Performance services

Badri Nath
Rutgers University
badri@cs.rutgers.edu

1. Pig latin
2. Dynamo
What's happening?

Clusters: thousands of machines

Efficient Processing
Analysis

Results

Tera, Peta, Exa bytes

$10^{12}, 10^{15}, 10^{18}$
Work load is different

- Huge data sets
  - Tera bytes, peta bytes are common
- Read-only data, Scan-centric
  - Applications need a run through the entire data set for analysis
- Need to take advantage of parallelism (cluster hardware)
  - Very little dependency, no txns, no index based lookups
Pig & Pig Latin

- A layer on top of map-reduce (Hadoop)
  - Pig is the system
  - Pig Latin is the query language
  - Pig Pen is a debugging environment

- Pig Latin is a hybrid between:
  - high-level declarative query language in the spirit of SQL
  - low-level, procedural programming à la map-reduce
  - Parts of DB engine is exposed

- Can we say that the DB community has gone hog wild!!!
## Example beer table

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>beer</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>23</td>
<td>amstel</td>
<td>0.5</td>
</tr>
<tr>
<td>B</td>
<td>34</td>
<td>bud</td>
<td>0.2</td>
</tr>
<tr>
<td>C</td>
<td>25</td>
<td>corona</td>
<td>0.7</td>
</tr>
<tr>
<td>D</td>
<td>42</td>
<td>bud</td>
<td>0.2</td>
</tr>
<tr>
<td>E</td>
<td>22</td>
<td>corona</td>
<td>0.7</td>
</tr>
<tr>
<td>F</td>
<td>27</td>
<td>heineken</td>
<td>0.6</td>
</tr>
<tr>
<td>G</td>
<td>25</td>
<td>amstel</td>
<td>0.5</td>
</tr>
<tr>
<td>H</td>
<td>26</td>
<td>corona</td>
<td>0.7</td>
</tr>
<tr>
<td>I</td>
<td>30</td>
<td>corona</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Example

- Table `BeerPreference`: `(name, age, beer, quality)`
- Find, for each high quality beer, the average age of beer drinkers in that category. In SQL:
  - `SELECT beer, AVG(age)`
  - `FROM BeerPreference WHERE quality > 0.2`
  - `GROUP BY beer`
Example in Pig Latin

- Same query in Pig Latin

- `good_beer = FILTER beerpreference BY quality > 0.2;
  beer_groups = GROUP good_beer BY beer;
  output = FOREACH beer_groups GENERATE
            beer, AVG(good_beer.age);`
Filter, GROUP

Good_beer = FILTER beerpref by quality > 0.2

Beer_group = GROUP good-beer by beer

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>beer</th>
<th>quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>23</td>
<td>amstel</td>
<td>0.5</td>
</tr>
<tr>
<td>C</td>
<td>25</td>
<td>corona</td>
<td>0.7</td>
</tr>
<tr>
<td>E</td>
<td>22</td>
<td>corona</td>
<td>0.7</td>
</tr>
<tr>
<td>F</td>
<td>27</td>
<td>heineken</td>
<td>0.6</td>
</tr>
<tr>
<td>G</td>
<td>25</td>
<td>amstel</td>
<td>0.5</td>
</tr>
<tr>
<td>H</td>
<td>26</td>
<td>corona</td>
<td>0.7</td>
</tr>
<tr>
<td>I</td>
<td>30</td>
<td>corona</td>
<td>0.7</td>
</tr>
</tbody>
</table>
# Beer-group= (beer,good-beer)

<table>
<thead>
<tr>
<th>Beer</th>
<th>(Author,Year,Brand,Confidence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>amstel</td>
<td>(A,23,Amstel,0.5), (G,25,Amstel,0.5)</td>
</tr>
<tr>
<td>Heineken</td>
<td>(F,27,heineken,0.6)</td>
</tr>
<tr>
<td>corona</td>
<td>(C,25,corona,0.7), (H,26,corona,0.7), (I,30,corona,0.7)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>amstel</td>
<td>26</td>
</tr>
<tr>
<td>heineken</td>
<td>27</td>
</tr>
<tr>
<td>corona</td>
<td>27</td>
</tr>
</tbody>
</table>
Example (in the paper)

- Table urls: (url, category, pagerank)
- Find, for each sufficiently large category, the average pagerank of high-pagerank urls in that category. In SQL:
  
