CS535 BIG DATA

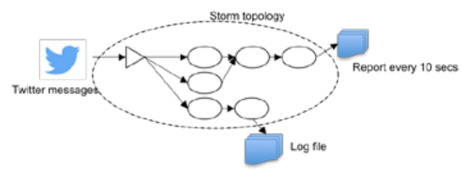
PART 2. SCALABLE FRAMEWORKS FOR REAL-TIME BIG DATA ANALYTICS

1. APACHE STORM

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

FAQs

- Google credit available
- Assignment 2 has been posted
- URL for zookeeper has been updated



Today's topics

- Storm model
  - Trident
  - Cassandra

Trident **functions**

- Consume **tuples** and optionally emit new **tuples**
- Trident functions are additive
  - The values emitted by functions are fields that are added to the tuple
  - They do not remove or mutate existing fields

```
public interface Function extends EachOperation {
      void execute(TridentTuple tuple,
                   TridentCollector collector);
}
```


Writing your BaseFunction

```
public class CityAssignment extends BaseFunction {
       private static final long serialVersionUID = 1L;
       private static final Logger LOG=
 LoggerFactory.getLogger( CityAssignment.class);
       private static Map < String, double[] > CITIES = new HashMap <
String, double[] >();
      {
            // Initialize the cities we care about.
            double[] phl = {39.875365, -75.249524 };
            CITIES.put("PHL", phl);
            double[] nyc = {40.71448, -74.00598 };
            CITIES.put("NYC", nyc);
            double[] sf = {-31.4250142, -62.0841809 };
            CITIES.put("SF", sf);
            double[] la = {-34.05374, -118.24307 };
            CITIES.put("LA", la);
       }
```


Writing your BaseFunction

```
       @Override
       public void execute( TridentTuple tuple, TridentCollector
collector) {
              DiagnosisEvent diagnosis=
                     (DiagnosisEvent)tuple.getValue(0);
              double leastDistance = Double.MAX_VALUE;
              String closestCity = "NONE";
              // Find the closest city.
              for (Entry < String, double[] > city : CITIES.entrySet()) {
                     double R = 6371;
                     // km
                     double x = (city.getValue()[0] - diagnosis.lng) *
                  Math.cos(( city.getValue()[0] + diagnosis.lng) / 2);
                     double y = (city.getValue()[ 1] - diagnosis.lat);
                     double d = Math.sqrt( x * x + y * y) * R;
                     if (d < leastDistance) {
                            leastDistance = d;
                            closestCity = city.getKey();
                     }
              }
```

Writing your BaseFunction

```
        // Emit the value.
        List < Object > values = new ArrayList < Object >();
        Values.add(closestCity);
        LOG.debug("Closest city to lat =[" + diagnosis.lat + "],
              lng =[" + diagnosis.lng + "] = = [" + closestCity + "],
              d =[" + leastDistance + "]");
        collector.emit(values);
        }
}
```


Trident aggregator

- Allows topologies to combine tuples
  - They replace tuple fields and values
    - Function does not change
  - `CombinerAggregator`
  - `ReducerAggregator`
  - `Aggregator`

CombinerAggregator

- Combines a set of tuples into a single field
- Storm calls the `init()` method with each tuple then repeatedly calls `combine()` method until the partition is processed

```
public interface CombinerAggregator {
      T init (TridentTuple tuple);
      T combine( T val1, T val2);
      T zero(); //if the partition is empty
}
```


ReducerAggregator

```
public interface ReducerAggregator < T > extends
Serializable {
      T init();
      T reduce(T curr, TridentTuple tuple);
}
```

- Storm calls the `init()` method to retrieve the initial value
- Then `reduce()` is called with each tuple until the partition is fully processed
- The first parameter into the `reduce()` method is **the cumulative partial aggregation**
- The implementation should return the result of incorporating the tuple into that partial aggregation

Aggregator [1/2]

