CC5212-1
PROCESAMIENTO MASIVO DE DATOS
OTOÑO 2015

Lecture 9: NoSQL I

Aidan Hogan
aidhog@gmail.com

Information Retrieval:
Storing Unstructured Information

BIG DATA:
STORING STRUCTURED INFORMATION

Relational Databases

Relational Databases:
One Size Fits All?

"One Size Fits All?" An Idea Whose Time Has Come and Gone

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, MIT, and
Research and Science, Inc.
stonebraker@csail.mit.edu

Ulle Endriss
Department of Computer Science
Bremen University, and
Research and Science, Inc.
ulle@cs.bremen.de

Abstract
The last 20 years of commercial DBMS development have been a complex interplay of forces: new applications, networks, mobile and wireless, new data types, cloud computing, and more. Many of these factors have led to the emergence of new ideas, such as NoSQL databases, column stores, and even new approaches to traditional relational databases.

To succeed, new approaches to database management systems must balance the needs of both scalability and flexibility. This talk will explore some of the challenges facing new database systems, and how they can be designed to meet these needs.

To succeed, new ideas must be validated in the real world. This talk will also present some of the latest research on the performance and scalability of new database systems, and how they can be used to drive innovation in the field.

The future of database management systems is bright, with new ideas and technologies driving the field forward. This talk will provide an overview of the current state of the art, and discuss some of the challenges and opportunities facing the field.

Information Retrieval: Storing Unstructured Information

Keywords:
Information overload, ranking, lemmatization, compression, pagerank, keywords, tf-idf, hepph-low, zipf, robports, site-map, DBMS, cosine, relevance, crawling, search, posting-lists, term-frequency, alias-encoding.
RDBMS: Performance Overheads

- Structured Query Language (SQL):
  - Declarative Language
  - Lots of Rich Features
  - Difficult to Optimise!
- Atomicity, Consistency, Isolation, Durability (ACID):
  - Makes sure your database stays correct
  - Even if there’s a lot of traffic!
  - Transactions incur a lot of overhead
    - Multi-phase locks, multi-versioning, write ahead logging
  - Distribution not straightforward

Transaction overhead: the cost of ACID

- 640 tps for system with transactional support
- 12,700 tps for system without logs, transactions or lock scheduling

RDBMS: Complexity

ALTERNATIVES TO RELATIONAL DATABASES FOR QUERYING BIG STRUCTURED DATA?

NoSQL

The Database Landscape
NoSQL: CAP (not ACID)

**CA**: Guarantees to give a correct response but only while network works fine (Centralised / Traditional)

**CP**: Guarantees responses are correct even if there are network failures, but response may fail (Weak availability)

**AP**: Always provides a "best-effort" response even in presence of network failures (Eventual consistency)

(No intersection)

NoSQL: KEY–VALUE STORE

The Database Landscape

- Not using the relational model
- Batch analysis of data
- Non-relational Databases with focus on scalability to compete with NoSQL, while maintaining ACID
- Stores documents (semi-structured / values)
- Not only OLAP
- Maps
- Graph-structured data
- In-memory

NoSQL

- **Distributed!**
  - Sharding: splitting data over servers "horizontally"
  - Replication

- **Lower-level** than RDBMS/SQL
  - Simpler ad hoc APIs
  - But you build the application (programming not querying)
  - Operations simple and cheap

- **Different flavours** (for different scenarios)
  - Different CAP emphasis
  - Different scalability profiles
  - Different query functionality
  - Different data models

http://db-engines.com/en/ranking
Key–Value Store Model

It’s just a Map / Associate Array 😊

- put(key, value)
- get(key)
- delete(key)

But You Can Do a Lot With a Map

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>country:Afghanistan</td>
<td>capital@city:Kabul,continent:Asia,pop:311087742011</td>
</tr>
<tr>
<td>country:Albania</td>
<td>capital@city:Tirana,continent:Europe,pop:301140542013</td>
</tr>
<tr>
<td>city:Kabul</td>
<td>country:Afghanistan,pop:347600042013</td>
</tr>
<tr>
<td>city:Tirana</td>
<td>country:Albania,pop:301140542013</td>
</tr>
<tr>
<td>user:10239</td>
<td>basedIn@city:Tirana,post:{103,10430,201}</td>
</tr>
</tbody>
</table>

... actually you can model any data in a map (but possibly with a lot of redundancy and inefficient lookups if unsorted).

THE CASE OF AMAZON

The Amazon Scenario

- Products Listings: prices, details, stock
- Customer info: shopping cart, account, etc.
- Recommendations, etc.
The Amazon Scenario

- Amazon customers:

![Graph showing Amazon transactions growth](image)

Key–Value Store: Amazon Dynamo(DB)

Goals:
- Scalability (able to grow)
- High availability (reliable)
- Performance (fast)

Don’t need full SQL, don’t need full ACID

Databases struggling ...

But many Amazon services don’t need:
- SQL (a simple map often enough)
- or even:
- transactions, strong consistency, etc.

Key–Value Store: Distribution

How might a key-value store be distributed over multiple machines?

- Or a custom partitioner...

What happens if a machine joins or leaves half way through?

