Data Management in the Cloud

INTRODUCTION

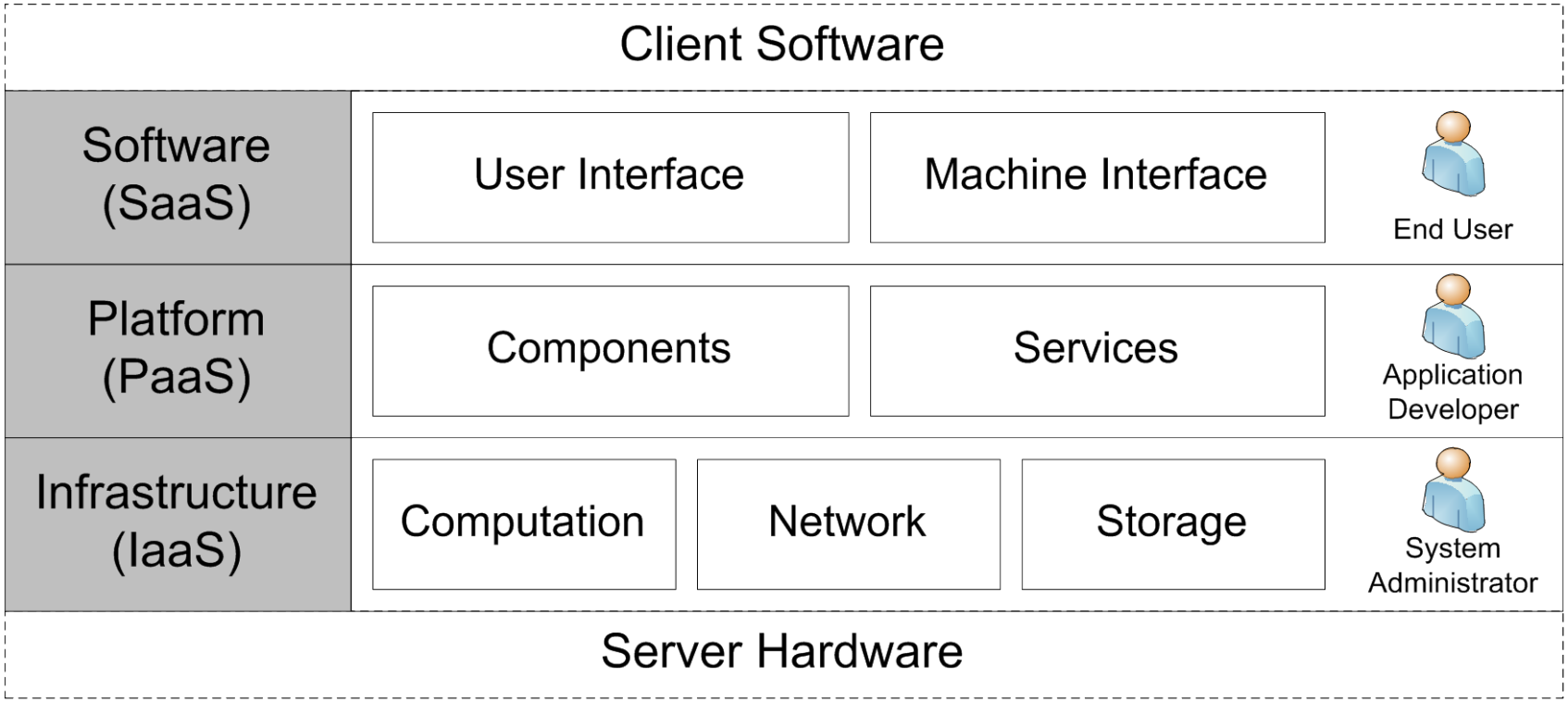
Outline

- Motivation
  - what is cloud computing?
  - what is cloud data management?
- Challenges, opportunities and limitations
  - what makes data management in the cloud difficult?
- New solutions
  - key/value, document, column family, graph, array, and object databases
  - scalable SQL databases
- Application
  - graph data and algorithms
  - usage scenarios

Cloud Computing

- Different definitions for “Cloud Computing” exist
  - http://tech.slashdot.org/article.pl?sid=08/07/17/2117221
- Common ground of many definitions
  - processing power, storage and software are **commodities** that are readily available from large infrastructure
  - **service-based view**: “everything as a service (*aaS)”, where only “Software as a Service (SaaS)” has a precise and agreed-upon definition
  - utility computing: **pay-as-you-go** model

Service-Based View on Computing



Source: Wikipedia (http://www.wikipedia.org)

Terminology

- Term **cloud computing** usually refers to both
  - **SaaS**: applications delivered over the Internet as services
  - **The Cloud**: data center hardware and systems software
- Public clouds
  - available in a **pay-as-you-go** manner to the public
  - service being sold is **utility computing**
  - Amazon Web Service, Microsoft Azure, Google AppEngine
- Private clouds
  - internal data centers of businesses or organizations
  - normally not included under **cloud computing**

Based on: "Above the Clouds: A Berkeley View of Cloud Computing", RAD Lab, UC Berkeley

Utility Computing

- Illusion of infinite computing resources
  - available on demand
  - no need for users to plan ahead for provisioning
- No up-front cost or commitment by users
  - companies can start small
  - increase resources only when there is an increase in need
- Pay for use on short-term basis as needed
  - processors by the hour and storage by the day
  - release them as needed, reward conservation

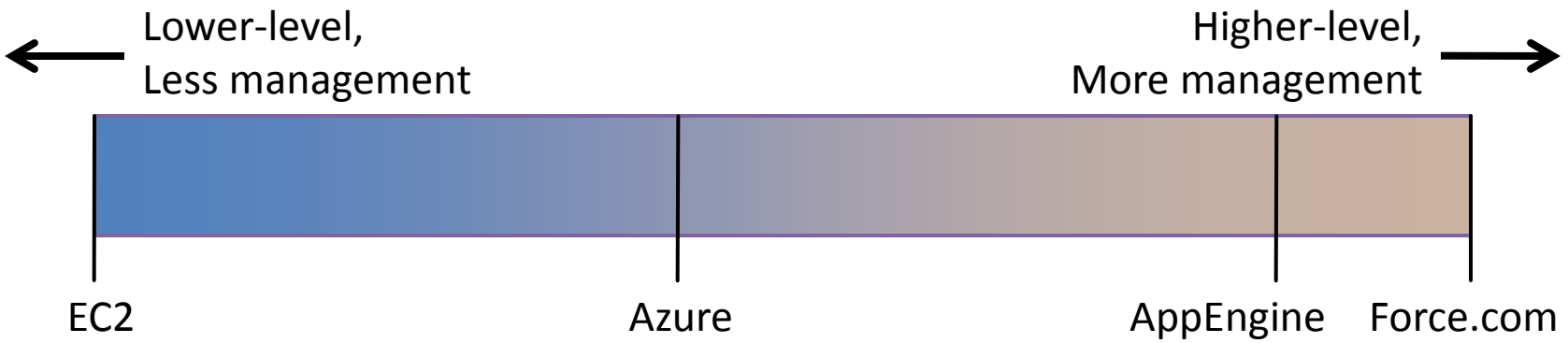
Based on: "Above the Clouds: A Berkeley View of Cloud Computing", RAD Lab, UC Berkeley

Virtualization

- Virtual resources abstract from physical resources
  - hardware platform, software, memory, storage, network
  - fine-granular, lightweight, flexible and dynamic
- Relevance to cloud computing
  - centralize and ease administrative tasks
  - improve scalability and work loads
  - increase stability and fault-tolerance
  - provide standardized, homogenous computing platform through hardware virtualization, i.e. **virtual machines**

Spectrum of Virtualization

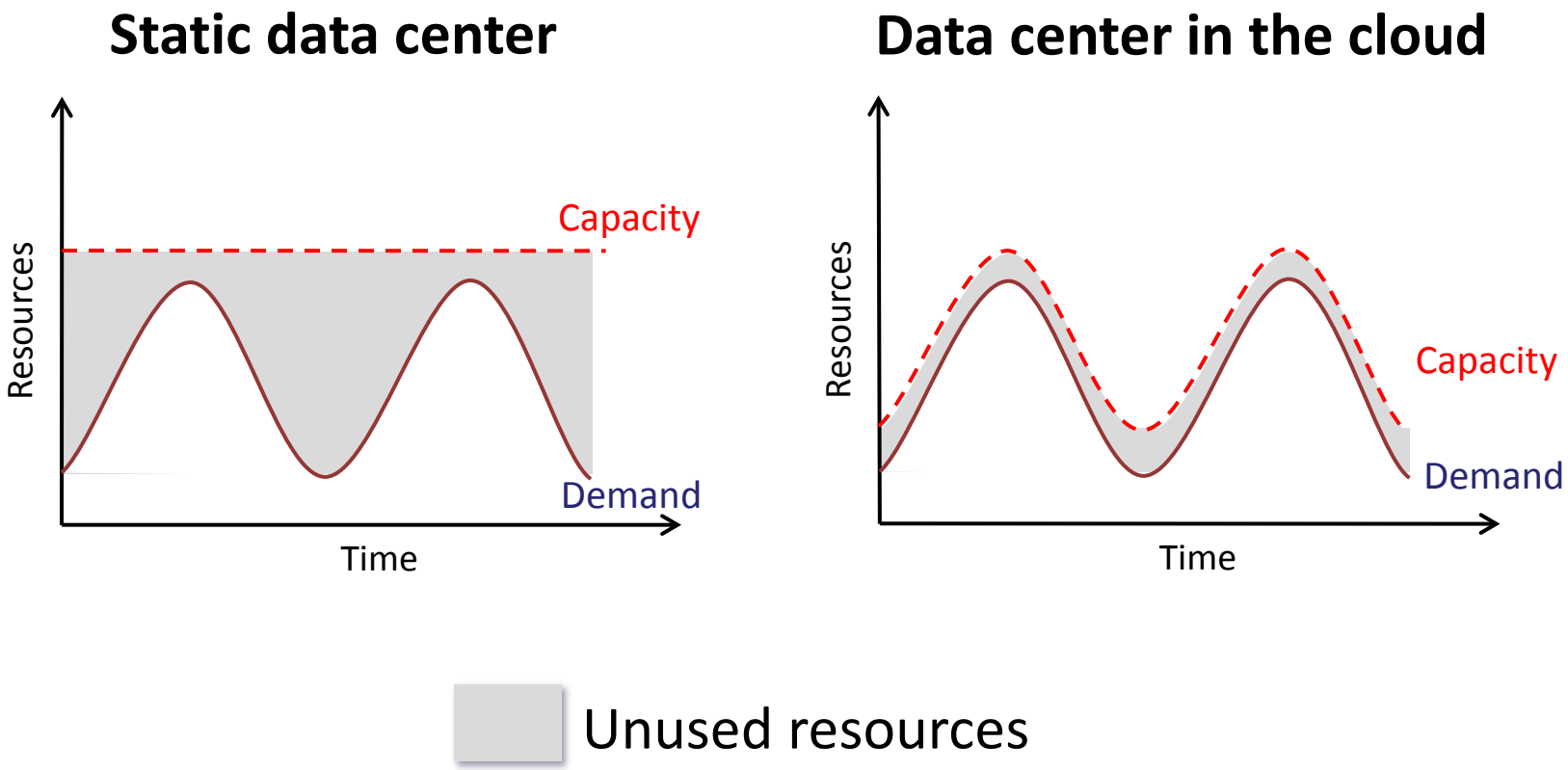
- Computation virtualization
  - Instruction set VM (Amazon EC2, 3Tera)
  - Byte-code VM (Microsoft Azure)
  - Framework VM (Google AppEngine, Force.com)
- Storage virtualization
- Network virtualization



Slide Credit: RAD Lab, UC Berkeley

Economics of Cloud Users

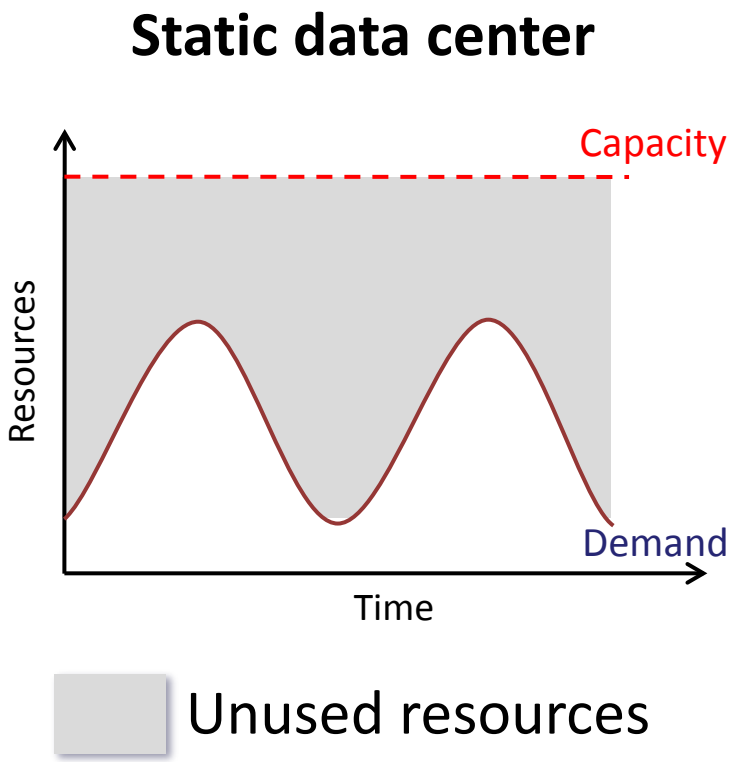
- Pay by use instead of provisioning for peak



Slide Credit: RAD Lab, UC Berkeley

Economics of Cloud Users

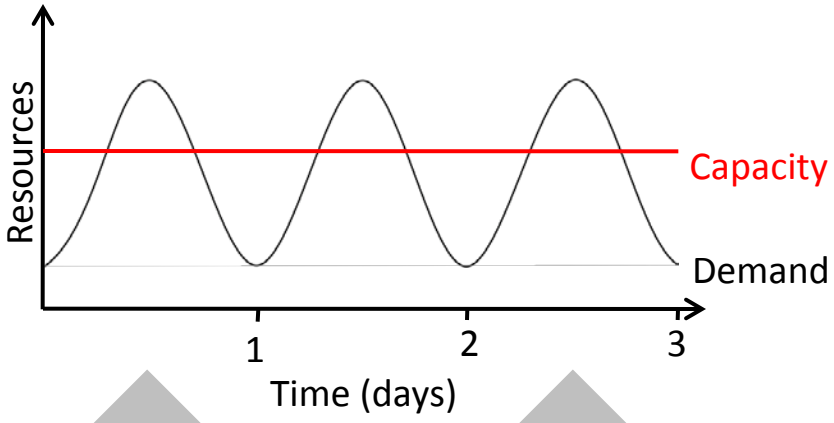
- Risk of over-provisioning: underutilization

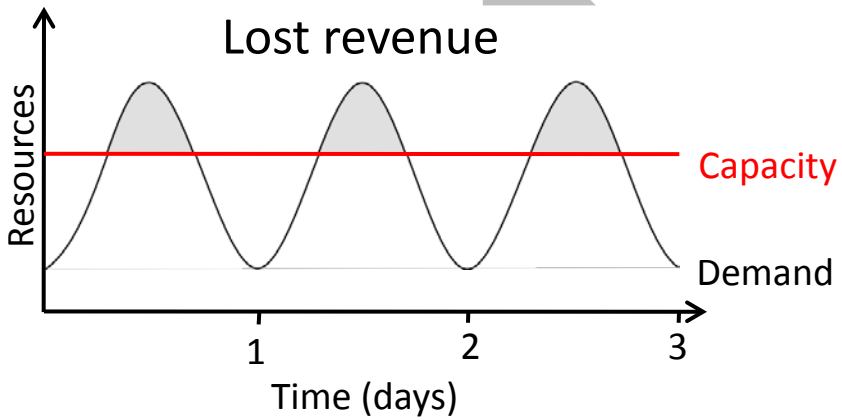


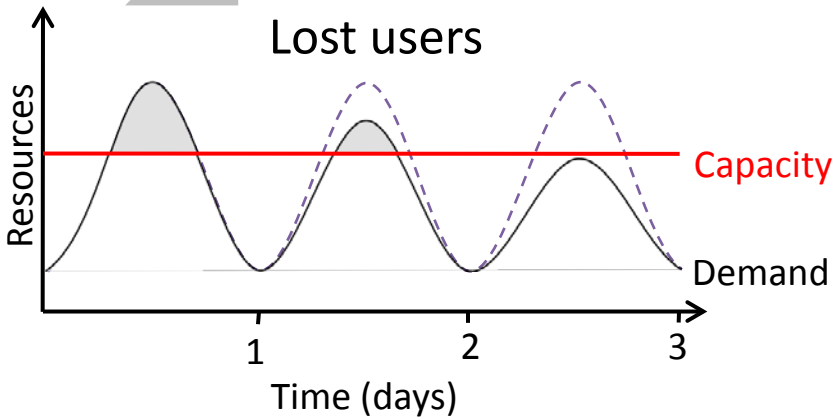
Slide Credit: RAD Lab, UC Berkeley

Economics of Cloud Users

- Heavy penalty for under-provisioning







Slide Credit: RAD Lab, UC Berkeley

Economics of Cloud Providers

Resource	Cost in Medium Data Center	Cost in Very Large Data Center	Ratio
Network	$95/Mbps/month	$13/Mbps/month	7.1x
Storage	$2.20/GB/month	$0.40/GB/month	5.7x
Administration	≈140 servers/admin	>1000 servers/admin	7.1x

Source: James Hamilton (http://perspectives.mvdirona.com)

- Cloud computing is 5-7x cheaper than traditional in-house computing
- Added benefits
  - utilize off-peak capacity (Amazon)
  - sell .NET tools (Microsoft)
  - reuse existing infrastructure (Google)

Slide Credit: RAD Lab, UC Berkeley

Data Management in the Cloud

- Data management applications are potential candidates for deployment in the cloud
  - **industry:** enterprise database system have significant up-front cost that includes both hardware and software costs
  - **academia:** manage, process and share mass-produced data in the cloud
- Many “Cloud Killer Apps” are in fact data-intensive
  - Batch Processing as with map/reduce
  - Online Transaction Processing (OLTP) as in automated business applications
  - Offline Analytical Processing (OLAP) as in data mining or machine learning

Scientific Data Management Applications

- Old model
  - "Query the world"
  - data acquisition coupled to a specific hypothesis
- New model
  - "Download the world"
  - data acquired en masse, in support of many hypotheses
- E-science examples
  - astronomy: high-resolution, high-frequency sky surveys, …
  - oceanography: high-resolution models, cheap sensors, satellites, …
  - biology: lab automation, high-throughput sequencing, ...

Slide Credit: Bill Howe, U Washington

Scaling Databases

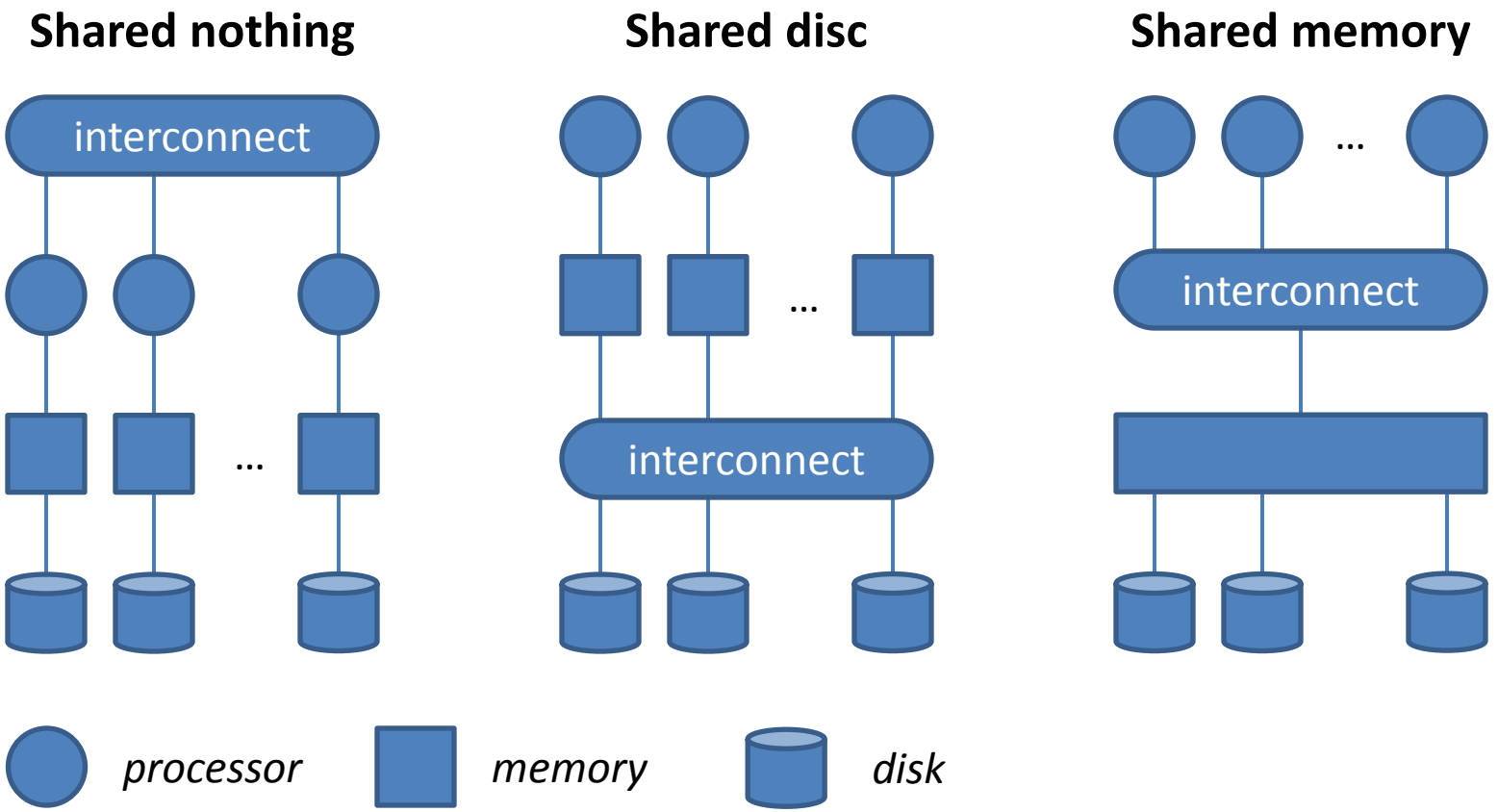
- Flavors of database scalability
  - lots of (small) transactions
  - lots of copies of the data
  - lots of processor running on a single query (compute intensive tasks)
  - extremely large data set for one query (data intensive tasks)
- Data replication
  - move data to where it is needed
  - managed replication for availability and reliability

Revisit Cloud Characteristics

- Compute power is elastic, but only if workload is parallelizable
  - transactional database management systems do not typically use a shared-nothing architecture
  - shared-nothing is a good match for analytical data management
- Scalability
  - **in the past**: out-of-core, works even if data does not fit in main memory
  - **in the present**: exploits thousands of (cheap) nodes in parallel

Based on: "Data Management in the Cloud: Limitations and Opportunities", IEEE, 2009.

Parallel Database Architectures



Source: D. DeWitt and J. Gray: "Parallel Database Systems: The Future of High Performance Database Processing", CACM 36(6), pp. 85-98, 1992.

Revisit Cloud Characteristics

- Data is stored at an untrusted host
  - there are risks with respect to privacy and security in storing transactional data on an untrusted host
  - particularly sensitive data can be left out of analysis or anonymized
  - sharing and enabling access is often precisely the goal of using the cloud for scientific data sets

Based on: "Data Management in the Cloud: Limitations and Opportunities", IEEE, 2009.

Revisit Cloud Characteristics

- Data is replicated, often across large geographic distances
  - it is hard to maintain ACID guarantees in the presence of large-scale replication
  - full ACID guarantees are typically not required in analytical applications
- Virtualizing large data collections is challenging
  - data loading takes more time than starting a VM
  - storage cost vs. bandwidth cost
  - online vs. offline replication

Based on: "Data Management in the Cloud: Limitations and Opportunities", IEEE, 2009.

