Tony Nguyen

Dr. Shawn Bowers

CPSC 324 01

7 March 2024

Reading 3

Part 1: Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Impementation (NSDI)*, pp. 15–28.

Reading Notes

1. Introduction
    - Current cluster computing frameworks lack abstractions for leveraging distributed memory.
    - Inefficient for applications that reuse intermediate results across multiple connections
        - Common for applications like PageRack, K-means, and logistic regression
        - Also, interactive data mining
        - The only way to do so, like in MapReduce, is to write to an intermediate file
    - RDD enables efficient data reuse
        - Fault tolerance (efficiently), parallel data structures, explicit persist intermediate results in memory, control their partitioning, and manipulate them
    - RDD's interface is coarse-grained transformations

    - Provide fault tolerance by logging the transformations used to build a dataset rather than the actual data
    - Good fit for many applications because they naturally apply the same operation to multiple data items anyway
- Spark:
    - Language-integrated programming
    - Can be used interactively to query big datasets
    - The first system that allows a general-purpose programming language to be used at interactive speeds for in-memory data mining on clusters

2. Resilient Distributed Datasets
    - RDD abstraction
        - Formally, it is a read-only, partitioned collection of records
        - Can be created through deterministic operations on either data in stable storage or other RDD
            - Called *transformations* (e.g., map, filter, join)
        - RDD does not need to be materialized (lazy)
            - Has enough information about how it was derived from other datasets to compute its partitions from data in stable storage (knows how to do stuff)
            - A powerful property: An RDD cannot be referenced if it cannot be reconstructed after a failure
    - Spark Programming Interface
        - Implements RDD through a language-integrated API

        - Each dataset is an object
        - Transformations are methods
    - See the paper for how to implement
    - Spark computes RDD lazily the first time it is used in action so it can pipeline transformations.
- Example: Console Log Mining
- Advantage

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

    - RDD is only written through coarse-grained transformations, while DSM allows reading and writing to each memory location.
        - Because of that, RDD only performs bulk writes but has better fault tolerance as it does not have to checkpoint (using lineage instead)
        - Only the lost portion of RDD needs to be recomputed, which can be done parallelly on a different node. It does not have to roll back the entire program.
    - RDD allows the system to mitigate slow nodes by running backup copies of slow tasks in MR

- RDD can schedule tasks based on data locality for better performance
- RDD degrades gracefully when there is not enough memory.

- Not suitable Applications
  - Best suited for batch applications
  - Not good for applications that make asynchronous fine-grained updates

3. Spark Programming Interface

- Overview
  - Spark provides RDD abstraction through an API
  - Developers write a *driver program* that connects to a cluster of *workers*
  - The user provides arguments to RDD operations like map by passing closures (functional literals)

- RDD Operations in Spark

Transformations	*map*(f : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(f : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(f : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(f : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(f : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(c : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(p : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(f : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(k : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

  - Transformation is lazy operations that define a new RDD of key-value pairs.

- Example Applications
  - See section 2.2.1 in the reading

- Logistic Regression

- ML algorithms are interactive in nature because they run interactive optimization procedures (gradient descent)
- See paper for code and algorithm explanation

- PageRank
  - More complex data-sharing pattern
  - Iteratively updates a rank for each document that links to it
  - See paper for algorithm
  - Optimize communication in PageRank by controlling the partitioning of the RDD so that the joint operation does not require communication.

4. Representing RDDs
   - Challenge: How to represent RDD across a wide range of transformation
   - Represent each RDD through a common interface
Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

       Table 3: Interface used to represent RDDs in Spark.
   - But how do we represent their dependencies?
     - Narrow: each partition of the parent RDD is used by at most one partition of the child RDD
       - Allow for pipelined execution on one cluster node, which can compute all the parent partitions
       - Recovery is more efficient

        - Example: one can apply a map followed by a filter
    - Wide: multiple child partitions may depend on it (lineage)
        - Requires data from all parent partitions to be available and to be shuffled across the nodes using MR
- Map leads to a narrow dependency, while join leads to a wide dependency
- See the paper for the algorithm explanation

