DSC 102
Systems for Scalable Analytics

Arun Kumar

Topic 4: Dataflow Systems

Spark Book; Chapter 2.2 of ML Sys Book
Q: How to shield users from needing to think about moving raw pages between disk/RAM/network to scale data-intensive programs?
Parallel RDBMSs

- Parallel RDBMSs are highly successful and widely used
- Typically shared-nothing data parallelism
- Optimized runtime performance + enterprise-grade features:
  - ANSI SQL & more
  - Business Intelligence (BI) dashboards/APIs
  - Transaction management; crash recovery
  - Indexes, auto-tuning, etc.

Q: So, why did people need to go beyond parallel RDBMSs?

Ad: Take CSE 132C for more on parallel RDBMSs
Beyond RDBMSs: A Brief History

- DB folks got blindsided by the rise of Web/Internet giants

- 4 new concerns of Web giants vs RDBMSs built for enterprises:
  - **Developability**: Custom data models and computations hard to program on SQL/RDBMSs; need for simpler APIs
  - **Fault Tolerance**: Need to scale to 1000s of machines; need for graceful handling of worker failure
  - **Elasticity**: Need to be able to easily upsize or downsize cluster size based on workload
  - **Cost**: Commercial RDBMSs licenses too costly; hired own software engineers to build custom new systems
A new breed of parallel data systems called **Dataflow Systems** jolted the DB folks from being smug and complacent!
Outline

❖ Beyond RDBMSs: A Brief History
❖ MapReduce/Hadoop Craze
❖ Spark and Dataflow Programming
❖ Scalable BGD with MapReduce/Spark
❖ Dataflow Systems vs Task-Parallel Systems
What is MapReduce?

- A programming model for parallel programs on **sharded data + distributed system** architecture
- **Map** and **Reduce** are terms from functional PL; software/data/ML engineer implements logic of Map, Reduce
- System handles data distribution, parallelization, fault tolerance, etc. under the hood
- Created by Google to solve “simple” data workload: index, store, and search the Web!
- Google’s engineers started with MySQL! Abandoned it due to reasons listed earlier (developability, fault tolerance, elasticity, etc.)

What is MapReduce?

- **Standard example**: count word occurrences in a doc corpus
- **Input**: A set of text documents (say, webpages)
- **Output**: A dictionary of unique words and their counts

```java
function map (String docname, String doctext) :
    for each word w in doctext :
        emit (w, 1)

function reduce (String word, Iterator partialCounts) :
    sum = 0
    for each pc in partialCounts :
        sum += pc
    emit (word, sum)
```

Part of MapReduce API
How MapReduce Works

Parallel flow of control and data during MapReduce execution:

The overall MapReduce word count process

Under the hood, each Mapper and Reducer is a separate process; Reducers face barrier synchronization (BSP)

Fault tolerance achieved using data replication
**Goal:** High-level *functional* ops to simplify data-intensive programs

**Key Benefits:**
- Map() and Reduce() are highly general; any data types/structures; great for ETL, text/multimedia
- Native scalability, large cluster parallelism
- System handles fault tolerance automatically
- Decent FOSS stacks (Hadoop and later, Spark)

**Catch:** Users must learn “art” of casting program as MapReduce
- Map operates record-wise; Reduce aggregates globally
- But MR libraries now available in many PLs: C/C++, Java, Python, R, Scala, etc.
Abstract Semantics of MapReduce

- **Map()**: Process one “record” at a time *independently*
  - A record can physically *batch* multiple data examples/tuples
  - Dependencies across Mappers *not* allowed
  - *Emit* 1 or more key-value pairs as output(s)
  - Data types of input vs. output can be different

- **Reduce()**: Gather all Map outputs across workers sharing same key into an Iterator (list)
  - Apply *aggregation* function on Iterator to get final output(s)

- **Input Split**:
  - Physical-level shard to batch many records to one file “block” (HDFS default: 128MB?)
  - User/application can create *custom* Input Splits
Emulate MapReduce in SQL?

Q: How would you do the word counting in RDBMS / in SQL?

