ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Spark Program

CHAPTER 5: SPARK AND NOSQL

Chapter Objectives

In this chapter, we will:

- Discuss NoSQL
- Introduce Cassandra and Mongo
- Use Python to access Cassandra and Mongo directly
- Use Spark to read and write to Cassandra and Mongo

Chapter Concepts

NoSQL

Cassandra

MongoDB

Chapter Summary

NoSQL Concept

- As the volume of transactions increases, a single SQL server can no longer handle the amount of transactions
- Scaling out to a multi-node cluster is the only way to handle the large volumes
- Compromises need to be made to allow for more transaction volume
  - Cannot have ACID properties
  - Since there are multiple redundant copies of the data on different nodes, it's possible that there could be an inconsistency in their values at a given moment
  - Systems are optimized for input, update, delete performance; select is limited to fetching by key
  - Long batch, warehouse, and analytical queries just don't perform well and features like JOIN and GROUP BY don't exist
- In order to run analytical queries, you need a different processing engine like Spark to handle them, while the NoSQL engine focuses only on the individual update transactions

NoSQL and Spark

- Spark and NoSQL such as Cassandra, Mongo, and HBase make a powerful combination
- NoSQL can be used to acquire real-time data in a transactional system
- Spark can be used to query the data stored in NoSQL in ways that cannot be done with the native NoSQL engine
- Most NoSQL does not support complex analytical queries like JOIN, GROUP, ORDER BY, etc.
- Using Spark to read data stored in NoSQL creates a combination that offers many of the features of a traditional SQL server, but with the power of a cluster to handle large volumes

Chapter Concepts

NoSQL

Cassandra

MongoDB

Chapter Summary

Cassandra Characteristics

- Distributed and decentralized
  - Runs on a cluster of machines
  - Potentially across many data centers
  - Every node is equal
    - No Master in architecture
  - Decentralized model provides high resilience
    - No single points of failure
- Elastic scalability
  - Cassandra scales horizontally
    - New nodes are automatically discovered
    - Work will be sent to the new servers
    - Including rebalancing of data across nodes
- High availability and fault tolerance
  - Cluster continues serving in the event of a failed node
  - New nodes can be added with no downtime
  - Replication can be over data centers
    - Improve performance with local access to data
    - Resilience in the event of data center unavailability

Cassandra Characteristics (continued)

- Tunable consistency
  - Consistency refers to a read being able to return the very latest write
  - Consistency can be tuned on Cassandra
    - Eventual consistency
    - Strict consistency
    - Causal consistency
- Column-oriented database
  - Data is stored in rows
  - Rows can be sparse
  - Each column can have multiple values
  - Can be viewed as a multi-dimensional hash table
  - No joins

CQLSH

- The Cassandra Query Language (CQL) is the language used to define and manipulate data in the Cassandra database
- Cassandra provides several ways to invoke CQL statements
  - Programming language drivers that use the CQL binary protocol
    - Very performant
    - Java, Python, JavaScript, C++, and others
  - Interactive CQL shell (CQLSH)
- **Do Now!**
  - Open a terminal window
  - From the terminal window, enter the following command: `cqlsh`
    - This will change the prompt to `cqlsh>`

Creating a Keyspace

- From the `cqlsh>` prompt, enter the following commands:

```
cqlsh> create keyspace classroom with
replication={'class':'SimpleStrategy',
'replication_factor':'1'};

cqlsh> describe keyspaces

cqlsh> describe keyspace classroom
```

Creating a Table

- From the `cqlsh:classroom>` prompt, enter the following commands:

```
cqlsh:classroom> describe tables;

cqlsh:classroom> create table student (id int PRIMARY KEY,
first_name text, last_name text, email_addresses
set<text>);

cqlsh:classroom> describe tables;

cqlsh:classroom> describe table student;
```

Writing and Reading Data

- From the `cqlsh:classroom>` prompt, enter the following commands:

```
cqlsh:classroom> insert into student
             ... (id, first_name, last_name, email_addresses)
             ... values
             ... (1, 'Joe', 'Smith',
             ... {'joes@xyz.com',
             ... 'joe.smith@some_univ.edu'});

cqlsh:classroom> select * from student;
```

Python and Cassandra

- Python can directly talk to Cassandra using the `cassandra-driver` package
  - `pip install cassandra-driver`

```
from cassandra.cluster import Cluster
cluster = Cluster(['127.0.0.1'])
session = cluster.connect()
session.execute("update student set firstname = 'Joseph'
where id = 1")
session.execute("insert into student (id, firstname,
lastname, emails) values (2, 'Mike', 'Jones',
{'mikej@xyz.com', 'mike.jones@def.net',
'mike1234@gmail.com'})")
rows = session.execute('SELECT id, firstname, lastname,
emails from student')
print(list(rows))
```

PySpark and Cassandra

- PySpark can directly talk to Cassandra using a built-in class library
- Must start PySpark with a special parameter
  - `pyspark --packages com.datastax.spark:spark-cassandra-connector_2.11:2.4.1`

```
people = spark.read.format("org.apache.spark.sql.cassandra"\
    .options(table="student", keyspace="classroom").load()
people.show()
```

```
x = sc.parallelize([(3, 'Mary', 'Johnson', \
     ['Mary1@gmail.com', 'Mary2@yahoo.com'])])
x1 = spark.createDataFrame(x, \
     schema = ['id', 'firstname', 'lastname', 'emails'])
x1.write.format("org.apache.spark.sql.cassandra")\
     .options(table="student", keyspace="classroom").\
     mode("append").save()
```

