 PART B. GEAR SESSIONS
SESSION 1: PETA-Scale STORAGE SYSTEMS

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

Topics of Today's Class
• GEAR Session I. Peta Scale Storage Systems
• Lecture 3.
  • Cassandra

GEAR Session 1. Peta-scale Storage Systems

This material is built based on,
• Datastax Documentation: Apache Cassandra
  • http://docs.datastax.com/en/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html
• Now: Apache’s open source project,
  http://cassandra.apache.org

http://www.cs.colostate.edu/~cs535
CAP Theorem

- Eric Brewer
- It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees
  - **Consistency**: Every read receives the most recent write or an error
  - **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

Facebook’s operational requirements

- Performance
- Reliability
- Failures are norm
- Efficiency
- Scalability
- Support continuous growth of the platform

Inbox search problem

- A feature that allows users to search through all of their messages
- By name of the person who sent it
- By a keyword that shows up in the text
- Search through all the previous messages
- In order to solve this problem,
  - Billions of writes per day
  - Large number of users

Now,

- Cassandra is in use at,
  - Apple
  - CERN
  - Easou
  - Comcast
  - eBay
  - GitHub
  - Hulu
  - Instagram
  - Netflix
  - Reddit
  - The Weather Channel
  - And over 1500 more companies

Data Model (1/2)

- Distributed multidimensional map indexed by a key
  - **Row key**
    - String with no size restrictions
    - Typically 16 – 36 bytes long
    - Every operation under a single row key is atomic
  - **Value is an object**
    - Highly structured

GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system
**Apache Cassandra**

Data Model

![cassandra](http://www.cs.colostate.edu/~cs535)
Data Model (2/2)

- Columns are grouped into column families
  - Similar to BigTable

COLUMN family is an ordered collection of rows

Comparison between RDMBS and Cassandra

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDBMS deals with structured data</td>
<td>Cassandra deals with unstructured data</td>
</tr>
<tr>
<td>It has a fixed schema</td>
<td>Cassandra has a flexible schema</td>
</tr>
<tr>
<td>In RDBMS, a table is an array of arrays, (ROW x COLUMN)</td>
<td>In Cassandra, a table is a list of &quot;nested key-value pairs&quot;, (ROW x COLUMN key x COLUMN value)</td>
</tr>
<tr>
<td>Databases in the outermost container that contains data corresponding to an application</td>
<td>Column family is the outermost container that contains data corresponding to an application</td>
</tr>
<tr>
<td>Tables are the entities of a database</td>
<td>Columns or column families are the entities of a super column family</td>
</tr>
<tr>
<td>Row is an individual record in RDBMS</td>
<td>Row is a unit of storage in Cassandra</td>
</tr>
<tr>
<td>Column represents the attributes of a relation</td>
<td>Column is a unit of storage in Cassandra</td>
</tr>
<tr>
<td>RDBMS supports the concept of foreign keys, joins</td>
<td>Relationships are represented using collections</td>
</tr>
</tbody>
</table>

Column family vs. a table of relational databases

<table>
<thead>
<tr>
<th>Relation Table</th>
<th>Cassandra column family</th>
</tr>
</thead>
<tbody>
<tr>
<td>A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value.</td>
<td>In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time.</td>
</tr>
<tr>
<td>Relational tables define only columns and the user fills in the table with values.</td>
<td>In Cassandra, a table contains columns, or can be defined as a super column family</td>
</tr>
</tbody>
</table>

Column: basic data structure of Cassandra with three values, namely key or column name, value, and a timestamp (e.g. name:byte[], value:byte[], clock:clock[])

Super Column: it is also a key-value pair (e.g. name:byte[], cols:map<byte[],column>)

Here, we have a data model. What do we have to consider?

- We will use the "key" to retrieve data
- Spread data evenly (as even as possible) around the cluster
- Rows are spread around the cluster based on a hash of the partition key, which is the first element of the PRIMARY KEY
- Cluster should be incrementally scalable
- Scale-out solution

API

- insert(table, key, rowMutation)
- get(table, key, columnName)
- delete(table, key, columnName)

http://www.cs.colostate.edu/~cs535
GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system
Apache Cassandra
Data Partitioning: Consistent Hashing

Consistent hashing

- Consistent hash function assigns each node and key on a circular identifier using a hashing function
- Hashing value of IP address
- m-bit identifiers
- m has to be big enough to make the probability of two nodes or keys hashing to the same identifier negligible

Consistent hashing assigns keys to nodes:
- Key k will be assigned to the first node whose identifier is equal to or follows k in the identifier space