❖ **First step:** Transform text docs into relations and load:
  Part of the ETL stage
  Suppose we pre-divide each doc into words w/ schema:
  **DocWords** (DocName, Word)

❖ **Second step:** a single, simple SQL query!

```
SELECT Word, COUNT (*)
FROM DocWords
GROUP BY Word
[ORDER BY Word]
```

Parallelism, scaling, etc. done by RDBMS under the hood
More MR Examples: Select Operation

❖ Input Split:
  ❖ Shard table tuple-wise

❖ Map():
  ❖ On tuple, apply selection condition; if satisfies, emit KV pair with dummy key, entire tuple as value

❖ Reduce():
  ❖ Not needed! No cross-shard aggregation here

❖ These kinds of MR jobs are called “Map-only” jobs
More MR Examples: Simple Agg.

❖ Suppose it is *algebraic* aggregate (SUM, AVG, MAX, etc.)

❖ **Input Split:**
  ❖ Shard table tuple-wise

❖ **Map():**
  ❖ On agg. attribute, compute incr. stats; emit pair with single global dummy key and incr. stats as value

❖ **Reduce():**
  ❖ Since only one global dummy key, Iterator has *all* suff. stats to unify into global agg.
More MR Examples: GROUP BY Agg.

❖ Assume it is *algebraic* aggregate (SUM, AVG, MAX, etc.)

❖ **Input Split:**
  ❖ Shard table tuple-wise

❖ **Map():**
  ❖ On agg. attribute, compute incr. stats; emit pair with *grouping attribute as key and stats as value*

❖ **Reduce():**
  ❖ Iterator has all suff. stats *for a single group*; unify those to get result for that group
  ❖ Different reducers will output different groups’ results
Assume it is \textit{algebraic} aggregate (L_{p,q} norm)

Very similar to simple SQL aggregates

\textbf{Input Split:}
- Shard table tuple-wise

\textbf{Map():}
- On agg. attribute, compute incr. stats; emit pair with single global dummy key and stats as value

\textbf{Reduce():}
- Since only one global dummy key, Iterator has \textit{all} suff. stats to unify into global agg.
What is Hadoop then?

- FOSS system implementation with MapReduce as prog. model and HDFS as filesystem
- MR user API; input splits, data distribution, shuffling, fault tolerances handled by Hadoop under the hood
- Exploded in popularity in 2010s: 100s of papers, 10s of products
- A “revolution” in scalable+parallel data processing that took the DB world by surprise
- But nowadays Hadoop largely supplanted by Spark

NB: Do not confuse MR for Hadoop or vice versa!
Outline

❖ Beyond RDBMSs: A Brief History
❖ MapReduce/Hadoop Craze
❖ Spark and Dataflow Programming
❖ Scalable BGD with MapReduce/Spark
❖ Dataflow Systems vs Task-Parallel Systems
Apache Spark

- **Dataflow programming** model (subsumes most of RA; MR)
  - Inspired by Python Pandas style of chaining functions
  - Unified storage of relations, text, etc.; custom programs
  - System impl. (re)designed from scratch
- Tons of sponsors, gazillion bucks, unbelievable hype!
- **Key idea vs Hadoop**: exploit distributed memory to cache data
- **Key novelty vs Hadoop**: lineage-based fault tolerance
- Open-sourced to Apache; commercialized as Databricks
Distributed Architecture of Spark

https://spark.apache.org/docs/latest/cluster-overview.html
**Spark’s Dataflow Programming Model**

