### FAQS

- PA2 has been posted (11/6)
- If you want to specify non-default port numbers to your nimbus
  - logviewer.port
  - nimbus.thrift.port
  - drpc.port
  - drpc.http.port
  - drpc.invocations.port
  - pacemaker.port

### Today's topics

- Trident Topology
- Introduction to Cassandra

### Trident Topologies

- Trident is a Java API that translates micro-batch processing topologies into the spouts and bolts of Storm
- Eliminates the details of transactional processing and state management
- Batching of tuples into a discrete set of transactions
- Abstracting operations on the data such as functions, filters and aggregations
Example case (1/2)

- Collecting medical reports to identify the outbreak of a disease

The topology will process diagnosis events that contain:

- Latitude
- Longitude
- Timestamp
- Diagnosis Code (ICD-9-CM)

E.g.,

- {39.9522, 75.1642, “03/13/2013 at 3:30 PM”, “320.0 (Hemophilus meningitis)”}

- Each event includes the Global Positioning System (GPS) coordinates of the occurrence

Example case (2/2)

- To detect an outbreak,
  1. The system will count the occurrence of specific disease codes within geographic location over a specified period of time
  2. The system will group the occurrences by hour and calculate a trend against the moving average
  3. The system will use a simple threshold to determine if there is an outbreak
  4. If the count of occurrences for the hour is greater than some threshold, the system will send an alert

Workflow for our example case

```
// Detect an outbreak
.each(new Fields("cityDiseaseHour", "count"),
    new OutbreakDetector(),
    new Fields("alert"));
// Dispatch the alert
.each(new Fields("alert"),
    new DispatchAlert(),
    new Fields());
```

Introducing Trident Spout

- Batch
  - Trident spouts must emit tuples in batches
- Composition of a batch
  - Non-transactional
    - No guarantee on the composition of the batches and might overlap
    - Two different batches might contain the same tuples
  - Transactional
    - Guaranteed and non-overlapping
    - Same batch contains the same tuples
  -Opaque
    - Guaranteed and non-overlapping
    - Contents of a batch may change

```java
public static StormTopology buildTopology() {
    TridentTopology topology = new TridentTopology();
    DiagnosisEventSpout spout = new DiagnosisEventSpout();
    Stream inputStream = topology.newStream("event", spout);
    // Filter for critical events.
    inputStream.each(new Fields("event"), new DiseaseFilter());
    // Locate the closest city
    inputStream.each(new Fields("event"), new CityAssignment(),
        new Fields("city"));
    // Derive the hour segment
    inputStream.each(new Fields("event", "city"),
        new HourAssignment(),
        new Fields("hour", "cityDiseaseHour"));
    // Group occurrences in same city and hour
    inputStream.groupby(new Fields("cityDiseaseHour"))
        .persistentAggregate(new OutbreakTrendFactory()),
        new Count(),
        new Fields("count"));
```
Trident Spout interface

```java
public interface TridentSpout extends Serializable {
    @Override public Emitter < T > getEmitter(String txStateId, Map conf, TopologyContext context);
    @Override public BatchCoordinator < T > getCoordinator(String txStateId, Map conf, TopologyContext context);
    @Override public Emitter < T > getEmitter(String txStateId, Map conf, TopologyContext context);
    @Override public class DiagnosisEventSpout implements TridentSpout < Long > {
        private static final long serialVersionUID = 1L;
        private static final Logger LOG = LoggerFactory.getLogger(BatchCoordinator.class);
        @Override public void close() {
            super.close();
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
    }
}
```

Continued

```java
public interface TridentSpout extends Serializable {
    @Override public Emitter < T > getEmitter(String txStateId, Map conf, TopologyContext context) {
        return emitter;
    }
    @Override public BatchCoordinator < T > getCoordinator(String txStateId, Map conf, TopologyContext context) {
        return null;
    }
    public class DiagnosisEventSpout implements TridentSpout < Long > {
        private static final long serialVersionUID = 1L;
        private static final Logger LOG = LoggerFactory.getLogger(BatchCoordinator.class);
        @Override public void close() {
            super.close();
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
    }
}
```

```java
public interface TridentSpout extends Serializable {
    @Override public Emitter < T > getEmitter(String txStateId, Map conf, TopologyContext context) {
        return emitter;
    }
    @Override public BatchCoordinator < T > getCoordinator(String txStateId, Map conf, TopologyContext context) {
        return null;
    }
    public class DiagnosisEventSpout implements TridentSpout < Long > {
        private static final long serialVersionUID = 1L;
        private static final Logger LOG = LoggerFactory.getLogger(BatchCoordinator.class);
        @Override public void close() {
            super.close();
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
    }
}
```

```java
public interface TridentSpout extends Serializable {
    @Override public Emitter < T > getEmitter(String txStateId, Map conf, TopologyContext context) {
        return emitter;
    }
    @Override public BatchCoordinator < T > getCoordinator(String txStateId, Map conf, TopologyContext context) {
        return null;
    }
    public class DiagnosisEventSpout implements TridentSpout < Long > {
        private static final long serialVersionUID = 1L;
        private static final Logger LOG = LoggerFactory.getLogger(BatchCoordinator.class);
        @Override public void close() {
            super.close();
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
        @Override public void success(TransactionAttempt ta) {
            super.success(ta);
        }
    }
}
```

```
```
DiagnosisEvent

