CS535 BIG DATA

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

2. SERVING LAYER: CASSANDRA

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

FAQs

- Assignment 2 has been posted
  - URL for zookeeper has been updated
  - Use the same machines and ports assignment
  - Use only English message

Today's topics

- Document-oriented storage system - continued
  - Apache Cassandra

ByteOrderPartitioner

- This partitioner orders rows lexically by key bytes
- The ordered partitioner allows ordered scans by primary key
  - If your application has user names as the partition key, you can scan rows for users whose names fall between Jake and Joe
- Disadvantage of this partitioner
  - Difficult load balancing
  - Sequential writes can cause hot spots
  - Uneven load balancing for multiple tables

Geohashes (2-dimensional geospatial data to DHT)

- Used in Galileo, MongoDB
  - Proximity search
- Subdivides the globe into a hierarchy represented by strings
- (40.573879, -105.084282)→9XJQBDJK4XUT
- Longer strings represent more precise coordinates
- Strings with similar prefixes are *geographically close*

Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

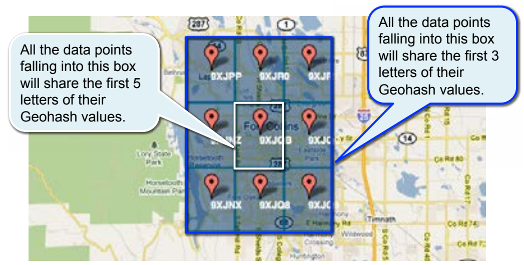
Example: Geohash of this Room

Example: Geohash of this Room

Example: Geohash of this Room

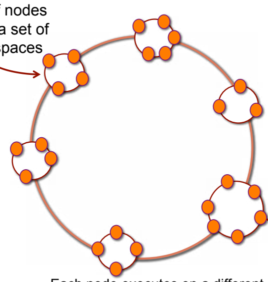
Adjacency with the geohash algorithm

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



Nodes are organized in a ring of rings

A group of nodes manages a set of geohash spaces



Individual nodes are responsible for managing the feature space

Each node executes on a different physical machine

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

11/3/2016 CS535 Big Data, Fall 2016 W11.B.18

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.

11/3/2016 CS535 Big Data, Fall 2016 W11.B.19

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.

11/3/2016 CS535 Big Data, Fall 2016 W11.B.20

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.

11/3/2016 CS535 Big Data, Fall 2016 W11.B.21

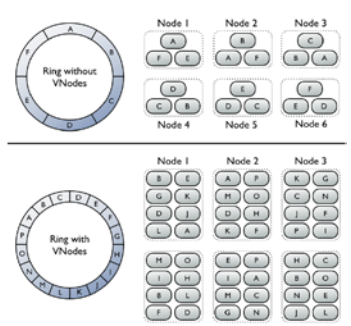
Apache Cassandra
Virtual Node

11/3/2016 CS535 Big Data, Fall 2016 W11.B.22

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

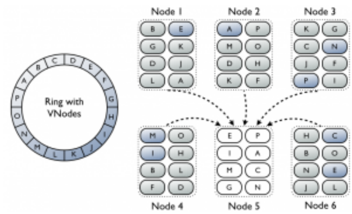
11/3/2016 CS535 Big Data, Fall 2016 W11.B.23



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



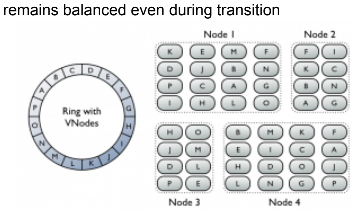
- If vnodes are spread throughout the entire cluster
  - Data transfers will be distributed on more machines

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

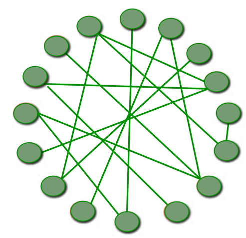
What is gossip?

- 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
  - Heartbeat
  - 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

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)

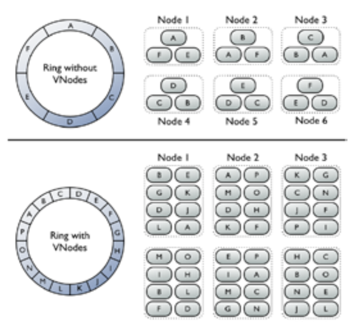
- Asymmetrical replication groupings
  - For example, you can maintain 4 replicas
    - Three replicas in one data center to serve real-time application requests
    - A single replica elsewhere for running analytics.

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



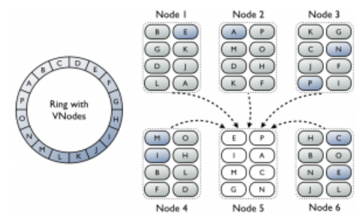
http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2

11/3/2016 CS535 Big Data, Fall 2016 W11.B.78

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

11/3/2016 CS535 Big Data, Fall 2016 W11.B.79



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

11/3/2016 CS535 Big Data, Fall 2016 W11.B.80

Restoring a new disk with vnodes

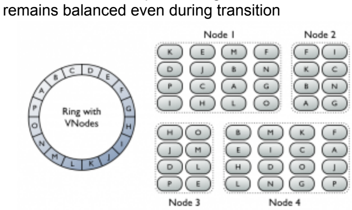
- Process of restoring a disk
  - Validating all the data and generating a Merkle tree
    - This might take an hour
  - Streaming when the actual data that is needed is sent
    - This phase takes a few minutes

- 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

11/3/2016 CS535 Big Data, Fall 2016 W11.B.81

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



11/3/2016 CS535 Big Data, Fall 2016 W11.B.82

Apache Cassandra

Gossip (Internode communications)

11/3/2016 CS535 Big Data, Fall 2016 W11.B.83

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

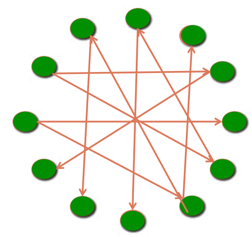
What is gossip?

- 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
```

Compares

```
# 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
```

Generates ACT

```
# 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

Version number

	A	B	After reconciliation
A	gen:1234 Hb: 994 Status: normal {4}	gen:1234 Hb: 990 Status: normal {4}	Hearbeats are different. B will update its Hb to A's status
B	Gen:2345 Hb: 10 Status: bootstrap {1}	Gen:2345 Hb:17 Status: normal {2}	The version numbers are different A will update the state to B's status
C	Gen:5555 Hb: 1111 Status: normal {5}		B will know that C exists in this cluster
D	Gen:2222 Hb: 4444 status: normal {3}	Gen:3333 Hb: 11 Status: normal {3}	Node D has been re-started A will take all of the metadata in B

4 nodes cluster A, B, C and D

Messaging summary

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

CS535 BIG DATA

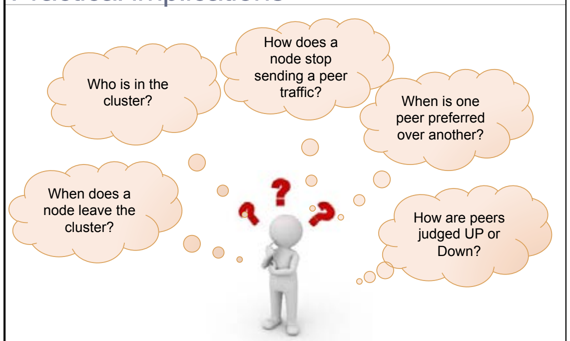
Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

Today's topics

- Document-oriented storage system - continued
  - Apache Cassandra
- Graph storage and processing models

Practical implications



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 Detector

- 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
  - Terminate sockets

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)
  - The updates are capped at a maximum of 10,000 per scoring interval
  - Scores reset every 10 minutes (default)
  - Uses statistically significant random sampling

http://www.datastax.com/dev/blog/dynamic-snitching-in-cassandra-past-present-and-future

How do nodes leave?