Transformations are relational ops, MR, etc. as functions

Actions are what force computation; aka *lazy evaluation*

<table>
<thead>
<tr>
<th>Transformations</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>map(f : T ⇒ U)</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>filter(f : T ⇒ Bool)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>flatMap(f : T ⇒ Seq[U])</code></td>
<td><code>RDD[T] ⇒ RDD[U]</code></td>
</tr>
<tr>
<td><code>sample(fraction : Float)</code></td>
<td><code>RDD[T] ⇒ RDD[T]</code> (Deterministic sampling)</td>
</tr>
<tr>
<td><code>groupByKey()</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, Seq[V])]</code></td>
</tr>
<tr>
<td><code>reduceByKey(f : (V, V) ⇒ V)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>union()</code></td>
<td><code>(RDD[T], RDD[T]) ⇒ RDD[T]</code></td>
</tr>
<tr>
<td><code>join()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]</code></td>
</tr>
<tr>
<td><code>cogroup()</code></td>
<td><code>(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]</code></td>
</tr>
<tr>
<td><code>crossProduct()</code></td>
<td><code>(RDD[T], RDD[U]) ⇒ RDD[(T, U)]</code></td>
</tr>
<tr>
<td><code>mapValues(f : V ⇒ W)</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, W)]</code> (Preserves partitioning)</td>
</tr>
<tr>
<td><code>sort(c : Comparator[K])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
<tr>
<td><code>partitionBy(p : Partitioner[El])</code></td>
<td><code>RDD[(K, V)] ⇒ RDD[(K, V)]</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>count()</code></td>
<td><code>RDD[T] ⇒ Long</code></td>
</tr>
<tr>
<td><code>collect()</code></td>
<td><code>RDD[T] ⇒ Seq[T]</code></td>
</tr>
<tr>
<td><code>reduce(f : (T, T) ⇒ T)</code></td>
<td><code>RDD[T] ⇒ T</code></td>
</tr>
<tr>
<td><code>lookup(k : K)</code></td>
<td><code>RDD[(K, V)] ⇒ Seq[V]</code> (On hash/range partitioned RDDs)</td>
</tr>
<tr>
<td><code>save(path : String)</code></td>
<td>Outputs RDD to a storage system, e.g., HDFS</td>
</tr>
</tbody>
</table>
Word Count Example in Spark

Spark RDD API available in Python, Scala, Java, and R

```
import scala.collection.JavaConverters

text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(line => line.split(" "))
   .map(word => (word, 1))
   .reduceByKey((a, b) => a + b)
counts.saveAsTextFile("hdfs://...")
```

