NoSQL & Cassandra Concepts

CMPT 732, Fall 2018

The Problem

So far, we have HDFS to store “files” but no way to read or write records: specifically-selected subsets of our data set that we can get at quickly.

That's usually the job of a database.

For big data, we need a database that will split the storage & queries across many nodes.

But full ACID guarantees are much harder in a distributed system where there is no central controller. Also, some operations are going to be hard in a distributed database…

Some DB Operations

- SELECT: easy. Each node can find the data it has. Calculation on data can be done locally.
- INSERT, DELETE: easy. Decide which node(s) should store that record and operate on it there.
- GROUP BY, JOIN: hard. Will require a lot of records from different (parts of different) tables. Leads to lots of network traffic.

We (probably) need something simpler than a traditional SQL database.

Non-Relational Databases

or NoSQL databases.

Motivated by the need for distributed databases: simplified data model, operations limited to what can be distributed efficiently.

e.g. Cassandra, CouchDB, Mongo, Redis, HBase.

NoSQL Limitations

There are many different technologies described as “NoSQL” but we generally have to give up:

- Some ACID guarantees.
- “Consistency” from CAP.
- Shuffling operations: JOIN, GROUP BY.
- SQL as the query language.

In exchange, we get:

- Can scale-out across a distributed cluster.
- Volume and Velocity.

CAP “Theorem”

You can have **at most two** of:

- Consistency: all nodes see the same data.
- Availability: reads/writes succeed (or obviously fail).
- Partition tolerance: keep running on network partition.

Traditional SQL databases are usually CA: don't handle P because there's only one server.

NoSQL + CAP

Any distributed system might partition: network failure between nodes/racks/data centres.

Consistency is hard in a truly distributed system:

> (circumference of earth)/(speed of light) = 134 ms

We're probably going to get AP for a distributed database.

NoSQL Categories

There are a wide variety of tools described as “NoSQL”, with varying degrees of quality. How useful they are will depend on their maturity and your problem.

Some rough categories…

Category	Data Model	Examples
Key-value	hash/dictionary	Redis, Membase, Amazon SimpleDB
Document	semi-structured ≈ JSON	Mongo, CouchDB, Amazon Dynamo
Wide-column	big tables	Cassandra, HBbase, Bigtable
Graph	graph	Neo4j, InfoGrid
Search	text-based document	Elasticsearch, Solr, Splunk

NewSQL

A distributed, relational, ACID database isn't actually impossible. The NewSQL databases aim to combine all of these.

Examples: Google Spanner, MemSQL, CockroachDB, MySQL Cluster.

Might not be *as* scalable or highly-availability as some of the NoSQL solutions, but you get to keep full SQL.

Cassandra

We will be using the *Cassandra* database. It's not the only non-relational database that makes sense for big data, but it's a good one.

- Distributed and decentralized: no node is in control, so no single point of failure.
- Non-relational: some concepts carry over from SQL, but not all. Specifically, no `JOIN`, very restricted *[added after the lecture]* `GROUP BY`.
- Fault tolerant: data is replicated, so failure can be handled gracefully.

The usual Cassandra setup is to have n nodes each acting as part of the cluster. They coordinate with each other and decide how to partition the data so it's evenly distributed and redundant.

A client can connect to any of them to make queries.

If we have many independent database nodes, it seems like a good match for YARN's many independent workers.

And it is: it's natural to create a MapReduce/Spark job that reads or writes Cassandra data. Each worker can read/write to a Cassandra node to distribute the load. The worker and database could even be on the same machine.

Cassandra Data Model

A Cassandra cluster has many *keyspaces*. Each keyspace has many *tables*.

A “table” is what you expect: columns of data; rows that have a cell for each column; columns have a fixed type; a *primary key* determinies how the data is organized.

The keyspace is a container for some tables but also…

A keyspace has rules about how it's replicated: can be a simple replication factor, or a description of how many replicas should be in each datacentre around the world.

For example, in the `cqlsh` shell:

```
cqlsh> CREATE KEYSPACE demo WITH REPLICATION =
   ... { 'class': 'SimpleStrategy', 'replication_factor': 3 };
cqlsh> USE demo;
cqlsh:demo> CREATE TABLE test (
        ... i1 INT, i2 INT, data TEXT,
        ... PRIMARY KEY (i1,i2) );
```

In this keyspace, every row will be replicated on three nodes.