- The most general aggregation operation

```
public interface Aggregator < T > extends Operation {
      T init( bject batchId, TridentCollector collector);
      void aggregate(T val, TridentTuple tuple,
TridentCollector collector);
      void complete(T val, TridentCollector collector);
}
```

- The `aggregate()` method is similar to the `execute()` method of a Function interface
  - It also includes a parameter for the value
  - This allows the Aggregator to accumulate a value as it processes the tuples. Notice that with an Aggregator, the collector is passed into both the `aggregate()` method as well as the `complete()` method
  - You can emit any arbitrary number of tuples

Aggregator [2/2]

- A key difference between Aggregator and other Trident aggregation interfaces
  - An instance of TridentCollector is passed as a parameter to every method
  - This allows Aggregator implementations to emit tuples at any time during execution

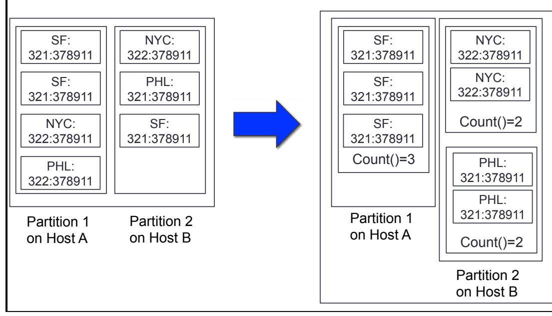
Writing and applying `Count`

```
public class Count implements CombinerAggregator < Long > {
      @Override
      public Long init(TridentTuple tuple){
            return 1L;
      }
      @Override
      public Long combine( Long val1, Long val2) {
            return val1 + val2;
      }
      @Override
      public Long zero() {
            return 0L;
      }
}
```

*Applying grouping and counting

```
.groupBy(new Fields("cityDiseaseHour"))
.persistentAggregate(new OutbreakTrendFactory(), new Count(), new
Fields("count")). newValuesStream()
```


Results



Trident state

- Trident has a first-level primitive for state
- State interface

```
public interface State {
      void beginCommit(Long transactionId);
      void commit(Long transactionId);
}
```

- Each batch (of tuples) has its own transaction identifier
- State object specifies when the state is being committed and when the commit should complete

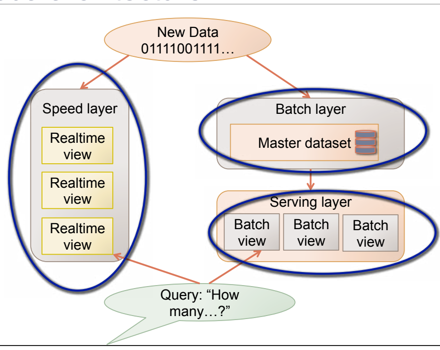
CS535 BIG DATA

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

Lambda architecture



Apache Cassandra

This material is built based on,

- Avinash Lakshman, Prashant Malik, "A Decentralized Structured Storage System" ACM SIGOPS Operation Systems Review, Vol. 44-(2), April 2010 pp. 35-40
- 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

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,
  - System should handle a very high write throughput
    - 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

Apache Cassandra

Data Model

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

Data Model (2/2)

- Columns are grouped into column families
  - Similar to Bigtable
- Columns are sorted within a simple column or super columns
  - Sorted by time or by name

Super column family vs. Simple column family

```
"alice": {
  "ccd17c10-d200-11e2-b7f6-29cc17aeed4c": {
    "sender": "bob",
    "sent": "2013-06-10 19:29:00+0100",
    "subject": "hello",
    "body": "hi"
  }
}
```

- Simple column family
  - Some uses require more dimensions
  - Family of values
    e.g. messages
- Cassandra's native data model is two-dimensional
  - Rows and columns.
  - Columns that contain columns

API

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

Apache Cassandra

Partitioning

Partitioning

- Cassandra is a **partitioned row store database**
- Uses consistent hashing
  - Order preserving hash function
  - Joining and deletion of node only affects its immediate neighbors
  - System **scales incrementally**
- `Partitioners` determines how data is distributed across the nodes in the cluster
  - `Murmur3 Partitioner`
  - `Random Partitioner`
  - `ByteOrdered Partitioner`