  ```sql
  SELECT category, AVG(pagerank)
  FROM urls
  WHERE pagerank > 0.2
  GROUP BY category
  HAVING COUNT(*) > 10^6
  ```
Example in Pig Latin

- Same query in Pig latin
- good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups
  BY COUNT(good_urls) > 10^6;
output = FOREACH big_groups GENERATE
category, AVG(big_groups.pagerank);

1. Sequence of steps; each step applies a transformation to data (potentially huge)
2. Data flow !!
3. User is specifying the query execution plan
4. High-level operations (filter group, aggregate)
Features

- Pig is a data flow language
  - Data fed to high level operations
- Nested data model
  - Forget 1NF !!
- UDF or user-defined functions
  - Spam-urls = FILTER urls BY isSpam(url).
  - Bugs!?!?
- Provides A debugging environment
- Free format (Schema is optional)
  - It can Process anything
Nested data model

- A given data item has multiple references
- Easy to capture that in a nested structure
  - (key, (ref1, ref2, ref3))
- Define complex, non-atomic data types
  - set, map, bag etc
- Data, atleast in web processing, is inherently nested
- Suited for data flow language
Example 2

- Beer-groups = GROUP good_beer BY category;
- output = FOREACH beer_groups GENERATE
category, youngest(good-beer.age);

<table>
<thead>
<tr>
<th>beer</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>amstel</td>
<td>{ (A, 23, Amstel, 0.5), (G, 25, Amstel, 0.5) }</td>
</tr>
<tr>
<td>heineken</td>
<td>{ (F, 27, heineken, 0.6) }</td>
</tr>
<tr>
<td>corona</td>
<td>{ (C, 25, corona, 0.7), (H, 26, corona, 0.7), (I, 30, corona, 0.7) }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>beer</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>amstel</td>
<td>23</td>
</tr>
<tr>
<td>heineken</td>
<td>27</td>
</tr>
<tr>
<td>corona</td>
<td>25</td>
</tr>
</tbody>
</table>
Data Model

- Atom, e.g., `alice` or `corona`
- Tuple – sequence of fields, e.g., (`alice', `lakers'); ( a, (a,25,corona,0.7))
- Bag- collection of tuples, e.g.,
  { (`alice', `lakers') (`alice', (`iPod', `apple'))}
- Map, e.g.,
  [ `fan of' → { (`lakers') (`iPod') } `age‘ → 20 ]
Expressions in Pig Latin

\[ t = \left( 'alice', \{ ('lakers', 1), ('iPod', 2) \}, [ 'age' \rightarrow 20 ] \right) \]

Let fields of tuple \( t \) be called \( f_1, f_2, f_3 \)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example</th>
<th>Value for ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>'bob'</td>
<td>Independent of ( t )</td>
</tr>
<tr>
<td>Field by position</td>
<td>$0</td>
<td>'alice'</td>
</tr>
<tr>
<td>Field by name</td>
<td>( f_3 )</td>
<td>( [ 'age' \rightarrow 20 ] )</td>
</tr>
<tr>
<td>Projection</td>
<td>( f_2$.0 )</td>
<td>( { ('lakers') )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( { ('iPod') } )</td>
</tr>
<tr>
<td>Map Lookup</td>
<td>( f_3#'age' )</td>
<td>20</td>
</tr>
<tr>
<td>Function Evaluation</td>
<td>( \text{SUM}(f_2$.1) )</td>
<td>1 + 2 = 3</td>
</tr>
<tr>
<td>Conditional Expression</td>
<td>( f_3#'age'&gt;18? )</td>
<td>'adult'</td>
</tr>
<tr>
<td></td>
<td>'adult': 'minor'</td>
<td></td>
</tr>
<tr>
<td>Flattening</td>
<td>( \text{FLATTEN}(f_2) )</td>
<td>( [ 'lakers', 1 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( 'iPod', 2 )</td>
</tr>
</tbody>
</table>

Table 1: Expressions in Pig Latin.
Specifying Input Data

- `queries = LOAD `query_log.txt' USING myLoad() AS (userId, queryString, timestamp);`
- Need to create a bag for processing
- LOAD returns a handle to a bag
- Myload() is the UDF that does data cleaning
- USING is the deserializer
- AS specifies the names for the fields in the bag
  - Optional, position ($0, $1, …) can be used to refer to fields
  - Field by name or field by position
FOREACH