If machine C leaves circle,
Successor(5) will point to A
If machine N joins circle,
successor(2) will point to N

Scalable Key location
- Each node need only be aware of its successor node on the circle
- Queries can be passed around the circle via these successor pointers until it finds the resource

What is the disadvantage of this scheme?
Scalable Key location

- In consistent hashing:
  - Each node need only be aware of its successor node on the circle
  - Queries can be passed around the circle via these successor pointers until it finds the resource

- What is the disadvantage of this scheme?
  - It may require traversing all N nodes to find the appropriate mapping

This material is built based on


Definition of variables for node n, using m-bit identifiers

- Let $m$ be the number of bits in the key/node identifiers
- Each node $n$ maintains:
  - A routing table with (at most) $m$ entries
  - Called the finger table

- The $i^{th}$ entry in the table at node $n$ contains the identity of the first node, $s$, that succeeds $n$ by at least $2^i-1$ on the identifier circle
  - i.e. $s = \text{successor} (n+2^i-1)$, where $\text{successor}$ and all arithmetic is modulo $2^m$

Example of use

- Apache Cassandra’s partitioning scheme
- Couchbase
- Openstack’s object storage service Swift
- Akamai Content delivery network
- Data partitioning in Voldemort
- Partitioning component of Amazon’s storage system Dynamo (zero-hop DHT)

http://www.cs.colostate.edu/~cs535
- Finger table
  - The Chord identifier
  - The IP address of the relevant node

- First finger of \( n \) is its immediate successor on the circle
  - Clockwise!

Finger tables

<table>
<thead>
<tr>
<th>Finger table</th>
<th>Fingers to node</th>
<th>Fingers to node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 2)</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>(4, 1)</td>
<td>(3, 5)</td>
</tr>
<tr>
<td>3</td>
<td>(5, 1)</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(4, 5)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>(7, 3)</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>(4, 8)</td>
<td>0</td>
</tr>
</tbody>
</table>

Lookup process

1. Request comes into node 3 to find the successor of id 1.
2. Node 3 asks node 0 to find successor of 1.
3. Node 0 finds successor of 1.
4. Node 3 asks node 3 to find successor of 4.
5. Node 3 asks node 3 to find successor of 4.

1/3

1. Go clockwise
2. Never overshoot.

- First, check the data is stored in \( n \)
  - If it is, return the data
- Otherwise,
  - \( n \) searches its finger table for the node \( j \)
  - Whose ID most immediately precedes \( k \)

- Ask \( j \) for the node it knows whose ID is closest to \( k \)
  - Do not overshoot!

1/3

1. Go clockwise
2. Never overshoot.

1. First, check the data is stored in \( n \)
  - If it is, return the data
- Otherwise,
  - \( n \) searches its finger table for the node \( j \)
  - Whose ID most immediately precedes \( i \)

- Ask \( j \) for the node it knows whose ID is closest to \( i \)
  - Do not overshoot!

1/3

1. Go clockwise
2. Never overshoot.

1. First, check the data is stored in \( n \)
  - If it is, return the data
- Otherwise,
  - \( n \) searches its finger table for the node \( j \)
  - Whose ID most immediately precedes \( i \)

- Ask \( j \) for the node it knows whose ID is closest to \( i \)
  - Do not overshoot!

1/3

1. Go clockwise
2. Never overshoot.

1. First, check the data is stored in \( n \)
  - If it is, return the data
- Otherwise,
  - \( n \) searches its finger table for the node \( j \)
  - Whose ID most immediately precedes \( i \)

- Ask \( j \) for the node it knows whose ID is closest to \( i \)
  - Do not overshoot!

1/3

1. Go clockwise
2. Never overshoot.

1. First, check the data is stored in \( n \)
  - If it is, return the data
- Otherwise,
  - \( n \) searches its finger table for the node \( j \)
  - Whose ID most immediately precedes \( i \)

- Ask \( j \) for the node it knows whose ID is closest to \( i \)
  - Do not overshoot!
Theorem 2.
- With high probability (or under standard hardness assumptions), the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$.

Proof
Suppose that node $n$ tries to resolve a query for the successor of $k$. Let $p$ be the node that immediately precedes $k$. We analyze the number of steps to reach $p$.

If $n \neq p$, then $n$ forwards its query to the closest predecessor of $k$ in its finger table. ($i$ steps) Node $k$ will finger some node $f$ in this interval. The distance between $n$ and $f$ is at least $2^{i-1}$.

