Πανεπιστήμιο Πατρών

Προχωρημένα Θέματα σε Κατανεμημένα Συστήματα

Cassandra
Cassandra

A Decentralized Structured Storage System
Overview

- Row store
- Decentralized
- Data Model: similar to relational
  - ...but not entirely!
- High resilience to failures
- High scalability, w.r.t. the amount of data
Properties

- Symmetric
  - No single point of failure
  - Linearly scalable
  - Ease of administration

- Flexible partitioning, replica placement

- Automated provisioning

- High availability (eventual consistency)
Influenced by

- **BigTable (Google)**
  - Strong consistency
  - Sparse map data model
  - GFS [Chubby et al.]

- **Dynamo (Amazon)**
  - $O(1)$ distributed hash table (DHT)
  - BASE (aka eventual consistency)
  - Client tunable consistency/availability
Dynamo

- Consistent Hashing
  - Routing, Load balancing, Replica placement

- Vector Clocks
  - Concurrent updates

- Gossip Protocol

- Hinted Handoffs
  - Failure detection/recovery
Data Model

- Data
  - Rows indexed by row keys
  - Each row has any number of columns (key/value pairs)
  - Set of columns of a row: column family → Dynamic, Sparse

- Sorting
  - Columns are sorted by key
  - Rows can be sorted by key, but usually are not!

- Essentially, this schema is equivalent to:
  - Map< RowKey, SortedMap<ColumnKey, ColumnValue> >
Data Model

- A column value may be composite, containing a set of columns (key/value pairs)
  - These are called **Super Columns**

- Essentially, this schema is equivalent to:
  - `Map< RowKey, SortedMap<SuperColumnKey, SortedMap<ColumnKey, ColumnValue>>>`
# Analogy to the Relational Model (E-R)

<table>
<thead>
<tr>
<th>Relational Model</th>
<th>Cassandra Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Keyspace</td>
</tr>
<tr>
<td>Table</td>
<td>Column Family (CF)</td>
</tr>
<tr>
<td>Primary key</td>
<td>Row key</td>
</tr>
<tr>
<td>Column name</td>
<td>Column name/key</td>
</tr>
<tr>
<td>Column value</td>
<td>Column value</td>
</tr>
</tbody>
</table>
Basic API

- Simple API, very similar to DHTs:
  - insert (keyspace, key, rowMutation)
  - get (keyspace, key, columnName)
  - delete (keyspace, key, columnName)

- Requests may be sent to *any* node

- It collects the results and sends them to the client
Additional API

Cassandra has introduced the additional following API calls

- Range queries – for both keys and columns
  - list<KeySlice> get_range_slices(column_parent, predicate, range, consistency_level)

- Multiget – collect data from multiple rows, not necessarily contiguous

- Indexing and select
  - list<KeySlice> get_indexed_slices(column_parent, index_clause, predicate, consistency_level)
Data Placement
Consistent Hashing

- Data is partitioned across nodes, based on **consistent hashing**
  - Rings a bell?! 😊

- Hash function applied on each row’s **partition key**
  - Typically, that’s the primary key
  - But can be defined otherwise

- Each row is, then, mapped to its (hashed) partition key’s successor node

---

```
Ring (key space)

2^320 0

A
B
C
D
E
F

canonical home (coordinator node) for key range A-B

Same hash function for data and nodes
idx = hash(key)

Coordinator: next available clockwise node
```
Partitioning

- Two hash function types
  - Random Partitioner
  - Order-Preserving Partitioner

- **RandomPartitioner** *(hash value used as id)*
  - Easy load balancing
  - Easy addition / removal of nodes

- **OrderPreservingPartitioner** *(string used as id)*
  - Range queries
  - Difficult load balancing
  - Difficult addition / removal of nodes
Failures: Node B crashes

Same hash function for data and nodes

\[ \text{idx} = \text{hash(key)} \]

Coordinator: next available clockwise node
Replication

- **N replicas per row**

![Diagram showing replication in a ring system](image)

- Key \( A_B \) hosted in B, C, D
- Data replicated in the N-1 clockwise successor nodes
- Node hosting Key \( F_A \), Key \( A_B \), Key \( B_C \)
Node B crashes with replication factor 3

Key membership and replicas are updated when a node joins or leaves the network. The number of replicas for all data is kept consistent.
Message Routing

- Distributed Hash Table (Peer to Peer)

Routing tables

Getting to B in 3 hops
Load Balancing? Virtual Nodes [1/2]

Different Strategies

Virtual Nodes

Random tokens per each physical node, partition by token value

- Node 1: tokens A, E, G
- Node 2: tokens C, F, H
- Node 3: tokens B, D, I
Load Balancing? Virtual Nodes [2/2]

Different Strategies

Virtual Nodes

$Q$ equal-sized partitions, $S$ nodes, $Q/S$ tokens per node (with $Q >> S$)

