PART B. GEAR SESSIONS
SESSION 1: PETA-SCALE STORAGE SYSTEMS

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

FAQs

• Quiz #3
  • 2/28 ~ 3/1
  • GEAR Session 1
  • 10 questions
  • 30 minutes
  • Answers will be available at 9PM 3/2

http://www.cs.colostate.edu/~cs535
Topics of Today's Class

- GEAR Session I. Peta Scale Storage Systems
- Lecture 3.
  - Cassandra

GEAR Session 1. Peta-scale Storage Systems
GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system
Column Family NoSQL Storage system:
Introduction to Apache Cassandra

This material is built based on,


- Datastax Documentation: Apache Cassandra

- Now, Apache’s open source project,
  - http://cassandra.apache.org
**CAP Theorem**

- Eric Brewer
  - it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees

- **Consistency**: Every read receives the most recent write or an error
- **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
- **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

**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
GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system

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
  - Column family is an ordered collection of rows

---

Column family vs. a table of relational databases

<table>
<thead>
<tr>
<th>Relational Table</th>
<th>Cassandra column Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>A schema in a relational model is fixed. Once we define certain columns for a table, while inserting data, in every row all the columns must be filled at least with a null value</td>
<td>In Cassandra, although the column families are defined, the columns are not. You can freely add any column to any column family at any time</td>
</tr>
<tr>
<td>Relational tables define only columns and the user fills in the table with values.</td>
<td>In Cassandra, a table contains columns, or can be defined as a super column family</td>
</tr>
</tbody>
</table>

Column: basic data structure of Cassandra with three values, namely key or column name, value, and a time stamp (e.g. name: byte[], value:byte[], clock:clock[])
Super Column: it is also a key-value pair. (e.g. name:byte[], cols: map<byte[],column>)
Super column family

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

API

- `insert(table, key, rowMutation)`
- `get(table, key, columnName)`
- `delete(table, key, columnName)`
Comparison between RDMBS and Cassandra

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDBMS deals with structured data.</td>
<td>Cassandra deals with unstructured data.</td>
</tr>
<tr>
<td>It has a fixed schema.</td>
<td>Cassandra has a flexible schema.</td>
</tr>
<tr>
<td>In RDBMS, a table is an array of arrays. (ROW x COLUMN)</td>
<td>In Cassandra, a table is a list of “nested key-value pairs”. (ROW x COLUMN key x COLUMN value)</td>
</tr>
<tr>
<td>Database is the outermost container that contains data corresponding to an application.</td>
<td>Keyspace is the outermost container that contains data corresponding to an application.</td>
</tr>
<tr>
<td>Tables are the entities of a database.</td>
<td>Tables or column families are the entity of a keyspace.</td>
</tr>
<tr>
<td>Row is an individual record in RDBMS.</td>
<td>Row is a unit of replication in Cassandra.</td>
</tr>
<tr>
<td>Column represents the attributes of a relation.</td>
<td>Column is a unit of storage in Cassandra.</td>
</tr>
<tr>
<td>RDBMS supports the concepts of foreign keys, joins.</td>
<td>Relationships are represented using collections.</td>
</tr>
</tbody>
</table>

Here, we have a data model. What do we have to consider?

- We will use the “key” to retrieve data

- Spread data evenly (as even as possible) around the cluster
  - Rows are spread around the cluster based on a hash of the partition key, which is the first element of the PRIMARY KEY

- Cluster should be incrementally scalable
  - Scale-out solution
GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system

Apache Cassandra
Data 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

Consistent hash function assigns each node and key an m-bit identifier using a hashing function.

Identifier circle with m = 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.

Machine B is the successor node of key 1. successor (1) = 1

Key 2 will be stored in machine C successor(2) = 5

Key 3 will be stored in machine C successor(3) = 5

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

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

If machine C leaves circle, Successor(5) will point to A
If machine N joins circle, 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?
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

GEAR Session 1. peta-scale storage systems
Lecture 3. Distributed No-SQL data storage system
Apache Cassandra
Data Partitioning: CHORD
This material is built based on


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 (zero-hop DHT)
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. $s = \text{successor}(n+2^{i-1})$, where $1 \leq i \leq m$ (and all arithmetic is modulo $2^m$)

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

- $\text{finger}[i].\text{start} = (n+2^{i-1}) \mod 2^m$, $1 \leq i \leq m$
- $\text{finger}[i].\text{interval} = [\text{finger}[i].\text{start}, \text{finger}[i+1].\text{start})$, if $i=m$, $[\text{finger}[i].\text{start}, \text{finger}[1].\text{start} - 1)$
- $\text{finger}[i].\text{node} = \text{first node} \geq n.\text{finger}[i].\text{start}$
- $\text{successor} = \text{the next node on the identifier circle}$
- $\text{predecessor} = \text{the previous node on the identifier circle}$
• **Finger table**
  - The Chord identifier
  - The IP address of the relevant node