Proof continued
If $f$ and $p$ are both in $n$'s $p$th finger interval, and the distance between them is at most $2^{i-1}$. This means $f$ is closer to $p$ than to $n$ or equivalently:
- Distance from $f$ to $p$ is at most half of the distance from $n$ to $p$.
- If the distance between the node handling the query and the predecessor $p$ halves in each step, and is at most $2^i$.

Within $m$ steps the distance will be $1$ (you have arrived at $p$).

The number of forwardings necessary will be $O(\log N)$.

After $\log N$ forwardings, the distance between the current query node and the key $k$ will be reduced at most $2^{-\log N}$.

The average lookup time is $\frac{1}{2}\log N$.

Requirements in node Joins
- In a dynamic network, nodes can join (and leave) at any time.
  1. Each node's successor is correctly maintained.
  2. For every key $k$, node $\text{successor}(k)$ is responsible for $k$.

Tasks to perform node join
1. Initialize the predecessor and fingers of node $n$.
2. Update the fingers and predecessors of existing nodes to reflect the addition of $n$.
3. Notify the higher layer software so that it can transfer state (e.g., values) associated with keys that node $n$ is now responsible for.

Step 1: Initializing fingers and predecessor (1/2)
- New node $n$ learns its predecessor and fingers by asking any arbitrary node in the network $n'$ to look them up:
  ```java
  n.init_finger_table(n');
  finger[i].node = n'.find_successor(finger[i].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for (i = 1 to m)
    if (finger[i].start is in (x, n.finger[i].node))
      finger[i].node = finger[i].node;
    else
      finger[i].node = n'.find_successor(finger[i].start);
  ```

- Create the finger-table at the new node $n$ by asking the node $n'$. 

---

http://www.cs.colostate.edu/~cs535
Step 1: Initializing fingers and predecessor (2/2)

- Naïve run for find_successor will take $O(\log N)$
- For $m$ finger entries
  - $O(m \log N)$

  How can we optimize this?
  - Check if $i^{th}$ node is also correct for the $(i+1)^{th}$ node (see the code in the step 1-(1/2))
  - Ask immediate neighbor and copy of its complete finger table and its predecessor
  - New node $n$ can use these table as hints to help it find the correct values
    - It shares some nodes

Updating fingers of existing nodes

- Node $n$ will be entered into the finger tables of some existing nodes

  $n$.update_others()
  
  for $i = 1$ to $m$
    $p = \text{find_predecessor}(n - 2i - 1)$;
    $p$.update_finger_table($n, i$);
    $p$.update_finger_table($s, i$) if ($s$ is in $[n, \text{finger}[i].node]$)

  $\text{finger}[i].node = s$;

  $p = \text{predecessor}();$

  $n$.update_finger_table($s, i$);

  $p$.update_finger_table($s, i$);

  Node $n$ will become the $i^{th}$ finger of node $p$ if and only if,
  - $p$ precedes $n$ by at least $2i - 1$
  - the $i^{th}$ finger of node $p$ succeeds $n$

  The first node $p$ that can meet these two condition
  - Immediate predecessor of $n - 2i$
  - $n$

  For the given $n$, the algorithm starts with the finger of node $n$
  - Continues to walk in the counter-clock-wise direction on the identifier circle
  - Number of nodes that need to be updated is $O(\log N)$ on the average

Transferring keys

- Move responsibility for all the keys for which node $n$ is now the successor
  - It involves moving the data associated with each key to the new node

- Node $n$ can become the successor only for keys that were previously the responsibility of the node immediately following $n$
  - $n$ only needs to contact that one node to transfer responsibility for all relevant keys

Join 5 (After $\text{init\_finger\_table}(n')$)

Join 5 (After $\text{update\_others}()$)

http://www.cs.colostate.edu/~cs535

Spring 2020 Colorado State University, page 8
Example

If you have following data,

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Car</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>36</td>
<td>Camaro</td>
<td>M</td>
</tr>
<tr>
<td>Carol</td>
<td>37</td>
<td>BMW</td>
<td>F</td>
</tr>
<tr>
<td>Jenny</td>
<td>10</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>Suzy</td>
<td>9</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

Cassandra assigns a hash value to each partition key

<table>
<thead>
<tr>
<th>Partition Key</th>
<th>Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>2245462676723223822</td>
</tr>
<tr>
<td>Carol</td>
<td>7723358927203680754</td>
</tr>
<tr>
<td>Jenny</td>
<td>6723372854036780875</td>
</tr>
<tr>
<td>Suzy</td>
<td>1168604627387940318</td>
</tr>
</tbody>
</table>

Cassandra cluster with 4 nodes

GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system
Apache Cassandra
Data Partitioning: Partitioners

