Paper Review –9

Resilient Distributed Datasets

Title: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Summary

The current programming models don't provide a generic solution to problems faced in a distributed system. After the introduction to Hadoop, frameworks have worked to provide a new framework by addressing some of issues that exist in Hadoop system when the problem was more about the memory model. RDD (Aka Resilient Distributed Datasets) have been introduced as a memory abstraction for distributed systems and its powerful interface that helps capture most of what other programming frameworks aim to do.

Good Aspects

RDD's offer both fault tolerance and a restricted form of shared memory that is parallelized. This is a powerful feature of RDD and unlike traditional systems, RDD use lineage to recover lost data. RDD are powerful when used in applications that use iterative algorithms and interactive data mining tools. RDD's are also expressive enough to capture a wide class of computations, including some recent frameworks like Pregel.

Review

Following details will cover all aspect of the paper providing crucial and also interesting aspects of the paper that appealed the reader.

Motivation?

Cluster computing frameworks like Map Reduce and Dryad have incorporated high level operators for parallel computing of data. How ever what they lack is abstractions that can work with distributed memory which does not make them suitable to many of the new applications like that use Iterative Machine learning to recomputed values. Another use case is interactive data mining where a user runs multiple queries on a large data set which is quite efficient in a map reduce model.

Key Points about RDD

- Efficient data reuse
- Programming interface that provides fault tolerance efficiently
- Coarse grained updates
- Fault tolerance via Lineage
- Implemented here using Scala programing language

RDD Abstraction

- A RDD is a read only (Immutable) partitioned collection of records or blocks.
- The two ways to create RDD are: -
  - From data in a stable storage like on any file system.
  - Or by transformation which creates a new RDD from existing RDD.
- Examples of transformations include map, filter and join.
- An RDD has enough information to determine how it was created due to lineage.
- Users can control both persistence and partitioning.
- User also has the ability to request RDD partitions to be placed in a specific way.

Example Usage

Console Log Mining

- Below example shows
  - Lines = spark.textFile("hdfs://a.txt")
  - Errors = lines.filter(_.startsWith("ERROR"))
  - Errors.persist()
- The first line creates RDD by loading from hdfs .
- The second line performs a filter operation which is a transformation that creates a new RDD called errors. It basically filters lines that have the word ERROR in it.
- Now the new RDD is persisted by calling the persist () which stores in memory or disk depending on storage level requested.
- At this point no actual operation is performed. Spark has just built dependency graph. To really start any computation actions are used.
  - Errors.count()
- Count being an action will not perform all the operations to display the number of lines in the file a.txt that contains the word ERROR.

How different is RDD from DSM (Distributed Shared Memory)

Aspect	RDD	DSM
Reads	Coarse/Fine Grained	Fine Grained
Writes	Coarse Grained	Fine Grained
Consistency	Immutable	Up to App/Runtime
Fault Recovery	Fine grained & low overhead	Requires checkpoints
Straggler Mitigation	Using backup tasks	Difficult
Work placement	Automatic based on data locality or user based	Up to App (Like Piccolo)
Behavior when RAM mem is not enough	Similar to existing systems since its persisted on disk	Lot of swapping and poor performance

Applications not suitable for RDD

- RDD is best suited for batch applications that apply the same operation to all elements since its easy to pipeline efficiently per node for Spark
- This means Iterative algorithms will gain a lot including interactive query for data mining
- However, if your application wants to make fine grained updates asynchronously over a period of time to a shared state its easier to use systems like Piccolo that checkpoint data often.

Spark Programming Interface

- To use spark, developers write a driver program that connects to workers. The driver defines one or more RDD's and invokes operations on them. The spark code in the driver tracks the lineage information of RDD.
- Every function or operation performs operation by acting on the data set using the function or closure enclosed. The closures are shipped to the workers who are responsible to execute the function inside it.

- **RDD Operations**
    - Transformations
        - These are lazy operations that only track what to do
    - Actions
        - These are operations that actually do the work described in the DAG transformation by the help of scheduler.

- **Example 1 - Logistic Regression**
  - An example machine learning algorithm is Logistic Regression that aims to find a hyper plane that best separates points into two sets of points
  - Logistic regression is an example of a iterative classification algorithm.
  - It can be expressed in RDD.

```
val points = spark.textFile(...).map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
        val gradient = points.map{ p =>
                p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
        }.reduce((a,b) => a+b)
        w -= gradient
}
```

- **Example 2 - Page Rank**
  - Page rank is another algorithm that can expressed in RDD in spark.
  - However due to its asynchronous data changes to a shared state, its useful to use distributed system like Piccolo.

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
        // Build an RDD of (targetURL, float) pairs
        // with the contributions sent by each page
        val contribs = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
}
// Sum contributions by URL and get new ranks
ranks = contribs.reduceByKey((x,y) => x+y)
        .mapValues(sum => a/N + (1-a)*sum)
}
```

- **RDD's Main Interface function: -**
  - Return A List of partitions.
  - Provide a List of nodes where a partition can be stored to enable fast access.
  - Return a list of dependencies.
  - Iterate through blocks of parent partitions.
  - Return metadata specifying whether RDD is hash/range partitioned.

- ## Implementation

- **Scheduler**

    - For every action, scheduler determines how to execute by building a DAG by examining RDD's lineage graph.
    - Scheduler pipelines operations such as to reduce the number of shuffle operations across nodes by performing aggregation with in a node.
    - Delay scheduling is used if task wants to process all partitions that contain preferred locations.
    - Scheduler must restart a job if it failed by selecting another node to execute.

- **Interpreter Integrations**

    - **Class Shipping –** A custom class loader that helps nodes pull the class files over the network from the driver.

    - **Modify Byte code generation –** Java does not ship the enclosing classes inside closures hence after the modification members referenced from previous lines are also shipped.

- **Memory Management**

    - DE serialized Java objects in memory
    - Serialized Java objects in memory
    - On disk Storage

- ## Performance

    - Spark outperforms Hadoop by up to 20X in iterative machine learning algorithms:
    - Spark achieves a speed up to 40X than Hadoop in analytics report.
    - Spark can query 1 TB dataset interactively with latencies of 5 – 7 seconds.

- ## User Applications

    - In memory analytics within Conviva achieved a speed up to 40X
    - Traffic modeling at Berkeley uses spark to develop a learning algorithm to infer traffic congestion and suggest new routes.
    - Twitter Spam classification
    - Interactive Data mining of Wiki pages

Conclusion and Summary

- In conclusion the RDDs have provided an efficient general-purpose and fault tolerant abstraction for sharing data in cluster applications.
- RDDs can express a wide range of parallel applications, including some specialized programming models like Pregel.
- Unlike existing storage abstractions for clusters, RDDs offer an API based on coarse grained transformations that lets them recover data efficiently using lineage.
- Spark is currently open sources at spark-project.org