Decommission

- STATUS = LEAVING
- Stream data
- Stream hints
- STATUS = LEFT, expiryTime (hard coded as 3 days)

Remove node

- STATUS = REMOVING
- Rebalance cluster
  - Notify coordinator
- Delete hint
- STATUS = REMOVED, expiryTime (hard coded as 3 days)

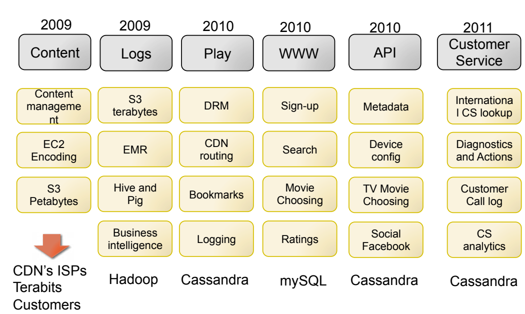
Replace node

- 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

Running Netflix on Cassandra in the Cloud



Cassandra Service Pattern

Production deployment

- Over 50 Cassandra Clusters
- Over 500 nodes
- Over 30 TB of daily backups
- Biggest cluster 72 nodes
- 1 cluster over 250 K Writes/s

High Availability

- Cassandra stores 3 local copies, 1 per zone
  - Synchronous access, durable, highly available
  - Read/Write One fastest, use for fire and forget
  - Read/Write Quorum 2 of 3, user for read-after-write
- AWS Availability Zones
  - Separate buildings
  - Separate power etc.
  - Fairly close together

Graph Storage systems

Graphs around us?

- Social graph
  - People you may know
- Ratings graph
  - Products you might like
- Social+ratings graph
  - Movies you should watch and the friends you should watch them with

Graph-like structure in RDBMS

- It can be stored and also handle queries
  - A single type of relationship
  - "who is my manager"
- Adding another relationship to the mix?
  - Requires schema changes and data movement
  - e.g. add preferences to lunch menu for each of the employees who have lunch at cafeteria
    - Add new table or column for new menu

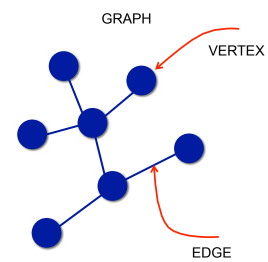
EmployeeID	Name	ManagerID
0001	ABC	5545
0002	BCD	5567
0003	CDE	6677

EmployeeID	Likes_caf_menu_A
0001	1
0002	0
0003	1

Traversal in Graph data

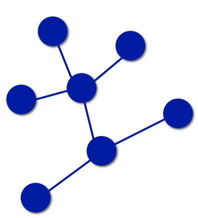
- A query on the graph is known as traversing the graph
- Traversing can be changed without having to change the nodes or edges
- Data can be traversed as it needs to be
- Traversing the joins or relationship is very fast
  - Relationship between nodes is not calculated at query time
  - It is actually persisted as a relationship

Graphs

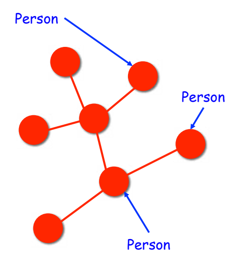


G=(V,E) where,
V is a set of vertices
E is a set of edges

Homogeneous set of vertices connected by a homogeneous set of edges

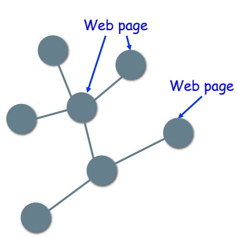


Homogeneous set of vertices connected by a homogeneous set of edges



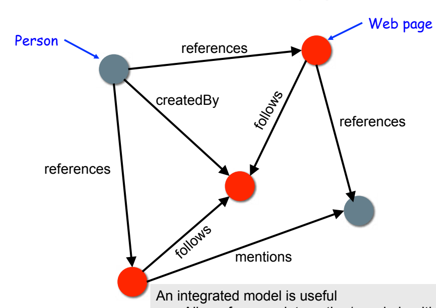
People follows relationships

Homogeneous set of vertices connected by a homogeneous set of edges



Web pages contain links

Multi-relational property graph



An integrated model is useful
- Allows for more interesting/novel algorithms
- Allows for a universal model of things and their relationships

The property graph

- G=(V,E,λ)

- Directed, attributed, edge-labeled graph
- Multi-relational graph with key/value pairs on the elements

Graph query language

- Gremlin
  - A graph traversal language
  - Expresses complex graph traversals
  - Supports traversal patterns
    - Backtrack pattern, Path pattern, Loop pattern, Split/Merge Pattern, etc.

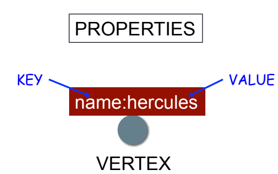
```
gremlin>
g.V('name','hercules').out('father').out('father').name
➔cronos
```

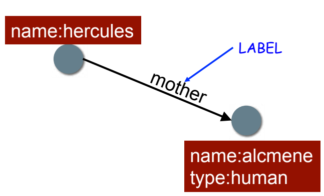
  - g: for the current graph
  - v('name','hercules'): get all vertices with name property "hercules"
  - out('father'): traverse outgoing father edge's from Hercules
  - out('father'): traverse outgoing father edge's from Hercules' father's vertex
  - name: get the name property of the "hercules" vertex's grandfather

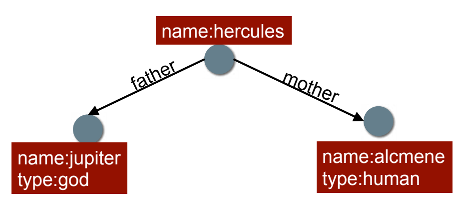
Traversing with Functions

- Adding vertex and edge

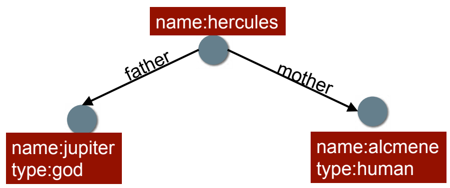
```
gremlin> theseus = g.addVertex([name:'theseus',type:'human'])
==>v[302]
gremlin> cerberus = g.V('name','cerberus').next()
==>v[48]
gremlin> g.addEdge(theseus,cerberus,'battled')
==>e[151200009:302:36028797018964038][302-battled->48]
```

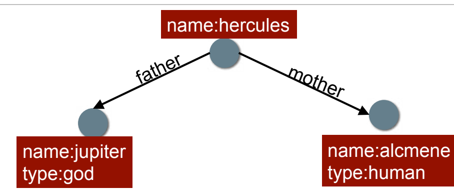




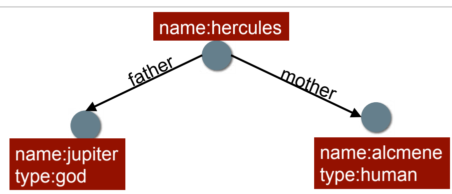
What is Hercules' type?



