CS535 BIG DATA

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

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (SIGCOMM '01). ACM, New York, NY, USA, 149-160. DOI=http://dx.doi.org/10.1145/383059.383071

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-1} on the identifier circle
  - i.e. $s = successor(n+2^{i-1})$, where $1 \le i \le m$ (and all arithmetic is modulo 2^m)

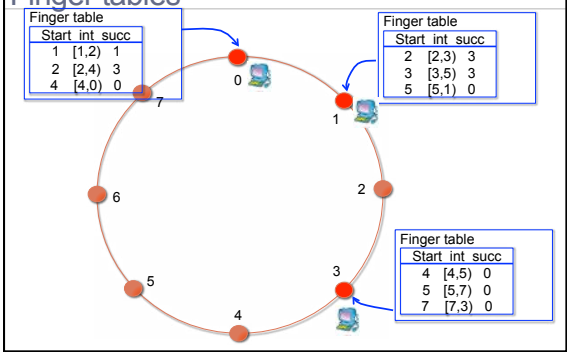
The i^{th} entry finger of node n

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

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

- 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



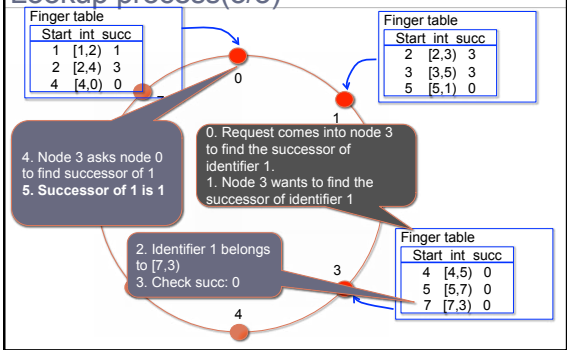
Lookup process(1/3)

- 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 does not know the successor of a key k?
  - If n finds a node whose ID is close than its own to k, that node will know more about the identifier circle in the region of k than n does

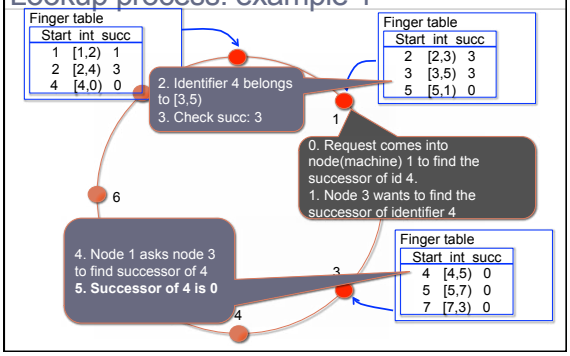
Lookup process(2/3)

- 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!**

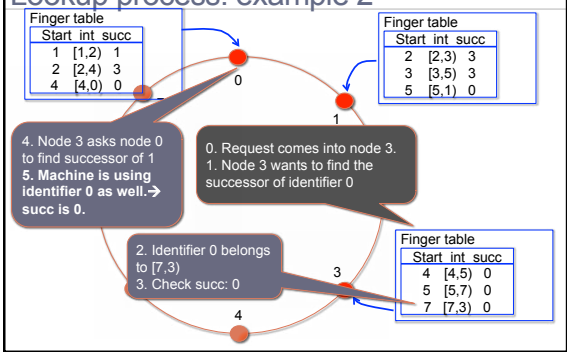
Lookup process(3/3)



Lookup process: example 1



Lookup process: example 2



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(logN)*

- 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 $O(logN)$

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 $\frac{1}{2}logN$

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 *successor(k)* is responsible for k

Tasks to perform node join

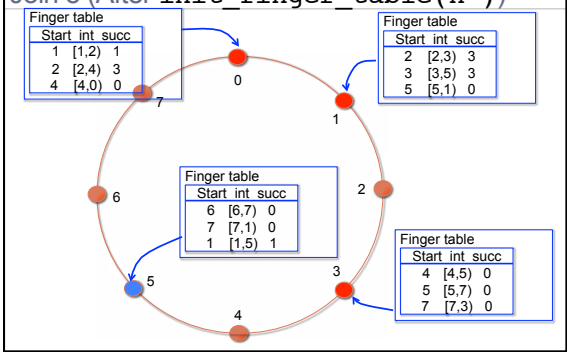
1. Initialize the predecessor and fingers of node n
2. Update the fingers and predecessors of existing nodes to reflect the addition of n
3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node n is now responsible for

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

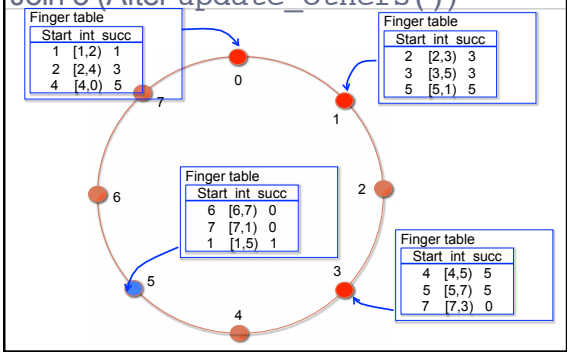


```
n.find_successor(id)
    n'=find_predecessor(id);
    return n'.successor;


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;
```


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



Join 5 (After `update_others()`)



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

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


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

- Naïve run for `find_successor` will take $O(logN)$
- For m finger entries
  - $O(mlogN)$
- **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 n can use these table as hints to help it find the correct values

