Data-Intensive Information Processing Applications — Session #7

Web-Scale Databases

A database perspective on the cloud

Andreas Thor
University of Maryland & University of Leipzig

Thursday, March 31, 2011

Agenda

- Cloud $\supset$ Hadoop/MapReduce
  1. Object Storage (Amazon S3)
  2. Cloud Database (BigTable/Hbase)
- Add Database stuff to Hadoop/MapReduce
  3. SQL to MapReduce
  4. Data Warehouse on top of Hadoop/MapReduce (Hive)

DeWitt & Stonebraker. MapReduce: A major step backwards.
http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html.

Stonebraker et. al: MapReduce and parallel DBMSs: friends or foes? CACM 2010

Storage Services

[Amazon] DeCandia et al. Dynamo: Amazon's Highly Available Key-value Store. SOSP'07

Object Storage Services

- Service for storing objects (binary data) in the cloud
  - Upload , storage on multiple nodes, download
- Simple structure
  - Buckets: simple (flat) containers
  - Objects: arbitrary data (e.g., files), arbitrary size
  - Authentication, access rights
- Simple API
  - HTTP requests (REST-ful API): PUT, GET, DELETE
  - Used by applications, e.g., DropBox (online backup & sync tool)
- Performance
  - fast, scalable, high availability
- Costs
  - “pay as you go”: #requests, data size, upload/download size
- Example: Microsoft Azure Storage, Amazon Simple Storage Service (S3)

Problems

- Concurrent user access
  - YouTube videos, collaborative work on documents,
- Problem: concurrent writes
  - Conditional Updates: “IF current version =X THEN Update”
  - Node-based versioning

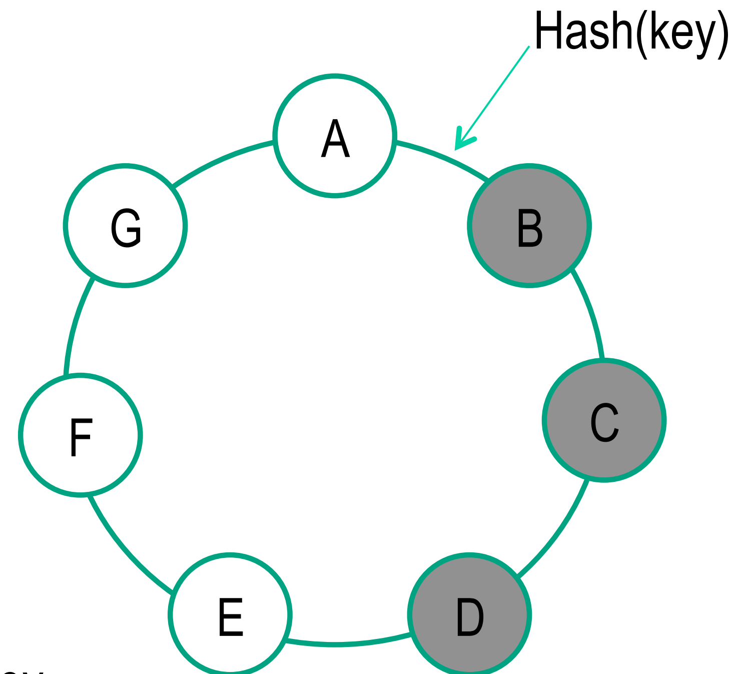
- Data copies on multiple nodes
  - Reliability: Redundancy against node outage
  - Read performance: Multiple clients can read different copies in parallel (locality)
- Problem: replica synchronization
  - Strong Consistency: Any read access will return the updated version
  - Eventual Consistency: All accesses will eventually return the updated version

Amazon S3/Dynamo: Overview

- Amazon S3 is based on Amazon Dynamo
- Distributed, scalable key-value store
  - designed for “small data objects” (1MB / key)
- Characteristics
  - high availability
  - low latency
- Eventually consistent data store
  - Write access always possible
  - relaxed consistency in favor of availability
- Performance SLA (Service Level Agreement)
  - “response within 300ms for 99.9% of requests for peak client load of 500 requests per second”
- P2P-like structure
  - no master nodes, all nodes have the same functionality
  - each node is aware of data at peers

Amazon Dynamo: Partitioning

- Each node is assigned a position in a ring
  - Position= random value of a hash function
- Node assignment
  - Compute hash value of key
  - Choose next N nodes on ring (clock-wise)
  - Example: Hash(key) between A and B
→ for N=3: nodes B, C, and D
  - Performant node insert / delete / remove because neighboring nodes affected only
- Preference list
  - List of N nodes that are assigned for a given key
  - each node has a preference list for all keys
- Consistent hashing
  - appropriate hash function needed for data locality and load balancing



Amazon Dynamo: Data access

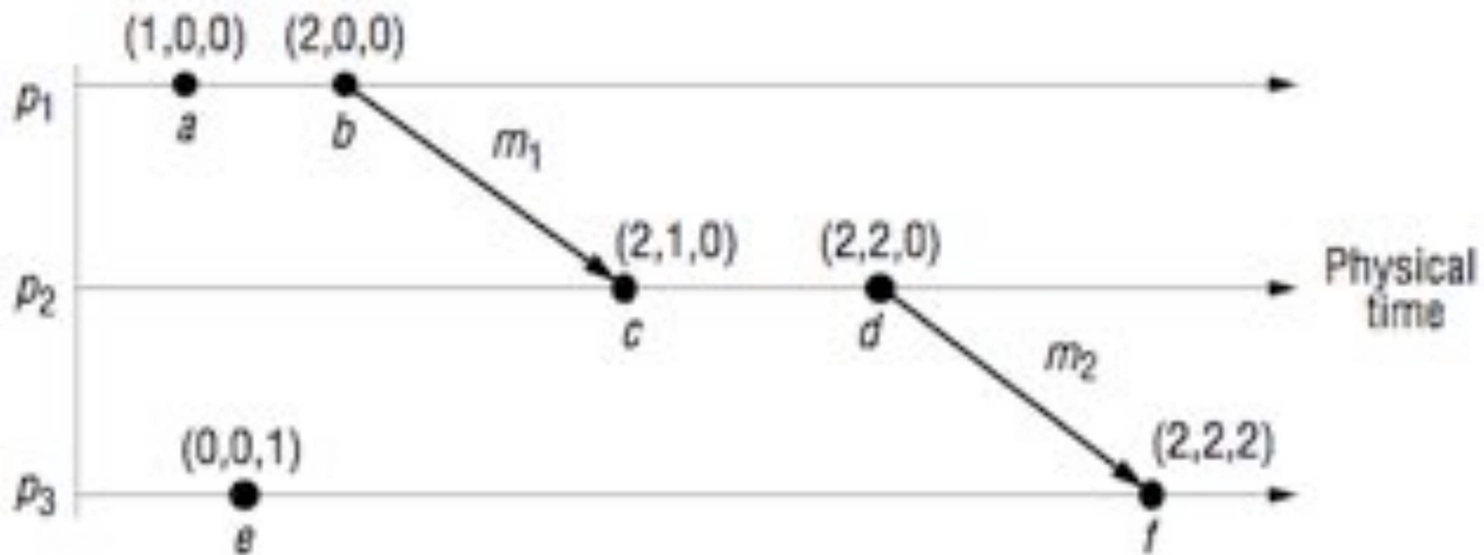
- Key value store interface
  - Primary key access, no complex queries
  - Request to any node of the ring
  - Request will be forwarded to one (first) node of the key's preference list
- Put (Key, Context, Object)
  - Coordinator creates vector clock (versioning) based on request's context
  - Coordinator writes object + vector clock
  - Replication
    - Write requests to N-1 other nodes out of the preference list
    - Success, if (at least) W-1 nodes succeed
    - asynchronous replica updates for W<N → consistency problems
- Get (Key)
  - Read request to N nodes of the preference list
  - Return responses from R nodes → may contain multiple versions; list of (object, context) pairs

Amazon Dynamo: Replication

- Read/Write Quorum
  - R/W = minimal number of replica nodes that must be synchronized for successful read/write operation
  - Application can adjust (N,R,W) to meet needs for performance, availability, and durability
- Consistency if $R + W > N$
  - User/application-controlled conflict resolution for different versions
- Variants
  - Read-optimized: R=1, W=N
  - Write-optimized: R=N, W=1
  - Default: (3,2,2)