* This example is from http://thinkaurelius.github.io/titan/

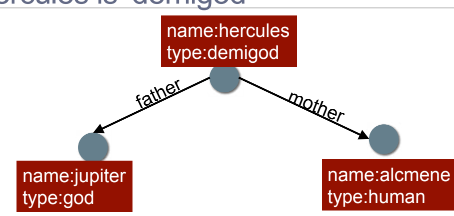



```
gremlin> hercules.out('mother','father')
==>v[1]
==>v[2]
```

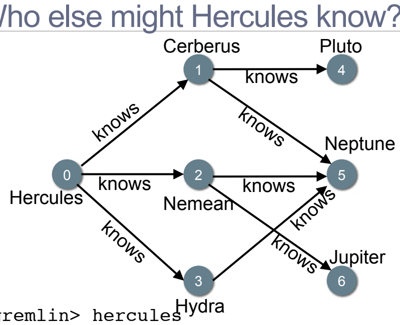

```
gremlin> hercules.out('mother','father').type
==>human
==>god
```


Hercules is 'demigod'

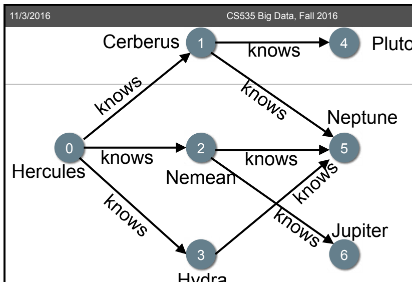



```
gremlin> hercules.type='demigod'
==>demigod
```

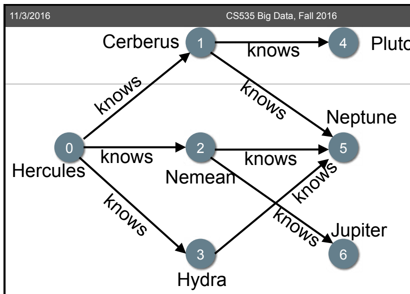

Who else might Hercules know?



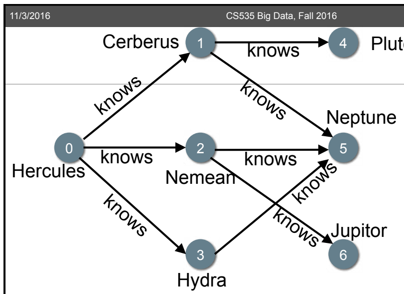

```
gremlin> hercules
==>v[0]
```


```
gremlin> hercules.out('knows')
==>v[1]
==>v[2]
==>v[3]
```

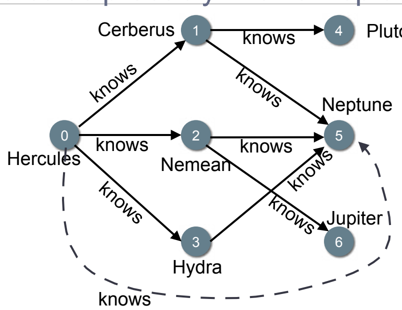



```
gremlin> hercules.out('knows').out('knows')
==>v[4]
==>v[5]
==>v[5]
==>v[6]
==>v[5]
```

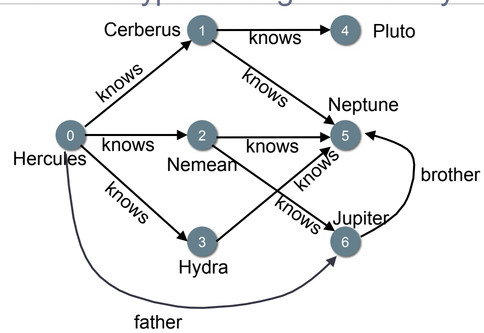

```
gremlin> hercules.out('knows').out('knows').groupCout.cap
==>v[4]=1
==>v[5]=3
==>v[6]=1
```

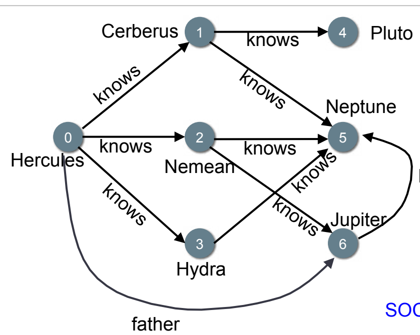

Hercules probably knows Neptune

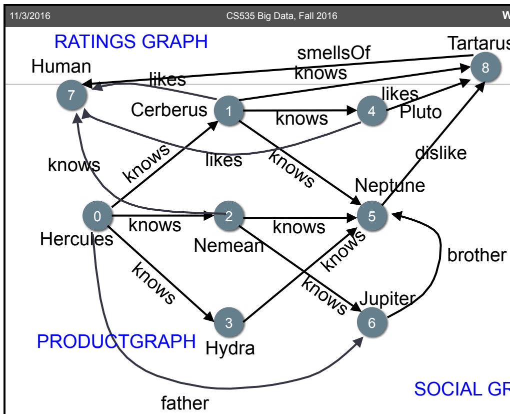


This is a "textbook" style graph!

What if the types of edges are analyzed?



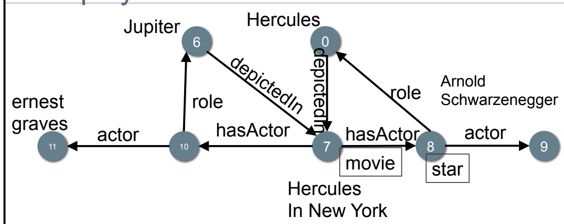




Finding paths

- Movie graph
  - How is this person related to this film?
- Book graph
  - Which authors of this book also wrote a New York Times Bestseller?
- Movie+Book graph
  - Which movies are based on a book by a New York Times Bestseller?

Who played **Hercules** in **What** movie?




```
gremlin> hercules.out('depictedIn').as('movie').
out('hasActor').out('role').retain(hercules).nack(2).
Out('actor').as('star').select{it.name}
==>[movie:hercules in new york, star"Arnold sch
Warzenegger]
```

Some part of the graph is removed for the space.

