Topics of Today's Class

- 3. Distributed Computing Models for Scalable Batch Computing
  - Data Frame
  - Spark SQL
  - Datasets
- 4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron
  - Apache Storm Model
  - Parallelism
  - Grouping Methods

What is the Spark SQL?

- Spark module for structured data processing
  - Interface is provided by Spark
  - SQL and the Dataset API
- Spark SQL to execute SQL queries
  - Available with the command-line or over JDBC/ODBC

FAQs

- Submission Deadline for the GEAR Session 1 review
- Feb 25
- Presenters: please upload (canvas) your slides at least 2 hours before the presentation session

What is the Datasets?

- Dataset is a distributed collection of data
- New interface added in Spark (since V1.6) provides
  - Benefits of RDDs (Storing typing, ability to use lambda functions)
  - Benefits of Spark SQL's optimized execution engine
- Available in Scala and Java
- Python does not support Datasets APIs
What is the DataFrame?

- DataFrame is a dataset organized into named columns
- Like a table in a relational database or a data frame in R/Python
- Strengthened optimization scheme
- Available with Scala, Java, Python, and R

Create a SparkSession: Starting Point

SparkSession

- The entry point into all functionality in Spark

```scala
import org.apache.spark.sql.SparkSession
val spark = SparkSession
  .builder()
  .appName("Spark SQL basic example")
  .config("spark.some.config.option", "some-value")
  .getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
```

Find full example code at the Spark repo
effects/streaming/spark/examples/sparkSQLExample.scala

Creating DataFrames

- With a SparkSession, applications can create DataFrames from
  - Existing RDD
  - Hive table
  - Spark data sources

```scala
val df = spark.read.json("/examples/src/main/resources/people.json")
```

Untyped Dataset Operation (A.K.A. DataFrame Operations)

- DataFrames are just Dataset of Rows in Scala and Java API
- Untyped transformations
  - "typed operations"?
  - Strongly typed Scala/Java Datasets

```scala
val df = spark.read.json("/examples/src/main/resources/people.json")
```

Untyped Dataset Operation (A.K.A. DataFrame Operations)

- Select only the 'name' column

```scala
df.select("name").show()
```
Untyped Dataset Operation (A.K.A. DataFrame Operations)

```javascript
// Select everybody, but increment the age by 1
$.select("name", $.age + 1).show();

// +----------
// | name     |
// | age       |
// +----------
// | Michael   |
// | null      |
// | Andy      |
// | 20        |
// | Justin    |
// +----------
```

Untyped Dataset Operation

```javascript
// Select people older than 21
$.filter($.age > 21).show();

// +----
// | age|
// | name|
// +----
// | 30  |
// | Andy|
// | 19  |
// | Justin|
// +----
```

Running SQL Queries

```javascript
// Register the DataFrame as a SQL temporary view
$.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show();

// +---
// | age|
// | name|
// +---
// | null|
// | Andy|
// | 19|
// | Justin|
// +---
```

Global Temporary View

```javascript
// Register the DataFrame as a global temporary view
$.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database
spark.sql("SELECT * FROM global_temp.people").show();

// +---
// | age|
// | name|
// +---
// | null|
// | Andy|
// | 19|
// | Justin|
// +---
```

Global Temporary View

```javascript
// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show();

// +---
// | age|
// | name|
// +---
// | null|
// | Andy|
// | 19|
// | Justin|
// +---
```

http://www.cs.colostate.edu/~cs535
Creating Datasets

- Datasets are similar to RDDs
- Serializes object with Encoder (not standard java/Kryo serialization)
- Many of Spark Dataset operations can be performed without deserializing object

```scala
// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()

// +----+
// | name|age |
// +----+
// | Andy|32 |
```

- Datasets are using non-standard serialization library (Spark's Encoder)

```scala
// Encoders for most common types are automatically provided by importing 
// spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
```

```scala
// Returns: Array(2, 3, 4)
```

