 49
```



- This solution is not optimal as we are taking data to spark and then counting, it would have been better if we could count it inside cassandra itself.
- The **cassndraCount()** method does which is same if se do count(*) inside CQL.
- We can do this using following method.

Counting Rows in a Cassandra Table

DATAS ACADE

Spark-Cassandra Connector API

Action	Description
cassandraCount()	Returns a number of rows in the source Cassandra RDD.

Our Challenge Solution

Counting movies with Johnny Depp that were released before 2015

```
sc.cassandraTable("killr_video","movies_by_actor")
  .where("actor = 'Johnny Depp' AND release_year < 2015")
  .cassandraCount

// Sample output: 49
```

- However, there are some conditions where, it is not efficient to do this optimization. Like this

However ...

cassandraCount is inapplicable

```
sc.cassandraTable("killr_video","movies_by_actor")
```

cassandraCount is inapplicable

```
sc.cassandraTable("killr_video","movies_by_actor")
  .filter(row => row.getFloat("rating")> 6.0)
  .count
```

count is a better choice

```
val movies = sc.cassandraTable("killr_video","movies_by_actor")
               .select("release_year")
               .where("actor = 'Johnny Depp' AND release_year < 2015")
               .cache

println(movies.count)
movies.keyBy(row => row.getInt("release_year"))
      .countByKey.foreach(println)
```

- In first we will not able to use it because , cassandra can do only **=** operation and it can't do < ,> and other operations, so we have to take it to RDD and then apply **count()** operation.

GroupByKey

Thursday, May 11, 2017 3:45 PM

The Challenge: Suboptimal Code

DATASTAX ACADEMY

Grouping movies by actor and release year

```
sc.cassandraTable("killr_video","movies_by_actor")
  .select("actor","release_year","title")
  .as((a:String, y:Int, t:String) => ((a,y),t))
  .groupByKey
  .takeSample(false, 100)
  .foreach(println)

// Sample output:
// ((Johnny Depp,2010),CompactBuffer(The Tourist, Alice in Wonderland))
// ((Johnny Depp,2014),CompactBuffer(Into the Woods, Transcendence, Tusk))
```

movies_by_actor

actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

This code requires expensive shuffling.

- Here groupByKey is doing shuffling after taking data out of cassandra, we can avoid that as following.

Grouping Rows by Primary Key Columns

DATASTAX ACADEMY

Spark-Cassandra Connector API

Transformation	Description
spanByKey()	A new RDD of (*K*,*Seq[V]*) pairs is formed by grouping values for each key in the source Cassandra-based RDD of (*K*,*V*) pairs. Grouping is performed on the Cassandra side based on primary key columns.
spanBy(*f*)	A new RDD of (*K*,*Seq[V]*) pairs is formed by grouping elements of type *V* in the source Cassandra-based RDD for each key *K* as defined by function *f*. Grouping is performed on the Cassandra side based on primary key columns.

spanByKey and *spanBy* are only applicable when grouping by a table partition key and, optionally, one or more clustering columns. The grouping key must respect the natural clustering key order as defined in the table schema.

- SpanByKey() and SpanBy() consider they key in table, to be the key of RDD also while grouping
- They will push to be grouping inside cassandra.

Our Challenge Solution 1

Grouping movies by actor and release year

```
sc.cassandraTable("killr_video","movies_by_actor")
  .select("actor","release_year","title")
  .as((a:String, y:Int, t:String) => ((a,y),t))
  .spanByKey
  .takeSample(false, 100)
  .foreach(println)

// Sample output:
// ((Johnny Depp,2010),ArrayBuffer(The Tourist, Alice in Wonderland))
// ((Johnny Depp,2014),ArrayBuffer(Into the Woods, Transcendence, Tusk))
```

ⓘ No shuffling is required!

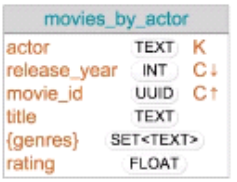
- As we can see only we had replace **groupByKey** by **spanByKey.**
- We can use **spanBy** like this,

Our Challenge Solution 2

DATASTAX ACADEMY

Grouping movies by actor and release year

```
sc.cassandraTable[(String,Int,String)]("killr_video","movies_by_actor")
  .select("actor","release_year","title")
  .spanBy{case (a,y,t) => (a,y)}
  .takeSample(false, 100)
  .foreach(println)

// Sample output:
// ((Johnny Depp,2010),ArrayBuffer((Johnny Depp,2010,The Tourist),
//                                  (Johnny Depp,2010,Alice in Wonderland)))
// ((Johnny Depp,2014),ArrayBuffer((Johnny Depp,2014,Into the Woods),
//              (Johnny Depp,2014,Transcendence), (Johnny Depp,2014,Tusk)))
```



- We can see the there is a little bit difference in the output.

Joining Table

Thursday, May 11, 2017 3:59 PM

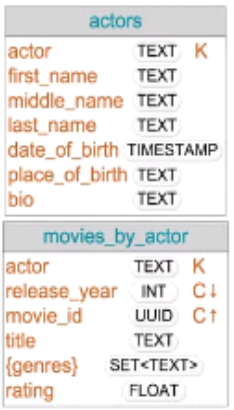
- Cassandra do not support joins, but we have already seen that using **cassendra connector** we can do join by taking two RDDs and join them,
  but it is very suboptimal as it is done by spark not by cassandra connector.

The Challenge: Suboptimal Code

DATASTAX ACADEMY

Joining two Cassandra tables on partition keys

```
val actors = sc.cassandraTable("killr_video","actors")
               .keyBy(row => row.getString("actor"))
val movies = sc.cassandraTable("killr_video","movies_by_actor")
               .keyBy(row => row.getString("actor"))
actors.join(movies).takeSample(false, 100).foreach(println)

// Sample output:
// (Johnny Depp,(
//  CassandraRow{actor: Johnny Depp, ..., place_of_birth: Owensboro, ...},
//  CassandraRow{actor: Johnny Depp, ..., title: Pirates ...}))
```



- In cassandra connector , There is another method which do this efficiently.

Joining Cassandra Tables on Primary Key Columns

DATASTAX ACADEMY

Spark-Cassandra Connector API

Transformation	Description
joinWithCassandraTable(*keyspace*, *table*)	A new RDD of (*E*, *R*) pairs is formed by combining elements *E* from the source RDD with Cassandra rows *R* from the specified table *keyspace.table*. The default join condition is the equality of the *table*'s partition key columns and respective fields of the source RDD.
on(*columns*)	Optionally used with *joinWithCassandraTable*() to specify which table *columns* to join on. This overrides the default join condition.

ⓘ Columns in a join condition can include a table partition key and, optionally, one or more clustering columns. The join condition must respect the natural clustering key order as defined in the table schema.

- We can tell on which column to join using **on()** method.

Example

DATASTAX ACADEMY

```
case class ActorYear(actor: String, release_year: Int)
val actors2014 = sc.parallelize(List(ActorYear("Johnny Depp",2014),
                                     ActorYear("Bruce Willis",2014)))

actors2014.joinWithCassandraTable("killr_video","movies_by_actor")
          .takeSample(false, 100).foreach(println)
// Sample output:
// (ActorYear(Johnny Depp,2014),CassandraRow{actor: Johnny Depp,
                                             release_year: 2010, ...})

actors2014.joinWithCassandraTable("killr_video","movies_by_actor")
          .on(SomeColumns("actor", "release_year"))
          .takeSample(false, 100).foreach(println)
// Sample output:
// (ActorYear(Johnny Depp,2014),CassandraRow{actor: Johnny Depp,
                                             release_year: 2014, ...})
```

movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

Our Challenge Solution

DATASTAX ACADEMY

Joining two Cassandra tables on partition keys

```
sc.cassandraTable("killr_video","actors")
  .joinWithCassandraTable("killr_video","movies_by_actor")
  .takeSample(false, 100).foreach(println)

