CS 133: Databases

Fall 2016
Lec 25 – 12/5
Prof. Beth Trushkowsky

Final exam timeline

• Take-home available: Friday Dec 9th, 4pm
• Due: Thursday, Dec 15th, 12pm (noon)
• Last class
  – Data analytics
  – Course evaluations
  – Course overview

Goals for Today

• Understand the motivation and goals for “NoSQL” data management systems

• Reason about the key concepts, techniques, and tradeoffs for NoSQL systems

• Touch on a couple specific NoSQL systems (Dynamo, MongoDB, Cassandra)

Some References Used

• *Ten Rules for Scalable Performance in “Simple Operation” Datastores*
  – Communications of the ACM 2011
  – Stonebraker and Cattell

• *Scalable SQL and NoSQL Data Stores*
  – SIGMOD Record 2011
  – Cattell

• *Dynamo: Amazon’s Highly Available Key-value Store*
  – SOSP 2007
  – DeCandia et al

• MongoDB and Cassandra web sites
Strong vs. Eventual Consistency

- **Strong**: after update to an object, subsequent reads see that update
  - *Eventual*: if updates ceased, eventually the system would reflect all updates

- **Eventual consistency** has some variation
  - Read-your-own-writes, special case of session or causal consistency
  - Monotonic reads

> "Eventually Consistent - Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability." - Vogels, CTO Amazon.com

- **BASE, not ACID!**
  - BASE: Basically available, soft state, eventually consistent

Werner Vogels on eventual consistency: [http://www.allthingsdistributed.com/2008/12/eventually_consistent.html](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)

What is “NoSQL”? A movement around **non-relational** data stores

- **Web 2.0**
  - Tons of user-generated content on the web
  - E.g., social networking sites
  - Web 1.0: few content creators, more static websites

- **Agile Development**
  - Develop and change web applications iteratively
  - E.g., schema changes and non-uniformity
  - Make changes while the application is live

- **Eventual Consistency** (BASE not ACID)
  - Performance gains at the expense of consistency
  - Growth incrementally by leveraging *leased* cloud resources like IaaS
  - Automatically grow resources as needed

Story of a “Successful” Web Startup

- **Start with a relational DBMS running on single machine**

  - Web site gets popular!!
  - Need to scale up...

    ... so manually partition/shard data across more nodes

- **Logic in web application manages directing queries**
  - Cross-shard filters and joins coded inside the app
  - App logic deals with data consistency

As the number of machines increases, the chance that *something* fails increases

Tons of NoSQL systems

- [**NoSQL**](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)
  - Your Ultimate Guide to the Non-Relational Universe
  - Including a historic *Archives* (2009-2011)
  - News feed covering every change in size

  "NoSQL" stands for Next Generation Databases, mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable.

  - The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often, other characteristics apply such as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge amount of data and more. So, the misleading term "NoSQL" (the community now translates it mostly with "Not only SQL") should be seen as an alias to something like the definition above. (based on 7 sources, 13 awesome feedbacks which thankret and a thinking common, agreed / disagreed?)

  - Register your event with a distributed systems event!

- **LIST OF NOSQL DATABASES** [currenty >225]
**CAP Theorem**

- “Eventually Consistent - Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.” – Vogels, Amazon CTO

- Eric Brewer’s CAP theorem: distributed system can **only have two** of the following three properties:
  - Consistency
  - Availability
  - Tolerance to network **Partitions**

**Summary: NoSQL Motivation**

- Development of NoSQL systems motivated by difficulty **scaling up Web 2.0 applications**
  - Thousands to millions of users
  - Many [small] reads and writes (“small operations”)

- Typically **make sacrifices for performance**
  - E.g., no ACID xacts, eventual consistency

**Exercise 2: Consistency Spectrum**

- Suppose have N replicas of some data object
  - W = # replicas write to before xact commits
  - R = # replicas read from

- Settings for W and R to favor C or A?