Amazon Dynamo: Versioning

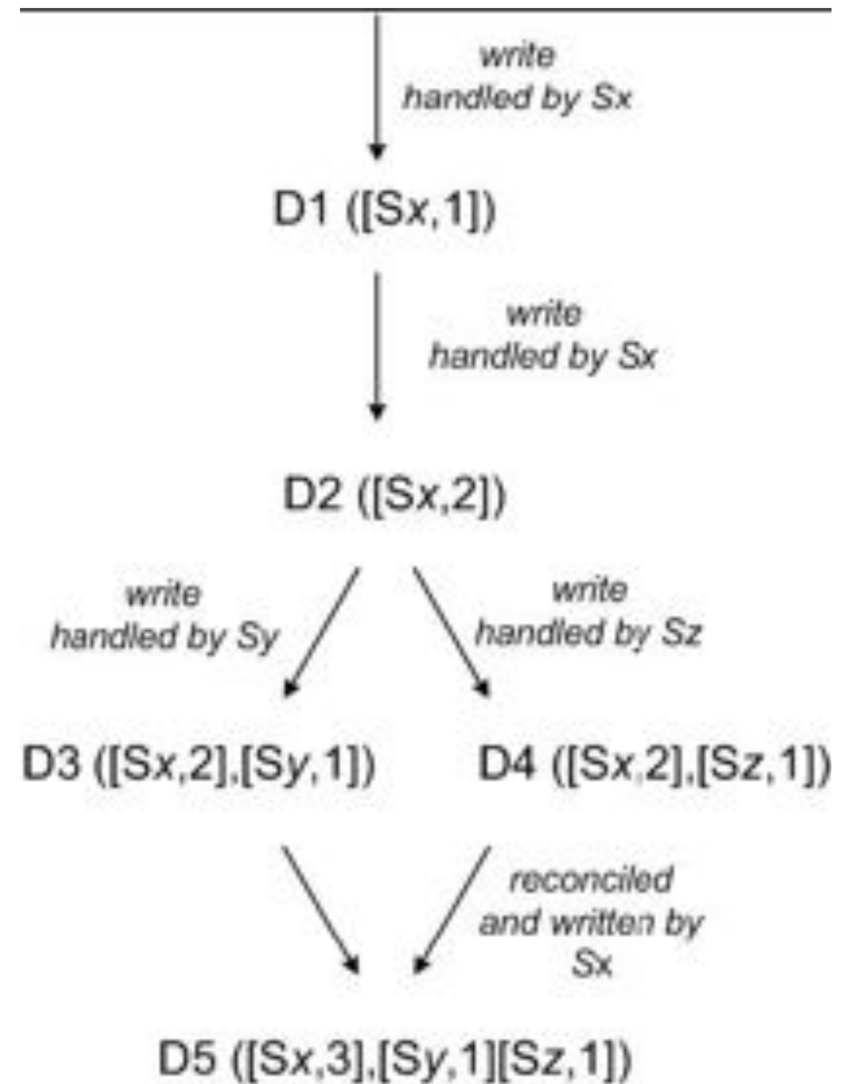
- Example of object versioning



- "Vector Clocks" represent dependencies between different versions of the same object → reconcile multiple versions
  - version counter per replica node,
    e.g., **D ([S_x, 1])** for object D, node S_x, version 1
  - Vector clock: list of `(node, counter)` pairs to indicate available object versions

Amazon Dynamo: Versioning (2)

- Vector Clocks to determine dependencies between 2 object versions
  - Counters of 1st vector clock ≤ all counters of 2nd vector clock→ 1st version is (direct) ancestor and can be deleted
  - otherwise: conflict resolution
- Read returns all known versions incl. vector clocks
  - subsequent update merges all version
- Application determines conflict resolution
  - vector clocks part of get/put requests

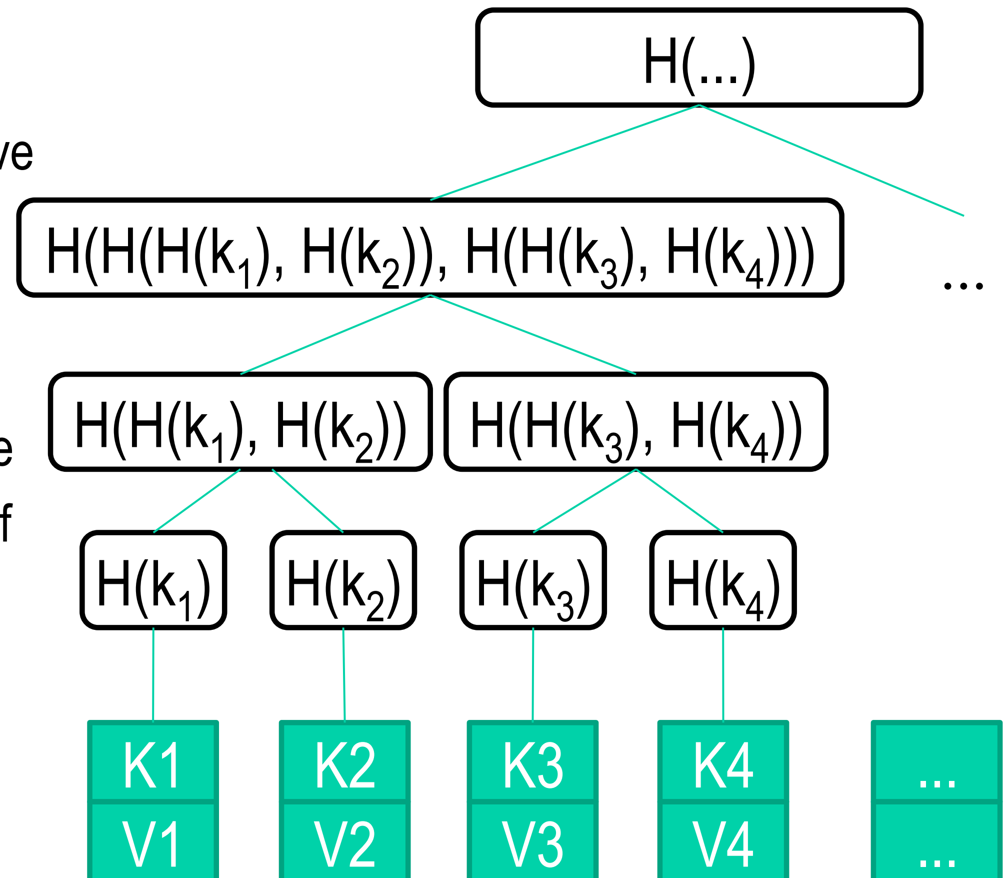


Amazon Dynamo: Temporary failures

- Temporary node failure should be transparent to the user
- Sloppy Quorum (N, R, W)
  - All operations performed on first N healthy nodes
  - still “writable” if replica not available (e.g., W=N)
- Hinted Handoff
  - If node is unavailable, replication request is sent to another node (“hinted replica”)
  - Background job: When original node has recovered, send hinted replica to original node

Amazon Dynamo: Replica synchronization

- Hash-Tree (Merkle Tree) for key range
  - Leafs = hash value of key value
  - Parents = hash value of respective child node values
- Advantages
  - Efficient check if two replicas are identical = roots have same value
  - Efficient recursive identification of out-of-sync sub trees
- Disadvantages
  - Computational costs during repartitioning (e.g., new nodes)



Amazon Dynamo: Techniques (Summary)

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Scalability
High availability of writes	Vector Clocks + conflict resolution during reads	Versioning independent from update frequency
Temporary node failure	Sloppy Quorum and Hinted Handoff	High availability; reliable
Recovering	Hash Tree (Merkle Tree)	Efficient background synchronization of replicas

- Additional techniques
  - Gossip protocol for P2P network (new nodes, failure identification,)

Amazon S3/Dynamo vs. Azure Storage

	Amazon Dynamo	Azure Storage
Partitioning	Hash function	Object name
Dynamically extensible	+	+
Routing	P2P	hierarchical
Replication	asynchronous	synchronous
Consistency	Eventual Consistency	Strong Consistency
Handling concurrent writes	during read; multiple versions with vector clock	during write; conditional updates
Performance	Adjustable by read/write quorum	Read optimized; CDN (eventual consistency)

Web (nonSQL) Databases

[BigTable] Chang et al. Bigtable: A Distributed Storage System for Structured Data. OSDI'06
[HBase] http://hadoop.apache.org/hbase/

Web Database: Usage scenario

- Web table
  - Table contains crawled web pages incl. date, time, ...
  - Key: web page URL
  - millions/billions of pages
- Random access
  - Crawler adds / updates web pages
  - Search engine delivers cached version of web pages
- Batch processing
  - Build search engine index

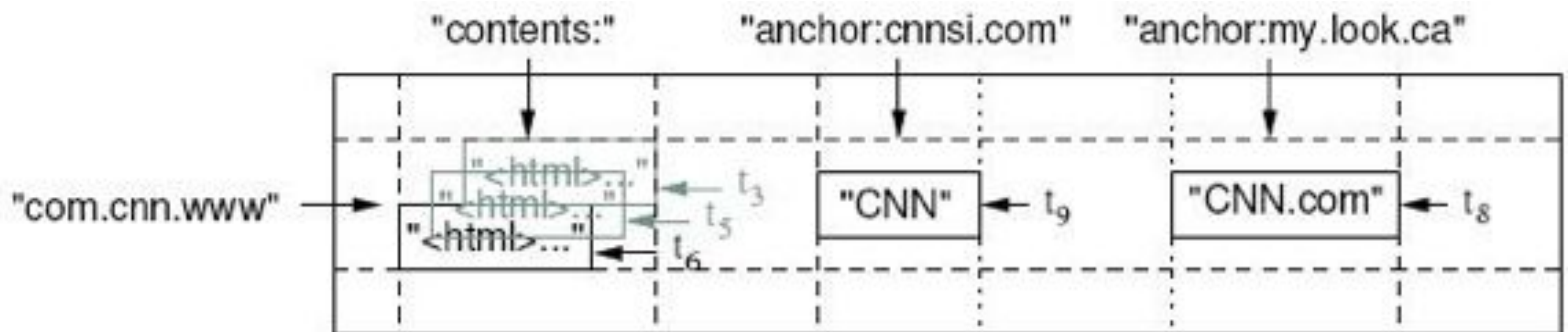
- Dynamic web applications (e.g., Facebook) need fast random access to (semi-) structured data