Challenges

- Trade-off between functionality and operational cost
  - restricted interface, minimalist query language, limited consistency guarantees
  - pushes more programming burden on developers
  - enables predictable services and service level agreements
- Manageability
  - limited human intervention, high-variance workloads, and a variety of shared infrastructures
  - need for self-managing and adaptive database techniques

Based on: "The Claremont Report on Database Research", 2008

Challenges

- Scalability
  - today's SQL databases cannot scale to the thousands of nodes deployed in the cloud context
  - hard to support multiple, distributed updaters to the same data set
  - hard to replicate huge data sets for availability, due to capacity (storage, network bandwidth, …)
  - **storage:** different transactional implementation techniques, different storage semantics, or both
  - **query processing and optimization:** limitations on either the plan space or the search will be required
  - **programmability:** express programs in the cloud

Based on: "The Claremont Report on Database Research", 2008

Challenges

- Data privacy and security
  - protect from other users and cloud providers
  - specifically target usage scenarios in the cloud with practical incentives for providers and customers
- New applications: "*mash up*" interesting data sets
  - expect services pre-loaded with large data sets, stock prices, web crawls, scientific data
  - data sets from private or public domain
  - might give rise to federated cloud architectures

Based on: "The Claremont Report on Database Research", 2008

Data Management in the Cloud

SCALABLE DATA STORES

Overview

- New systems have emerged to address requirements of data management in the cloud
  - so-called “NoSQL” data stores
  - scalable SQL databases
- **Horizontal Scaling**
  - shared nothing
  - replicating and partitioning data over thousands of servers
  - distribute “simple operation” workload over thousands of servers
- **Simple Operations**
  - key lookups
  - read and writes of one or a small number of records
  - **no** complex queries or joins

Sharding, Horizontal, Vertical

- *There should be a slide to better explain these concepts in order to motivate some of the "NoSQL" data models.*

Defining “NoSQL”

- No agreed upon definition
  - “not only SQL”
  - “not relational”
  - …
- Six key features
  1. ability to horizontally scale simple operation throughput over many servers
  2. ability to replicate and distribute (partition) data over many servers
  3. simple call level interface or protocol (in contrast to a SQL binding)
  4. weaker concurrency model than ACID transactions of most relational (SQL) database systems
  5. efficient use of distributed indexes and RAM for data storage
  6. ability to dynamically add new attributes to data records

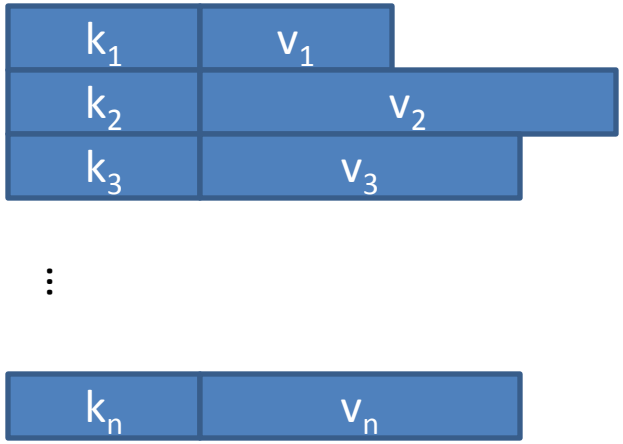
Based on: “Scalable SQL and NoSQL Data Stores” by R. Cattell, 2010

Data Models

- Terminology
  - **tuple:** row in a relational table, where attribute names and types are defined by a schema, and values must be scalar
  - **document:** supports both scalar values and nested documents, and the attributes are dynamically defined for each document
  - **column family:** groups key/value pairs (columns) into families to partition and replicate them; one column family is similar to a document as new (nested, list-valued) attributes can be added
  - **object:** analogous to objects in programming languages, but without procedural methods
- Relational
  - data is stored in relations (tables) of tuples (rows) of scalar values
  - queries expressed over arbitrary (combinations of) attributes
  - indexes defined over arbitrary (combinations of) attributes

Based on: "Scalable SQL and NoSQL Data Stores" by R. Cattell, 2010

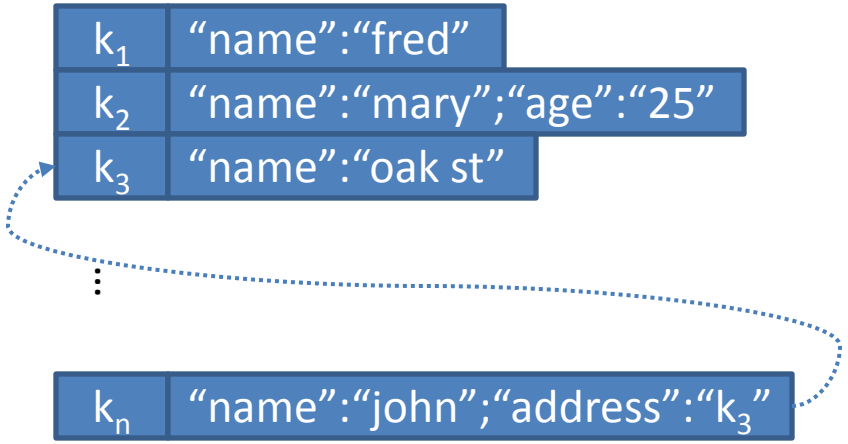
Key/Value Data Model



- Interface
  - put(key, value)
  - get(key): value

- Data storage
  - values (data) are stored based on programmer-defined keys
  - system is agnostic as to the structure (semantics) of the value
- Queries are expressed in terms of keys
- Indexes are defined over keys
  - some systems support secondary indexes over (part of) the value

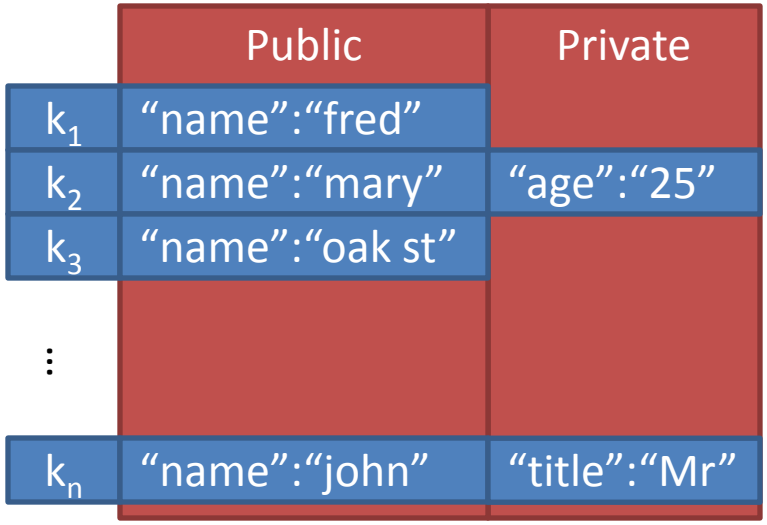
Document Data Model



- Interface
  - set(key, document)
  - get(key): document
  - set(key, name, value)
  - get(key, name): value

- Data storage
  - documents (data) is stored based on programmer-defined keys
  - system is aware of the (arbitrary) document structure
  - support for lists, pointers and nested documents
- Queries expressed in terms of key (or attribute, if index exists)
- Support for key-based indexes and secondary indexes

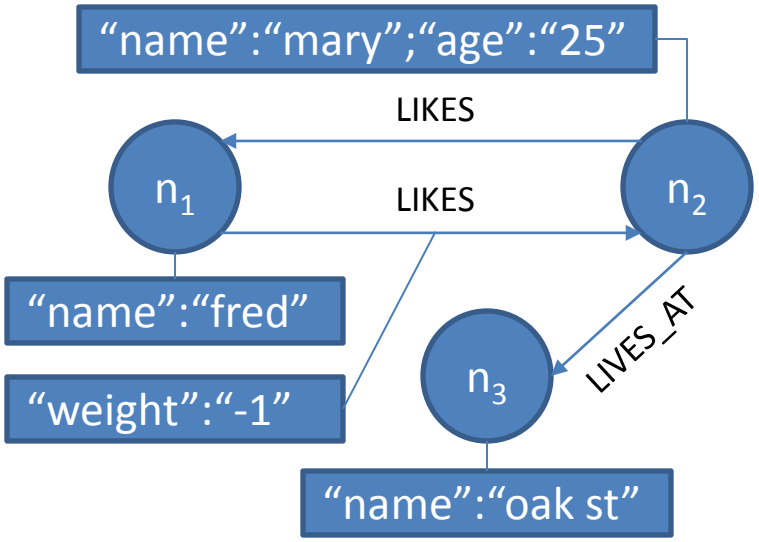
Column Family Data Model



- Interface
  - define(family)
  - insert(family, key, columns)
  - get(family, key): columns

- Data storage
  - <name, value, timestamp> triples (so-called columns) are stored based on a column family and key; a column family is similar to a document
  - system is aware of (arbitrary) structure of column family
  - system uses column family information to replicate and distribute data
- Queries are expressed based on key and column family
- Secondary indexes per column family are typically supported

Graph Data Model

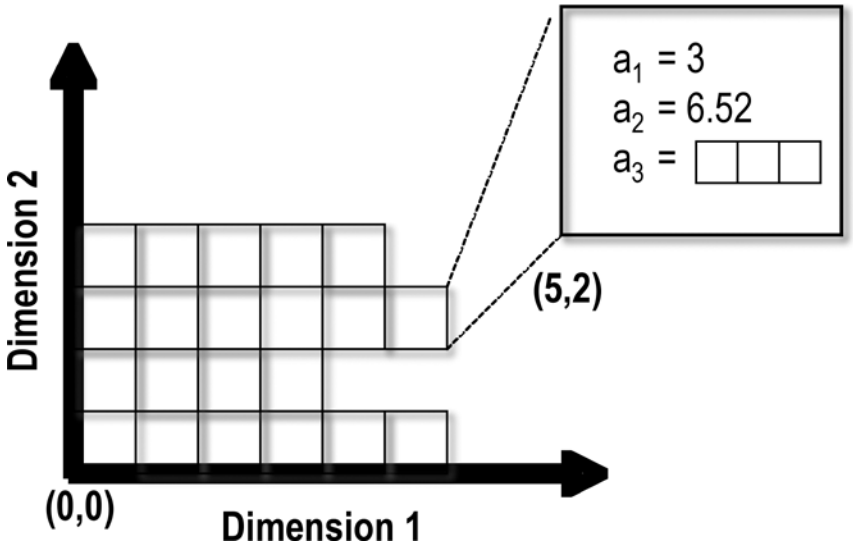


- Interface
  - create: id
  - get(id)
  - connect(id_1, id_2): id
  - addAttribute(id, name, value)
  - getAttribute(id, name): value

- Data storage
  - data is stored in terms of nodes and (typed) edges
  - both nodes and edges can have (arbitrary) attributes
- Queries are expressed based on system ids (if no indexes exist)
- Secondary indexes for nodes and edges are supported
  - retrieve nodes by attributes and edges by type, start and/or end node, and/or attributes

Array Data Model

- Nested multi-dimensional arrays
  - cells can be tuples or other arrays
  - can have non-integer dimensions
- Additional “History” dimension on updatable arrays
- Ragged arrays allow each row or column to have a different length
- Supports multiple flavors of “null”
  - array cells can be “EMPTY”
  - user-definable treatment of special values

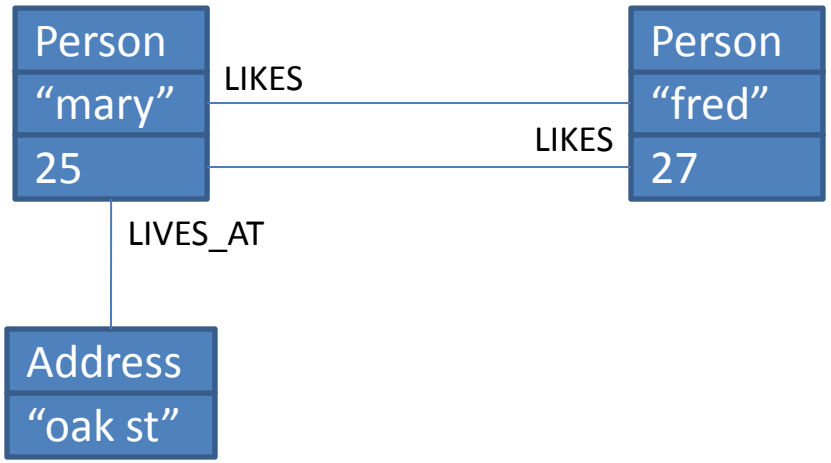


SciDB DDL

```
CREATE ARRAY Test_Array
     < A: integer NULLS,
       B: double,
       C: USER_DEFINED_TYPE >
      [I=0:99999,1000, 10, J=0:99999,1000, 10 ]
      PARTITION OVER ( Node1, Node2, Node3 )
      USING block_cyclic();
```

Attribute names	A, B, C
Index names	I, J
Chunk size	1000
Overlap	10

Object Data Model



- Interface
  - set(object)
  - get(query): object

- Data storage
  - typed programming language objects (plus referenced objects) stored
  - attribute can be collection-valued
  - database is aware of the type (schema) of objects
- Objects are retrieved using queries or by traversal from "roots"
- Specialized indexes can be expressed based on schema

Data Management in the Cloud

SCALABLE CONSISTENCY AND TRANSACTION MODELS

Brewer's Conjecture

- Three properties that are desirable and expected from real-world shared-data systems
  - **C:** data consistency
  - **A:** availability
  - **P:** tolerance of network partition
- At *PODC 2000* (Portland, OR), Eric Brewer made the conjecture that only two of these properties can be satisfied by a system at any given time
- Conjecture was formalized and confirmed by MIT researchers Seth Gilbert and Nancy Lynch in 2002
- Now known as the **CAP Theorem**

Data Consistency

- Database systems typically implement ACID transactions
  - **A**tomicity: “all or nothing”
  - **C**onsistency: transactions never observe or result in inconsistent data
  - **I**solation: transactions are not aware of concurrent transactions
  - **D**urability: once committed, the state of a transaction is permanent
- Useful in automated business applications
  - banking: at the end of a transaction the sum of money in both accounts is the same as before the transaction
  - online auctions: the last bidder wins the auction
- There are applications that can deal with looser consistency guarantees and periods of inconsistency

Availability

- Services are expected to be highly available
  - every request should receive a response
  - it can create real-world problems when a service goes down
- Realistic goal
  - service should be as available as the network it run on
  - if any service on the network is available, the service should be available

Partition-Tolerance

- A service should continue to perform as expected
  - if some nodes crash
  - if some communication links fail
- One desirable fault tolerance property is resilience to a network partitioning into multiple components
- In cloud computing, node and communication failures are not the exception but everyday events

Problems with CAP

- Asymmetry of CAP properties
  - **C** is a property of the system in general
  - **A** is a property of the system only when there is a partition
- There are not three different choices
  - in practice, **CA** and **CP** are indistinguishable, since **A** is only sacrificed when there is a partition
- Used as an excuse to not bother with consistency
  - "Availability is really important to me, so CAP says I have to get rid of consistency"

Source: Daniel Abadi, Yale University

Another Problem to Fix

- Apart from availability in the face of partitions, there are other costs to consistency
- Overhead of synchronization schemes
- Latency
  - if workload can be partitioned geographically, latency is not so bad
  - otherwise, there is no way to get around at least one round-trip message

Source: Daniel Abadi, Yale University

A Cut at Fixing Both Problems

- PACELC
  - In the case of a partition (**P**), does the system choose availability (**A**) or consistency (**C**)?
  - Else (**E**), does the system choose latency (**L**) or consistency (**C**)?
- PA/EL
  - Dynamo, SimpleDB, Cassandra, Riptano, CouchDB, Cloudant
- PC/EC
  - ACID compliant database systems
- PA/EC
  - GenieDB
- PC/EL
  - Existence is debatable

Source: Daniel Abadi, Yale University

A Case for P*/EC

- Increased push for horizontally scalable transactional database systems
  - cloud computing
  - distributed applications
  - desire to deploy applications on cheap, commodity hardware
- Vast majority of currently available horizontally scalable systems are P*/EL
  - developed by engineers at Google, Facebook, Yahoo, Amazon, etc.
  - these engineers can handle reduced consistency, but it's really hard, and there needs to be an option for the rest of us
- Also
  - distributed concurrency control and commit protocols are expensive
  - once consistency is gone, atomicity usually goes next → NoSQL

Source: Daniel Abadi, Yale University

Key Problems to Overcome

- High availability is critical, replication must be a first class citizen
- Today's systems generally act, then replicate
  - complicates semantics of sending read queries to replicas
  - need confirmation from replica before commit (increased latency) if you want durability and high availability
  - In progress transactions must be aborted upon a master failure
- Want system that replicates then acts
- Distributed concurrency control and commit are expensive, want to get rid of them both

Source: Daniel Abadi, Yale University

Key Idea

- Instead of weakening ACID, strengthen it
- Challenges
  - guaranteeing equivalence to *some* serial order makes active replication difficult
  - running the same set of transactions on two different replicas might cause replicas to diverge
- Disallow any nondeterministic behavior
- Disallow aborts caused by DBMS
  - disallow deadlock
  - distributed commit much easier if there are no aborts

Source: Daniel Abadi, Yale University

Consequences of Determinism

- Replicas produce the same output, given the same input
  - facilitates active replication
- Only initial input needs to be logged, state at failure can be reconstructed from this input log (or from a replica)
- Active distributed transactions not aborted upon node failure
  - greatly reduces (or eliminates) cost of distributed commit
  - don't have to worry about nodes failing during commit protocol
  - don't have to worry about effects of transaction making it to disk before promising to commit transaction
  - just need one message from any node that potentially can deterministically abort the transaction
  - this message can be sent in the middle of the transaction, as soon as it knows it will commit

Source: Daniel Abadi, Yale University

Strong vs. Weak Consistency

- Strong consistency
  - after an update is committed, each subsequent access will return the updated value
- Weak consistency
  - the systems does not guarantee that subsequent accesses will return the updated value
  - a number of conditions might need to be met before the updated value is returned
  - **inconsistency window:** period between update and the point in time when every access is guaranteed to return the updated value

Based on: "Eventual Consistency" by W. Vogels, 2008

Eventual Consistency

- Specific form of weak consistency
- “If no new updates are made, eventually all accesses will return the last updated values”
- In the absence of failures, the maximum size of the inconsistency window can be determined based on
  - communication delays
  - system load
  - number of replicas
  - …
- Not a new esoteric idea!
  - Domain Name System (DNS) uses eventual consistency for updates
  - RDBMS use eventual consistency for asynchronous replication or backup (e.g. log shipping)

Based on: “Eventual Consistency” by W. Vogels, 2008

Models of Eventual Consistency

- Causal Consistency
  - if A communicated to B that it has updated a value, a subsequent access by B will return the updated value, and a write is guaranteed to supersede the earlier write
  - access by C that has no causal relationship to A is subject to normal eventual consistency rules
- Read-your-writes Consistency
  - special case of the causal consistency model
  - after updating a value, a process will always read the updated value and never see an older value
- Session Consistency
  - practical case of read-your-writes consistency
  - data is accessed in a session where read-your-writes is guaranteed
  - guarantees do not span over sessions

Based on: "Eventual Consistency" by W. Vogels, 2008

Models of Eventual Consistency

- Monotonic Read Consistency
  - if a process has seen a particular value, any subsequent access will never return any previous value
- Monotonic Write Consistency
  - system guarantees to serialize the writes of one process
  - systems that do not guarantee this level of consistency are hard to program

- Properties can be combined
  - e.g. monotonic reads plus session-level consistency
  - e.g. monotonic reads plus read-your-own-writes
  - quite a few different scenarios are possible
  - it depends on an application whether it can deal with the consequences

Based on: "Eventual Consistency" by W. Vogels, 2008

Configurations

- Definitions
  - **N**: number of nodes that store a replica
  - **W**: number of replicas that need to acknowledge a write operation
  - **R**: number of replicas that are accessed for a read operation
- W+R > N
  - e.g. **synchronous replication** (N=2, W=2, and R=1)
  - write set and read set always overlap
  - strong consistency can be guaranteed through **quorum protocols**
  - risk of reduced availability: in basic quorum protocols, operations fail if fewer than the required number of nodes respond, due to node failure
- W+R = N
  - e.g. **asynchronous replication** (N=2, W=1, and R=1)
  - strong consistency cannot be guaranteed

Based on: "Eventual Consistency" by W. Vogels, 2008

Configurations

- R=1, W=N
  - optimized for **read access**: single read will return a value
  - write operation involves all nodes and risks not to succeed
- R=N, W=1
  - optimized for **write access**: write operation involves only one node and relies on lazy (epidemic) technique to update other replicas
  - read operation involves all nodes and returns “latest” value
  - durability is not guaranteed in presence of failures
- W < (N+1)/2
  - risk of conflicting writes
- W+R <= N
  - **weak/eventual consistency**

Based on: “Eventual Consistency” by W. Vogels, 2008

BASE

- **B**asically **A**vailable, **S**oft state, **E**ventually Consistent
- As consistency is achieved eventually, conflicts have to be resolved at some point
  - read repair
  - write repair
  - asynchronous repair
- Conflict resolution is typically based on a global (partial) ordering of operations that (deterministically) guarantees that all replicas resolve conflicts in the same way
  - client-specified timestamps
  - vector clocks

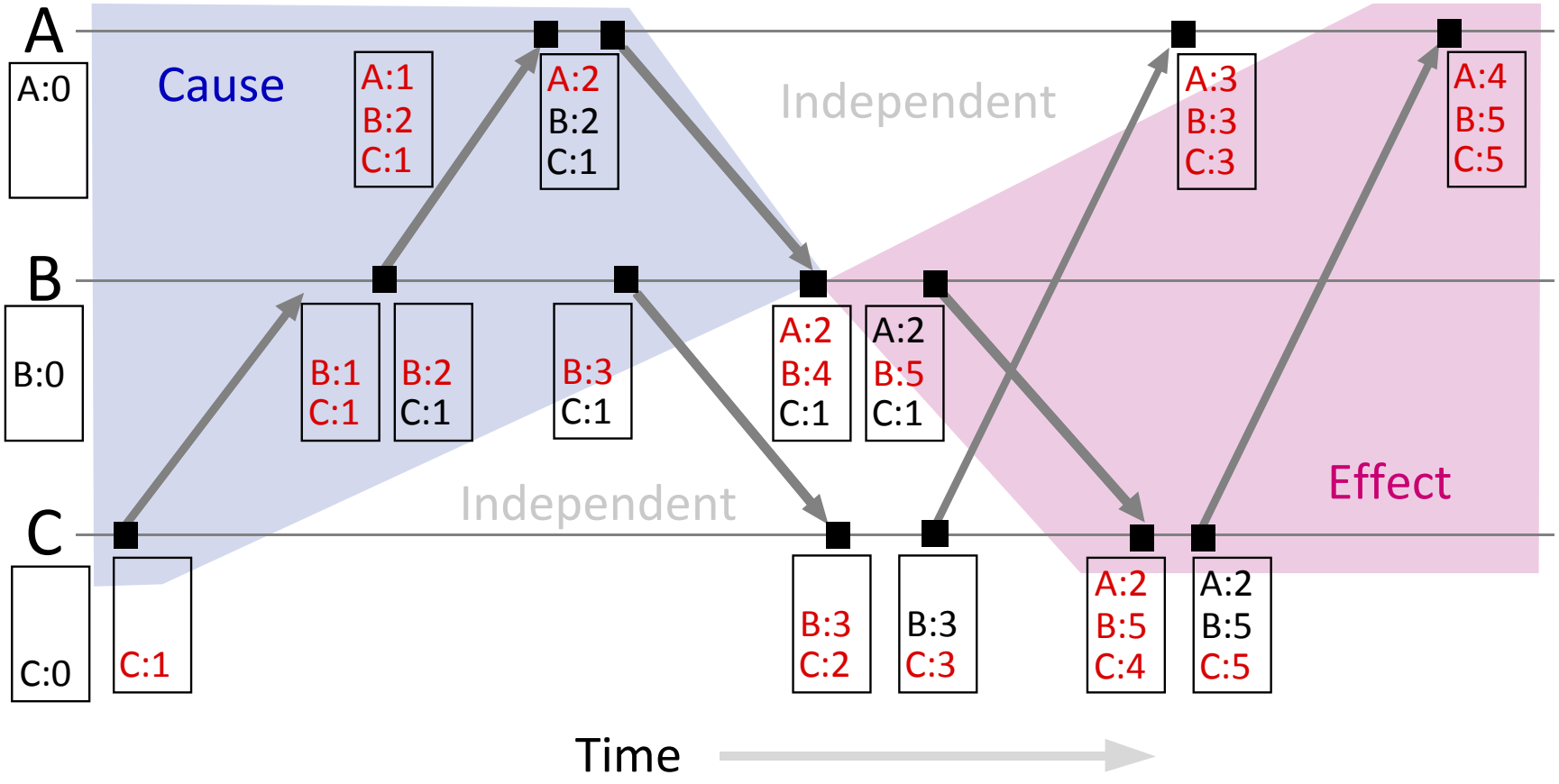
Vector Clocks

- Generate a partial ordering of events in a distributed system and detecting causality violations
- A **vector clock** of a system of *n* processes is an vector of *n* logical clocks (one clock per process)
  - messages contain the state of the sending process's logical clock
  - local “smallest possible values” copy of the global vector clock is kept in each process
- Vector clocks algorithm was independently developed by Colin Fidge and Friedemann Mattern in 1988