The **first** part of the primary key is the *partition key*. It controls which node(s) store the record. Records with the same partition key will all be on the same nodes.

We had `PRIMARY KEY (i1,i2)`, so `i1` is the partition key. Any records with the same `i1` will be stored on the same nodes.

The primary key has to be unique: `(i1,i2)` must be unique for each row.

The first part of the primary key controls how data is partitioned: `i1`.

The primary key has two roles: unique row identifier, and decider of data partitioning.

Some of the data types you can have in Cassandra columns are what you'd expect: `INT`, `BIGINT`, `BLOB`, `DATE`, `TEXT`=`VARCHAR`.

Some you might not: `LIST<t>`, `SET<t>`, `MAP<t,u>`.

CQL

Cassandra's query language is CQL. It is not entirely unlike SQL.

Basically: it's as much like SQL as possible, while expressing Cassandra's semantics. We saw `CREATE TABLE`, which looks familiar.

Inserting and selecting seems to work the way you'd expect:

```
cqlsh:demo> INSERT INTO test (i1,i2,data) VALUES (1,2,'aaa');
cqlsh:demo> INSERT INTO test (i1,i2,data) VALUES (2,3,'bbb');
cqlsh:demo> SELECT * FROM test;

 i1 | i2 | data
----+----+------
  1 |  2 |  aaa
  2 |  3 |  bbb
```

... and it works like you'd expect, right up until it doesn't.

```
cqlsh:demo> SELECT * FROM test WHERE i1=1;

 i1 | i2 | data
----+----+------
  1 |  2 |  aaa

cqlsh:demo> SELECT * FROM test WHERE i2=2;
InvalidRequest: Error from server: code=2200 [Invalid query]
message=Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance. If you
want to execute this query despite the performance
unpredictability, use ALLOW FILTERING
```

Even though `i2` is part of the primary key, we haven't filtered by `i1`, so `WHERE i2=2` implies a full table scan.

The primary key determines the layout of data on the nodes/disk. Accessing data in any other way is potentially very expensive.

```
cqlsh:demo> SELECT * FROM test WHERE i2=2 ALLOW FILTERING;

 i1 | i2 | data
----+----+------
  1 |  2 |  aaa
```

CQL `INSERT` isn't really an insert: it's “insert or update by primary key”. Since we had `PRIMARY KEY (i1,i2)`:

```
cqlsh:demo> INSERT INTO test (i1,i2,data) VALUES (1,2,'ccc');
cqlsh:demo> SELECT * FROM test;

 i1 | i2 | data
----+----+------
  1 |  2 |  ccc
  2 |  3 |  bbb

(2 rows)
```

That implies that primary keys **must** be unique. Sometimes the easiest way around that is to just use a UUID.

```
cqlsh:demo> CREATE TABLE test2 ( id UUID PRIMARY KEY, data TEXT );
cqlsh:demo> INSERT INTO test2 (id,data) VALUES (UUID(), 'ddd');
cqlsh:demo> INSERT INTO test2 (id,data) VALUES (UUID(), 'eee');
cqlsh:demo> SELECT * FROM test2;

 id                                   | data
--------------------------------------+------
 403b9c83-fc57-4df4-a79b-d32fa66003fd |  ddd
 aefdcb9e-8d0c-4fee-9d12-5a2f2b12ecb6 |  eee

(2 rows)
```

You can `GROUP BY` in CQL, but **must include** the partition key. This allows aggregation to happen locally on each node, with no data shuffling.

This was added in Cassandra 3.10. [Our cluster is running 3.7.]

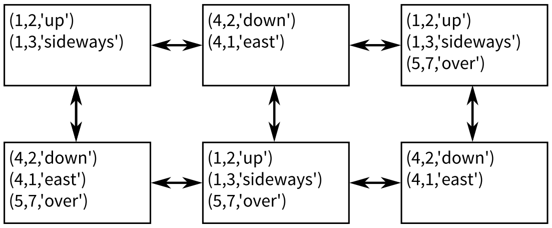
[added after the lecture]

Fault Tolerance

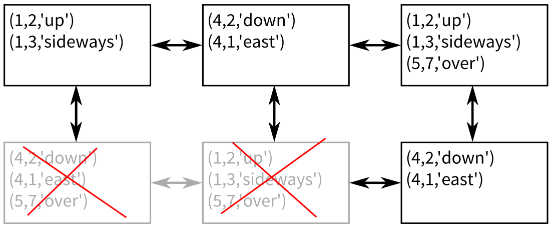
The replication factor in the keyspace lets Cassandra handle failures.

