Online Analytics with Hadoop and Cassandra

Robbie Strickland

who am i?

Robbie Strickland
Managing Architect, E3 Greentech

rostrickland@gmail.com
linkedin.com/in/robbiestrickland
github.com/rstrickland

in scope

- offline vs online analytics
- Cassandra overview
- Hadoop-Cassandra integration
- writing map/reduce against Cassandra
- Scala map/reduce
- Pig with Cassandra
- questions

common hadoop usage pattern

- **batch analysis** of unstructured/semi-structured data
- analysis of transactional data requires moving it into **HDFS or offline HBase** cluster
- use Sqoop, Flume, or some other **ETL** process to aggregate and pull in data from external sources
- well-suited for **high-latency offline analysis**

the online use case

- need to **analyze live data** from a transactional DB
- **too much data or too little time** to efficiently move it offline before analysis
- need to **immediately** feed results back into **live** system
- reduce complexity/improve maintainability of M/R jobs
- run **Pig scripts against live data** (i.e. data exploration)

Cassandra + Hadoop makes this possible!

what is cassandra?

- open-source NoSQL **column store**
- **distributed** hash table
- tunable **consistency**
- dynamic **linear scalability**
- masterless, **peer-to-peer** architecture
- highly **fault tolerant**
- extremely **low latency** reads/writes
- keeps active data set in RAM -- tunable
- all writes are **sequential**
- automatic **replication**

cassandra data model

- **keyspace** roughly == database
- **column family** roughly == table
- each **row** has a **key**
- each row is a collection of **columns** or **supercolumns**
- a **column** is a key/value pair
- a **supercolumn** is a collection of columns
- column names/quantity do not have to be the same across rows in the same column family
- keys & column names are **often composite**
- supports **secondary indexes** with low cardinality

cassandra data model

keyspace
  standard_column_family
    key1
      column1 : value1
      column2 : value2
    key2
      column1 : value1
      column3 : value3
      column4 : value4

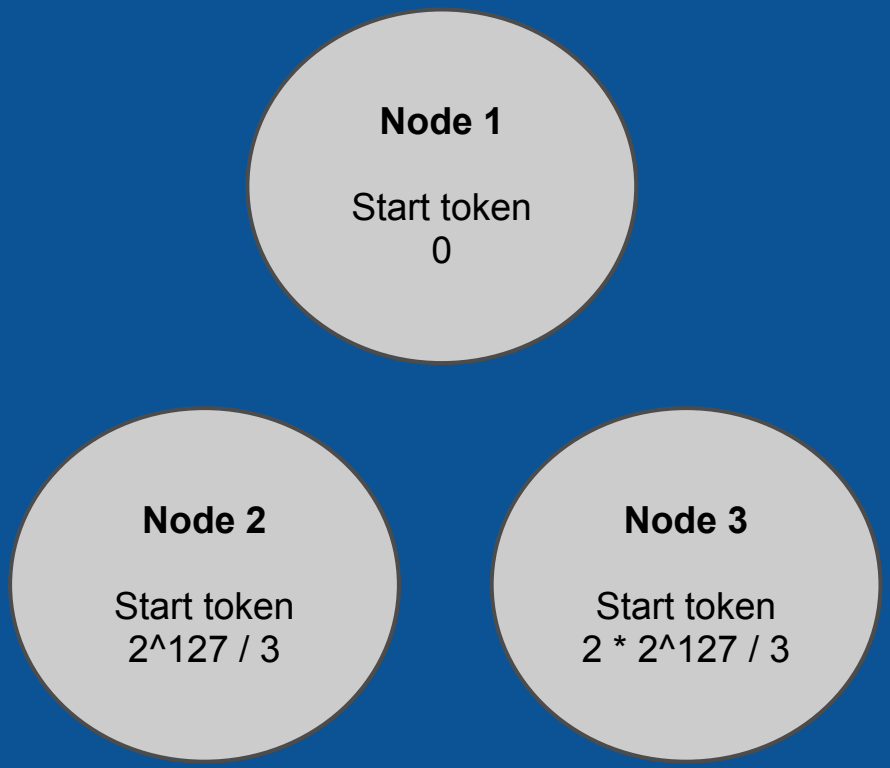
keyspace
  super_column_family
    key1
      supercol1
        column1 : value1
        column2 : value2
      supercol2
        column3 : value3
    key2
      supercol3
        column4 : value4
      supercol4
        column5 : value5

let's have a look ...