Update Rules for Vector Clocks

- All clocks are initialized to zero
- A process increments its own logical clock in the vector by one
  - each time it experiences an internal event
  - each time a process prepares to send a message
  - each time a process receives a message
- Each time a process sends a message, it transmits the entire vector clock along with the message being sent
- Each time a process receives a message, it updates each element in its vector by taking the pair-wise maximum of the value in its own vector clock and the value in the vector in the received message

Vector Clock Example



Source: Wikipedia (http://www.wikipedia.org)

References

- S. Gilbert and N. Lynch: **Brewer's Conjecture and the Feasibility of Consistent, Available and Partition-Tolerant Web Services.** *SIGACT News 33(2), pp. 51-59, 2002.*
- W. Vogels: **Eventually Consistent.** *ACM Queue 6(6), pp. 14-19, 2008.*

Data Management in the Cloud

CLOUD-SCALE FILE SYSTEMS

Google File System (GFS)

- Designing a file system for the Cloud
  - design assumptions
  - design choices
- Architecture
  - GFS Master
  - GFS Chunkservers
  - GFS Clients
- System operations and interactions
- Replication
  - fault tolerance
  - high availability

Design Assumptions

- System is built from many inexpensive commodity components
  - component failures happen on a routine basis
  - monitor itself to detect, tolerate, and recover from failures
- System stores a modest number of large files
  - a few million files, typically 100 MB or larger
  - multi-GB files are common and need to be managed efficiently
  - small files are to be supported but not optimized for
- System workload
  - **large streaming reads:** successive reads from one client read contiguous region, commonly 1 MB or more
  - **small random reads:** typically a few KB at some arbitrary offset
  - **large sequential writes:** append data to files; operation sizes similar to streaming reads; small arbitrary writes supported, but not efficiently

Design Assumption

- Support concurrent appends to the same file
  - efficient implementation
  - well-defined semantics
  - use case: producer-consumer queues or many-way merging, with hundreds of processes concurrently appending to a file
  - atomicity with minimal synchronization overhead is essential
  - file might be read later or simultaneously
- High sustained bandwidth is more important than low latency

Design Decisions: Interface

- GFS does not implement a standard API such as POSIX
- Supports standard file operations
  - **create/delete**
  - **open/close**
  - **read/write**
- Supports additional operations
  - **snapshot:** creates a copy of a file or a directory tree at low cost, using copy on write
  - **record append:** allows multiple clients to append data to the same file concurrently, while guaranteeing the atomicity of each individual client's append

Design Decisions: Architecture

- GFS cluster
  - single master and multiple chunkservers
  - accessed by multiple clients
  - components are typically commodity Linux machines
  - GFS server processes run in user mode
- Chunks
  - files are divided into fixed-size chunks
  - identified by globally unique chunk handle (64 bit), assigned by master
  - chunks are replicated for reliability, typically the replication factor is 3

Design Decisions: Architecture

- Multiple chunkservers
  - store chunks on local disk as Linux files
  - accept and handle data requests
  - no special caching, relies on Linux's buffer cache
- Single master simplifies overall design
  - enables more sophisticated **chunk placement** and **replication**, but **single point of failure**
  - maintains **file system metadata**: namespace, access control information, file-to-chunk mapping, current chunk location
  - performs **management activities**: chunk leases, garbage collection, orphaned chunks, chunk migration
  - **heart beats**: periodic messages sent to chunkservers to give instructions or to collect state
  - does not accept or handle data requests

Architecture

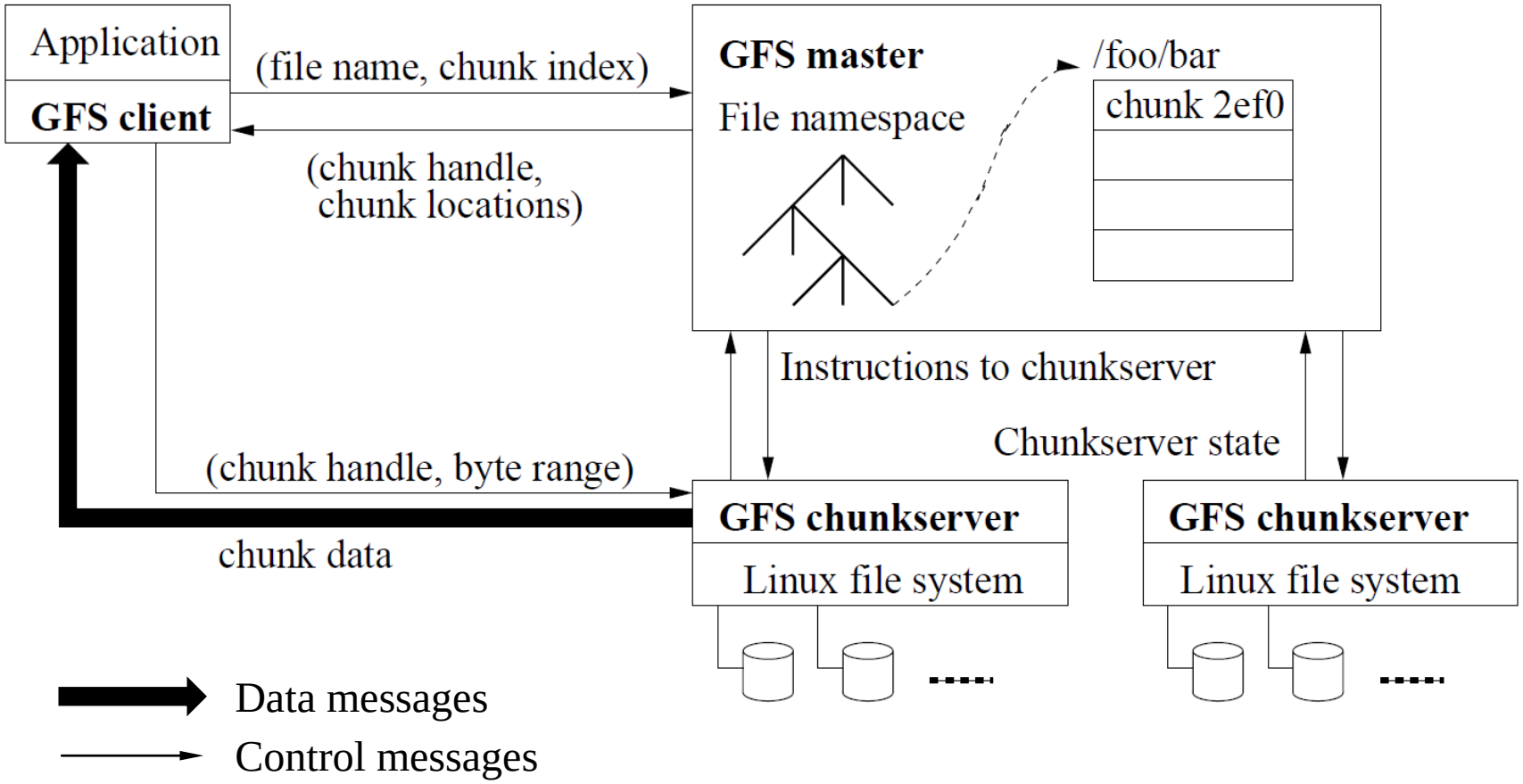


Figure Credit: "The Google File System" by S. Ghemawat, H. Gobioff, and S.-T. Leung, 2003

Design Decisions: Chunk Size

- One of the key parameters
  - set to a large value, i.e. 64 MB
  - to avoid fragmentation, chunkservers use lazy space allocation, i.e. files are only extended as needed
- Advantages
  - reduce interaction between client and master
  - reduce network overhead by using persistent TCP connection to do many operations on one chunk
  - reduce size of metadata stored on master
- Disadvantages
  - small files consist of very few chunks
  - risk of hot spots $\rightarrow$ increase replication factor for small files

Design Decision: Metadata

- All metadata is kept in the master's main memory
  - **file and chunk namespaces:** lookup table with prefix compression
  - **file-to-chunk mapping**
  - **locations of chunk replicas:** not stored, but queried from chunkservers
- Operation log
  - stored on master's local disc and replicated on remote machines
  - used to recover master in the event of a crash
- Discussion
  - size of master's main memory limits number of possible files
  - master maintains less than 64 bytes per chunk

Design Decisions: Consistency Model

- Relaxed consistency model
  - tailored to Google's highly distributed applications
  - simple and efficient to implement
- File namespace mutations are **atomic**
  - handled exclusively by the master
  - namespace locking guarantees atomicity and correctness
  - master's operation log defines global total order of operations
- State of file region after data mutation
  - **consistent:** all clients always see the same data, regardless of the replica they read from
  - **defined:** consistent, plus all clients see the entire data mutation
  - **undefined but consistent:** result of concurrent successful mutations; all clients see the same data, but it may not reflect any one mutation
  - **inconsistent:** result of a failed mutation

Design Decisions: Consistency Model

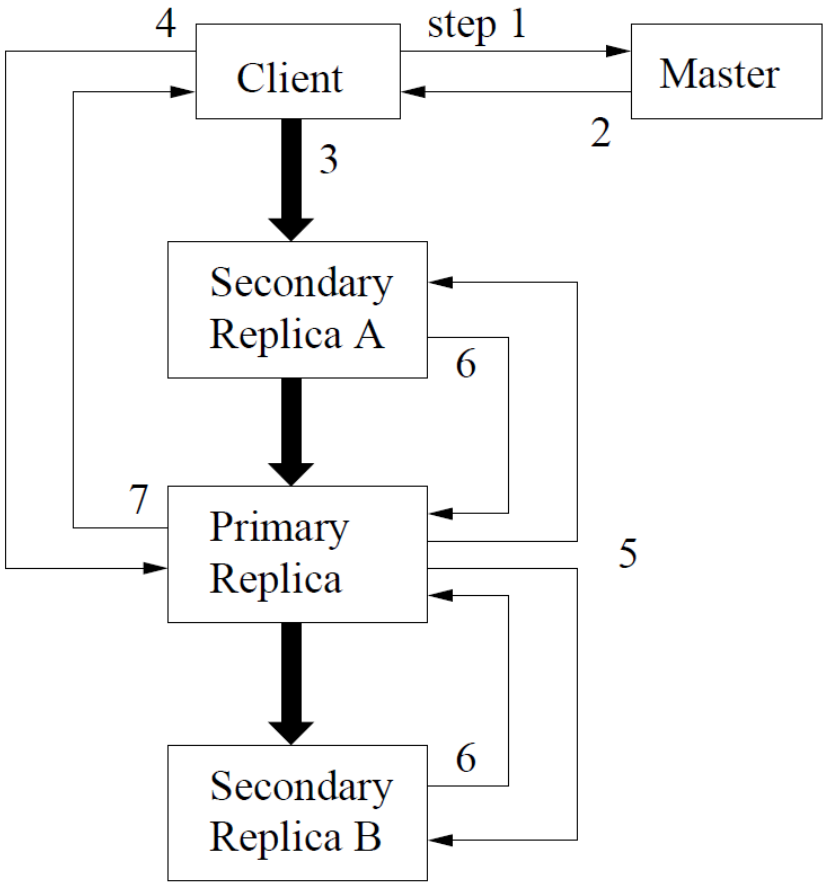
- Write data mutation
  - data is written at an application-specific file offset
- Record append data mutation
  - data ("the record") is appended *atomically at least once* even in the presence of concurrent mutations
  - GFS chooses the offset and returns it to the client
  - GFS may insert padding or record duplicates in between

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Design Decisions: Concurrency Model

- Implications for applications
  - rely on appends rather than overwrites
  - checkpointing
  - application-level checksums
  - writing self-validating, self-identifying records
- Typical use cases (or “hacking around relaxed consistency”)
  - writer generates file from beginning to end and then atomically renames it to a permanent name under which it is accessed
  - writer inserts periodical checkpoints, readers only read up to checkpoint
  - many writers concurrently append to file to merge results, reader skip occasional padding and repetition using checksums

Operations: Writing Files



- client ↔ master (1, 2)
  - chunkserver with chunk lease
  - chunkservers with replicas
- client ⟶ chunkservers
  - push data to chunkservers (3)
  - write request to primary (4)
- primary ⟶ secondary
  - forward write request (5)
- secondary ⟶ primary
  - operation status (6)
- primary ⟶ client
  - operation status

Figure Credit: "The Google File System" by S. Ghemawat, H. Gobioff, and S.-T. Leung, 2003

Operations: Atomic Record Appends

- Follows previous control flow with only little extra logic
  - client pushes data to all replicas **of the last chunk of the file** (3’)
  - client sends its request to the primary replica (4)
- Additionally, primary checks if appending the record to the chunk exceeds the maximum chunk size (64 MB)
  - **yes:** primary and secondary pad the chunk to the maximum size; primary instructs client to **retry** operation on the next chunk
  - **no:** primary appends data to its replica and instructs secondaries to write data at the exact same offset
- To keep worst-case fragmentation low, record appends are restricted to at most one fourth of the maximum chunk size

Operations: Snapshots

- Copies a file or directory (almost) instantaneously and with minimal interruption to ongoing mutations
  - quickly create branch copies of huge data sets
  - checkpointing the current state before experimenting
- Lazy copy-on-write approach
  - upon snapshot request, master first revokes any outstanding leases on the chunks of the files it is about to snapshot
  - after leases are revoked or have expired, operation is logged to disk
  - in-memory state is updated by duplicating metadata of source file or directory tree
  - reference counts of all chunks in the snapshot are incremented by one
  - upon subsequent write request, server detects reference count > 1 and allocates a new chunk by replicating the existing chunk

Namespace Management and Locking

- Differences to traditional file systems
  - no per-directory structures that list files in a directory
  - no support for file or directory aliases, e.g. soft and hard links in Unix
- Namespace implemented as a “flat” lookup table
  - full path name → metadata
  - prefix compression for efficient in-memory representation
  - each “node in the namespace tree” (absolute file or directory path) is associated with a read/write lock
- Each master operation needs to acquire locks before it can run
  - **read locks** on all “parent nodes”
  - **read** or **write lock** on “node” itself
  - file creation does not require a write lock on “parent directory” as there is no such structure
  - note metadata records have locks, whereas data chunks have leases

Replica Placement

- Goals of placement policy
  - distribute data for **scalability**, **reliability** and **availability**
  - maximize **network bandwidth utilization**
- Background: GFS clusters are highly distributed
  - 100s of chunkservers across many racks
  - accessed from 100s of clients from the same or different racks
  - traffic between machines on different racks may cross many switches
  - in/out bandwidth of rack typically lower than within rack
- Possible solution: spread chunks across machines **and** racks
- Selecting a chunkserver
  - place chunks on servers with below-average disk space utilization
  - place chunks on servers with low number of recent writes
  - spread chunks across racks (see above)

Re-replication and Rebalancing

- Master triggers **re-replication** when replication factor drops below a user-specified goal
  - chunkservers becomes unavailable
  - replica is reported corrupted
  - a faulty disk is disabled
  - replication goal is increased
- Re-replication prioritizes chunks with a low replication factor, chunks of live files, and actively used chunks
- Master **rebalances** replicas periodically
  - better disk space utilization
  - load balancing
  - gradually “fill up” new chunkservers

Garbage Collection

- GFS does not immediately reclaim physical storage after a file is deleted
- Lazy garbage collection mechanism
  - master logs deletion immediately by renaming file to a “hidden name”
  - master removes any such hidden files during regular file system scan
- Orphaned chunks
  - chunks that are not reachable through any file
  - master identifies them in regular scan and deletes metadata
  - uses heart beats to inform chunkservers about deletion
- Stale replicas
  - detected based on *chunk version number*
  - chunk version number is increased whenever master grants a lease
  - removed during regular garbage collection

Fault Tolerance

- High availability
  - **fast recovery:** master and chunkserver designed to recover in seconds; no distinction between normal and abnormal termination
  - **chunk replication:** different parts of the namespace can have different replication factors
  - **master replication:** operation log replicated for reliability; mutation is considered committed only once all replicas have written the update; “shadow masters” for read-only access
- Data integrity
  - chunkservers use checksums to detect data corruption
  - idle chunkservers scan and verify inactive chunks and report to master
  - each 64 KB block of a chunk has a corresponding 32 bit checksum
  - if a block does not match its check sum, client is instructed to read from different replica
  - checksums optimized for write appends, not overwrites

Hadoop Distributed File System (HDFS)

- Open-source clone of GFS
  - similar assumptions
  - very similar design and architecture
- Differences
  - no support for random writes, append only
  - emphasizes platform independence (implemented in Java)
  - *possibly,* HDFS does not use a lookup table to manage namespace
  - terminology (see next bullet)
- "Grüezi, redet si Schwyzerdütsch?"
  - namenode → master
  - datanode → chunkserver
  - block → chunk
  - edit log → operation log

HDFS Architecture

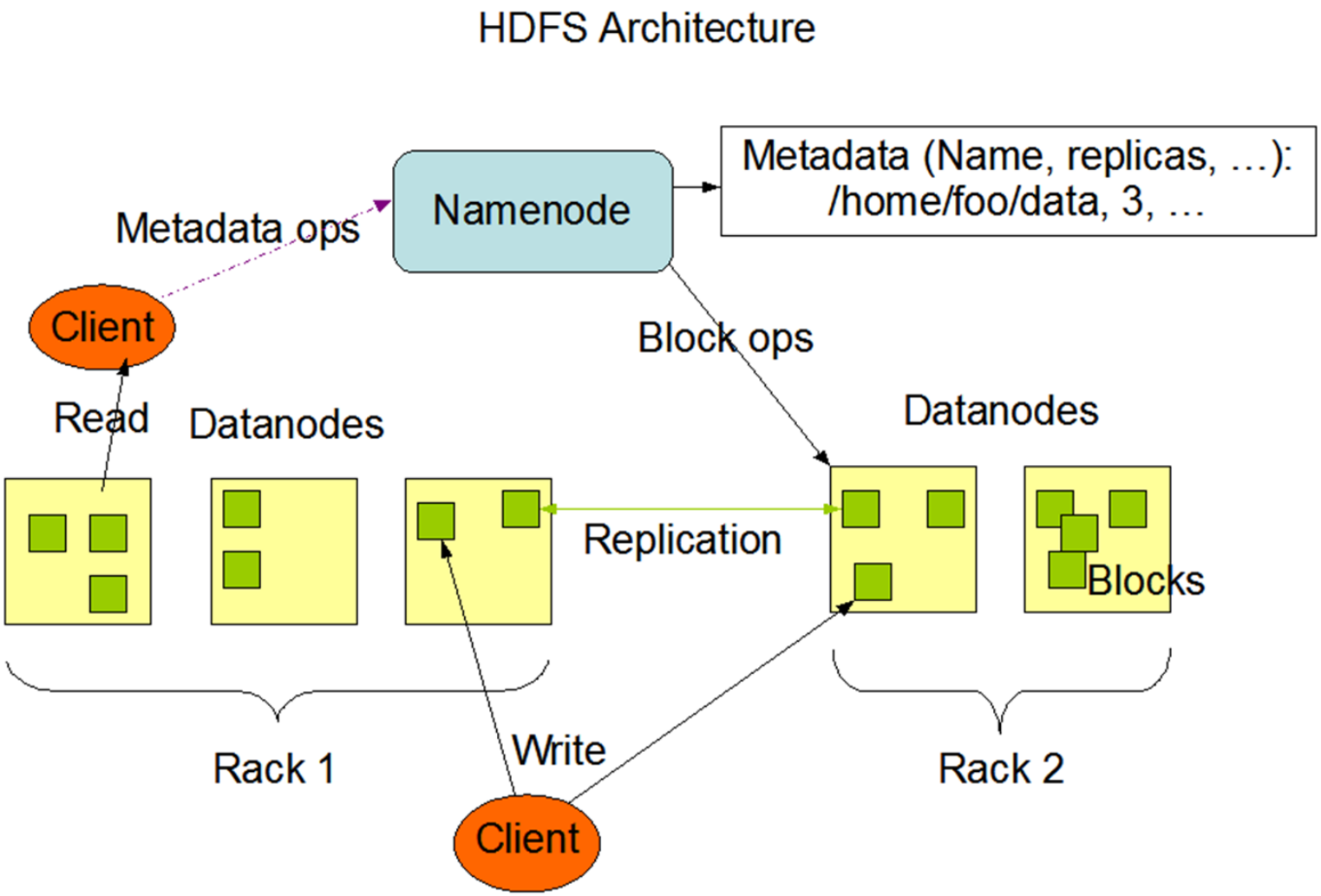


Figure Credit: "HDFS Architecture Guide" by D. Borthakur, 2008

Example Cluster Sizes

- GFS (2003)
  - 227 chunkservers
  - 180 TB available space, 155 TB used space
  - 737k files, 232k dead files, 1550k chunks
  - 60 MB metadata on master, 21 GB metadata on chunkservers
- HDFS (2010)
  - 3500 nodes
  - 60 million files, 63 million blocks, 2 million new files per day
  - 54k block replicas per datanode

  - all 25k nodes in HDFS clusters at Yahoo! provide 25 PB of storage

References

- S. Ghemawat, H. Gobioff, and S.-T. Leung: **The Google File System.** *Proc. Symp. on Operating Systems Principles (SOSP), pp. 29-43, 2003.*
- D. Borthakur: **HDFS Architecture Guide.** *2008.*
- K. Shvachko, H. Kuang, S. Radia, and R. Chansler: **The Hadoop Distributed File System.** *IEEE Symp. on Mass Storage Systems and Technologies, pp.1-10, 2010.*

Data Management in the Cloud

MAP/REDUCE

Map/Reduce

- Programming model
- Examples
- Execution model
- Criticism
- Iterative map/reduce

Motivation

- Background and Requirements
  - computations are conceptually straightforward
  - input data is (very) large
  - distribution over hundreds or thousands of nodes
- Programming model for processing of large data sets
  - abstraction to express simple computations
  - hide details of parallelization, data distribution, fault-tolerance, and load-balancing

Programming Model

- Inspired by primitives from functional programming languages such as Lisp, Scheme, and Haskell
- Input and output are sets of key/value pairs
- Programmer specifies two functions
  - **map** $(k_1, v_1) \rightarrow \texttt{list}(k_2, v_2)$
  - **reduce** $(k_2, \texttt{list}(v_2)) \rightarrow \texttt{list}(v_2)$
- Key and value domains
  - input keys and values are drawn from a different domain than intermediate and output keys and values
  - intermediate keys and values are drawn from the same domain as output keys and values

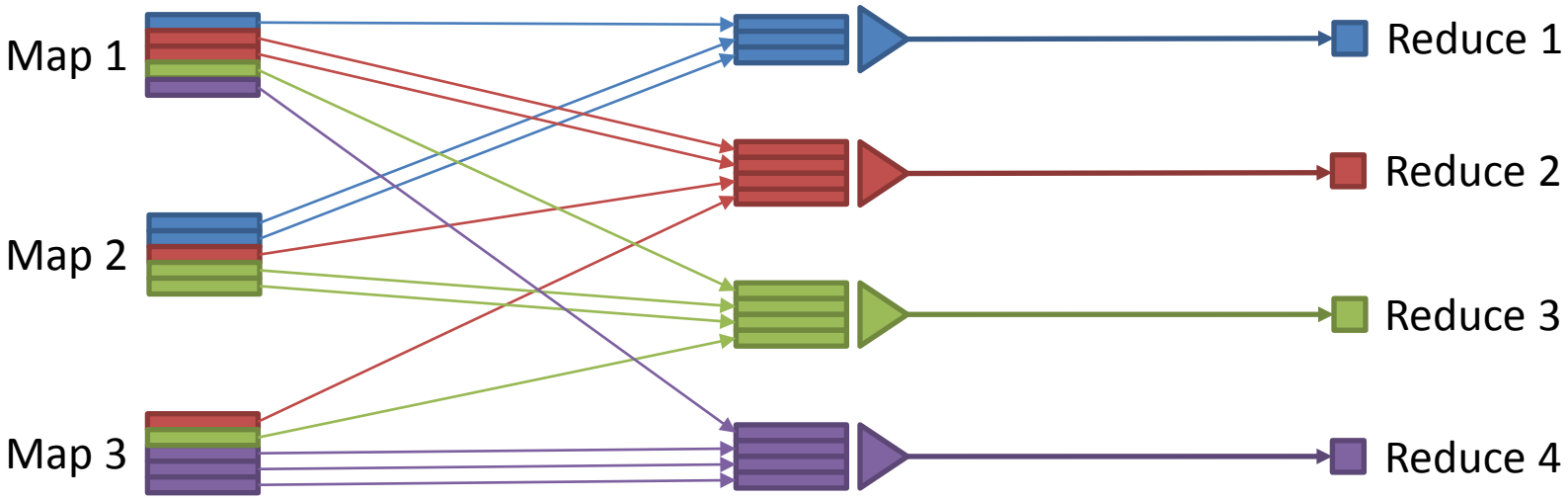
Map Function

- User-defined function
  - processes input key/value pair
  - produces a set of *intermediate* key/value pairs
- Map function I/O
  - **input:** read from GFS file (chunk)
  - **output:** written to intermediate file on local disk
- Map/reduce library
  - executes map function
  - groups together all intermediate values with the same key
  - “passes” these values to reduce functions
- Effect of map function
  - processes and partitions input data
  - builds distributed map (transparent to user)
  - similar to “group by” operation in SQL