- expanded_queries = FOREACH queries GENERATE
  userId, UPPERCASE(queryString);

- Apply GENERATE to each tuple
- each tuple is processed independently; easy to parallelize
- To remove one level of nesting:
  expanded_queries = FOREACH queries GENERATE
  userId, FLATTEN(expandQuery(queryString));
ForEach and Flattening

queries:
(userId, queryString, timestamp)

(alice, lakers, 1)
(bob, iPod, 3)

FOREACH queries GENERATE
expandQuery(queryString)
(without flattening)

(alice, {lakers rumors})
(bob, {iPod nano, iPod shuffle})

Figure 1: Example of flattening in FOREACH.
## Flattening Example

<table>
<thead>
<tr>
<th>T</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 2</td>
<td>A 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 4</td>
<td>B 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 7</td>
<td>C 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 9</td>
<td>D 2</td>
</tr>
<tr>
<td></td>
<td>angelina</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>brad</td>
<td>2 2</td>
<td>E 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 4</td>
<td>A 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 7</td>
<td>D 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 9</td>
<td>C 2</td>
</tr>
</tbody>
</table>

\[
\text{ANSWER1} = \text{FOREACH T GENERATE A, FLATTEN (B), C}
\]

\[
\text{ANSWER2} = \text{FOREACH T GENERATE A, FLATTEN (C)}
\]
3.4 Filter

- real_queries = FILTER queries BY userId neq `bot';
- real_queries = FILTER queries BY NOT isBot(userId);

Oink, Oink, Basically a select: PIG operates on BAG, SQL operates on table
**Co-Group**

- Groups two related data sets; for example:
  - `results`: (queryString, url, position)
  - `revenue`: (queryString, adSlot, amount)
- `grouped_data = COGROUP results BY queryString, revenue BY queryString;`
- Apply `GROUP BY` to each bag and produce another bag
  - `commonattr, category1 category2`
- `url_revenues = FOREACH grouped_data GENERATE FLATTEN(distributeRevenue(results, revenue));`
- Co-Group more flexible than SQL JOIN
**CoGroup vs Join**

Results:
- (queryString, url, rank)
  - (lakers, nba.com, 1)
  - (lakers, espn.com, 2)
  - (kings, nhl.com, 1)
  - (kings, nba.com, 2)

Grouped Data:
- (group, results, revenue)
  - (lakers, {lakers, nba.com, 1}, {lakers, espn.com, 2}, {lakers, top, 50})
  - (lakers, {lakers, espn.com, 2}, {lakers, side, 20})
  - (kings, {kings, nhl.com, 1}, {kings, nba.com, 2}, {kings, top, 30})
  - (kings, {kings, nba.com, 2}, {kings, side, 10})

Revenue:
- (queryString, adSlot, amount)
  - (lakers, top, 50)
  - (lakers, side, 20)
  - (kings, top, 30)
  - (kings, side, 10)

Figure 2: COGROUP versus JOIN.
Group (only one data set)

- grouped_revenue =
  GROUP revenue BY queryString;
- {(queryString1, bag) (queryString2, bag)….}
- query_revenues = FOREACH grouped_revenue
  GENERATE queryString, SUM(revenue.amount) AS totalRevenue;
Join in Pig Latin

- join_result = JOIN results BY queryString, revenue BY queryString;
- Shorthand for:
- temp_var = COGROUP results BY queryString, revenue BY queryString;
- join_result = FOREACH temp_var GENERATE FLATTEN(results), FLATTEN(revenue);
- The above is a equi join
MapReduce in Pig Latin

- `map_result = FOREACH input GENERATE FLATTEN(map( ));`
- Map is a UDF works on all documents, contents and produces a bag of key value pairs
- Flatten removes one level of nesting to produce [K2,V2]
- `key_groups = GROUP map_result BY $0;`
- `output = FOREACH key_groups GENERATE reduce( );`
To materialize result in a file:

STORE query_revenues
    INTO `myoutput' USING myStore();

myStore is a UDF
Map-Reduce Plan Compilation

- Map tasks assign keys for grouping, and the reduce tasks process a group at a time.
- Compiler:
- Converts each (CO)GROUP command in the logical plan into a distinct MapReduce job consisting of its own MAP and REDUCE functions.
Debugging Environment

