Spark SQL

cerenode

cerenode.io

Spark SQL

- **SQL** is a Spark module to simplify working with structured data using DataFrame and DataSet abstraction
- These abstractions are the distributed collection of data organized into named columns.
- It provides a good optimization technique.
- Using Spark SQL we can query data, both from inside a Spark program and from external tools that connect through standard database connectors (JDBC/ODBC) to Spark SQL.

Spark Session

- Spark 2.0 introduced SparkSession, an entry point that subsumed SQLContext and HiveContext
- It is the entry point for reading data
- It can be used to execute SQL queries over data, getting the results back as a DataFrame (i.e. Dataset[Row]).
- It includes a catalog method that contains methods to work with the metastore (i.e. data catalog). Methods there return Datasets so you can use the same Dataset API to play with them.
- SparkSession.sparkContext returns the underlying SparkContext, used for creating RDDs as well as managing cluster resources.

DataFrame

- Like an RDD, a DataFrame is an immutable distributed collection of data.
- Unlike an RDD, data is organized into named columns, like a table in a relational database.
- In Spark 2.0 DataFrame APIs merged with Dataset APIs,
- Supports different data formats (Avro, CSV, Elastic Search and Cassandra) and storage systems (HDFS, HIVE Tables, MySQL, etc.).
- Can be easily integrated with all Big Data tools and frameworks via Spark-Core.

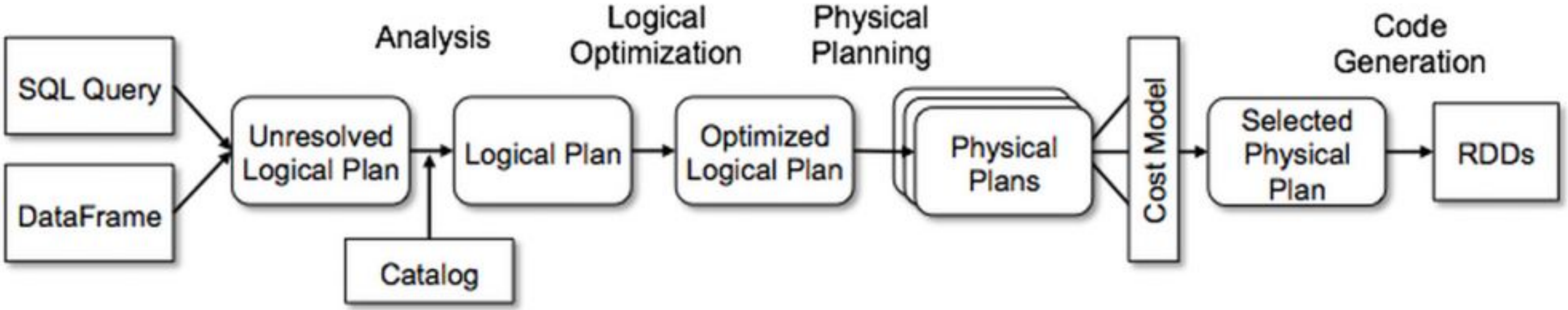
Dataset

- It is an extension of the DataFrame API that provides a *type-safe, object-oriented programming interface*.
- It is a strongly-typed, immutable collection of objects that are mapped to a relational schema
- Dataset provides the benefits of RDDs along with the benefits of Apache Spark SQL's optimized execution engine
- The Dataset API is accessible in *Scala* and *Java*. Dataset API is not supported by Python.
- Since Spark understands the structure of data in Datasets, it can create a more optimal layout in memory when caching Datasets
- At the core of the Dataset API is a new concept called an encoder, which is responsible for converting between JVM objects and tabular representation

Catalyst Optimizer

- The *optimizer* used by Spark SQL is **Catalyst optimizer**.
- It optimizes all the queries written in Spark SQL and DataFrame DSL.
- The optimizer helps us to run queries much faster than their counter RDD part.

Stages of Catalyst Optimizer



Where to Use DataFrames/Dataset?

- If you want rich semantics, high-level abstractions, and domain specific APIs, use DataFrame or Dataset.
- If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.
- If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset.

User-Defined Functions

- **User-Defined Functions** (aka **UDF**) is a feature of Spark SQL to define new Column-based functions that extend the vocabulary of Spark SQL's DSL for transforming Datasets.
- Use the higher-level standard Column-based functions (with Dataset operators) whenever possible before reverting to developing user-defined functions since UDFs are a blackbox for Spark SQL and it cannot (and does not even try to) optimize them.
- You can register UDFs to use in SQL-based query expressions via UDFRegistration

RDDs Partition Example

```
rdd = spark.hadoopFile("hdfs://click_logs/")
```

Partitions = 1 per HDFS block

Dependencies = None

compute(partition) = read corresponding HDFS block

Partitioner = None

RDDs Partition Example (contd.)

```
filtered = rdd.filter(lambda x: x contains “ERROR”)
```

Partitions = parent partitions

Dependencies = a single parent

compute(partition) = call parent.compute(partition) and filter

Partitioner = parent partitioner

RDD’s Partition Example (contd.)