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

- 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

1. Go clockwise
2. Never overshoot

- First, check the data is stored in $n$
  - If it is, return the data
- Otherwise,
  - $n$ 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!
0. Request comes into node 3 to find the successor of identifier 1.
1. Node 3 wants to find the successor of identifier 1
2. Identifier 1 belongs to \([7,3)\)
3. Check \(\text{succ}\): 0
4. Node 3 asks node 0 to find successor of 1
5. Successor of 1 is 1

0. Request comes into node 1 to find the successor of id 4.
1. Node 3 wants to find the successor of identifier 4
2. Identifier 4 belongs to \([3,5)\)
3. Check \(\text{succ}\): 3
4. Node 1 asks node 3 to find successor of 4
5. Successor of 4 is 0
Lookup process: example 2

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

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

Theorem 2.

- With high probability (or under standard hardness assumptions), the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$

- Proof

Suppose that node $n$ tries to resolve a query for the successor 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 $2^{i-1}$.
Proof continued

Let \( f \) and \( p \) be both in \( n \)'s \( i \)th finger interval, and the distance between them is at most \( 2^{i-1} \). This means \( f \) is closer to \( p \) than to \( n \) or equivalently:

- Distance from \( f \) to \( p \) is at most half of the distance from \( n \) to \( p \)
- If the distance between the node handling the query and the predecessor \( p \) halves in each step, and is at most \( 2^m \)
- Within \( m \) steps the distance will be 1 (you have arrived at \( p \))

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

After \( \log N \) forwardings, the distance between the current query node and the key \( k \) will be reduced at most \( 2^m/N \)

- The average lookup time is \( \frac{1}{2} \log N \)

Requirements in node Joins

- In a dynamic network, nodes can join (and leave) at any time

1. Each node's successor is correctly maintained
2. For every key \( k \), node \( \text{successor}(k) \) is responsible for \( k \)
Tasks to perform node join

1. Initialize the predecessor and fingers of node \( n \)

2. Update the fingers and predecessors of existing nodes to reflect the addition of \( n \)

3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node \( n \) is now responsible for

---

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

```java
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);
```

- Create the finger-table at the new node \( n \) by asking the node \( n' \)
Join 5 (After `init_finger_table(n')`)

Join 5 (After `update_others()`)

http://www.cs.colostate.edu/~cs535
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 (see the code in the step 1-(1/2))
- 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
  - It shares some nodes

Updating fingers of existing nodes

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

```python
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 meet these two conditions
  • 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 $O(\log N)$ on the average

---

Transferring keys

• Move responsibility for all the keys for which node $n$ is now the successor
  • It involves moving the data associated with each key to the new node

• Node $n$ can become the successor only for keys that were previously the responsibility of the node immediately following $n$
  • $n$ only needs to contact that one node to transfer responsibility for all relevant keys
Example

- If you have following data,

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
<th>Car</th>
<th>Gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>36</td>
<td>Camaro</td>
<td>M</td>
</tr>
<tr>
<td>Carol</td>
<td>37</td>
<td>BMW</td>
<td>F</td>
</tr>
<tr>
<td>Jonny</td>
<td>10</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>Suzy</td>
<td>9</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

- Cassandra assigns a hash value to each partition key

<table>
<thead>
<tr>
<th>Partition Key</th>
<th>Mumer 3 Hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>-224562676723223822</td>
</tr>
<tr>
<td>Carol</td>
<td>7723358927203680754</td>
</tr>
<tr>
<td>Jonny</td>
<td>-6723372854036780875</td>
</tr>
<tr>
<td>Suzy</td>
<td>1168604627387940318</td>
</tr>
</tbody>
</table>

Cassandra cluster with 4 nodes
GEAR Session 1. peta-scale storage systems

Lecture 3. Distributed No-SQL data storage system

**Apache Cassandra**

Data 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
1. **Murmur3Partitioner**

- Murmur hash is a non-cryptographic hash function
  - Created by Austin Appleby in 2008
  - Multiply (MU) and Rotate (R)

- Current version Murmur3 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\(^1\)
  \[
  \sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)}
  \]
  - 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

---

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”
  - $P(\text{Output bit } i \text{ changes} \mid \text{Input bit } j \text{ changes}) = 0.5$
    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.

2. RandomPartitioner

- RandomPartitioner was the default partitioner prior to Cassandra 2.1
  - Uses MD5
  - 0 to $2^{127} - 1$
3. 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

**GEAR Session 1. peta-scale storage systems**
Lecture 3. Distributed No-SQL data storage system

**Apache Cassandra**

Data Replication

http://www.cs.colostate.edu/~cs535  
Spring 2020 Colorado State University, page 30
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

---

1. **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
2. 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.

2. 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.
3. **NetworkTopologyStrategy**

   - **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.

---

**Questions?**