- Or a custom partitioner...

![Diagram of distributed key-value store](image)
Key–Value Store: Distribution

How can we solve this?

Or a custom partitioner ...

\[ \text{mod}(\text{hash}([\text{key}]), m) \]

Consistent Hashing

Avoid re-hashing everything

- Hash using a ring
- Each machine picks \( n \) psuedo-random points on the ring
- Machine responsible for arc after its point
- If a machine leaves, its range moves to previous machine
- If machine joins, it picks new points
- Objects mapped to ring

How many keys (on average) need to be moved if a machine joins or leaves?

Amazon Dynamo: Hashing

- Consistent Hashing (128-bit MDS)

Key–Value Store: Replication

- A set replication factor (here 3)
- Commonly primary / secondary replicas
  - Primary replica elected from secondary replicas in the case of failure of primary

Amazon Dynamo: Replication

- Replication factor of \( n \)
  - Easy: pick \( n \) next buckets (different machines!)

Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - PUT doesn’t overwrite: pushes version
  - GET returns most recent version

Key–Value Store: Distribution

Or a custom partitioner ...

\[ \text{mod}(\text{hash}([\text{key}]), m) \]
Amazon Dynamo: Object Versioning

- Object Versioning (per bucket)
  - `DELETE` doesn’t wipe
  - `GET` will return not found

- PERMANENT DELETE by version... wiped

Amazon Dynamo: Model

- Named table with primary key and a value
  - Primary key is hashed / unordered

<table>
<thead>
<tr>
<th>Countries</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albania</td>
<td>capital: Tirana, continent: Europe, pop: 3,011,405,2013</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cities</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kabul</td>
<td>country: Afghanistan, pop: 4,741,000,2013</td>
</tr>
<tr>
<td>Tirana</td>
<td>country: Albania, pop: 3,011,405,2013</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Amazon Dynamo: Dual primary key also available:

- Hash: unordered
- Range: ordered

<table>
<thead>
<tr>
<th>Countries</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vatican City</td>
<td>capital: Vatican City, continent: Europe</td>
</tr>
<tr>
<td>Nauru</td>
<td>capital: Nauru, continent: Oceania</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Amazon Dynamo: CAP

Two options for each table:

- **AP**: Eventual consistency, High availability
- **CP**: Strong consistency, Lower availability

What’s an AP system again?

What’s a CP system again?
Amazon Dynamo: Consistency

- **Gossiping**
  - Keep alive messages sent between nodes with state

- **Quorums**:
  - $N$ nodes responsible for a read/write
  - Multiple nodes acknowledge read/write for success
  - At the cost of availability!

- **Hinted Handoff**
  - For transient failures
  - A node “covers” for another node while its down

Amazon Dynamo: Vector Clocks

- **Vector Clock**: A list of pairs indicating a node (i.e., a server) and a time stamp
- **Used to track/order versions**

Amazon Dynamo: Eventual Consistency using Merkle Trees

- **Merkle tree** is a hash tree
- **Nodes have hashes of their children**
- **Leaf node hashes from data: keys or ranges**

Amazon Dynamo: Eventual Consistency using Merkle Trees

- Easy to verify regions of the Map
- Can compare level-at-a-time

Amazon Dynamo: Budgeting

- Assign throughput per table: allocate resources
- **Reads (4 KB resolution):**
  - Expected Item Size
  - Desired Reads Per Second
  - Provisioned Throughput Required
  - 1 KB: 50
  - 8 KB: 50

- **Writes (1 KB resolution):**
  - Expected Item Size
  - Desired Writes Per Second
  - Provisioned Throughput Required
  - 1 KB: 10
  - 2 KB: 50
Dynamo: Amazon’s Highly Available Key-value Store

Oluwaseun DeClerck, Daniel Hoste, Marian Jarasni, Gunmerthu Kalaikapili, Anuradha Lakshman, Alex Mikan, Saurabhak Sundaraman, Peter Vossell, and Werner Vogels
Amazon.com

ABSTRACT

Decentralization of server state is one of the biggest challenges facing cloud computing. As web services and applications grow in complexity, and more of their logic is pushed down to the client, the centralization of server state becomes more and more fragile. Amazon Dynamo is a highly available key-value store built on top of a distributed hash table. We describe the server state of the system, and how it is distributed across a large datacenter to achieve high availability and low latency.

Other Key–Value Stores

riak

COMCAST
Symantec
at&t
Boeing
Best Buy
Ask
Aol.

redis

StackExchange
digg
Twitter
Pinterest
github

NOSQL: DOCUMENT STORE

Other Key–Value Stores

 cassandra

Adobe
Instagram
accenture
answers.com
Dell
Disney
google
NASA
The Database Landscape

Key–Value Stores: Values are Documents

Document Stores

Recap

RECAP
Recap

• Key-value stores inspired by Amazon Dynamo
  – Distributed maps
  – Hash keys and range keys
  – Table names
  – Consistent hashing
  – Replication
  – Object versioning / vector clocks
  – Gossiping / Quorums / Hinted Hand-offs
  – Merkle trees
  – Budgeting

• Document stores: documents as values
  – Support for JSON, XML values, field indexing, etc.

Questions

?