Chapter Concepts

NoSQL

Cassandra

MongoDB

Chapter Summary

Document Data Stores

- Idea of document data stores is to replace concept of row with more flexible model
  - No fixed schema, every record potentially different
  - Complex relationships stored as simple document
    - All data for the document stored in the one document
    - Aim for self-contained documents
  - Documents can be queried on contents
- Many document data stores are available
  - CouchDB
  - MongoDB
  - Terrastore
  - RavenDB

MongoDB

- Document database with document data field/value pairs
  - Documents are JSON objects
  - Values in a field can be other documents
  - Rich data model not constrained by schema
- Key features include:
  - Easy scaling by scale out (horizontal scalability)
  - Replication and high availability
  - Rich querying
  - Flexible aggregation and data processing including by Hadoop
- Has an API with bindings for many languages
  - Also comes with interactive shell

MongoDB Key Concepts

- Has the concept of a database
  - Within a MongoDB instance it's possible to have multiple databases
- Database stores documents in collections
  - Collections are similar to tables—but no schema
- Collections are made up of documents
  - Documents are equivalent of records/rows in relational database
- Documents are made up of fields
  - Similar to columns
    - But fields can themselves be documents allowing nested documents
- MongoDB has indexes
- MongoDB uses Cursors
  - Enable counting, forwarding, etc. without fetching data in document

Working with MongoDB

- MongoDB enables creation of databases
  - Within a database documents are grouped into collections
- Document is an ordered set of keys with associated values
- Following examples will give a feel for working with MongoDB

```
use productdb

product = {"manufacturer": "Bosch", "price": 199.99, "retailer": 199}

db.products.insert(product)
```

Switch to `productdb` database

Create document product

`db` refers to currently selected database

- MongoDB generates a unique id for each item added

Bulk Inserts

- Multiple documents can be inserted with one insert
  - Known as batch insert
  - Can only be used for inserting into one collection at a time

```
use productdb

products = [
{"manufacturer" : "Bosch", "price" : 199.99, "retailer" : 199},
{"manufacturer" : "AEG", "price" : 179.99, "retailer": 179},
…
]

db.products.insert(products)

Object.bsonsize(products)
```

Array of records

Now a bulk insert

Storage size of document

Python and MongoDB

- Python can directly talk to MongoDB using the `pymongo` package
  - `pip install cassandra-driver`

```
import pymongo
client = pymongo.MongoClient("mongodb://127.0.0.1:27017/")
classroom = client["classroom"]
if 'classroom' in (x['name'] for x in
client.list_databases()):
    client.drop_database('classroom')

people = classroom['people']
name = {"firstname" : "Adam", "personid":4}
x = people.insert_one(name)
names = [{"firstname" : "Betty", "personid":5}
         ,{"firstname" : "Charlie", "personid":6}]
x = people.insert_many(names)
x = people.find()
print (list(x))
```

PySpark and MongoDB (continued)

- PySpark can directly talk to MongoDB using a built-in class library
- Must start PySpark with a special parameter
  - `pyspark --packages org.mongodb.spark:mongo-spark-connector_2.11:2.4.1`

```
spark = SparkSession.builder.appName("myApp") \
    .config("spark.mongodb.input.uri", \
    "mongodb://127.0.0.1/classroom") \
    .config("spark.mongodb.output.uri", \
    "mongodb://127.0.0.1/classroom").getOrCreate()

df = spark.read.format("mongo"). \
   option("uri", "mongodb://127.0.0.1/classroom.people") \
   .load()
df.show()
```

Write to MongoDB


```
x = sc.parallelize([(7, 'David')])
x1 = spark.createDataFrame(x, schema = ['personid',
'firstname'])
x1.write.format("mongo").options(collection="people",
database="classroom").mode("append").save()

df = spark.read.format("mongo").\
option("uri", "mongodb://127.0.0.1/classroom.people").load()
df.show()
```

Putting It All Together

- NoSQL clusters can be used to quickly scale acquired data in situations where SQL servers could not handle the volume
- To analyze the data, you could:
  - Export periodically from NoSQL into files stored in HDFS and use Spark to process it
  - Export it to a format that is optimal for Spark queries
  - Preserve a snapshot of it as of a moment in time
- The better alternative is to allow Spark to directly query from the NoSQL cluster
  - Zero latency query to real data
  - No need to keep a separate copy of it
- Use NoSQL for the transactional system and Spark for the data warehouse and reporting queries
  - Together, the two different clusters come together to provide the full feature set of a single SQL server but with the scalability of a cluster
- Could store results of a query into static files or write back to a NoSQL database

Chapter Concepts

NoSQL

Cassandra

MongoDB

Chapter Summary

Chapter Summary

In this chapter, we have:

- Discussed NoSQL
- Introduced Cassandra and Mongo
- Used Python to access Cassandra and Mongo directly
- Used Spark to read and write to Cassandra and Mongo