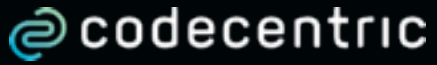


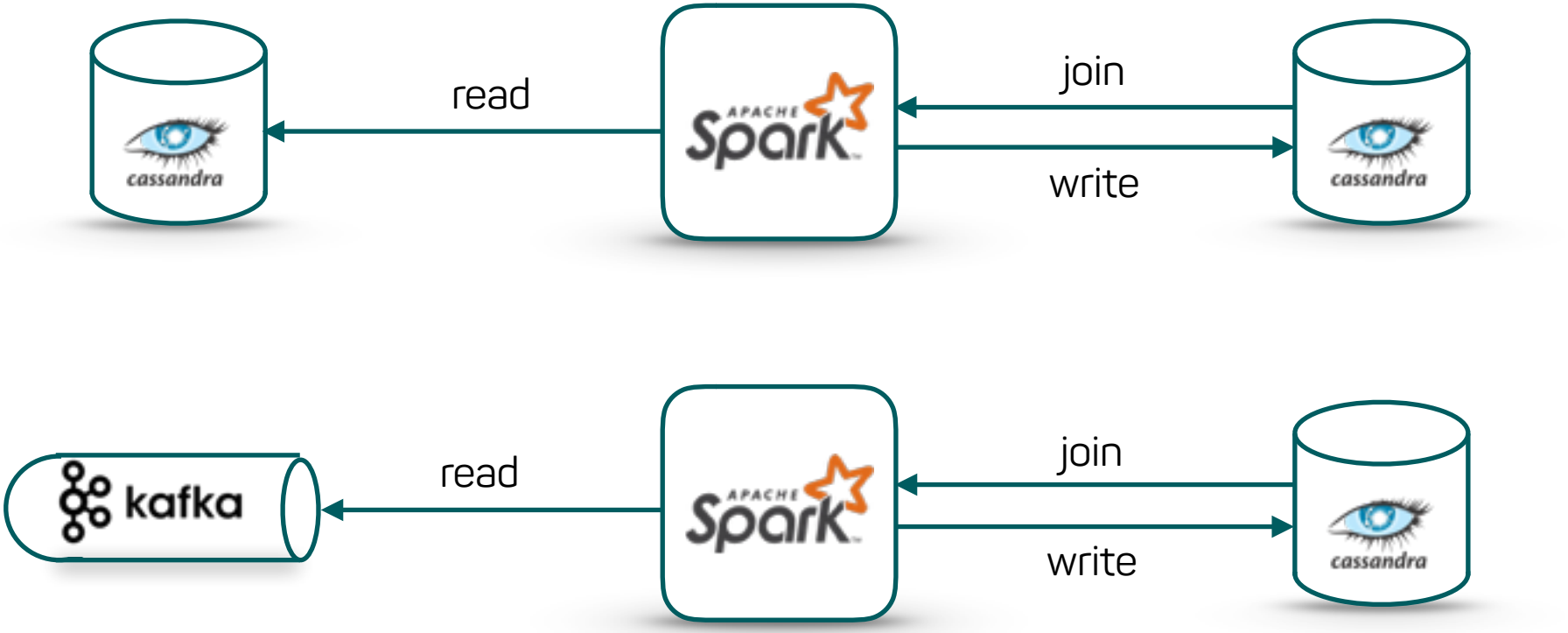
Lessons Learned with Cassandra & Spark_

Matthias Niehoff

Strata Data London 2017

@matthiasniehoff
@codecentric

Our Use Cases_



Lessons Learned with Cassandra

Data Modeling: Primary key_

- Primary key defines access to a table
  - *efficient access only by key*
  - *reading one or multiple entries by key*
- C*annot* be changed after creation

- Need to query by another key
  → create a new table
- Need to query by a lot of different keys
  → Cassandra might not be a got fit

Data Modeling: Care about bucketing_

- Strategy to reduce partition size
- Becomes part of the partition key
- Must be easily calculable for querying
- Aim for even sized partitions
- Do the math for partition sizes!
  - *value count*
  - *size in bytes*

flickr, joeshlabotnik

Data Modeling: Deletions_

- Well known:
  If you delete a column or whole row,
  the data is not really deleted
  Rather a tombstone is created to mark the deletion

- Tombstones are removed during compactions

Unexpected Tombstones: Built-in Maps, Lists, Sets_

- Inserts / Updates on collections
- Frozen collections
  - *treats collection as one big blob*
  - *no tombstones on insert*
  - *does not support field updates*
- Non frozen collections
  - *incr. inserts/appends w/o tombstones*
  - *tombstones for every other update/insert*

flickr, ndrwfgg

Debug Tool: sstable2json_

- sstable2json shows sstable file in json format
- Usage: go to /var/lib/cassandra/data/keyspace/table
- > sstable2json *-Data.db
- See the individual rows of the data files
- sstabledump in 3.6

