PART 2. SCALABLE FRAMEWORKS FOR REAL-TIME BIG DATA ANALYTICS
2. SERVING LAYER: CASSANDRA

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

FAQs
- Assignment 2 has been posted
  - URL for zookeeper has been updated
  - Use the same machines and ports assignment

Today’s topics
- Document-oriented storage system - continued
  - Apache Cassandra

Apache Cassandra
Partitioning
Chord-based DHT

This material is built based on

Scalable Key location in Chord
- 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, \( s \),
  - Succeeds \( n \) by at least \( 2^i \) on the identifier circle
  - I.e., \( s = \text{successor}(n \times 2^i) \), where \( \text{successor} \) (and all arithmetic is modulo \( 2^m \))

The \( i \)th entry finger of node \( n \)
Definition of variables for node $n$, using $m$-bit identifiers

- $\text{finger}[i].\ start = (n+2^{i-1}) \mod 2^m$, $1 \leq k \leq m$
- $\text{finger}[i].\ interval = [\text{finger}[i].\ start, \text{finger}[i+1].\ start)$
- $\text{finger}[i].\ node = \text{first node} \geq n.\text{finger}[i].\ start$
- $\text{successor} = \text{the next node on the identifier circle}$
- $\text{predecessor} = \text{the previous node on the identifier circle}$

Finger tables

<table>
<thead>
<tr>
<th>Start</th>
<th>Int</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1,2)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>[2,4)</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>[4,0)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>[2,3)</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>[3,5)</td>
<td>3</td>
</tr>
<tr>
<td>5</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>[5,7)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>[7,3)</td>
<td>0</td>
</tr>
</tbody>
</table>

Lookup process

1. Request comes into node 3 to find the successor of identifier 1.
2. Identifier 1 belongs to [7,3)
3. Check succ: 0
4. Node 3 asks node 0 to find successor of 1
5. Successor of 1 is 1
6. Node 3 wants to find the successor of identifier 1

- Each node stores information about only a small number of other nodes
- A node’s finger table generally does not contain enough information to determine the successor of an arbitrary key $k$
- What happens when a node $n$ does not know the successor of a key $k$
  - If $n$ finds a node whose ID is closer than its own to $k$, that node will know more about the identifier circle in the region of $k$ than $n$ does
- Do not overshoot!
### Lookup process: example 1

1. Node 1 asks node 3 to find successor of 4
2. Successor of 4 is 0
3. Request comes into node (machine) 1 to find the successor of identifier 4
4. Node 4 belongs to [4, 5)
5. Check succ: 3

### Lookup process: example 2

1. Node 3 wants to find the successor of identifier 0
2. Identifier 0 belongs to [7, 3)
3. Check succ: 0
4. Node 3 asks node 0 to find successor of 1
5. Machine is using identifier 0 as well → succ is 0

### 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 $k$. 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

$f$ and $p$ are both in $n$’s $i$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 $2^m$.

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 $2^{m}/N$.

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
#define successor finger[1].node

// node n joins the network
// n' is an arbitrary node in the network
n.join(n')
if (n')
    init_finger_table(n');
    update_others();
else // if n is going to be the only node in the network
    for i = 1 to m
        finger[i].node = n;
        predecessor = n;

n.find_successor(id)
    n' = find_predecessor(id);
    return n'.successor;

n.find_predecessor(id)
    n' = n;
    while(id is NOT in (n', n'.successor))
        n' = n.closest_preceding_finger(id);
    return n';

n.closest_preceding_finger(id)
    for i = m down to 1
        if(finger[i].node is in (n, id))
            return finger[i].node;
    return n;

n.init_predecessor()
yIELD "init Inherits Predecessor"
Step 2: Updating fingers of existing nodes [1/2]

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

```plaintext
n.update_others()
for i=1 to m
  p = find_predecessor(n-2i-1);
  p.update_finger_table(n,i);
  if (s is in [n, finger[i].node])
    finger[i].node = s;
  p = predecessor;//get first node preceding n
  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 conditions
- Immediate predecessor of n-2i+1
- Continues to walk in the counter-clock-wise direction on the identifier circle

Number of nodes that need to be updated is O(logN)

Step 3: 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>M</td>
<td></td>
</tr>
<tr>
<td>Suzy</td>
<td>9</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Cassandra assigns a hash value to each partition key

<table>
<thead>
<tr>
<th>Partition Key</th>
<th>Mumber 3 Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>7233728540368700754</td>
</tr>
<tr>
<td>Carol</td>
<td>8234562067293239022</td>
</tr>
<tr>
<td>Jonny</td>
<td>9233728540368700754</td>
</tr>
<tr>
<td>Suzy</td>
<td>116890462738794318</td>
</tr>
</tbody>
</table>

Cluster with 4 nodes

Apache Cassandra 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

### 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

### Testing with 42 Million keys

Measuring the quality of hash function

\[ \text{Function of Keys Inserted} \]

- Hash function quality
  \[ b_j(b_j + 1)/2 \]
  \[ (n/2m)(n + 2m - 1) \]
  - Where, \( b_j \) 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"
  \[ P(\text{Output bit i changes l Input bit j changes}) = 0.5 \]
  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.
RandomPartitioner
- RandomPartitioner was the default partitioner prior to Cassandra 2.1
  - Uses MD5
  - 0 to $2^{127} - 1$

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

Geohashes
(2-dimensional geospatial data to DHT)
- Used in Galileo, MongoDB
  - Proximity search
- Subdivides the globe into a hierarchy represented by strings
  - (40.573879, -105.084282) → 9XJQBDJK4XUT
  - Longer strings represent more precise coordinates
  - Strings with similar prefixes are geographically close

Example: Geohash of this Room
- 9XJQBDJK4XUT
- Example: Geohash of this Room
- 9XJQBDJK4XUT
Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room
Adjacency with the geohash algorithm

- Geohash for Colorado State University: (40.573879, -105.084282) → 9XJQEDJK4XUT

All the data points falling into this box will share the first 3 letters of their Geohash values.

All the data points falling into this box will share the first 5 letters of their Geohash values.

Nodes are organized in a ring of rings

A group of nodes manages a set of geohash spaces

Individual nodes are responsible for managing the feature space

Each node executes on a different physical machine