- Iterative process for programming.
- Sandbox data set generated automatically to show results for the expressions.

```
visits = LOAD 'visits.txt' AS (user, url, time);

pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;

users = GROUP v_p BY user;

useravg = FOR EACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;

answer = FILTER useravg BY avgpr > '0.5';

visits: (Amy, cnn.com, 8am)
(Amy, frogs.com, 9am)
(Fred, snails.com, 11am)

pages: (cnn.com, 0.8)
(frogs.com, 0.8)
(snails.com, 0.5)

v_p: (Amy, cnn.com, 8am, cnn.com, 0.8)
(Amy, frogs.com, 9am, frogs.com, 0.8)
(Fred, snails.com, 11am, snails.com, 0.3)

users: (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8),
(Amy, frogs.com, 9am, frogs.com, 0.8) })
(Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })

useravg: (Amy, 0.8)
(Fred, 0.3)

answer: (Amy, 0.8)
```

Figure 4: Pig Pen screenshot; displayed program finds users who tend to visit high-pagerank pages.
PIG is SQL for clusters

- UDF debugging
- Format errors
- User initiated optimization
- Targets a new demographic of programmers
- The cool type
- Knows Python, map-reduce,
- Carries ipod, iphone
Dynamo: Amazon’s key-value store

SOSP 2007 paper

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels
Amazon

- World’s largest online store
- Approx 85 M users
- Approx 3-4 M checkouts per day, 20B in revenues
- A highly available data store
- Downtime ➔ $$$
- Response time, latency important for customers
- User operations cannot be lost
- Massive store of key, value
Dynamo

- Scalable approach to manage persistent store
- Design space
- Simple read and write operations on data
- Trade off consistency for availability, latency
  - ACID vs BASE
- Run on geo redundant cluster
- Cluster is controlled by Amazon (hosts within the cluster can be trusted)
Architecture

S3: hosted storage service
Data Access Model

- Simple API  `put(key, object), get(key)`
- Data stored as (key, object) pairs:
  - Handle or “identifier’ generated as a hash for object
  - Objects: Opaque
  - Application examples: shopping carts, customer preferences, session management, sales rank, and product catalog
Design requirements

- Just a key-value store
- Not a full fledged database
- Store small objects (size < 1 MB)
- Operations on Single Objects
  - Similar to ops on one row in big table
- No constraints across shopping carts
Service level agreements

- User experience is key
- Provide < 300 msec response time for 99.9% of requests
- Peak load of 500 requests/sec
- Customer updates should not be rejected due to system problems
Dynamo: uses all solutions known in Distributed systems

- Replication
  - Consistent hashing
- Version reconciliation
  - Vector clocks
- Replication
  - Sloppy quorum
- Handling temporary failures
  - Hinted handoff
- Recovering from permanent failures
  - Merkle trees
- Membership and failure detection
  - Gossip, anti-entropy
<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Eventual consistency, Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>‘Sloppy’ quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>
Partition Algorithm: Consistent hashing

- Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”.
- Advantage: incremental scalability
- Finger pointers to all nodes O(N) state
Node hashes to multiple points on the ring

- Uses Vnode, when a node is entered into the system it hashes to multiple points in the ring
- Multiple key ranges map to one node
- Node fails; lose a factor of the load
- Node rejoins, absorb load from all keys that map to this node
Replication at N different hosts

- Each data item is replicated at N hosts.
- Succ(k) + N-1 succ nodes
- “preference list”: The list of nodes that is responsible for storing a particular key.
Data Versioning

- Consistency model
- Eventual consistency
- A put() call may return to its caller before the update has been applied at all the replicas
- Data is immutable; create a new version
- A get() call may return many versions of the same object.