Apache Cassandra

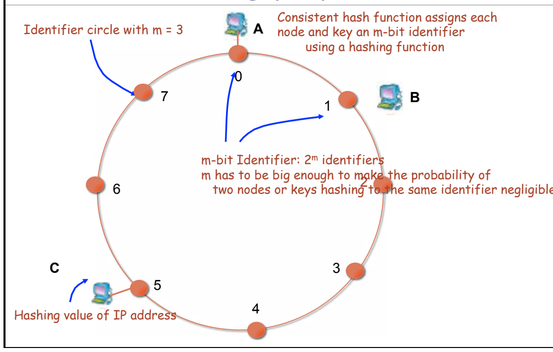
Partitioning

Consistent Hashing

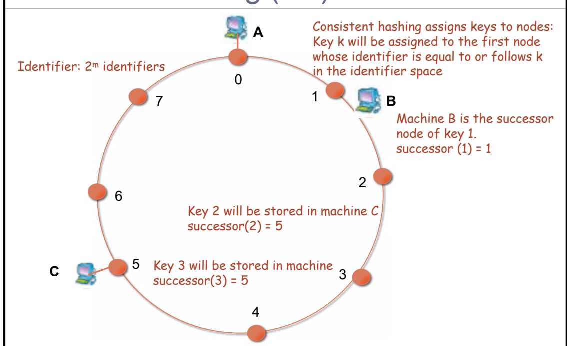
Non-consistent hashing vs. consistent hashing

- 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

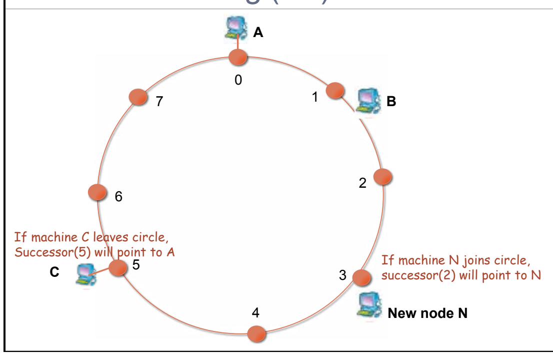
Consistent hashing (1/3)



Consistent hashing (2/3)



Consistent hashing (3/3)



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

Apache Cassandra

Partitioning

Consistent Hashing: Chord

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

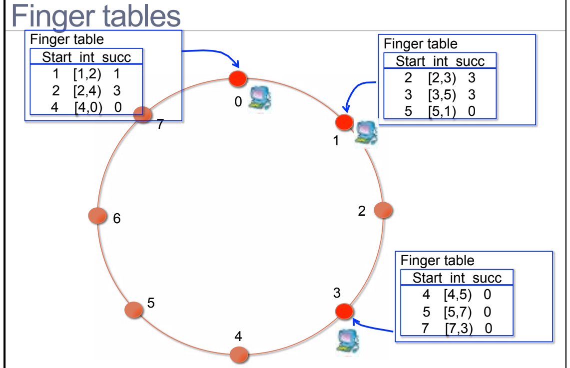
Definition of variables for node n, using m-bit identifiers

- *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



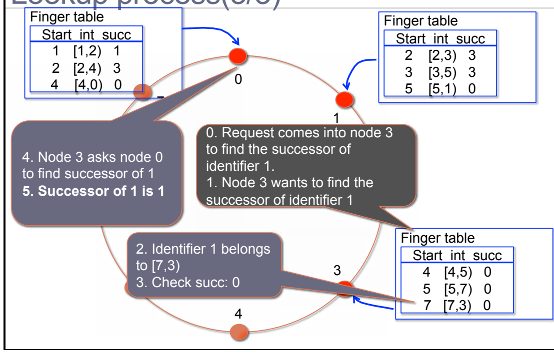
Lookup process(1/3)

