MongoDB and Cloud Storage

Frank Walsh

Agenda

- Cloud Databases
  - The CAP theorem
  - Distributed Storage
  - Distributed Data Models
- MongoDB
  - Querying
  - Integrating with Node.js
  - The Contacts API implementation

Databases in Enterprise Apps

- Most data driven enterprise applications need a database
- In traditional enterprise applications, this requires
  - Backups
  - Fail over
  - Maintenance
  - Capacity provisioning
- Usually handled by a Database Administrator.

Databases in the Cloud

- For some apps, a traditional database may not be the best fit
  - Does the app require transactional integrity?
  - Do you need db schema definition?
  - Do you know your scaling requirements, particularly if it's a web app?
- One approach is to use the **Cloud** for you DB
  - Designed for scale
  - Can be outsourced so you don't have to deal with infrastructure requirements.

Cloud DB Advantages

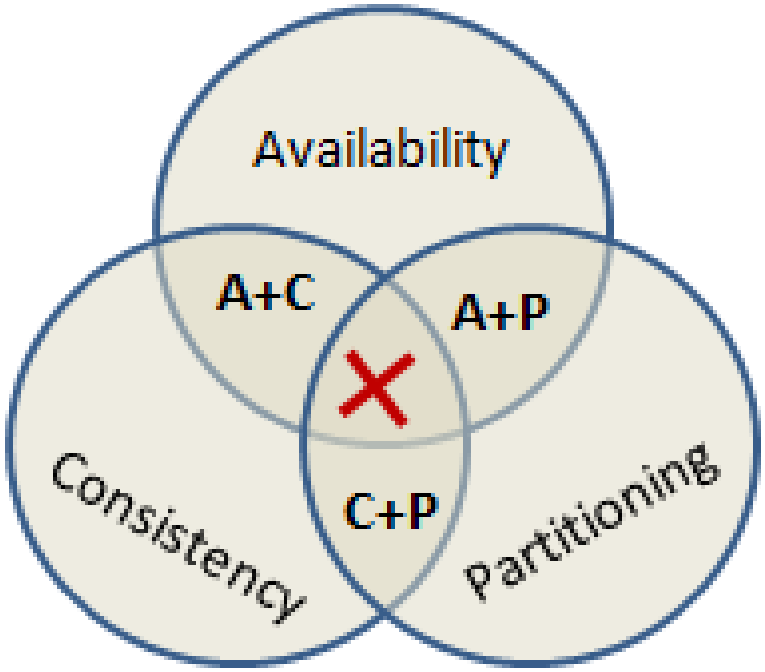
- Removes Management costs
- Inherently scalable
- Latency is predictable and constant
- No need to define schemas etc.
- Lots of Cloud DB offerings out there
  - SQL based
  - NoSQL based
- If organisation policy/standards do not allow outsourcing:
  - Can host yourself, most NoSQL DBs are free.

ACID

- Traditional databases offer ACID semantics for data transactions:
  - Atomicity: transactions must complete fully, or roll back entirely
  - Consistency: completed transactions must comply with all database schema constraints
  - Isolation: transactions should not interact with each other in non-deterministic ways
  - Durability: completed transactions are guaranteed to be permanently stored
- Retaining these characteristics in large scale distributed cloud databases is not feasible
  - transactions require data locking and/or versioning
- Even in traditional SQL databases, these constraints are often relaxed to gain speed and scale - e.g. different levels of isolation (dirty reads, etc.)

CAP (in theory)

- Distributed, scalable, cloud databases can't give you ACID semantics without making trade-offs.
- You can only have two of:
  - **Consistency**: clients always see the same data
  - **Availability**: clients can always get a response in reasonable time
  - **Partition tolerance**: the system keeps working even if parts of it are down



CAP (realistic approach…)

- The CAP trade-off was introduced as a conjecture by Eric Brewer (Inktomi co-founder) in 2000
- According to Brewer…
  the '2 of 3' view is misleading on several fronts.

1. *Partitions are rare, there is little reason to forfeit C or A when the system is not partitioned.*
2. *The choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved.*
3. *All three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.“*[1]

CAP and ACID…

- According to Brewer
- *In ACID, the C means that a transaction preserves all the database rules…., the C in CAP refers only to single-copy consistency, a strict subset of ACID consistency.*
- *ACID consistency also cannot be maintained across partitions; partition recovery will need to restore ACID consistency…..*
- *maintaining invariants during partitions might be impossible, thus the need for careful thought about which operations to disallow and how to restore invariants during recovery."*

ALPS Systems

1. **Availability**. All operations issued to the data store complete successfully. No operation can block indefinitely or return an error signifying that data is unavailable.
2. **Low Latency.** Client operations complete "quickly." Commercial service-level objectives suggest average performance of a few milliseconds and worse-case performance (i.e., 99.9th percentile) of 10s or 100s of milliseconds [16].
3. **Partition Tolerance.** The data store continues to operate under network partitions, e.g., one separating datacenters in Asia from Europe.
4. **High Scalability.** The data store scales out linearly. Adding N resources to the system increases aggregate throughput and storage capacity by O(N).
5. **Stronger Consistency**. An ideal data store would provide linearisability - sometimes informally called strong/strict consistency - which dictates that operations appear to take effect across the entire system at a single instance in time between the invocation and completion of the operation.