Google's BigTable

- Distributed data storage system
  - column-oriented key-value store
  - multi-dimensional
  - Versioning
  - High availability
  - High performance
- Goals
  - Billions of rows, millions of columns, thousands of version
  - Real-time read/write random access
  - Large data (PB)
  - linear scalability with the number of nodes
- Idea / techniques
  - Architecture allows efficient but simple data access method
  - no additional overhead (e.g., ACID)
- HBase is Hadoop implementation of BigTable

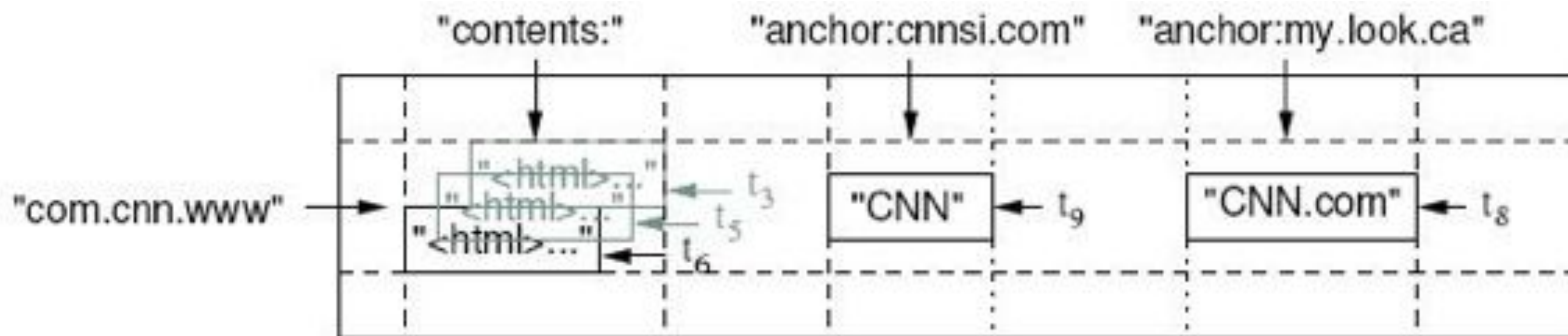
Data model

- Distributed, multi-dimensional, sorted map
  (row:string, column:string, time:int64) → string
  - Keys for row and columns
  - time stamp
  - Arbitrary data (Strings / Byte strings)
- Rows
  - Read and write operations are atomic per row only
  - Data stored in (lexicographical) order of row keys



Data model (2)

- Columns
  - can be added dynamically at run-time
- Column families
  - Group together n similar columns
  - column key = family: qualifier
  - Disk/memory storage w.r.t. to column families (columns of the same family are stored „close together")
- Time stamp
  - different versions of data per cell
  - garbage collection of older versions („keep t versions only")



Data model (3)

- Conceptual (alternative)

Row Key	Time Stamp	Column Contents	Column Family Anchor	
“com.cnn.www”	T9		Anchor:cnnsi.com	CNN
	T8		Anchor:my.look.ca	CNN.COM
	T6	“<html>.. “		
	T5	“<html>.. “		

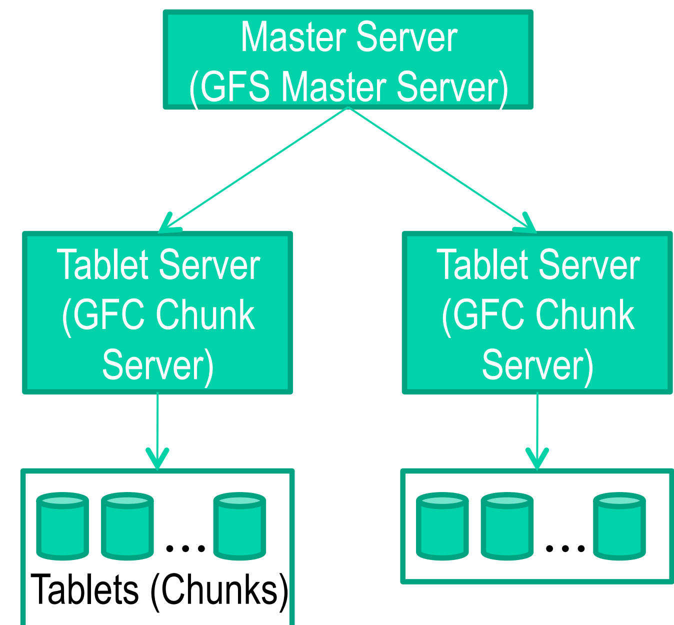
- Physical storage

Row Key	Time Stamp	Contents
com.cnn.www	T6	“<html>..”
	T5	“<html>..”

Row Key	Time Stamp	Anchor	
com.cnn.www	T9	Anchor:cnnsi.com	CNN
	T5	Anchor:my.look.ca	CNN.COM

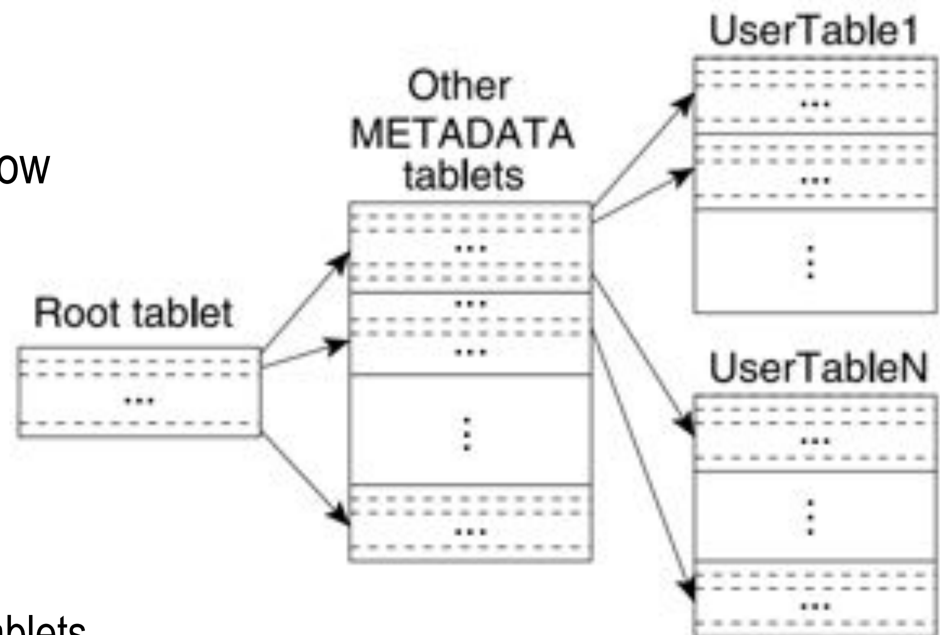
Architecture

- Data partitioning
  - Rows sorted by key
  - Horizontal table partitioning into tablets
  - Tablet distribution across multiple tablet servers
- Master Server
  - Assignment: Tablet <-> Tablet Server
  - Add/delete tablet servers
  - Load balancing for tablet servers
- Tablet Server
  - Manages 10-1,000 tablets
  - Realizes read and write access
  - Tablet split if tablet too large (100-200MB)
- Client
  - Communication with tablet server for reading / writing

