FAQs
- Quiz 7: 5/1
- Workshop IV

Today’s topics
- Cassandra
  - Partitioner, replication, and virtual nodes

This material is built based on

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

\begin{tabular}{|c|c|c|c|c|}
\hline
Start & int & succ & 1 & 2 \\
\hline
1 & 2 & 1 & 1 & 2 \\
\hline
2 & 2 & 3 & 3 & 4 \\
\hline
3 & 3 & 5 & 5 & 6 \\
\hline
4 & 4 & 0 & 0 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|}
\hline
Start & int & succ & 2 & 3 \\
\hline
2 & 3 & 3 & 3 & 4 \\
\hline
3 & 3 & 5 & 5 & 6 \\
\hline
4 & 4 & 0 & 0 & 1 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|}
\hline
Start & int & succ & 4 & 5 \\
\hline
4 & 5 & 0 & 0 & 1 \\
\hline
5 & 5 & 0 & 0 & 1 \\
\hline
7 & 7 & 0 & 0 & 1 \\
\hline
\end{tabular}

\begin{itemize}
\item 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 \(O(\log N)\).
\end{itemize}

\textbf{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 \(2^{i-1}\).

\textbf{Requirements in node Joins}

- In a dynamic network, nodes can join (and leave) at any time
- Each node's successor is correctly maintained
- For every key \(k\), node \(\text{successor}(k)\) is responsible for \(k\) (query \(k\) to the system)

\textbf{Tasks to perform node join}

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

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

```plaintext
n.init_finger_table(n');
finger[1].node = n'; find_successor(finger[1].start);
predecessor = successor.predecessor;
successor.predecessor = n;
for (i = 1; i < m;
    if (finger[i].start is in [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 $n'$

Join 5 (After init_finger_table($n'$))

Step 1: Initializing fingers and predecessor (2/2)

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

- How can we optimize this?
  - Check if $n$ node is also correct for the $(n+1)$ 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

Join 5 (After update_others())

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

```plaintext
n.update_others();
for (i = 1; i <= m;
    p = find_successor(n.2-i);
    p.update_finger_table(n,i);
    p.update_others();
```

Updating fingers of existing nodes

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

```plaintext
n.update_others();
for (i = 1; i <= m;
    p = find_successor(n.2-i);
    p = get_first_node_preceding_x
    p.update_finger_table(n,i);
```

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

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>Jonny</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>Murmur 3 Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>224546287673233822</td>
</tr>
<tr>
<td>Carol</td>
<td>772335892705685754</td>
</tr>
<tr>
<td>Jonny</td>
<td>872337455043788575</td>
</tr>
<tr>
<td>Suzy</td>
<td>116650462738790318</td>
</tr>
</tbody>
</table>

Cassandra cluster with 4 nodes

GEAR Workshop IV | Scalable Data Storage, Retrieval, and Analytics
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 (MU) and Rotate (R)

- Current version Murmur 3 yields 32 or 128-bit hash value
- Murmur3 has low bias of under 0.5% with the Avalanche analysis

http://www.cs.colostate.edu/~cs535
Testing with 42 Million keys

Measuring the quality of hash function

\[ \text{Hash function quality} = \frac{h(k)+1/2}{\sqrt{2m(n+2m-1)}} \]

where,
- \( h(k) \) is the number of items in the \( k \)-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”
  - \( P( \text{output bit } i \text{ changes } | \text{input bit } j \text{ changes}) = 0.5 \)
  - 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 \( 1/2 \)
  - 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 MD5
  - \( 0 \) or \( 2^n - 1 \)

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

Spring 2019 Colorado State University, page 5
3. **ByteOrderPartitioner**

- This partitioner orders rows **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

---

**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
    - Use for a single data center only
    - **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.

---

- 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 (3/3)

- 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.

What are Vnodes?

- With consistent hashing, a node owns exactly one contiguous range in the ring-space

- Vnodes change from one token or range per node to many per node
  - Within a cluster these can be randomly selected and be non-contiguous, giving us many smaller ranges that belong to each node