// Sample output:
// (CassandraRow{actor: Johnny Depp, ..., place_of_birth: Owensboro, ...},
//  CassandraRow{actor: Johnny Depp, ..., title: Pirates ...})
```

ⓘ No shuffling is required!

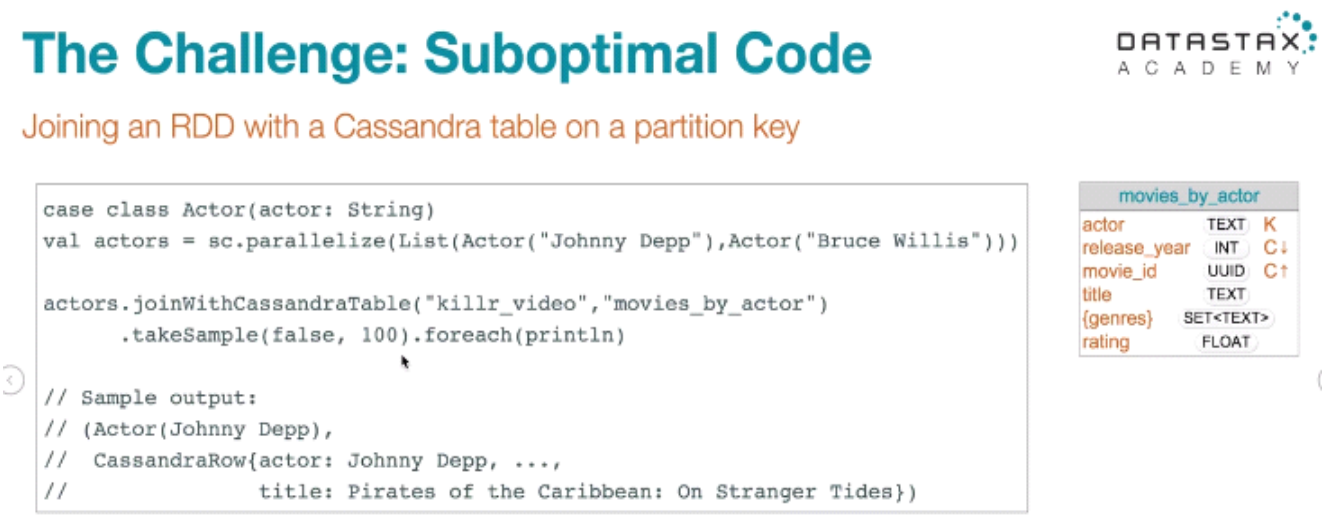
actors		
actor	TEXT	K
first_name	TEXT	
middle_name	TEXT	
last_name	TEXT	
date_of_birth	TIMESTAMP	
place_of_birth	TEXT	
bio	TEXT	

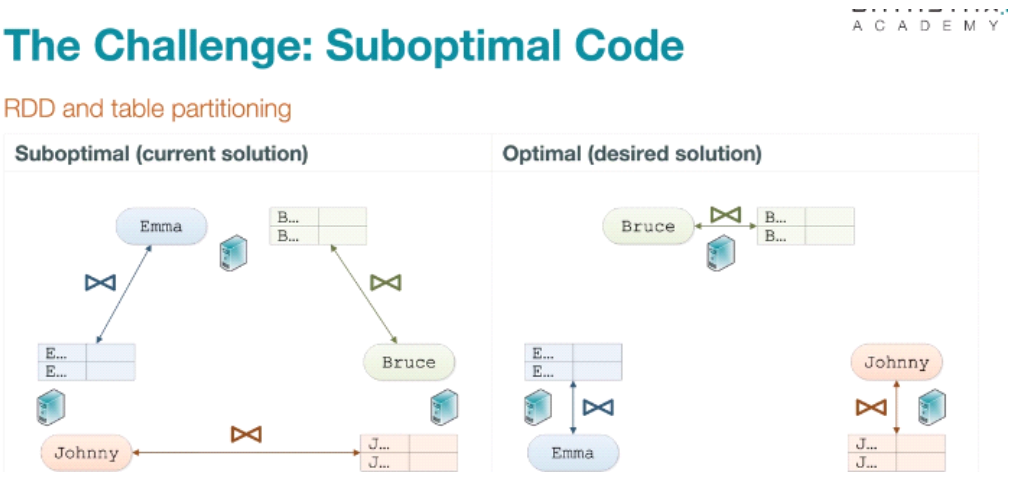
movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

Cassandra aware partitioning

Thursday, May 11, 2017 4:18 PM

- We have already seen how we can optimize cassandara join using joinWithCasssandraTable() method, which will do join inside cassandra connector.





- In the Ideal optimal case, table part which has **Emma** as key should be on same Node in which **Emma** is there, in this way we don't have to do any data movement across nodes.

- We can make this using following method,

Cassandra-Aware Repartitioning of an RDD

Spark-Cassandra Connector API

Transformation	Description
repartitionByCassandraReplica(*keyspace*, *table*, [*numPartitionsPerHost*])	A new RDD is formed by shuffling elements of the source RDD into *numPartitionsPerHost* new partitions per host (10 by default) according to the replication strategy of a given *table* and *keyspace*. The source RDD must contain information about values that correspond to the table partition key columns. The new RDD partitioner is set to *ReplicaPartitioner*.

- Source RDD(from which we are calling) should contain key which break down the other table.

Our Challenge Solution

Joining an RDD with a Cassandra table on a partition key

```
case class Actor(actor: String)
val actors = sc.parallelize(List(Actor("Johnny Depp"),Actor("Bruce Willis")))
               .repartitionByCassandraReplica("killr_video","movies_by_actor")

actors.joinWithCassandraTable("killr_video","movies_by_actor")
      .takeSample(false, 100).foreach(println)

// Sample output:
// (Actor(Johnny Depp),
//  CassandraRow{actor: Johnny Depp, ...,
//               title: Pirates of the Caribbean: On Stranger Tides})
```

movies_by_actor

actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

- We have to add only one method **repartitionByCassandraReplica()** after making RDD. (repartitionize it accordingly after making RDD)

Spark SQL Basics

Thursday, May 11, 2017 4:44 PM

- Spark SQL provide SQL like syntax.
- It is build on top of stack to provide realtional abstraction, and we can write SQL like query.

DATASTA
A C A D E M

Spark SQL

A relational engine on top of Spark

- Starting point: *CassandraSQLContext*, *HiveContext*, *SQLContext*
- Data representation: DataFrame
- Structured queries: SQL, language-integrated queries, HiveQL

- CSC is also a native object like SC(spark context)

DATASTA
A C A D E M

CassandraSQLContext

CassandraSQLContext extends *SQLContext*

Spark shell

```
scala> csc
org.apache.spark.sql.cassandra.CassandraSQLContext
```

Standalone application

```
import org.apache.spark.sql.cassandra.CassandraSQLContext
// ...

val conf = new SparkConf(true)
   .setAppName("SQL Example").setMaster("spark://127.0.0.1:7077")
   .set("spark.cassandra.connection.host", "127.0.0.1")
   .setJars(Array("your-app.jar"))

   val sc = new SparkContext(conf)
   val csc = new CassandraSQLContext(sc)
```

- There is also HC , if we have to execute any hive query, we can use it.

HiveContext

HiveContext extends SQLContext

Spark shell

```
scala> hc
org.apache.spark.sql.hive.HiveContext
```

Standalone application

```
import org.apache.spark.sql.hive.HiveContext
// ...

val conf = new SparkConf(true)
   .setAppName("HiveQL Example").setMaster("spark://127.0.0.1:7077")
   .set("spark.cassandra.connection.host", "127.0.0.1")
   .setJars(Array("your-app.jar"))

   val sc = new SparkContext(conf)
   val hc = new HiveContext(sc)
```

DataFrame

Main programming abstraction in Spark SQL

- Distributed collection of data organized into named columns
- Similar to a table in a relational database
- Has schema, rows, and rich API

```
scala> val movies = csc.sql("SELECT * FROM killr_video.movies")
movies: org.apache.spark.sql.DataFrame =
[movie_id: uuid, genres: array<string>, rating: float, release_year: int, title: string]
```

- We can use **sql()** method to run Spark SQL queries.
- result of CSC query comes in form of dataframe.
- To show dataframe in table formate(like SQL terminal) we can use **show()** method.

SQL Queries

Declarative approach

```
csc.sql(" SELECT COUNT(*) AS total          " +
        " FROM killr_video.movies_by_actor " +
        " WHERE actor = 'Johnny Depp'       ")
   .show


   +-----+
   |total|
   +-----+
   |   54|
   +-----+
```

Language-Integrated Queries

Functional approach

```
val movies = csc
    .read
    .format("org.apache.spark.sql.cassandra")
    .options(Map( "keyspace" -> "killr_video", "table" -> "movies_by_actor" ))
    .load

movies.filter("actor = 'Johnny Depp'")
      .agg(Map("*" -> "count"))
      .withColumnRenamed("COUNT(1)", "total")
      .show

   +-----+
   |total|
   +-----+
   |   54|
   +-----+
```

- In functional approach we use functions for everything.
- We can see **functional approach** is very difficult and painful.

HiveQL Queries

Similar to SQL

```
hc.sql(" SELECT COUNT(*) AS total          " +
       " FROM killr_video.movies_by_actor " +
       " WHERE actor = 'Johnny Depp'       ")
  .show


   +-----+
   |total|
```

HiveQL Queries

Similar to SQL

```
hc.sql(" SELECT COUNT(*) AS total          " +
       " FROM killr_video.movies_by_actor " +
       " WHERE actor = 'Johnny Depp'       ")
  .show


  +-----+
  |total|
  +-----+
  |   54|
  +-----+
```

Creating DataFrame

Thursday, May 11, 2017 5:07 PM

- Data frame is distributed but it is like table in relational database.
  So we can say dataframe is half table and half RDD.

Working with DataFrames

Structured data processing

- DataFrame
  - Distributed collection of data organized into named columns
  - Similar to a table in a relational database
- Working with DataFrames
  - **Creating DataFrames**
  - Accessing schema and rows
  - RDD operations

Creating a DataFrame from an RDD

Inferring schema using reflection

An RDD of case class objects

```
import csc.implicits._

case class Movie(title: String, year: Int)

val rdd = sc.parallelize(Array( Movie("Alice in Wonderland", 2010), ... ))
// rdd: org.apache.spark.rdd.RDD[Movie]

val df  = rdd.toDF()
// df: org.apache.spark.sql.DataFrame = [title: string, year: int]
```

- We can use **toDF()** method to convert RDD to DF.
- If we have case class defined we do not have to pass anything to **toDF()** function as the name of columns will be infererred from the property name.
- If we do not have **case class** and we only have tuples , we have to give the column name,
  because we do not have any column name associated with tuples.

Specifying schema programmatically

An RDD of Rows

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types._

val rdd = sc.parallelize(Array( ("Alice in Wonderland", 2010), ... ))
            .map{case(t,y) => Row(t,y)}
// rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]

val schema = StructType( List (
                StructField("title", StringType,  false),
                StructField("year",  IntegerType, false) ) )
                // true = nullable, false = not nullable

val df = csc.createDataFrame(rdd, schema)
// df: org.apache.spark.sql.DataFrame = [title: string, year: int]
```

- We have converted the tuple first in **spark.sql.Row** objects
- For making schema, we have to create **StructType and StructField.**
- Here 'StringType' etc. Comes from **spark.sql.types** package.
- After that we can use **createDataFrame(rdd,schema)** function which we take rdd and schema and form new dataframe.
- The Unique thing about this method is , we can create schema according to our need and we didn't need to depend upon the schema we are getting from the cassendra table.

