CSE 544
Principles of Database Management Systems

Alvin Cheung
Fall 2015
Finale – NoSQL
Poster Session Information

- Time: 12/15 (next Tuesday!), 2-4:30pm
- Place: CSE first floor atrium

- Easels: will be provided, please set up in the atrium
- Poster boards: poster printer room (CSE 415) or 1st floor copy room (next to CSE main office)
- Anyone planning to do demos?

- Format: please prepare an 8 min talk-through of your poster
  - We have 14 groups to go through

- There will be a best poster award !!!
Final Reports

• Final report due on 12/18, 11:45 pm
  – Turn in on dropbox
  – Instructions and format on website

• No class on Thursday
  – This will be the last 544 lecture 😞
References

- **Scalable SQL and NoSQL Data Stores**, Rick Cattell, SIGMOD Record, December 2010 (Vol. 39, No. 4)

- **Dynamo: Amazon’s Highly Available Key-value Store**. By Giuseppe DeCandia et. al. SOSP 2007.

- Online documentation: Amazon DynamoDB.

- Online documentation: MongoDB.
Outline

• NoSQL overview

• Two example systems
  – Amazon Dynamo
  – MongoDB
NoSQL Overview
NoSQL Motivation

• Originally motivated by Web 2.0 applications
  – Examples?

• Goal is to scale simple OLTP-style workloads to thousands or millions of users

• Users are doing both updates and reads
Why NoSQL as the Solution?

• Hard to scale *transactions*
  – Need to partition the database across multiple machines
  – If a transaction touches one machine, life is good
  – If a transaction touches multiple machines, ACID becomes extremely expensive! Need two-phase commit as we saw

• Replication
  – Replication can help increase throughput and lower latency
  – Create multiple copies of each database partition
  – Spread queries across these replicas
  – Easy for reads
  – But writes are expensive! (remember all the log shipping business)
NoSQL Key Feature Decisions

• **Want a data management system that is**
  – Elastic and highly scalable
  – Flexible (different records have different schemas)

• **To achieve above goals, willing to give up**
  – Complex queries: e.g., give up on joins
  – Multi-object transactions
  – ACID guarantees: e.g., *eventual consistency* is OK
    • Eventual consistency: If updates stop, all replicas will *converge* to the same state and all reads will return the same value
  – *Not all NoSQL systems give up all these properties*

All updates *eventually* reach all replicas
“Not Only SQL” or “Not Relational”

Six key features:
1. Scale horizontally “simple operations”
   - key lookups, reads and writes of one record or a small number of records, simple selections
2. Replicate/distribute data over many servers
3. Simple call level interface (contrast w/ SQL)
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schema
ACID vs BASE

- ACID = Atomicity, Consistency, Isolation, and Durability

- BASE = Basically Available, Soft state, Eventually consistent
NoSQL Data Models

- **Tuple** = row in a relational db

- **Extensible record** = families of attributes have a schema, but new attributes may be added

- **Document** = nested values, extensible records (think XML, JSON, attribute-value pairs)

- **Object** = like in a programming language, but without methods
Different Types of NoSQL

Taxonomy based on data models:

• **Key-value stores**
  – e.g., Project Voldemort, Memcached

• **Extensible Record Stores**
  – e.g., HBase, Cassandra, PNUTS

• **Document stores**
  – e.g., CouchDB, MongoDB

• **New types of RDBMSs.. not really NoSQL, not just SQL**
  – (not exactly sure what they are..)
Key-Value Stores: Dynamo
Key-Value Store: Dynamo

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

• **Main observation:**
  – “There are many services on Amazon’s platform that only need **primary-key access** to a data store.”
  – Best seller lists, shopping carts, customer preferences, session management, sales rank, product catalog
Basic Features

• **Data model**: (key, value) pairs
  – Values are binary objects (blobs)
  – No further schema

• **Operations**
  – Insert, delete, and lookup operations on keys
  – No operations across multiple data items