Interoperating with RDDs

1. Using Reflection

- Automatic converting of an RDD (containing case classes) to a DataFrame
- The case class defines the schema of the table
- E.g. the names of the arguments to the case class are read using reflection
- become the names of the columns
- Case classes can also be nested or contain complex types such as Seq or Arrays
- RDD will be implicitly converted to a DataFrame and then be registered as a table

```scala
// For implicit conversions from RDDs to Dataframes
import spark.implicits._

// Create an RDD of Person objects from a text file, convert it to a dataframe
val peopleDF = spark.sparkContext 
  .textFile("examples/src/main/resources/people.txt") 
  .map{ attributes => Person(attributes(0), attributes(1).trim.toInt) }
  .toDF()

// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by Spark
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age BETWEEN 13 AND 19")
```
### Interoperating with RDDs

2. Specifying the Schema Programmatically

- When case classes cannot be defined ahead of time
- Step 1: Create an RDD of Rows from the original RDD
- Step 2: Create the schema represented by a `StructType` matching the structure of Rows in the RDD created in Step 1
- Step 3: Apply the schema to the RDD of Rows via `createDataFrame` method provided by SparkSession.

```scala
// Example Java code for creating a temporary view using DataFrame
val peopleDF = spark.sql("SELECT name FROM people")
peopleDF.createTempView("people")
```

### Datasets and Type Safety

- Datasets are composed of typed objects
- Spark SQL is composed of a string
- DataFrames are composed of untyped Row objects
- DataFrames can be used to create temporary views
- Syntax errors and analysis errors are only caught at compile time
- Incorrect input variable type can be caught at runtime
- Spark SQL can be used to create temporary views
- Syntax errors (e.g., a typo in the method name) and analysis errors (e.g., an incorrect input variable type) can be caught at runtime
- Analysis errors (e.g., an incorrect input variable type) can be caught at runtime

### This material is built based on

- Storm programming guide
  - http://storm.apache.org/releases/2.0.0-RELEASE/index.html
4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Apache Storm Model

- One-at-a-time stream processing
- Represents the entire stream processing pipeline as a graph of computation called a topology
- A single program is deployed across a cluster
- A stream is represented an infinite sequence of tuples
  - A tuple: a named list of values

Why use Storm?

- Distributed real-time computation system
- Realtime analytics
  - Online machine learning
  - Continuous computation
  - Distributed RPC
  - ETL
- Use Case
  - Twitter
  - Amazon Kinesis and Storm
    - KinesisSpoutConfig
    - KinesisSpout

Spout in the Storm model

- Spout
  - A source of streams in a topology
  - A spout can read from a Keatnel or Kafka queue
  - Turns the data into a tuple stream

Bolt in the Storm model

- Bolt
  - Performs actions on streams
  - Takes any number of streams as input and produces any number of streams as output
  - Runs functions, filters data, computes aggregations, does streaming joins, updates database, etc.

Topology in the Storm model

- Topology
  - A network of spouts and bolts with each edge representing a bolt that processes the output stream of another spout or bolt
- Task
  - Each instance of a spout or bolt

http://www.cs.colostate.edu/~cs535
4. Real-time Streaming Computing Models: Apache Storm and Twitter Heron

Apache Storm

Word Count Example

<table>
<thead>
<tr>
<th>Sentence Spout</th>
<th>Split Sentence Bolt</th>
<th>Word Count Bolt</th>
<th>Report Bolt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence Spout</td>
<td>Split Sentence Bolt</td>
<td>Word Count Bolt</td>
<td>Report Bolt</td>
</tr>
</tbody>
</table>

- Sentence spout
  - Emits a stream of single-value tuples continuously with the key name "sentence" and a string value "{"sentence":"my dog has fleas"}

- Split Sentence Bolt
  - Subscribes to the sentence spout’s tuple stream
    - "word":"my"
    - "word":"dog"
    - "word":"has"
    - "word":"fleas"

- Word Count Bolt
  - Subscribes to the output of the SplitSentenceBolt class
  - Keeps a count of how many times it has seen a particular word
  - Whenever it receives a tuple, it will increment the counter and emit
    - "word":"dog", "count":5

- Report Bolt
  - Subscribes to the output of the WordCountBolt class
  - Keeps a count of how many times it has seen a particular word
  - Whenever it receives a tuple, it will update the table and print the contents to the console