cassandra query model

- get by **key(s)**
- **range slice** - from key A to key B in sort order (partitioner choice matters here)
- **column slice** - from column A to column B ordered according to specified comparator type
- **indexed slice** - all rows that match a predicate (requires a secondary index to be in place for the column in the predicate)
- "**write it like you intend to read it**"
- Cassandra Query Language (**CQL**)

cassandra topology



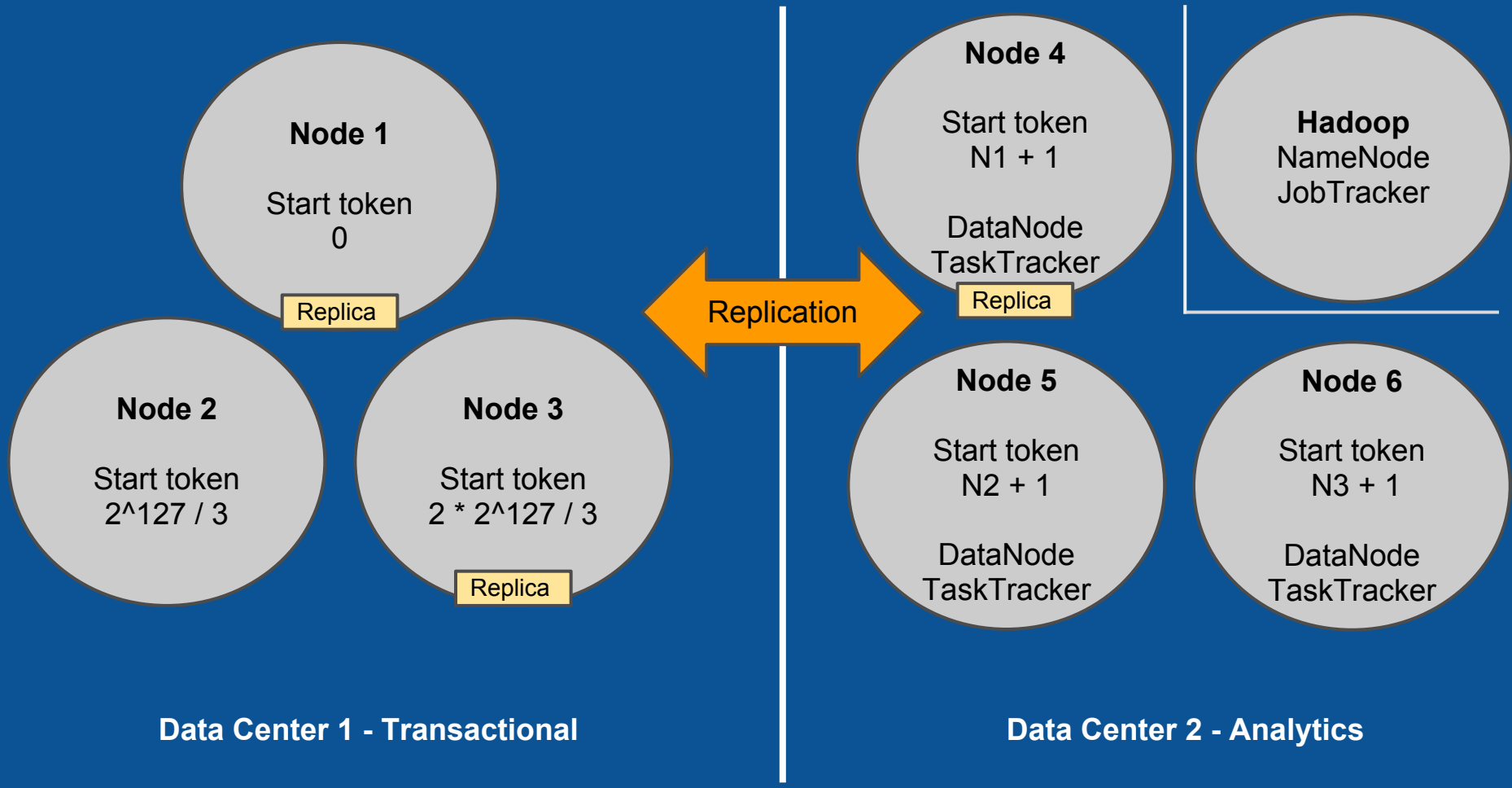
cassandra client access

- uses **Thrift** as the underlying protocol with a number of language bindings
- **CLI** is useful for simple CRUD and schema changes
- **high-level clients** available for most common languages -- this is the recommended approach
- **CQL** supported by some clients