- Strong consistency: overlap the W and R sets
  - R + W > N
  - E.g., Read-one-write-all: R=1, W= N

**Achieving Scalable Performance**

- Rule #1: Shared-nothing **scalability**

- Rule #4: High **availability** and automatic recovery essential

- Rule #5: **On-line** everything (system always “up”)

- Rule #6: **Avoid multi-node** operators

---

Stonebraker and Cattell
Rule #4: High Availability and Auto-Recovery

- Goal: updates always succeed!
  - Issue: conflicting writes on disjoint sets of replicas

Exercise 3

- Example: shopping cart
  - Add 2 items to cart, update goes to two replicas
  - Partition! Add 1 (different) item to each replica
- Dynamo: vector clocks

- Or latest timestamp wins?

"...in the case of a timestamp tie, Cassandra follows two rules: first, deletes take precedence over inserts/updates. Second, if there are two updates, the one with the lexically larger value is selected."

http://wiki.apache.org/cassandra/FAQclocktie

Replication/Availability Examples

- MongoDB: automatic failover for primary

- Cassandra/Dynamo: peer-to-peer replication
  - tunable consistency, e.g., quorum or not
  - Option: “hinted handoff”

Rule #5: On-line Everything (Schema)

- Recall:
  - A data model is a collection of high-level data description constructs
  - A schema is a description of a particular collection of data, using a given data model

- Relational model has a rigid, structured schema
  - Attributes for relation pre-defined, shared by all tuples
  - Data and integrity constraints
  - Referential constraints
NoSQL: Non-Relational Data Models

- Agile development, live schema changes
  - No enforcement of structure
  - E.g., every “tuple” could have different attributes

- In essence, these data models are key-based
  - Key: some unique identifier to look up a corresponding “value”
  - What the value is can be complex

Key-Value Data Model

- Example system: Amazon’s Dynamo

- Key is some unique identifier, value can be anything, BLOB interpreted by app logic
  - E.g., id → shopping cart contents

- Query functionality
  - Get(key), put(key, value)
  - Only primary key index
  - No index lookups on non-keys (secondary indexes)

Document Data Model

- Example system: MongoDB

- Stores collections of “documents” (e.g., JSON)
  - Relation: tuple :: Collection: document
  - Key → Document
  - Document has key-value pairs, can be nested lists or scalars (and not defined in a global schema)

- Query functionality
  - Primary key lookups
  - Secondary indexes on other attributes

MongoDB Example 1

- Example: info about people who have multiple addresses

```javascript
db.createCollection("people")
db.people.insert({
  name: 'Alice',
  ssn: '123-456-7890',
  favorite-color: 'green'
  addresses: [
    { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
    { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
})
```

One-to-many relationship captured with embedded documents

Source: http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1
**MongoDB Example 2**

Example: info about products, which have many parts

```javascript
db.createCollection("parts")
db.createCollection("products")
```

// example part
```javascript
{  
  _id : ObjectID('AAAA'),
  partno : '123-aff-456',
  name : '#4 grommet',
  qty: 94,
  cost: 0.94,
  price: 3.99
}
```

// example product
```javascript
{  
  name : 'smoke shifter',
  manufacturer : 'Acme Corp',
  catalog_number: 1234,
  parts : [
    ObjectID('AAAA'),
    ObjectID('F17C'),
    ObjectID('D2AA'),
    // etc
  ],
  // etc
}
```

One-to-many relationship captured with references

Source: [http://blog.mongodb.org/post/87200945828/6HrulesHofHthumbHforHmongodbHschemaHdesignHpart1](http://blog.mongodb.org/post/87200945828/6HrulesHofHthumbHforHmongodbHschemaHdesignHpart1)