Queries generates social influence

- Who are the most influential people in
  - Java, mathematics, art, surreal art, politics?
- Which region of the social graph will propagate this advertisement the furthest?
- Which 3 experts should review this submitted article?
- Which people should I talk to at the upcoming conference and what topics should I talk to them about?

Pattern identification

- This connectivity pattern is a sign of a financial fraud.
  - Transaction graph
- Healthy discourse is typified by a discussion board with a branch factor in this range and a concept clique score in this range
  - Discussion graph

Graph computing engines: Memory-based graph framework

- Graph size is constrained by local machine's RAM
- Rich graph algorithm and visualization packages
- Applications
  - iGraph
    - http://igraph.sourceforge.net
  - NetworkX
    - http://hetworkx.lanl.gov
  - JUNG
    - http://jung.sourceforge.net

Graph computing engines: Disk-based graphs

- Graph size is constrained by the local disk
- Optimized for local graph algorithms
- Oriented towards property graphs
  - Graph database
  - Neo4J
    - http://neo4j.org
  - InfiniteGraph
    - http://objectivity.com
  - OrientDB
    - http://orientdb.org
  - DEX
    - http://www.sparsity-technologies.com/dex

Graph computing engines: Cluster-based graphs

- Graph size is constrained by cluster's total RAM
- Optimized for global graph algorithm
- Bulk synchronous parallel processing
  - Pregel/Hama
    - http://incubator.apache.org/hama
  - Giraph
    - http://incubator.apache.org/giraph
  - GoldenOrb
    - http://goldenorbos.org

Titan: a Distributed Graph storage

Demands on Titan

- Processes graphs that have a 100+ billion edge scale with thousands of concurrent transactions
- Local graph traversals and batch graph processing
- Extremely scalable in the presence of:
  - Graph volume
  - Number of concurrent access to the graph

Key features

- Scalable graph
- Scalable access
- Real-time local traversals
- Global batch processing via Hadoop
- Open-source with Apache2 license

Options for the backend storage system

- Hbase
- Cassandra
- Oracle's BerkeleyDB

Inherited features from Cassandra

- High availability
- No single point of failure
- Replication scheme
- Caching
- Elastic scalability

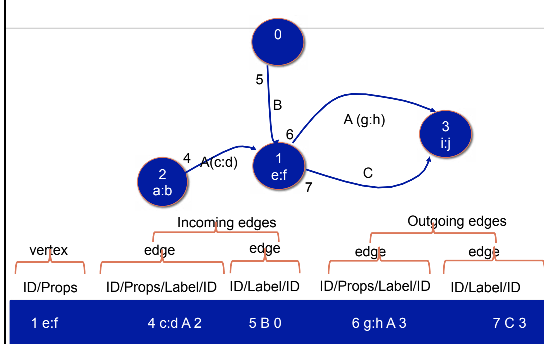
Distinctive challenges

- Storage model
- Graph Partitioning
- Edge compression
- Vertex-centric indices

Titan
Storage Model

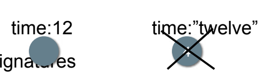
Titan Storage Model

- Adjacency list in one column family
- Row key= vertex id
- Each property and edge in one column
  - Denormalized (i.e. stored twice)
- Direction and label/key as column prefix
  - Use slice predicate for quick retrieval

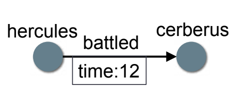


Type check (1/2)

- Type check for each of the data types
  - `time:"twelve"` is not acceptable

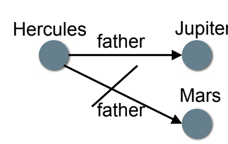


- Edge label signatures
  - Create association using signature of particular key
  - `TitanLabel battled=g.makeType().name("battled").signature(timeKey)`



Type check (2/2)

- Functional Declarations
  - `TitanLabel father = g.makeType().name("father").functional()`



Locking system

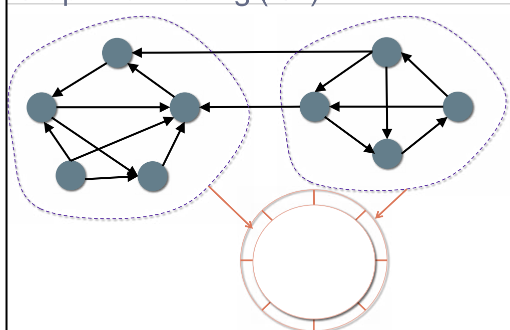
- Ensures consistency over non-consistent storage backends
1. Acquire lock at the end of the transaction
   - Locking mechanism depends on storage layer consistency guarantees
2. Verify original read
3. Fail transaction if any precondition is violated

GRAPH PARTITIONING

Graph Partitioning (1/2)

- Maximize locality of vertices
  - Co-locate vertices
- Titan maintains multiple ID rings
- `OrderedPartitioner` (from Cassandra) in storage backend
- Dynamically determines good partition and allocates corresponding IDs
- Graph Partitioning is NP complete problem
- Ongoing effort

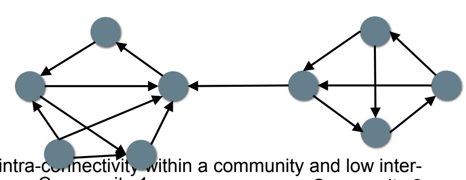
Graph Partitioning (2/2)



EDGE COMPRESSION

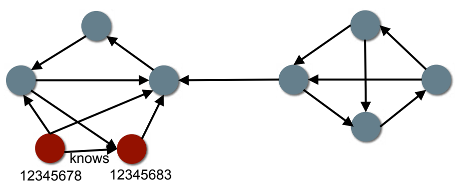
Small-world networks

- Natural graphs have a small world, community/cluster property

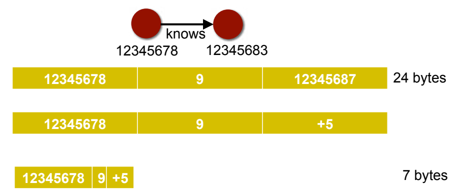


- High intra-connectivity within a community and low inter-connectivity between communities
- Watts,D.J., Strogatz, S. H., "Collective Dynamics of 'Small-World' Networks," Nature 393(6684), pp. 440-442, 1998

Edge compression (1/2)



Edge compression (2/2)



VERTEX-CENTRIC INDICES

Top 5 Twitters based on Followers

5) 34,952,307
- Taylor Swift

4) 37,849,134
- Barack Obama

3) 40,242,656
- Lady Gaga

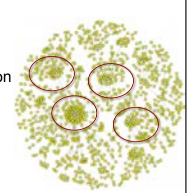
2) 44,951,137
- Katy Perry

1) 45,728,072
- Justin Bieber

http://twitaholic.com/top100/followers

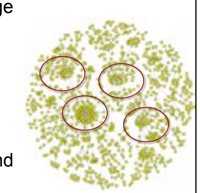
The super node problem (1/2)

- The super node problem
  - Natural, real-world graphs contain vertices of high degree
  - Even if rare, their degree ensures that they exist on many paths
  - Traversing a high degree vertex means touching numerous incident edges and potentially touching most of the graph in only a few steps