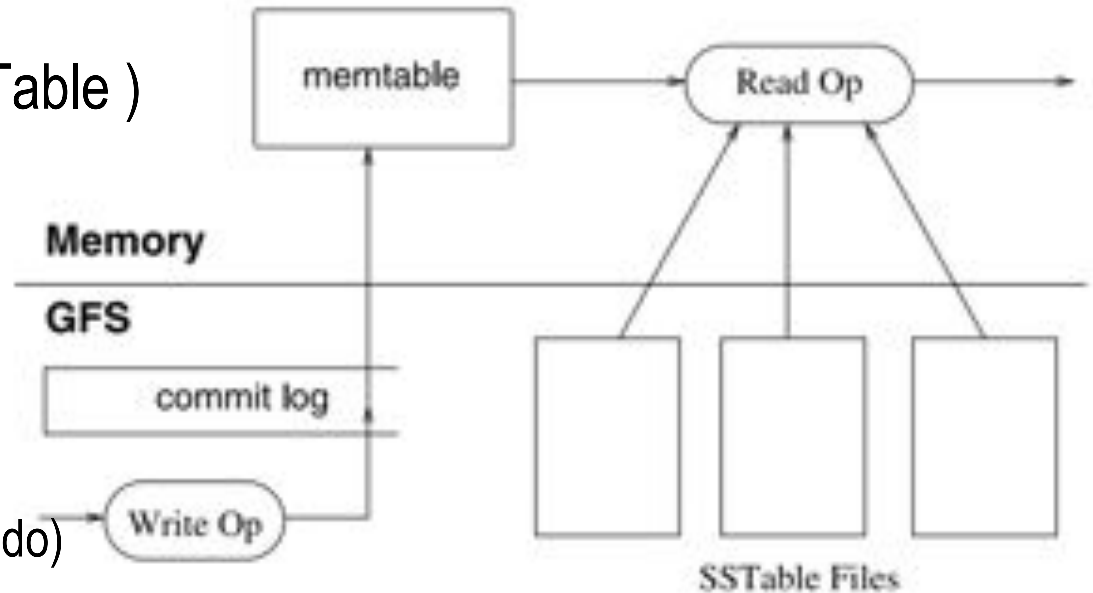


Tablet Location

- 2-level catalog management with Root and METADATA table
- Root table
  - Links to all tablets of a METADATA table
  - Stored in 1 Tablet (never split)
- METADATA table
  - Links to all tablets (of user tables)
  - Identifier: table name + key of last row
  - Table are sorted by key
- Address space
  - Entry size: 1KB
  - Tablet size: 128MB
  - Addressable tablets:
    - METADATA: 128MB / 1KB = 2^{17} tablets
    - User Table: $2^{17} \times 2^{17} = 2^{34}$ tablets
  - Size of all user tablets: $2^{34} \times 128$ MB = 2^{41} MB = 2 million TB



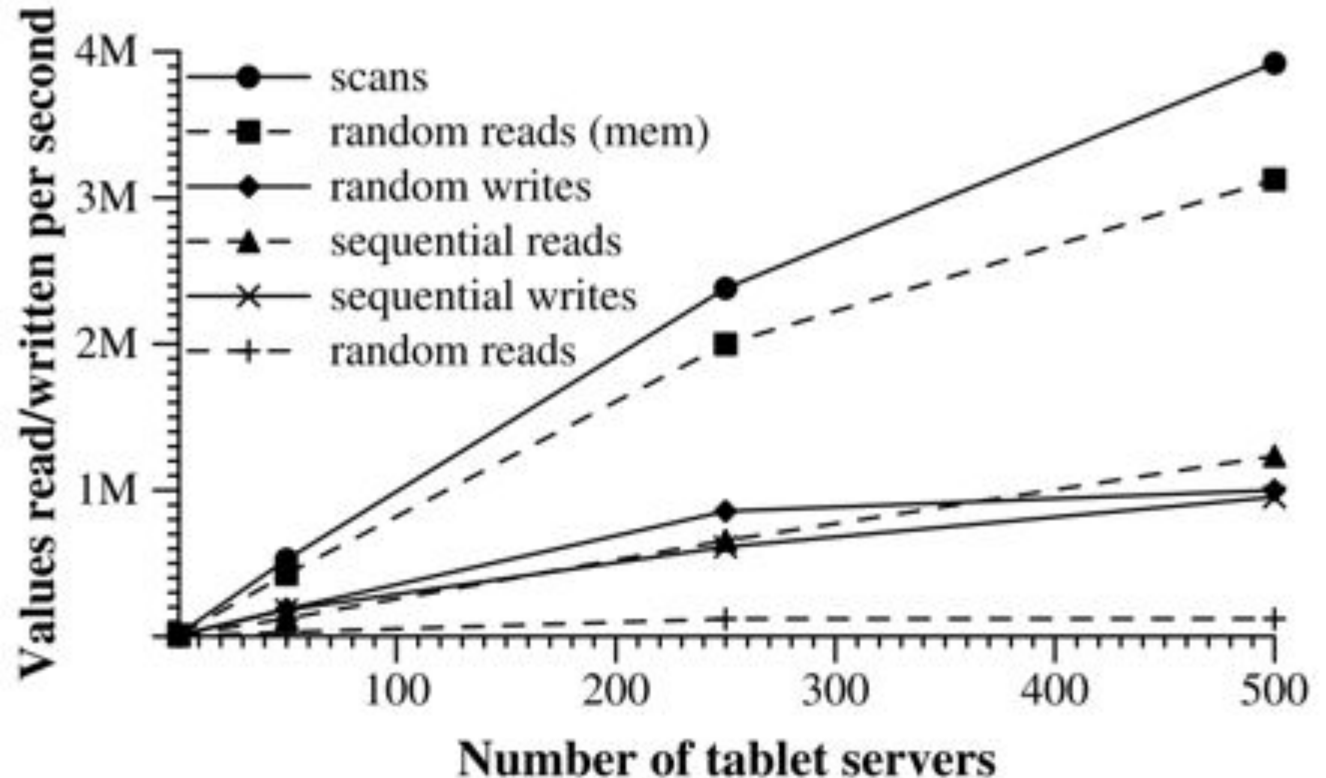
Tablet: Read and write access



- SSTable File (Sorted String Table)
  - Immutable sorted map
  - Bloom Filter to check if SSTable contains data for row+column
- Write access
  - Write to transaction Log (for redo)
  - Write to MemTable (RAM)
- Asynchronous: Compaction
  - Minor: Copy data from MemTable to SSTable (and delete from log)
  - Merge: Merge MemTable and SSTable(s) to new SSTable
  - Major: Remove deleted data (=merge to one SSTable)
- Read access
  - Read from MemTable and SSTables to find data

Performance

- #Read/WriteOps per second for 1000Byte
- Good scalability for up to 250 tablet servers



- Write is faster than read
  - Commit-Log is append only; Read requires access to MemTable + SSTable
- Random reads slowest
  - Access (all) SSTables
- Scanning and sequential reads are more efficient
  - Make use of sorted keys

Bigtable vs. RDBMS

	BigTable / HBase	RDBMS
Assumption	(hardware) failures are prevalent	(hardware) failures are rare
Replication	built-in	external
Normalization	unnormalized data (wide, sparse tables)	normalized data (3NF) (compact, redundant free tables)
Query	key-based access: point and range	SQL
Scalability	linear, unlimited	limited (due to ACID, foreign keys, views, trigger, ...)
Index	primary key	primary key + secondary indexes
Transactions	-	+
Atomicity	row level	transaction level
Consistency	No integrity constraints, no referential integrity	Integrity constraints and referential integrity
Isolated execution	-	+
Durability	+	+

MapReduce and SQL

[CouchDB] http://couchdb.apache.org/
[Data] http://labs.mudynamics.com/wp-content/uploads/2009/04/icouch.html

Query transformation

- (manual) rewrite from SQL to MapReduce

- Example: CouchDB
- Document-oriented data store
  - no schema
  - JSON format
  - simple versioning concept
- Query/view definition
  - specify map and reduce function in Javascript (or other language)

Example data

- Conceptional: nested table

id	name	time	user	camera	info			tags
					width	height	size	
1	fish.jpg	17:46	bob	nikon	100	200	12345	[tuna, shark]
2	trees.jpg	17:57	john	canon	30	250	32091	[oak]
3	snow.png	17:56	john	canon	64	64	1253	[tahoe, powder]
4	hawaii.png	17:59	john	nikon	128	64	92834	[maui, tuna]
5	hawaii.gif	17:58	bob	canon	320	128	49287	[maui]
6	island.gif	17:43	zztop	nikon	640	480	50398	[maui]

- Internal representation as document set (JSON format)

```
{“_id”:”1”, "name":"fish.jpg",”time":”17:46","user":"bob“,"camera":"nikon",
          "info":{"width":100,"height":200,"size":12345},"tags":["tuna","shark"]}
{“_id”:”2”, "name":"trees.jpg",“time":”17:57”,"user":"john”,"camera":"canon",
          "info":{"width":30,"height":250,"size":32091},"tags":["oak"]}
....
```

Selection

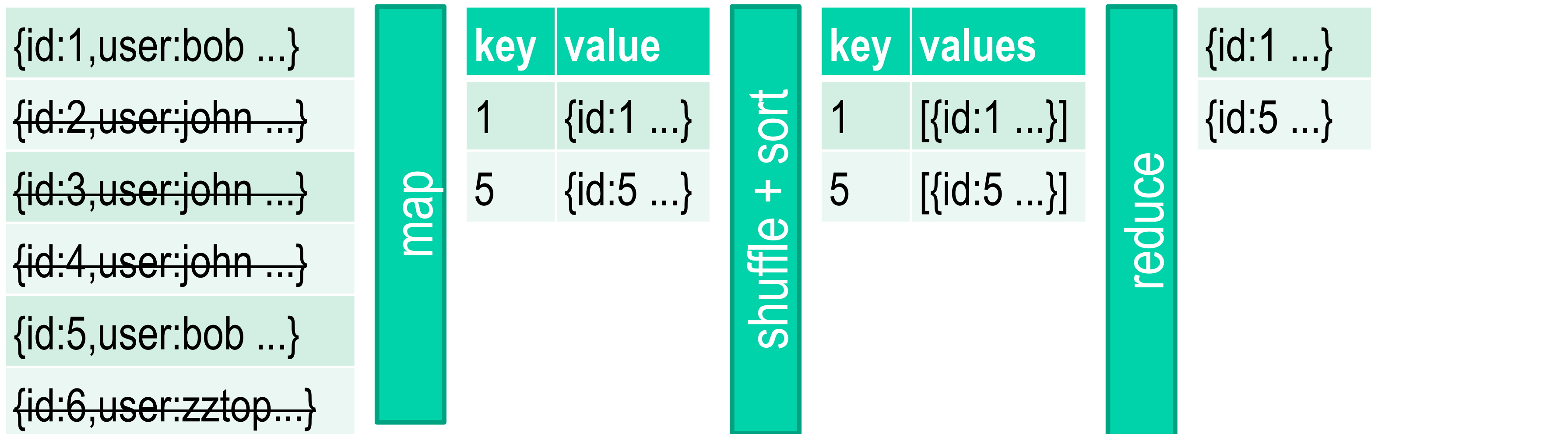
- Selection = attribute value condition
  - SQL: ... WHERE attr = “xy”
- Map
  - check condition using IF statement
  - return selected document
- Reduce
  - id function
- Example
  - SQL: SELECT * FROM table WHERE user = “bob”

id	name	time	user	camera	info			tags
					width	height	size	
1	fish.jpg	17:46	bob	nikon	100	200	12345	[tuna, shark]
5	hawaii.gif	17:58	bob	canon	320	128	49287	[maui]