• **Consistency**
  – Replication with eventual consistency
  – Goal is to NEVER reject any writes (bad for business)
    • That’s why conflict resolution is pushed to reads
  – Multiple versions with conflict resolution during reads
Operations

• **get(key)**
  - Locates object replicas associated with *key*
  - Returns a single *object*
  - Or a list of objects with conflicting versions
  - Also returns a *context*
    • Context holds metadata including version
    • Context is opaque to caller

• **put(key, context, object)**
  - Determines where replicas of object should be placed
  - Location depends on key value
  - Data stored persistently including context
Storage: Distributed Hash Table

Implements a distributed storage engine:
- Each key-value pair \((k,v)\) is stored at some server \(h(k)\)
- API: \(\text{write}(k,v); \ \text{read}(k)\)

Use standard hash function: service key \(k\) by server \(h(k)\)
- Problem 1: a client knows only one server, doesn’t know how to access \(h(k)\)

- Problem 2. if new server joins, then \(N \rightarrow N+1\), and the entire hash table needs to be reorganized

- Problem 3: we want replication, i.e., store the object at more than one server
Distributed Hash Table

h = 2^n - 1  h = 0

Responsibility of A

Responsibility of B

Responsibility of C

Responsibility of D
Distributed Hash Table Details

- This type of hashing called "consistent hashing"

- Basic approach leads to load imbalance
  - Why?
  - Solution: Use V virtual nodes for each physical node
  - Virtual nodes provide better load balance
  - Number of virtual nodes can vary based on capacity
Problem 1: Routing

A client doesn’t know server h(k), but some other server

• Naive routing algorithm:
  – Each node knows its neighbors
  – Send message to nearest neighbor
  – Hop-by-hop from there
  – Obviously this is $O(n)$, so no good

• Better algorithm: “finger table”
  – Memorize locations of other nodes in the ring
    – $a, a + 2, a + 4, a + 8, a + 16, \ldots a + 2^n - 1$
  – Send message to closest node to destination
  – Hop-by-hop again: this is $\log(n)$
Problem 1: Routing

\[ h=2^n-1 \quad h=0 \]

Read(k) !

h(k) handled by server G

Client only “knows” server A

O(log n)
Problem 2: Joining

When X joins:
- select random ID

Responsibility of D
Problem 2: Joining

When X joins:
select random ID

Responsibility of D
Problem 2: Joining

When X joins:

- select random ID

Redistribute the load at D

Responsibility of X

Responsibility of D
Problem 3: Replication

• Need to have some degree of replication to cope with node failures

• Let N=degree of replication

• Assign key k to h(k), h(k)+1, ..., h(k)+N-1
Problem 3: Replication

$h = 2^n - 1$

$h = 0$

Responsibility of A, B, C

Responsibility of B, C, D

Responsibility of C, D, A
Additional Dynamo Details

• Each key assigned to a coordinator
• Coordinator responsible for replication
  – Replication skips virtual nodes that are not distinct physical nodes
• Set of replicas for a key is its preference list
• One-hope routing:
  – Each node knows preference list of each key
• “Sloppy quorum” replication
  – Each update creates a new version of an object
  – Vector clocks track causality between versions
Vector Clocks

• An extension of Multiversion Concurrency Control (MVCC) to multiple servers

• Standard MVCC:
  each data item X has a timestamp t:
  \(X_4, X_9, X_{10}, X_{14}, ..., X_t\)

• Vector Clocks:
  X has set of [server, timestamp] pairs
  \(X([s1,t1], [s2,t2], ...)\)
Vector Clocks

Figure 3: Version evolution of an object over time.
Vector Clocks: Example

- A client writes D1 at server SX:
  \[ \text{D1 ([SX,1])} \]
- Another client reads D1, writes back D2; also handled by server SX:
  \[ \text{D2 ([SX,2]) (D1 garbage collected)} \]
- Another client reads D2, writes back D3; handled by server SY:
  \[ \text{D3 ([SX,2], [SY,1])} \]
- Another client reads D2, writes back D4; handled by server SZ:
  \[ \text{D4 ([SX,2], [SZ,1])} \]
- Another client reads D3, D4: CONFLICT!
Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td></td>
</tr>
</tbody>
</table>
## Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td></td>
</tr>
</tbody>
</table>
## Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>No</td>
</tr>
</tbody>
</table>
**Vector Clocks: Conflict or not?**

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,10])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td></td>
</tr>
</tbody>
</table>
### Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,10])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,10])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3],[SY,10])</td>
<td>([SX,3],[SY,20],[SZ,2])</td>
<td></td>
</tr>
</tbody>
</table>
Vector Clocks: Conflict or not?