SentenceSpout.java

```java
public class SentenceSpout extends BaseRichSpout {
  private SpoutOutputCollector collector;
  private String[] sentences = {
    "my dog has fleas",
    "i like cold beverages",
    "the dog ate my homework",
    "don't have a truck",
    "i don't think i like fleas"
  };
  private int index = 0;
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("sentence"));
  }
  public void open(Configuration conf, StreamZooKeeper stream) {
    collector = new SpoutOutputCollector();
    collector.open(conf);
  }
  public void nextBatch(Listzug<
```
SentenceSpout.java: Continued

```java
public class SentenceSpout {
    public void open(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void nextTuple() {
        String sentence = new Values(this.collector.poll(new Values(0)));
        if (sentence != null) {
            index = 0;
        }
        else {
            System.out.println("The end!");
            this.collector.emit(this.collector.poll(new Values(0)), index);
            return;
        }
        String[] sentences = sentence.split(" ");
        for (String word : sentences) {
            int count = 0;
            this.collector.emit(word, count);
        }
    }
}
```

WordCountBolt.java

```java
public class WordCountBolt extends BaseRichBolt {
    private HashMap<String, Long> counts = null;
    private static final String REPORT_BOLT_ID = "report-bolt";
    private static final String COUNT_BOLT_ID = "count-bolt";
    private static final String SPLIT_BOLT_ID = "split-bolt";
    private static final String SENTENCE_SPOUT_ID = "sentence-spout";
    public void open(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        long count = tuple.getLongByField("count");
        count++;
        if(count == 0) count = 0;
        this.collector.emit(word, count);
    }
    public void cleanup() {
        for (String key : keys) {
            Collections.sort(keys);
            this.counts.put(key, this.counts.get(key) + " : " + this.counts.get(key));
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

SplitSentenceBolt.java

```java
public class SplitSentenceBolt extends BaseRichBolt {
    private HashMap<String, Long> counts = null;
    public void open(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for (String word : words) {
            Long count = this.counts.get(word);
            if (count == null) count = 0;
            count++;
            this.collector.emit(word, count);
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

WordCountTopology.java

```java
public class WordCountTopology {
    private static final String SENTENCE_SPOUT_ID = "sentence-spout";
    private static final String COUNT_BOLT_ID = "count-bolt";
    private static final String SPLIT_BOLT_ID = "split-bolt";
    private static final String REPORT_BOLT_ID = "report-bolt";
    private static final String TOPLOGY_NAME = "word-count-topology";
    public static void main(String[] args) throws Exception {
        LocalCluster cluster = new LocalCluster();
        SplittableBolt splitBolt = new SplittableBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout(SENTENCE_SPOUT_ID, new SentenceSpout());
        builder.setBolt(SENTENCE_SPOUT_ID, splitBolt, SPLIT_BOLT_ID);
        builder.setBolt(SPLIT_BOLT_ID, countBolt, COUNT_BOLT_ID);
        builder.setBolt(COUNT_BOLT_ID, reportBolt, REPORT_BOLT_ID);
        builder.declareOutputFields(new Values(0));
        builder.createTopology();
    }
}
```

WordCountTopology.javaContinued

```java
// SplitSentenceBolt -> WordCountBolt
builder.setBolt(SENTENCE_SPOUT_ID, countBolt, COUNT_BOLT_ID);
// WordCountBolt -> ReportBolt
// ReportBolt -> WordCountBolt
Config config = new Config();
Cluster cluster = new LocalCluster();
cluster.submitTopology(TOPLOGY_NAME, config, builder.createTopology());
```
Results
--- FINAL COUNTS ---
a : 1426
ate : 1426
beverages : 1426
cold : 1426
cow : 1426
dog : 2852
don't : 2851
fleas : 2851
has : 1426
have : 1426
homework : 1426
i : 4276
like : 2851
man : 1426
my : 2852
the : 1426
think : 1425
--------------

Components of the Storm cluster

- **Nodes** (machines)
  - Executes portions of a topology
- **Workers** (JVMs)
  - Independent JVM processes running on a node
  - Each node is configured to run one or more workers
  - A topology may request one or more workers to be assigned to it
- **Executors** (threads)
  - Java threads running within a worker JVM process
  - Multiple tasks can be assigned to a single executor
  - Unless explicitly overridden, Storm will assign one task to each executor
- **Tasks** (bolt/spout instances)
  - Instances of spouts and bolts whose nextTuple() and execute() methods are called by executor threads