Reduce Function

- User-defined function
  - accepts *one* intermediate key and a set of values for that key
  - merges these values together to form a (possibly) smaller set
  - typically, zero or one output value is generated per invocation
- Reduce function I/O
  - **input:** read from intermediate files using remote reads on local files of corresponding mapper nodes
  - **output:** each reducer writes its output as a file back to GFS
- Effect of reduce function
  - similar to aggregation operation in SQL

Map/Reduce Interaction



- Map functions create a user-defined “index” from source data
- Reduce functions compute grouped aggregates based on index
- Flexible framework
  - users can cast raw original data in any model that they need
  - wide range of tasks can be expressed in this simple framework

MapReduce Example

```
map(String key, String value):
   // key:   document name
   // value: document contents
   for each word w in input_value:
      EmitIntermediate(w, “1”);

reduce(String key, Iterator values):
   // key:    word
   // values:
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

More Examples

- Distributed “grep”
  - **goal:** find positions of a pattern in a set of files
  - **map:** (File, String) → list(Integer, String), emits a <line#, line> pair for every line that matches the pattern
  - **reduce:** identity function that simply outputs intermediate values
- Count of URL access frequency
  - **goal:** analyze Web logs and count page requests
  - **map:** (URL, String) → list(URL, Integer), emits <URL, 1> for every occurrence of a URL
  - **reduce:** (URL, list(Integer)) → list(Integer), sums the occurrences of each URL
- Workload of first example is in map function, whereas it is on the reduce in the second example

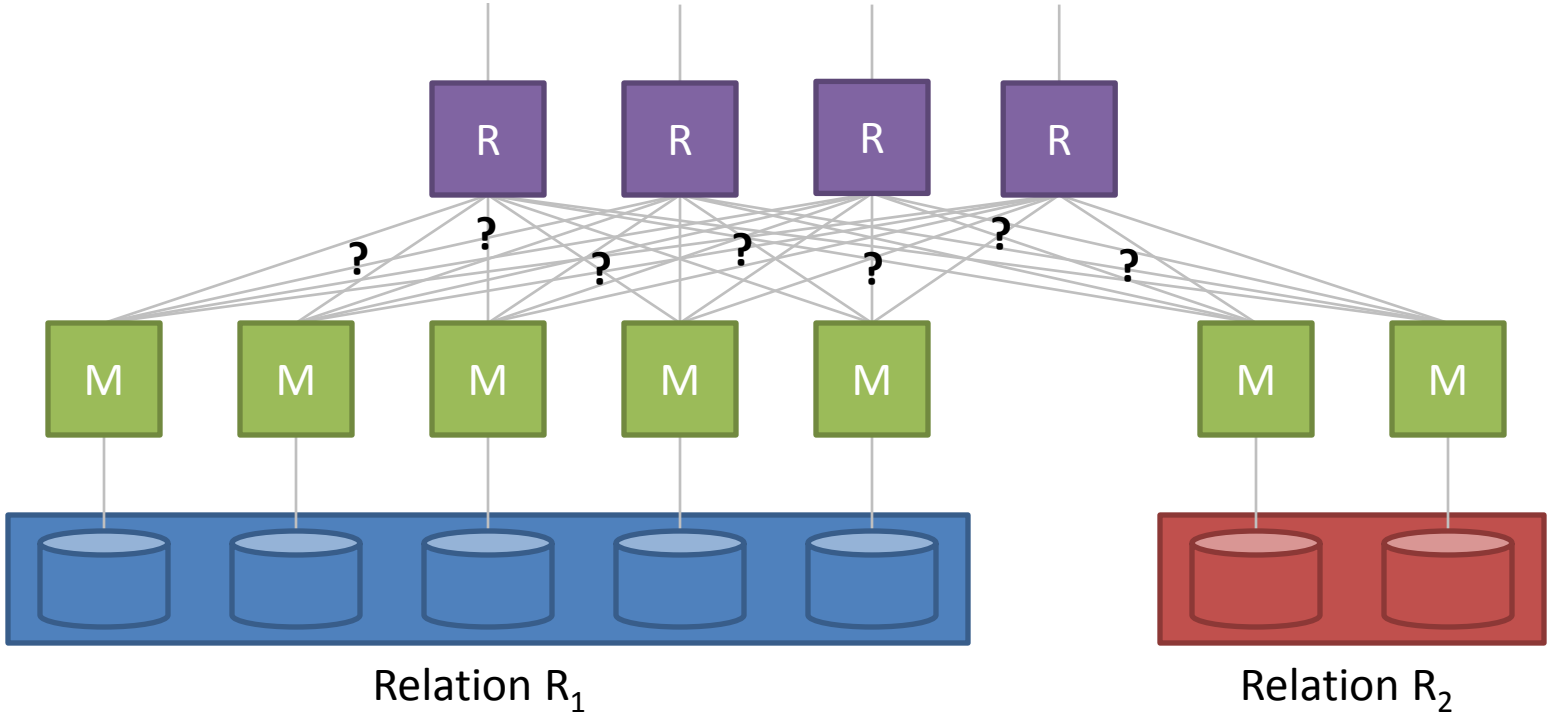
More Examples

- Reverse Web-link graph
  - **goal:** find which source pages link to a target page
  - **map:** (URL, CLOB) → list(URL, URL), parses the page content and emits one <target, source> pair for every target URL found in the source page
  - **reduce:** (URL, list(URL)) → list(URL), concatenates all lists for one source URL
- Term-vector per host
  - **goal:** for each host, construct its term vector as a list of <word, frequency> pairs
  - **map:** (URL, CLOB) → list(String, List), parses the page content (CLOB) and emits a <hostname, term vector> pair for each document
  - reduce: (String, list(List<String, Integer>)) → list(List<String, Integer>), combines all per-document term vectors and emits final <hostname, term vector> pairs

More Examples

- Inverted index
  - **goal:** create an index structure that maps search terms (words) to document identifiers (URLs)
  - **map:** (URL, CLOB) → list(String, URL), parses document content and emits a sequence of <word, document id> pairs
  - **reduce:** (String, list(URL)) → list(URL), accepts all pairs for a given word, and sorts and combines the corresponding document ids
- Distributed sort
  - **goal:** sort "records" according to a user-defined key
  - **map:** (? , Object) → list(Key, Record), extracts the key from each "record" and emits <key, record> pairs
  - **reduce:** emits all pairs unchanged
  - Map/reduce guarantees that pairs in each partition are processed ordered by key, but still requires clever *partitioning function* to work!

Relational Join Example



- Map function **M**: "hash on key attribute"
  - (? , tuple) → list(key, tuple)
- Reduce function **R**: "join on each k value"
  - (key, list(tuple)) → list(tuple)

Implementation

- Based on the “Google computing environment”
  - same assumptions and properties as GFS
  - builds on top of GFS
- Architecture
  - one master, many workers
  - users submit jobs consisting of a set of tasks to a scheduling system
  - tasks are mapped to available workers within the cluster by master
- Execution overview
  - map invocations are distributed across multiple machines by automatically partitioning the input data into a set of *M* splits
  - input splits can be processed in parallel
  - reduce invocations are distributed by partitioning the intermediate key space into *R* pieces using a partitioning function, e.g. “*hash(key)* **mod** *R*”

Execution Overview

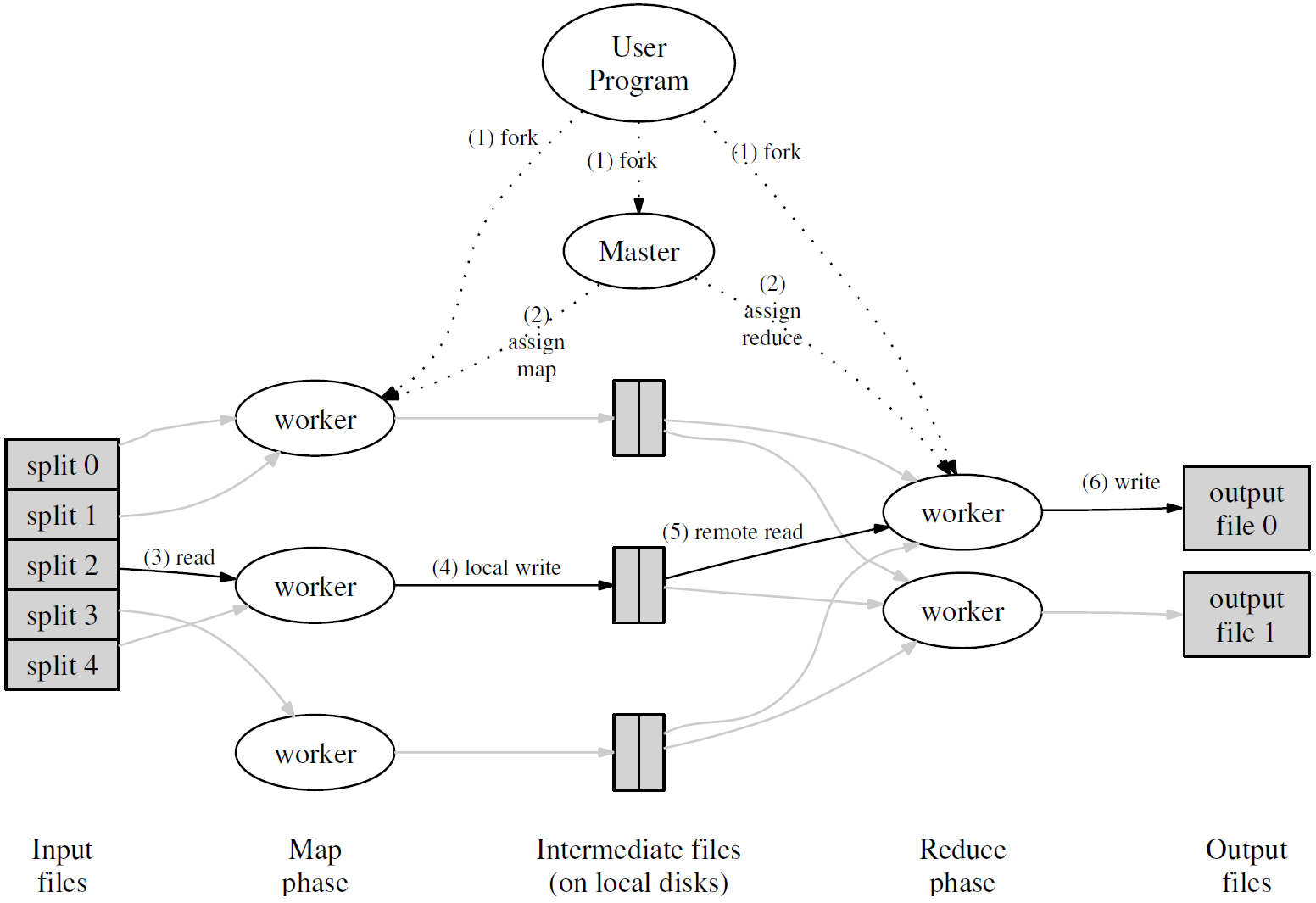


Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Execution Overview

1. Map/reduce library splits input files into *M* pieces and then starts copies of the program on a cluster of machines
2. One copy is the master, the rest are workers; master assigns *M* map and *R* reduce tasks to idle workers
3. Map worker reads its input split, parses out key/value pairs and passes them to user-defined map function
4. Buffered pairs are written to local disk, partitioned into *R* regions; location of pairs passed back to master
5. Reduce worker is notified by master with pair locations; uses RPC to read intermediate data from local disk of map workers and sorts it by intermediate key to group tuples by key
6. Reduce worker iterates over sorted data and for each unique key, it invokes user-defined reduce function; result appended to reduce partition
7. Master wakes up user program after all map and reduce tasks have been completed

Master Data Structures

- Information about all map and reduce task
  - **worker state:** idle, in-progress, or completed
  - **identity** of the worker machine (for non-idle tasks)
- Intermediate file regions
  - propagates intermediate file locations from map to reduce tasks
  - stores locations and sizes of the *R* intermediate file regions produced by each map task
  - updates to this location and size information are received as map tasks are completed
  - information pushed incrementally to workers that have in-progress reduce tasks

Fault Tolerance

- Worker failure
  - master pings workers periodically; assumes failure if no response
  - completed/in-progress map and in-progress reduce tasks on failed worker are rescheduled on a different worker node
  - dependency between map and reduce tasks
  - importance of chunk replicas
- Master failure
  - checkpoints of master data structure
  - “given that there is only a single master, failure is unlikely”
- Failure semantics
  - if user-defined functions are *deterministic*, execution with faults produces the same result as execution without faults
  - rely on atomic commits of map and reduce tasks

More Implementation Aspects

- Locality
  - network bandwidth is scarce resource
  - move computation close to data
  - master takes GFS metadata into consideration (location of replicas)
- Task granularity
  - master makes *O(M + R)* scheduling decisions
  - master stores *O(M * R)* states in memory
  - *M* is typically larger than *R*
- Backup Tasks
  - “stragglers” are a common cause for suboptimal performance
  - as a map/reduce computation comes close to completion, master assigns the same task to multiple workers

Refinements

- Partitioning function
  - default function can be replaced by user
  - supports “application-specific” partitioning
- Ordering guarantees
  - within a give partition, intermediate key/value pairs are processed in increasing key order
- Combiner function
  - addresses significant key repetitions generated by some map functions
  - partial merging of data by map worker, before it is sent over network
  - typically the same code is used as by the reduce function
- Input and output types
  - support to read input and produce output in several formats
  - user can define their own “readers” and “writers”

Refinements

- Skipping bad records
  - map/reduce framework detects on which record task failed
  - when task is restarted this record is skipped
- Local execution
  - addresses challenges debugging, profiling and small-scale testing
  - alternative implementation that executes task sequentially on local machine
- Counters
  - counter facility to count occurrences of various events
  - counter values from worker machines propagated to master
  - master aggregates counters from successful tasks

Performance Experiments

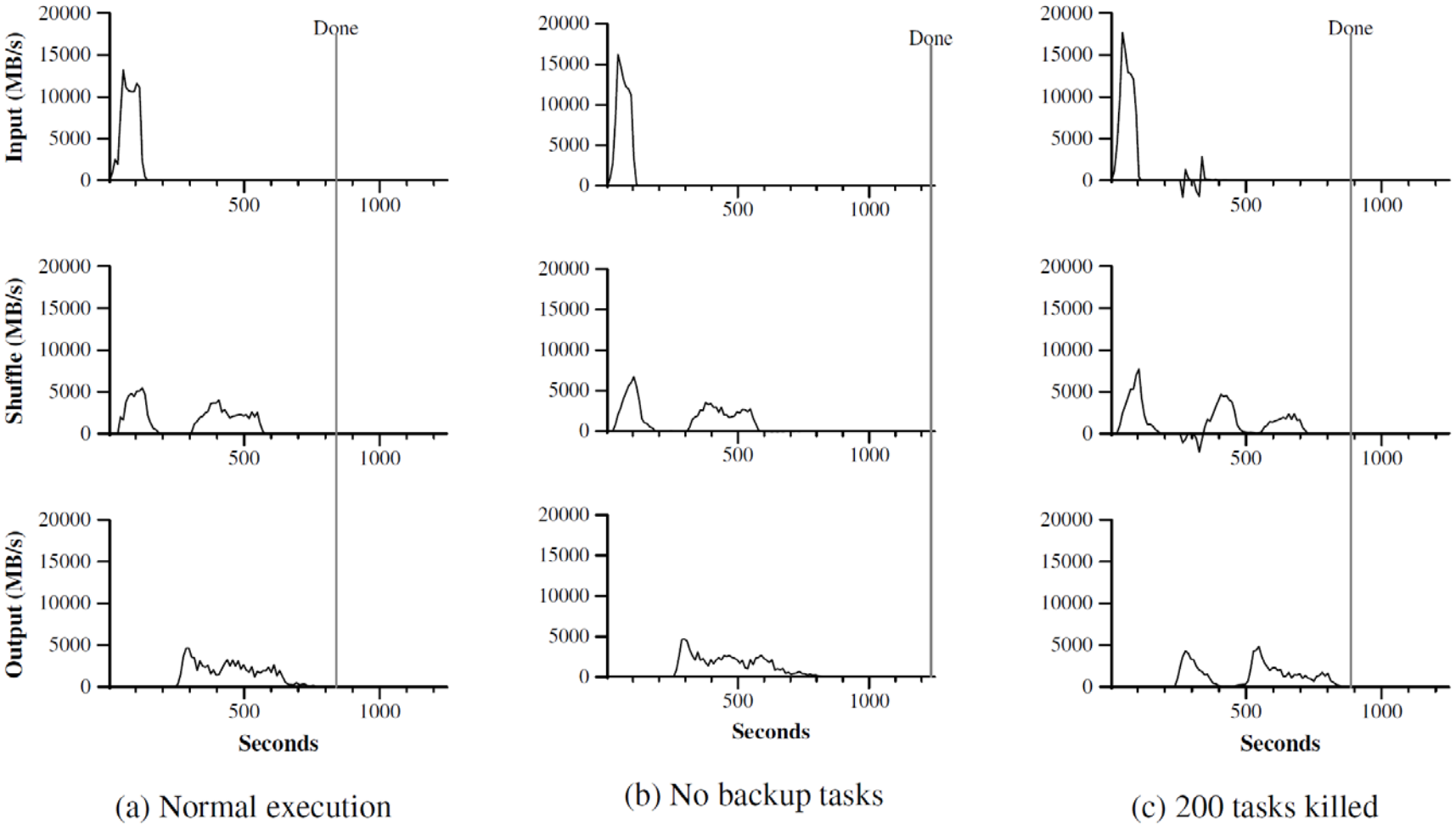


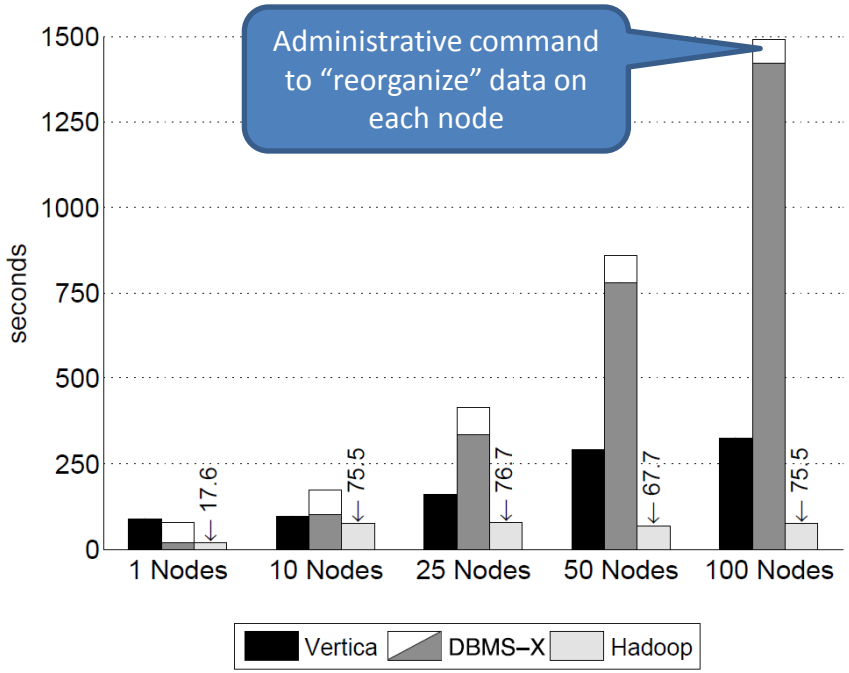
Figure Credit: "MapReduce: Simplified Data Processing on Large Clusters" by J. Dean and S. Ghemawat, 2004

Map/Reduce Criticism

- "Why not use a parallel DBMS instead?"
  - map/reduce is a "giant step backwards"
  - no schema, no indexes, no high-level language
  - not novel at all
  - does not provide features of traditional DBMS
  - incompatible with DBMS tools
- Performance comparison of approaches to large-scale data analysis
  - Pavlo et al. "A Comparison of Approaches to Large-Scale Data Analysis", Proc. Intl. Conf. on Management of Data (SIGMOD), 2009
  - parallel DBMS (Vertica and DBMS-X) vs. map/reduce (Hadoop)
  - original map/reduce task: "grep" from Google paper
  - typical database tasks: selection, aggregation, join, UDF
  - 100-node cluster

Grep Task: Load Times

535 MB/node



1 TB/cluster

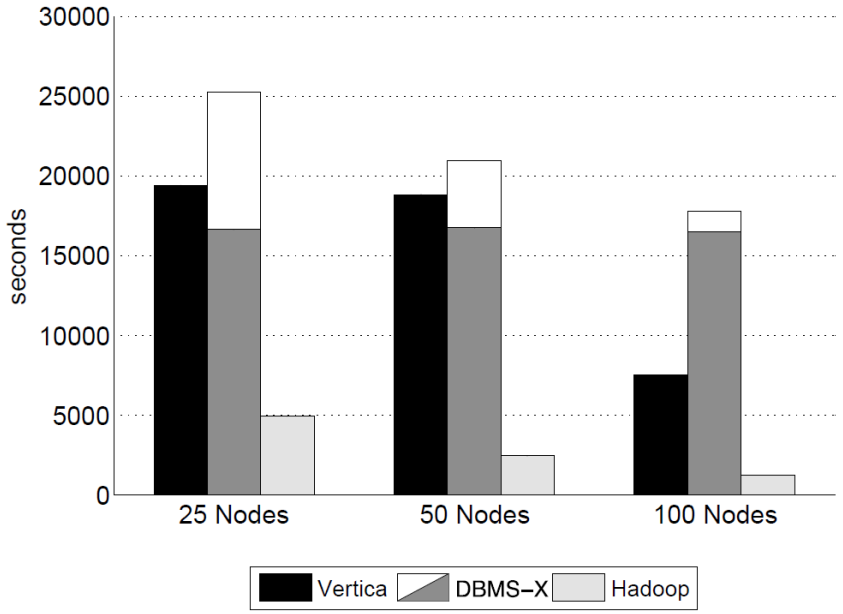
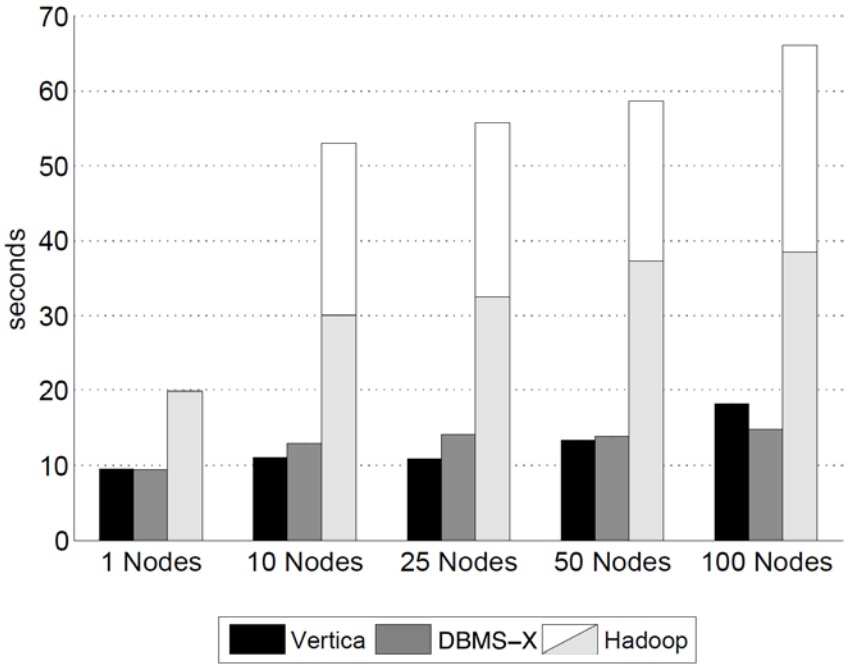


Figure Credit: "A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004

Grep Task: Execution Times

535 MB/node



1 TB/cluster

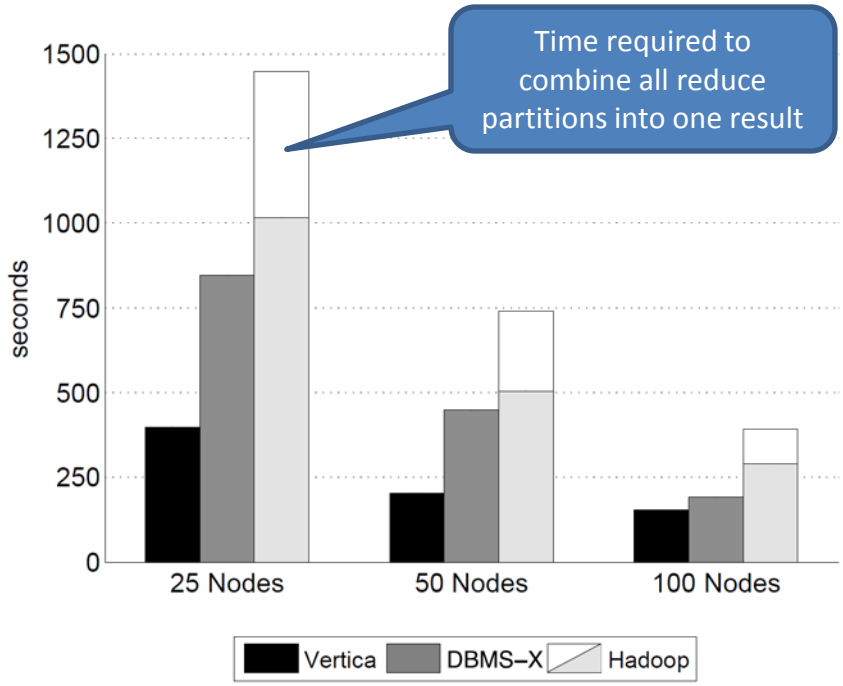


Figure Credit: "A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004

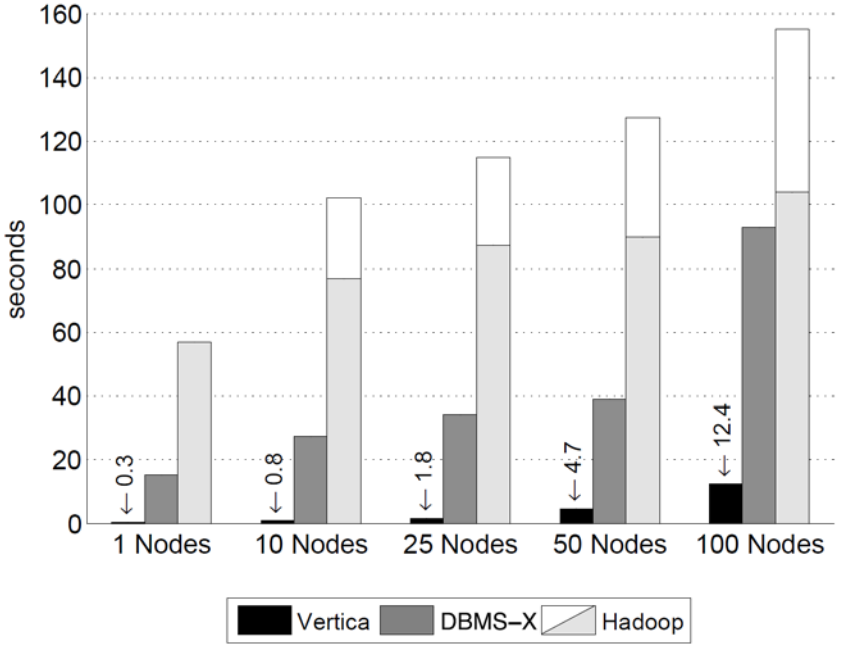
Analytical Tasks

