PART 2. SCALABLE FRAMEWORKS FOR REAL-TIME BIG DATA ANALYTICS

2. SERVING LAYER: CASE STUDY - CASSANDRA

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

FAQs
- PA2 deadline has been extended (11/7)

Today's topics
- Cassandra
- Partitioning

Non-consistent hashing vs. consistent hashing
- When a hash table is resized
  - Non-consistent hashing algorithm requires re-hash of the complete table
  - Consistent hashing algorithm requires only partial rehash of the table

Consistent hashing (1/3)

Identifier circle with m = 3
Consistent hash function assigns each node and key an m-bit identifier
Using a hashing function

Hashing value of IP address
m-bit Identifier: 2^m identifiers
m has to be big enough to make the probability of two nodes or keys hashing to the same identifier negligible
Consistent hashing (2/3)

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

- Key 2 will be stored in machine C
  \( \text{successor}(2) = 5 \)

- Machine B is the successor node of key 1.
  \( \text{successor}(1) = 1 \)

Consistent hashing (3/3)

If machine C leaves circle, \( \text{successor}(5) \) will point to A
If machine N joins circle, \( \text{successor}(2) \) will point to N

New node N

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

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

This material is built based on

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan.
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, $s_i$, that succeeds $n$ by at least $2^{i-1}$ on the identifier circle
  - i.e. $s_i = \text{successor}(n + 2^{i-1} \mod 2^m)$, where $1 \leq i \leq m$

Definition of variables for node $n_i$, using $m$-bit identifiers

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

Finger tables

### Lookup process (1/3)

- 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 $n_i$ does not know the successor of a key $k$?
  - If $n_i$ 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 $n_i$ does

### Lookup process (2/3)

- First, check the data is stored in $n_i$
  - If it is, return the data
- Otherwise,
  - $n_i$ 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
• reduced at most

After the distance between the node handling the query and the predecessor p halves in each step, and is at most \( f \) steps, the distance will be 1 (you have arrived at \( p \)).

\[ \text{Distance from } n \text{ to } p \text{ is at least } \frac{1}{2} \text{ the distance from } n \text{ to } p. \]

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

Proof continued

\( f \) and \( p \) are both in \( n \)'s \( f \) pointer interval, and the distance between them is at most \( 2^f \). This means \( f \) is closer to \( p \) than \( n \) or \( n \) is equivalent.

Distance from \( n \) 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 \( 2^f \). Within \( f \) 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 \( k \)

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 \( 2^{\log N} \).
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

```c
#define successor(n).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();
// move keys in (n', n].successor
else // if n is going to be the only node in the network
for i = 1 to m
    finger[i].node = n;
    predecessor = successor.i.predecessor;
    successor.i.predecessor = n;
```

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

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

```c
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;
```

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

```c
n.init_finger_table(n')
    for i = 1 to m
        if (finger[i].start is in (n, n.finger[i].node))
            finger[i].node = finger[i].node;
        else
            finger[i].node = n'.find_successor(finger[i].start);
```

Join 5 (After `init_finger_table(n')`)
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
  - Ask immediate neighbor and copy of its complete finger table and its predecessor
  - New node can use these table as hints to help it find the correct values

Updating fingers of existing nodes

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

```java
n.update_others();
for (i = 1 to m)

  p = find_predecessor(n - 2\(^i\) - 1);
  p.update_finger_table(n, 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 $2^{i-1}$
  - The $i^{th}$ finger of node $p$ succeeds $n$

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>Jenny</td>
<td>15</td>
<td>M</td>
<td>F</td>
</tr>
<tr>
<td>Suzy</td>
<td>9</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

- Cassandra assigns a hash value to each partition key

<table>
<thead>
<tr>
<th>Partition Key</th>
<th>Summer 3 hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>2249582670023323232</td>
</tr>
<tr>
<td>Carol</td>
<td>7772336923236907684</td>
</tr>
<tr>
<td>Jenny</td>
<td>5723337285048380876</td>
</tr>
<tr>
<td>Suzy</td>
<td>1168854627309403158</td>
</tr>
</tbody>
</table>

Cassandra cluster with 4 nodes

- Node A
  - 46165861549427867904
  - 46165861549427867903
  - Data Center ABC
- Node B
  - 22332303869473607
  - 22332303869473607
  - 46165861549427867904
  - 206500977023529
- Node C
  - 116850864739239015
  - 46165861549427867903
- Node D
  - 9223372854036780875
  - 9223372854036780875
  - 46165861549427867904
  - 206500977023529

- Cassandra assigns a hash value to each partition key

<table>
<thead>
<tr>
<th>Partition Key</th>
<th>Summer 3 hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>2249582670023323232</td>
</tr>
<tr>
<td>Carol</td>
<td>7772336923236907684</td>
</tr>
<tr>
<td>Jenny</td>
<td>5723337285048380876</td>
</tr>
<tr>
<td>Suzy</td>
<td>1168854627309403158</td>
</tr>
</tbody>
</table>
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

- Hash function quality
  
  \[
  \frac{1}{n} \sum_{j=0}^{n} \frac{k_j}{(n/2^m)(n + 2m - 1)}
  \]

  - Where, \( k_j \) 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"
  - \( P(\text{output bit } i \text{ changes} | \text{input bit } j \text{ changes}) = 0.5 \) 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 \( \frac{1}{2} \)
  - 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.

RandomPartitioner

- RandomPartitioner was the default partitioner prior to Cassandra 2.1
- Uses MD5
- 0 to \( 2^{64} - 1 \)

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