- 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

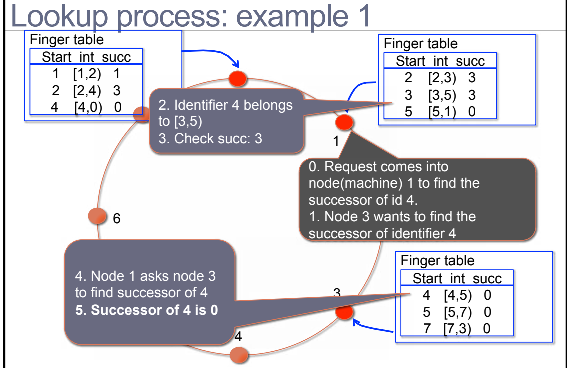
Lookup process(2/3)

- 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!**

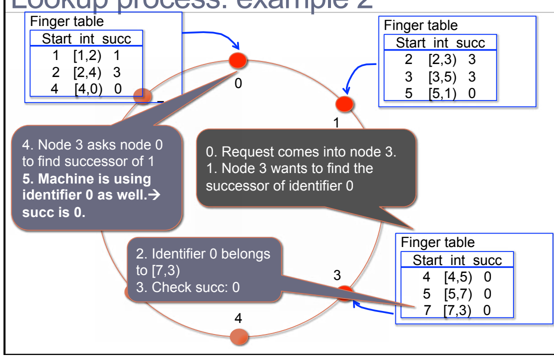
Lookup process(3/3)



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;
```



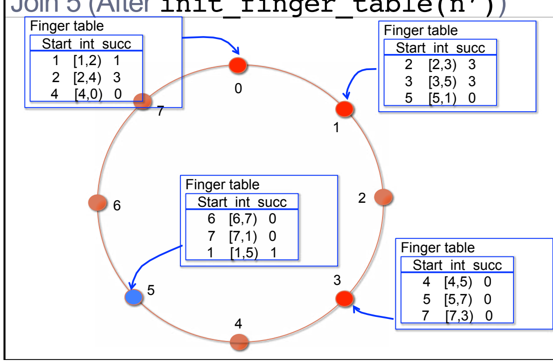
```
n.find_successor(id)
    n'=find_predecessor(id);
    return n'.successor;

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;
```


Step1:
Initializing fingers and predecessor (1/2)

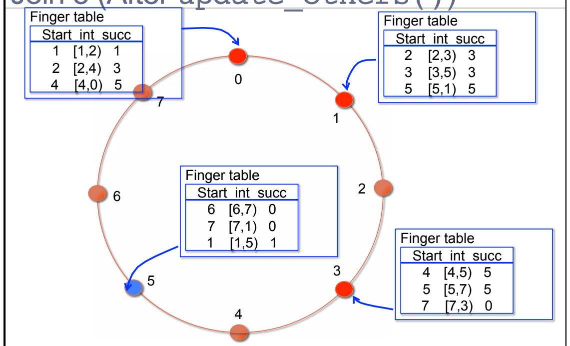
- New node *n* learns its predecessor and fingers by asking any arbitrary node in the network *n'* to look them up

```
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);
```


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



Join 5 (After `update_others()`)



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 $(i+1)^{th}$ 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

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 = 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);
```