**Challenge**: an object having distinct version sub-histories, which the system will need to reconcile in the future.

**Solution**: uses vector clocks in order to capture causality between different versions of the same object.
Logical Clocks

- How to impose causality in a distributed system
- Solution: timestamp updates with *logical clocks* [Lamport]
  
  - Each site maintains a monotonically increasing clock value $LC$
  - Increment $LC$ on each new event
  - Send $LC$ along with any message sent
  - Receiver sets its $LC = \text{Recv}(LC) + 1$
  - Send happens-before receive
  - **Clock condition**: $e_1 < e_2$ implies that $LC(e_1) < LC(e_2)$
Logical clocks example

If $LC(e1) < LC(e2)$ does not imply causality
Vector Clocks

- Need to infer from clock values if two events are related (can replicas be reconciled?)
- In a system with $N$ nodes, each site keeps a vector timestamp $TS[N]$ as well as a logical clock $LC$.
  - $TS[j]$ at site $i$ is the most recent value of site $j$’s logical clock that site $i$ “heard about”.
  - $TS_i[j] = LC_i$; each site $i$ keeps its own $LC$ in $TS[i]$.
- 2. When site $i$ generates a new event, it increments its logical clock.
  - $TS_i[i] = TS_i[i] + 1$
- 3. A site $r$ observing an event (e.g., receiving a message) from site $s$ sets its $TS_r$ to the pairwise maximum of $TS_s$ and $TS_r$.
  - For each site $i$, $TS_r[i] = \text{MAX}(TS_r[i], TS_s[i])$
Vector clocks & ordering

- Use the vector timestamp at different sites
- $e_1$ happened-before $e_2$ if and only if $TS(e_2)$ dominates $TS(e_1)$
  - $e_1 < e_2$ iff $TS(e_1)[i] \leq TS(e_2)[i]$ for each site $i$
  - “Every event or update visible when $e_1$ occurred was also visible when $e_2$ occurred.
  - If Dominates $\Rightarrow$ replace old version
- if two events are concurrent, one does not dominate the other then
  - If does not dominate, then Reconcile
Vector Clocks: Example

X
(1, 0, 0)
D1
(2, 0, 0)
D2
Y
(2, 0, 0)(2, 1, 0)
D3
Z
(2, 0, 0)
D4
(2, 0, 1)
D5
(2, 0, 1)
(2, 1, 0)
D6
(2, 0, 1)
D7
(2, 0, 1, 0)
(2, 1, 0, 0)
Vector clock example

D1 ([Sx,1])

D2 ([Sx,2])

D3 ([Sx,2],[Sy,1])

D4 ([Sx,2],[Sz,1])

D5 ([Sx,3],[Sy,1],[Sz,1])
Executing `put()` and `get()`

- Any node can process `put()`, `get()`
- Directed to any node by LB
  - Application agnostic of dynamo
- Directed to coordinator (dynamo aware)
  - Client knows how to map key to node
‘Sloppy’ Quorum

Traditional quorum system:
- R/W: the minimum number of nodes that must participate in a successful read/write operation.
- Setting R + W > N yields a quorum-like system.
- At least one intersecting node between R & W

‘Sloppy’ quorum:
- Set R + W <= N
- More chance to skip slower nodes
Handling failures: Hinted handoff

- A node D is down
- D is required for quorum
- Move Updates to D’
- D’ checks for D until D is up again
- D’ synchs with D
- D’ deletes data that was originally meant for D
Handling permanent failures

- Replica synchronization
- Anti-entropy
- Each node selects a random peer
  - {
    - Get(peer, peerstate)
    - New_state = mergestate(peerstate, mystate)
    - Put(peer, new_state)
    - My_state = new_state
  - }
Merkle Tree

- Hash tree where leaves are the hashes of the values of individual keys
- Parent nodes are hashes of their children
- Each branch can be checked independently without downloading entire tree
Merkle Tree

Node
Crashes

Replica
**Merkle Tree**

Compare root

**Comes back!**

**Replica modified!**
Merkle Tree

Compare left subtree
Merkle Tree

Compare right sub tree
Merkle Tree

Leaf Level mismatch
Evaluation

(hourly plot of latencies during our peak season in Dec. 2006)
Trading latency & durability
Load balance

- **Fraction of nodes out-of-balance**
- **Request Load (scaled down by a constant)**

The graph shows the fraction of nodes out-of-balance and the request load over time. The request load decreases as the fraction of out-of-balance nodes decreases.
Number of versions

- 99.94% time only single version
  - Updates well coordinated
- Dynamo aware client does better job at LB
  - Latency halved compared to a generic LB
Conclusions

- Uses So many concepts from Distributed systems design
- Amazing that they have been able to put together a complex working system
- Can a simpler design worked just the same
- Difficult to get all concepts related to consistency working in the same system