Creating a DataFrame from a Cassandra Table

Specifying an SQL query

```
val df = csc.sql("SELECT * FROM killr_video.movies")

// df: org.apache.spark.sql.DataFrame =
// [movie_id: uuid, genres: array<string>, rating: float, release_year: int, title: string]
```

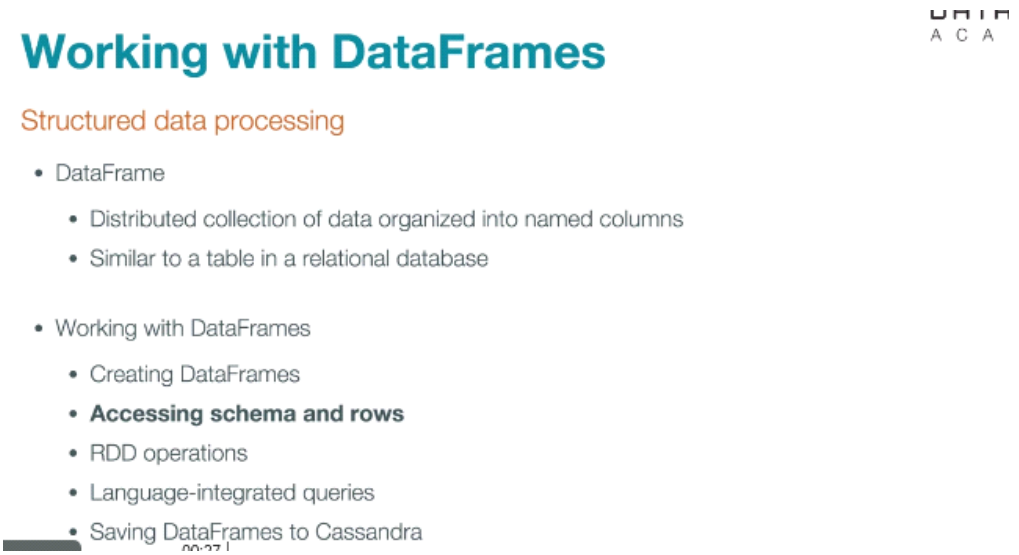
Using *DataFrameReader*

```
val df = csc.read
            .format("org.apache.spark.sql.cassandra")
            .options(Map( "keyspace" -> "killr_video", "table" -> "movies" ))
            .load

// df: org.apache.spark.sql.DataFrame =
// [movie_id: uuid, genres: array<string>, rating: float, release_year: int, title: string]
```

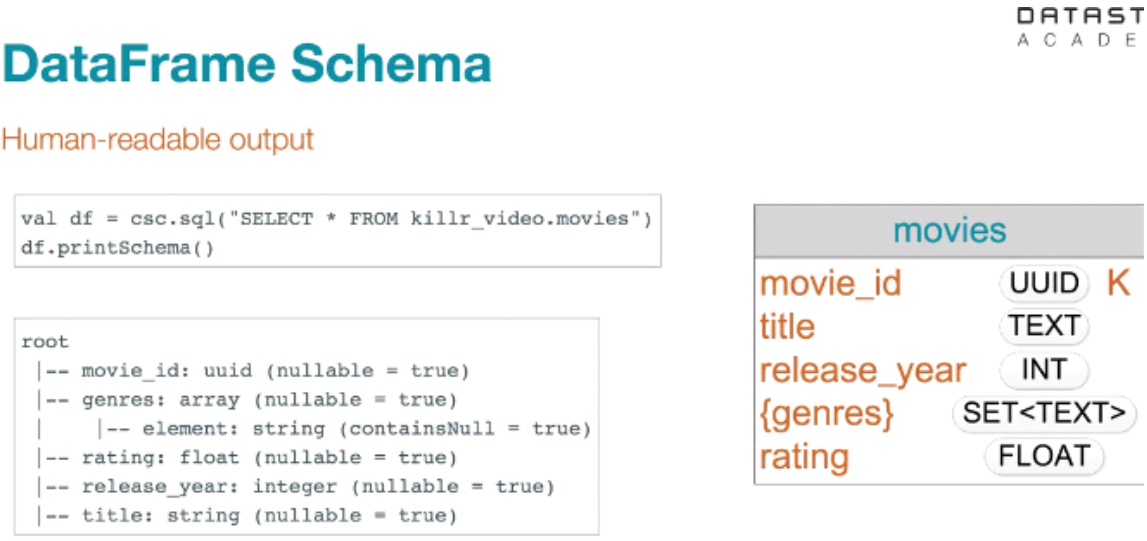
Accessing df schema and rows

Thursday, May 11, 2017 5:30 PM

- As we now get DF, we want to get rows and then data from it.



- **PrintSchema()** method of DF object schema prints schema of the DF in human readable format. (like desc of sql)

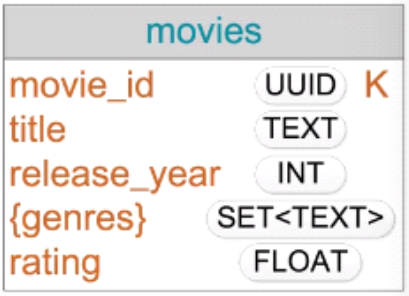


- **Dtypes()** method gives all column name and their types.

DataFrame Schema

Column names and types

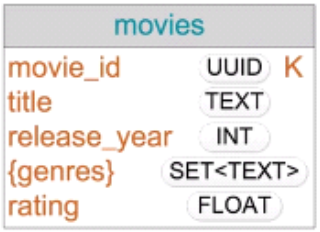
```
val df = csc.sql("SELECT * FROM killr_video.movies")
val schema = df.dtypes

// schema: Array[(String, String)] =
//   Array((movie_id,UUIDType),
//         (genres,ArrayType(StringType,true)),
//         (rating,FloatType),
//         (release_year,IntegerType),
//         (title,StringType))
```



- **Schema()** method also gives same kind of result but the ans is in Struct types.

Complete schema definition

```
val df = csc.sql("SELECT * FROM killr_video.movies")
val schema = df.schema

// schema: org.apache.spark.sql.types.StructType =
//   StructType(StructField(movie_id,UUIDType,true),
//              StructField(genres,ArrayType(StringType,true),true),
//              StructField(rating,FloatType,true),
//              StructField(release_year,IntegerType,true),
//              StructField(title,StringType,true))
```



- Now, we want to access data from the DF .

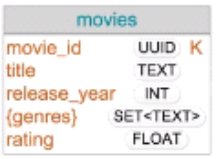
Accessing primitive values

```
val df = csc.sql(" SELECT title, release_year " +
                 " FROM killr_video.movies " +
                 " WHERE title = 'Alice in Wonderland'")
// df: org.apache.spark.sql.DataFrame = [title: string, release_year: int]

val row = df.first
// row: org.apache.spark.sql.Row = [Alice in Wonderland,2010]

println(row(0))          // Alice in Wonderland

println(row.isNullAt(1)) // false

println(row.getInt(1))   // 2010
```

- We have here used **first()** method which returns first row as **spark.sql.Row** object.
- We can use array like indexing to access the value -> **row(0**) => this function used mostly.
- Before accessing it is good to check if it is null or not using **isNullAt(index)**
- If we know type we can also use **get<Type>()** method.

Accessing complex values

```
val df = csc.sql(" SELECT genres " +
                 " FROM killr_video.movies " +
                 " WHERE title = 'Alice in Wonderland'")
// df: org.apache.spark.sql.DataFrame = [genres: array<string>]

val row = df.first
// row: org.apache.spark.sql.Row = [ArrayBuffer(Adventure, Family, Fantasy)]

row(0).asInstanceOf[Seq[String]].foreach(println)
// Adventure
// Family
// Fantasy
```



- Here row(0) has is type casted using **asInstanceOf[] .**

asInstanceOf[]

Use Scala's `asInstanceOf` method to cast an instance to the desired type. In the following example, the object returned by the `lookup` method is cast to an instance of a class named `Recognizer`:

```
val recognizer = cm.lookup("recognizer").asInstanceOf[Recognizer]
```

This Scala code is equivalent to the following Java code:

```
Recognizer recognizer = (Recognizer)cm.lookup("recognizer");
```

The `asInstanceOf` method is defined in the Scala `Any` class and is therefore available on all objects.

RDD operations on DF

Thursday, May 11, 2017 5:52 PM

- There are some methods which are RDD alike or result in RDD formation.

Repartitioning and Persistence Transformations

DATASTAX ACADEMY

DataFrame in, DataFrame out

Transformation	Description
repartition(*numPartitions*)	DataFrame repartitioning transformations
coalesce(*numPartitions*)	
persist([*storageLevel*])	DataFrame persistence transformations
cache()	
unpersist()	

- Cache()/Persist() here also same as cache() or persist() of RDD.

DataFrame in, RDD out

Transformation	Description
map(*f*)	A new RDD is formed by applying a function *f* on each row of the source DataFrame. There is a *one-to-one* correspondence between DataFrame rows and RDD elements.
flatMap(*f*)	A new RDD is formed by applying a function *f* on each row of the source DataFrame. There is a *one-to-many* correspondence between DataFrame rows and RDD elements if *f* returns a collection with more than one element.
rdd	A new RDD of rows is formed from the source DataFrame.
toJSON	A new RDD of JSON strings is formed by converting rows to JSON in the source DataFrame.

- **Rdd()** method is going to convert DF to RDD.

DataFrame Actions

DATASTAX ACADEMY

Triggering computation

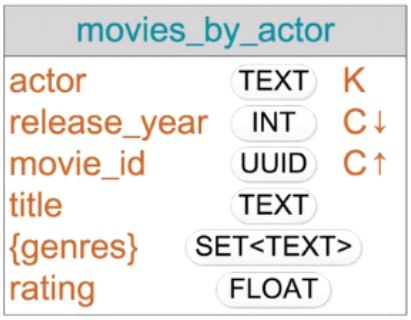
Action	Description
collect()	Returns an *Array* with all rows of the source DataFrame.
count()	Returns a total number of rows in the source DataFrame.
first() or **head**()	Returns the first row of the source DataFrame.
take(*n*) or **head**(*n*)	Returns an *Array* with the first *n* rows of the source DataFrame.
show([*n*])	Displays the first *n* rows of the source DataFrame in a tabular form. Strings more than 20 characters are truncated. By default, *n* = *20*.
foreach(*f*)	Executes a function *f* on each row of the source DataFrame. The function usually implements a side effect, such as updating an *accumulator* variable or interacting with an external system.