`Joined RDD`

Partitions = number chosen by user or heuristics

Dependencies = ShuffleDependency on two or more parents

compute(partition) = read and join data from all parents

Partitioner = HashPartitioner(# partitions)

Performance Improvements

- Larger the better
- ML Jobs / ETL Jobs
- Executor heap size to 64GB or less
- Changing compression format
  - `conf.set("spark.io.compression.codec", "lzf")`
- Turn on
  - `conf.set("spark.speculation", "true")`
- Dedicated shuffle SSDs or disks
  - Set inside configuration file of YARN/Mesos

Serialization

```
val conf = new SparkConf()
conf.set("spark.serializer","org.apache.spark.serializer
    .KryoSerializer")

// Be strict about class registration

conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass],
    classOf[MyOtherClass]))
```

Serialization

- Passing non-serialized value to a worker node

```
nonSerializable = …
rdd = sc.textFile(hdfs://…).
      map(line => line + nonSerailizable).
      take(n)
print rdd
```

Machine Specs for YARN Cluster Example

- 6 nodes
- 16 cores each
- 64 GB each

Most Granular Configuration

- Smallest sized executors
- 1 executor per core ~ 6 * 16 = 96 executors
- 16 executors per node
- 64 GB / 16 = 4 GB per executor

- Not using benefits of multiple tasks that can be run in same JVM

Least Granular

- Each node acts as a single executor
- 6 nodes = 6 executors
- Each executor takes 64 GB

- Need to add overhead for OS/Hadoop daemons

Least Granular - with overhead

- 6 executors
- 63 GB memory each
- 15 cores each

HDFS Throughput

- 16 cores per executor can lead to bad I/O throughput
- 5 cores per executor is a good option

Estimations

- 5 cores per executor (considering max HDFS thorughput)
- 6 nodes in total, 6 * 15 = 90 cores (1 core per node for OS/Hadoop daemons)
- 90 cores / 5 cores per executor = 18 executors
- 1 executor for application master = 18 - 1 = 17 executors
- 6 nodes, 18 executors / 6 nodes = 3 executors per node
- 64 GB - 1 GB (for OS/Hadoop) = 63 GB
- 63 GB / 3 executors = 21 GB per executor
- Considering heap overhead (= max(394 MB, 0.07 * executor memory))
  - 21 GB - 21 * 0.07 ~ 19 GB per executor

Partitions

- Spark shuffle block size cannot be greater than 2GB
- Problematic in Spark SQL
- Default no:of partitions during shuffling = 200
    - Leads to huge shuffle blocks
- Set `spark.sql.shuffle.partitions` (for Spark SQL)
- For normal Spark code use `repartition() / coalesce()`

Partitions

- Partitions less than 2 GB is safe
- If no:of partitions are less then parallelism is reduced
- Good partitions size = 128 MB
- If #no:of partitions is almost 2000, increase partitions to more than 2000

Skewness

- Salting - adding extra value to skewed key
  - eg: normal key - ‘foo’
    salted key - ‘foo’ + rand.nextInt()
- Steps:
  - Convert to salted key, value pairs
  - Reduce by salted key
  - Convert to key, value pairs
  - Reduce by key

Skewness

- Isolated salting
  - Isolate skewed keys and salt them
  - Execute reduce operations
  - Filter salted keys
  - Map to normal keys and execute reduce operations
  - Join the results

Skewness

- Map Joins
  - Filter out isolated keys and use Map Join/aggregate on them
  - Normal reduce on rest of the data

DAG Management

- Shuffles are to be avoided
  - Broadcast join if possible
    - `spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)`
    - `left.join(broadcast(right), columns)`
  - Do as much as possible within single shuffle
  - Avoid skew and cartesians
- ReduceByKey over GroupByKey
- TreeReduce over reduce

Actions on RDD

- collect(), countByKey(), countByValue(), etc. are expensive unbounded operations
- Might induce out of memory exception in driver
- Use bounded output operation like count(), take()
- Use actions that saves data directly from the executors, like saveAsTextFile

Encoders

- Converts data between JVM objects and Spark SQL's specialized representation
- Every dataset has an encoder associated with it
- Generate custom bytecode for serialization and deserialization of data
- Uses less significant memory than Java/Kryo serialization