5. Implementation

- Job Scheduling
    - Consider which partitions of persistent RDDs are available
    - Whenever a user runs an action on an RDD, the scheduler examines that RDD lineage graph to build a DAG of stages to execute.
        - Each stage has many pipelined transformations with narrow dependencies
        - The boundaries are the shuffle operations
    - The scheduler then launches tasks to compute. It assigns tasks based on data locality
- Interpreter Integration
    - Has an interactive shell like Python. It compiles a class for each line typed by the user
    - Made two changes to the interpreter in Spark
        - Class shipping: fetch the bytecode for each class, serve over HTTP
        - Modified code generation: reference the instance of each line object directly

- Memory Management
    - Three storage options
        - In memory as a deserialized Java object: fastest performance
        - In memory as serialized data: more-memory efficient representation
        - On-disk storage: good for large files that are too big for RAM but costly to recompute
- Support for Checkpointing
    - Checkpointing to stable storage for fault tolerance and efficiency
    - Useful for wide dependencies since loss of data may occur if a node fails, requiring a full recomputation

6. Evaluation

- Overview
    - Spark outperforms Hadoop by 20x times
    - Scale well
    - Can recover quickly by only rebuilding the lost RDD partitions
- Interactive ML Applications
    - The main difference is the amount of computation performed per byte of data
    - Sharing data via RDDs generally speeds up the future iteration
    - Hadoop is slower on an in-memory file system because of:
        - Minimum overhead of the Hadoop software stack
        - The overhead of HDFS while serving data
        - Deserialization cost to convert binary records to usable in-memory Java objects

- Spark is able to achieve this as it saves all of the RDD elements as Java objects, which avoids the conversion cost

Answers to Questions

1. Describe course-grained and fine-grained
   - Course-grained refers to processing data in large blocks rather than individual ones. This helps reduce operation overhead and better fault tolerance (using lineage to recompute data)
   - Fine-grained involves more frequent data shuffling and communication across the network, making them less efficient
2. Describe Lineage
   - Lineage is a tracking scheme that tracks the sequence of changes and transformations in distributed resilient datasets
   - Lineage allows Spark to achieve fault tolerance by only recomputing lost data without having to replicate data
   - MapReduce, on the other hand, relies on data replication, which can be used to restore lost data. This requires more hardware, as well as software, to handle the computation compared to lineage.
3. Describe narrow and wide dependencies
   - Narrow dependencies: each partition of the parent RDD is used by at most one partition of the child RDD
     - Allow for pipelined execution on one cluster node, which can compute all the parent partitions
     - Recovery is more efficient

- Wide dependencies: requires shuffling data across many partitions using JOIN or GROUP BY statements. This makes recovery less efficient as it takes more computing to recompute the lost portion
- These two types of dependencies allow Spark to recompute the lost portions effectively. Narrow dependency enables Spark to only compute the lost portions independently (allowing minimal computation). Wide dependencies, on the other hand, may require more extensive computing requirements.

4. Spark evaluation
    - Spark is evaluated against Hadoop using PageRank and interactive data mining tasks
    - Spark outperforms Hadoop due to its in-memory data storage and efficient data recovery through lineage, avoiding replications like MapReduce (Hadoop)
    - Spark is also able to support a lower latency query execution

Additional Questions and Observations

1. I am impressed by Spark's ability to compute large datasets efficiently. I have heard about the system for a while but never had a chance to understand it. And yet the way it is implemented is truly impressive. When we were doing MapReduce and Google Cluster Management (Borg), we talked about how Google replicates data across several nodes to preserve recovery. I have always wondered if there can be a better and more efficient way to maintain fault tolerance like that while lowering the hardware need. Now, with lineage in Spark, we finally have the answer.

Part 2: Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google File System. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 29–43.

Reading Notes

1. Introduction
    - Dremel is a distributed data analysis system
        - Fully managed, serverless data warehouse
        - Allows scalable analytics
    - Throwing hardware at the problem is insufficient. It is better to understand the structure of the data.
    - Main ideas
        - SQL: embraces SQL API for querying and retrieving data
        - Disaggregated compute and storage: decouples machines and storage (share disk)
        - In situ analysis: Dremel uses a DFS system (GFS) to have MR and other data processing operations interoperate (at the same time) seamlessly with SQL-based analysis
        - Serverless computing: switch to on-demand (elastic) resources
        - Columnar storage: introduces a novel encoding scheme for nested data structure
2. Embracing SQL
    - Google moved away from SQL at first to sacrifice for scalability. Dremel was one of the first systems to bring it back.