Step 2: Updating fingers of existing nodes [1/2]

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

```
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)
    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);
```


Step 2: Updating fingers of existing nodes [2/2]

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

  and
  - the i^{th} finger of node p succeeds n
- The first node p that can met these two condition
  - Immediate predecessor of $n-2^{i-1}$
- 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 $O(logN)$

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

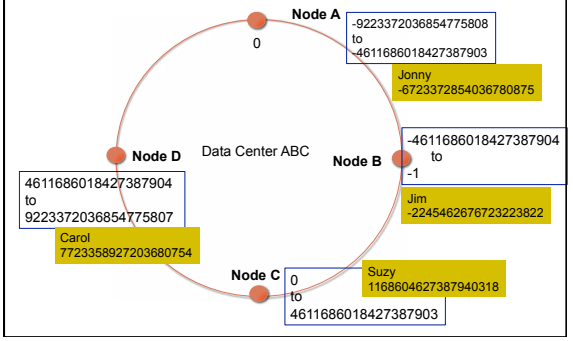
- If you have following data,

Name	Age	Car	Gender
Jim	36	Camaro	M
Carol	37	BMW	F
Jonny	10		M
Suzy	9		F

- Cassandra assigns a hash value to each partition key

Partition Key	Mumer 3 Hash value
Jim	-2245462676723223822
Carol	7723358927203680754
Jonny	-6723372854036780875
Suzy	1168604627387940318

Cluster with 4 nodes



Apache Cassandra
Partitioning

Partitioners

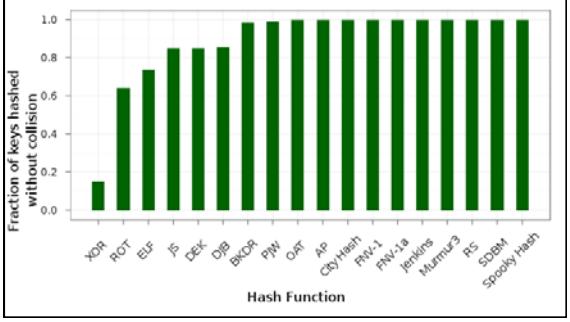
Partitioning

- 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

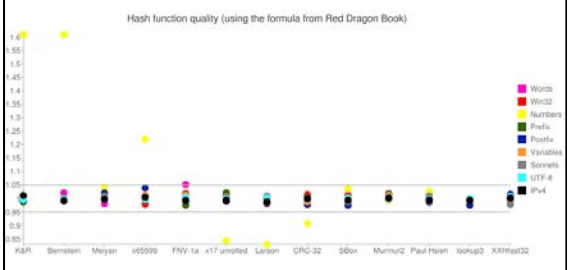
Testing with 42 Million keys



Measuring the quality of hash function

- Hash function quality $\sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)}$
  - 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
    - A. V. Aho, M. S. Lam, R. Sethi and J. D Ullman, "Compilers, Principles, Techniques, and Tools", Pearson Education, Inc.

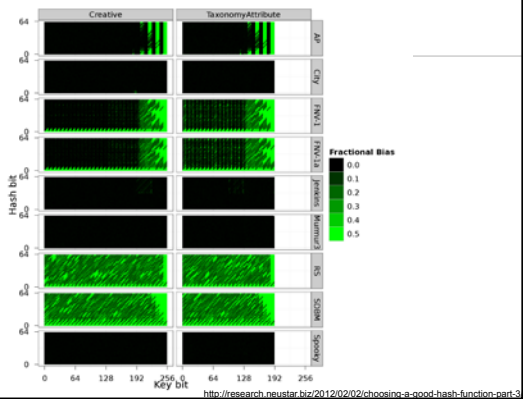
Comparison between hash functions



http://www.strchr.com/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"
  - $P(\text{Output bit } i \text{ changes} \mid \text{Input bit } j \text{ changes}) = 0.5$ $\text{for all } i, j$
- 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 ½
- 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.



http://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3

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

Example: Geohash of this Room

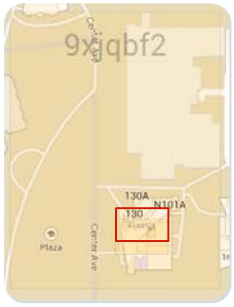
Example: Geohash of this Room

Example: Geohash of this Room

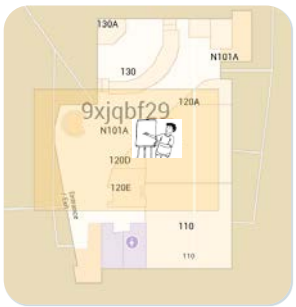
Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

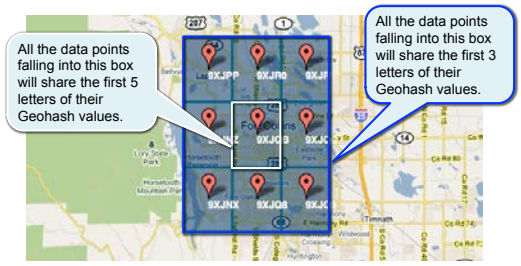


Example: Geohash of this Room



Adjacency with the geohash algorithm

- Geohash for Colorado State University:
  (40.573879, -105.084282)→9XJQBDJK4XUT



Nodes are organized in a ring of rings