- 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 $O(logN)$

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

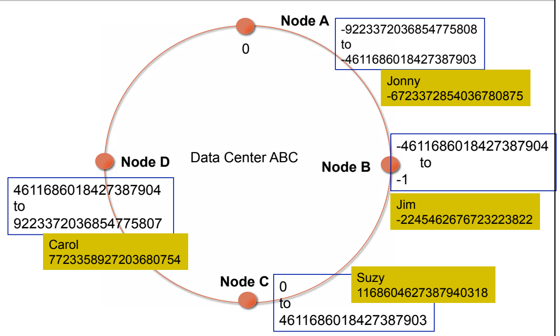
- If you have following data,

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

Cassandra cluster with 4 nodes



Apache Cassandra
Partitioning
Partitioners

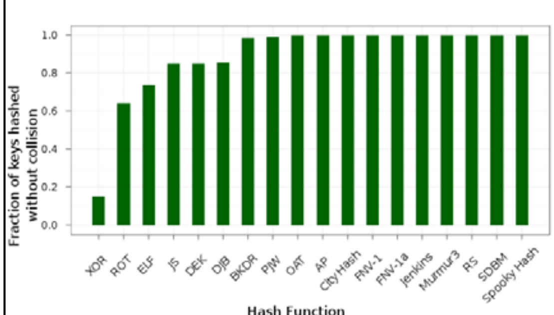
Partitioning

- 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.
  - **ByteOrderedPartitione**r: 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

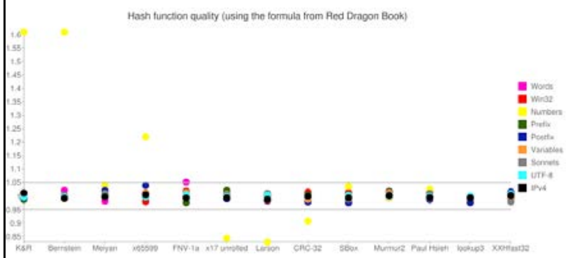
Testing with 42 Million keys



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.

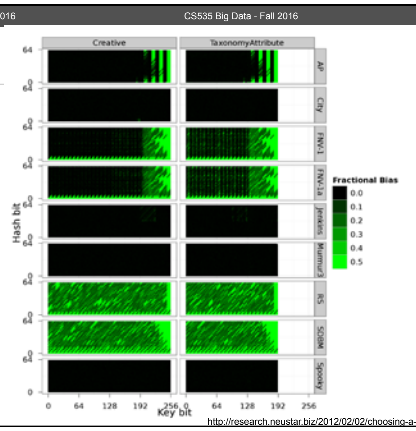
Comparison between hash functions



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(Output bit i changes | 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 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

Apache Cassandra
Replication

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

`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` (1/3)

- 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.

`NetworkTopologyStrategy` (2/3)

- 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 scenario
- 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.

`NetworkTopologyStrategy` (3/3)

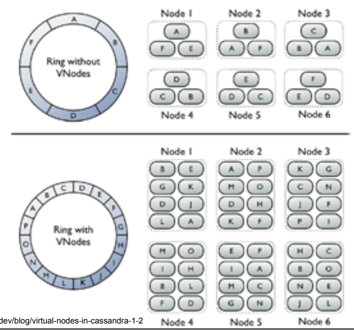
- Asymmetrical replication groupings
  - For example, you can maintain 4 replicas
    - Three replicas in one data center to serve real-time application requests
    - A single replica elsewhere for running analytics.

Apache Cassandra

Virtual Node

What are Vnodes?

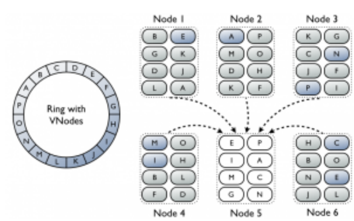
- With consistent hashing, a node owns exactly one contiguous range in the ring-space
- Vnodes change from one token or range per node, to many per node
  - Within a cluster these can be randomly selected and be non-contiguous, giving us many smaller ranges that belong to each node



http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2

Advantages of Vnodes

- Example
  - 30 nodes and replication factor of 3
  - A node dies completely, and we need to bring up a replacement
  - A replica for 3 different ranges to reconstitute
    - 1 set of the first natural replica
    - 2 sets of replica for replication factor of 3
  - Since our RF is 3 and we lost a node, we logically only have 2 replicas left, which for 3 ranges means there are up to 6 nodes we can stream from
  - With the setup of RF3, data will be streamed from 3 other nodes total



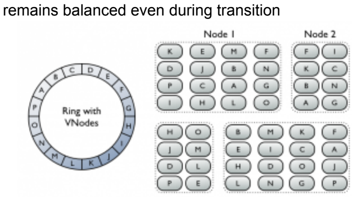
- If vnodes are spread throughout the entire cluster
  - Data transfers will be distributed on more machines