- Hierarchical schemas were a big departure from typical SQL. It ignores the normal form concept to reduce the JOIN operations, allowing Dremel to scale and perform faster and more efficient.
- Introduce GoogleSQL to address the variety of dialects

3. Disaggregation
    - Disaggregated storage
        - Initially implemented as shared-nothing servers. It is then moved to Borg. The golden formula are managed clusters and replicated storage.
        - However, as issues arise, they try to fix them, which ends up looking very similar to GFS.
        - Google then implemented a GFS-based Dremel system.
        - Dremel on disaggregated storage outperformed the local disk-based system in both latency and throughput.
            - Improve robustness, faster, easier to use, scalable,
    - Disaggregated memory
        - Use the shuffle primitive like in MR, which utilizes local RAM and disk to sort intermediate results (which is also a bottleneck)
        - Build a new disaggregated shuffle infrastructure using the Colossus DFS, which supports full in-memory query execution.
            - Reduced the shuffle latency
            - Enabled larger shuffles
            - Reduced resource cost
    - Observations

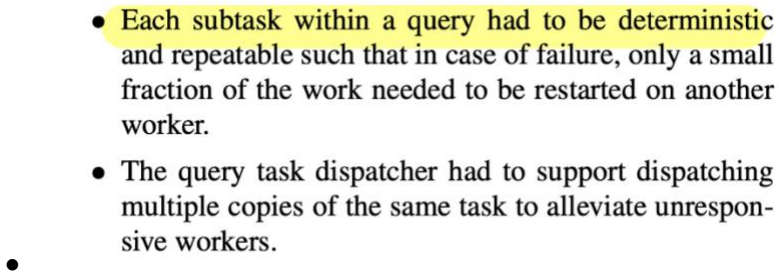
    - Disaggregation allows better cost-performance and elasticity
        - Economies of scale: from RAID disks to data warehouse
        - Universality: embraced by many systems
        - High-level APIs: access at a higher level of abstraction
        - Value-added re-packaging: raw resources are packaged as services providing enhanced capabilities

4. In situ data analysis refers to accessing data in its original place without upfront data loading and transformation steps
    - Dremel's listener evolution to in situ analysis
        - Dremel's initial design was reminiscent of traditional DBMS
        - Open-source Dremel's storage when migrating to GFS
            - Columnar
            - Self-describing
        - Each file stores a partition of a table
        - All Google services can now be run on GFS/Colossus without having to be manually loaded into a data warehouse.
        - The in situ approach is then developed into two directions
            - Adding file formats beyond the original columnar format
            - Expanding in situ analysis through federation: allowing Dremel to take advantage of other tools' strength
    - Drawbacks of in situ analysis
        - Don't always want to or have the capability to manage their own data safely and securely

- Cannot optimize storage layout or compute statistics in the general case

5. Serverless computing

- Serverless roots
  - Only possible using multi-tenant and provided on-demand resource planning
  - Three ideas
    - Diassgregation: Allows operations to be independent of the storage
    - Fault tolerance and restartability: assume that the workers are inherently unreliable



- Each subtask within a query had to be deterministic and repeatable such that in case of failure, only a small fraction of the work needed to be restarted on another worker.
- The query task dispatcher had to support dispatching multiple copies of the same task to alleviate unresponsive workers.

    - Virtual scheduling unit: designs to work with slots
  - Leads many enterprises to adopt the cloud over on-premises systems
- Evolution of serverless architecture
  - Centralized scheduling
    - Allows more fine-grained resource allocation and reservations
    - Use the entire cluster state to make scheduling decisions, which enables better utilization and isolation
  - Shuffle Persistence Layer
    - Allows decoupling scheduling and execution of different stages of the query
    - Can dynamically preempt workers, reducing resource allocation

- Flexible Execution DAGs
  - See paper
- Dynamic Query Execution
  - Choose a path where the query execution can dynamically change during runtime based on the execution statistics.

