CS 347:
Parallel and Distributed Data Management

Notes X:
BigTable, HBASE, Cassandra
Sources


Lots of Buzz Words!

• “Apache Cassandra is an open-source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tunably consistent, column-oriented database that bases its distribution design on Amazon’s dynamo and its data model on Google’s Big Table.”

• Clearly, it is buzz-word compliant!!
Basic Idea: Key-Value Store

Table T:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>v1</td>
</tr>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
</tbody>
</table>
### Basic Idea: Key-Value Store

Table T:

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>v1</td>
</tr>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
</tbody>
</table>

- keys are sorted

- **API:**
  - `lookup(key) → value`
  - `lookup(key range) → values`
  - `getNext → value`
  - `insert(key, value)`
  - `delete(key)`

- Each row has timestamp
- Single row actions atomic (but not persistent in some systems?)
- No multi-key transactions
- No query language!
Fragmentation (Sharding)

- use a partition vector
- “auto-sharding”: vector selected automatically
Tablet Replication

- **Cassandra:**
  - Replication Factor (# copies)
  - R/W Rule: One, Quorum, All
  - Policy (e.g., Rack Unaware, Rack Aware, ...)
  - Read all copies (return fastest reply, do repairs if necessary)

- **HBase:** Does not manage replication, relies on HDFS
Need a “directory”

- Table Name: Key $\rightarrow$ Server that stores key $\rightarrow$ Backup servers
- Can be implemented as a special table.
Design Philosophy: Only write to disk using sequential I/O.
Tablet Internals

- tablet is merge of all layers (files)
- disk segments immutable
- writes efficient; reads only efficient when all data in memory
  - use bloom filters for reads
- periodically reorganize into single layer (merge, major compaction)

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k8</td>
<td>v8</td>
</tr>
<tr>
<td>k9</td>
<td>delete</td>
</tr>
<tr>
<td>k15</td>
<td>v15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k6</td>
<td>v6</td>
</tr>
<tr>
<td>k9</td>
<td>v9</td>
</tr>
<tr>
<td>k12</td>
<td>v12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
<tr>
<td>k5</td>
<td>delete</td>
</tr>
<tr>
<td>k10</td>
<td>v10</td>
</tr>
<tr>
<td>k20</td>
<td>v20</td>
</tr>
<tr>
<td>k22</td>
<td>v22</td>
</tr>
</tbody>
</table>

Memory:
- tombstone

Disk:
- flush periodically (minor compaction)
## Column Family

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>k2</td>
<td>a2</td>
<td>null</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
</tr>
<tr>
<td>k3</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>d3</td>
<td>e3</td>
</tr>
<tr>
<td>k4</td>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>e4</td>
<td>e4</td>
</tr>
<tr>
<td>k5</td>
<td>a5</td>
<td>b5</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
### Column Family

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>k2</td>
<td>a2</td>
<td>null</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
</tr>
<tr>
<td>k3</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>d3</td>
<td>e3</td>
</tr>
<tr>
<td>k4</td>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>e4</td>
<td>e4</td>
</tr>
<tr>
<td>k5</td>
<td>a5</td>
<td>b5</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- for storage, treat each row as a single “super value”
- API provides access to sub-values (use family:qualifier to refer to sub-values e.g., price:euros, price:dollars)
- Cassandra allows “super-column”: two level nesting of columns (e.g., Column A can have sub-columns X & Y)
Vertical Partitions

can be manually implemented as
Vertical Partitions

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td>e1</td>
</tr>
<tr>
<td>k2</td>
<td>a2</td>
<td>null</td>
<td>c2</td>
<td>d2</td>
<td>e2</td>
</tr>
<tr>
<td>k3</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>d3</td>
<td>e3</td>
</tr>
<tr>
<td>k4</td>
<td>a4</td>
<td>b4</td>
<td>c4</td>
<td>e4</td>
<td>e4</td>
</tr>
<tr>
<td>k5</td>
<td>a5</td>
<td>b5</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