<table>
<thead>
<tr>
<th>Data 1</th>
<th>Data 2</th>
<th>Conflict ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3])</td>
<td>([SX,5])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,6])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>No</td>
</tr>
<tr>
<td>([SX,3],[SY,10])</td>
<td>([SX,3],[SY,6],[SZ,2])</td>
<td>Yes</td>
</tr>
<tr>
<td>([SX,3],[SY,10])</td>
<td>([SX,3],[SY,20],[SZ,2])</td>
<td>No</td>
</tr>
</tbody>
</table>
Executing Writes and Reads

• Write operations
  – Initial request sent to coordinator
  – Coordinator generates vector clock & stores locally
  – Coordinator forwards new version to all N replicas
  – If at least W-1 < N-1 nodes respond then success!

• Read operations
  – Initial request sent to coordinator
  – Coordinator requests data from all N replicas
  – Once gets R responses, returns data

• Sloppy quorum: Involve first N *healthy* nodes
Amazon DynamoDB

Additional functionality:
• Offers choice of eventual consistent vs strongly consistent read
• Offers secondary indexes to enable queries over non-key attributes
  – So can support selection queries

When would you want to use DynamoDB?

Try it at: http://aws.amazon.com/dynamodb/
Document Stores: MongoDB
Popular NoSQL System

From Wikipedia:


- As of February 2015, MongoDB is the fourth most popular type of database management system, and the most popular for document stores.
Data Model

http://docs.mongodb.org

- Data model
  - Collections of documents (analogue of a table)
  - BSON documents (attribute-value pairs with nesting and arrays)
  - Documents can reference each other
    - Apps must issue follow-up queries to resolve the references
  - Documents can be embedded in each other (i.e., nested)
    - But then have to worry about documents getting very large
Consistency and Replication

• Consistency
  – Write operations are atomic at the document level
  – Operations that modify more than a single document in a collection still operate on one document at a time

• Replication
  – Master scheme with master performing all reads & writes by default
    • Achieves strong consistency by always going through the master
  – Eventual consistency by default when reading from replicas
    • Updates propagate to replicas asynchronously
    • But can be configured to use synchronous replication
Programming Model

- Javascript API and Javascript shell
- Querying: Selection
  - A query targets a collection of documents
  - Selection queries on attribute values (including arrays)
    \[
    \text{db.inventory.find( \{ type: "snacks" \} )}
    \]
  - Can also have conditions on embedded documents
  - Can also do projections, sort, limit and skip
- Querying: Aggregation
  - Aggregation pipelines
    \[
    \text{db.orders.distinct("cust_id")}
    \]
  - MapReduce API (JavaScript map/reduce functions)
Query Optimization

• Sharding
  – Horizontal partitioning
  – User selects the “shard key”

• Indexing
  – MongoDB automatically indexes the ID field
  – Users can add indexes on other attributes

• Other interesting features
  – Can insert data and specify a “time to live” to expire docs
    • Why is that useful?
Takeaway

• Claim: NoSQL is really no SQL

• We have seen noSQL systems with:
  – Relational data models
  – Consistency support
  – Replication support
  – Querying interface
  – Optimization engine

• We have encountered all these during this quarter
That’s it!

- What you achieved in 10 weeks:
  - Various data models
  - Processing and optimizing SQL queries
  - Data analytics
  - Transactions
  - Replication and recovery
  - DaaS and real-world DBMS

- Thanks for everything
  - Have fun finishing your projects
  - Have a great winter break!