Paper Summary

Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing
Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications

Qiyang Lin(linqy), Ruying Sun(sruying)

Problem and Motivation:

Traditional cluster computing frameworks such as MapReduce provide abstractions for big data analytics, but they are not really suitable for two kinds of applications: one is iterative algorithms, which are common in graph and machine learning fields. The other is interactive data mining, where the users may run several ad-hoc queries. These old cluster computing systems mostly read and update data through external stable storage (HDFS), and bring large amount of overheads due to data replication, disk I/O and serialization. System performance will be significantly influenced when data reuse frequently occurs.

Hypothesis:

The authors pointed out some old solutions like Pregel and HaLoop that had limitations for only supporting specific computation patterns. They were looking for a abstraction for more general cases. To meet the requirements, the new abstraction should firstly enables data reuse among various applications. Users can persist intermediate results in memory. Secondly, it should provide fault tolerance efficiently unlike the replicating data or log updating methods based on fine-grained updates. At last such abstraction should be successfully implemented by existing systems and programming languages.

Solutions Overview:

1.The paper introduces a new abstraction called resilient distributed datasets (RDDs) that outperforms other frameworks. RDDs have several characteristics :

a. Read-only, partitioned collection of records.
b. No need to be materialized at all times and store information in lineage.
c. Users can control persistence and partitioning of RDDs.

2.There are some important concepts when designing RDDs:

(1) Lineage: Keep track of and log transformations (map, filter and join). It helps RDDs address fault tolerance.

(2) Driver program and workers:

a. Driver program's duty: connects to a cluster of workers; defines RDDs and invokes actions; tracks RDDs lineage.
b. Workers' duty: store RDD partitions in RAM.

(3) Narrow dependencies and wide dependencies:

Narrow dependencies mean the parent RDD is used by at most one partition of the child RDD, while in wide dependencies multiple child partitions depend on one parent RDD.

a. Narrow dependencies: allow pipeline execution; more efficient after failure.
b. Wide dependencies: need shuffle across nodes; may require a complete re-execution after failure.

(4) Transformations and actions: Transformations are lazy operations while actions launch a computation to return a value.
(5) RDDs representing: Represent RDDs with simple graph design and represent each RDD through a common interface.
(6) Job scheduler's duty: builds a DAG of stages; determines running locations for each task; handles failures.
(7) Storage options: Spark provides three options for persistent RDDs considering a trade-off between performance and space.

3.RDDs are built with some features that are responsible for improving the drawbacks of existing frameworks. The improvements can be categorized into following parts:
(1) Efficiency improvement (in general):

a. Mitigate slow nodes by running backup copies of slow tasks due to immutable nature of RDDs.
b. Create RDDs lazily by transformation and launch computation by actions.
c. Assign tasks to machine based on data locality by scheduler to reduce the cost of data transfer.
d. Pipeline as many transformations with narrow dependencies as possible when building a DAG of stages.

(2) Efficiency improvement for iterative algorithms and MapReduce:

a. Keep data in memory across iterations which avoids I/O.
b. Save intermediate RDDs that is likely to be reused in the future by persist method.
c. Control the partitions of the RDDs to optimize the communication between nodes.
d. Caching Java objects avoids the deserialization costs.

(3) Efficient Fault tolerance:

a. Recover failure using lineage without incurring the overhead of checkpointing. (more suitable for narrow dependencies case)
b. Materialize intermediate results for wide dependencies to simplify fault recovery.
c. Only those lost partitions need to be recomputed instead of rolling back whole program compared to DSM.
d. Write checkpoint in long lineage for more efficient recovery without requiring pause on program due to read-only nature of RDDs which is simpler.

(4) Expressivity of RDDs:

a. Not limited by coarse-grained transformation because parallel program naturally apply the same operation to many records.
b. Provide a rich set of transformation and actions which enables users to do various tasks.
c. Users can implement transformations in less than 20 lines of code by virtue of common interface (spark).

Limitations and Possible Improvements:

While RDDs work better for batch application that apply the same operation to all elements of a dataset, it doesn't perform well on those application that make asynchronous fine-grained updates to shared state. For example, a storage system for a web application or an incremental web crawler will prefer traditional methods. We can just deliver this kind of work to traditional database systems, Piccolo or RAMCloud.

Another problem is that if the memory is restricted, or is insufficient to store all the RDDs. User will need to spill them to disk, and the performance will degrade gracefully.

Designing a big data system for data that resides on NVM(Non-volatile Memory) may be a new idea or a solution for the limitations in RDDs. Instead of writing data into disk, we can consider NVM as a second level caching and let data go to NVM. We don't need to maintain lineage as much as before. If the speed of NVM is really fast, we can even throw away RAM completely.

Instead, people can choose DataFrames rather than RDDs. A DataFrame is a table which has additional metadata. It is lazy triggered as well but has huge performance improvement over RDDs because it stores data in off-heap memory in binary format, optimizes execution plan and is able to filter data in data source.

Summary of class discussion:

In class, we mainly discussed the concept of RDDs and the working mechanism of it. Besides the content of the paper, we exchanged our thoughts on the usage of RDDs in other systems such as Hadoop. Similar ideas of lineage have appeared earlier than RDDs. We discussed the checkpoint for wide dependencies. It will store data into the disk when cases like shuffle occur in order to provide efficient fault tolerance. After that, some basic concept about DraydLINQ was introduced. We also talked about closure. Closure is a function that can be stored as a variable, which can access other variables local to the scope it was created in. In the end, Apache Tez is a new open-source framework that supports DAG.It has the ability to transfer multiple mapreduce jobs to a single job and improve the performance.