Selection: Example

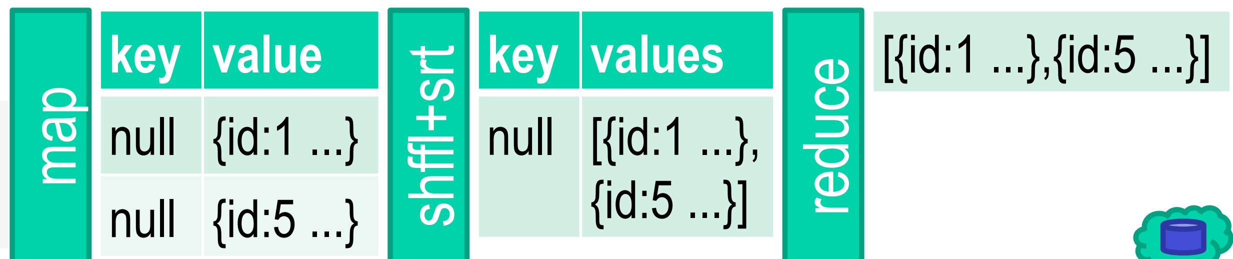
```
map
function (doc) {
   if (doc.user == "bob")
      emit (doc.id, doc);
}
```

```
reduce
function (key, values) {
   return values[0];
}
```



Alternative

```
emit (null, doc);
return values;
```



Projection

- Projection = restrict set of attributes
  - SQL: SELECT Attr1, Attr2 FROM ...
- Map
  - create new (“restricted”) document
- Reduce
  - id function
- Duplicate removal
  - map: key = projected attributes
  - reduce: return first value
- Example
  - SQL: SELECT (DISTINCT) user FROM table

user
bob
john
john
john
bob
zztop

user
bob
john
zztop

Projection: Example (w/o duplicate removal)

map
```
function (doc) {
  emit(doc.id,{"user":doc.user});
}
```

reduce
```
function (key, values) {
  return values[0];
}
```

input		key	value		key	value		output
{id:1,user:bob ...}	map	1	{user:bob }	shuffle + sort	1	[{user:bob }]	reduce	{user:bob }
{id:2,user:john ...}		2	{user:john}		2	[{user:john}]		{user:john}
{id:3,user:john ...}		3	{user:john}		3	[{user:john}]		{user:john}
{id:4,user:john ...}		4	{user:john}		4	[{user:john}]		{user:john}
{id:5,user:bob ...}		5	{user:bob}		5	[{user:bob}]		{user:bob}
{id:6,user:zztop...}		6	{user:zztop}		6	[{user:zztop}]		{user:zztop}

Projection: Example (w/ duplicate removal)

map
```
function (doc) {
  emit(doc.user,{"user":doc.user});
}
```

reduce
```
function (key, values) {
  return values[0];
}
```

input		key	value		key	value		output
{id:1,user:bob ...}	map	bob	{user:bob }	shuffle + sort	bob	[{user:bob }, {user:bob }]	reduce	{user:bob }
{id:2,user:john ...}		john	{user:john}		john	[{user:john}, {user:john}, {user:john}]		{user:john}
{id:3,user:john ...}		john	{user:john}		zztop	[{user:zztop}]		{user:zztop}
{id:4,user:john ...}		john	{user:john}					
{id:5,user:bob ...}		bob	{user:bob}					
{id:6,user:zztop...}		zztop	{user:zztop}					

Grouping and aggregate functions

- Grouping
  - Divides records into groups based on shared attribute values
  - Produces one record (row) per group
  - Aggregate functions to compute aggregated values (per group), e.g., SUM
- Map
  - Key = group attribute values
- Reduce
  - Return first key value
  - Optional: Apply aggregate function(s)
- Example
  - SELECT camera, AVG(info.size) as avgsize
    FROM Table
    GROUP BY camera

camera	avgsize
canon	27543.3
nikon	51859

Grouping and aggregate functions: Example

map
```
function (doc) {
  emit(doc.camera,
   doc.info.size);
}
```

reduce
```
function (key, values) {
  sum = 0;
  for (i=0; i<values.length; i++) {
    sum = sum + values[i];
  }
  return {"camera":keys,
          "avgsize":sum/values.length};
}
```

{id:1,user:bob ...}
{id:2,user:john ...}
{id:3,user:john ...}
{id:4,user:john ...}
{id:5,user:bob ...}
{id:6,user:zztop...}

map

key	value
nikon	12345
canon	32091
canon	1253
nikon	92834
canon	49287
nikon	50398

shuffle + sort

key	value
canon	[32091, 1253, 49287]
nikon	[12345, 92834, 50398]

reduce

{camera:canon, avgsize: 27543.3}
{camera:nikon, avgsize: 51859}

Equi-join + multi-valued attribute

- Equi-join = combine records from two relations based on attribute equality
  - SQL: ... WHERE Tab1.Attr1 = Tab2.Attr2
- Multi-valued attribute in 1NF
  - 1-to-many, many-to-many relationships
  - equi-joins needed
- Map
  - Key = join attribute value
- Reduce
  - Iteration over all value pairs
- Example (SQL)
  - SELECT Tab1.name AS name1, Tab2.name AS name2
    FROM table AS Tab1, table AS Tab2,
    tagtab AS Tag1, tagtab AS Tag2
    WHERE Tag1.id=Tab1.id AND Tag2.id=Tab2.id
    AND Tag1.tag = Tag2.tag
    AND Tab1.name < Tab2.name

id	tag
1	tuna
1	shark
4	maui
4	tuna
5	maui

name1	name2
hawaii.png	island.gif
hawaii.gif	hawaii.png
hawaii.gif	island.gif
fish.jpg	hawaii.png

Equi join + multi-valued attribute: Example (1)

map
```
function (doc) {
  for (i=0; i<doc.tags.length; i++) {
  emit (doc.tags[i], doc.name);
}
```

reduce
```
function (key, values) {
  var result = new Array();
  for (i=0; i<values.length; i++) {
    for (k=0; k<values.length; k++) {
      if (values[i]<values[k] {
        result.push ({name1:values[i], name2:values[k]});
      }
    }
  }
  return result;
}
```

Equi join + multi-valued attribute: Example (2)

{id:1,...}
{id:2,...}
{id:3,...}
{id:4,...}
{id:5,...}
{id:6,...}

map

key	value
tuna	fish.jpg
shark	fish.jpg
oak	tree.jpg
tahoe	snow.png
powder	snow.png
maui	hawaii.png
tuna	hawaii.png
maui	hawaii.gif
maui	island.gif

shuffle + sort

key	value
maui	[hawaii.png, hawaii.gif, island.gif]
oak	[tree.jpg]
power	[snow.png]
shark	[fish.jpg]
tahoe	[snow.png]
tuna	[fish.jpg, hawaii.png]

reduce

[{name1:hawaii.png, name2: island.gif}, {name1:hawaii.gif, name2:hawaii.png}, {name1:hawaii.gif, name2:island.gif}]
[]
[]
[]
[]
[{name1:fish.jpg, name2:hawaii.png}]

MapReduce and Data Warehouses

[Hive] Thusoo et.al.: Hive-a petabyte scale data warehouse using hadoop. ICDE 2010
[HiveUrl] http://hadoop.apache.org/hive/
[Hive1] http://www.slideshare.net/zshao/hive-data-warehousing-analytics-on-hadoop-presentation
[Hive2] http://www.slideshare.net/ragho/hive-user-meeting-august-2009-facebook
[Hive3] http://www.slideshare.net/jsichi/hive-evolution-apachecon-2010

Hadoop/MR vs. Parallel DBS

- Hadoop/MR advantages
  - Scalability, fault tolerance
  - configuration effort, costs
  - no initial data loading
- Parallel DBS advantages
  - Declarative query language
  - Queries run faster by order of magnitude
  - Support for compressed data
  - Random access
- Common use cases MapReduce
  - ETL
  - Data mining, data clustering
  - Analysis of semi-structured data (e.g., web log files)
  - Ad-hoc data analysis

Data analysis: Facebook

- Facebook
  - 4TB compressed data per day
  - 135TB compressed data are analyzed per day
- Aggregations
  - #clicks/page views per day/month/...
- Ad-hoc analysis
  - How many uploaded pictures per county / state on New Year's Eve?
- Data Mining
  - User profiles based on attributes (#pageviews, #sessions, time, ...)
- Spam detection
  - (suspicious) frequent patterns in user generated content
- Analysis / optimization of online advertisement
  - #AdClicks per user (type) ...