- Good for sparse data;
- Good for column scans
- Not so good for tuple reads
- Are atomic updates to row still supported?
- API supports actions on full table; mapped to actions on column tables
- API supports column "project"
- To decide on vertical partition, need to know access patterns
Failure Recovery (BigTable, HBase)

memory → tablet server ← master node

ping

write ahead logging

log

GFS or HDFS

spare tablet server
Failure recovery (Cassandra)

- No master node, all nodes in “cluster” equal

server 1  server 2  server 3
Failure recovery (Cassandra)

- No master node, all nodes in “cluster” equal

access any table in cluster at any server

server 1  server 2  server 3

that server sends requests to other servers
CS 347:  
Parallel and Distributed  
Data Management  

Notes X: MemCacheD
MemCacheD

• General-purpose distributed memory caching system
• Open source
What MemCacheD Is

put(cache 1, myName, X)

get_object(cache 1, MyName)

cache 1

data source 1

cache 2

data source 2

cache 3

data source 3

x
What MemCacheD Is

- put(cache 1, myName, X)
- get_object(cache 1, MyName)

Can purge MyName whenever each cache is hash table of (name, value) pairs

data source 1

data source 2

data source 3

Each cache has no connection.
What MemCacheD Could Be (but ain't)

data source 1

get_object(X)

distributed cache

cache 1

data source 1

cache 2

data source 2

cache 3

data source 3
What MemCacheD Could Be (but ain't)

get_object(X) -> distributed cache

- data source 1
- cache 1
- data source 2
- cache 2
- data source 3
- cache 3

Note: The 'x' marks the caches that have the requested data.
Persistence?

put(cache 1, myName, X)

get_object(cache 1, MyName)

Can purge MyName whenever

each cache is hash table of (name, value) pairs
Persistence?

Example: MemcacheDB = memcached + BerkeleyDB

each cache is hash table of (name, value) pairs

put(cache 1, myName, X)

cache 1

get_object(cache 1, MyName)

cache 2

Can purge MyName whenever

cache 3

Example: MemcacheDB = memcached + BerkeleyDB
CS 347:
Parallel and Distributed
Data Management

Notes X: ZooKeeper
ZooKeeper

- Coordination service for distributed processes
- Provides clients with high throughput, high availability, memory only file system

![ZooKeeper diagram]

znode /a/d/e (has state)
ZooKeeper Servers

client

client

client

client

server

state replica

server

state replica

server

state replica
ZooKeeper Servers

client
client
client
client
client

server → state replica
server → state replica
server → state replica

read
ZooKeeper Servers

writes totally ordered:
used Zab algorithm

client

client

client

client

server

state replica

server

state replica

server

state replica

write

propagate & sych
If your server dies, just connect to a different one!
ZooKeeper Notes

- Differences with file system:
  - all nodes can store data
  - storage size limited
- API: insert node, read node, read children, delete node, ...
- Can set triggers on nodes
- Clients and servers must know all servers
- ZooKeeper works as long as a majority of servers are available
- Writes totally ordered; read ordered w.r.t. writes
ZooKeeper Applications

- Metadata/configuration store
- Lock server
- Leader election
- Group membership
- Barrier
- Queue
CS 347:
Parallel and Distributed Data Management

Notes X: S4
Material based on:

2010 IEEE International Conference on Data Mining Workshops

**S4: Distributed Stream Computing Platform**

Leonardo Neumeyer  
*Yahoo! Labs*  
Santa Clara, CA  
neumeyer@yahoo-inc.com

Bruce Robbins  
*Yahoo! Labs*  
Santa Clara, CA  
robbins@yahoo-inc.com

Anish Nair  
*Yahoo! Labs*  
Santa Clara, CA  
anishn@yahoo-inc.com

Anand Kesari  
*Yahoo! Labs*  
Santa Clara, CA  
anands@yahoo-inc.com
S4

- Platform for processing unbounded data **streams**
  - general purpose
  - distributed
  - scalable
  - partially fault tolerant
    (whatever this means!)
Data Stream

Terminology: event (data record), key, attribute

Question: Can an event have duplicate attributes for same key? (I think so...)