Example_

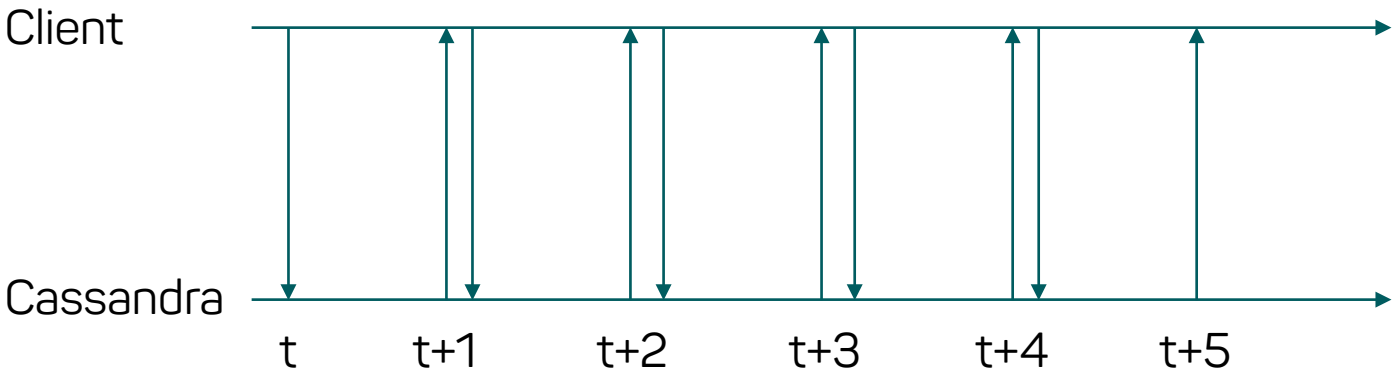
```
 name | status
------+--------
   ru | ACTIVE
   es | ACTIVE
   jp | ACTIVE
   vn | ACTIVE
   pl | ACTIVE
   cz | ACTIVE
```

```
{"key": "ru",
 "cells": [["status","ACTIVE",1464344127007511]]},
{"key": "it",
 "cells": [[„status“,"ACTIVE",1464344146457930, T]]},
{"key": "de",
 "cells": [["status","ACTIVE",1464343910541463]]},
{"key": "ro",
 "cells": [["status","ACTIVE",1464344151160601]]},
{"key": "fr",
 "cells": [["status","ACTIVE",1464344072061135]]},
{"key": "cn",
 "cells": [["status","ACTIVE",1464344083085247]]},
{"key": "kz",
 "cells": [["status","ACTIVE",1467190714345185]]}
```

deletion marker

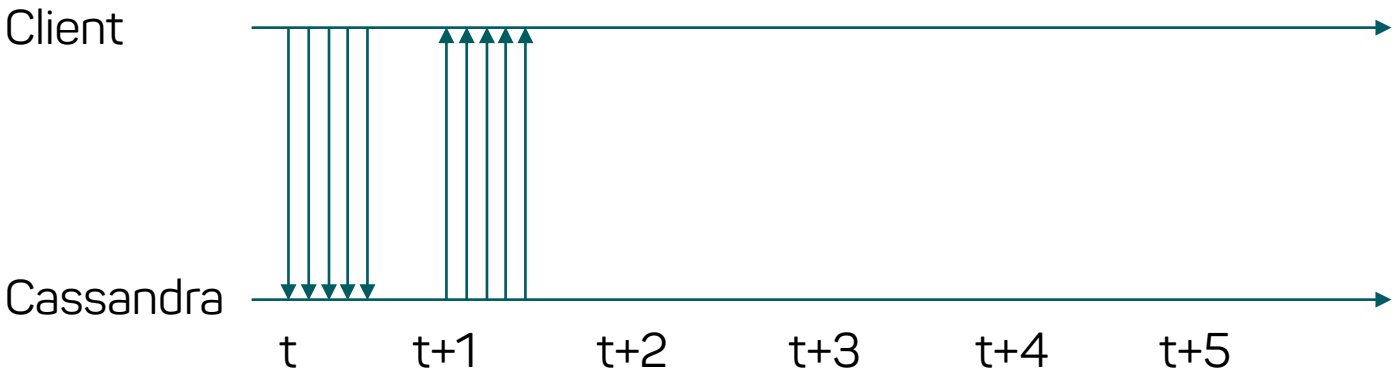
Bulk Reads or Writes_

- synchronous query introduce unnecessary delay



Bulk Reads or Writes: Async_

- parallel async queries



Example_

```
Session session = cc.openSession();
PreparedStatement getEntries =
  session.prepare("SELECT * FROM keyspace.table WHERE key=?");

private List<ResultSetFuture> sendQueries(Collection<String> keys) {
 List<ResultSetFuture> futures =
     Lists.newArrayListWithExpectedSize(keys.size());
  for (String key : keys {
   futures.add(session.executeAsync(getEntries.bind(key)));
  }
  return futures;
}
```

Example_

```
private void processAsyncResults(List<ResultSetFuture> futures)
{
  for (ListenableFuture<ResultSet> future :
   Futures.inCompletionOrder(futures)) {
     ResultSet rs = future.get();
       if (rs.getAvailableWithoutFetching() > 0 ||
           rs.one() != null) {
           // do your program logic here
       }
  }
}
```

Separating Data of Different Tenants_

- One cluster per tenant?
- One keyspace per tenant?
- One table per tenant?
- One table for all?

- Table per tenant (shared keyspace)
- Feasible only for limited number of tenants (~1000)

Monitoring_

- Switch on monitoring
- ELK, OpsCenter, self built, ...
- Avoid Log level *debug* for C* messages
  - *Drowning in irrelevant messages*
  - *Substantial performance drawback*
- Log level *info* for development, pre-production
- Log level *error* in production sufficient

Monitoring: Disk Space_

- Cassandra never checks if there is enough space left on disk for writing
- Keeps writing data till the disk is full
- Can bring the OS to a halt
- Cassandra error messages are confusing at this point
- Thus monitoring disk space is *mandatory*

Monitoring: Disk Space_

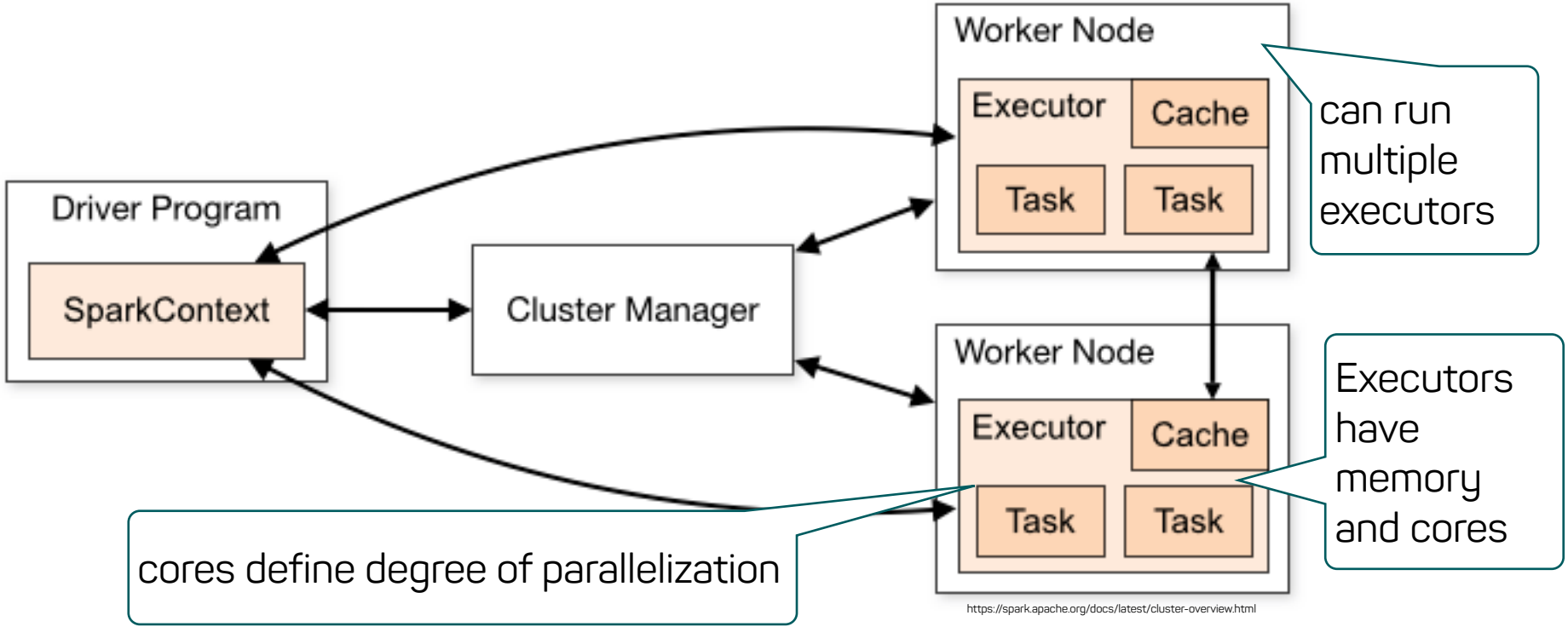
- A lot of disk space is required for compaction
- I.e. for SizeTieredCompaction up to 50% free disk space is needed
  - *Set-up monitoring on disk space*
  - *Alert if the data carrying disk partition fills up to 50%*
  - *Add nodes to the cluster and rebalance*

Lessons Learned with

Spark (Streaming)

codecentric

Quick Recap - Spark Resources_



Scaling Spark_

- Resource allocation is static per application
- Streaming jobs need fixed resources over a long time
- Unused resource for the driver
- Overestimate resources for peek load

Scaling - Overallocating_

- Spark Core is just a logical abstraction
- Microbatches idle most of the time

- Monitor resource when overusing CPUs!
- Leave space for temporary glitches

pic by wikipedia

Use Back Pressure Mechanism_

- Bursts off data increase processing time
- May result in OOM

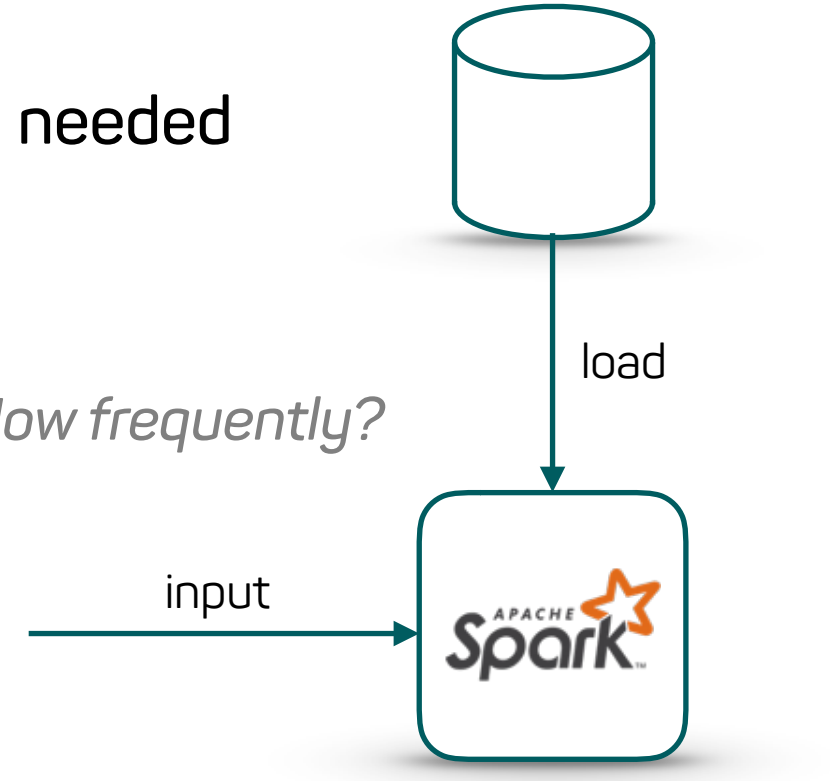
```
spark.streaming.backpressure.enabled

spark.streaming.backpressure.initialRate

spark.streaming.kafka.maxRatePerPartition
```

Lookup Additional Data_

- In batch: just load it, when needed
- In streaming:
  - *Long running application*
  - *Is the data static?*
  - *Does it change over time? How frequently?*



Lookup Additional Data_

- Broadcast data
  - *Static data*
  - *Load once at the start of the application*
- Use mapPartitions()
  - *Connection & lookup for every partition*
  - *High load*
  - *Connection overhead*

Lookup Additional Data_

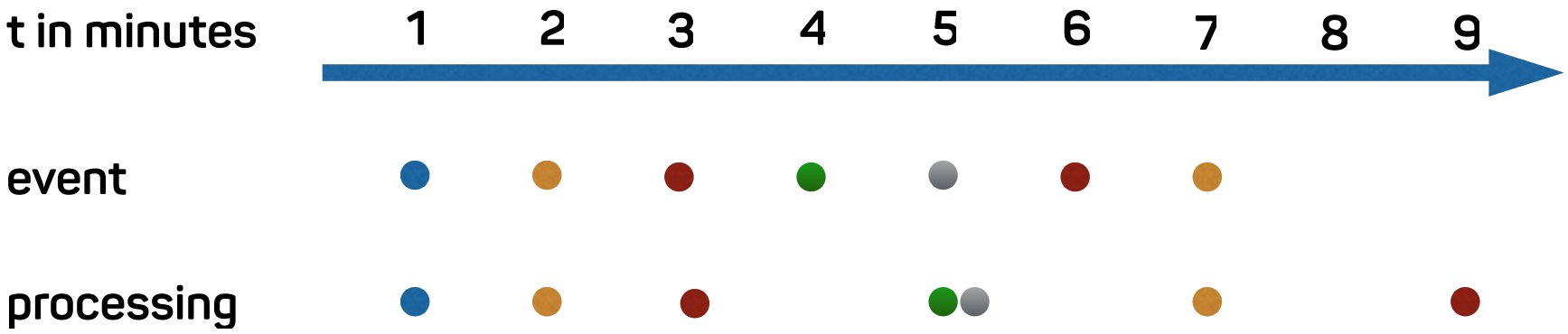
- Broadcast connection
  - *Lookup for every partition*
  - *Connection created once per executor*
  - *Still high load on datasource*
- mapWithState()
  - *Maintains keyed state*
  - *Initial state at application start*
  - *Technical messages trigger updates*
  - *Can only be used with key (no update all)*

Don’t hide the Spark UI_

Don't hide the Spark UI_

- Missing information, i.e. for streaming
- Crucial for debugging
- Do not build yourself!
  - *High frequency of events*
  - *Not all data available using REST API*
- Use the history server to see stopped/failed jobs

Event Time Support yet to come_



- Support starting with Spark 2.1
- Still alpha
- Concepts in place, implementation ongoing
- Solve some problems on your own, i.e. event time join

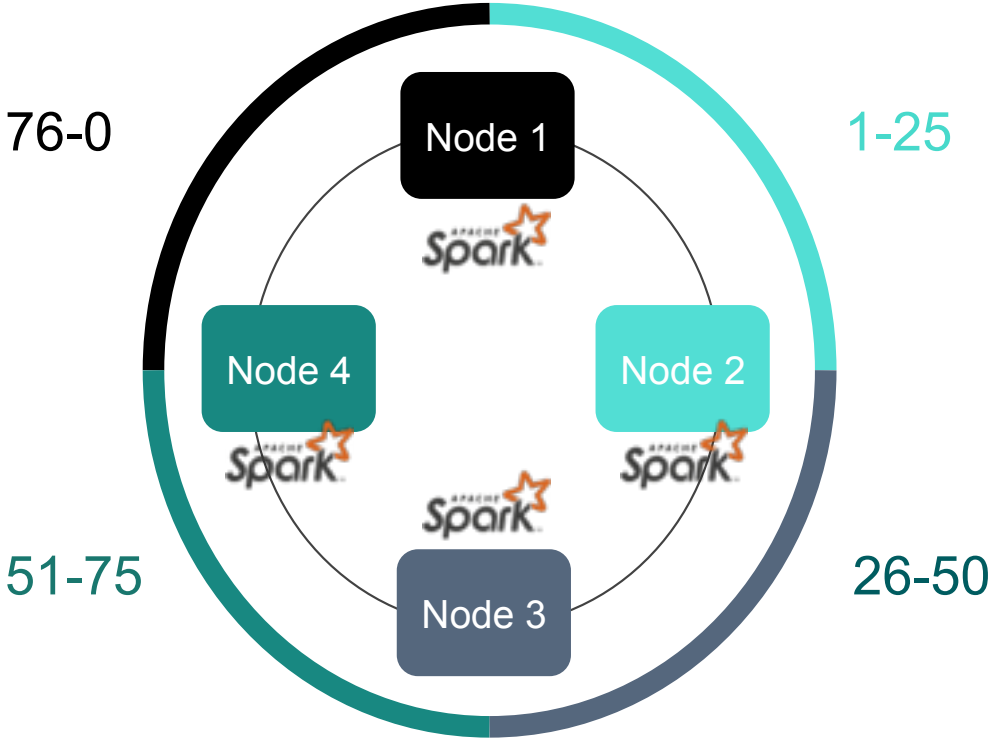
Operating Spark is not easy_

- First of all: it is distributed
- Centralized logging and monitoring
  - *Availability*
  - *Perfomance*
  - *Errors*
  - *System Load*
- Upgrade is tough

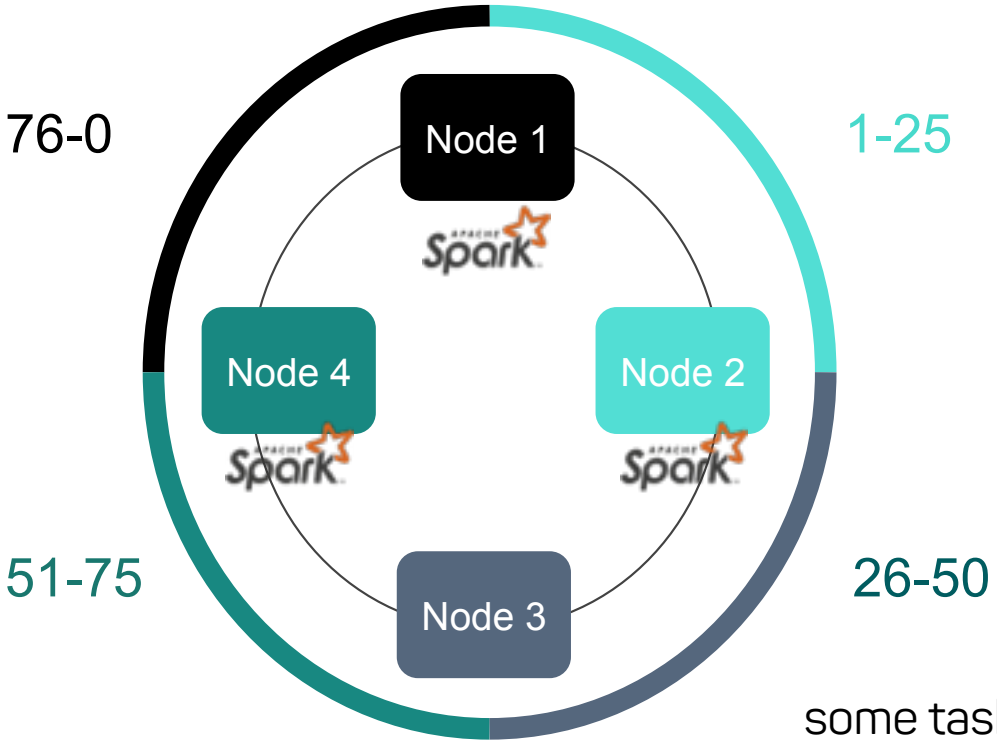
Lessons Learned with

Cassandra & Spark

repartitionByCassandraReplica_



repartitionByCassandraReplica_



some tasks took ~3s longer..

Spark locality_

- Watch for Spark Locality Level
  - *Aim for process or node local*
  - *Avoid any*

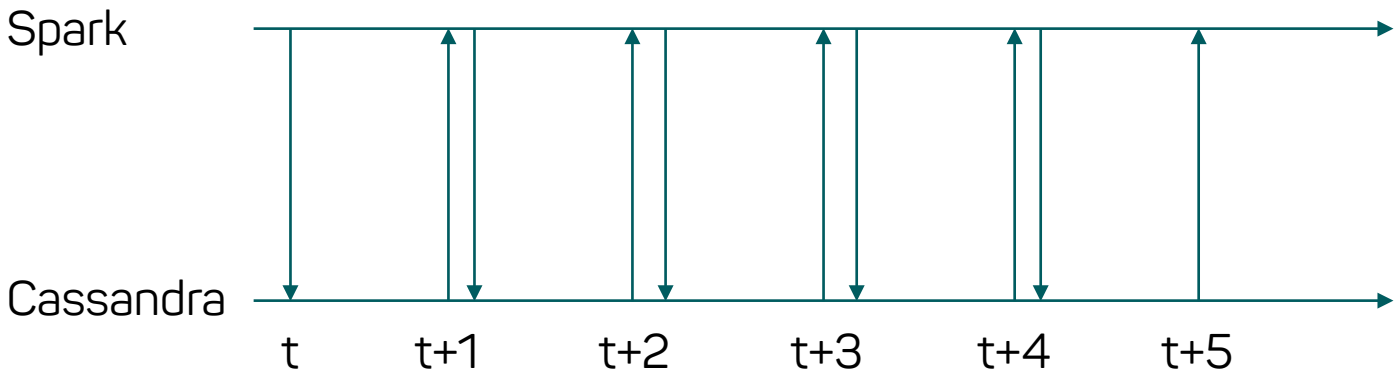
188	609	0	SUCCESS	PROCESS_LOCAL
189	610	0	RUNNING	ANY
190	611	0	RUNNING	ANY

```
spark.locality.wait 3s
```

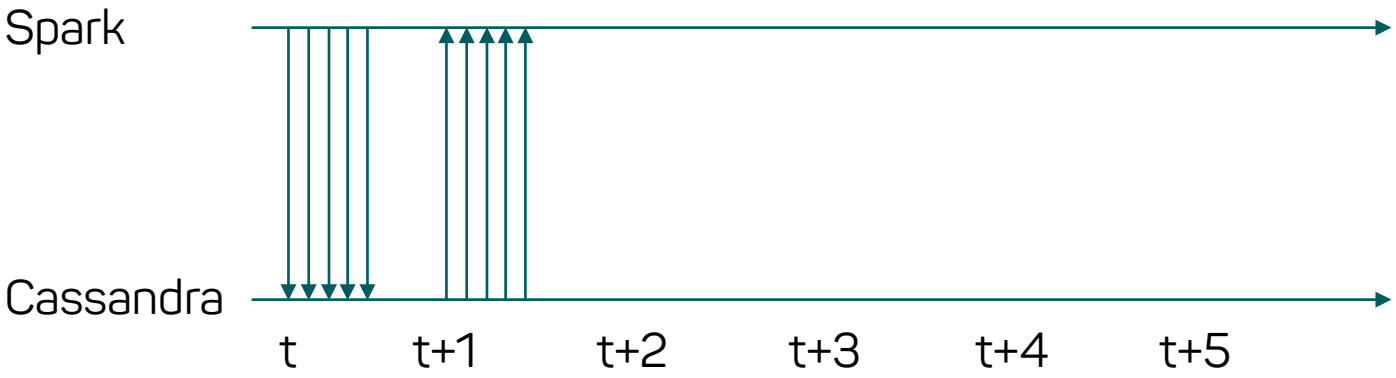
Do not use repartitionByCassandraReplica when ...

- Spark job does not run on every C* node
  - *# spark nodes < # cassandra nodes*
  - *# job cores < # cassandra nodes*
  - *spark job cores all on one node*
- time for repartition > time saving through locality

- One query per partition key
- One query at a time per executor



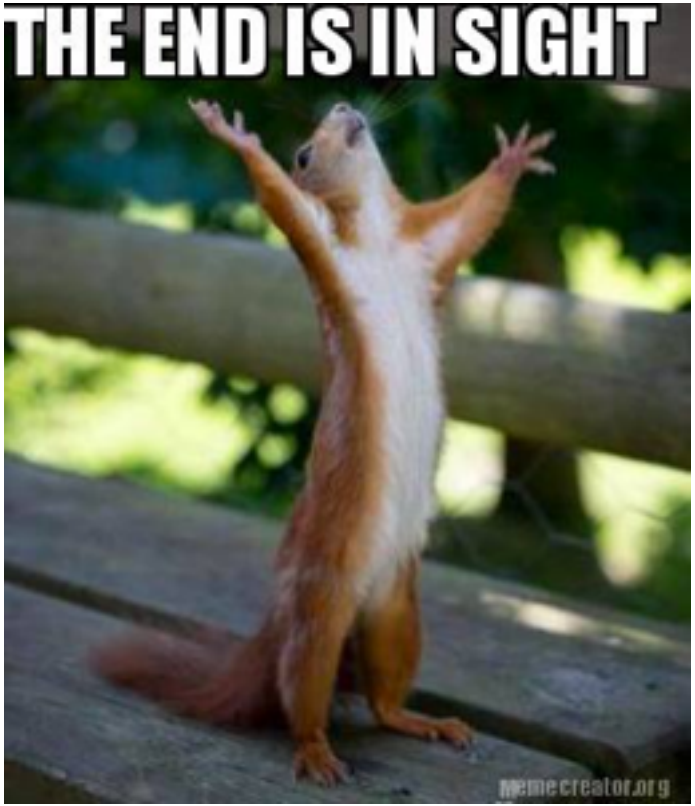
- Parallel async queries



- built a custom async implementation

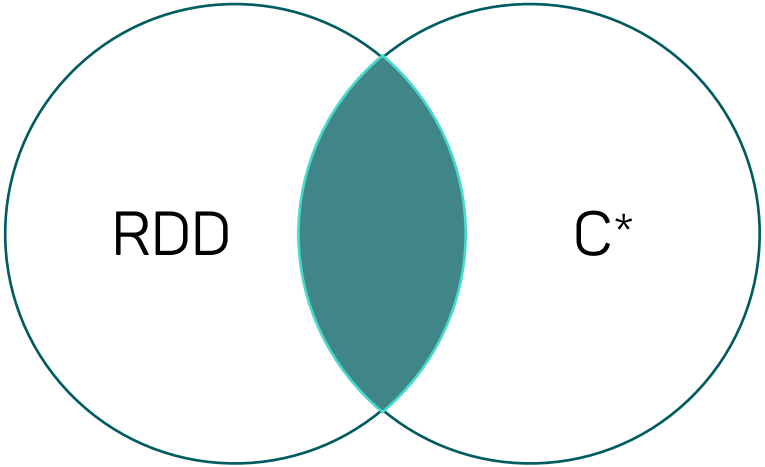
```
someDStream.transformToPair(rdd -> {
   return rdd.mapPartitionsToPair(iterator -> {
    ...
    Session session = cc.openSession()) {
    while (iterator.hasNext()) {
      ...
      session.executeAsync(..)
    }
    [collect futures]
    return List<Tuple2<Left,Right>>
  });
});
```

- Solved with SPARKC-233 (1.6.0 / 1.5.1 / 1.4.3)
- 5-6 times faster than sync implementation!

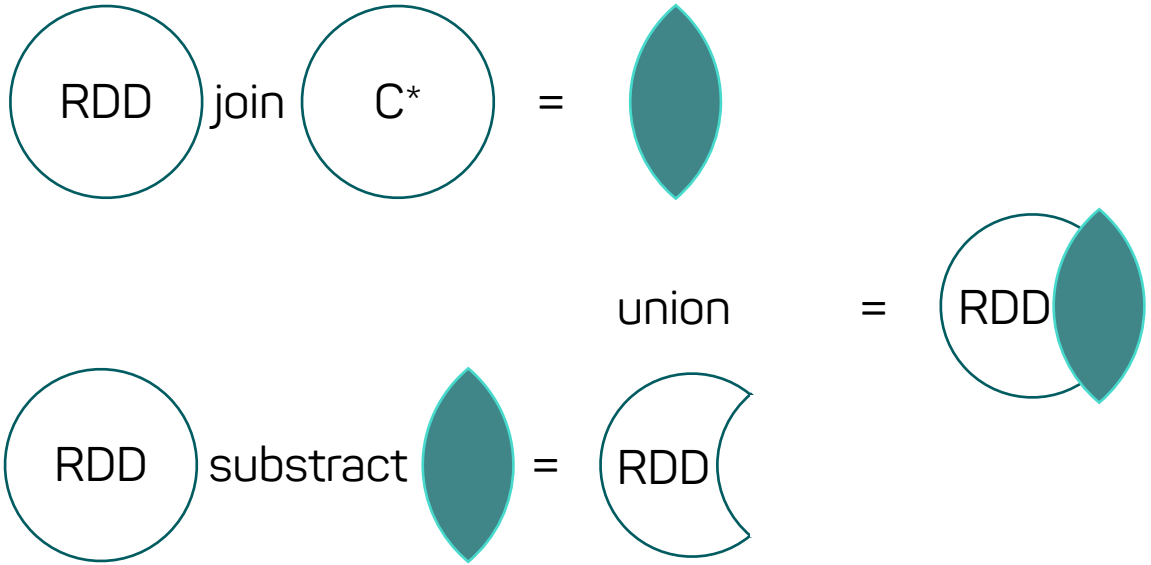


Left join with Cassandra_

- joinWithCassandraTable is a full inner join



Left join with Cassandra_



- Might include shuffle --> quite expensive

Left join with Cassandra_

- built a custom async implementation

```
someDStream.transformToPair(rdd -> {
   return rdd.mapPartitionsToPair(iterator -> {
    ...
    Session session = cc.openSession()) {
    while (iterator.hasNext()) {
      ...
      session.executeAsync(..)
      ...
    }
    [collect futures]
     return List<Tuple2<Left,Optional<Right>>>
  });
});
```

Left join with Cassandra_

- solved with SPARKC-1.81 (2.0.0)
- basically uses async joinWithC* implementation

Connection keep alive_

- spark.cassandra.connection.keep_alive_ms
- Default: 5s

- Streaming Batch Size > 5s
- Open Connection for every new batch

- Should be multiple times the streaming interval!

Cache! Not only for performance_

```
val changedStream = someDStream.map(e -> someMethod(e)).cache()
changedStream.saveToCassandra("keyspace","table1")
changedStream.saveToCassandra("keyspace","table1")

ChangedEntry someMethod(Entry e) {
  return new ChangedEntry(new Date(),...);
}
```

- Cache saves performance by preventing recalculation
- Sometimes necessary in regards to correctness!

Summary_

- Know the most important internals
- Know your tools
- Monitor your cluster
- Use existing knowledge resources
- Use the mailing lists
- Participate in the community

Questions?

Matthias Niehoff
IT-Consultant

codecentric AG
Hochstraße 11
42697 Solingen, Germany

matthias.niehoff@codecentric.de

www.codecentric.de
blog.codecentric.de

matthiasniehoff