- ICD-9-CM codes
- 320 Bacterial meningitis
- 321 Meningitis due to other organisms
- 322 Meningitis of unspecified cause
- 323 Encephalitis myelitis and encephalomyelitis

```
public class DiagnosisEvent implements Serializable {
    private static final long serialVersionUID = 1L;
    public double lat;
    public double lng;
    public long time;
    public String diagnosisCode;

    public DiagnosisEvent(double lat, double lng, long time, String diagnosisCode) {
        super();
        this.time = time;
        this.lat = lat;
        this.lng = lng;
        this.diagnosisCode = diagnosisCode;
    }
}
```

Trident operations - filters and functions

- Operations
  - Adding the logic components that implement the business process
- Filters
  - Functions
  - Join
  - Aggregation
  - Group
- Implementing methods on the `Stream` object

```
public class Stream implements IAggregatableStream {
    public Stream each( Fields inputFields, Filter filter) {
        ...
    }
    public IAggregatableStream each( Fields inputFields, Function function, Fields functionFields) {
        ...
    }
    public GroupedStream groupBy( Fields fields) {
        ...
    }
    public TridentState persistentAggregate( StateFactory stateFactory, CombinerAggregator agg, Fields functionFields) {
        ...
    }
}
```

Revisit the disease outbreak example code

```
inputStream.each( new Fields(" event"), new DiseaseFilter());
.each( new Fields("event"), new CityAssignment());
.each( new Fields("event", "city"),
    new HourAssignment(),
    new Fields("hour", "cityDiseaseHour"))
.groupBy( new Fields("cityDiseaseHour"))
.persistentAggregate( new OutbreakTrendFactory(),
    new Count(),
    new Fields(" count")
    .newValuesStream();
.each( new Fields("cityDiseaseHour", "count"),
    new OutbreakDetector(),
    new Fields(" alert"))
.each( new Fields("alert"),
    new DispatchAlert(),
    new Fields());
```

Trident filters

- For example, the system wants to ignore disease events that are not of concern
- Focus on meningitis (code 320,321, and 322)
- Providing a `BaseFilter` class

```
public interface Filter extends EachOperation {
    boolean isKeep( TridentTuple tuple);
}
```
DiseaseFilter
public class DiseaseFilter extends BaseFilter {
    private static final long serialVersionUID = 1L;
    private static final Logger LOG = LoggerFactory.getLogger(DiseaseFilter.class);

    @Override
    public boolean isKeep(TridentTuple tuple) {
        DiagnosisEvent diagnosis = (DiagnosisEvent) tuple.getValue(0);
        Integer code = Integer.parseInt(diagnosis.diagnosisCode);
        if (code.intValue() <= 322) {
            LOG.debug(" Emitting disease [" +
                      diagnosis.diagnosisCode + "]");
            return true;
        } else {
            LOG.debug(" Filtering disease [" +
                      diagnosis.diagnosisCode + "]");
            return false;
        }
    }
}

Applying filter to each tuple
inputStream.each(new Fields(" event"), DiseaseFilter())

Trident functions
Consumes 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

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 = { 37.7749297, -122.419416 };
    CITIES.put(" SF", sf);
    double[] la = { 34.05374, -118.24307 };
    CITIES.put(" LA", la);

    @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();
            }
        }

        // 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);
    }
}
Applying function to each tuple

```java
inputStream.each(new Fields("event"), new DiseaseFilter())
.each(new Fields("event"), new CityAssignment(), new Fields("city"))
```

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

```java
public interface CombinerAggregator {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
    T zero(); // emits and returns value
}
```