```
CREATE TABLE Documents (
   url VARCHAR(100)
       PRIMARY KEY,
   contents TEXT );

CREATE TABLE Rankings (
   pageURL VARCHAR(100)
           PRIMARY KEY,
   pageRank INT,
   avgDuration INT );
```

```
CREATE TABLE UserVisits (
   sourceIP VARCHAR(16),
   destURL VARCHAR(100),
   visitDate DATE,
   adRevenue FLOAT,
   userAgent VARCHAR(64),
   countryCode VARCHAR(3),
   languageCode VARCHAR(3),
   searchWord VARCHAR(32),
   duration INT );
```

- Data set
  - 600K unique HTML documents
  - 155M user visit records (20 GB/node)
  - 18M ranking records (1 GB/node)

Selection Task



- SQL Query

```
SELECT pageURL, pageRank
FROM Rankings
WHERE pageRank > X
```

- Relational DBMS use index on pageRank column
- Relative performance degrades as number of nodes increases
- Hadoop start-up cost increase with cluster size

Figure Credit: "A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004

Aggregation Task

- Calculate the total ad revenue for each source IP using the user visits table
- **Variant 1:** 2.5M groups

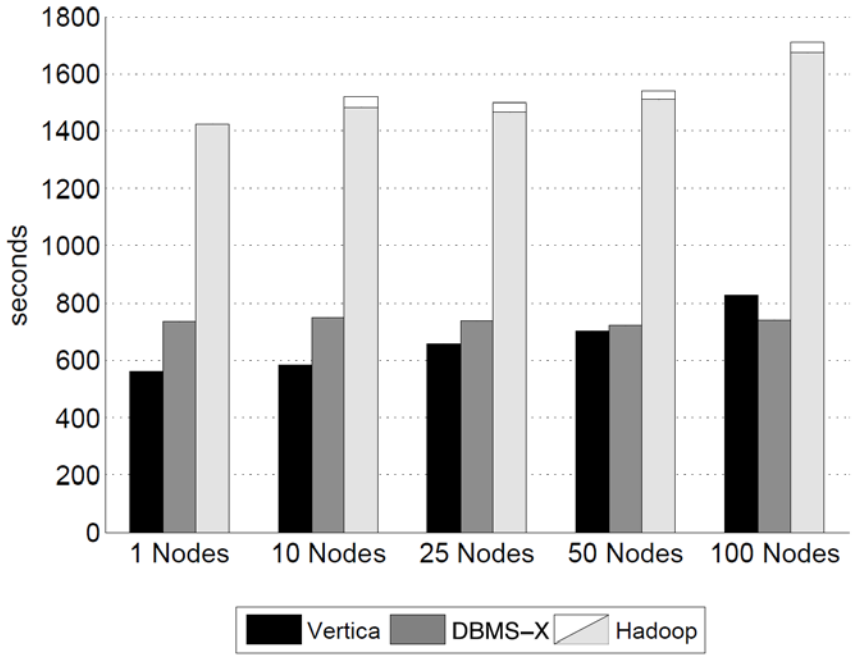
```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits
GROUP BY sourceIP
```

- **Variant 2:** 2,000 groups

```
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
FROM UserVisits
GROUP BY SUBSTR(sourceIP, 1, 7)
```

Aggregation Task

2.5M Groups



2,000 Groups

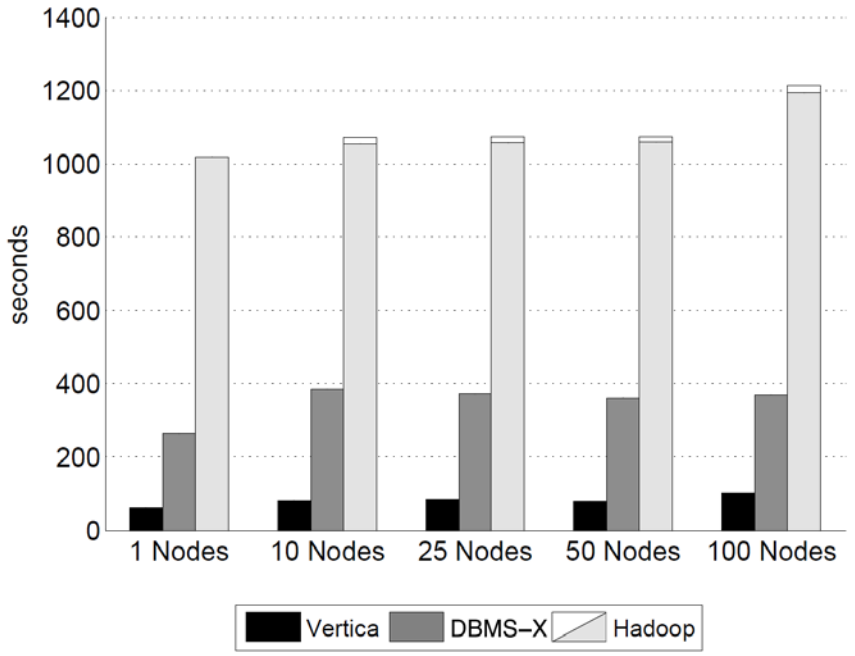


Figure Credit: "A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004

Join Task

SQL Query

```
SELECT INTO Temp
   UV.sourceIP,
   AVG(R.pageRank) AS avgPageRank,
   SUM(UV.adRevenue) AS totalRevenue
FROM
   Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
  AND UV.visitDate BETWEEN
        DATE('2000-01-15') AND
        DATE('2000-01-22')
GROUP BY UV.sourceIP


SELECT sourceIP,
       avgPageRank,
       totalRevenue
FROM Temp
ORDER BY totalRevenue DESC LIMIT 1
```

Map/reduce program

- Uses three phases
  - **Phase 1:** filters records outside date range and joins with rankings file
  - **Phase 2:** computes total ad revenue and average page rank based on source IP
  - **Phase 3:** produces the record with the largest total ad revenue
- Phases run in strict sequential order

Join Task

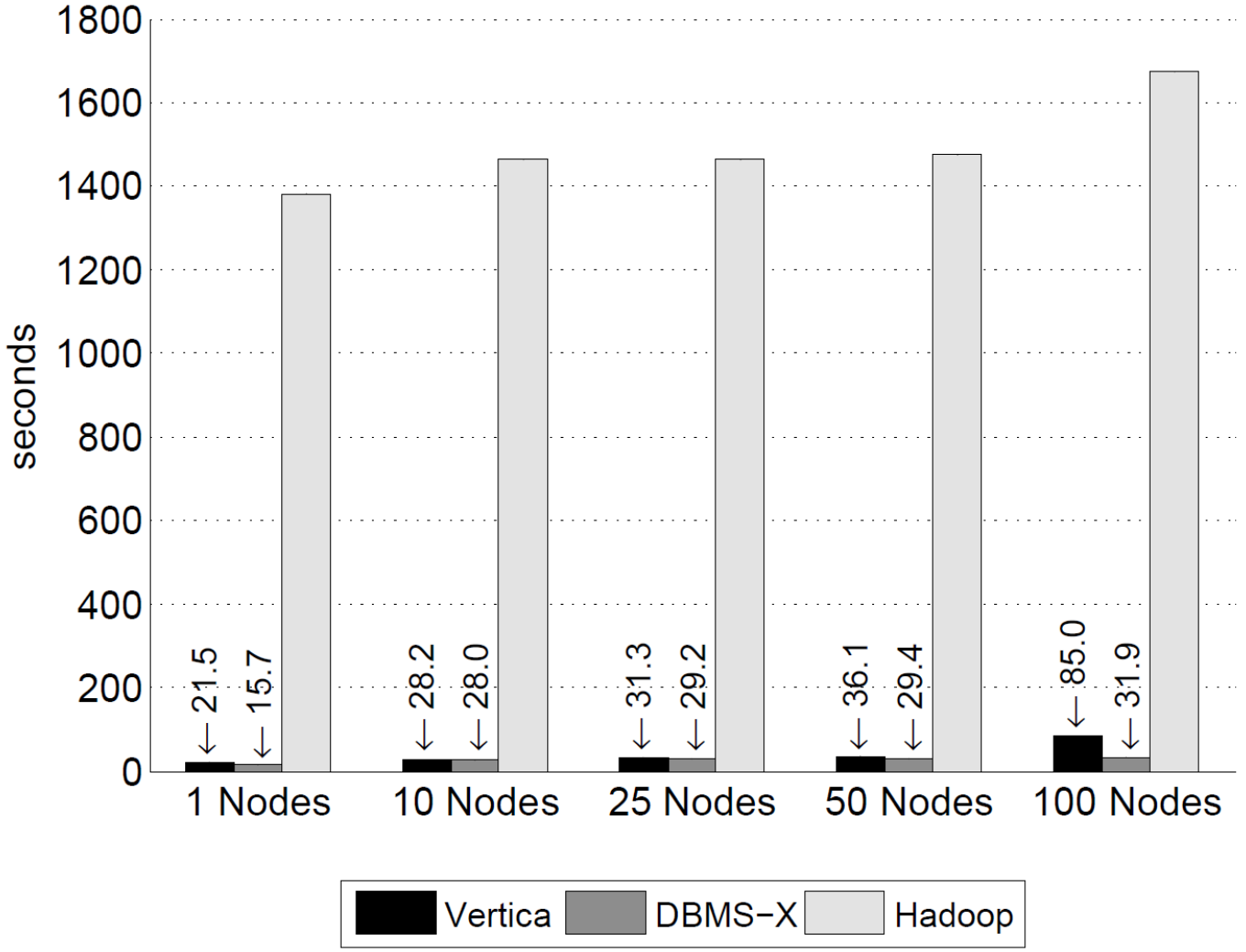


Figure Credit: "A Comparison of Approaches to Large-Scale Data Analysis" by A. Pavlo et al., 2004

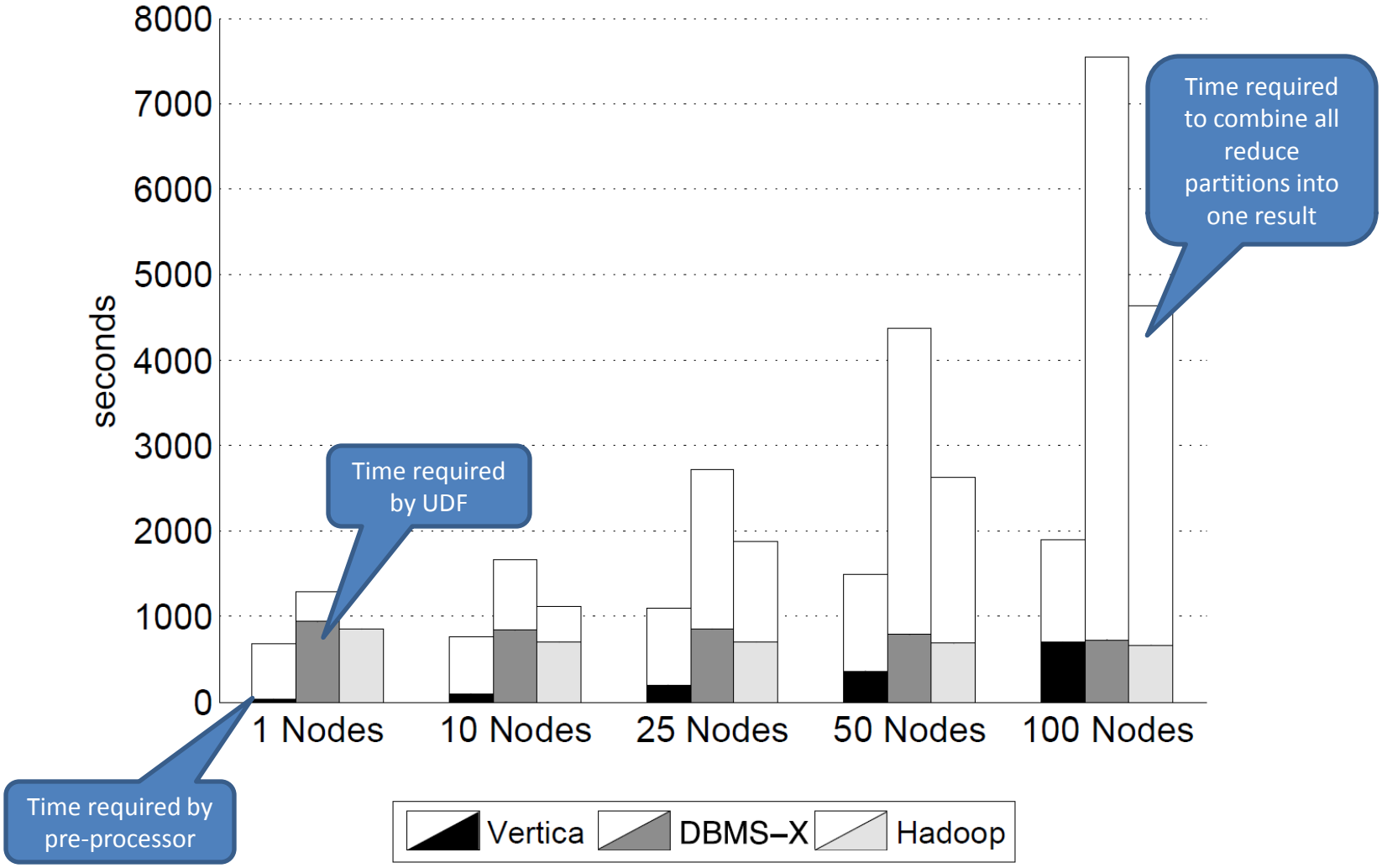
UDF Aggregation Task

- Compute in-link count for each document in the data set
- SQL Query

```
SELECT INTO Temp UDF(contents) FROM Documents
SELECT url, SUM(value) FROM Temp GROUP BY url
```

- Map/reduce program
  - documents are split into lines
  - input key/value pairs: <line number, line contents>
  - **map:** uses regex to find URLs and emits <URL, 1> for each URL
  - **reduce:** counts the number of values for a given key
- Issues
  - DBMS-X: not possible to run UDF over contents stored as BLOB in database; instead UDF has to access local file system
  - Vertica: does not currently support UDF, uses a special pre-processor

UDF Aggregation Task



Map/Reduce vs. Parallel DBMS

- No schema, no index, no high-level language
  - faster loading vs. faster execution
  - easier prototyping vs. easier maintenance
- Fault tolerance
  - restart of single worker vs. restart of transaction
- Installation and tool support
  - easy to setup map/reduce vs. challenging to configure parallel DBMS
  - no tools for tuning vs. tools for automatic performance tuning
- Performance per node
  - results seem to indicate that parallel DBMS achieve the same performance as map/reduce in smaller clusters

Iterative Map/Reduce

- Task granularity
  - *one* map stage followed by *one* reduce stage
  - map stage reads from replicated storage
  - reduce stage writes to replicated storage
- Complex queries typically require several map/reduce phase
  - no fault tolerance between map/reduce phases
  - “data shuffling” between map/reduce phases

PageRank Example

Initial Rank Table R_0

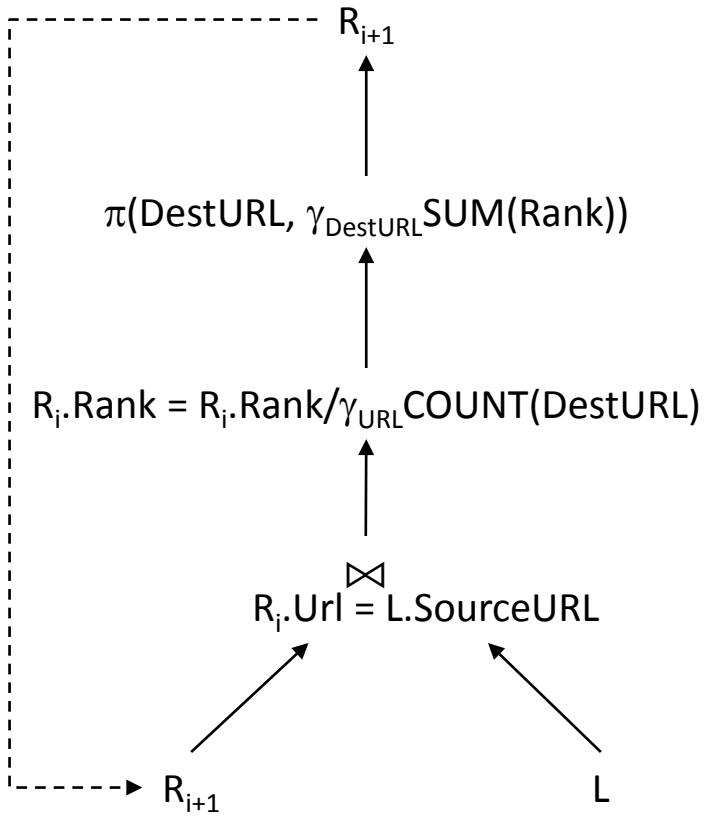
URL	Rank
www.a.com	1.0
www.b.com	1.0
www.c.com	1.0
www.d.com	1.0
www.e.com	1.0

Rank Table R_3

URL	Rank
www.a.com	2.13
www.b.com	3.89
www.c.com	2.60
www.d.com	2.60
www.e.com	2.13

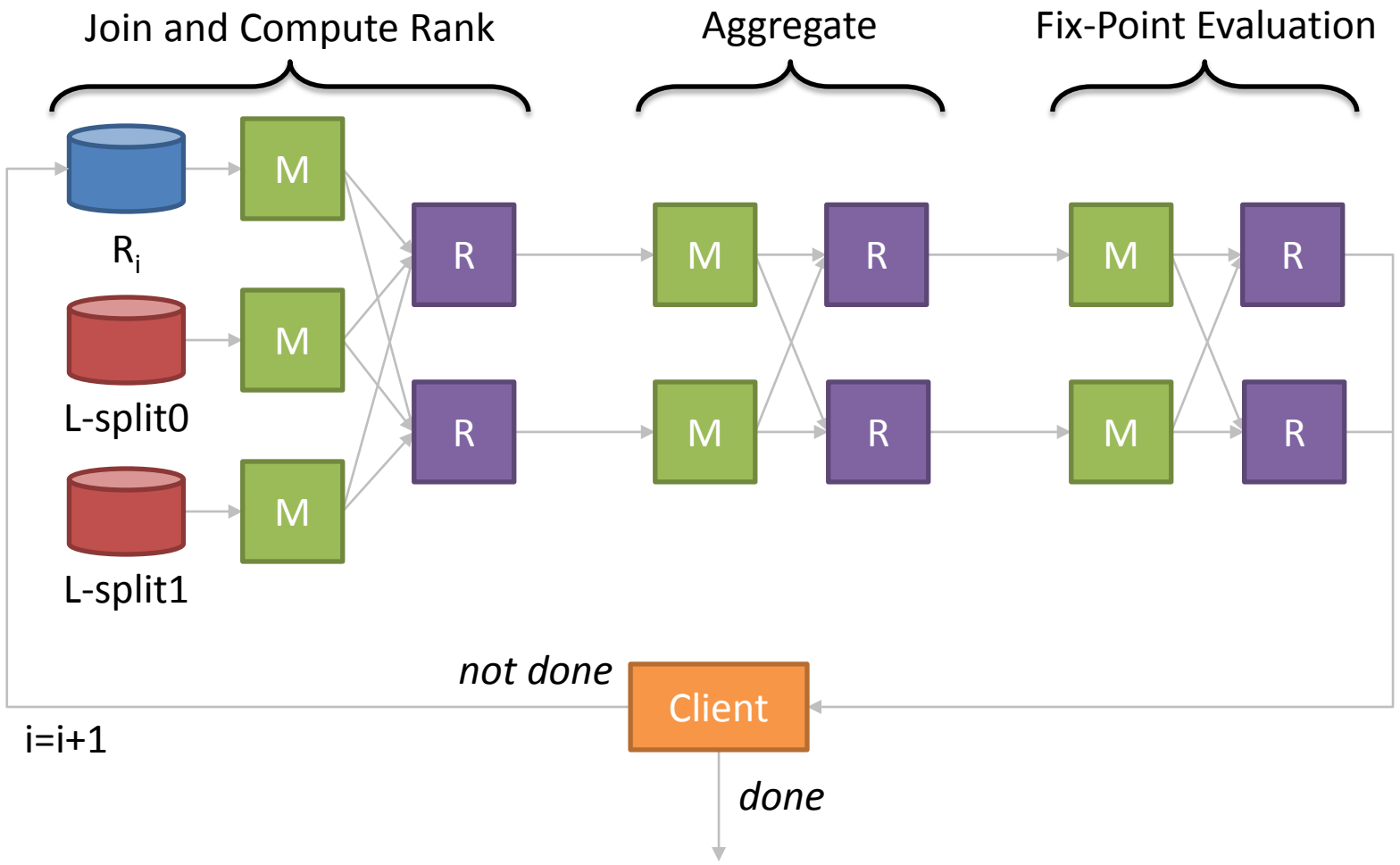
Linkage Table L

SourceURL	DestURL
www.a.com	www.b.com
www.a.com	www.c.com
www.c.com	www.a.com
www.e.com	www.d.com
www.d.com	www.b.com
www.c.com	www.e.com
www.e.com	www.c.com
www.a.com	www.d.com



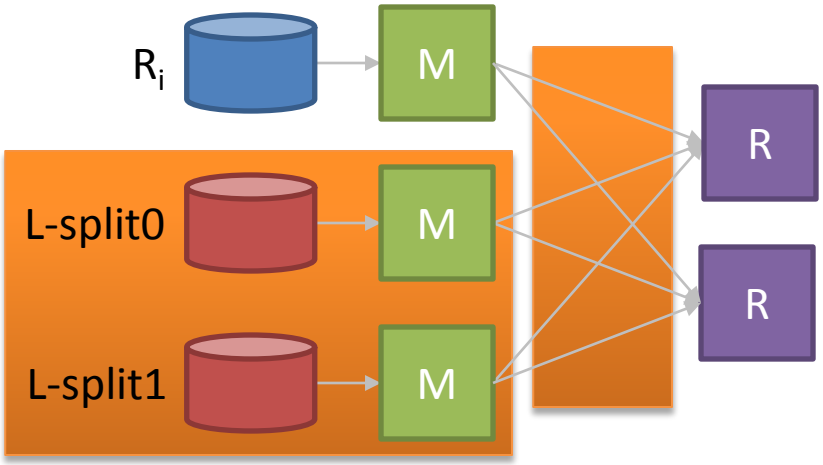
Slide Credit: B. Howe, U Washington

Map/Reduce Implementation



Slide Credit: B. Howe, U Washington

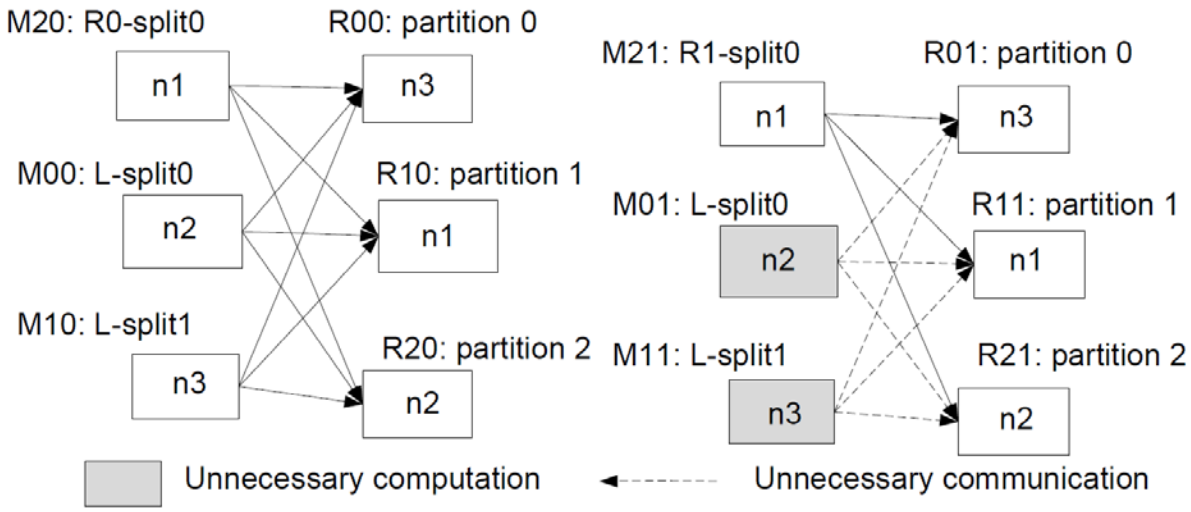
What's the Problem?



- *L* is loaded and shuffled in each iteration
- *L* never changes
- Fix-point evaluated as a separate map/reduce task in each iteration

Slide Credit: B. Howe, U Washington

Inter-Iteration Locality



- Goal of HaLoop scheduler
  - place map and reduce tasks that occur in different iterations but access the same data on the same physical machines
  - thereby increase data re-use between iterations and reduce shuffeling
- Restriction
  - HaLoop requires that the number of reduce tasks is invariant over iterations

Figure Credit: "HaLoop: Efficient Iterative Data Processing on Large Clusters" by Y. Bu et al., 2010

Scheduling Algorithm