Restoring a new disk with vnodes

- Process of restoring a disk
  - Validating all the data and generating a Merkle tree
    - This might take an hour
  - Streaming when the actual data that is needed is sent
    - This phase takes a few minutes
- Advantage of using Vnodes
  - Since the ranges are smaller, data will be sent to the damaged node in a more incremental fashion
    - Instead of waiting until the end of a large validation phase
  - The validation phase will be parallelized across more machines, causing it to complete faster

The use of heterogeneous machines with vnodes

- Newer nodes might be able to bear more load immediately
  - You just assign a proportional number of vnodes to the machines with more capacity
  - e.g. If you started your older machines with 64 vnodes per node and the new machines are twice as powerful, give them 128 vnodes each and the cluster remains balanced even during transition



Apache Cassandra

Gossip (Internode communications)

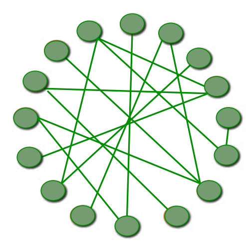
Use of Gossip in Cassandra

- Peer-to-peer communication protocol
  - Periodically exchange state information about nodes themselves and about other nodes they know about
- Every node talks to up to three other nodes in the cluster
- A gossip message has a version associated with it
  - During a gossip exchange, older information is overwritten with the most current state for a particular node

What is gossip?

- Broadcast protocol for disseminating data
- Decentralized, peer-to-peer networks
- 'epidemic'
- Fault tolerant
- Epidemic broadcast protocol provides a resilient and efficient mechanism for data dissemination
- Cassandra uses gossip for peer discovery and metadata propagation

Broadcast protocol for disseminating data



Why gossip for Cassandra?

- Reliably disseminate node metadata to peers
  - Cluster membership
  - Heatbeat
  - Node status
  - Each node maintains a view of all peers

What gossip is not for in Cassandra?

- Streaming
- Repair
- Reads/write
- Compaction
- Hint
- CQL query parsing/execution

Data structure

- `HeartBeatState`
- `ApplicationState`
- `EndpointState`
  - Wrapper of a heartbeat state and a set of application state

HeartBeatState

- Generation
- Heartbeat
  - Periodically update monotonically increasing value

Application state

- {enum_name, value, version}
- Contained as a map in `EndpointState` per peer

`ApplicationState enum`

- DC/RACK
  - Where you are
- SCHEMA
- LOAD
  - Updated every 60 seconds
- SEVERITY
  - I/O load
- STATUS

Status (`AppState`)

- Bootstrap
  - For new nodes
- Hibernate
- Normal
- Leaving/Left
- Removing/Removed

Gossip messaging

- Every second, each node starts a new round
- Peer selection (1-3 peers)
  - Live peer
  - Seed (maybe)
  - Unreachable peer(maybe)

Gossip Exchange

- SYN/ACK/ACK2
  - Similar to TCP 3-way handshake
  - Add anti-entropy to gossiping

SYN: GossipDigestSynMessage

- Initiator sends a digest of all the nodes it knows about to a peer
- {ipAddr, generation, heartbeat}

GossipDigestSynMessage

```
#Suppose we are in 10.0.0.1

EndPointState 10.0.0.1
  HeartBeatState: generation 1259909635, version 325
  ApplicationState "load-information": 5.2, generation 1259909635, version 45
  ApplicationState "bootstrapping": bxLpassF3XD8Kyks, generation 1259909635, version 56
  ApplicationState "normal": bxLpassF3XD8Kyks, generation 1259909635, version 87
EndPointState 10.0.0.2
  HeartBeatState: generation 1259911052, version 61
  ApplicationState "load-information": 2.7, generation 1259911052, version 2
  ApplicationState "bootstrapping": AujDMftpyUvebtnn, generation 1259911052, version 31
EndPointState 10.0.0.3
  HeartBeatState: generation 1259912238, version 5
  ApplicationState "load-information": 12.0, generation 1259912238, version 3
EndPointState 10.0.0.4
  HeartBeatState: generation 1259912942, version 18
  ApplicationState "load-information": 6.7, generation 1259912942, version 3
  ApplicationState "normal": bj05IVc0lvRXw2xH, generation 1259912942, version 7
```

