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

Inside a Processing Unit

processing element
Example:
- Stream of English quotes
- Produce a sorted list of the top K most frequent words

Processing Nodes
- As a processing node sees new key attributes, it dynamically creates new PEs to handle them
- Think of PEs as threads

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
- Communication layer detects node failures and provides failover to standby nodes
- What happens events in transit during failure? (My guess: events are lost!)

Failures
How do we do DB operations on top of S4?

- Selects & projects easy!

What about joins?

- 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

```plaintext
key 2 rel R C 55 D TRUE
```

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???

Maybe add sequence numbers to events to enforce correct window?

Another Idea for Window Join

```plaintext
key false rel R C 15 D TRUE
```

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?

R

S

Final Comment: Managing state of PE

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

CS 347: Parallel and Distributed Data Management

Notes X: Hyracks

Hector Garcia-Molina

Hyracks

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

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

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

Example: Activity Node Graph
Example: Parallel Instantiation

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 rage partitioner (with partition vector)
- N:M replicator
- 1:1
- Can add more!
Hyracks Fault Tolerance: Work in progress?

CS 347 Lecture 9B 37

Hadoop/MR approach: save partial results to persistent storage; after failure, redo all work to reconstruct missing data

CS 347 Lecture 9B 38

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

CS 347 Lecture 9B 39

Hector Garcia-Molina

Material based on:
Pregel: A System for Large-Scale Graph Processing
Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ian Horn, Naty Leiser, and Grigorios Cachatowski
Boostrap. (malewicz.austern.ajcbik.dehnert.ian.naty.grgot@ias.google.com)

- In SIGMOD 2010
- Note there is an open-source version of Pregel called GIRAPH

CS 347 Lecture 9B 40

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

CS 347 Lecture 9B 41

PR(i+1,x) = \sum_{i \in S_i} \frac{PR(i,a)}{na} \cdot PR(i,b)/nb

CS 347 Lecture 9B 42
Pregel

- Synchronous computation in iterations
- In one iteration, each node:
  - gets messages from neighbors
  - computes
  - sends data to neighbors

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

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

Termination

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

\[ Vote to halt \]

\[ Message received \]

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
  end
- if (superstep = 1) OR change then
  for [t, w] in OutEdges do
    add [t, MyState.value] to OutputMessages
  end
else vote to halt

Page Rank Example

class PageRankVertex:
  public double, void, double:
  public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() < 30) {
      MyState.value += msgs->Next();
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      MyState.value = 0.15 / NumVertices() + 0.85 * sum;
    }
    iteration count
    iterate thru InputMessages
    MyState.value
    shorthand: send same msg to all
  }
  else {
    VoteToHalt();
  }
}

Single-Source Shortest Paths

class ShortestPathVertex:
  public 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()) {
      *NotableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
      VoteToHalt();
    }
  }

Architecture

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

input data 1
input data 2

sample record:
[a, value]

Architecture

partition graph and assign to workers
worker A reads input data
\begin{itemize}
\item vertexes: a, b, c
\item input data 1
\end{itemize}
forwards input values to appropriate workers
\begin{itemize}
\item vertexes: d, e
\item input data 2
\end{itemize}

worker C
\begin{itemize}
\item vertexes: f, g, h
\end{itemize}

\textbf{Architecture}

worker A runs superstep 1
\begin{itemize}
\item vertexes: a, b, c
\item input data 1
\end{itemize}

worker B
\begin{itemize}
\item vertexes: d, e
\item input data 2
\end{itemize}

worker C
\begin{itemize}
\item vertexes: f, g, h
\end{itemize}

\textbf{Architecture}

worker A runs superstep 2
\begin{itemize}
\item vertexes: a, b, c
\item input data 1
\end{itemize}

worker B
\begin{itemize}
\item vertexes: d, e
\item input data 2
\end{itemize}

worker C
\begin{itemize}
\item vertexes: f, g, h
\end{itemize}

\textbf{Architecture}

worker A halts?
\begin{itemize}
\item vertexes: a, b, c
\item input data 1
\end{itemize}

worker B
\begin{itemize}
\item vertexes: d, e
\item input data 2
\end{itemize}

worker C
\begin{itemize}
\item vertexes: f, g, h
\end{itemize}

\textbf{Architecture}

worker A
\begin{itemize}
\item vertexes: a, b, c
\item input data 1
\end{itemize}

worker B
\begin{itemize}
\item vertexes: d, e
\item input data 2
\end{itemize}

worker C
\begin{itemize}
\item vertexes: f, g, h
\end{itemize}

\textbf{Architecture}

checkpoint
\begin{itemize}
\item write to stable store: MyState, OutEdges, InputMessages (or OutputMessages)
\end{itemize}
Interesting Challenge

• How best to partition graph for efficiency?

Sources

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>

- 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 → Server that stores key → Backup servers
- Can be implemented as a special table.

Tablet Internals

- Design Philosophy (?): Primary scenario is where all data is in memory.
- Disk storage added as an afterthought
### Tablet Internals

- Tablet is a merge of all segments (files)
- Disk segments are immutable
- Writes efficient; reads only efficient when all data in memory
- Periodically reorganize into a single segment

#### Column Family

- Table: KABCDE
- **Key:** K
- **Values:** A1 B1 C1 D1 E1

#### Vertical Partitions

- Can be manually implemented as

```
<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

### Column Family

- Table: KABCDE
- **Key:** K
- **Values:** A1 B1 C1 D1 E1

### Vertical Partitions

- Can be manually implemented as

```
<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>
```

### Failure Recovery (BigTable, HBase)

- **Memory:**
- **Tablet Server:**
- **Master Node:**
- **Spare Tablet Server:**
- **Log:**
- **Write Ahead Logging:**
- **GFS or HFS:**
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

server 1  server 2  server 3

access any table in cluster equal at any server

that server sends requests to other servers

CS 347: Parallel and Distributed Data Management

Notes X: MemCacheD

Hector Garcia-Molina

MemCacheD
- General-purpose distributed memory caching system
- Open source

What MemCacheD Should Be (but ain't)

get object(X)

distributed cache
cache 1  cache 2  cache 3
data source 1  data source 2  data source 3

What MemCacheD Should Be (but ain't)

ged_object(X)
distributed cache
cache 1  cache 2  cache 3
data source 1  data source 2  data source 3

x
What MemCacheD Is (as far as I can tell)

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

**CS 347 Lecture 9B 85**

---

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**

**CS 347 Lecture 9B 86**

---

CS 347:
Parallel and Distributed
Data Management

**Notes X: ZooKeeper**

Hector Garcia-Molina

**CS 347 Lecture 9B 87**

---

ZooKeeper

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

**CS 347 Lecture 9B 88**

---

ZooKeeper Servers

**CS 347 Lecture 9B 89**

---

ZooKeeper Servers

**CS 347 Lecture 9B 90**
ZooKeeper Servers

Client

Server

State replica

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

Kestrel

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