Hive

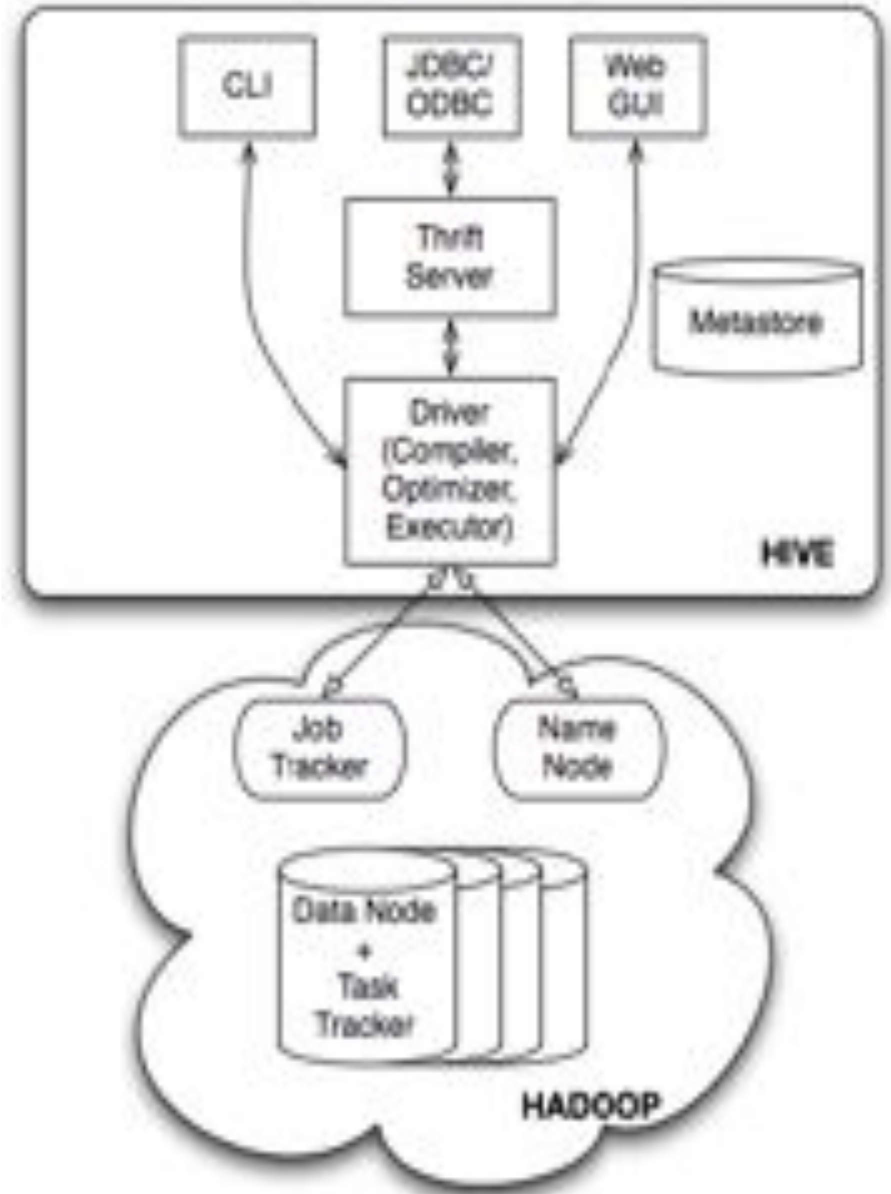
- Data Warehouse based on Hadoop
- Hive = MapReduce + SQL
  - SQL is simple and widely-used
  - MapReduce scalability
- Automatic translation SQL to MapReduce necessary
  - Programs hard to maintain, almost no reuse
  - Difficult for non experts
  - Limited expressiveness, e.g., long code (development time!) to realize simple count/ average queries

Hive: Overview

- Management and analysis of structured data using Hadoop
  - no OLTP database, high latency
- File-based data storage (HDFS)
  - metadata for mapping files to tables
  - complex data types (e.g., list, map)
  - direct file access, different data formats
- HiveQL queries are executed using MapReduce
  - include scripts (e.g., written in Python) in queries
  - metadata, e.g., for optimizing joins
- Scalability and fault tolerance
  - HDFS + MapReduce
- Extensibility
  - User-Defined Table-Generating Functions (UDTF)
  - User-Defined Aggregate Functions (UDAF)

Hive: Architecture

- Metastore
  - Tables, columns (type)
  - Location, partitions
  - Information on (de)serialization
- CLI / Web-GUI
  - Browse metastore
  - Send queries
- Thrift
  - Cross-language Service → HiveQL
- Compiler + Optimizer
  - Query optimization and translation of HiveQL query to DAG of MapReduce jobs
- Executor
  - Execute MR-jobs of DAG



Hive: Data type & data access

- Data types
  - simple and composite data types
  - list, map
- Flexible (de)serialization of tables
  - multiple (user-defined) format, e.g. XML, JSON, CSV
  - multiple "storage engines", e.g., file
- Advantages
  - no initial data loading into data warehouse (no data replication!)
  - no data transformation to relational model but direct file access
- Disadvantages
  - no pre-processing, e.g., indexing
  - always full (file) table scan necessary

Hive: Tables, partitions, and files

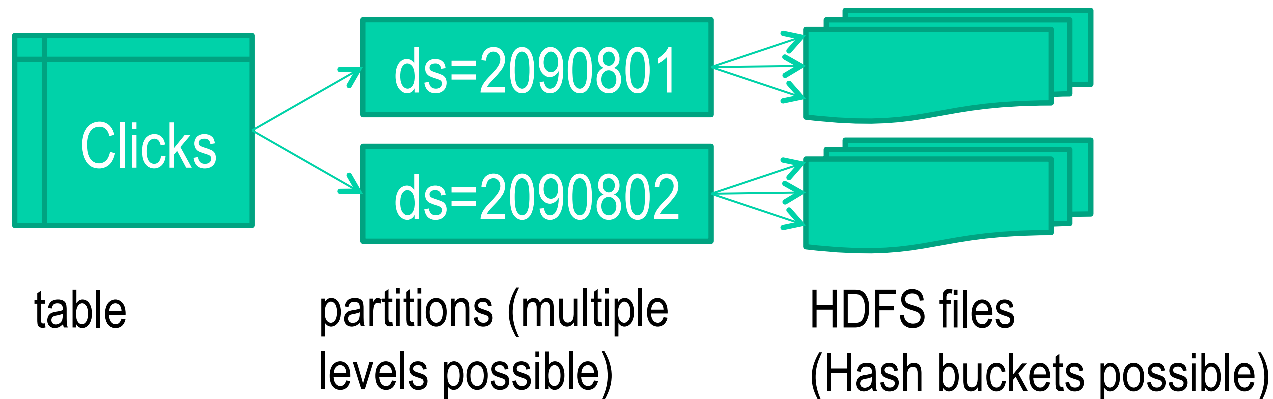
- Table links to existing file(s) in HDFS
  - Table has corresponding HDFS directory: `/wh/pvs`
  - Definition of columns for data partitioning

    `/wh/pvs/ds=20090801/ctry=US`

    `/wh/pvs/ds=20090801/ctry=CA`
  - Bucketing: Split data of a directory based on hash value

    `/wh/pvs/ds=20090801/ctry=US/part-00000` …

    `/wh/pvs/ds=20090801/ctry=US/part-00020`



Hive: Table

- Create

```
CREATE EXTERNAL TABLE pvs
(userid int, pageid int, ds string, stry string)
PARTITIONED ON(ds string, ctry string)
STORED AS textfile
LOCATION '/path/to/existing/file'
```

- Load

```
status_updates
(user_id int, status string, ds string)
LOAD DATA LOCAL
INPATH '/logs/status_updates'
INTO TABLE status_updates
PARTITION (ds='2009-03-20')
```

Hive-QL

- Similar to SQL
  - Selection, projection, equi-join, union, sub-queries, group by, aggregate functions
  - Sort by vs. order by
- Extend queries by
  - MapReduce scripts
  - UDF, may operate on complex data structures (lists, map)

```
FROM (
   FROM pv_users
   SELECT TRANSFORM(pv_users.userid, pv_users.date)
   USING 'map_script'
   AS(dt, uid)
   CLUSTER BY(dt)
) map
INSERT INTO TABLE pv_users_reduced
SELECT TRANSFORM(map.dt, map.uid)
USING 'reduce_script'
AS (date, count);
```