Cloud Database Practices

- Drop Consistency
  - this makes distributed systems much easier to build
- Drop SQL and the relational model
  - simpler structures are easier to distribute:
    - key/value pairs
    - **structured documents**
    - **pseudo-tables**
    - tend to be schema-free,accepting data as-is
- Offer HTTP interfaces using XML or **JSON**
- Use in-memory storage aggressively

Designing Distributed Data

- App data is not homogeneous
  - some kinds of data will be much larger

- consider using different databases for different requirements:
- user details,billing - needs consistency
  - require traditional database
- user data,content - needs partition tolerance
  - replicate to keep safe
- analytics,sessions - needs availability
  - "eventually consistent" is good enough

Denormalisation

- Join operations, even on traditional databases, have high overhead
- With large scale distributed databases: Assume you can only perform queries against one "table" at a time
- Place all the data you need in each table,where you need
  - "denormalize" your data by copying it where needed this does increase code complexity
  - often necessary to achieve desired performance

Sharding

- Once data becomes too big for one machine,what do you do?
  - break it into "shards" - subsets of the data and run each one as an independent database
- Sharding is typically performed using a hash function on a subset of data fields
- Shard resolution is handled by the client or hidden by a service gateway

Map-Reduce

- Primary data processing algorithm for distributed data
- First MAP data into required form, and then REDUCE it to get desired result:
- MAP: transform individual data entities
  - e.g.count number of words in a document
- REDUCE:collect output and summarize
  - e.g.total the word counts
- By splitting data into independent subsets, Map-Reduce can be performed in a parallel, fault tolerant manner
- Often the easiest way to perform complex queries and transformations on large distributed data sets
- http://tarnbarford.net/journal/mapreduce-on-mongo

MONGODB

Introduction

- Document-oriented database
  - but closer to traditional SQL databases than others
- Uses JSON natively - perfect fit for Node.js
- Query language with many SQL features
  - Uses JSON too, and has an easy learning curve
- Inbuilt sharding support means you can scale
  - Aggressively uses memory for high speed
- be careful:default configuration does not sync to disk
- Commercial support - 10gen.com product
  - cloud hosting providers - e.g.mongoLab.com
- Community support - popular choice

Mongo Terminology

- Each database contains a set of "Collections"
- Collections are analogous to SQL tables
- Collections contain a set of JSON documents
  - there is no schema
- the documents can all be different
  - means you have rapid development
  - adding a property is easy - just starting using in your code
- makes deployment easier and faster
  - roll-back and roll-forward are safe - unused properties are just ignored
- Collections can be indexed and queries
- Operations on individual documents are atomic

The MongoDB Query Language

- MongoDB provides a JavaScriptAPI and JSON-based query language
- Use the MongoDB console to execute queries
  - similar to usingMySQL console
- Access by running the mongo command
  - Example: list of employees

```
db.employees.find()
```

- db = current database
- employees = the employees collection
- .find() = collectionAPI method (coorresponds to collection URL in last lecture…)
- The Result Set is a list of JavaScript objects, representing matched documents

MongoDB: Inserts

- Collections do not need to be created explicitly
  - just insert a document
- MongoDB automatically assigns a 12 byte unique identifier to any document
  - the _id property
- Stored internally as binary
  - the ObjectId wrapper object is provided to work with these identifiers - i.e.specify them using hex strings

```
> db.city.insert({name:'Waterford',
country:'Ireland'})
> db.city.find()
{ "_id" : ObjectId("4f3a3f530b74e3768d4801ca"),
name:'Waterford', country:'Ireland'}
```

- See http://www.mongodb.org/display/DOCS/Inserting for more

MongoDB:Queries

- Documents are retrieved by specifying a set of conditions to match against
- simplest case:query-by-example
- provide a subset of properties that must match

```
> db.city.find( {name:'Waterford'} )
{ "_id" : ObjectId("4f3a3f530b74e3768d4801ca"),
name:'Waterford',country:'Ireland'}
```

- More complex queries use a convention of embedded meta- properties to specify conditions these are signified with a $ prefix Example:{name:{$exists:true}}

  returns documents that have a name property

MongoDB:Query Meta Properties

- Common meta-properties used with the update command are:
  - **$gt, $gte, $lt, $lte**
    meaning:
    >, >=, <,<=
    - Example:

      {price:{$gte:10, $lte:20}} //select documents where 10 <= price <= 20
  - **$and, $or**
    meaning
    and or
    - Example
      {$and: [{price:10},{price:20}]}
  - **$in, $nin**
    analagous to SQL IN, the property must be "in" or "not in" the array of values
    {price: $in:[10,20]}
  - **regular expressions**
    {word: /th^/i }
- See http://www.mongodb.org/display/DOCS/Advanced+Queries for more

MongoDB:Updates

- Documents are updated by providing:
  - a query to select the relevant subset of documents,and an update specification,which is either: a complete replacement document,or meta-properties that modify specific document properties
- example:
  $set changes specific properties
  Example:complete replacement:
  > db.city.insert({name:'dublin'})
  > db.city.update({name:'dublin'}, {name:'Dublin',county:'Dublin'})

- Example:modify specific properties:

  > db.city.insert({name:'Cork',county:'cork'})
  > db.city.update({name:'Cork'}, {$set:{county:'Cork'}})
- See http://www.mongodb.org/display/DOCS/Updating for more

MongoDB:Update Properties

- Common meta-properties used with the update command are:
  - **$set** - sets specified properties,but leaves others alone
    $set:{name:'New Name'}
- **$unset** - deletes specified properties
    $unset:{name:1}
- **$inc** - increments a numeric property
    inc:{ counter: 2 }
  adds 2 to the counter property, or if it does not exist, sets it to 2
- **$push, $pop** - add to or remove values from,an array
  - $push: { comments: {who:..., msg:...} }
  - $pop: {comments: -1 }

MongoDB:Upserts

- The MongoDB update command can optionally insert a document if it is not found. This is known as an 'upsert'
- This is useful when starting counters as it avoids corrupting the count when two independent updates try to initialize the counter

```
db.counters.update( {name:'foo'}, {$inc:{value:1}}, true)
```

- The first update will create the counter: {name:'foo', value:1}
- The second update will increment the counter: {name:'foo', value:2}

MONGO DB NODE.JS DRIVER

MongoDB Node.jsDriver

- To connect Node.js to MongoDB you need a *database driver*
- A Node.js module that communicates over the wire to MongoDB,and presents an API over the database.
- The one we'll look at available with NPM:

http://docs.mongodb.org/ecosystem/drivers/node-js/

```
npm install mongodb
var mongodb = require('mongodb')
```

node-mongodb-native

- Provides low-level interface to MongoDB and replicates the MongoDB Console
  - suffers badly from callbacks !
- Used by higher level MongoDB modules:
  - mongoose:Object Relational Mapper
  - mongoskin:simplerAPI to reduce callbacks
- Sufficient for smaller apps

Connecting to MongoDB using Node

```
var mongo = require('mongodb');
var mongoClient = mongo.MongoClient;
var mongoDb;
var url = 'mongodb://localhost:27017/myproject'
mongoClient.connect(url, function(err, db) {
      if(!err) {
            console.log("We are connected");
            mongoDb = db; }
      else
            {console.log("Unable to connect to
            the db");
      }
}
);
```

Inserting a Document

The following function that will insert a document:

```
if (mongoDb){
      var collection = mongoDb.collection('contacts');
      collection.insert(contact, {w:1}, function(err, result) {
            if (err) {
                console.log({'error':'An error has occurred'});
            } else {
                console.log('Success: ' + JSON.stringify(result[0]));}
        });
    }
  else
  {
    console.log('No database object!');
  }
```

- The insert function returns a result object that contains several fields that might be useful.
  - **result** Contains the result document from MongoDB
  - **ops** Contains the documents inserted with added **_id** fields
  - **connection** Contains the connection used to perform the insert

Updating a Document

- The following simple document update by adding a new field **b** to the document that has the field **a** set to **2**.

```
// Get the documents collection
  var collection = db.collection('documents');
  // Update document where a is 2, set b equal to 1
  collection.update({ a : 2 }
    , { $set: { b : 1 } }, function(err, result) {
console.log("Updated the document with the field a equal to 2");
    callback(result);
  });
```

- The method will update the first document where the field **a** is equal to **2** by adding a new field **b** to the document set to **1**.

Deleting a Document

- This will remove the first document where the field **a** equals to **3**.

```
var removeDocument = function(db, callback) {
  // Get the documents collection
  var collection = db.collection('documents');
  // Insert some documents
  collection.remove({ a : 3 }, function(err, result) {#
If (!err){
console.log("Removed the document with the field a equal to 3");}
}
else{
console.log(“Did not remove document");}

}
    callback(result);
  });
}
```