Stream unbounded, generated by say user queries, purchase transactions, phone calls, sensor readings, ...
S4 Processing Workflow

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>34</td>
</tr>
<tr>
<td>B</td>
<td>abcd</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

user specified “processing unit”
Inside a Processing Unit

processing element

key=a

key=b

key=z

PE1

PE2

PEn
Example:

- Stream of English quotes
- Produce a sorted list of the top K most frequent words
A keyless event (EV) arrives at PE1 with quote: "I meant what I said and I said what I meant.”, Dr. Seuss

**QuoteSplitterPE** (PE1) counts unique words in Quote and emits events for each word.

**WordCountPE** (PE2–4) keeps total counts for each word across all quotes. Emits an event any time a count is updated.

**SortPE** (PE5–7) continuously sorts partial lists. Emits lists at periodic intervals.

**MergePE** (PE8) combines partial TopK lists and outputs final TopK list.

<table>
<thead>
<tr>
<th>PE ID</th>
<th>PE Name</th>
<th>Key Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE1</td>
<td>QuoteSplitterPE</td>
<td>null</td>
</tr>
<tr>
<td>PE2</td>
<td>WordCountPE</td>
<td>word=&quot;said&quot;</td>
</tr>
<tr>
<td>PE4</td>
<td>WordCountPE</td>
<td>word=&quot;i&quot;</td>
</tr>
<tr>
<td>PE5</td>
<td>SortPE</td>
<td>sortID=2</td>
</tr>
<tr>
<td>PE6</td>
<td>SortPE</td>
<td>sortID=9</td>
</tr>
<tr>
<td>PE7</td>
<td>SortPE</td>
<td>sortID=9</td>
</tr>
<tr>
<td>PE8</td>
<td>MergePE</td>
<td>topK=1234</td>
</tr>
</tbody>
</table>
Processing Nodes

hash(key) = 1
key = a

hash(key) = m
key = b
Dynamic Creation of PEs

• As a processing node sees new key attributes, it dynamically creates new PEs to handle them
• Think of PEs as threads
Another View of Processing Node
Failures

- Communication layer detects node failures and provides failover to standby nodes
- What happens events in transit during failure? (My guess: events are lost!)
How do we do DB operations on top of S4?

• Selects & projects easy!

• What about joins?
What is a Stream Join?

- For true join, would need to store all inputs forever! Not practical...
- Instead, define window join:
  - at time $t$ new $R$ tuple arrives
  - it only joins with previous $w$ $S$ tuples
One Idea for Window Join

### Table 1: Attributes of R and S

<table>
<thead>
<tr>
<th>key</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>R</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>S</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

**Code for PE:**

```python
for each event e:
    if e.rel=R:
        [store in Rset(last w)
        for s in Sset:
            output join(e,s) ]
    else ...
```

“key” is join attribute
One Idea for Window Join

<table>
<thead>
<tr>
<th>key</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>R</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>S</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

code for PE:
for each event e:
  if e.rel=R
    [store in Rset(last w)
     for s in Sset:
     output join(e,s) ]
else ...

“key” is join attribute

Is this right???
(enforcing window on a per-key value basis)

Maybe add sequence numbers to events to enforce correct window?
Another Idea for Window Join

<table>
<thead>
<tr>
<th>key</th>
<th>fake</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>R</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>fake</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>S</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

code for PE:
for each event e:
  if e.rel=R
    [store in Rset(last w)
     for s in Sset:
       if e.C=s.C then
         output join(e,s) ]
  else if e.rel=S ...