... and recover gracefully from failures of nodes or networks.

Copies of the data are made according to replication settings (3 here, with first field as partition key):



If n-1 nodes fail, we can still read and write data.



(1,2,'up')
(1,3,'sideways')

(4,2,'down')
(4,1,'east')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

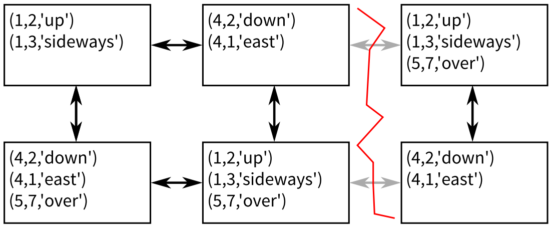
(4,2,'down')
(4,1,'east')
(5,7,'over')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

(4,2,'down')
(4,1,'east')

For writes, live nodes storing that partition key do the write; others catch up when they are back.

If the cluster *partitions*, we may still be able to read/write.



(1,2,'up')
(1,3,'sideways')

(4,2,'down')
(4,1,'east')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

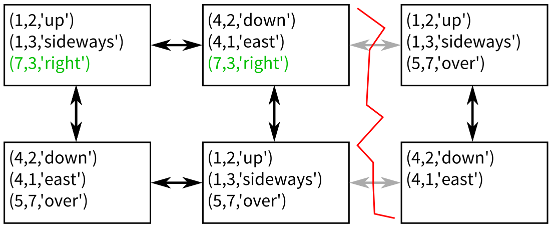
(4,2,'down')
(4,1,'east')
(5,7,'over')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

(4,2,'down')
(4,1,'east')

… datacentre-aware replication will help.

If there are writes, data will be inconsistent during the partition.



(1,2,'up')
(1,3,'sideways')
(7,3,'right')

(4,2,'down')
(4,1,'east')
(7,3,'right')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

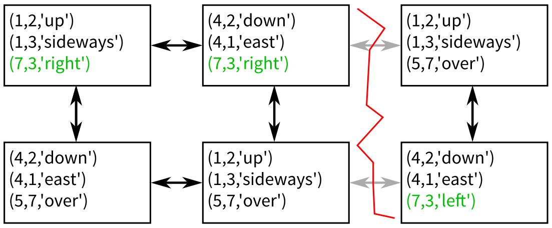
(4,2,'down')
(4,1,'east')
(5,7,'over')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

(4,2,'down')
(4,1,'east')

Again, writes will catch up when nodes can communicate.

What happens to conflicting writes during a partition?



(1,2,'up')
(1,3,'sideways')
(7,3,'right')

(4,2,'down')
(4,1,'east')
(7,3,'right')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

(4,2,'down')
(4,1,'east')
(5,7,'over')

(1,2,'up')
(1,3,'sideways')
(5,7,'over')

(4,2,'down')
(4,1,'east')
(7,3,'left')

Every cell in Cassandra has a timestamp: when it was inserted/updated.

```
cqlsh:demo> SELECT data, WRITETIME(data) FROM test;

 data | writetime(data)
------+------------------
  ccc | 1508624881598411
  bbb | 1508624823229052

(2 rows)
```

Most-recent timestamp wins if there's a consistency question.

Consistency

This means that Cassandra has *eventual consistency*: the data will be consistent, but with some delay.

Usually the delay will be short (network latency + milliseconds). If there's a network partition, it will be longer (after communication is restored).

When the cluster had failed nodes or a partition, we “may still be able to read/write”. “May”?

Cassandra gives us a choice of how consistent we demand to be with our reads and writes. We can have different requirements for consistency for each session/query.

The levels of consistency let you be very expressive about your requirements.