- Show() is just human readable form of take(n) or head(n)

Example

Counting and displaying Johnny Depp's movies

```
val df = csc.sql(" SELECT title, release_year, rating " +
                 " FROM killr_video.movies_by_actor " +
                 " WHERE actor = 'Johnny Depp'")
            .coalesce(1).cache
println("Total: " + df.count)
df.show(4)

// Total: 54
// +--------------------+------------+------+
// |               title|release_year|rating|
// +--------------------+------------+------+
// |Pirates of the Ca...|        2017|  null|
// |Alice Through the...|        2016|  null|
// |         Yoga Hosers|        2015|  null|
// |           Mortdecai|        2015|   5.5|
// +--------------------+------------+------+
```



- **Coalesce(n) will gether the data in n node, here we have just store in 1 as data is very small.**

Language-Integrated Queries

Thursday, May 11, 2017 6:06 PM

- We have seen how we can use spark SQL queries, but we may sometimes wanted to do this using functions, using programming, as we hav already seen with

Working with DataFrames

DATASTA
ACADEM

Structured data processing

- DataFrame
  - Distributed collection of data organized into named columns
  - Similar to a table in a relational database
- Working with DataFrames
  - Creating DataFrames
  - Accessing schema and rows
  - RDD operations
  - **Language-integrated queries**
  - Saving DataFrames to Cassandra

DataFrame Query API

DATASTAX
ACADEMY

Unary transformations

Transformation	Description
select(*columns*)	A new DataFrame is formed by selecting a set of *columns* from the source DataFrame.
withColumnRenamed(*oldColumn*, *newColumn*)	A new DataFrame is formed by renaming a column in the source DataFrame.
distinct()	A new DataFrame is formed by eliminating duplicate rows in the source DataFrame.
where(*condition*) or **filter**(*condition*)	A new DataFrame is formed by rows from the source DataFrame that satisfy a *condition*.
agg(*column-aggregates*)	A new DataFrame is formed by applying aggregate functions to columns in the source DataFrame.

groupBy(*columns*)	A new DataFrame is formed by grouping data based on *columns* in the source DataFrame.
orderBy(*columns*) or **sort**(*columns*)	A new DataFrame is formed by sorting rows based on *columns* of the source DataFrame in the ascending (default) or descending order.
limit(*n*)	A new DataFrame is formed by the first *n* rows of the source DataFrame.

- We can use **agg()** for aggregating.

Binary transformations

Transformation	Description
join(*otherDF*, [*condition*], [*type*])	A new DataFrame is formed by joining the source DataFrame and *otherDF* based on an optional *condition*. The optional join *type* parameter allows switching between inner (default) and outer joins.
unionAll(*otherDF*)	A new DataFrame is formed by all rows of the source DataFrame and *otherDF*. The source DataFrame and *otherDF* must be union-compatible. Duplicate rows in the result are retained.
intersect(*otherDF*)	A new DataFrame is formed by only those rows that appear in both the source DataFrame and *otherDF*. The source DataFrame and *otherDF* must be union-compatible. Duplicate rows in the result are eliminated.
except(*otherDF*)	A new DataFrame is formed by rows that appear in the source DataFrame but not in *otherDF*. The source DataFrame and *otherDF* must be union-compatible. Duplicate rows in the result are eliminated.

Supporting Functions

import org.apache.spark.sql.functions._

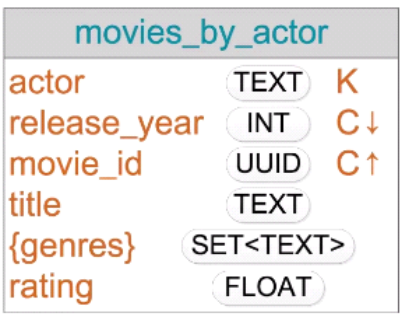
Category	Sample functions
Aggregate functions	*avg*, *count*, *max*, *min*, *sum*
Collection functions	*array_contains*, *sort_array*, *size*
Date-time functions	*current_date*, *current_timestamp*, *second*, *hour*, *month*, *year*
Math functions	*ceil*, *floor*, *round*, *pow*, *sqrt*, *log*, *sum*, *sin*, *cos*, *tan*
Sorting functions	*asc*, *desc*
String functions	*concat*, *length*, *substring*, *trim*
UDF functions	*udf*, *callUDF*
Window functions	*denseRank*, *percentRank*, *rank*, *lag*, *lean*
Miscellaneous functions	*col*, *column*, *rand*, *randn*

Example

DATASTAX ACADEMY

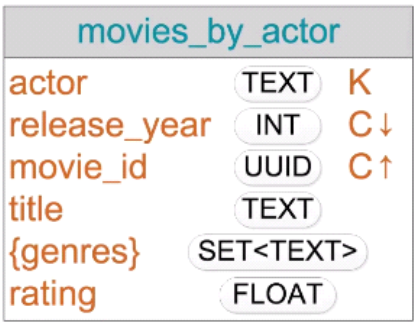
STEP 1: Creating a DataFrame from a Cassandra table

```
val df = csc.read
            .format("org.apache.spark.sql.cassandra")
            .options(Map( "keyspace" -> "killr_video",
                          "table" -> "movies_by_actor" ))
            .load
```



STEP 2: Executing a language-integrated query

```
import org.apache.spark.sql.functions._

df.filter("actor = 'Johnny Depp'")
  .groupBy("release_year")
  .agg(Map("*" -> "count", "rating" -> "avg"))
  .withColumnRenamed("COUNT(1)", "total_movies")
  .withColumnRenamed("AVG(rating)", "average_rating")
  .select("release_year", "total_movies", "average_rating")
  .orderBy(desc("total_movies"), desc("average_rating"))
  .limit(3)
  .show

// +------------+------------+-----------------+
// |release_year|total_movies|   average_rating|
// +------------+------------+-----------------+
// |        2004|           4|6.850000023841858|
// |        2000|           3|6.933333396911621|
// |        2011|           3|6.733333269755046|
// +------------+------------+-----------------+
```



Saving DF to Cassandra

Thursday, May 11, 2017 6:33 PM

Working with DataFrames

Structured data processing

- DataFrame
  - Distributed collection of data organized into named columns
  - Similar to a table in a relational database
- Working with DataFrames
  - Creating DataFrames
  - Accessing schema and rows
  - RDD operations
  - Language-integrated queries
  - **Saving DataFrames to Cassandra**

Running Example

Simple ETL scenario

1. Reading from table *movies*
2. Filtering movies by genre "Family"
3. Saving into table *family_movies*

movies		
movie_id	UUID	K
title	TEXT	
release_year	INT	
{genres}	SET<TEXT>	
rating	FLOAT	

family_movies		
movie_id	UUID	K
title	TEXT	
release_year	INT	
{genres}	SET<TEXT>	
rating	FLOAT	

- ETL – extract, transform , load. \

Reading from a Cassandra Table

Using *DataFrameReader*

```
val movieDF = csc.read
                 .format("org.apache.spark.sql.cassandra")
                 .options(Map( "keyspace" -> "killr_video",
                               "table" -> "movies" ))
                 .load
```

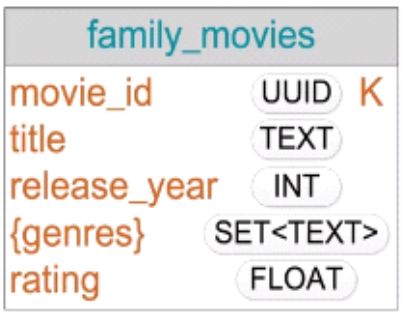
movies		
movie_id	UUID	K
title	TEXT	
release_year	INT	
{genres}	SET<TEXT>	
rating	FLOAT	

Using language-integrated query API

```
import org.apache.spark.sql.functions._

val familyDF = movieDF.filter(col("genres").contains("Family"))

familyDF.show

// +--------------------+--------------------+------+------------+--------------------+
// |            movie_id|              genres|rating|release_year|               title|
// +--------------------+--------------------+------+------------+--------------------+
// |3391c072-af52-4d3...|ArrayBuffer(Adven...|  null|        2016|Alice Through the...|
// |87032a77-0ccd-416...|ArrayBuffer(Biogr...|   7.8|        2004|   Finding Neverland|
// |f8ecbd4a-c3e0-41e...|ArrayBuffer(Adven...|   6.5|        2010| Alice in Wonderland|
// |89ccddea-a845-499...|ArrayBuffer(Adven...|   6.7|        2005|Charlie and the C...|
// +--------------------+--------------------+------+------------+--------------------+
```

Using *DataFrameWriter*

```
familyDF.write
        .format("org.apache.spark.sql.cassandra")
        .options(Map( "keyspace" -> "killr_video",
                      "table" -> "family_movies" ))
        .save
```



Querying Cassandra with SQL

Thursday, May 11, 2017 6:41 PM

Executing SQL Queries

DATASTAX ACADEMY

CassandraSQLContext API

Method	Description
setKeyspace(*keyspace*)	Sets a default Cassandra *keyspace*.
sql(*query*) or **cassandraSql**(*query*)	Executes an SQL *query* and returns a DataFrame.

- These method will be used CSC, object which we have used earlier also.
- We can set database (like **use** in SQL).

DataFrame API

Method	Description
registerTempTable(*tableName*)	Registers the source DataFrame as a temporary table that can be used in SQL queries by name *tableName*.
explain()	Prints a physical query execution plan to the console.

Example: Querying a Cassandra Table

DATASTAX ACADEMY

Find the largest rating for Johnny Depp's movie

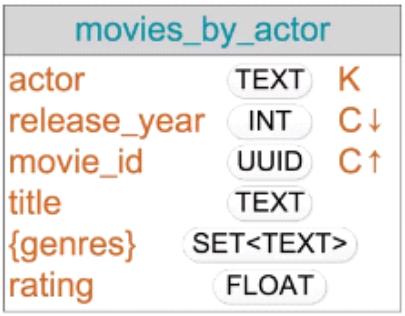
```
csc.setKeyspace("killr_video")

val maxDF = csc.sql(
     " SELECT actor, MAX(rating) AS max_rating " +
     " FROM movies_by_actor                    " +
     " WHERE actor = 'Johnny Depp'             " +
     " GROUP BY actor                          " )

maxDF.show

// +-----------+----------+
// |      actor|max_rating|
// +-----------+----------+
// |Johnny Depp|       8.6|
// +-----------+----------+
```

movies_by_actor		
actor	TEXT	K
release_year	INT	C↓
movie_id	UUID	C↑
title	TEXT	
{genres}	SET<TEXT>	
rating	FLOAT	

Example: Querying a Temporary Table

Find Johnny Depp's movies with ratings higher than the largest rating - 1

```
maxDF.registerTempTable("max_rating")

val movieDF = csc.sql(
     " SELECT M.actor, title, release_year, rating     " +
     " FROM max_rating AS R JOIN movies_by_actor AS M " +
     " ON (R.actor = M.actor)                          " +
     " WHERE rating > max_rating - 1                   " +
     " ORDER BY release_year DESC, rating DESC         " )

movieDF.show(2)
// +-----------+--------------------+------------+------+
// |      actor|               title|release_year|rating|
// +-----------+--------------------+------------+------+
// |Johnny Depp|    Finding Neverland|        2004|   7.8|
// |Johnny Depp|Pirates of the Ca...|        2003|   8.1|
```



Writing Efficient SQL Queries

Thursday, May 11, 2017 6:48 PM

- We can optimize query by pushing down filtering as soon as possible.
- Pushdown automatically happens in cassendra.

