Recap / finish dynamo

**Sloppy Quorum (healthy N)**

Dynamo authors don't think quorums are sufficient, for 2 reasons:
- Decreased durability (want to write all data at least 3 times)
- Decreased availability in the case of partitioning.

Example

N = 4, W = 3, R = 2  
Can't write if A & B are partitioned from C & D

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td>v1</td>
</tr>
</tbody>
</table>

In sloppy quorum, writes & reads just keeps going around the ring until it has written to / read from N nodes.

If a node not in the top N gets a write request for a key, the write will include a "hint" that has the original target for the node.

In example E gets request intended for D if D is unavailable. E will check to see if it can reach D periodically, and if so, will send the update to it (or reconcile). "Hinted handoff"

What's the problem with sloppy quorum? Can get divergent versions -- i.e., doesn't actually ensure we will always read the most recent version, even with R + W > N

Example

N = 4, W = 3, R = 2

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>v2</td>
<td>v2</td>
<td>v2'</td>
<td>v2'</td>
<td>v2'</td>
</tr>
</tbody>
</table>

A reader may contact C & B, see two different writes!

Hinted handoff works best with low churn and transient failure — with long permanent failures will lead to lots of inconsistent reads.

**How to detect conflicts?** Vector clocks to capture *causality* between versions. A vector clock is just a list of (node,counter) pair attached to each data item. Also called the write *context*. 
Example -- data item x  
\(N = 4, W = 3, R = 2\)  
Can't write if A & B are partitioned from C & D

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

\([A,1]\) \([A,1]\) \([A,1]\) \([A,1]\) \(\Rightarrow\) \([A,1]\) is the vector clock attached to this data item because A was the last coordinator for the write

\(\begin{array}{cc}
2 & 2 \\
\end{array}\) \(\begin{array}{c}
2 \\
1 \\
\end{array}\)

\([A,2]....\) \(\Rightarrow\) At this point, \([A,1] < [A,2]\), so these are causally related (D saw a subset of everyone else's writes

\(3\) \(3\) \(3\) \(3\) \(3\) \(3\) \(3\) \(3\)    \[A,3\] ... \[A,2],[C,1] ... \(\Rightarrow\) \([A,3]\) is incomparable to \([A,2],[C,1]\)  
Nodes on two sides of partition saw different writes

At this point, a read sent to B and C will get both versions, can tell they are incomparable, so must reconcile via read repair. Can either use latest writer, or do something application specific (e.g., for shopping cart union items in cart — won’t lose items, but deleted items can re-appear)

After reconciliation, write back. (Also do a write back if did a read and some node had an older version that was comparable.)

Partition heals -- B write back
\(4\) \(4\) \(4\) \(4\) \(4\) \(4\)
\([A,3]\) \([A,3],[C,1],[B,1]\) .... \([A,2],[C,1]\) \(\Rightarrow\) A,B,C,D have new value  
Note that E & F don’t get written to so still have some old version!

Although write back on read can help synchronize versions, want even more anti-entropy measures to ensure replicas stay in sync.

Could just compare the vector clocks of all data items between all nodes as a part of gossip, but that’d be a lot of data transfer. Instead, use a cute trick called a Merkle tree.

Idea is for each key range a node is responsible for it computes a hash tree, and compares that with other nodes also responsible for key range.

Which key ranges is a node responsible for?
If $N=3$, $A$ responsible for keys in dashed range

Ex for EA range, $B$ and $C$ are also responsible

What is a Merkle tree?

Suppose EA range has keys $u,v,w,x,y,z$

This whole tree is as big as data, but only need to exchange parts of it that are different, i.e., no need to send light gray nodes in diagram, since parent hashes are all equal
Summary:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitioning</td>
<td>consistent hashing</td>
<td>incremental scalability</td>
</tr>
<tr>
<td>highly available for writes</td>
<td>vector clocks with read repair</td>
<td>version size decoupled from update rate</td>
</tr>
<tr>
<td>handle temporary failures</td>
<td>sloppy quorum and hinted handoff</td>
<td>HA with some durability</td>
</tr>
<tr>
<td>recovery from permanent failures</td>
<td>anti-entropy with merkle trees</td>
<td>sync replicas</td>
</tr>
<tr>
<td>membership / failure detection</td>
<td>gossip based membership</td>
<td>symmetry and no centralized repo</td>
</tr>
</tbody>
</table>

Thoughts on this design?

**Spark**

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 use 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
- search 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 MapReduce?
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"

Key observation is that an intermediate object can always be recreated by re-running program to reconstruct it. "Lineage"

But reconstruction is expensive so instead you might want to keep intermediate results or entire load data sets in memory of 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.

How does it do this?
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.

Ok, so what is the actual execution model?

Operators related by narrow dependencies can be running in a pipelined fashion, since each output depends on only one input partition.

Operators connected by wide dependencies need to collect input from multiple partitions, which in general are on different machines. So need to perform a repartitioning of data.

These repartitioning operations for wide dependencies create the boundaries of so-called stages.

Program defines a sequence of stages and partitions that need to be computed. Stages form a DAG.

Writing a line of code in spark shell doesn't execute -- nothing is run until some output is requested -- just building up the DAG in memory.

All parent stages have to be completed computed before a child can be started, since in general the input of stage n depends on the input of one or more stages < n.

Show slide: -- here B is already cached (Black) so stage 1 can be skipped. Will first compute all partitions of Stage 2, e.g.,: Read C, apply map, read E 1st partition of E (2nd already in memory), apply union, etc.
Each circle here is an RDD, of a specific type (e.g., unionRDD). Each RDD type / instances defines methods to enumerate partitions, iterate over the entries of a particular partition, get the dependencies of the RDD, etc.

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.

As noted above, schedule stages from leaf to root in DAG.

**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