With the cluster partitioned, we would expect:

```
cqlsh:demo> CONSISTENCY ONE;
cqlsh:demo> SELECT * FROM test WHERE id=3; -- succeeds
cqlsh:demo> CONSISTENCY ALL;
cqlsh:demo> SELECT * FROM test WHERE id=3; -- fails
```

There are many options. For `SELECT`/read operations:

- `ONE`: any one node (with each record) can tell us.
- `TWO`: any two nodes (with each record) can tell us.
- `LOCAL_QUORUM`: >half of the nodes *in this data centre* with that data must respond.
- `QUORUM`: >half of nodes with that data must respond.
- `ALL`: every node with that data must respond.

Which you choose will depend how much consistency you want to wait for.

For `INSERT`/`UPDATE`/write operations to succeed/return:

- `ONE`: any one node must write.
- `TWO`: any two nodes must write.
- `LOCAL_QUORUM`: >half *in this data centre* that will store the data must write.
- `QUORUM`: >half of nodes that will store must write
- `EACH_QUORUM`: >half *in every data centre*.
- `ALL`: every node that will store this record must write.

Which consistency level you choose will likely depend on the application.

Cassandra gives the choice of “real” consistency if you need it (and are willing to trade working while partitioned), or you can trade speed for a low probability of data loss.

Relational Data

Since Cassandra has no `JOIN` operation, working with relational data is going to be tricky. We have lots of experience building tables that are related by foreign keys and joining them.

That was a good way to model lots of kinds of data, and a convenient way to work with it. Sadly, we have to give it up to easily do distributed computation.

If we have relational data, there are now several options to deal with it.

- Join the data in our programming language.
- Make a bunch of queries to get the data we need.
- Reshape the data so it can be efficiently queried with Cassandra.
- ⋮

Denormalizing Data

One common trick: abandon data normalization, and have more than one copy of a particular fact.

The goal is to have the data you need right there, with the other data you're about to query.

Example: the usual thing in a relational database would be to join these tables to produce a class list. *

Course	Grade	Student #
CMPT 123	B-	400000123
CMPT 123	A	400000124
CMPT 189	C-	400000124

Student #	Name
400000123	Vivian Hine
400000124	Charles Haydon

If joins are impossible/expensive and we know we need to produce class lists often, we could **store the table** as:

Course	Grade	Student #	Name
CMPT 123	B-	400000123	Vivian Hine
CMPT 123	A	400000124	Charles Haydon
CMPT 189	C-	400000124	Charles Haydon

Denormalizing the data effectively requires that we know ahead of time **what queries we will perform**. (e.g. if we didn't need to produce class lists, that structure would be stupid.)

When updating data, we have to be careful to update **every instance** of it. That could be very tricky.

Lesson: denormalizing data is a bad idea. But sometimes it's the least-bad idea available.

Idempotence

You might have learned the word in a linear algebra class, where it was some weird property a matrix could have: $MM = M$.

For us: an idempotent operation is one that can be done many times with the same effect: $f(x) = f(f(x))$. Still sounds like a piece of trivia.

An idempotent operation can be repeated if it **might have** failed. So if YARN (or similar) can't determine what happened with a node, it can send the task out to another node and let it run (possibly again).

This is a real possibility for us: node failure, job failure (because of memory limits), just random failures.

Idempotent operations: all pure functions; make sure today's data is in the database; running make/Ant/Maven; (over)write contents for `part-r-00026`; HBase's put/`UPSERT` in Phoenix/`INSERT` in Cassandra.

Not idempotent: *append* to a file; *add* today's data; SQL `INSERT`

We want to make sure our operations are idempotent to allow graceful (partial) failure. Hadoop's/Spark's file usage & Cassandra's `INSERT` semantics give us that for free.

Pure Functions

A *pure function* is one that…

- depends only on arguments (no external state),
- has no side effects (only returns a value).

In Hadoop/Spark, we are almost forced to write pure functions: there is no shared state between nodes to read; the only way to produce results was to return them.

All of our map/reduce/filter/… have been pure (except maybe random number generators and counters).

All mathematical functions are pure.

Also method calls like `o.hashCode()`, `o.getThing(x)`: think of the object as an argument to the function. Methods are non-pure if they modify the object's state: `o.setThing(x)`.

All pure functions are idempotent: they can be repeated/retried freely.

Parallelism is almost free for pure functions: with no side effects, many copies can execute concurrently with no worries. That's why we have been seeing them so much.