The super node problem (2/2)

- A "super node" only exists from the vantage point of classic "textbook style" graphs
- In the world of property graphs, intelligent disk-based filtering can interpret a "super node" as a more manageable low-degree vertex
- Vertex-centric querying utilizes B-Trees and sort orders for speedy lookup of incident edges with particular qualities

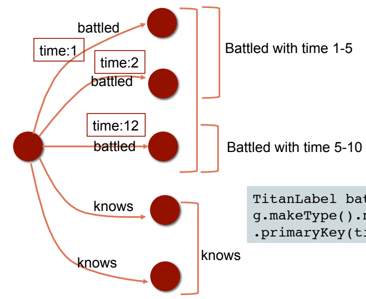


Vertex-centric Indices

- Sort and index the incident edges (adjacent vertices)
  - Per vertex by primary key
  - Based on the edge's labels and properties
- Enables efficient focused traversals
  - Only retrieve edges that matter
- Uses push down predicates for quick, index-driven retrieval

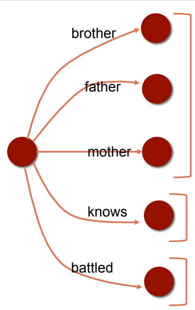
  - https://github.com/thinkaurelius/titan/wiki/Vertex-Centric-Indices

Vertex-centric indices




```
TitanLabel battled =
g.makeType().name("battled")
.primaryKey(time)
```


Vertex-centric indices




```
Vertex.query().group("family")..
```


Vertex-centric Indices

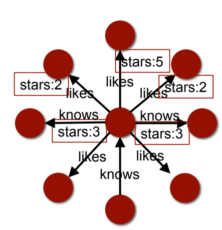
Incident edge size	No vertex-centric indices	Vertex-centric indices
1000	0.82 ms	0.50 ms
10000	7.06 ms	0.52 ms
100000	70.90 ms	0.46 ms
1000000	778.94 ms	0.46 ms

Pushdown predicates

- Vertex-centric indices enables pushdown predicates efficiently
- A vertex has following information about the incident edges
  - Labels
  - Properties
  - Directions
- Vertex-centric query provides query from the perspective of a vertex.
- Edges that meet the query criteria are selected from the underlying graph representation

Pushdown predicates

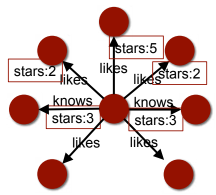
```
vertex.query()
```



8 edges

Pushdown predicates

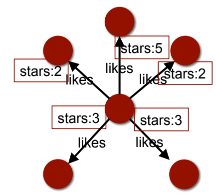
```
vertex.query().direction(out)
```



7 edges

Pushdown predicates

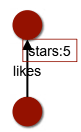
```
vertex.query().direction(OUT).labels("likes")
```



5 edges

Pushdown predicates

```
vertex.query().direction(OUT).labels("likes").ha
s("stars",5)
```



1 edges

Pushdown predicates

- Query Query.direction(Direction)
- Query Query.labels(String… labels)
- Query Query.has(String, Object, Compare)
- Query Query.has(String, Object)
- Query Query.range(String, Object, Object)

- Iterable<Vertex> Query.vertices()
- Iterable<Edge> Query.edges()

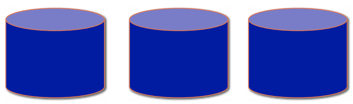
INTERACTING WITH GRAPH ANALYTICS

Faunus

- Hadoop-based Graph computing framework
- Graph analytics
- Breath-first traversals
- Global graph computations
- Batch big graph data

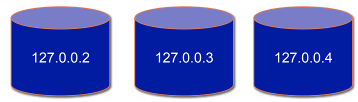
An Adjacency list + Cluster

0
1
2
3
4
5
6
7
8
9
10
11

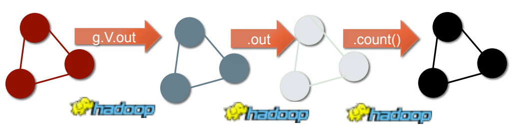


A Distributed adjacency list

0	4	8
1	5	9
2	6	10
3	7	11



Fanus workflow



Compressed HDFS Graphs
- Stored in sequence files
- Variable length encoding
- Prefix compression

Graph analytics with Titan

- Titan
  - Stores a massive-scale property graph, allowing real time traversals and updates

MapReduce
Analysis results
Back to Titan
Bulk load

- Faunus
  - Batch processing of large graphs with Hadoop
- Fulgora
  - Runs global graph algorithms on large, compressed in-memory graphs

Load

PREGEL: A SYSTEM FOR LARGE-SCALE GRAPH PROCESSING

This material is built based on,

- Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, Names C. Dehnert, Ilan Horn, Naty Leiser, Grzegorz Czajkowski, “Pregel: a system for large-scale graph processing”, Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135-146
- Apache’s Hama
  - Open source project inspired by Pregel
  - http://hama.apache.org

Graph analysis at Google?

- MapReduce tasks
  - Google's 80% of data analysis
    - Large-scale web search indexing
    - Clustering problems for Google News
    - Produce reports for popular queries (e.g. Google Trend)
    - Processing of satellite imagery data
    - Language model processing for statistical machine translations
    - Large-scale machine learning problems
    - Back-up/restore
- The other 20%?

Graph analysis at Google?

- Large graph analysis
  - Graph algorithms
    - PageRank
    - Shortest path
    - Connected components
    - Clustering techniques
  - Graph data
    - Web graph
    - Transportation routes
    - Citation relationships
    - Social networks

MapReduce is NOT great for graph processing

- Many iterations are needed for parallel graph processing
- Materializations of intermediate results at every MapReduce iteration causes performance bottleneck

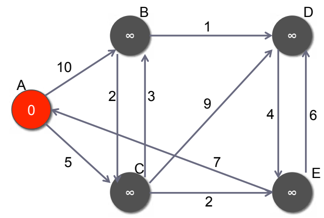
Single Source Shortest Path (SSSP)

- Find shortest path from a source node to all target nodes
- If you have a single processor machine?
  - Dijkstra's algorithm