All R & S events have “key=fake”;
Say join key is C.
Another Idea for Window Join

<table>
<thead>
<tr>
<th>key</th>
<th>fake</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>R</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>fake</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>S</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

code for PE:
for each event e:
    if e.rel=R
        [store in Rset(last w)
         for s in Sset:
             if e.C=s.C then
                output join(e,s) ]
    else if e.rel=S ...

All R & S events have “key=fake”; Say join key is C.

Entire join done in one PE; no parallelism
Do You Have a Better Idea for Window Join?

<table>
<thead>
<tr>
<th>key</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>R</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>rel</td>
<td>S</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Final Comment: Managing state of PE

Who manages state?

S4: user does
Mupet: System does
Is state persistent?

<table>
<thead>
<tr>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>34</td>
</tr>
<tr>
<td>B</td>
<td>abcd</td>
</tr>
<tr>
<td>C</td>
<td>15</td>
</tr>
<tr>
<td>D</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

\[ 34 \]
\[ \text{abcd} \]
\[ 15 \]
\[ \text{TRUE} \]
CS 347:
Parallel and Distributed
Data Management

Notes X: Hyracks
Hyracks

• Generalization of map-reduce
• Infrastructure for “big data” processing
• Material here based on:

Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing

Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, Rares Vernica

Computer Science Department, University of California, Irvine
Irvine, CA 92697
vborkar@ics.uci.edu

Appeared in ICDE 2011
A Hyracks data object:

- Records partitioned across N sites
- Simple record schema is available (more than just key-value)
Operations

operator

distribution rule
Operations (parallel execution)
Example: Hyracks Specification

```sql
select C_MKTSEGMENT, count(O_ORDERKEY)
from CUSTOMER join ORDERS on C_CUSTKEY = O_CUSTKEY
group by C_MKTSEGMENT
```

Fig. 1: Example Hyracks job specification
Example: Hyracks Specification

```
select C_MKTSEGMENT, count (O_ORDERKEY)
from CUSTOMER join ORDERS on C_CUSTKEY = O_CUSTKEY
group by C_MKTSEGMENT
```

Fig. 1: Example Hyracks job specification
Notes

- Job specification can be done manually or automatically
Example: Activity Node Graph

![Diagram of Activity Node Graph]

Fig. 2: Example Hyracks Activity Node graph
Example: Activity Node Graph

Fig. 2: Example Hyracks Activity Node graph
Example: Parallel Instantiation

Fig. 3: Parallel instantiation of the example
Example: Parallel Instantiation

Fig. 3: Parallel instantiation of the example
System Architecture

Fig. 4: Hyracks system architecture
Library of Operators:

- File reader/writers
- Mappers
- Sorters
- Joiners (various types)
- Aggregators

- Can add more
Library of Connectors:

- N:M hash partitioner
- N:M hash-partitioning merger (input sorted)
- N:M range partitioner (with partition vector)
- N:M replicator
- 1:1

- Can add more!
Hyracks Fault Tolerance: Work in progress?
Hyraccs Fault Tolerance: Work in progress?

One approach: save partial results to persistent storage; after failure, redo all work to reconstruct missing data
Hyracks Fault Tolerance: Work in progress?

Can we do better?
Maybe: each process retains previous results until no longer needed?

Pi output: r1, r2, r3, r4, r5, r6, r7, r8
have "made way" to final result

current output
CS 347:
Parallel and Distributed
Data Management

Notes X: Pregel
Material based on:

**Pregel: A System for Large-Scale Graph Processing**

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

Google, Inc.

{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

- In SIGMOD 2010

- Note there is an open-source version of Pregel called GIRAPH
Pregel

- A computational model/infrastructure for processing large graphs
- Prototypical example: Page Rank

PR[i+1,x] = f(PR[i,a]/na, PR[i,b]/nb)
Pregel

PR[i+1,x] = f(PR[i,a]/na, PR[i,b]/nb)

- Synchronous computation in iterations
- In one iteration, each node:
  - gets messages from neighbors
  - computes
  - sends data to neighbors
Pregel vs Map-Reduce/S4/Hyracks/...