ReducerAggregator

- 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

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

Aggregator

- The most general aggregation operation

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

Writing and applying Count

```java
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

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

Lambda architecture

2. Serving layer: Case study-Cassandra
Introduction to Apache Cassandra

This material is built based on,
- Datastax Documentation: Apache Cassandra
- 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
  - 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
  - Reakt
  - The Weather Channel
  - And over 1500 more companies

2. Serving layer: Case study-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

- 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)`

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

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
- $m$ has to be big enough to make the probability of two nodes or keys hashing to the same identifier negligible

**Consistent hashing (2/3)**

- Key 2 will be stored in machine $C$
- 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

---

2. Serving layer: Case study-Cassandra
   
   Consistent Hashing: Chord

---

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 \((\text{at most}) \ m\) entries
  - Called the finger table

- The \( i \text{th} \) entry in the table at node \( n \), contains the identity of the first node, \( s \).
  - \( s = \text{successor}(n + 2^{i-1}) \), where \( \text{successor} \) (and all arithmetic is modulo \( 2^m \))

Finger table

- The Chord identifier
- The IP address of the relevant node

- First finger of \( n \) is its immediate successor on the circle
  - Clockwise!

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 closer than its own to \( k \), that node will know more about the identifier circle in the region of \( k \) than \( n \) does

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

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

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!

Finger tables

- Start
- int
- succ
- 1
- 0
- 2
- 4
- 6
- 3
- 5
- 7
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 0
- Interval
- [1, 2)
- [2, 4)
- [4, 0)
- [2, 3)
- [3, 5)
- [5, 1)
- [4, 5)
- [5, 7)
- [7, 3)
- Node
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 0

Proof continued

Let \( f \) and \( p \) be both in \( n \)'s \( i \)th finger interval, and the distance between them is at most \( 2^{-i} \). 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 \( 2^n \) 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 \( \frac{1}{2}\log N \)

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 \( 2^{-i} \).

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 \( k \)

\[ \square \]
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 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;
```
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 $(i+1)$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 $2^i - 1$
  - $i$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 $a$ can become the successor only for keys that were previously the responsibility of the node immediately following $a$
  - $a$ 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>10</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Sury</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>
<th>Summer 4 Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>7223372854036780875</td>
<td>7223372854036780875</td>
</tr>
<tr>
<td>Carol</td>
<td>7723395672305685764</td>
<td>7723395672305685764</td>
</tr>
<tr>
<td>Jenny</td>
<td>4723372854036780875</td>
<td>4723372854036780875</td>
</tr>
<tr>
<td>Sury</td>
<td>116850462738740318</td>
<td>116850462738740318</td>
</tr>
</tbody>
</table>

Cassandra cluster with 4 nodes
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

Measuring the quality of hash function

- Hash function quality
  \[
  \frac{1}{n} \sum_{j=0}^{m} \left( \frac{h(j + \text{hash} + 1)}{2n} \right) \text{mod}(n + 2m - 1)
  \]
- Where, \( h \) is the number of items in \( j \)-th slot.
- \( n \) is the total number of items
- \( m \) is the number of slots

Testing with 42 Million keys

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 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.

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

- 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 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.

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
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 16 vnodes per node and the new machines are twice as powerful, give them 32 vnodes each and the cluster remains balanced even during transition
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)

**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

**ACK2: GossipDigestAct2Message**

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

**AppState Reconciliation**

- Generation
- Heartbeat
- AppState based on comparing version
Reconciliation example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gen:1234</td>
<td>Gen:2345</td>
<td>Gen:5555</td>
<td>Gen:2222</td>
</tr>
<tr>
<td></td>
<td>Hb: 994</td>
<td>Hb: 10</td>
<td>Hb: 1111</td>
<td>Hb: 4444</td>
</tr>
<tr>
<td>Status: normal</td>
<td>Status: bootstrap 1</td>
<td>Status: normal 2</td>
<td>Status: normal 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gen:1236</td>
<td>Gen:2345</td>
<td>Gen:5555</td>
<td>Gen:3333</td>
</tr>
<tr>
<td>Hb: 990</td>
<td>Hb: 17</td>
<td>Hb: 11</td>
<td>Hb: 11</td>
<td></td>
</tr>
<tr>
<td>Status: normal</td>
<td>Status: normal 2</td>
<td>Status: normal 3</td>
<td>Status: normal 3</td>
<td></td>
</tr>
</tbody>
</table>

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

Decommission
- 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**

- 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
  - Lock-up a key
- To narrow down the lookup process
  - A bloom filter is stored in each data file and memory