Finding SSSP using Dijkstra's Algorithm

11/3/2016 CS535 Big Data, Fall 2016 W11.B.204

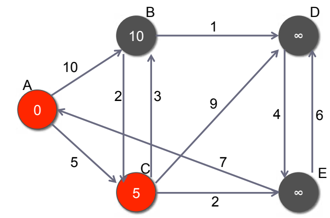
Dijkstra's Algorithm



A(Source)	B	C	D	E
0	∞	∞	∞	∞

11/3/2016 CS535 Big Data, Fall 2016 W11.B.205

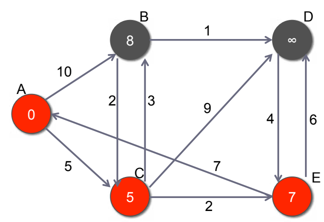
Dijkstra's Algorithm



A(Source)	B	C	D	E
0	∞	∞	∞	∞
0	10	5/A	∞	∞

11/3/2016 CS535 Big Data, Fall 2016 W11.B.206

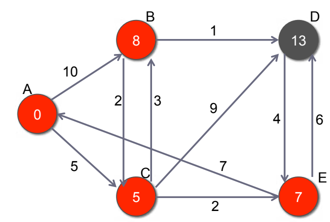
Dijkstra's Algorithm



A(Source)	B	C	D	E
0	∞	∞	∞	∞
0	10	5/A	∞	∞
0	8	5/A	14	7/C

11/3/2016 CS535 Big Data, Fall 2016 W11.B.207

Dijkstra's Algorithm



A(Source)	B	C	D	E
0	∞	∞	∞	∞
0	10	5/A	∞	∞
0	8	5/A	14	7/C
0	8/C	5/A	13	7/C

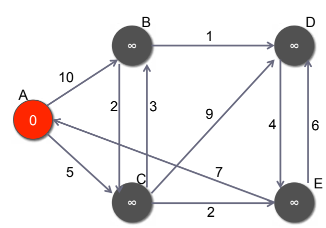
Finding SSSP using MapReduce

11/3/2016 CS535 Big Data, Fall 2016 W11.B.209

Using MapReduce

	A	B	C	D	E
A		10	5		
B			2	1	
C		3		9	2
D					4
E	7			6	

Adjacency Matrix



Adjacency list
A: (B,10, (C, 5)
B: (C,2), (D,1)
C: (B,3), (D,9)
D: (E,4)
E: (A,7), (D,6)

Using MapReduce

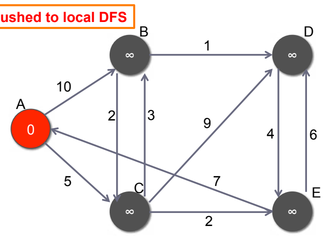
Map input: <nodeID, :dist, adj list>>
<A, <0, <(B,10), (C, 5)>>>
<B, <inf, <(C,2), (D,1)>>>
<C, <inf, <(B,3), (D,9)>>>
<D, <inf, <(E,4)>>>
<E, <inf, <(A,7), (D,6)>>>

Flushed to local DFS

Map output:<dest node ID, dist>
<B,10><C,5>
<C, inf><D, inf>
<B, inf><D, inf><E. inf>
<E, inf>
<A, inf><D, inf>

<A, <0, <(B,10), (C, 5)>>>
<B, <inf, <(C,2), (D,1)>>>
<C, <inf, <(B,3), (D,9)>>>
<D, <inf, <(E,4)>>>
<E, <inf, <(A,7), (D,6)>>>

Flushed to local DFS



Using MapReduce

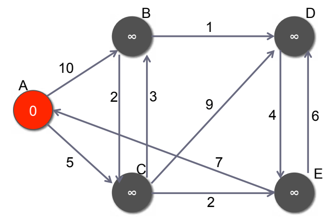
Reduce input: <nodeID, dist>
<A, <0, <(B,10), (C, 5)>>>
<A, inf>

<B, <inf, <(C,2), (D,1)>>>
<B,10><B, inf>

<C, <inf, <(B,3), (D,9)>>>
<C,5><C, inf>

<D, <inf, <(E,4)>>>
<D, inf><D, inf><D, inf>

<E, <inf, <(A,7), (D,6)>>>
<E. inf><E. inf>



Using MapReduce

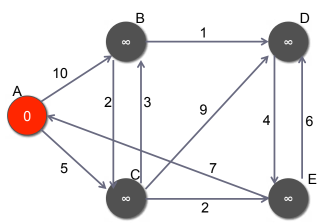
Reduce input: <nodeID, dist>
<A, <0, <(B,10), (C, 5)>>>
~~<A, inf>~~

<B, <~~inf~~, <(C,2), (D,1)>>>
<B,10>~~<B, inf>~~

<C, <~~inf,~~ <(B,3), (D,9)>>>
<C,5>~~<C, inf>~~

<D, <~~inf~~, <(E,4)>>>
~~<D, inf><D, inf><D, inf>~~

<E, <~~inf~~, <(A,7), (D,6)>>>
~~<E. inf><E. inf>~~



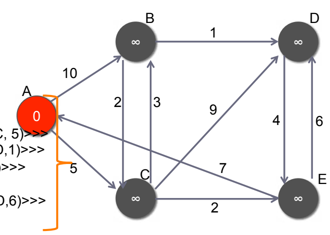
Using MapReduce

Reduce input: <nodeID, <dist,adj list>>
= Map input for next iteration

<A, <0, <(B,10), (C, 5)>>>
<B, <10, <(C,2), (D,1)>>>
<C, <5, <(B,3), (D,9)>>>
<D, <inf, <(E,4)>>>
<E, <inf, <(A,7), (D,6)>>>

Map output: <dest node ID, dist>
<B,10><C,5> <A, <0, <(B,10), (C, 5)>>>
<C,12><D,11> <B, <10, <(C,2), (D,1)>>>
<B,8><D,14><C, <5, <(B,3), (D,9)>>>
<E. inf> <D, <inf, <(E,4)>>>
<A, inf><C,inf> <E, <inf, <(A,7), (D,6)>>>

Flushed to local DFS



Using MapReduce

Reduce input: <nodeID, <dist,adj list>>
= Map input for next iteration

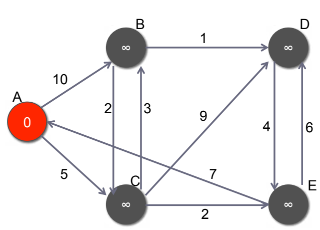
<A, <0, <(B,10), (C, 5)>>>
~~<A,inf>~~

<B, <~~10~~, <(C,2), (D,1)>>>
~~<B,10>~~<B, 8>

<C, <5, <(B,3), (D,9)>>>
<C,5>~~<C,12>~~

<D, <~~inf,~~ <(E,4)>>>
<D,11>~~<D,14><D,inf>~~

<E, <~~inf,~~ <(A,7), (D,6)>>>
~~<E, inf>~~<E,7>



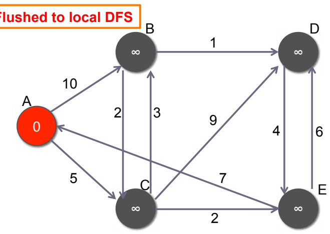
Using MapReduce

Reduce input: <nodeID, <dist,adj list>>
= Map input for next iteration

Flushed to local DFS

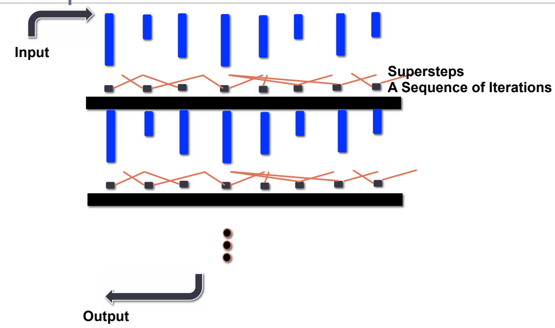
<A, <0, <(B,10), (C, 5)>>>
<B, <8, <(C,2), (D,1)>>>
<C, <5, <(B,3), (D,9)>>>
<D, <11, <(E,4)>>>
<E, <7,<(A,7), (D,6)>>>

Keep going..