Parallelism in the WordCount topology

- In our example, we have NOT used any of Storm’s parallelism
- Default setting is a factor of one
- Topology execution flow

Adding workers to a topology

- Through configuration
- Through APIs
  - Passing `Config` object to the `submitTopology()` method
- Bolts and spouts do not have to change

```java
Config config = new Config();
config.setNumWorkers(2);
```

Adding executors and tasks

- Specify the number of executors when defining a stream grouping
- `StreamSpout` class
- `builder.setSpout()` method
- Assigns two tasks and each task is assigned its own executor thread

http://www.cs.colostate.edu/~cs535

Spring 2020 Colorado State University, page 9
Two spout tasks (if we are using one worker)

In `SplitSentenceBolt` and `WordCountBolt`,
- Set up the split sentence bolt to execute as 4 tasks and 2 executors
  - Parallelism hint
  - Storm will run 2 tasks per executor (thread)
  - Each executor thread will be assigned two tasks to execute

```
builder.setBolt(SPLIT_BOLT_ID, splitBolt, 2)  
  .setNumTasks(4)  
  .shuffleGrouping(SENTENCE_SPOUT_ID);
```

- How many workers will work for this example?
  - Answer: 2 workers (JVMs)
  - 2 executors per worker

### Stream Grouping

Seven built-in stream groupings (1/3)

- **Shuffle grouping**
  - Randomly distributes tuples across the target bolt’s tasks

- **Fields grouping**
  - Routes tuples to bolt tasks based on the values of the fields specified in the grouping
    - Grouped on the “word” field
    - Tuples with the same value for the “word” field will always be routed to the same bolt task

- **All grouping**
  - Replicates the tuple stream across all bolt tasks
Seven built-in stream groupings (2/3)

- **Global grouping**
  - Routes all tuples in a stream to a single task
  - Chooses the task with the lowest task ID value

- **None grouping**
  - Functionally equivalent to the shuffle grouping
  - Reserved for future use

- **Direct grouping**
  - The source stream decides which component will receive a given tuple
  - By calling the `emitDirect()` method
  - Only for streams that have been declared as direct streams

Seven built-in stream groupings (3/3)

- **Local or shuffle grouping**
  - Shuffles tuples among bolt tasks running in the same worker process, if any
  - Otherwise, performs shuffle grouping
  - Depending on the parallelism of a topology, the local or shuffle grouping can increase topology performance by limiting network transfer

Custom Grouping Stream

```java
public interface CustomStreamGrouping extends Serializable {
    void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer> targetTasks);
    List<Integer> chooseTasks(int taskId, List<Object> values);
}
```

Example of grouping (1/2)

```
public void nextTuple() {
    if(index < sentences.length){
        this.collector.emit(new Values(sentences[index]));
        index ++;
    }
    Utils.waitForMillis(1);
}
```

Example of grouping (2/2)

```
WHY?

- The CountBolt parameter is stateful
  - It maintains a count for each word it’s seen
  - Shuffle grouping will not capture the total count of appearances of a word

```

Why?

```
--- FINAL COUNTS ---
 a : 2
 ate : 2
 beverages : 2
 cold : 2
 cow : 2
 dog : 4
 don't : 4
 have : 2
 homework : 2
 i : 6
 like : 4
 man : 2
 my : 4
 the : 2
 think : 2
--------------
```

```
--- FINAL COUNTS ---
 a : 2
 ate : 2
 beverages : 2
 cold : 2
 cow : 2
 dog : 4
 don't : 4
 have : 2
 homework : 2
 i : 6
 like : 4
 man : 2
 my : 4
 the : 2
 think : 2
--------------
```

--- FINAL COUNTS ---
 a : 2
 eat : 2
 beverages : 2
 cold : 2
 cow : 2
 dog : 4
 don't : 4
 have : 2
 homework : 2
 i : 6
 like : 4
 man : 2
 my : 4
 the : 2
 think : 2
--------------
```

--- FINAL COUNTS ---
 a : 2
 eat : 2
 beverages : 2
 cold : 2
 cow : 2
 dog : 4
 don't : 4
 have : 2
 homework : 2
 i : 6
 like : 4
 man : 2
 my : 4
 the : 2
 think : 2
--------------
Questions?