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
   .map(word => (word, 1))
   .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
   .flatMap(s => Arrays.asList(s.split(" ")).iterator())
   .mapToPair(word => new Tuple2<>(word, 1))
   .reduceByKey((a, b) => a + b);
counts.saveAsTextFile("hdfs://...");
```

Spark DataFrame API of SparkSQL offers an SQL interface
Can also interleave SQL with DF-style function chaining!
Databricks now recommends SparkSQL/DataFrame API; avoid RDD API unless really needed!

**Key Reason:** Automatic query optimization becomes more feasible

AKA (painfully) re-learn 40 years of database systems research! :)

Query Optimization in Spark

- Common automatic query optimizations (from RDBMS world) are now performed in Spark’s Catalyst optimizer:
  - **Projection pushdown:**
    - Drop unneeded columns early on
  - **Selection pushdown:**
    - Apply predicates close to base tables
  - **Join order optimization:**
    - Not all joins are equally costly
  - Fusing of aggregates
  - ...


**Ad:** Take CSE 132C for more on relational query optimization
def add_demographics(events):
    u = sqlCtx.table("users")
    events \n        .join(u, events.user_id == u.user_id) \n        .withColumn("city", zipToCity(df.zip))  # Run udf to add city column
    events = add_demographics(sqlCtx.load("/data/events", "parquet"))
    training_data = events.where(events.city == "New York").select(events.timestamp).collect()
Reinventing the Wheel?
## Comparing Spark’s APIs

Check out TA’s PA 2 slides for more on Spark APIs

<table>
<thead>
<tr>
<th>Feature</th>
<th>RDD</th>
<th>DataFrame</th>
<th>Koalas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Level</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Named Columns</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Support for Query Optimization</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Programming Mode</td>
<td>map-reduce</td>
<td>Dataflow, SQL</td>
<td>Pandas-like</td>
</tr>
<tr>
<td>Best suited for</td>
<td>Unstructured data Low-level ops Folks who like func. PLs and MapReduce</td>
<td>Structured data High-level ops Folks who know SQL, Python, R</td>
<td>Structured data Lower barrier to entry for folks who only know Pandas or Dask</td>
</tr>
</tbody>
</table>

**Ad:** Take Yoav’s DSC 291 to learn more Spark programming
Spark-based Ecosystem of Tools

The Berkeley Data Analytics Stack (BDAS)
New Paradigm of Data “Lakehouse”

Data “Lake”: *Loose coupling* of data file format and data/query processing stack (vs RDBMS’s tight coupling); many frontends

... which too is a form of DBMS! 😊

Time for a rant...

A data problem is not "data independence" (which is about change), but about being able to get the task done first.

A data model feature

Those comments are doo doo...

Yes, agreed on your point. I put it more colorfully. 👇 A wheel is a wheel but we don't use an Egyptian chariot wheels for a car nor a car's for a space shuttle. 😊
References and More Material

❖ MapReduce/Hadoop:

❖ Spark:
❖ Online Guide: https://spark.apache.org/docs/2.1.0/sql-programming-guide.html
Outline

❖ Beyond RDBMSs: A Brief History
❖ MapReduce/Hadoop Craze
❖ Spark and Dataflow Programming
❖ Scalable BGD with MapReduce/Spark
❖ Dataflow Systems vs Task-Parallel Systems
Example: Batch Gradient Descent

\[ \nabla L(w^{(k)}) = \sum_{i=1}^{n} \nabla l(y_i, f(w^{(k)}, x_i)) \]

- Very similar to algebraic SQL; vector addition
- **Input Split**: Shard table tuple-wise
  - **Map()**: On tuple, compute per-example gradient; add these across examples in shard; emit partial sum with single dummy key
  - **Reduce()**: Only one global dummy key, Iterator has partial gradients; just add all those to get full batch gradient
Outline

❖ Beyond RDBMSs: A Brief History
❖ MapReduce/Hadoop Craze
❖ Spark and Dataflow Programming
❖ More Scalable ML with MapReduce/Spark
❖ Dataflow Systems vs Task-Parallel Systems
Dataflow Sys. vs. Task-Par. Sys.

❖ **Pros:**
  - Discussion in class

❖ **Cons:**
  - Discussion in class
More Specific to Spark vs. Dask?

❖ **Pros:**
  Discussion in class

❖ **Cons:**
  Discussion in class
Optional: Advanced examples of use of MapReduce to scale ML algorithms
Not included in syllabus
Primer: K-Means Clustering

❖ **Basic Idea:** Identify clusters based on Euclidean distances; formulated as an optimization problem

❖ **Llyod’s algorithm:** Most popular heuristic for K-Means

❖ **Input:** \( n \times d \) examples/points

❖ **Output:** \( k \) clusters and their centroids

1. Initialize \( k \) centroid vectors and point-cluster ID assignment

2. **Assignment step:** Scan dataset and assign each point to a cluster ID based on which centroid is *nearest*

3. **Update step:** Given new assignment, scan dataset again to recompute centroids for all clusters

4. Repeat 2 and 3 until convergence or fixed # iterations
K-Means Clustering in MapReduce

- **Input Split**: Shard the table tuple-wise
  - Assume each tuple/example/point has an *ExampleID*
- Need 2 jobs! 1 for Assignment step, 1 for Update step
- 2 external data structures needed for both jobs:
  - Dense matrix $A$: $k \times d$ centroids; ultra-sparse matrix $B$: $n \times k$ assignments
  - $A$ and $B$ first broadcast to all Mappers via HDFS; Mappers can read small data directly from HDFS files
  - Job 1 read $A$ and creates new $B$
  - Job 2 reads $B$ and creates new $A$
K-Means Clustering in MapReduce

- **A**: $k \times d$ centroid matrix; **B**: $n \times k$ assignment matrix

- **Job 1 Map()**: Read A from HDFS; compute point’s distance to all $k$ centroids; get nearest centroid; emit new assignment as output pair (PointID, ClusterID)

- No Reduce() for Job 1; new B now available on HDFS

- **Job 2 Map()**: Read B from HDFS; look into B and see which cluster point got assigned to; emit point as output pair (ClusterID, point vector)

- **Job 2 Reduce()**: Iterator has all point vectors of a given ClusterID; add them up and divide by count; got new centroid; emit output pair as (ClusterID, centroid vector)

Ad: Take Yoav’s DSC 291 to write more MR/Spark programs