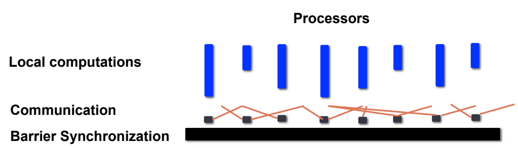
Pregel's Computational Model

Computational Model



Computational Model

Inspired by Valiant's Bulk Synchronous Parallel model (1990)



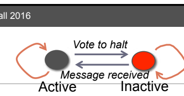
Computation Model (1/2)

- Superstep: the vertices compute in parallel
  - Each vertex
    - Receives messages from the previous superstep
    - Executes the same user-defined function
    - Sends messages to other vertices
    - Mutates the topology of the graph if need be
    - Votes to halt if it has no further work to do
  - When to terminate?
    - All vertices are simultaneously inactive
    - There are no messages in transit

Computation Model (2/2)

- Input to the Pregel computation
  - A directed graph
    - Vertex
      - String vertex ID
      - Associated user defined value
    - Edge
      - Associated with their source vertices
      - User defined value and a target vertex ID
- Computation in the vertex
  - Executes the same user-defined function
  - Modifies the state
    - Sometimes changes the outgoing edges
  - Receive/send message
  - Mutate topology
  - There is no computation associated with the edges

Vertex State Machine



- In superstep 0,
  - Every vertex is in the active state
  - All active vertices participate in the computation of any given superstep
- A vertex deactivates itself by voting to halt
  - The vertex has no further work to do unless triggered externally
  - Pregel will not process that vertex in subsequent supersteps
- Once a vertex is re-activated
  - It must explicitly deactivate itself again

Output of a Pregel program

- Set of values explicitly output by the vertices
  - Often a directed graph isomorphic to the input
  - E.g. clustering algorithm
    - Creates small set of disconnected vertices selected from a large graph
  - E.g. graph mining algorithm
    - Generates aggregated statistics mined from the graph

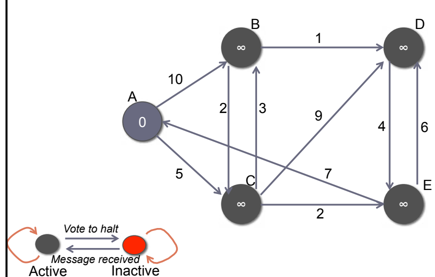
Message Passing (1/2)

- Vertices communicate directly with one another by sending messages
  - Message value
  - Name of the destination vertex
- A vertex can send any number of messages in a superstep
- There is no guaranteed order of messages in the iterator. However,
  - Message is delivered reliably
  - There will be no duplicate

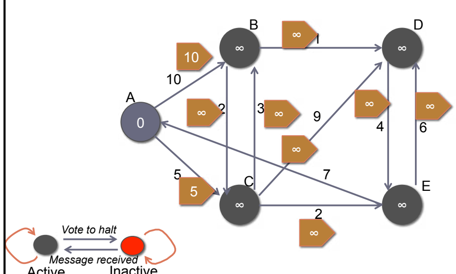
Message Passing (2/2)

- Common usage pattern
  - A vertex V to iterate over its outgoing edges and sending a message to the destination vertex of each edge
- Destination vertex need not be a neighbor of V
  - E.g.: A vertex can learn the identifier of a non-neighbor from a message received earlier
  - E.g.: implicitly vertex info is distributed
- If destination does not exist, user-defined handler will be executed.
  - Create the missing vertex or remove the dangling edge

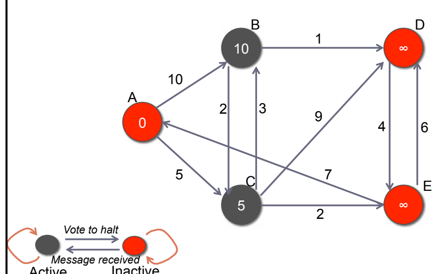
SSSP using parallel BFS in Pregel



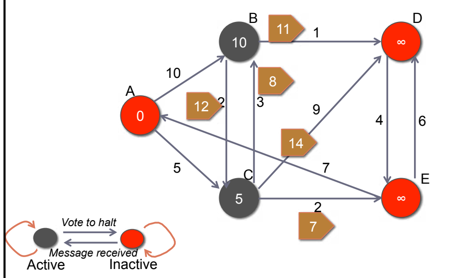
SSSP using parallel BFS in Pregel



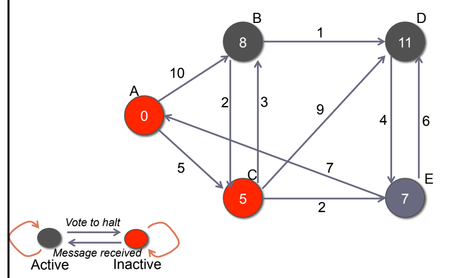
SSSP using parallel BFS in Pregel



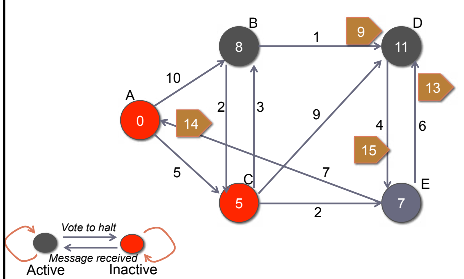
SSSP using parallel BFS in Pregel



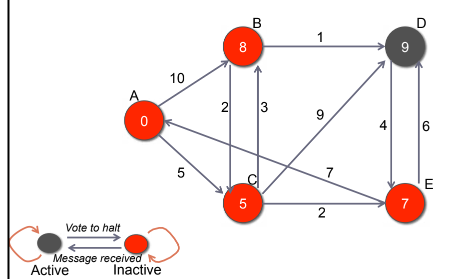
SSSP using parallel BFS in Pregel



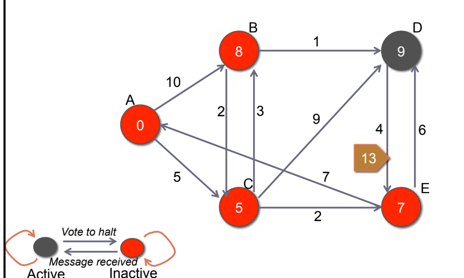
SSSP using parallel BFS in Pregel



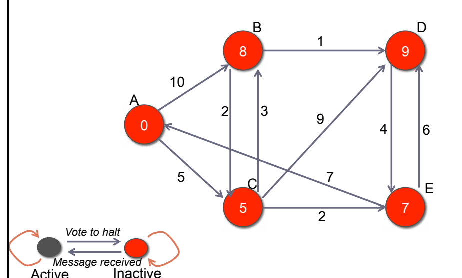
SSSP using parallel BFS in Pregel



SSSP using parallel BFS in Pregel



SSSP using parallel BFS in Pregel



Advanced Message Passing Features

Combiners

- Sending message incurs overhead
- System can combine several messages intended for a vertex V into a single message
  - E.g. calculating sum: values intended for V will be combined as the local sum and then passed to the vertex V
- Combiner class overrides a virtual Combine() method
- For SSSP, more than 4x reduction in message traffic