6. Columnar storage for nested data

- Overview
  - Repetition level specifies for repeated values whether each ancestor record is appended into or starts a new value
  - Definition level specifies which ancestor records are absent when an optional field is absent
  - E.g., Name.Language.Country
    - Maximum repetition level: 2 (Name and Language)
    - Maximum definition level: 3
    - Such a design is nice since it allows information within the column field to be accessed without reading the ancestor fields. However, this has redundant data storage.
- Embedded evaluation
  - Embed directly into the Capacitor data access library
  - It has a mini query processor which evaluates SQL predicates
  - Allow more efficient filter support in all data management applications.
    - Partition and predicate pruning:

        - Vectorization: columnar storage lends itself to columnar block-oriented vectorized processing
        - Skip-indexes: High selectivity requires a fast implementation of skipping where the records of implementation yield false.
            - The capacitor combines column values into segments.
            - The column contains an index with offsets pointing to the beginning of each segment.
        - Predicate reordering
            - Capacitor uses a number of heuristics to make filtering decisions
- Row reordering
    - Uses several standard techniques to encode values, including dictionary and run-length encodings
    - RLE is very sensitive to row ordering
    - It is an NP-complete problem, which is impractical
    - Not all columns are born equal: they can be different in size
    - Row reordering works are surprisingly good in practice
- More complex schema
    - Allow defining recursive message schemas. Capacitor now supports recursive of arbitrary depth.

7. Interactive query latency over big data
    - The three design principles seem to be counterproductive to building a system for interactive query latency.

- Latency-reducing techniques
    - Stand-by server pool: ready to process queries as soon as a job is submitted
    - Speculative execution: breaking one job into multiple smaller tasks so that slower machines only run fewer jobs and faster machines can run faster jobs
    - Multi-level execution trees: resolve tree coordination by using a tree architecture with a root server on top of intermediate servers on top of leaf servers
    - Column-oriented schema representation: the storage format is designed to be self-describing (using columnar format)
    - Balancing CPU and IO with lightweight compression: Using a columnar format makes compression more effective because similar values are stored sequentially.
    - Approximate results: Forgone absolute accuracy to reduce latency
    - Query latency tiers: Achieve high utilization in a shared server pool by supporting multiple users by issuing multiple queries immediately
    - Reuse of file operations: fetch in a batch at the root server and pass it through the execution tree to the leaf servers for data reads
    - Guaranteed capacity: reservations introduced with the centralized scheduler helped with latency
    - Adaptive query scaling: dynamically build an aggregation tree whose depth depends on the size of the input.

Answers to Questions

1. Explain Disaggregation
    - Disaggregation refers to separating memory from specific computing units, allowing it to have multi-tenant accessibility (independently). It works by saving the intermediate results (in-memory values) in a distributed, shared memory space/layer. These values are then accessible through multiple users at the same time. Disaggregation reduces data movement and, thus, improves performance, as well as making the entire operation more efficient.
2. Explain In Situ
    - In Situ refers to accessing data in its original place without upfront data loading and transformation steps into a specific format. At Google, In Situ allows the data to be processed using all of the company's services without having to load it into a data warehouse.
    - In Situ supports faster processing time by reducing preprocessing steps and being able to read diverse data formats.
    - However, it does not always want to or have the capability to manage their own data safely and securely. Also, it has challenges in optimizing performance and performing complex queries, something that is easier if we have a standardized database.
3. Describe Serverless
    - Serverless allows scalability since it is fully managed and has the ability to automatically provision the resources. Doing so can lower overhead costs and increase cost efficiency.

- Dremel enables serverless computing by abstracting the management of the server (by being fully managed and auto-provisioning), allowing the user to focus on querying the data without having to worry about the infrastructure. It also dynamically allocates resources and scales to meet query demands effectively, optimizing performance and cost.

Additional Questions and Observations

1. I don't have any significant thoughts or questions after this paper. However, I think this paper is easier to read as it summarizes and discusses the content from other papers. This is great as it does not introduce any ideas but rather offers a big picture of how things are connected to each other. I can also see how Spark has become the new standard for the industry given its ability to process things much more quickly.