Max version number for this endpoint: 325, 61, 5, and 18
GossipDigestSynMessage
10.0.0.1:1259909635:325
10.0.0.2:1259911052:61
10.0.0.3:1259912238:5
10.0.0.4:1259912942:18

Source: https://wiki.apache.org/cassandra/ArchitectureGossip

ACK: GossipDigestActMessage

- Peer receives `GossipDigestSynMessage`
- Sort gossip digest list according to the difference in max version number between sender's digest and own information in descending order
  - Handle those digests first that differ mostly in version number
- Produces a diff and sends back an ACK
- Diff contains
  - Map of APPStates (for any node) that the peer has which the initiator does not
  - Digest of nodes (and their corresponding metadata) which a peer needs from an initiator

GossipDigestActMessage

```
# Suppose we are in 10.0.0.2
# The EndPointState in 10.0.0.2 will be

EndPointState 10.0.0.1
  HeartBeatState: generation 1259909635, version 324
  ApplicationState "load-information": 5.2, generation 1259909635, version 45
  ApplicationState "bootstrapping": bxLpassF3XD8Kyks, generation 1259909635, version 56
  ApplicationState "normal": bxLpassF3XD8Kyks, generation 1259909635, version 87
EndPointState 10.0.0.2
  HeartBeatState: generation 1259911052, version 63
  ApplicationState "load-information": 2.7, generation 1259911052, version 2
  ApplicationState "bootstrapping": AujDMftpyUvebtnn, generation 1259911052, version 31
  ApplicationState "normal": AujDMftpyUvebtnn, generation 1259911052, version 62
EndPointState 10.0.0.3
  HeartBeatState: generation 1259812143, version 2142
  ApplicationState "load-information": 16.0, generation 1259812143, version 1803
  ApplicationState "normal": W2U1XYUC3wMppcY7, generation 1259812143, version 6
```

```
# GossipDigestSynMessage
From 10.0.0.1
10.0.0.1:1259909635:325
10.0.0.2:1259911052:61
10.0.0.3:1259912238:5
10.0.0.4:1259912942:18
```

```
# The GossipDigestActMessage from 10.0.0.2 is
10.0.0.1:1259909635:324
10.0.0.3:1259912238:0
10.0.0.4:1259912942:0
10.0.0.2:[ApplicationState "normal": AujDMftpyUvebtnn, generation 1259911052, version 62], [HeartBeatState, generation 1259911052, version 63]
```


ACK2: GossipDigestAct2Message

- Initiator receives ACK
  - Applies any AppState and sends back an ACK2
  - ACK2 has a map of AppStates which the peer does not have

```
# The GossipDigestAct2Message from 10.0.0.1 is
10.0.0.1:[ApplicationState "load-information": 5.2, generation 1259909635, version 45],
    [ApplicationState "bootstrapping": bxLpassF3XD8Kyks, generation 1259909635, version 56],
    [ApplicationState "normal": bxLpassF3XD8Kyks, generation 1259909635, version 87],
    [HeartBeatState, generation 1259909635, version 325]
10.0.0.3:[ApplicationState "load-information": 12.0, generation 1259912238, version 3],
    [HeartBeatState, generation 1259912238, version 3]
10.0.0.4:
    [ApplicationState "load-information": 6.7, generation 1259912942, version 3],
    [ApplicationState "normal": bj05IVc0lvRXw2xH, generation 1259912942, version 7],
    [HeartBeatState: generation 1259912942, version 18]
```

