Deep Analysis with Apache Flink

Andreas Kunft - TU Berlin / DIMA - andreas.kunft@tu-berlin.de

with Material from Asterios Katsifodimos, Alexander Alexandrov, Andra Lungu and Aljoscha Krettek
Outline

1. Flink Introduction
2. Machine Learning with Flink
3. Graph Analysis with Flink
4. Relational Queries with Flink
5. Research / Emma
Flink Introduction
What is Apache Flink

• Massive parallel data flow engine with unified batch- and stream-processing

• Evolved from the joint research project Stratosphere funded by DFG

• Now Apache top-level project

• About 120 contributors, highly active community
What is Apache Flink

- **Declarativity**
- **Query optimization**
- **Robust out-of-core**

- **Iterations**
- **Adv. dataflows**
- **General APIs**

- **Scalability**
- **UDFs**
- **Complex data types**
- **Schema on read**

Taken from Database technology

Taken from MapReduce technology

5
System Stack
The case for Flink

• Performance and ease of use
  • Exploits in-memory processing and pipelining, language-embedded logical APIs

• Unified batch and real streaming
  • Batch and Stream APIs on top of a streaming engine

• A runtime that "just works" without tuning
  • custom memory management inside the JVM

• Predictable and dependable execution
  • Bird’s-eye view of what runs and how, and what failed and why
Rich set of operators

Map, Reduce, Join, CoGroup, Union, Iterate, Delta Iterate, Filter, FlatMap, GroupReduce, Project, Aggregate, Distinct, Vertex-Update, Accumulators
Built-in vs. driver-based looping

Loop outside the system, in driver program

Iterative program looks like many independent jobs

Dataflow with Feedback edges

System is iteration-aware, can optimize the job
Flink Runtime: Operators & UDFs

Language APIs automatically convert objects to tuples
• Tuples mapped to pages of bytes
• Operators work on pages
• Full control over memory, out-of-core enabled
• Address individual fields (not deserialize whole object)
• UDFs work on deserialized objects

```java
public class WC {
    public String word;
    public int count;
}
```
SAY "WORD COUNT"

ONE MORE TIME...
Wordcount: Program

case class Word (word: String, frequency: Int)

val env = ExecutionEnvironment.getExecutionEnvironment()

val lines: DataSet<String> = env.readTextFile(...)

lines
  .flatMap { line =>
    line.split(" ").map( word => Word(word, 1) )
  }
  .groupBy("word")
  .sum("frequency")
  .print()

env.execute()
Wordcount: Execution
Machine Learning with Flink
K-Means Clustering

• Cluster analysis in data mining

• Partitions $n$ observations into $k$ clusters

• Assign points to cluster with smallest (euclidian) distance
**K-Means Clustering**

**Input:**
- set of observations $X = \{x_1, x_2, ..., x_n\}$
- number of clusters $k$
- convergence criteria $\xi$

**Result:**
- set of clusters $C = \{c_1, c_2, ..., c_k\}$

**Init:**
- select $k$ data points as cluster centroids $C = \{c_1, c_2, ..., c_k\}$

**Compute:**
- do
  - foreach $x$ in $X$
    - assagin $x$ to cluster with closest centroid
    - recompute centroid of each cluster
  - while $\xi$ is not reached (or fixed number of iterations)
K-Means in Flink

// initialize
// points: n observations, centroids: initial k centroids
val cntrds = centroids.iterate(10) { currCntrds =>

    val newCntrds = points
        .map(findNearestCntrd).withBroadcastSet(currCntrds, "cntrds")
        .map( (c, p) => (c, p, 1L) )
        .groupBy(0).reduce( (x, y) =>
            (x._1, x._2 + y._2, x._3 + y._3) )
        .map( x => Centroid(x._1, x._2 / x._3) )

    newCntrds
}
Reduce

\{1, 3, 5, 7\}.reduce \{ (x, y) \to x + y \}
K-Means in Flink

// initialize
// points: n observations, centroids: initial k centroids

```scala
val cntrds = centroids.iterate(10) {
  currCntrds =>
    val newCntrds = points
    .map(findNearestCntrd).withBroadcastSet(currCntrds, "cntrds")
    .map((c, p) => (c, p, 1L))
    .groupBy(_0).reduce((x, y) => (x._1, x._2 + y._2, x._3 + y._3))
    .map(x => Centroid(x._1, x._2 / x._3))

    newCntrds
}
```

Machine learning library

• Recently started effort
• Currently available algorithms
  • Classification
  • Logistic Regression
  • Clustering
  • Recommendation (ALS)
Machine Learning Library

val featureExtractor = HashingFT()
val factorizer = ALS()

val pipeline = featureExtractor.chain(factorizer)

val clickstreamDS = 
  env.readCsvFile[(String, String, Int)](clickStreamData)

val parameters = ParameterMap()
  .add(HashingFT.NumFeatures, 1000000)
  .add(ALS.Iterations, 10)
  .add(ALS.NumFactors, 50)
  .add(ALS.Lambda, 1.5)

val factorization = pipeline.fit(clickstreamDS, parameters)
Graph Analysis with Flink
Gelly