- In Map-Reduce, S4, Hyracks,... workflow separate from data

- In Pregel, data (graph) drives data flow
Pregel Motivation

- Many applications require graph processing
- Map-Reduce and other workflow systems not a good fit for graph processing
- Need to run graph algorithms on many processors
Example of Graph Computation

![Graph Diagram]

Superstep 0

Superstep 1

Superstep 2

Superstep 3

Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.
Termination

- After each iteration, each vertex votes to halt or not
- If all vertexes vote to halt, computation terminates

Figure 1: Vertex State Machine
Vertex Compute (simplified)

• Available data for iteration i:
  – InputMessages: { [from, value] }
  – OutputMessages: { [to, value] }
  – OutEdges: { [to, value] }
  – MyState: value

• OutEdges and MyState are remembered for next iteration
Max Computation

- change := false
- for [f,w] in InputMessages do
  if w > MyState.value then
    [ MyState.value := w
    change := true ]
- if (superstep = 1) OR change then
  for [t, w] in OutEdges do
    add [t, MyState.value] to OutputMessages
  else vote to halt
Page Rank Example

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();
    }
  }
};
Page Rank Example

class PageRankVertex : public Vertex<double, void, void> {
  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();
      }
    }
};
Single-Source Shortest Paths

class ShortestPathVertex
    : public Vertex<int, int, int> {
    void Compute(MessageIterator* msgs) {
        int mindist = IsSource(vertex_id()) ? 0 : INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for (; !iter.Done(); iter.Next())
                SendMessageTo(iter.Target(),
                mindist + iter.GetValue());
        }
        VoteToHalt();
    }
};
Architecture

master

graph has nodes a, b, c, d...

worker A

input data 1

worker B

input data 2

worker C

sample record: [a, value]
Architecture

master

worker A
vertexes
a, b, c

worker B
vertexes
d, e

worker C
vertexes
f, g, h

input data 1

input data 2

partition graph and
assign to workers
Architecture

master

read input data

worker A
vertexes
a, b, c

input data 1

input data 2

worker B
vertexes
d, e

worker C
vertexes
f, g, h

worker A forwards input values to appropriate workers
master

run superstep 1

worker A
vertexes
a, b, c

worker B
vertexes
d, e

worker C
vertexes
f, g, h

input data 1
input data 2

Architecture
Architecture

master

worker A
vertexes a, b, c

worker B
vertexes d, e

worker C
vertexes f, g, h

input data 1

input data 2

halt?

at end superstep 1, send messages
Architecture

- **Master**
- **Worker A**
  - Vertexes: a, b, c
- **Worker B**
  - Vertexes: d, e
- **Worker C**
  - Vertexes: f, g, h

Run superstep 2
Architecture

master

checkpoint

worker A

vertexes
a, b, c

input data 1

worker B

vertexes
d, e

input data 2

worker C

vertexes
f, g, h
Architecture

master

worker A
vertexes a, b, c

worker B
vertexes d, e

worker C
vertexes f, g, h

checkpoint
write to stable store: MyState, OutEdges, InputMessages (or OutputMessages)
Architecture

master

worker A
vertexes
a, b, c

worker B
vertexes
d, e

if worker dies, find replacement & restart from latest checkpoint
Architecture

master

worker A

 vertexes
 a, b, c

input data 1

worker B

 vertexes
 d, e

input data 2

worker C

 vertexes
 f, g, h
Interesting Challenge

- How best to partition graph for efficiency?
CS 347:
Parallel and Distributed Data Management

Notes X: Kestrel
Kestrel

- Kestrel server handles a set of reliable, ordered message queues
- A server does not communicate with other servers (advertised as good for scalability!)
Kestrel

- Kestrel server handles a set of reliable, ordered message queues
- A server does not communicate with other servers (advertised as good for scalability!)

![Diagram of client-server communication with queues and logs]