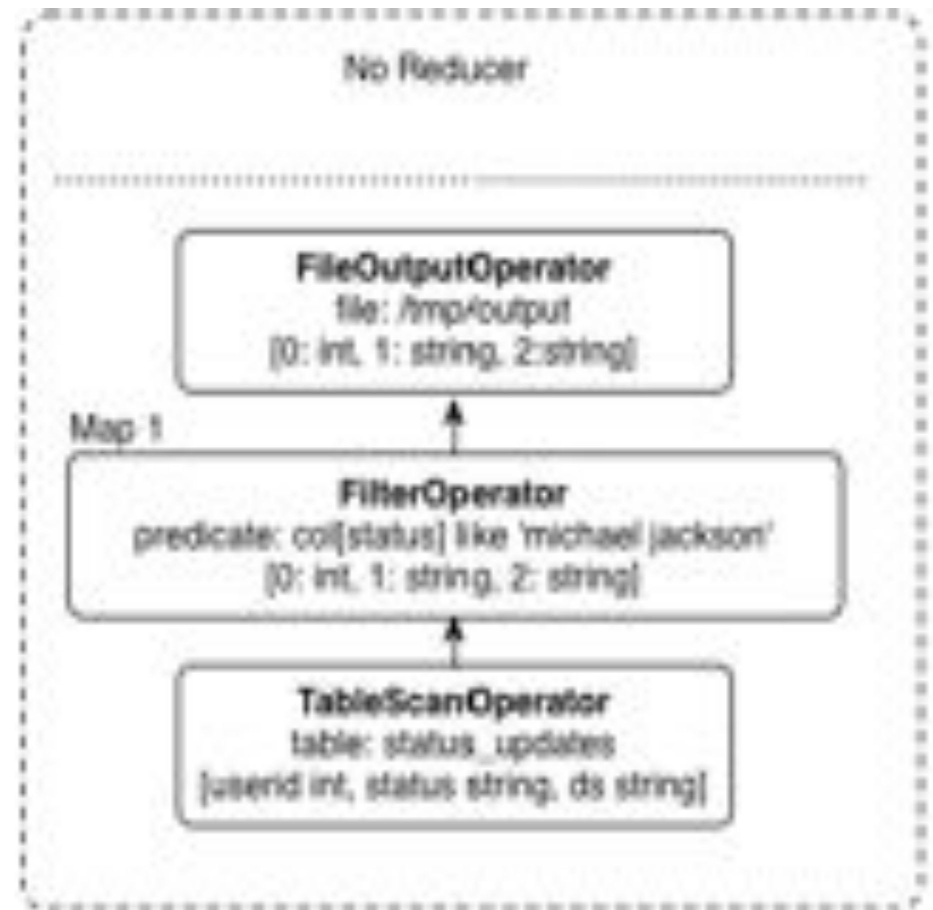
Hive-QL: Query transformation

- Hive-QL query is transformed into DAG (directed acyclic graph)
- Nodes: operators
  - TableScan
  - Select, Extract
  - Filter
  - Join, MapJoin, Sorted Merge Map Join
  - GroupBy, Limit
  - Union, Collect
  - FileSink, HashTableSink, ReduceSink
  - UDTF
- Graph represents data flow
- multiple (parallel) Map/Reduce phases possible

Hive-QL: Query transformation (Example)

- Example

```
SELECT *
FROM status_updates
WHERE status
   LIKE 'michael jackson'
```



Hive-QL: Query transformation (Example) (2)

```
SELECT COUNT(*)
FROM status_updates
WHERE ds='2009-08-01'
```

All updates

Materialize
map output

Updates per Map

Reduce 1

FileOutputOperator
table: school_summary
[0: bigint]

SelectOperator
expressions: [col[0], col[1]]
[0: bigint]

GroupByOperator
aggregations: [count(1)]
keys: [col[0]]
mode: mergepartial
[0:bigint]

Map 1

ReduceSinkOperator
partition cols: col[0]
[0: null, 1:bigint]

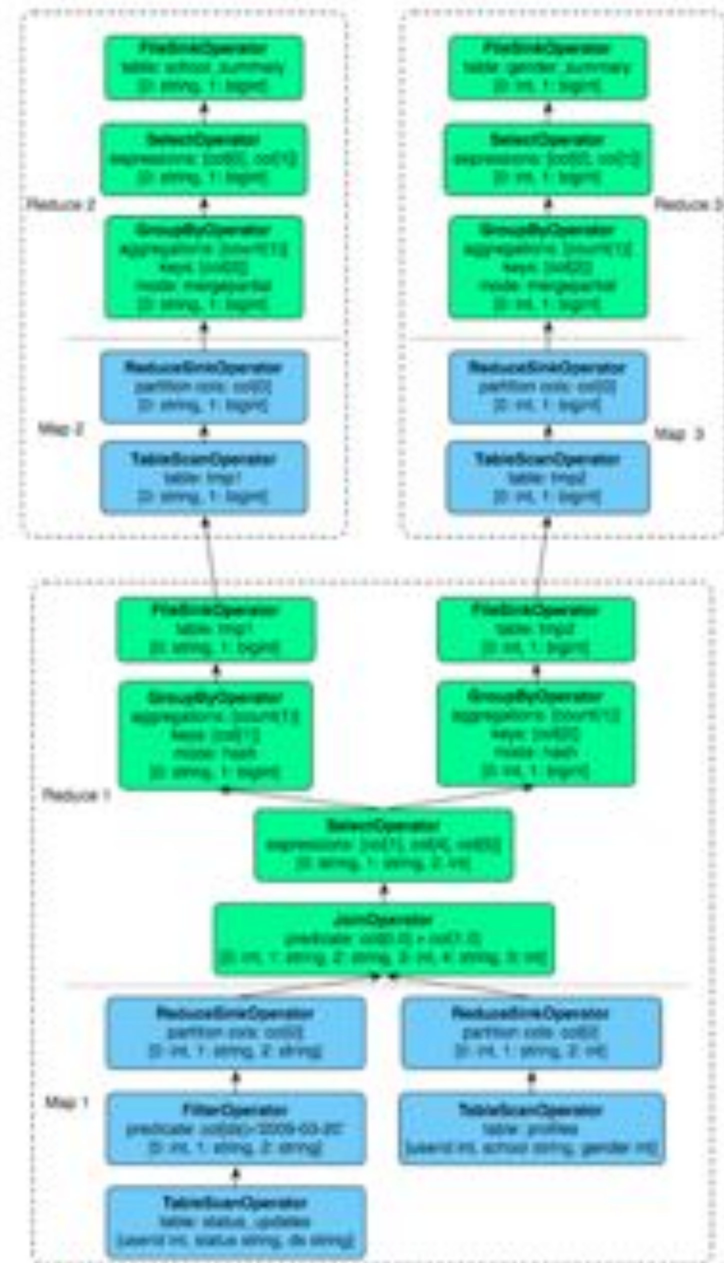
GroupByOperator
aggregations: [count(1)]
keys: [col[0]]
mode: hash
[0:null, 1: bigint]

FilterOperator
predicate: col[ds]='2009-08-01'
[0: int, 1: string, 2: string]

TableScanOperator
table: status_updates
[userid int, status string, ds string]

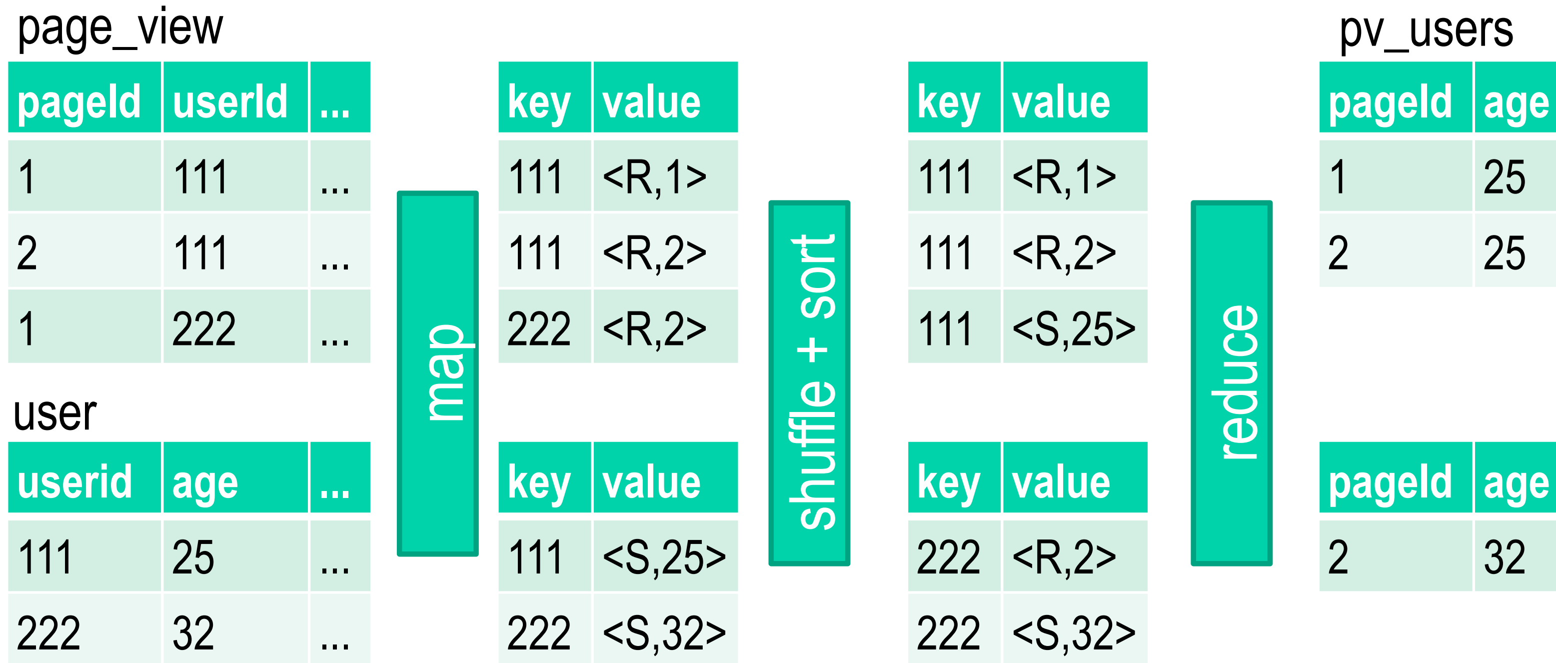
Hive: Query transformation and optimization