```
Input: Node node
// The current iteration's schedule; initially empty
Global variable: Map<Node, List<Partition>> current
// The previous iteration's schedule
Global variable: Map<Node, List<Partition>> previous
 1: if iteration == 0 then
 2:    Partition part = hadoopSchedule(node);
 3:    current.get(node).add(part);
 4: else
 5:    if node.hasFullLoad() then
 6:       Node substitution = findNearestIdleNode(node);
 7:       previous.get(substitution).addAll(previous.remove(node));
 8:       return;
 9:     end if
10:     if previous.get(node).size() > 0 then
11:        Partition part = previous.get(node).get(0);
12:        schedule(part, node);
13:        current.get(node).add(part);
14:        previous.remove(part);
15:     end if
16: end if
```

Same as Hadoop

Find a susbtitution

Iteration-local schedule

Slide Credit: B. Howe, U Washington

Caching and Indexing

- HaLoop caches loop-invariant data partitions on a physical node's local disk to reduce I/O cost
- Reducer input cache
  - enabled if intermediate table is loop-invariant
  - recursive join, PageRank, HITS, social network analysis
- Reducer output cache
  - used to reduce costs of evaluating fix-point termination costs
- Mapper input cache
  - aims to avoid non-local data reads in non-initial iterations
  - K-means clustering, neural network analysis
- Cache reloading
  - host node fails
  - host node has full load and a map or reduce task must be scheduled on a different substitution node

HaLoop Architecture

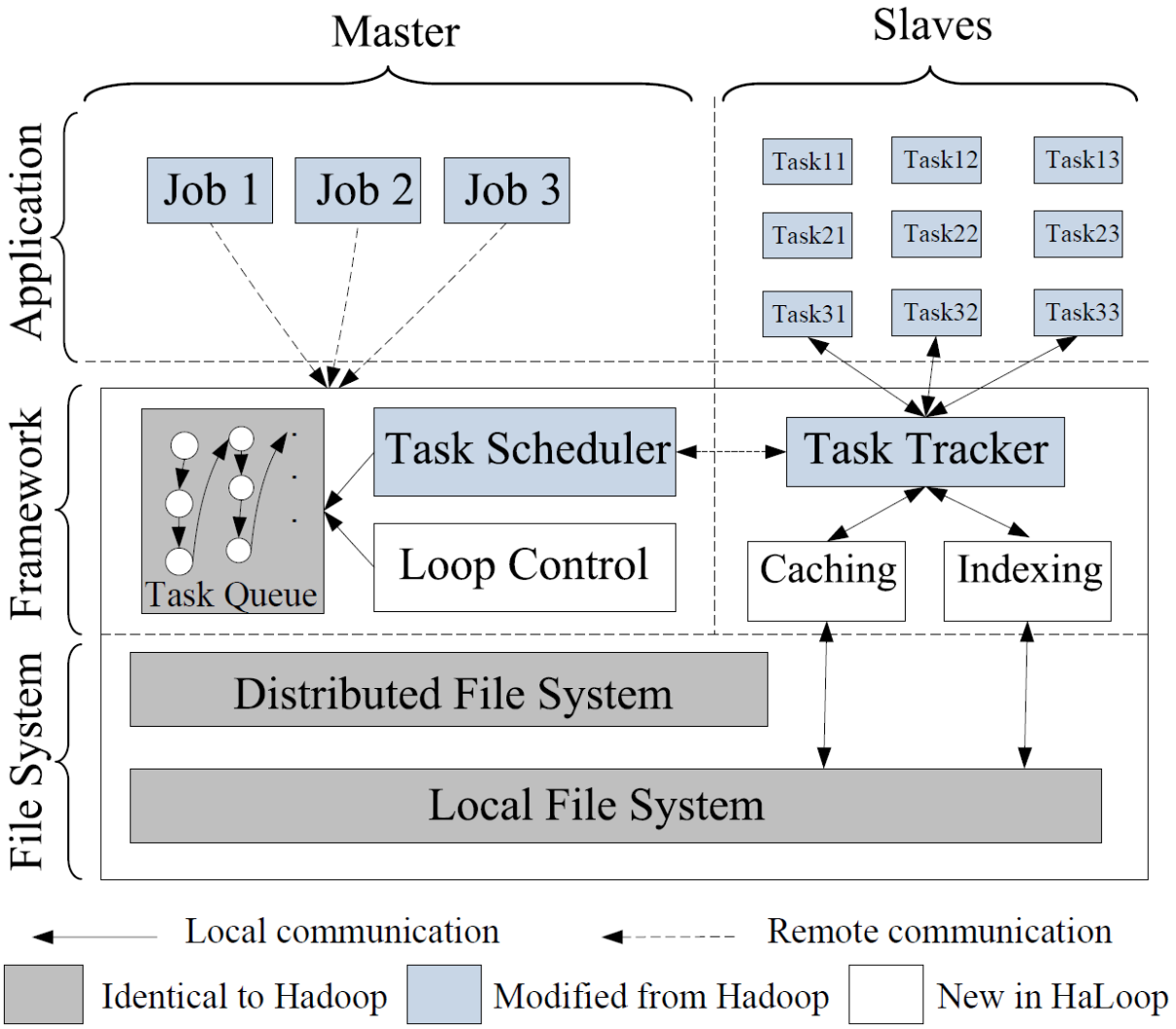


Figure Credit: "HaLoop: Efficient Iterative Data Processing on Large Clusters" by Y. Bu et al., 2010

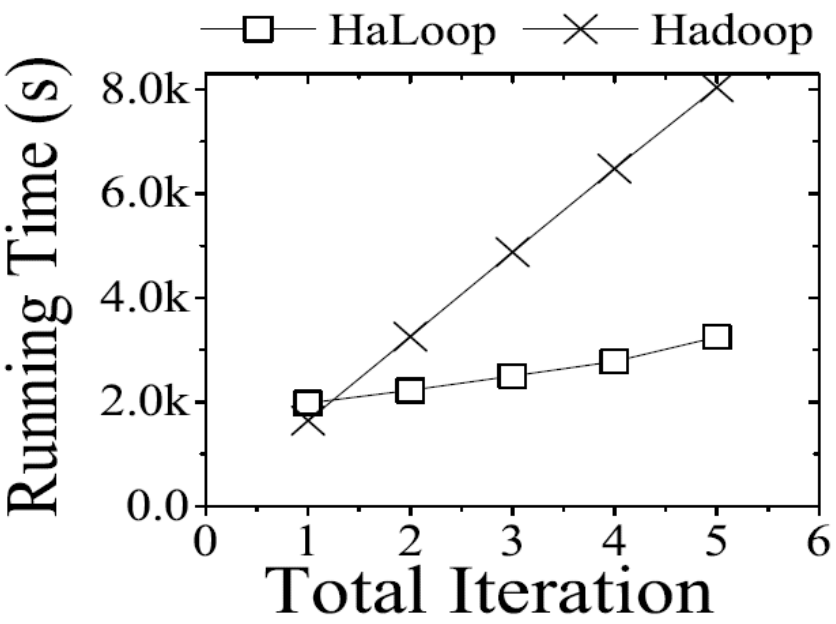
Experiments

- Amazon EC2
  - 20, 50, 90 default small instances
- Datasets
  - billions of triples (120 GB)
  - Freebase (12 GB)
  - Livejournal social network (18 GB)
- Queries
  - transitive closure
  - PageRank
  - k-means

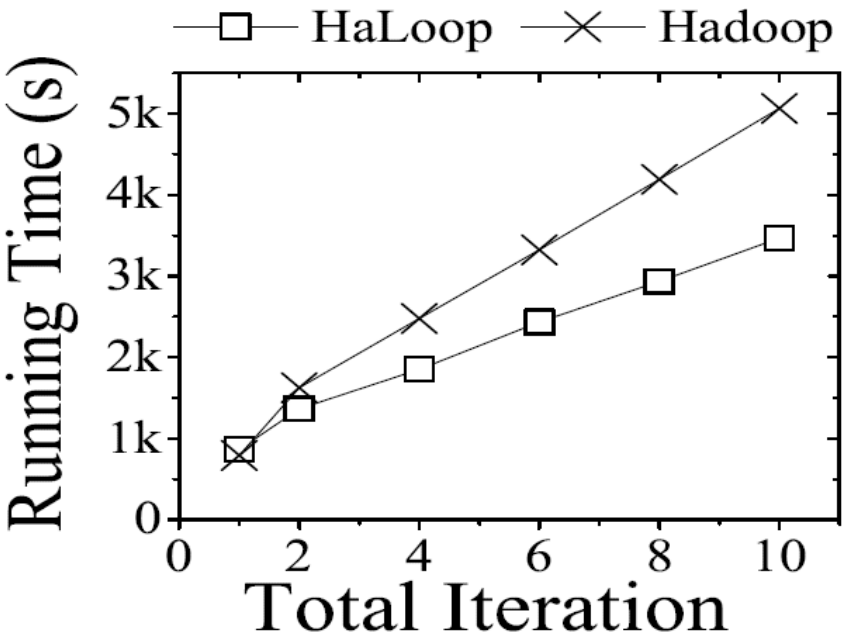
Slide Credit: B. Howe, U Washington

Application Run Time

Transitive Closure
(Triples Dataset, 90 nodes)



PageRank
(Freebase Dataset, 90 nodes)

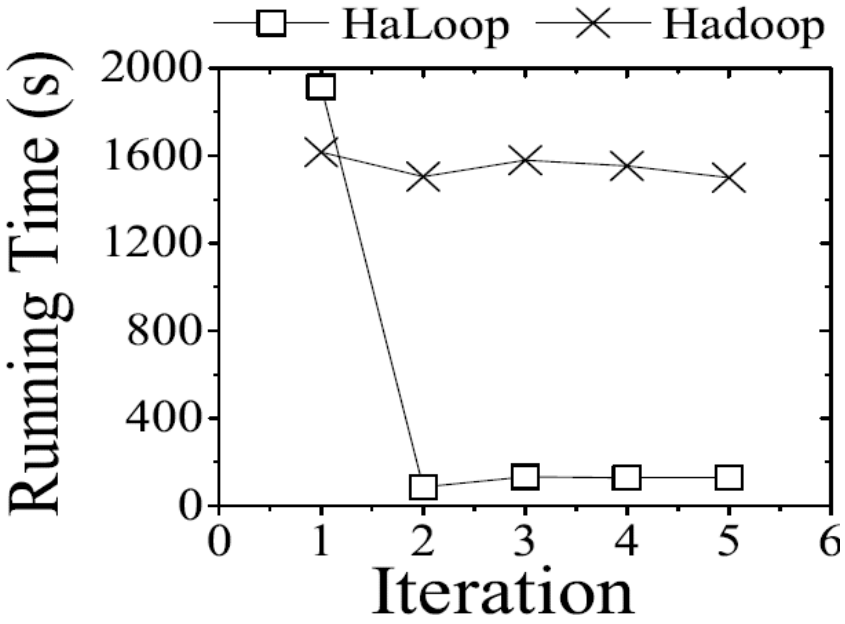


Slide Credit: B. Howe, U Washington

Join Time

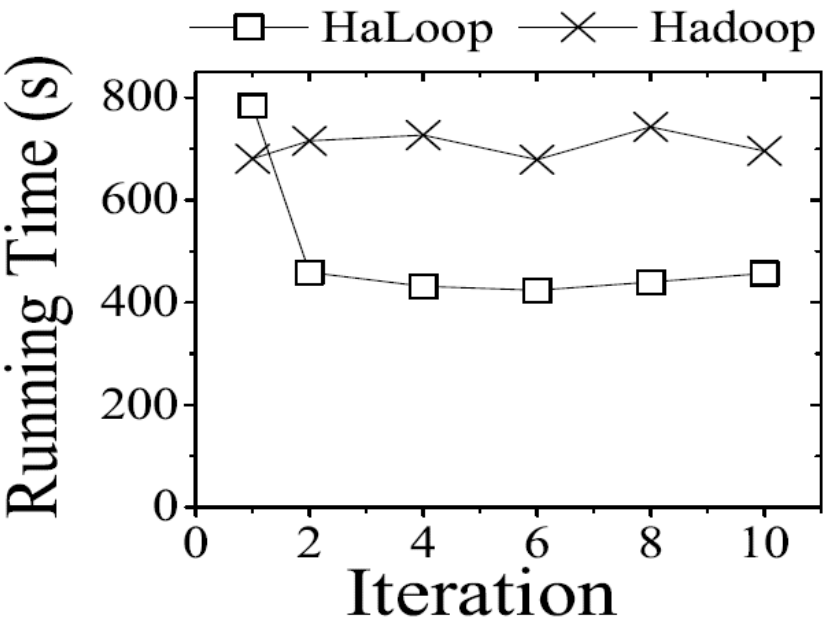
Transitive Closure

(Triples Dataset, 90 nodes)



PageRank

(Freebase Dataset, 90 nodes)

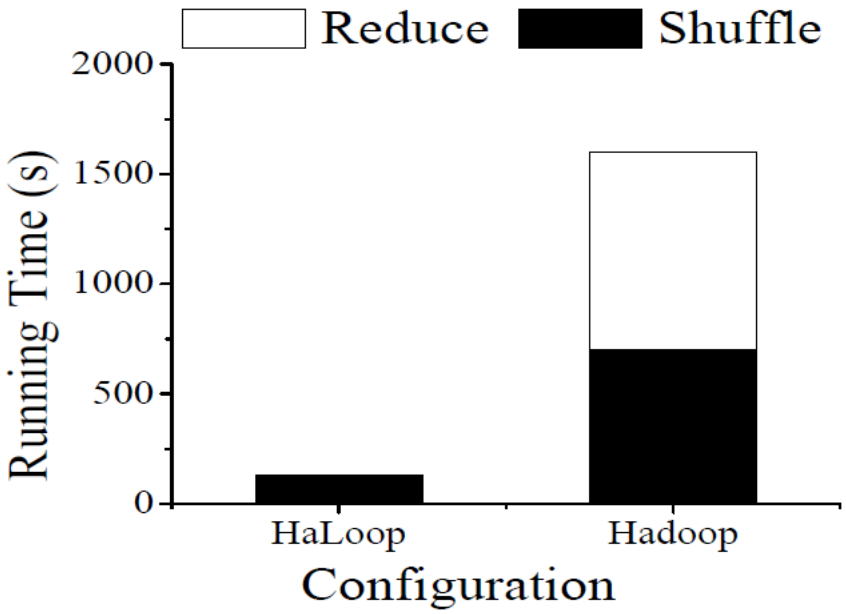


Slide Credit: B. Howe, U Washington

Run Time Distribution

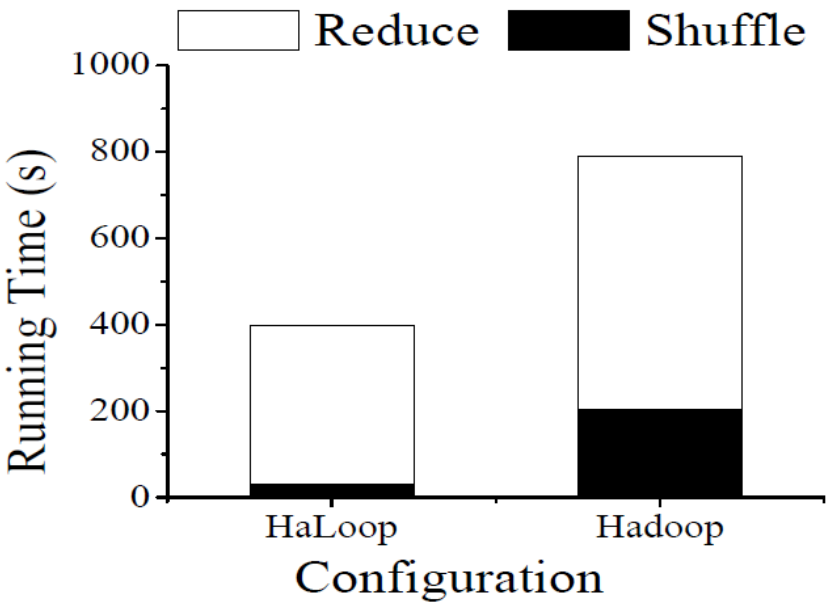
Transitive Closure

(Triples Dataset, 90 nodes)



PageRank

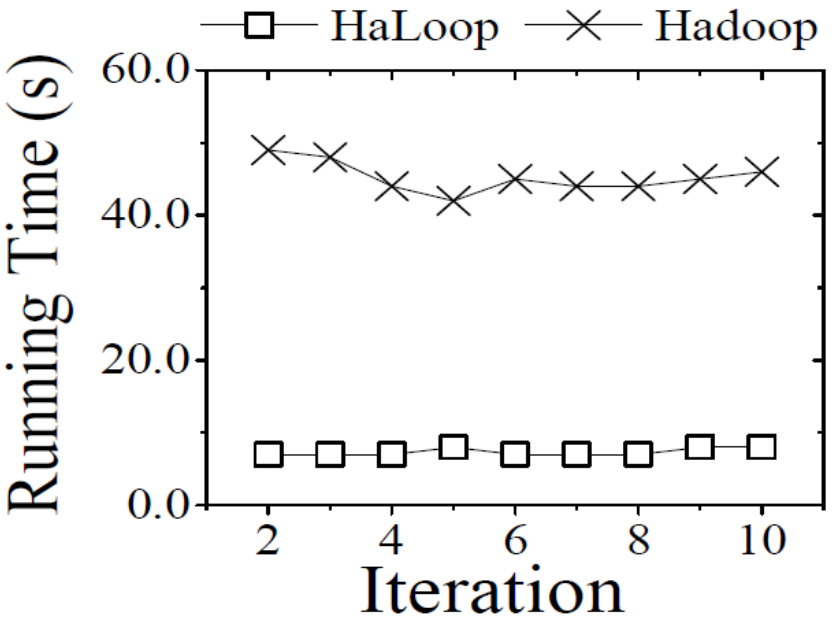
(Freebase Dataset, 90 nodes)



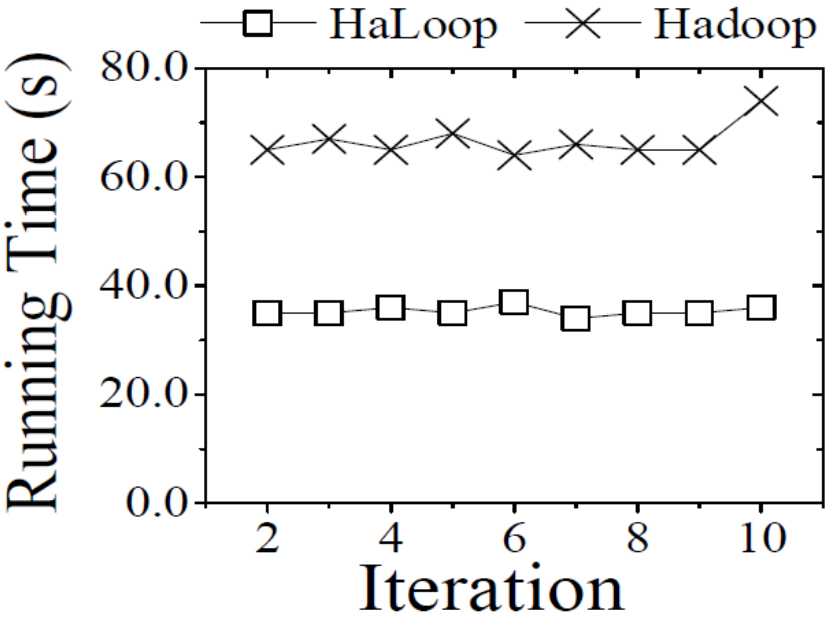
Slide Credit: B. Howe, U Washington

Fix-Point Evaluation

RageRank
(Livejournal Dataset, 50 nodes)



PageRank
(Freebase Dataset, 90 nodes)



Slide Credit: B. Howe, U Washington

References

- J. Dean and S. Ghemawat: **MapReduce: Simplified Data Processing on Large Clusters.** *Proc. Symp. on Opearting Systems Design & Implementation (OSDI), pp. 137-149, 2004.*
- A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker: **A Comparison of Approaches to Large-Scale Data Analysis.** *Proc. Intl. Conf. on Management of Data (SIGMOD), pp. 165-178, 2009.*
- Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst: **HaLoop: Efficient Iterative Data Processing on Large Clusters.** *Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 285-296, 2010.*

Data Management in the Cloud

NEO4J: GRAPH DATA MODEL

Nodes and Relationships

- Nodes
  - have a system-assigned id
  - can have key/value properties
  - there is a *reference node* (“starting point” into the node space)
- Relationships
  - have a system-assigned id
  - are directed
  - have a type
  - can have key/value properties
- Key/value properties
  - values always stored as strings
  - support for basic types and arrays of basic types

Operations

- Nodes are managed using the **`GraphDatabaseService`** interface
  - **`createNode()`** creates and returns a new node
  - **`getNodeById(id)`** returns the node with the given id
  - **`getReferenceNode()`** returns the reference node
  - **`getAllNodes()`** returns an iterator over all nodes
- Relationships are managed using the **`Node`** interface
  - **`createRelationshipTo(target,type)`** creates and returns a relationship
  - **`getRelationships(direction,types)`** returns an iterator over a node's relationships
  - **`hasRelationship(type,direction)`** queries the existence of a certain relationship

Operations

- Relationships are also managed using the **`GraphDatabaseService`** interface
  - **`getRelationshipById(id)`** retrieves a relationship by id
  - but there is no **`getAllRelationships()`** method…
- Node and relationship properties are managed using the **`PropertyContainer`** interface
  - **`setProperty(key,value)`** sets (or creates) a property
  - **`getProperty(key)`** returns a property value (or throws exception)
  - **`hasProperty(key)`** checks if a key/value property exists
  - **`removeProperty(key)`** deletes a key/value property
  - **`getPropertyKeys()`** returns all the keys of a node's properties
- Nodes and relationships are deleted using the corresponding method in the **`Node`** and **`Relationship`** interfaces

Example