Partitioning
- Partitioner is a function for deriving a token representing a row from its partition key, typically by hashing
- Each row of data is then distributed across the cluster by value of the token
- Read and write requests to the cluster are also evenly distributed
- Each part of the hash range receives an equal number of rows on average

Cassandra offers three partitioners:
- **Murmur3Partitioner** (default): uniformly distributes data across the cluster based on MurmurHash hash values.
- **RandomPartitioner**: uniformly distributes data across the cluster based on MD5 hash values.
- **ByteOrderedPartitioner**: keeps an ordered distribution of data lexically by key bytes

1. **Murmur3Partitioner**
- Murmur hash is a non-cryptographic hash function
  - Created by Austin Appleby in 2008
  - Multiply (M0) and Rotate (R)
- Current version Murmur3 yields 32 or 128-bit hash value
- Murmur3 has low bias of under 0.5% with the Avalanche analysis

Testing with 42 Million keys

http://www.cs.colostate.edu/~cs535
Measuring the quality of hash function

• Hash function quality

\[ \sum_{i} \frac{b(i) + 1}{2} \]  
  Where, \( b_i \) is the number of items in \( j \)-th slot.  
  \( n \) is the total number of items  
  \( m \) is the number of slots


Comparison between hash functions

Avalanche Analysis for hash functions

• Indicates how well the hash function mixes the bits of the key to produce the bits of the hash  
  - Whether a small change in input causes a significant change in the output  
  - Whether or not it achieves “avalanche”  
  - \( P(\text{Output bit } i \text{ changes | Input bit } j \text{ changes}) = 0.5 \)  
    for all \( i, j \)  
  - If we keep all of the input bits the same, and flip exactly 1 bit  
    - Each of our hash function’s output bits changes with probability \( \frac{1}{2} \)  
  - The hash is “biased”  
    - If the probability of an input bit affecting an output bit is greater than or less than 50%  
    - Large amounts of bias indicate that keys differing only in the biased bits may tend to produce more hash collisions than expected.

2. RandomPartitioner

• RandomPartitioner was the default partitioner prior to Cassandra 2.1  
• Uses MDS  
• \( 0 \leq b < 2^{31} - 1 \)

3. ByteOrderPartitioner

• This partitioner orders lexically by key bytes  
• The ordered partitioner allows ordered scans by primary key  
  - If your application has user names as the partition key, you can scan rows for users whose names fall between Jake and Joe
  - Disadvantage of this partitioner  
    - Difficult load balancing  
    - Sequential writes can cause hot spots  
    - Uneven load balancing for multiple tables

http://www.cs.colostate.edu/~cs535
### Replication

- Provides high availability and durability
- For a replication factor (replication degree) of N
  - The coordinator replicates these keys at N-1 nodes
  - Client can specify the replication scheme
    - Rack-aware/Rack-unaware/Datacenter-aware
- There is no master or primary replica
- Two replication strategies are available
  - SimpleStrategy
    - Used only for a single data center
    - Places the first replica on a node determined by the partitioner
    - Places additional replicas on the next nodes clockwise in the ring without considering topology
    - Does not consider rack or data center location
- NetworkTopologyStrategy
  - Multi-data center setup

#### 1. SimpleStrategy

- Used only for a single data center
- Places the first replica on a node determined by the partitioner
- Places additional replicas on the next nodes clockwise in the ring without considering topology
- Does not consider rack or data center location

#### 2. NetworkTopologyStrategy

- For the data cluster deployed across multiple data centers
  - This strategy specifies how many replicas you want in each data center
- Places replicas in the same data center by walking the ring clockwise until it reaches the first node in another rack
  - Attempts to place replicas on distinct racks
  - Nodes in the same rack (or similar physical grouping) often fail at the same time due to power, cooling, or network issues.

#### 2. NetworkTopologyStrategy (1/3)

- When deciding how many replicas to configure in each data center, you should consider:
  - being able to satisfy reads locally, without incurring cross data-center latency
  - failure scenarios

#### 2. NetworkTopologyStrategy (2/3)

- The two most common ways to configure multiple data center clusters
  - Two replicas in each data center
    - This configuration tolerates the failure of a single node per replication group and still allows local reads at a consistency level of ONE
  - Three replicas in each data center
    - This configuration tolerates either the failure of one node per replication group at a strong consistency level of LOCAL_QUORUM or multiple node failures per data center using consistency level ONE

#### 3. NetworkTopologyStrategy

- Asymmetrical replication groupings
  - For example, you can maintain 4 replicas
    - Three replicas in one data center to serve real-time application requests
    - A single replica elsewhere for running analytics.

### Questions?