TODO : read more about partition key and clustering key.

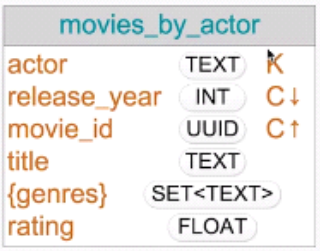
Reading Data from Cassendra

Wednesday, May 10, 2017 2:03 PM

Cassandra as a Data Source for Spark

Example Cassandra table from the KillrVideo domain

```
CREATE TABLE movies_by_actor (
  actor TEXT,
  release_year INT,
  movie_id UUID,
  title TEXT,
  genres SET<TEXT>,
  rating FLOAT,
  PRIMARY KEY ((actor), release_year, movie_id)
) WITH CLUSTERING ORDER BY (release_year DESC, movie_id ASC);
```



The Challenge

Retrieve five most recent movies featuring Johnny Depp that were released after 2010

CQL solution

```
SELECT title, release_year
FROM movies_by_actor
WHERE actor = 'Johnny Depp' AND release_year > 2010
ORDER BY release_year DESC
LIMIT 5;
```

How can you express this logic using *Spark-Cassandra Connector* API?

Spark-Cassandra Connector API

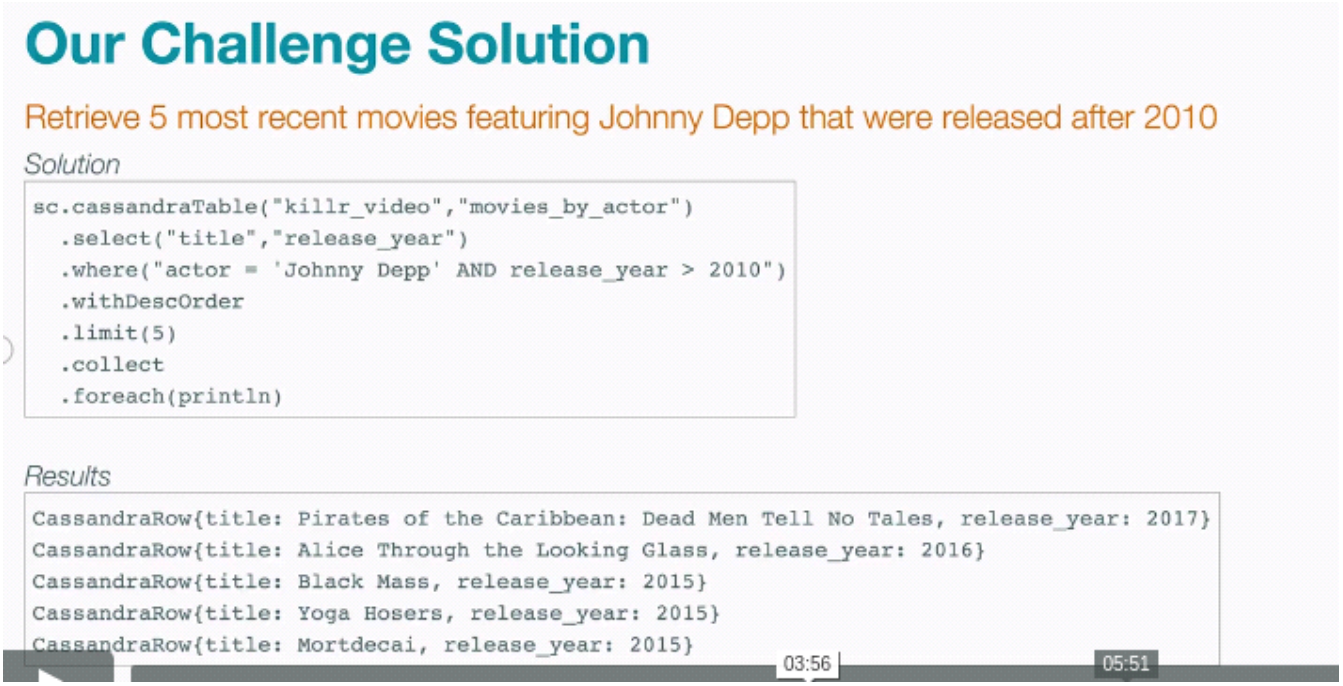
Common methods for data retrieval

API Call	Description
cassandraTable(*keyspace*, *table*)	Returns an RDD that contains all rows from a Cassandra *table* in a specified *keyspace*. This method is called on a *SparkContext* object.
select(*columns*)	Optionally used with *cassandraTable*() to specify which table *columns* to retain in the result.
where(*condition*, [*parameters*])	Optionally used with *cassandraTable*() to specify a CQL *condition* to only retrieve rows that satisfy the *condition*. The *condition* may optionally be parameterized.
withAscOrder or **withDescOrder**	Optionally used with *cassandraTable*() to specify how to order retrieved rows from a single Cassandra partition based on clustering columns.
limit(*n*)	Optionally used with *cassandraTable*() to specify how many rows to retrieve.

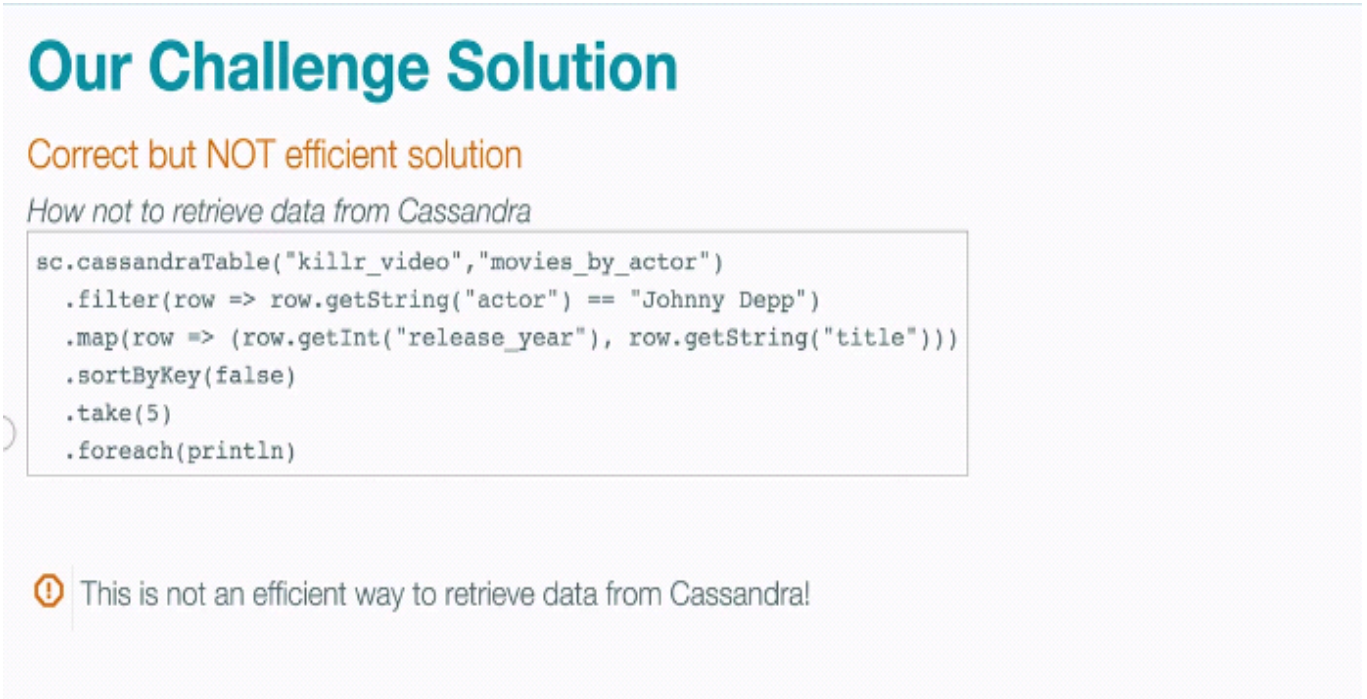
Mapping Spark-Cassandra Connector API to CQL

As simple as it can be …

```
select(columns)                    -->    SELECT columns
cassandraTable(keyspace, table)    -->    FROM keyspace.table
where(condition, [parameters])     -->    WHERE condition
withAscOrder | withDescOrder       -->    ORDER BY clustering_columns ASC | DESC
limit(n)                           -->    LIMIT n
```



- There is another bad way of doing this, in this method we are taking all rows from cassendra database and then filtering which is very bad as the data could be a lot than the limit