- DAG can become very complex
- Optimization techniques
  - Ignore unnecessary columns
  - Apply selection as early as possible
  - Ignore unnecessary partitions



Hive: Join

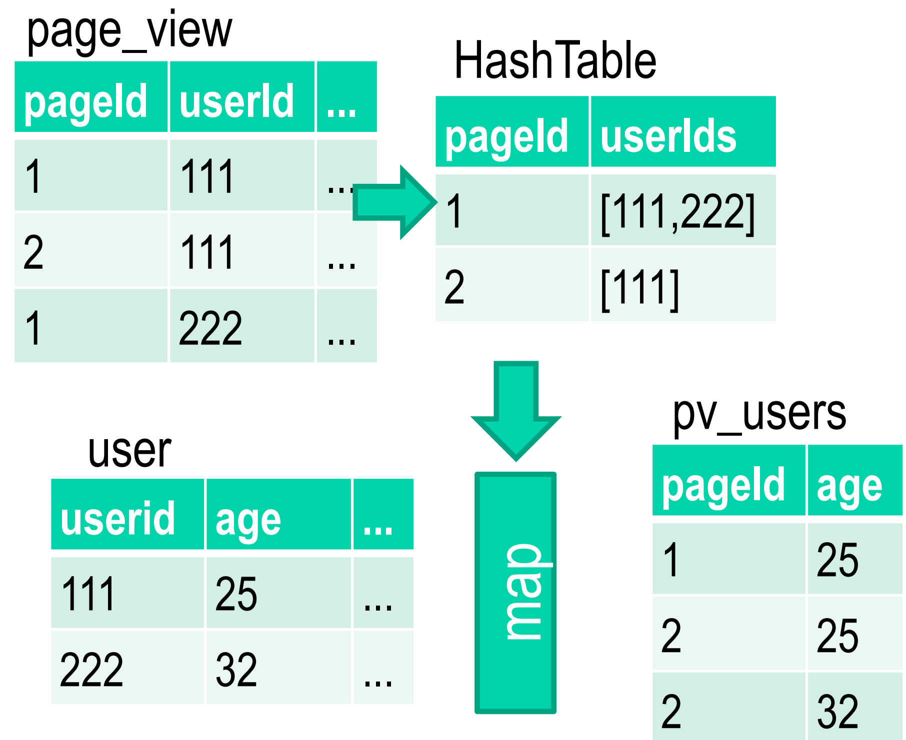
```
INSERT INTO TABLE pv_users
SELECT pv.pageid, u.age
FROM page_view pv
JOIN user u ON (pv.userid = u.userid)
```



- Key = Join-Key, Value has flag (R or S) to distinguish between tables
- Multi-way join using the same join key → 1 MapReduce job
- Multi-way join using *n* join keys → *n* MapReduce jobs

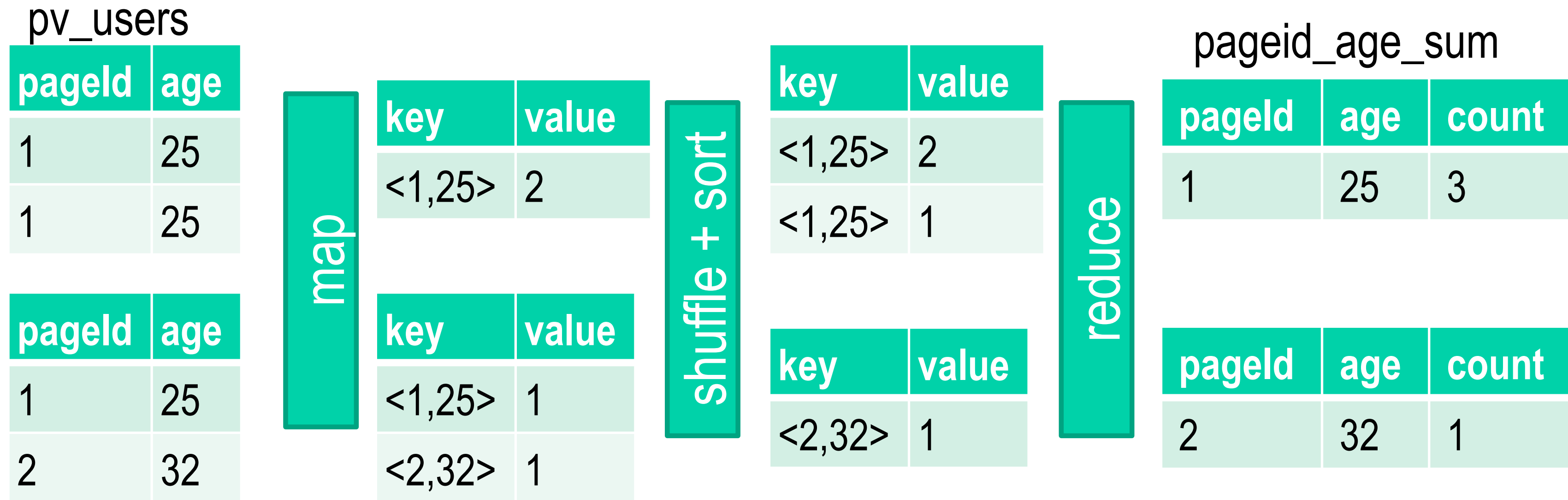
MapJoin: Performance improvement

- MapJoin
  - small table as additional map input
  - can be transformed into hash table
  - no reduce necessary
  - *n* way join possible if *n*-1 tables can be made available as additional map input
- Dynamic optimization
  - Determine small/large table at runtime
  - Apply MapJoin if possible, e.g., if small table(s) fit into memory



Hive: Group By

```
INSERT INTO TABLE pageid_age_sum
SELECT pageid, age, count(*)
FROM pv_users
GROUP BY pageid, age
```



- Key = group attributes
- Reduce = aggregation function
  - pre-aggregation using a map combiner is possible (e.g., (<1,25>,2))

User-defined scripts

- Include user-defined scripts in HiveQL queries using TRANSFORM operator
  - Data (de)serialization
  - Transfer via stdin/stdout

```
computeAuthorityValue.py

import sys
for line in sys.stdin:
  id = line.strip()
  ... compute authval ...
  print '\t'.join([id, authval])
```

user

userid	age	...
111	25	...
222	32	...

userid	authority_value
111	0.1
222	0.8

```
ADD FILE computeAuthorityValue.py;
SELECT
    TRANSFORM (userid)
    USING 'computeAuthorityValue.py'
    AS id, authority_value
  FROM user
```

Hadoop/MR vs. Parallel DBS

	Hadoop / MapReduce	Shared Nothing-RDBMS
Data size	PB	TB-PB
Data structure	semi-structured data	static schema
Partitioning	Blocks in DFS (byte-wise)	Horizontal
Query	MapReduce programs	Declarative (SQL)
Data access	Batch	via indexes (e.g., range)
Updates	Write once read many times	Read and write many times
Scheduling	Runtime	Compile-time
Processing	Parse tuples at runtime	efficient access to attributes (Storage Manager)
Data flow	Pull – materialize intermediate results	Push – tuple pipelining between operators
Fault tolerance	Restart map/reduce task	query restart (operator restart)
Scalability	linear, unlimited	linear, limited
Hardware	heterogeneous (cheap commodity hardware)	homogeneous (expensive high end hardware)
Software	free, open source	very expensive

Summary

- New database-like developments in the cloud
- Database techniques integrated in Hadoop/MR

- There is many many more
  - Pig Latin – a programming language for MapRedue-based data processing
  - HadoopDB – a hybrid of Hadoop/MR and RDBMS
  - Megastore – "BigTable + ACID"
  - Dremel – ad-hoc query system for analysis of read-only nested data
  - RDBMS in the Cloud – e.g., IBM DB2 running on Amazon EC2
  - Data management optimizations in the cloud – e.g., load balancing
  - ...