```
GraphDatabaseService db = ...
Transaction tx = db.beginTx();
try {
   Node mike = db.createNode();
   mike.setProperty(“name”, “Michael”);
   Node pdx = db.createNode();
   Relationship edge = mike.createRelationshipTo(pdx, LIVES_IN);
   edge.setProperty(“years”, new int[] { 2010, 2011, 2012 });
   for (edge: pdx.getRelationship(LIVES_IN, INCOMING)) {
      Node node = edge.getOtherNode(pdx);
   }
   tx.success();
} catch (Exception e) {
   tx.fail();
} finally {
   tx.finish();
}
```

Transactions

- Unlike other “NoSQL” systems, Neo4j supports transactions and ACID properties
- All modifications to data must be wrapped in transactions
  - default isolation level is **`READ_COMMITTED`**
  - data retrieved by traversals is not protected from modification by other transactions
  - non-repeatable reads may recur as only write locks are held until the end of the transaction
  - it is possible to achieve higher isolation levels by manually acquiring locks on nodes and relationships
  - locks are acquired at the node and relationship level
  - deadlock detection is built into the core transaction management and causes Neo4j to throw an exception

Indexes

- Neo4j does not support any value-based retrieval of nodes and relationships without indexes
- Interface **`IndexManager`** supports the creation of node and relationship indexes
  - **`forNodes(name,configuration)`** returns (or creates) a node index
  - **`forRelationships(name,configuration)`** returns (or creates) a relationship index
- Behind the scenes, Neo4j indexes is based on Apache Lucene as an indexing service
- Values are indexed as strings by default, but a so-called *value context* can be used to support numeric indexing
- Neo4j also supports auto indexers for nodes and relationships

Node Indexes

- Index maintenance
  - **`add(node,key,value)`** indexes the given node based on the given key/value property
  - **`remove(node)`** removes all index entries for the given node
  - **`remove(node,key)`** removes all index entries for the given node with the given key
  - **`remove(node,key,value)`** removes a key/value property from the index for the given node
- Index lookups
  - **`get(key,value)`** supports equality index lookups
  - **`query(key,query)`** does a query-based index lookup for one key
  - **`query(query)`** does a query-based index lookup for arbitrary keys

Example

```
Index<Node> people = db.index().forNodes(“people_idx”);

// do an exact lookup
Node mike = people.get(“name”, “Michael”).getSingle();

// do a query-based lookup for one key
for (Node node: people.query(“name”, “M* OR m*”)) {
   System.out.println(node.getProperty(“name”);
}

// do a general query-based lookup
for (Node node: people.query(“name:M* AND title:Mr”) {
   System.out.println(node.getId());
}
```

Relationship Indexes

- Index maintenance is analogous to node indexes
- Additional index lookup functionality
  - **`get(key,value,source,target)`** does an exact lookup for the given key/value property, taking the given source and target node into account
  - **`query(key,query,source,target)`** does a query-based lookup for the given key, taking the given source and target node into account
  - **`query(query,source,target)`** does a general query-based lookup, taking the given source and target node into account

Example

```
Index<Node> homes = db.index().forRelationships("homes_idx");

// do an exact lookup
Relationship r = homes.get("span", "2", mike, pdx).getSingle();

// do a query-based lookup for one key
for (Relationship r: homes.query("span", "*", mike, null)) {
   System.out.println(r.getOtherNode(mike));
}

// do a general query-based lookup
for (Relationship r:
         homes.query("type:LIVES_IN AND span:3", mike, null) {
   System.out.println(r.getOtherNode(mike));
}
```

Traversal Framework

- Neo4j provides a traversal interface to specify navigation through a graph
  - based on callbacks
  - executed lazily on demand
- Main concepts
  - **expanders** define what to traverse, typically in terms of relationships direction and type
  - the **order** guides the exploration, i.e. depth-first or breadth-first
  - **uniqueness** indicates whether nodes, relationships, or paths are visited only once or multiple times
  - an **evaluator** decides what to return and whether to stop or continue traversal beyond the current position
  - a **starting node** where the traversal will begin

Example: Finding Bridges

```
List<Relationship> result = ...
Set<Node> roots = ...

IndexManager manager = this.database.index();
Index<Node> dfsNodes = manager.forNodes("dfsNodes");
RelationshipIndex treeEdges = manager.forRelationships("treeEdges");

TraversalDescription traversal = new TraversalDescriptionImpl();
traversal = traversal.order(Traversal.postorderDepthFirst());
traversal = traversal.relationships(EDGE, OUTGOING);

int treeId = 0;
while (!roots.isEmpty()) {
   Node root = roots.iterator().next();
   Traverser traverser = traversal.traverse(root);
   int pos = 0;
   for (Node node : traverser.nodes()) {
      dfsNodes.add(node, P_DFSPOS, treeId + ":" + pos);
      roots.remove(node);
      pos++;
   }
   for (Relationship relationship : traverser.relationships()) {
      treeEdges.add(relationship, P_ID, relationship.getId());
   }
   result.addAll(this.tarjan(dfsNodes, treeEdges, treeId));
   treeId++;
}
```

Graph Algorithms

- Some common graph algorithms are directly supported
  - all **shortest paths** between two nodes up to a maximum length
  - **all paths** between two nodes up to a maximum depth
  - all **simple paths** between two nodes up to a maximum length
  - **"cheapest" path** based on Dijkstra or A*
- Class **`GraphAlgoFactory`** provides methods to create **`PathFinder`**s that implement these algorithms

Example: Shortest Path

```
// unweighted case
PathFinder<Path> pathFinder = GraphAlgoFactory.shortestPath(
      Traversal.expanderForTypes(EDGE, OUTGOING),
      Integer.MAX_VALUE);
Path path = pathFinder.findSinglePath(source, target);
for (Node node: path.nodes()) {
   System.out.println(node);
}

// weighted case
PathFinder<WeightedPath> pathFinder = GraphAlgoFactory.dijkstra(
      Traversal.expanderForTypes(EDGE, OUTGOING), P_WEIGHT);
Path path = pathFinder.findSinglePath(source, target);
for (Relationship relationship: path.relationships()) {
   System.out.println(relationship);
}
```

Queries

- Support for the Cypher graph query language has recently been added to Neo4j
- Unlike the imperative graph scripting language Gremlin, Cypher is a declarative language
- Cypher is comprised of four main concepts
  - **START**: starting points in the graph, obtained by element IDs or via index lookups
  - **MATCH**: graph pattern to match, bound to the starting points
  - **WHERE**: filtering criteria
  - **RETURN**: what to return
- Implemented using the Scala programming language

Example: Average Path Length and Diameter

```
// start n=(nodes_idx, “id:*”)
// match (n)-[tc:TC_EDGE]->(x)
// return max(tc.weight), sum(tc.weight), count(tc.weight)

ExecutionEngine engine = new ExecutionEngine(db);
CypherParser parser = new CypherParser();
Query query = parser.parse("start n=(" + IDX_NODES + ",\"" + P_ID
      + ":*\") match (n)-[tc:" + TC_EDGE.name()
      + "]->(x) return max(tc." + P_WEIGHT + "), sum(tc." + P_WEIGHT
      + "), count(tc." + P_WEIGHT + ")");
ExecutionResult result = engine.execute(query);
Float max = (Float) this.getResultValue(result,
      "max(tc." + P_WEIGHT + ")");
Float sum = (Float) this.getResultValue(result,
      "sum(tc." + P_WEIGHT + ")");
Integer count = (Integer) this.getResultValue(result,
      "count(tc." + P_WEIGHT + ")");
Double[] value = new Double[] { max.doubleValue(),
      sum.doubleValue() / count.intValue() };
```

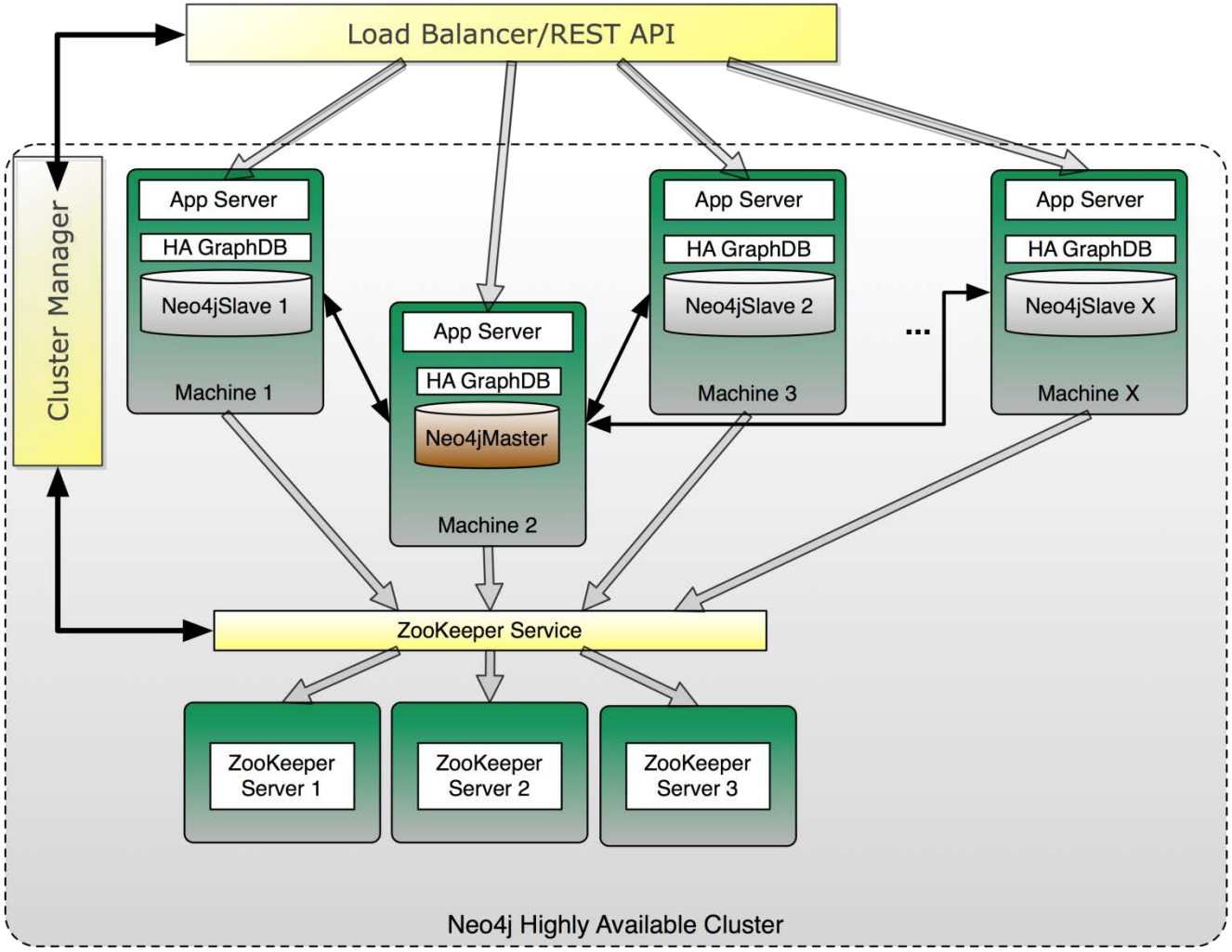
Deployments

- Several deployment scenarios are supported
- Embedded database
  - wraps around a local directory
  - implements the **`GraphDatabaseService`** interface
  - runs in the same process as application, i.e. no client/server overhead
- Client/server mode
  - server runs as a standalone process
  - provides Web-based administration
  - communicates with clients through REST API
- High availability setup
  - one master and multiple slaves, coordinated by ZooKeeper
  - supports fault tolerance and horizontal scaling
  - implements the **`GraphDatabaseService`** interface

REST API

- Functionality of REST API is analogous to the functionality of the Java API
  - http://localhost:7474/db/data/node (nodes)
  - http://localhost:7474/db/data/relationship (relationships)
  - http://localhost:7474/db/data/types (relationship types)
  - http://localhost:7474/db/data/node/927/properties (node properties)
  - http://localhost:7474/db/data/relationship/339/properties
  - http://localhost:7474/db/data/index (indexes)
  - http://localhost:7474/db/data/node/54/traverse (traversals)
  - http://localhost:7474/db/data/node/7311/path (algorithms)
- JSON documents are used to transfer graph data between client and server
- Give it a rest...

High Availability Setup



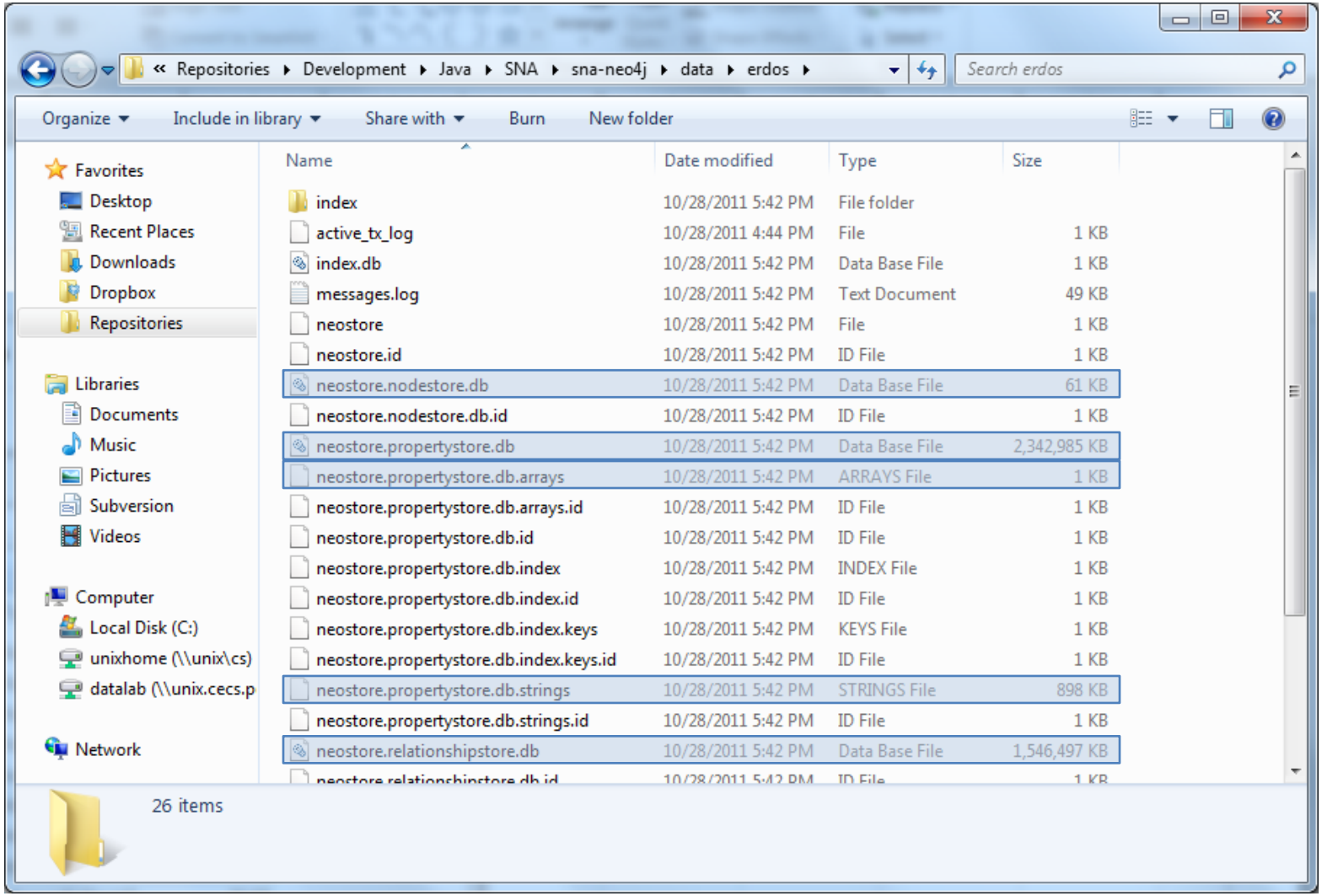
High Availability Setup

- High availability
  - reads are highly available
  - updates to master are replicated asynchronously to slaves
  - updates to slaves are replicated synchronously to master
  - transactions are atomic, consistent and durable on the master, but eventually consistent on slaves
- Fault tolerance
  - depending on ZooKeeper setup, Neo4j can continue to operate from any number of machines down to a single machine
  - machines will be reconnected automatically to the cluster whenever the issue that caused the outage (network, maintenance) is resolved
  - if the master fails a new master will be elected automatically
  - if the master goes down any running write transaction will be rolled back and during master election no write can take place

Data Storage and Memory Management

- A Neo4j graph database consists of several files
  - **`neostore.nodestore.db`** (9 bytes per node)
  - **`neostore.relationshipstore.db`** (33 bytes per relationship)
  - **`neostore.propertystore.db`**
  - **`neostore.propertystore.db.strings`**
  - **`neostore.propertystore.db.arrays`**
- Data in memory is managed in two ways
  - **memory mapped** database files (see above)
  - **object caches** that contain Java representations of node and edges
- Performance tuning
  - Java heap size and garbage collector can be configured
  - sizes of memory mapped files can be configured
  - type of cache can be configured, i.e. none, soft, weak, or strong

Tuning


« Repositories ▸ Development ▸ Java ▸ SNA ▸ sna-neo4j ▸ data ▸ erdos ▸
Search erdos
Organize
Include in library
Share with
Burn
New folder
Favorites
Desktop
Recent Places
Downloads
Dropbox
Repositories
Libraries
Documents
Music
Pictures
Subversion
Videos
Computer
Local Disk (C:)
unixhome (\\unix\cs)
datalab (\\unix.cecs.p
Network
Name
Date modified
Type
Size
index 10/28/2011 5:42 PM File folder
active_tx_log 10/28/2011 4:44 PM File 1 KB
index.db 10/28/2011 5:42 PM Data Base File 1 KB
messages.log 10/28/2011 5:42 PM Text Document 49 KB
neostore 10/28/2011 5:42 PM File 1 KB
neostore.id 10/28/2011 5:42 PM ID File 1 KB
neostore.nodestore.db 10/28/2011 5:42 PM Data Base File 61 KB
neostore.nodestore.db.id 10/28/2011 5:42 PM ID File 1 KB
neostore.propertystore.db 10/28/2011 5:42 PM Data Base File 2,342,985 KB
neostore.propertystore.db.arrays 10/28/2011 5:42 PM ARRAYS File 1 KB
neostore.propertystore.db.arrays.id 10/28/2011 5:42 PM ID File 1 KB
neostore.propertystore.db.id 10/28/2011 5:42 PM ID File 1 KB
neostore.propertystore.db.index 10/28/2011 5:42 PM INDEX File 1 KB
neostore.propertystore.db.index.id 10/28/2011 5:42 PM ID File 1 KB
neostore.propertystore.db.index.keys 10/28/2011 5:42 PM KEYS File 1 KB
neostore.propertystore.db.index.keys.id 10/28/2011 5:42 PM ID File 1 KB
neostore.propertystore.db.strings 10/28/2011 5:42 PM STRINGS File 898 KB
neostore.propertystore.db.strings.id 10/28/2011 5:42 PM ID File 1 KB
neostore.relationshipstore.db 10/28/2011 5:42 PM Data Base File 1,546,497 KB
26 items

Tuning

- Optimizing for traversals
  - memory map as much as possible of the node and relationship database file
  - set object cache type to soft
  - garbage collection issue may occur under high load if frequently accessed paths do not fit into memory
- Optimizing for high throughput property access
  - memory map as much as possible of the property database files
  - set object cache type to weak
- Optimizing for graphs that fit into memory
  - fully memory map all database files
  - set object cache type to strong

Example (Assuming Java Heap Space 3GB)

```
// betweenness centrality
neostore.nodestore.db.mapped_memory=1M
neostore.relationshipstore.db.mapped_memory=1234M
neostore.propertystore.db.mapped_memory=814M
neostore.propertystore.db.strings.mapped_memory=1M
neostore.propertystore.db.arrays.mapped_memory=0M
cache_type=weak

// bridges
neostore.nodestore.db.mapped_memory=1M
neostore.relationshipstore.db.mapped_memory=750M
neostore.propertystore.db.mapped_memory=500M
neostore.propertystore.db.strings.mapped_memory=1M
neostore.propertystore.db.arrays.mapped_memory=0M
cache_type=soft
```

Tuning Guidelines

Primitives	RAM Size	Heap Size	RAM for OS	Memory Mapped
10M	2GB	512MB	*the rest*	100-512MB
100M	8GB+	1-4GB	1-2GB	*the rest*
1B+	16-32GB+	4GB+	1-2GB	*the rest*

- Or… Buy a solid state drive!
  - "with a solid state drive the heap settings can be configured lower since disk access isn't as expensive, thus making caching less important and memory mapping more important"

Array Databases

David Maier

Portland State
UNIVERSITY

SciDB

A Different Point in Data Space

- Many of the big-data approaches we've considered have been for web-data ...

  ... or web analytics
- Often huge numbers of modest-sized items
- Array data management directed at huge individual items

  Single item may need 100s of nodes

Lots of Science Data is Arrays

- Remote imaging (up and down)
- Tomographic reconstructions
- Computational simulation outputs
- In-situ sensing
- Next-Generation Sequencing

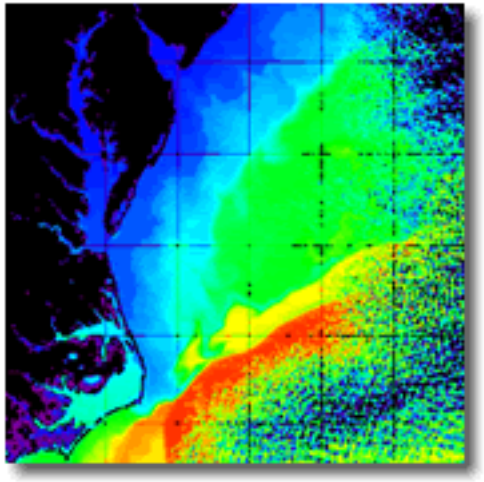
Also business apps: finance, pharma

Implicit Information in the Structure

- Logical organization of an array can indicate order, adjacency, correlation
- However, meaning is different for different arrays

Example: Image Data

- Might have two dimensions corresponding to latitude and longitude
  - Neighboring entries adjoin in space
  - Lose information if you rearrange rows or columns
  - Operations – smoothing, edge detection, object extraction



NOAA CoastWatch

Example: Bi-gram Frequencies

- Entries are bi-gram frequencies
  - A(i, j) = number of times word i precedes word j in some corpus of text
  - Adjacency doesn't mean much: OK to permute rows and columns (in the same way)
  - Operations: row or column correlations; matrix multiplication

$$P = \begin{bmatrix} 5 & 0 & 8 & 0 & 0 & 7 & 0 & 4 & 0 \\ 4 & 0 & 2 & 9 & 0 & 5 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 & 3 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 6 & 0 & 9 & 4 & 0 & 7 \\ 0 & 3 & 0 & 5 & 0 & 0 & 8 & 0 & 9 \end{bmatrix}$$

Example: Sequencing Data

- Have 2-D array, indexed by sample ID and DNA base position
  - Array element is a read call (A C G T N) and a confidence
  - Sample order could be shuffled, but not order of reads
  - Operations: aggregate (across base position or whole array); “array induction” – count values for x in every $b_1b_2xb_3b_4$, indexed by (b_1, b_2, b_3, b_4)

Support for Array Storage

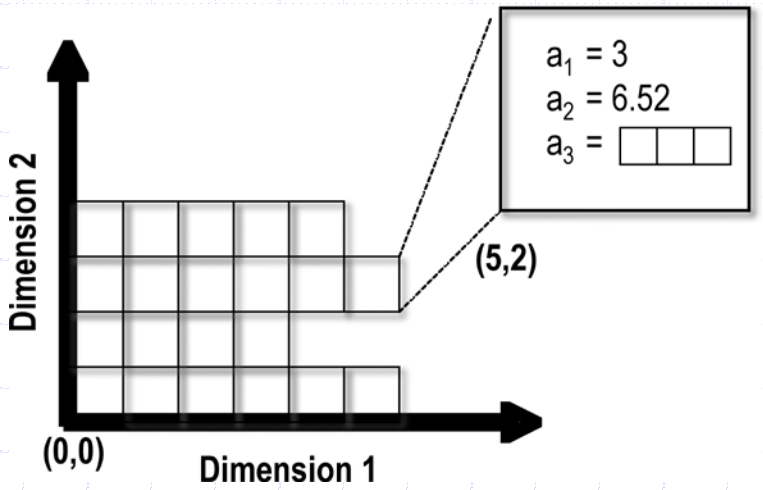
- netCDF, HDF, other interchange formats
- Rasdaman – rasters over DBMS
- SQL 1-D arrays
- RAM Layer on MonetDB
- SciDB – relatively new effort

Variations in Array Models

- Scalar or complex elements
  - Records
  - Nested arrays
- “Ragged” boundaries
- Special values
- Non-integer dimensions
- Updates vs. versions

SciDB Data Model

- Nested multi-dimensional arrays
  - Cells can be tuples or other arrays
  - Can have non-integer dimensions
- Additional “History” dimension on updatable arrays
- Ragged arrays allow each row or column to have a different length
- Support for multiple flavors of “null”
  - Array cells can be ‘EMPTY’
  - User-definable treatment of special values



SciDB DDL