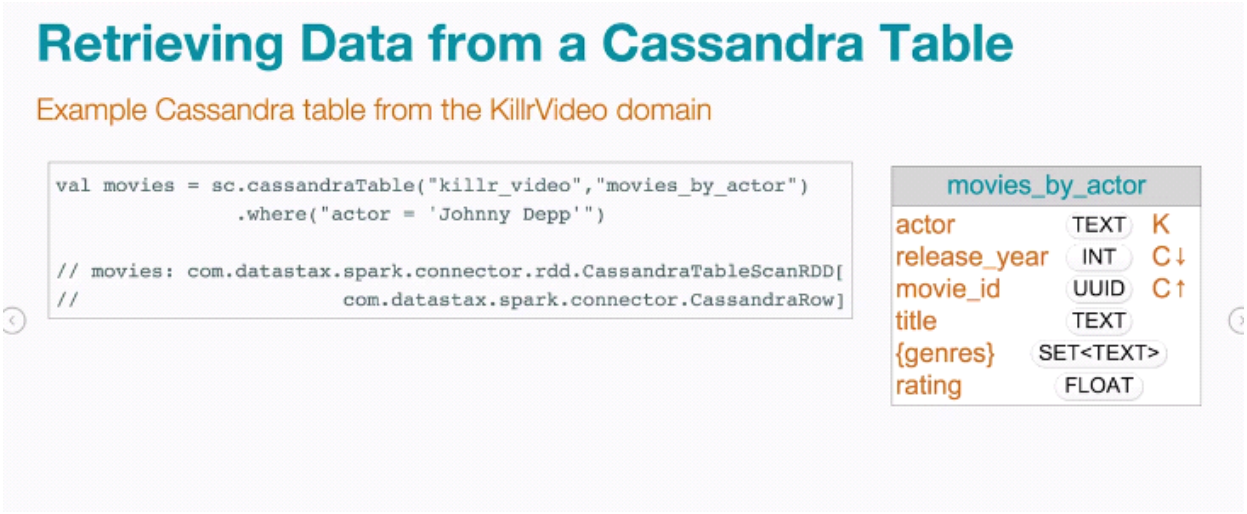


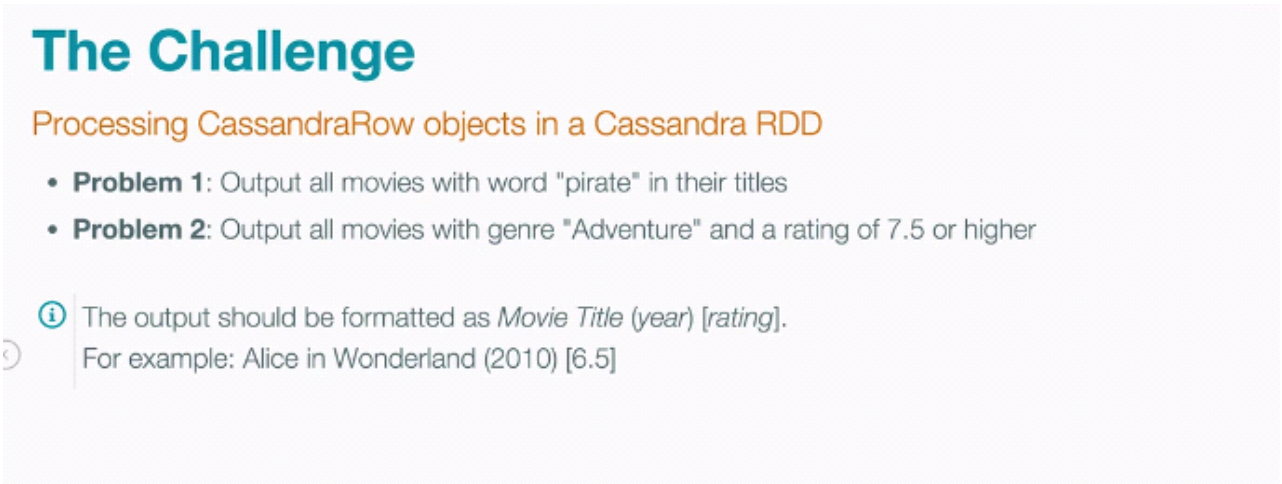
.

Processing Casendra data

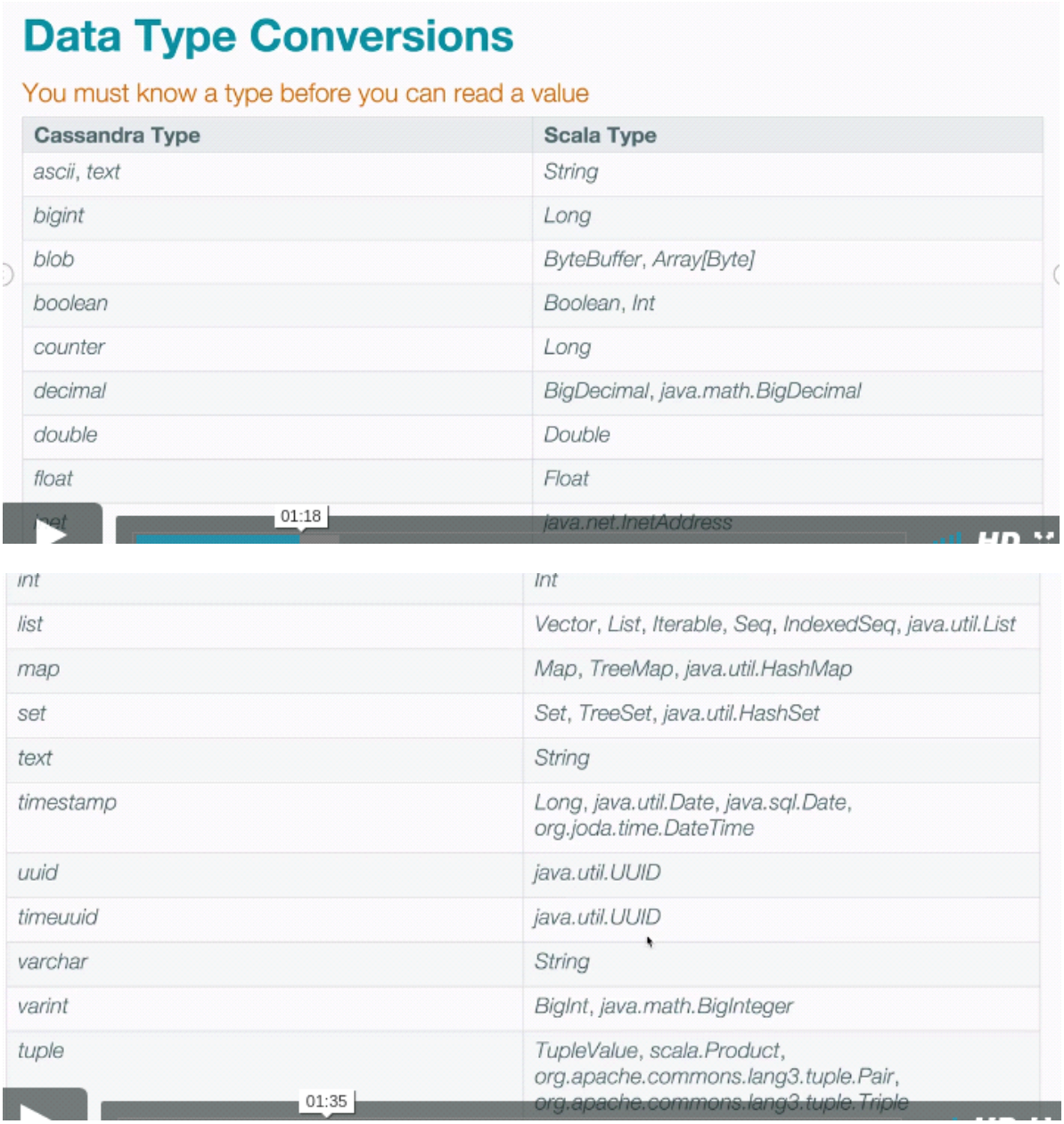
Wednesday, May 10, 2017 2:13 PM

- We have seen, how we can get data frm cassendraTable using methods.





- The following mapping exists xbetween scala and cassendra data type.

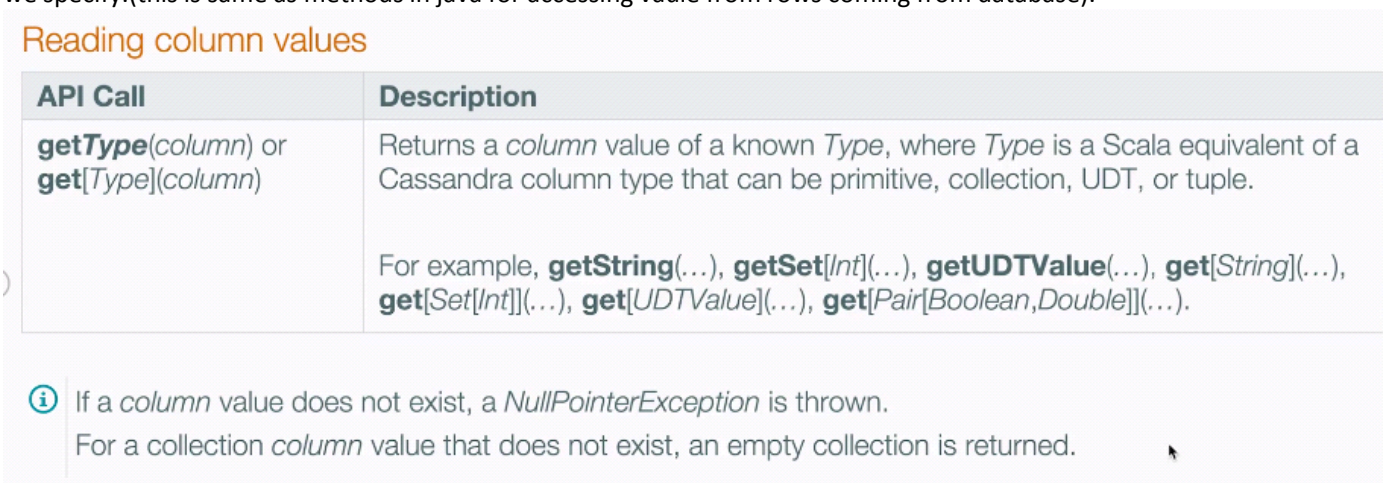


Data Type Conversions

You must know a type before you can read a value

Cassandra Type	Scala Type
ascii, text	String
bigint	Long
blob	ByteBuffer, Array[Byte]
boolean	Boolean, Int
counter	Long
decimal	BigDecimal, java.math.BigDecimal
double	Double
float	Float
inet	java.net.InetAddress
int	Int
list	Vector, List, Iterable, Seq, IndexedSeq, java.util.List
map	Map, TreeMap, java.util.HashMap
set	Set, TreeSet, java.util.HashSet
text	String
timestamp	Long, java.util.Date, java.sql.Date, org.joda.time.DateTime
uuid	java.util.UUID
timeuuid	java.util.UUID
varchar	String
varint	BigInt, java.math.BigInteger
tuple	TupleValue, scala.Product, org.apache.commons.lang3.tuple.Pair, org.apache.commons.lang3.tuple.Triple

- For reading values from cassendra rows we can use following function, get<type>(col), here type we specify.(this is same as methods in java for accessing vaule from rows coming from database).

