6.830 Lecture 21 -- Spark **11/24/2014**

Spark is a dataflow computation model, much like MapReduce. Data flow just means a sequence of operations that data is pumped through (much like a relational query plan).

Let's review MapReduce:

Map Reduce
Idea: provide a way to distribute other apps over the Google cluster, and to compute things like the inverted index, etc (although they no longer us MR for this.)

What does map do?

Reads in a set of key, value pairs and produces a new set of key value pairs.
These are grouped by key and passed to reduce.

What does reduce do?

Reduce applies a reducing function merges all values for a given key together and produces zero or one output values for each key.

What SQL query is this equivalent to?

```
SELECT key, agg(value)    // reduce
GROUP BY key
FROM
(SELECT key,value FROM documents
WHERE cond(doc))  // map
```

Map: Doc Name, Doc Contents -> k1, v1
Reduce: k1,{v1..vn} -> k1, aggregate

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = length(values)
    Emit(AsString(result));
```

Show Example

What apps can we build:
- grep
- document processing
- seach index construction
... (joins, databases -- see Pig)

Widely used in settings where there is a big chunk of data that needs to be processed and transformed into something that, e.g., could be loaded into a database -- "ETL"

What's interesting about Hadoop?

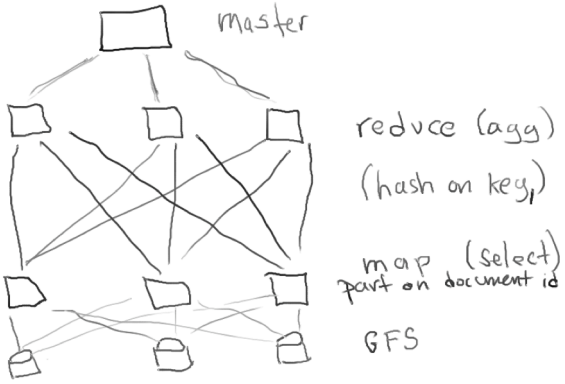
1) How to make it work in a distributed environment
2) How to tolerate faults
3) separation of concerns -- computation is separate from storage is separate from programming language -- databases tend to wrap all of these together

GFS:

networked file system distributed across all nodes.
each node can access all files that map applies to
files are replicated for fault tolerance

Distribution:

master and workers
master divvies up map tasks
they notify when done, writing results to local files
master divvies up reduce tasks
reduce workers read data from local FS of map workers, writing results back to GFS
try to preserve locality by scheduling map on the site that has the data.



Hadoop is an implementation of MapReduce used outside of Google (originally developed at Yahoo!). HDFS is the Hadoop equivalent of GFS.

Has become a punching bag for systems researchers, due to its bad performance.

What's inefficient about it? (say, in comparison to a relational database)

Possible bottlenecks:

- Use of sort on output of map tasks -- blocking, slow
- Writing of map tasks out to disk
- Writing of reduce results out to disk
- Lack of pre-aggregation
- Lack of indexing
- Long startup times
- …

Why is Hadoop so popular? Why has it become the face of big data?

Provides an easy way to parallelize computation across data that isn't already in a relational database. It's free, and it mostly works. Relational databases don't parallelize well and the good ones aren't free.

But Hadoop's performance is a dog.

This is where Spark comes in. Spark is a "data flow programming" language. Sort of like MapReduce, but with a much nicer syntax, quite similar to Pig or DryadLink (if you are familiar with it.)

Data is partitioned across multiple nodes, but can be programmed as though it is a single data file. [This is a cool abstraction but not really their idea]

Programmer can control partitioning

Let's look at an example (show slide).

What operations does spark support?

map (f) : [T] ==> [U]
filter (f : T => Bool) : [T] ==> [T]
group by : [(K,V)] ==> [(K, Seq[V])] ; one occurrence of each key in output
reduce by : [(K,V)] ==> [(K,V)] ; one occurrence of each key in output
union
join
...

Each operation runs on each input partition on its data set, and produces an output data set.

Key idea in this paper is "**Resilient Distributed Datasets**"

Idea is that you can create and load data sets into memory of a cluster and operate on them *without writing them back to disk*.

How does this help with performance? Intermediate results are in memory (unlike in MapReduce where every result set goes to disk.)

To provide fault tolerance, programmer can declare that an object be "persistent" meaning that it is kept in memory if possible so that other queries can access it (adds it to some sort of global catalog).

Persistent objects are cached, but can be evicted, in which case system can recompute them. In addition, if a node crashes, system can always recreate a cached RDD object. It does this by tracking the *lineage* of each data set, in terms of the computations that were used to generate it.

Idea is that each data set is *immutable* -- no updates are allowed, and we only create new copies of data sets. In this way all Spark has to do is remember the data sets and partitions and commands that contributed to each RDD, and it can regenerate any RDD (assuming that the base objects are stored persistently and reliably in some underlying file system like HDFS).

Example lineage graph (show slide).

Tracking lineage requires keeping a graph of dependencies as objects run.

Dependencies can be one-to-one (e.g., map) or many to one (e.g., join, reduce)

(Show slide)

What's actually involved in tracking these? Just keep a compiled representation of each task, and some information about whether they are cached or not.

Other topics: scheduling, memory management, checkpointing

Scheduling: each operator involves invoking a number of operations over data sets, which may or may not be in memory

1) what tasks to schedule next
2) where to schedule it

For 1)

Want to schedule tasks whose results are available in memory, while maintaining fairness.

They use a technique called *delay scheduling*, which does not schedule tasks that run on data not in memory until they have been waiting for more than some amount of time.

When data is not available in memory, re-run tasks needed to regenerate the data.

For 2) want to schedule tasks on nodes that have data in memory, obviously.

Memory management: What objects to keep in memory? Evict an object from the least recently accessed RDD, unless this is the one that is being loaded from (since this RDDs is likely to be scanned in its entirety at that point.)

Checkpointing. When persisting, have the option of writing RDDs to stable storage so they don't have to be recomputed.

What about very large RDDs (bigger than aggregate cluster memory)? Seems that all operators support on-disk RDDs, which work a lot like map/reduce -- records are read and processed one a time and (presumably) sorts are used for joins, group bys, and reduces.

Performance --

Versus Hadoop -- it's a lot faster than Hadoop -- 20x or more faster.

a) Startup costs (25s to start a job setup)
b) HDFS overheads -- even when storing in an in memory file system, Hadoop does a lot of copying of each block
c) Serialization / deserialization costs