cassandra with hadoop

- supports input via **ColumnFamilyInputFormat**
- supports output via **ColumnFamilyOutputFormat**
- HDFS still used by Hadoop unless you use DataStax Enterprise version
- supports **Pig** input/output via **CassandraStorage**
- preserves **data locality**
- use **2 "data centers"** -- one for transactional and the other for analytics
- **replication** between data centers is automatic and immediate

cassandra with hadoop - topology



configuring transactional nodes

- install **Cassandra 1.1**
- config properties are in **cassandra.yaml** (in /etc/cassandra or $CASSANDRA_HOME/conf) and in **create keyspace** options
- placement strategy = **NetworkTopologyStrategy**
- **strategy options** = DC1:2, DC2:1
- endpoint snitch = **PropertyFileSnitch** -- put DC/rack specs in cassandra-topology.properties
- designate at least one node as **seed(s)**
- make sure **listen & rpc addresses** are correct
- use **nodetool** -h <host> ring to verify everyone is talking

configuring analytics nodes

- configure Cassandra using transactional instructions
- insure there are **no token collisions**
- install **Hadoop 1.0.1+ TaskTracker & DataNode**
- add **Cassandra jars & dependencies** to Hadoop classpath -- either copy them to $HADOOP_HOME/lib or add the path in hadoop-env.sh
- Cassandra jars are typically in /usr/share/cassandra, and dependencies in /usr/share/cassandra/lib if installed using a package manager
- you may need to increase Cassandra **RPC timeout**

let's have a look ...

writing map/reduce

Assumptions:

- Hadoop 1.0.1+
- new Hadoop API (i.e. the one in org.apache.hadoop.mapreduce, not mapred)
- Cassandra 1.1

job configuration

- use Hadoop job configuration API to set general Hadoop configuration parameters
- use Cassandra ConfigHelper to set Cassandra-specific configuration parameters

job configuration - example

Hadoop config related to Cassandra:

```
job.setInputFormatClass(ColumnFamilyInputFormat.class)
job.setOutputFormatClass(ColumnFamilyOutputFormat.class)
job.setOutputKeyClass(ByteBuffer.class)
job.setOutputValueClass(List.class)
```

job configuration - example

Cassandra-specific config:

```
ConfigHelper.setInputRpcPort(conf, "9160")
ConfigHelper.setInputInitialAddress(conf, cassHost)
ConfigHelper.setInputPartitioner(conf, partitionerClassName)
ConfigHelper.setInputColumnFamily(conf, keyspace, columnFamily)
ConfigHelper.setInputSlicePredicate(conf, someQueryPredicate)
ConfigHelper.setOutputRpcPort(conf, "9160")
ConfigHelper.setOutputInitialAddress(conf, cassHost)
ConfigHelper.setOutputPartitioner(conf, partitionerClassName)
ConfigHelper.setOutputColumnFamily(conf, keyspace, columnFamily)
```

the mapper

- input key is a **ByteBuffer**
- input value is a **SortedMap<ByteBuffer, IColumn>**, where the ByteBuffer is the column name
- use Cassandra **ByteBufferUtil** to read/write ByteBuffers
- map output must be a **Writable** as in "standard" M/R

example mapper

Group integer values by column name:

```
for (IColumn col : columns.values())
    context.write(new Text(ByteBufferUtil.string(col.name())),
                  new LongWritable(ByteBufferUtil.toLong(col.value())));
```

the reducer

- input is whatever you output from map
- output key is ByteBuffer == row key in Cassandra
- output value is List<Mutation> == the columns you want to change
- uses Thrift API to define Mutations

example reducer

Compute average of each column's values:

```
int sum = 0;
int count = 0;
for (LongWritable val : values) {
    sum += val.get();
    count++;
}

Column c = new Column();
c.setName(ByteBufferUtil.bytes("Average"));
c.setValue(ByteBufferUtil.bytes(sum/count));
c.setTimestamp(System.currentTimeMillis());

Mutation m = new Mutation();
m.setColumn_or_supercolumn(new ColumnOrSuperColumn());
m.column_or_supercolumn.setColumn(c);
context.write(ByteBufferUtil.bytes(key.toString()), Collections.singletonList(m));
```

let's have a look ...