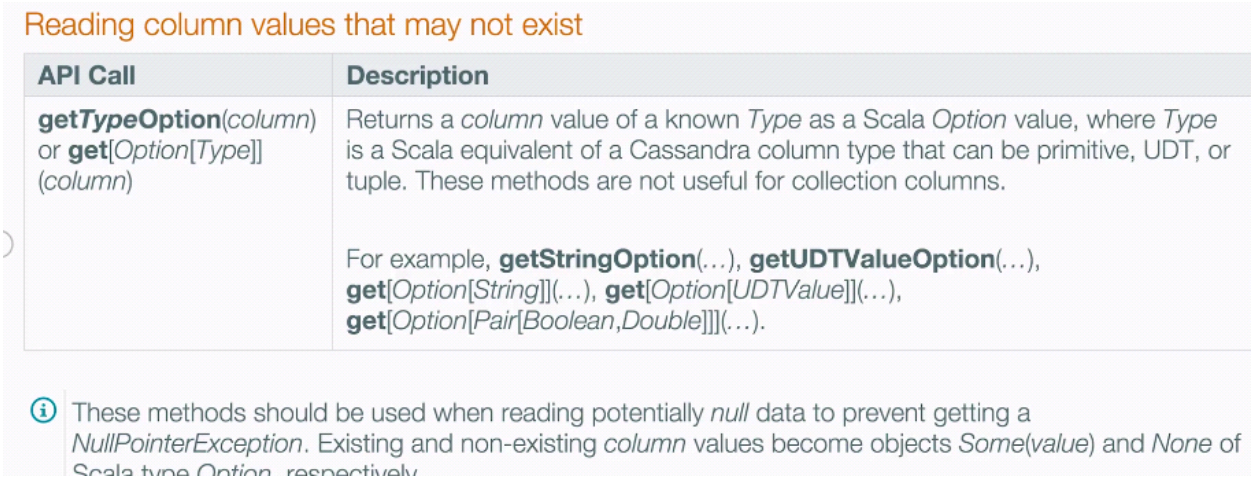


Reading column values

API Call	Description
get*Type*(*column*) or **get**[*Type*](*column*)	Returns a *column* value of a known *Type*, where *Type* is a Scala equivalent of a Cassandra column type that can be primitive, collection, UDT, or tuple. For example, **getString**(…), **getSet**[*Int*](…), **getUDTValue**(…), **get**[*String*](…), **get**[*Set*[*Int*]](…), **get**[*UDTValue*](…), **get**[*Pair*[*Boolean*,*Double*]](…).

ⓘ If a *column* value does not exist, a *NullPointerException* is thrown.
For a collection *column* value that does not exist, an empty collection is returned.

- The problem is if column is not present we get N**ullPointerException ,** and for avoiding we use following function.



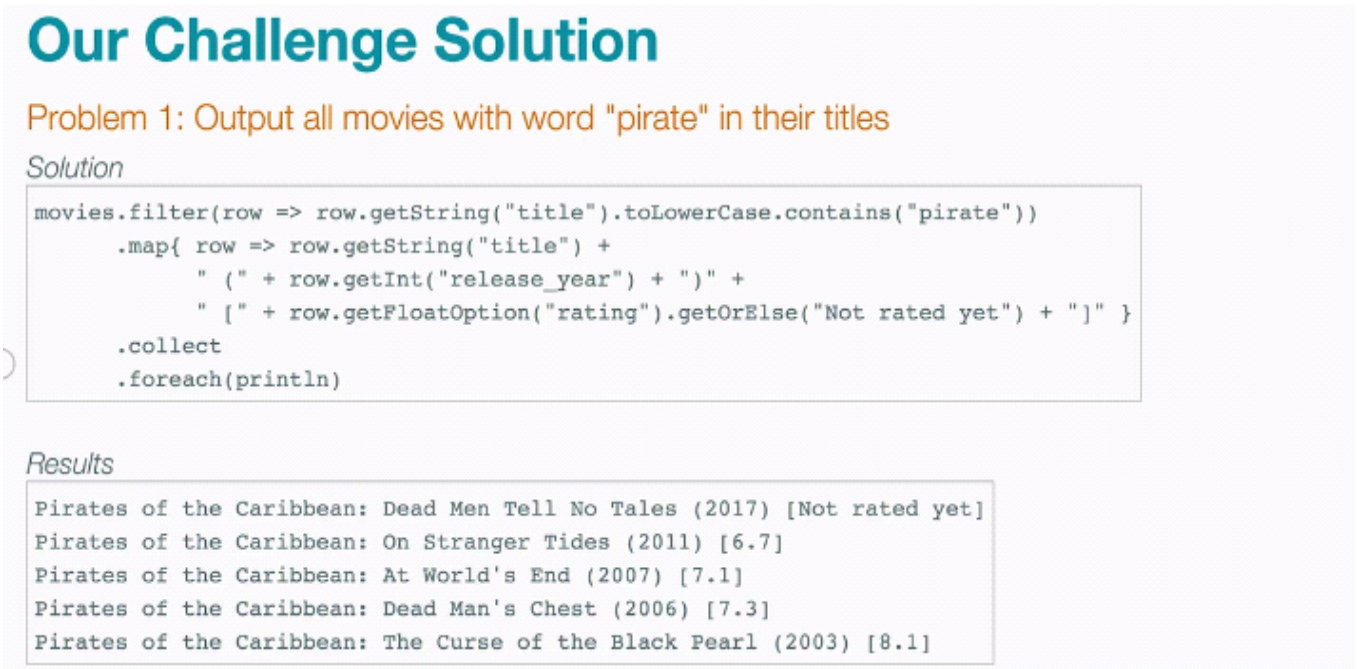
Reading column values that may not exist

API Call	Description
get*Type*Option(*column*) or **get**[*Option*[*Type*]] (*column*)	Returns a *column* value of a known *Type* as a Scala *Option* value, where *Type* is a Scala equivalent of a Cassandra column type that can be primitive, UDT, or tuple. These methods are not useful for collection columns. For example, **getStringOption**(…), **getUDTValueOption**(…), **get**[*Option*[*String*]](…), **get**[*Option*[*UDTValue*]](…), **get**[*Option*[*Pair*[*Boolean*,*Double*]]](…).

These methods should be used when reading potentially *null* data to prevent getting a *NullPointerException*. Existing and non-existing *column* values become objects *Some*(*value*) and *None* of Scala type *Option*, respectively.

- In **get[Option[Type]] , we are wrapping 'Type' by 'Option' so we do not get null pointer exception.**

- Let we have RDD "movies", already made from cassedra table and having movie data.



Our Challenge Solution

Problem 1: Output all movies with word "pirate" in their titles

Solution

```
movies.filter(row => row.getString("title").toLowerCase.contains("pirate"))
      .map{ row => row.getString("title") +
            " (" + row.getInt("release_year") + ")" +
            " [" + row.getFloatOption("rating").getOrElse("Not rated yet") + "]" }
      .collect
      .foreach(println)
```

Results

```
Pirates of the Caribbean: Dead Men Tell No Tales (2017) [Not rated yet]
Pirates of the Caribbean: On Stranger Tides (2011) [6.7]
Pirates of the Caribbean: At World's End (2007) [7.1]
Pirates of the Caribbean: Dead Man's Chest (2006) [7.3]
Pirates of the Caribbean: The Curse of the Black Pearl (2003) [8.1]
```

- In filter() method, we first fetch the column 'title' as string, and then check if it contains 'pirate' or not.
- In the map() function we are fetching 'title' column(which is a string) , and then 'release_year', for 'ratings' we have used **getFloatOption**() and then **getOrElse()** so if 'rating' is **null** then, the value which are in **getOrElse()** will be returned.
  TO LEARN – **getIntOption** means getInt() + Option need to be given as **getOrElse()**

Our Challenge Solution

Problem 2: Output all movies with genre "Adventure" and a rating of 7.5 or higher

Solution

```
movies.filter{ row => row.getSet[String]("genres").contains("Adventure") &&
                      row.get[Option[Float]]("rating").isDefined &&
                      row.get[Option[Float]]("rating").get >= 7.5 }
      .map{ row => row.getString("title") +
            " (" + row.getInt("release_year") + ")" +
            " [" + row.getFloat("rating") + "]" }
      .collect
      .foreach(println)
```

Results

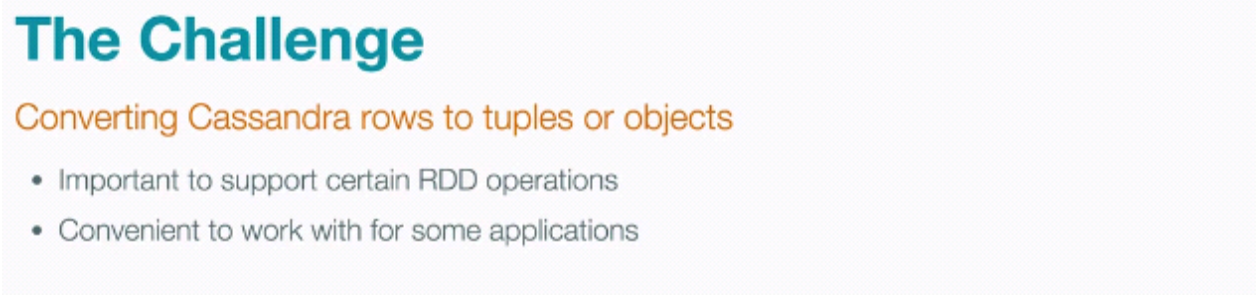
```
Pirates of the Caribbean: The Curse of the Black Pearl (2003) [8.1]
```

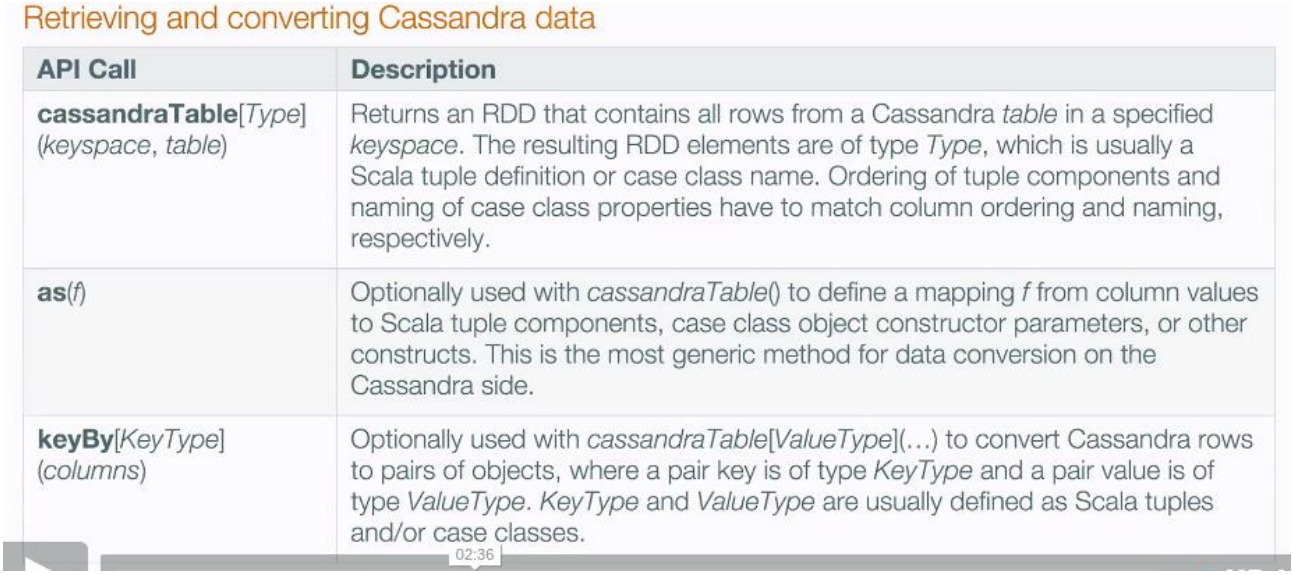
- In filter(), we are first fetching "genres" column which is "set of Strings",
  And we have used **get[Option[Float]]()** , we could have used **getFloatOption()** also.