```
CREATE ARRAY Test_Array
     < A: integer NULLS,
       B: double,
       C: USER_DEFINED_TYPE >
     [I=0:99999,1000,10, J=0:99999,1000,10 ]
     PARTITION OVER ( Node1, Node2, Node3 )
     USING block_cyclic();
```

attribute names

A, B, C

dimension names

I, J

chunk size

1000

overlap

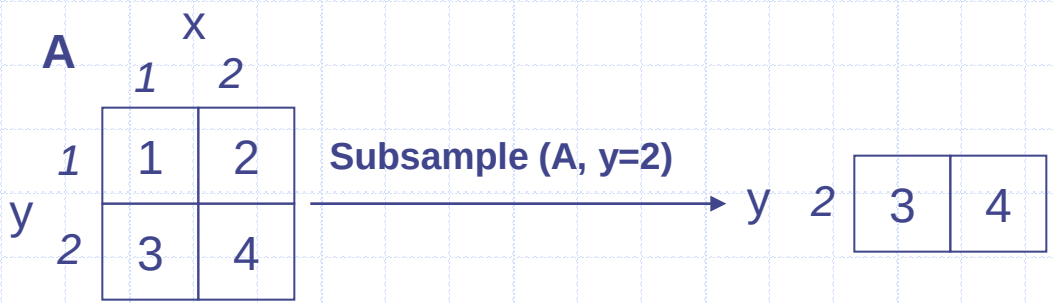
10

Operations on Arrays

- Need to preserve array structure
- Purely structural ops
- Content-based ops
- Linear algebra (if array represents a matrix)

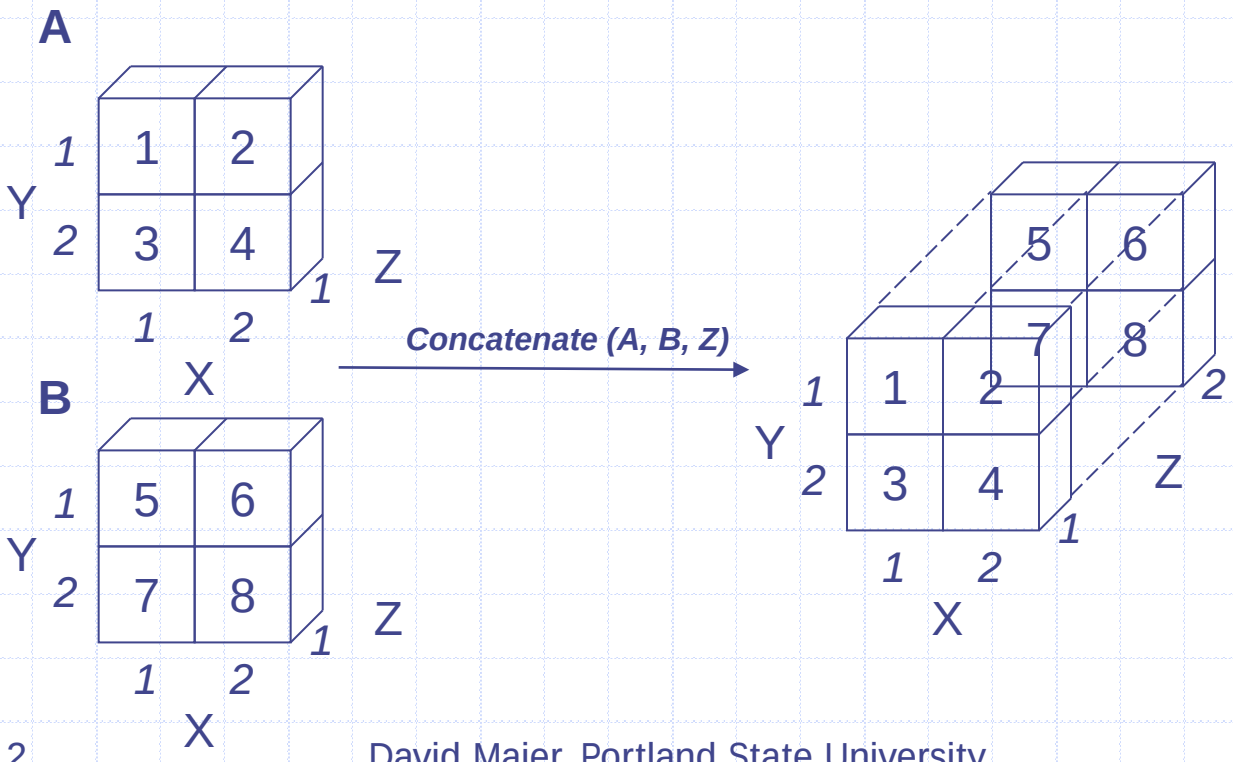
Subsample

Restrict an array by index ranges



Concatenate

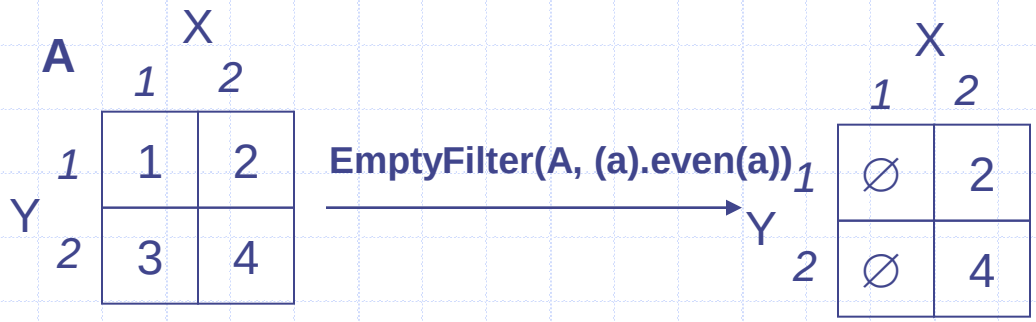
Append arrays along specified dimension



Filter

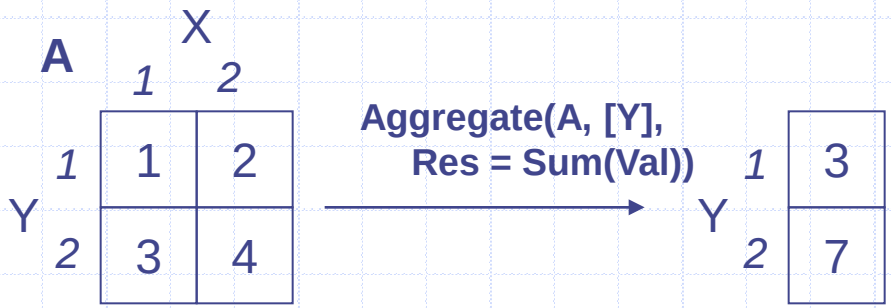
Apply predicate to array elements

Keeps array shape: Inserts empty elements



Aggregate

Reduce across one or more dimensions



Languages for Arrays

Many proposals, old and new

- APL: Falkoff, Iverson
- AML: Marathe, Salem
- NewS, R, Matlab
- rasql: Baumann
- SciQL: Kersten, Zhang, Ivanova, Nes

Array Comprehensions

Like MArray in rasql, Build in SciDB docs

- Supply a spatial domain S

  e.g. [I=0:999, J=0:4999]

- Have an expression g:S → ET (element type)

```
BUILD(S,(i,j) →
      <r=A[i,j+100].va,
    s=B[j].ba*5.0>
           )
```

SciDB: Array Query Language (AQL)

```
SELECT Geo-Mean ( T.B )
FROM Test_Array T
WHERE
     T.I BETWEEN :C1 AND :C2
AND T.J BETWEEN :C3 AND :C4
AND T.A = 10
GROUP BY T.I;
```

User-defined aggregate on an attribute B in array T

Subsample

Filter

Group-by

SciDB: Array Functional Language (AFL)

Lexical syntax for the algebra

```
A<va:int>[I=0:999,J=0:4999]
B<vb:int>[J=0:4999,K=0:2499]
aggregate(
  apply(
    sjoin(A,B,J=J),
    res=A.va*B.vb
        ),
  [I,K],vr=sum(res)
            )
```

Physical Representation

- Array of records → record of arrays

```
Array<va=int, fa=float>[I=0:99, J=0:499] →
<va=Array<int>[I=0:99, J=0:499],
 fa=Array<float>[I=0:99, J=0:499]>
```

- Nested array → merge dimensions

```
Array<va=int, fa=Array<r=float>[K=0:9]>
[I=0:99, J=0:499] →
<va=Array<int>[I=0:99, J=0:499],
 fa=Array<Array<r=float>[K=0:9]>[I=0:99, J=0:499]>
→
<va=Array<int>[I=0:99, J=0:499],
 fa=Array<float>[K=0:9, I=0:99, J=0:499]>
```

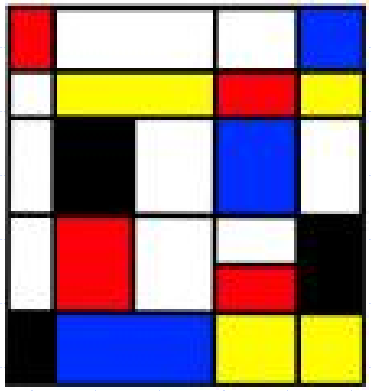
Physical Representation 2

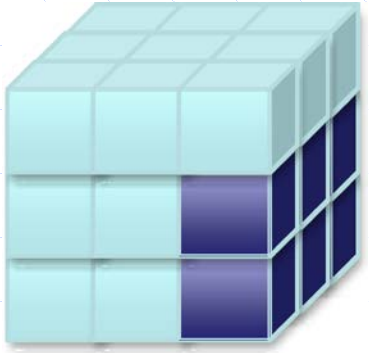
- Non-integer indices → mapping array

```
Array A<a1: int32, a2: double>
 [I(string)=100, J(double)=1000] →
Array BasicA<a1: int32, a2: double>
 [BI=0:99, BJ=0:999]
IMap<I=string>[BI=0:99]
JMap<J=double>[BJ=0:999]

A = Sjoin(BasicA, IMap, JMap,
          A.BI=IMap.BI, A.BJ=JMap.BJ)
```

Partitioning

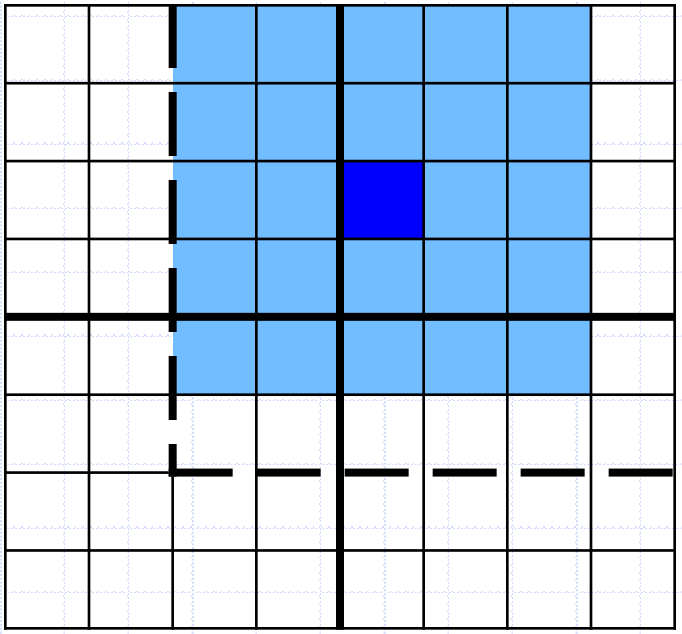
- Rasdaman tiling of rasters
  - Many options, needn't be uniform
  - Can isolate regions of interest
- SciDB chunking
  - Regular divisions along dimensions
  - Distribution pattern, e.g., block cyclic





Issue: Neighborhood Ops

- Doing a 5 x 5 stenciled average over a chunk requires up to 8 adjoining chunks
- Can specify an overlap (e.g., 2 elements)



Issue: Logical vs. Physical Size

- Dividing an array evenly in logical space can give unequal physical chunks after compression
- Equal physical chunks are easier for I/O, but makes it hard to align 2 arrays
- SciDB: Equal-sized logical chunks, but combine multiple physical chunks into an I/O segment

Versions

- Conceptually, updates in SciDB are additions along a History dimension
- Implemented as reverse deltas at a chunk granularity

Application Programming Interface (API)

- Can do embedded queries in general-purpose programming languages, e.g., C++, Python
- Would like a more transparent interface from analysis environments such as R
  - Dynamically accumulate expressions (à la Ohkawa, RIOT)
  - Evaluate intelligently on demand, e.g., minimize data movement

Current R Support for Large Data Not Very Transparent

- Native R

```
result <- sum(array);
```

- Chunked access to netCDF

```
chunk.size <- 1000;
num.chunks <- ceiling(total.size/chunk.size);
for(i in num.chunks) {
  array.part <- get.var.ncdf(file.path,chunk.size);
   result <- result + sum(array.part);
   remove(array.part); gc(); }
```

- Call out to RDBMS

```
result <- sqlQuery(DBconn, "select sum(value)
                            from array_table");
```

- Specialized Libraries

Accumulate Expressions

Want to have as large of scope as possible before evaluating

```
A <- B + C;
…
D <- A[1:10];
…
print(A);
```

Accumulate to

```
print((B + C)[1:10]);
```

Minimize Data Transfer

- Reductive Transforms: less data to move (**bold** = op or arg in SciDB)

```
print((B + C)[1:10]); →
print((B + τ(C))[1:10]); →
print((B[1:10] + τ(C[1:10]));
```

- Consolidating Transforms: fewer transfers

```
print((B + C) + D); →
print((B + τ(C)) + τ(D)); →
print(B+ τ(C + D)); →
```

Additional Aspects

- Needs to be cost based

```
print((B%*%C)%*%D); →
print(B%*%(C%*%D));
B[20,500], C[500,1], D[1,300]
```

- Other considerations
  - Availability of operators in each engine
  - Data representation and distribution
  - Estimate execution time

Thanks to

- SciDB
  - Marilyn Matz, Suchi Raman, Paul Brown, Paradigm4
  - www.scidb.org
- R-SciDB Interface
  - Patrick Leyshock, PSU
- Novartis
  - Proof of concept for SciDB in pharmaceuticals
  - http://www-conf.slac.stanford.edu/xldb2011/talks/xldb2011_wed_1100_Novartis.pdf

NewSQL: Flying on ACID

David Maier

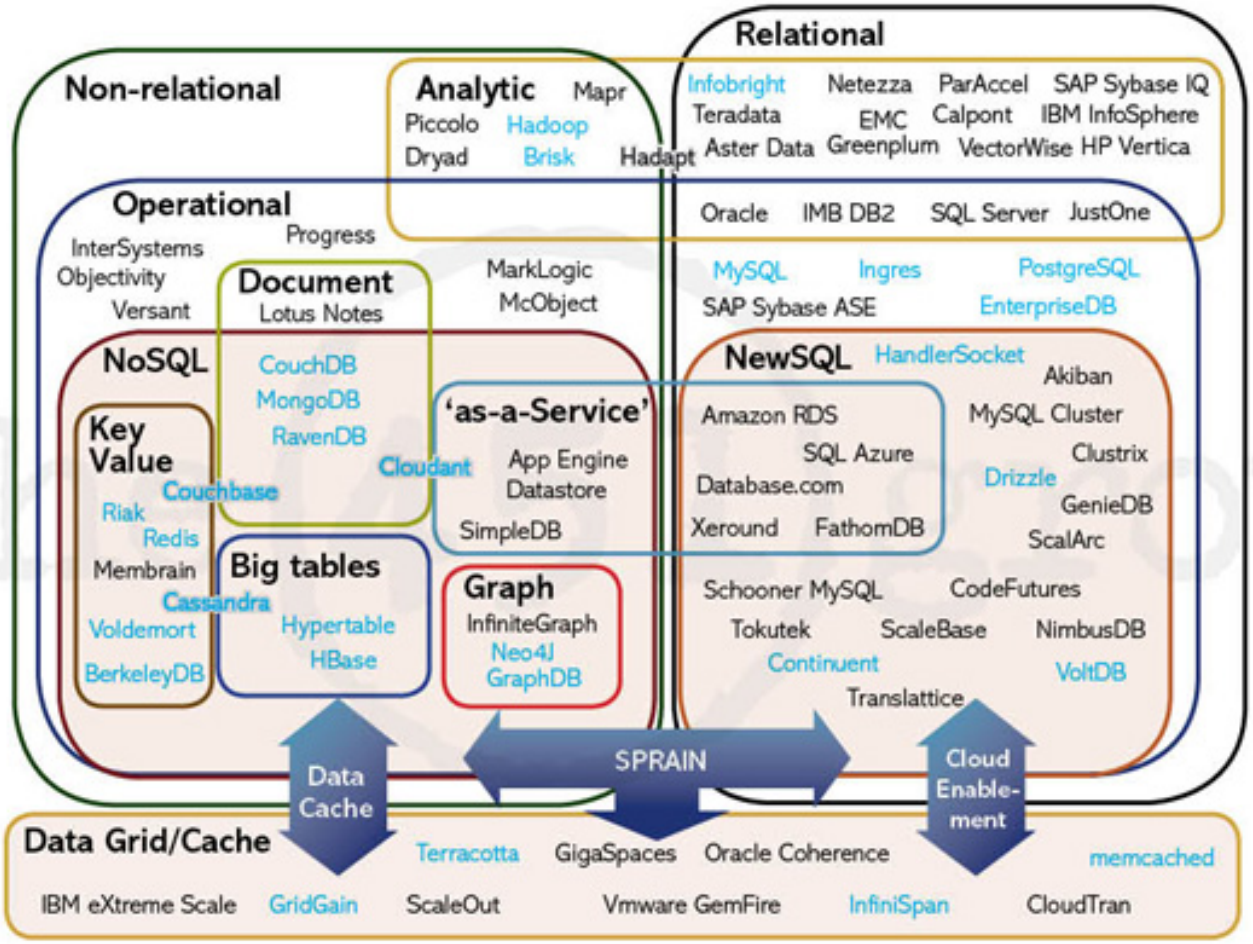
Thanks to H-Store folks, Mike Stonebraker, Fred Holahan

NewSQL

- Keep SQL (some of it) and ACID
- But be speedy and scalable

Database Landscape

From: the 451 group



OLTP Focus

- On-Line Transaction Processing
- Lots of small reads and updates
- Many transactions no longer have a human intermediary

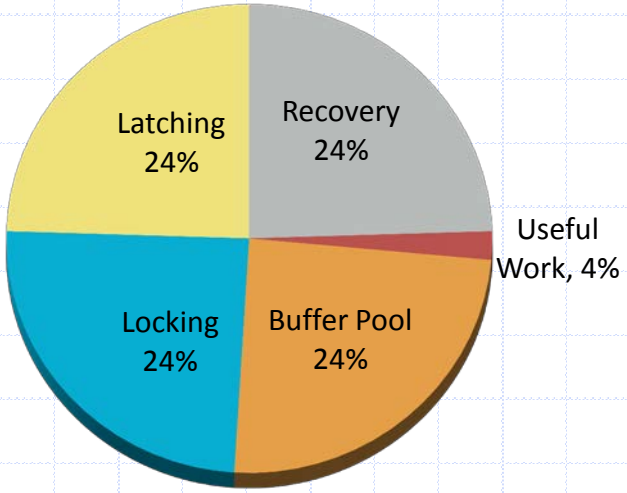
  For example, buying sports or show tickets
- 100K+ xact/sec, maybe millions
- Horses for courses

Premises

- If you want a fast multi-node DBMS, you need a fast single-node DBMS.
- If you want a single-node DBMS to go 100x as fast, you need to execute 1/100 of the instructions.
  - You won't get there on clever disk I/O: Most of the data is living in memory

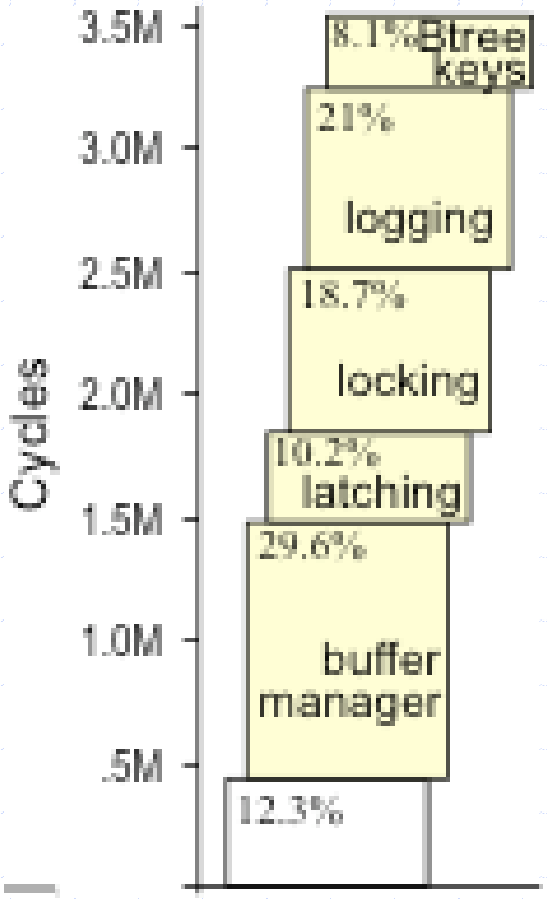
Where Does the Time Go?

- TPC-C CPU cycles
- On Shore DBMS
- Instruction counts have similar pattern



A Bit More Detail

Source: S. Harizopoulos, D. J. Abadi, S. Madden, M. Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008.



What are These Different Parts?

Buffer manager: Manages the slots that holds disk pages

- Locate pages by a hash table
- Employs an eviction strategy (clock scan – approximates LRU)
- Coordinates with recovery system

Different Parts 2

Locks: Logical-level shared and exclusive claims to data items and index nodes

- Locks are typically held until the end of a transaction
- Lock manager must also manage deadlocks

Different Parts 3

Latches: Low-level locks on shared structures

- Free-space list
- Buffer-pool directory (hash table)
- Buffer “clock”

Also, “pinning” pages in the buffer pool

Different Parts 4

Logging: Undo and redo information in case of transaction, application or system failure

- Must be written to disk before corresponding page can be removed from buffer pool

Strategies to Reduce Cost

- All data lives in main memory
- Multi-copy for high-assurance
  - Still need undo info (in memory) for rollback and disk-based information for recovery
- No user interaction in transactions
- