Lessons Learned with Cassandra & Spark

Matthias Niehoff
Strata Data London 2017

@matthiasniehoff
@codecentric
Our Use Cases

1. Read from Cassandra → Join with Apache Spark → Write to Cassandra
2. Read from Kafka → Join with Apache Spark → Write to Cassandra
Lessons Learned with Cassandra
Primary key defines access to a table
- efficient access only by key
- reading one or multiple entries by key
- Cannot 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
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
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.
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
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
<table>
<thead>
<tr>
<th>name</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ru</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>es</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>jp</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>vn</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>pl</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>cz</td>
<td>ACTIVE</td>
</tr>
</tbody>
</table>

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

```java
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)
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
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
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
Quick Recap - Spark Resources

- Executors can run multiple executors
- Executors have memory and cores
- Cores define degree of parallelization

https://spark.apache.org/docs/latest/cluster-overview.html
Resource allocation is static per application
Streaming jobs need fixed resources over a long time
Unused resource for the driver
Overestimate resources for peak load
- Spark Core is just a logical abstraction
- Microbatches idle most of the time

Monitor resource when overusing CPUs!
Leave space for temporary glitches
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`
In batch: just load it, when needed

In streaming:
- Long running application
- Is the data static?
- Does it change over time? How frequently?
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

ONE DOES NOT SIMPLY

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
  - Performance
  - Errors
  - System Load
- Upgrade is tough
Lessons Learned with Cassandra & Spark
repartitionByCassandraReplica

Node 1

Node 2

Node 3

Node 4

76-0

1-25

51-75

26-50
some tasks took ~3s longer..
Spark locality

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

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>SUCCESS</th>
<th>PROCESS_LOCAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>188</td>
<td>609</td>
<td>0</td>
<td>SUCCESS</td>
<td>PROCESS_LOCAL</td>
</tr>
<tr>
<td>189</td>
<td>610</td>
<td>0</td>
<td>RUNNING</td>
<td>ANY</td>
</tr>
<tr>
<td>190</td>
<td>611</td>
<td>0</td>
<td>RUNNING</td>
<td>ANY</td>
</tr>
</tbody>
</table>
```

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

joinWithCassandraTable

Spark

Cassandra

<table>
<thead>
<tr>
<th>t</th>
<th>t+1</th>
<th>t+2</th>
<th>t+3</th>
<th>t+4</th>
<th>t+5</th>
</tr>
</thead>
</table>

codecentric
built a custom async implementation

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

- Solved with SPARKC-233 (1.6.0 / 1.5.1 / 1.4.3)
- 5-6 times faster than sync implementation!
JoinWithCassandraTable is a full inner join.
Left join with Cassandra

\[
\text{RDD} \quad \text{join} \quad C^* \quad = \quad \text{union} \quad = \quad \text{RDD}
\]

- Might include shuffle --> quite expensive
built a custom async implementation

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

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

```scala
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(),...);
}
```
Summary

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