Converting Cassendra data

Wednesday, May 10, 2017 3:04 PM

- We may wanted to convert
  - Cs row to tuple
  - Cs row to instance of Class





- We can pass type in **cassendraTable()** function to create all RDD element to given **Type**,
  Which is by default scala tuple or case class.
- **KeyBy** will be used in **pair RDD**,
  some columns form a key and rest will be value.

Solution 1: Rows-to-Tuples

Using method cassandraTable[Type](...)

```
val movies = sc.cassandraTable[(Int,String,Option[Float])]("killr_video","movies_by_actor")
               .where("actor = 'Johnny Depp'").select("release_year","title","rating")
// movies: com.datastax.spark.connector.rdd.CassandraTableScanRDD[(Int, String, Option[Float])]
```

Using method as(f)

```
val movies = sc.cassandraTable("killr_video","movies_by_actor")
               .where("actor = 'Johnny Depp'").select("release_year","title","rating")
               .as((y:Int,t:String,r:Option[Float]) => (y,t,r))
// movies: com.datastax.spark.connector.rdd.CassandraRDD[(Int, String, Option[Float])]
```

- We have used here **cassendraTable[Type](keyspace,table)** method to specify type in which the RDD will be converted automatically.
- In **as()** example, we are making tuple of the result coming.

Solution 2: Rows-to-Objects

Using method cassandraTable[Type](...)

```
case class Record(releaseYear: Int, title: String, rating: Option[Float])
val movies = sc.cassandraTable[Record]("killr_video","movies_by_actor")
               .where("actor = 'Johnny Depp'").select("release_year","title","rating")
// movies: com.datastax.spark.connector.rdd.CassandraTableScanRDD[Record]
```

Using method as(f)

```
case class Record(releaseYear: Int, title: String, rating: Option[Float])
val movies = sc.cassandraTable("killr_video","movies_by_actor")
               .where("actor = 'Johnny Depp'").select("release_year","title","rating")
               .as((y:Int,t:String,r:Option[Float]) => new Record(y,t,r))
// movies: com.datastax.spark.connector.rdd.CassandraRDD[Record]
```

- For converting rows to object we use **case class** , and then we use **cassendraTable[Type]()** Syntax and give **case class** as type, so it will be converted to case class object type automatically.
- **Case class** is lot like **struct** in C programming language.
- With **as()** function , we will convert the data coming as a new object using **new** keyword.

Solution 3: Correct but Less Efficient

Row-to-tuple solution using transformation map(f)

```
val movies = sc.cassandraTable("killr_video","movies_by_actor")
               .where("actor = 'Johnny Depp'").select("release_year","title","rating")
               .map(row =>
(row.getInt("release_year"),row.getString("title"),row.getFloatOption("rating")))
// movies: org.apache.spark.rdd.RDD[(Int, String, Option[Float])]
```

Row-to-object solution using transformation map(f)

```
case class Record(releaseYear: Int, title: String, rating: Option[Float])
val movies = sc.cassandraTable("killr_video","movies_by_actor")
               .where("actor = 'Johnny Depp'").select("release_year","title","rating")
               .map(row => new Record(row.getInt("release_year"),row.getString("title"),
                                      row.getFloatOption("rating")))
```

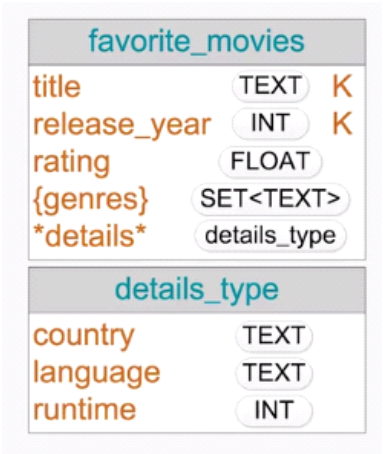
```
                .where("actor = 'Johnny Depp'").select("release_year","title","rating")
                .map(row => new Record(row.getInt("release_year"),row.getString("title"),
                                       row.getFloatOption("rating")))
// movies: org.apache.spark.rdd.RDD[Record]
```

- This is not efficient method as lot of new objects are being created.

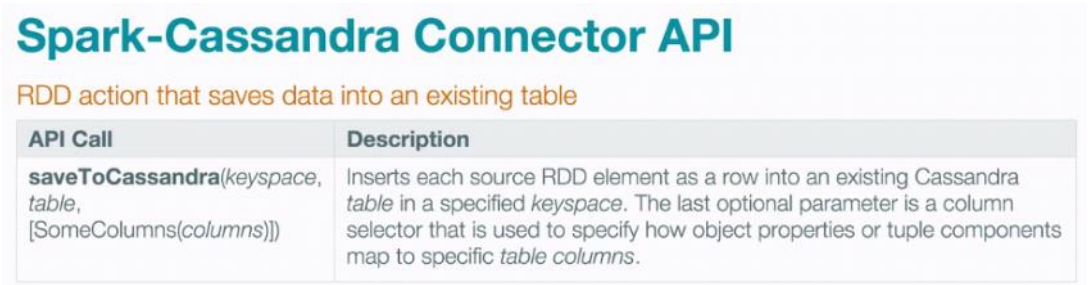
Saving Data back to Cassendra

Wednesday, May 10, 2017 4:05 PM

- Let we have following table 'favorite_movies', we also have a UDT(user defined datatype) 'detail_type' which is stored in col umn 'details'.



- The key method for this is **saveToCassandra(keyspace,table)**



- **SomeColumns()** will get the name of column we wanted to map the row columns to the table column in database.
- We have **saveAsCassandraTable()** which will create a new table and save to it.
- The method **saveAsCassandraTableEx()** will create a new table with given table definition.

RDD actions that create and save data into a new table

API Call	Description
saveAsCassandraTable(*keyspace*, *table*, [SomeColumns(*columns*)])	Creates a Cassandra *table* in an existing *keyspace* and inserts each source RDD element as a row into the *table*. A newly created *table* has a primary key consisting of the first column in the *SomeColumns(...)* parameter, if present, or a column corresponding to the first property name in an RDD element class definition.
saveAsCassandraTableEx(*tableDefinition*, [SomeColumns(*columns*)])	Creates a Cassandra table in an existing keyspace according to a specified *tableDefinition* and inserts each source RDD element as a row into the table. *tableDefinition* can be used to customize partition key and clustering key columns, as well as any additional columns and their data types.

- As we know **cassandraTable()** will return objects of class **cassandraRow,** so we can use **saveToCassandra() on that.**
-

Saving an RDD with *CassandraRow* Objects

movie: RDD[CassandraRow]

```
val movie = sc.cassandraTable("killr_video","movies_by_actor")
              .where("actor = 'Johnny Depp'")
              .select("title","release_year","rating")
              .filter(row => row.getString("title") == "Alice in Wonderland")

movie.saveToCassandra("killr_video","favorite_movies",
                      SomeColumns("title","release_year","rating"))
```

title	release_year	details	genres	rating
Alice in Wonderland	2010	null	null	6.5

- Here we have saved the data to table **favorite_movies** and the data goes to columns 'title','release_year' and 'rating'.

Saving an RDD with Case Class Objects

genres: RDD[GenresInfo]

```
case class GenresInfo (title: String, releaseYear: Int, genres: Set[String])

val genres = sc.parallelize(Seq(
                new GenresInfo("Alice in Wonderland",2010,Set("Adventure","Family"))
               ))

genres.saveToCassandra("killr_video","favorite_movies",
                       SomeColumns("title","release_year","genres"))
```

title	release_year	details	genres	rating
Alice in Wonderland	2010	null	{'Adventure', 'Family'}	6.5

Saving an RDD with Case Class Objects

genres: RDD[GenresInfo]

```
case class GenresInfo (title: String, releaseYear: Int, genres: Set[String])

val genres = sc.parallelize(Seq(
                new GenresInfo("Alice in Wonderland",2010,Set("Adventure","Family"))
               ))
```

Saving an RDD with Case Class Objects

genres: RDD[GenresInfo]

```
case class GenresInfo (title: String, releaseYear: Int, genres: Set[String])

val genres = sc.parallelize(Seq(
                new GenresInfo("Alice in Wonderland",2010,Set("Adventure","Family"))
               ))

genres.saveToCassandra("killr_video","favorite_movies",
                       SomeColumns("title","release_year","genres"))
```

title	release_year	details	genres	rating
Alice in Wonderland	2010	null	{'Adventure', 'Family'}	6.5

- We have to convert the object in to **Seq() ,** so that we can parallelize.
- **Seq** are collections which have fixed order.

Saving an RDD with Tuples

details: RDD[(String, Int, UDTValue)]

```
val details = sc.parallelize(Seq(
                 ("Alice in Wonderland",2010,
                  UDTValue.fromMap(Map("country"->"USA","language"->"English","runtime"->108)))
                ))

details.saveToCassandra("killr_video","favorite_movies",
                        SomeColumns("title","release_year","details"))
```

title	release_year	details	genres	rating
Alice in Wonderland	2010	{country: 'USA', language: 'English', runtime: 108}	{'Adventure', 'Family'}	6.5

- For tuples also we have to Sequatialize before parallelizing, but here as we have created our own UDT, we have to also map, after which we can simply save as we did with objects.