---

**Extensible Record (aka Column Family)**

- Example system: BigTable, Cassandra
- A bit more complex than document model
  - Relation: tuple :: ColumnFamily: Row
  - Key ➔ Set of columns ("wide-column store")
  - Each column has key-value pairs
  - Different records can have different columns
- Query functionality in “CQL”
  - Primary key lookups by row (with sorted columns)
  - Secondary indexes

---

**Cassandra Examples**

CREATE TABLE people {
  user_id text PRIMARY KEY,
  name text,
  addresses list
};

CREATE TABLE comments {
  article_id uuid,
  posted_at timestamp,
  author text,
  content text,
  PRIMARY KEY (article_id,posted_at)
};

<table>
<thead>
<tr>
<th>alicious</th>
<th>addresses</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>West</td>
<td>Alice</td>
<td></td>
</tr>
<tr>
<td>North</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drinkward</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bobtastic</th>
<th>addresses</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Lonely Ave</td>
<td>Bob</td>
<td></td>
</tr>
</tbody>
</table>

First argument is partition key, others are "clustering" columns

Roughly based on: [http://www.datastax.com/dev/blog/thrice/to-cql3](http://www.datastax.com/dev/blog/thrice/to-cql3)

---

**Canonical Social Networking App**

- People follow other people
- People post tweets/statuses/whatever
- People can view posts from the people they follow in a “timeline”

Exercise 4-5: Reasoning about schema
Rule #6: Avoid Multi-Node Queries

• No ACID transactions across primary keys

• No joins! Denormalization helps

• Systems offer different levels of “protection”
  – Key-value stores: get(key) method requires key
  – MongoDB: Table scans discouraged
  – Cassandra: Table scans prohibited

MongoDB: “Social Inboxes”

```javascript
// Shard on "recipient","sent" — efficient lookup by recipient
db.shardCollection("inbox", {"recipient": 1, "sent":1})

msg = {
  from: “Carl”,
  to: ["Bob", “Alice”]
  sent: new Date()
  message: “Hi!
,}

// Send the message to all recipients
for (recipient in msg.to)
  msg.recipient = msg.to[recipient]
  db.inbox.insert(msg);

// Read Bob’s inbox
db.inbox.find({"recipient": "Bob"}).sort({sent:-1})
```

Source: http://blog.mongodb.org/post/65612078649/schema-design-for-social-inboxes-in-mongodb

Cassandra: Achieving “Twissandra”

• Modeling Twitter-like functionality using Cassandra

• Storing tweets:

```cql
CREATE TABLE tweets (  
tweetid uuid PRIMARY KEY,  
username text,  
body text  
);
```


Cassandra: Achieving “Twissandra”

• Tweets from each user; used to look at my own tweets

```cql
CREATE TABLE userline(  
username text,  
body text,  
tweetid timeuuid,  
PRIMARY KEY(username, tweetid)  
);
```

• Tweets from people users follow; used to look at my friends’ tweets

```cql
CREATE TABLE timeline(  
username text,  
body text,  
posted by text,  
tweetid timeuuid,  
PRIMARY KEY(username, tweetid)  
);
```

Cassandra: Achieving “Twissandra”

- Data redundancy!

BEGIN BATCH
  INSERT INTO tweets ...
  INSERT INTO userline ...
  INSERT INTO timeline ...
  INSERT INTO timeline ...
APPLY BATCH

- Cassandra promises atomicity for batches
  - But not isolation

Rule #1: Shared-nothing scalability

- Goal: as application grows, need more servers added seamlessly
  - Don’t want manual management of scaling up

- Example techniques:
  - Consistent hashing (Dynamo and Cassandra)
  - Periodic re-balancing of partitions (MongoDB)

Consistent Hashing (in Dynamo)

- Range of a hash function is treated as a fixed circular space
- Variant: each physical node assigned several spots on the ring
- Data item replicated at N spots along the ring

Shard Re-balancing (in MongoDB)

- Collection might be range-partitioned across shards
- When shard has too many “chunks”:
  - Splitter re-associates chunks to two shards
  - Background process actually re-balances the data