outputting to multiple CFs

- currently not supported out of the box
- patch available against 1.1 to allow this using Hadoop MultipleOutputs API
- not necessary to patch production Cassandra, only the Cassandra jars on Hadoop's classpath
- patch will not break existing reducers
- to use:
  - git clone Cassandra 1.1 source
  - use git apply to apply patch
  - build with ant
  - put the new jars on Hadoop classpath

multiple output example

```
// in job configuration
ConfigHelper.setOutputKeyspace(conf, keyspace);

MultipleOutputs.addNamedOutput(job, columnFamily1,
    ColumnFamilyOutputFormat.class, ByteBuffer.class,
    List.class);

MultipleOutputs.addNamedOutput(job, columnFamily2,
    ColumnFamilyOutputFormat.class, ByteBuffer.class,
    List.class);
```

multiple output example

```
// in reducer
private MultipleOutputs _output;

public void setup(Context context) {
    _output = new MultipleOutputs(context);
}

public void cleanup(Context context) {
    _output.close();
}
```

multiple output example

```
// in reducer - writing to an output

Mutation m1 = ...
Mutation m2 = ...

_output.write(columnFamily1, ByteBufferUtil.bytes(key1), m1);
_output.write(columnFamily2, ByteBufferUtil.bytes(key2), m2);
```

let's have a look ...

map/reduce in scala

why? because **this**:

```
values.groupBy(_.name)
        .map { case (name, vals) => name -> vals.map(ByteBufferUtil.toInt).sum }
        .filter { case (name, sum) => sum > 100 }
        .foreach { case (name, sum) =>
          context.write(new Text(ByteBufferUtil.string(name)), new IntWritable(sum))
        }
```

... would require **many more lines of code** in Java and be **less readable**!

map/reduce in scala

... and because we can parallelize it by doing this:

```
values.par  // make this collection parallel
        .groupBy(_.name)
        .map { case (name, vals) => name -> vals.map(ByteBufferUtil.toInt).sum }
        .filter { case (name, sum) => sum > 100 }
        .foreach { case (name, sum) =>
          context.write(new Text(ByteBufferUtil.string(name)), new IntWritable(sum))
        }
```

(!!!)

map/reduce in scala

- add **Scala library** to Hadoop classpath
- import scala.collection.JavaConversions._ to get Scala coolness with Java collections
- main executable is an **empty class** with the implementation in a **companion object**
- **context parm** in map & reduce methods must be:
  - context: Mapper[ByteBuffer, SortedMap[ByteBuffer, IColumn], <MapKeyClass>, <MapValueClass>]#Context
  - context: Reducer[<MapKeyClass>, <MapValueClass>, ByteBuffer, java.util.List[Mutation]]#Context
  - it will compile but not function correctly without the type information

let's have a look ...

pig

- allows **ad hoc queries** against Cassandra
- schema applied at query time
- can run in **local** mode or using **Hadoop cluster**
- useful for data exploration, mass updates, back-populating data, etc.
- cassandra supported via **CassandraStorage**
- both **LoadFunc** and **StoreFunc**
- now comes **pre-built** in Cassandra distribution
- **pygmalion** adds some useful Cassandra UDFs -- github.com/jeromatron/pygmalion

pig - getting set up

- **download pig** - better not to use package manager
- get the **pig_cassandra** script from Cassandra source (it's in examples/pig) -- or I put it on github with all the samples
- set **environment variables** as described in README
  - JAVA_HOME = location of your Java install
  - PIG_HOME = location of your Pig install
  - PIG_CONF_DIR = location of your Hadoop config files
  - PIG_INITIAL_ADDRESS = address of one Cassandra analytics node
  - PIG_RPC_PORT = Thrift port -- default 9160
  - PIG_PARTITIONER = your Cassandra partitioner

pig - loading data

Standard CF:

```
rows = LOAD 'cassandra://<keyspace>/<column_family>'
        USING CassandraStorage()
        AS (key: bytearray, cols: bag {col: tuple(name,value)});
```

Super CF:

```
rows = LOAD 'cassandra://<keyspace>/<column_family>'
        USING CassandraStorage()
        AS (key: bytearray, cols: bag {(supercol, subcols: bag {(name,
value)})});
```

pig - writing data

STORE command:

STORE <relation_name>
INTO 'cassandra://<keyspace>/<column_family>'
USING CassandraStorage();

Standard CF schema:

(key, {(col_name, value), (col_name, value)});

Super CF schema:

(key, {supercol: {(col_name, value), (col_name, value)},
{supercol: {(col_name, value), (col_name, value)}});

let's have a look ...

questions?

check out my github for this presentation and code samples!

Robbie Strickland
Managing Architect, E3 Greentech

rostrickland@gmail.com
linkedin.com/in/robbiestrickland
github.com/rstrickland