- Node 1
- Node 2
- Node 3
- Node 4
...
Replica Placement Strategies

- Rack unaware
  - Simply choose N-1 successors

- Rack/DC aware:
  - Place one replica on different Data Center
  - Place another replica on different Rack of the same Data Center

- Totally configurable
  - Any replica placement strategy can be defined.
# Consistency level

<table>
<thead>
<tr>
<th>WRITE</th>
<th>READ</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>ZERO</td>
<td>Cross fingers</td>
</tr>
<tr>
<td>ANY</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; Response (including HH)</td>
</tr>
<tr>
<td>ONE</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; Response</td>
</tr>
<tr>
<td>QUORUM</td>
<td>N/2 + 1 replicas</td>
</tr>
<tr>
<td>ALL</td>
<td>All replicas</td>
</tr>
</tbody>
</table>
Creating a Schema
Relational Schema

**User**

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jay</td>
<td><a href="mailto:jp@ebay.com">jp@ebay.com</a></td>
</tr>
<tr>
<td>456</td>
<td>John</td>
<td><a href="mailto:jh@ebay.com">jh@ebay.com</a></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**User_Item_Like**

<table>
<thead>
<tr>
<th>ID</th>
<th>UserID &lt;fk&gt;</th>
<th>ItemID &lt;fk&gt;</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123</td>
<td>111</td>
<td>120101010000</td>
</tr>
<tr>
<td>2</td>
<td>123</td>
<td>222</td>
<td>120101020000</td>
</tr>
<tr>
<td>3</td>
<td>456</td>
<td>111</td>
<td>120101030000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Item**

<table>
<thead>
<tr>
<th>ItemID</th>
<th>Title</th>
<th>Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>iphone</td>
<td>It’s a phone</td>
</tr>
<tr>
<td>222</td>
<td>ipad</td>
<td>It’s a tablet</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Σπύρος Βούλγαρης – Πανεπιστήμιο Πατρών
### Exact replica of relational model

<table>
<thead>
<tr>
<th>User</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>111</td>
</tr>
<tr>
<td>Name</td>
<td>Title</td>
</tr>
<tr>
<td>Jay</td>
<td>iphone</td>
</tr>
<tr>
<td>Email</td>
<td>Desc</td>
</tr>
<tr>
<td><a href="mailto:jp@ebay.com">jp@ebay.com</a></td>
<td>It's a phone</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>User_Item_Like</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>123</td>
</tr>
</tbody>
</table>

...
Normalized entities w/custom indexes

<table>
<thead>
<tr>
<th>User</th>
<th>123</th>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Jay</td>
<td><a href="mailto:jp@ebay.com">jp@ebay.com</a></td>
</tr>
<tr>
<td>User_By_Item</td>
<td>111</td>
<td>123</td>
<td>456</td>
</tr>
<tr>
<td></td>
<td></td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>111</th>
<th>Title</th>
<th>Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>iphone</td>
<td>It’s a phone</td>
</tr>
<tr>
<td>Item_By_User</td>
<td>123</td>
<td>111</td>
<td>222</td>
</tr>
<tr>
<td></td>
<td></td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
More denormalization

- Normalized entities with de-normalization into custom indexes

![Diagram](image.png)

- User
  - 123
  - Name: Jay
  - Email: jp@ebay.com

- Item
  - 111
  - Title: iphone
  - Desc: It’s a phone

- User_By_Item
  - 111
  - 123
  - 456
  - Jay
  - John

- Item_By_User
  - 123
  - 111
  - 222
  - iphone
  - ipad
**Partially de-normalized entities**

### User

<table>
<thead>
<tr>
<th>UserID</th>
<th>UserInfo</th>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Name: Jay, Email: <a href="mailto:jp@ebay.com">jp@ebay.com</a></td>
<td>111, Likes: iPhone, iPad</td>
</tr>
</tbody>
</table>

### Item

<table>
<thead>
<tr>
<th>ItemID</th>
<th>ItemInfo</th>
<th>LikedBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Title: iPhone, Desc: It’s a phone</td>
<td>123, 4556</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jay, John</td>
</tr>
</tbody>
</table>
Easy “get N most recent” queries

**User**

<table>
<thead>
<tr>
<th>123</th>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Jay</td>
<td><a href="mailto:jp@ebay.com">jp@ebay.com</a></td>
</tr>
</tbody>
</table>

**Item**

<table>
<thead>
<tr>
<th>111</th>
<th>Title</th>
<th>Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iphone</td>
<td>It’s a phone</td>
</tr>
</tbody>
</table>

**User_By_Item**

| 111 | 120101010000 | 123 | 120101030000 | 456 | ... |

|     | Jay       | John   |

**Item_By_User**

| 123 | 120101010000 | 111 | 120101020000 | 222 | ... |

|     | iphone    | ipad   |
Extra material

  - http://en.wikipedia.org/wiki/Paxos_%28computer_science%29


- Distributed Hash Tables και Consistent hashing