Aggregators

- A mechanism for global communication
- Each vertex can provide a value to an aggregator in superstep S
  - System combines these values using a reduction operator
- Resulting value is available to all vertices in the superstep S+1

Aggregators

- Pre-defined aggregator
  - min, max or sum operations
- User-defined aggregator
  - Specify the process in the Aggregator class
- Sticky aggregator
  - Uses input values from all supersteps
  - E.g. maintaining global edge count

Topology Mutations

- Some graph algorithms need to change the graph's topology
  - Clustering algorithm
    - Might replace each cluster with a single vertex
  - Minimum spanning tree
    - Might remove all but the tree edges
- Pregel allows the edges to be removed/added

Resolving conflicting requests (1/2)

- Multiple vertices may issue conflicting requests in the same superstep
  - E.g. two requests to add a vertex V, with different initial values
- Partial ordering
  - Mutations become effective in the superstep after the requests were issued
  - Within that superstep, removals are performed first
    - Edge removal before the vertex removal
  - After removal, addition will be performed
    - Vertex addition before the edge addition

Resolving conflicting requests (2/2)

- If there are multiple requests to create the same vertex in the same superstep
  - By default the system just picks one arbitrarily
  - Users can specify their own conflict resolution policy
    - By defining a handler method in their `Vertex` subclass

Global mutation

- Lazy coordination mechanism
  - Global mutation does not require coordination until the point when they are applied
  - A vertex V will handle its own modifications

Contrasting Pregel & MapReduce

Pregel vs. MapReduce

- Many of graph algorithms can be written as a series of chained MapReduce invocations

- Pregel
  - Once the vertices and edges are loaded into computing nodes, they will stay on that machine
  - Only messages will be transferred through the network

- MapReduce
  - Passes the entire state of graph for every iteration
  - External coordinator is required to create a “chain” of MapReduce jobs

System Architecture

System Architecture

- Master/worker model

  - Worker
    - Processes user-defined tasks
    - Communicates with other workers (messaging)
  - Master
    - Maintains information about workers
    - No portion of graph assigned
    - Recovers from faults
    - Uses monitoring tools

- Underlying persistent data storage: GFS or BigTable
- Temporary data is stored on local disk

Step-by-step execution (1/4)

1. A client launches a Pregel job
   - Many copies of the user program begin executing on a cluster of machines
   - One of these copies acts as the master
   - Workers use the cluster management system's name service to discover the master's location
     - Send registration messages to the master
2. The master assigns a partition of the input to each worker
   - Each worker loads the vertices and marks them as active

Step-by-step execution (2/4)

2. The master assigns a partition of the input to each worker
   - Worker:
     - Loads the vertices and marks them as active
     - Maintains the state of its section of the graph
     - Executes user's `Compute()` method on its vertices
     - Manages messages to and from other workers

Step-by-step execution (3/4)

3. The master instructs each worker to perform a superstep
   - Performs user-defined function on the active vertices
   - Messages are sent ***asynchronously***
     - Before the end of the superstep
   - This step is repeated until: ***(a) all of the vertices are inactive simultaneously && (b) no messages are transferred***

Step-by-step execution (4/4)

4 After the computation halts, the master may instruct each worker to save its portion of the graph

Fault tolerance (1/2)

- System maintains checkpoints
- The master periodically requests the workers to save the state of their partitions to persistent storage
  - State is saved as checkpoints, and includes …
    - Vertex values, edge values, incoming messages

Fault tolerance (2/2)

- Failure detection
  - Regular "ping" message
- Recovery
  - The master reassigns graph partitions to the current available workers
  - The workers all reload their partition state from most recent available checkpoint

PageRank Algorithm in Pregel

PageRank Algorithm

- Link analysis algorithm
- Probability distribution
  - Represents the likelihood that a person randomly clicking on links will arrive at any particular page
- Probability
  - Between 0 and 1
  - PageRank of 0.5
    - There is a 50% chance that a person clicking on a random link will be directed to the document with the 0.5 PageRank

Iterative approach

- A link to a page counts as a vote of support
- At t=0,
- PR(pi;0)=1/N
- At each time step, the computation yields,

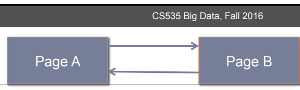
$$PR(p_j;t+1) = \frac{1-d}{N} + d\sum \frac{PF(p_j;t)}{L(p_j)}$$

Iterative approach

- Damping factor, d
  - An imaginary surfer who is randomly clicking on links and he/she will eventually stop
  - The probability that the imaginary person will continue in that step
  - Generally assumed as around 0.85

$$PR(p_i;t+1) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PF(p_j;t)}{L(p_j)}$$

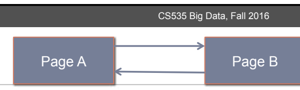
- Where, PR(p,t) : PageRank for the page p at timestep t
- L(p): number of links from page p



- PR(A)=1, PR(B)=1, d =0.85
- PR(A) = (1-d) +(0.85)x1=1
- PR(B) = (1-d) + (0.85)x1=1

- PR(A)=0, PR(B)=0, d =0.85
- PR(A) = (1-d) +(0.85)x0=0.15
- PR(B) = (1-d) + (0.85)x0.15=0.2775

- PR(A)=0.15, PR(B)=0.2775, d =0.85
- PR(A) = (1-d) +(0.85)x0.2775=0.385875
- PR(B) = (1-d) + (0.85)x0.385875=0.47799375



- PR(A)=0.385875, PR(B)=0.47799375, d =0.85
- PR(A) = (1-d) +(0.85)x0.47799375=0.5562946875
- PR(B) = (1-d) + (0.85)x0.5562946875=0.622850484375

- The numbers just keep going up.
- But will the numbers stop increasing when they get to 1.0?
- What if a calculation over-shoots and goes above 1.0?

- PR(A)=40, PR(B)=40, d =0.85
- PR(A) = (1-d) +(0.85)x40=34.25
- PR(B) = (1-d) + (0.85)x34.25=29.1775

- PR(A)=34.25, PR(B)=29.1775, d =0.85
- PR(A) = (1-d) +(0.85)x29.1775=24.950875
- PR(B) = (1-d) + (0.85)x34.25=21.35824375

- The numbers are heading down.
- The numbers will get to 1.0 and stop

In Pregel

```
class PageRankVertex:public Vertex<double, void,
double>{
  public:
   virtual void Compute(MessageIterator* msgs){
    if (superstep()>=1){
     double sum = 0;
     for(; !msgs->Done(); msgs->Next())
       sum +=msgs->Value();
       *MutableValue()=0.15/NumVertices(+0.85*sum;
    }
    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue()/n);
    }else{
      VoteToHalt();
    }
   }
 };
```