AppState Reconciliation

- Generation
- Heartbeat
- AppState based on comparing version

Reconciliation example

	A	
A	gen:1234 Hb: 994 Status: normal {4}	gen:1234 Hb: 990 Status: normal {4}
B	Gen:2345 Hb: 10 Status: bootstrap {1}	Gen:2345 Hb:17 Status: normal {2}
C	Gen:5555 Hb: 1111 Status: normal {5}	
D	Gen:2222 Hb: 4444 status: normal {3}	Gen:3333 Hb: 11 Status: normal {3}

Messaging summary

- Each node starts a gossip round every second
- 1-3 peers per round
- 3 messages passed
- Constant amount of network traffic

Practical implications

- Who is in the cluster?
- How are peers judged UP or DOWN?
- When does a node stop sending a peer traffic?
- When is one peer preferred over another?
- When does a node leave the cluster?

Cluster membership

- Gossip with a seed upon startup
- Learn about all peers
- Gossip
- Lather, rinse, repeat

UP/DOWN?

- Local to each node
  - Not shared via gossip
- Determined via heartbeat

Failure Detection

- Glorified heartbeat listener
- Records timestamp when heartbeat update is received for each peer
- Keeps backlog of timestamp intervals between updates
- Periodically checks all peers to make sure that we've heard from them recently

UP/DOWN affects

- Stop sending writes (hints)
- Sending reads
- Gossip
  - It is down
  - This node is treated as an unavailable node
- Repair/stream sessions are terminated

What if a peer is really slow?

- Peer is NOT marked down
  - We will try to avoid it

Dynamic "Snitch"

- Determine when to avoid a slow node
- Scoring peers based on response times
  - Scores recalculated every 100ms (default)
  - Scores reset every 10m (default)

How do nodes leave?

- STATUS = LEAVING
- Stream data
- Stream hints
- STATUS = LEFT, expiryTime

Decomission

- STATUS = LEAVING
- Stream data
- Stream hints
- STATUS = LEFT, expiryTime

Remove node

- STATUS = REMOVING
- Rebalance cluster
  - Notify coordinator
- Delete hint
- STATUS = REMOVED, expiryTime

Replace node

- Cassandra.replace_address
- "shadow gossip"
- Take tokens/hostID(hints)
  - Check that previous owner hasn't gossiped
- Stream data

"Assassinate!"

- Managing hanging non-functional nodes
- `unsafeAssassinateEndpoint(ipAddr)`
  - Use with caution
- Forces change to peer

Failure detection: Φ Accrual Failure Detector

Failure detection (1/3)

- Φ Accrual Failure Detector
- Accrual Failure Detection does not emit a Boolean value stating a node is up or down
  - Emits a value which represents a suspicion level for each of the monitored nodes
  - This value is defined as Φ
- Dynamically adjusts to reflect network and load conditions at the monitored nodes

Failure detection (2/3)

- Given some threshold Φ, and assuming that we decide to suspect a node A
  - e.g. when Φ =1, then the likelihood that we will make a mistake is about 10%.
  - The likelihood is about 1% with Φ =2
  - The likelihood is about 0.1% with Φ =3

Failure detection (3/3)

- Every node maintains a sliding window of inter-arrival times of gossip messages from other nodes in the cluster
- Exponential Distribution
  - The nature of gossip channel and its impact on latency

Bootstrapping and persistence

Bootstrapping

- When a node joins the ID ring, the mapping is persisted to the disk locally and in Zookeeper
  - Then the token information is gossiped around the cluster
- With bootstrapping, a node joins with a configuration file that contains a list of a few contact points
  - Seeds of the cluster
- Seeds can be provided by a configuration service (e.g. Zookeeper)

Local persistence: Write Operation

- Write into a commit log
  - Durability and recoverability
  - Dedicated disk for each node
- Write into an in-memory data structure
  - When in-memory data structure crosses a certain threshold, it dumps itself to disk
- Write into disk
  - Generates an index for efficient lookup based on row key
- Similar to Bigtable (compaction)

Local persistence: Read Operation

- First queries the in-memory data structure
- Disk lookup
  - Look-up a key
- To narrow down the lookup process
  - a bloom filter is stored in each data file and memory