- Large-scale graph processing API
- On top of Flink’s Java API
- Official release in Flink 0.9
Execution model

- Pregel-like or *Bulk Synchronous Parallel* (BSP) execution model
- *Synchronization barrier* after each superstep
- At each superstep
  - Receives messages from previous superstep
  - Modifies its value
  - Sends messages to vertices
Vertex State Machine

- Active
- Inactive

Votes to halt
Message received
Example: Maximum Value

Red lines are messages, grey vertices voted to halt.
Example: Maximum Value

Red lines are messages, grey vertices voted to halt.
Example: Maximum Value

Red lines are messages, grey vertices voted to halt.
Example: Maximum Value

Red lines are messages, grey vertices voted to halt.
Single Source Shortest Paths (SSSP)

\[ i = 0 \]

\[ i = 2 \]

\[ i = 1 \]

\[ i = 3 \]

Red nodes are updated, grey nodes are inactive
SSSP – Code snippet

```java
shortestPaths = graph.runVertexCentricIteration(
    new DistanceUpdater(), new DistanceMessanger()).getVertices();
```
SSSP – Code snippet

```java
shortestPaths = graph.runVertexCentricIteration(
    new DistanceUpdater(), new DistanceMessenger()).getVertices();
```

**DistanceUpdater: VertexUpdateFunction**

```java
updateVertex(K key, Double value,
    MessageIterator msgs) {
    Double minDist = Double.MAX_VALUE;
    for (double msg : msgs) {
        if (msg < minDist) {
            minDist = msg;
        }
    }
    if (value > minDist) {
        setNewVertexValue(minDist);
    }
}
```
SSSP – Code snippet

```java
shortestPaths = graph.runVertexCentricIteration(
    new DistanceUpdater(), new DistanceMessanger()).getVertices();

DistanceUpdater: VertexUpdateFunction

updateVertex(K key, Double value, 
              MessageIterator msgs) {
    Double minDist = Double.MAX_VALUE;
    for (double msg : msgs) {
        if (msg < minDist) {
            minDist = msg;
        }
    }
    if (value > minDist) {
        setNewVertexValue(minDist);
    }
}

DistanceMessenger: MessagingFunction

sendMessages(K key, Double newDist) {
    for (Edge edge : getOutgoingEdges()) {
        sendMessageTo(edge.getTarget(), 
                      newDist + edge.getValue());
    }
}
```
Graph Partitioning

• Real world graphs often have a power law distribution

• Problem for BSP, as all nodes have to wait for stragglers at barrier

> Graph Partitioning
Partitioning

Edge-Cut

Vertex-Cut

Relational Queries with Flink
First Things First

Table activeUsers = users.join(clickCounts)
  .where("id = userId && count > 10")
  .select("username, count");

val activeUsers = users.join(clickCounts)
  .where('id === 'userId && 'count > 10)
  .select('username, 'count)
Under the Hood

’a === ‘b && ‘c > 3
Log Analysis

• Collect clicks from a webserver log
• Find interesting URLs
• Combine with user data
Getting the clicks

```java
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// Read log file
DataSet<String> log = env.readTextFile("hdfs:///log");

// Process log file
DataSet<Tuple2<String, Integer>> clicks = log.flatMap(
    (String line, Collector<Tuple2<String, Integer>> out) -> {
        String[] parts = line.split("*magic regex*");
        if (isClick(parts)) {
            out.collect(new Tuple2<>(parts[1], Integer.parseInt(parts[2])));
        }
    }
);
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Path</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>post</td>
<td>/foo/bar...</td>
<td>313</td>
</tr>
<tr>
<td>get</td>
<td>/data/pic.jpg</td>
<td>128</td>
</tr>
<tr>
<td>post</td>
<td>/bar/baz...</td>
<td>128</td>
</tr>
<tr>
<td>post</td>
<td>/hello/there...</td>
<td>42</td>
</tr>
</tbody>
</table>
Counting the clicks

TableEnvironment tableEnv = new TableEnvironment();

Table clicksTable = tableEnv.toTable(clicks, "url, userId");

Table urlClickCounts = clicksTable
   .groupBy("url, userId")
   .select("url, userId, url.count as count");
Getting the user information

Table userInfo = tableEnv.toTable(..., "name, id, ...");

Table resultTable = urlClickCounts.join(userInfo)
  .where("userId = id && count > 10")
  .select("url, count, name, ...");
Work with the result

class Result {
    public String url;
    public int count;
    public String name;
    ...
}

DataSet<Result> set = tableEnv.toSet(resultTable, Result.class);
DataSet<Result> result =
    set.groupBy("url").reduceGroup(new ComplexOperation());
result.writeAsText("hdfs:///result");
env.execute();
Thanks for your attention

• Play with it
  https://flink.apache.org/

• Get involved
  https://github.com/apache/flink
  dev@flink.apache.org
  news@flink.apache.org
  @ApacheFlink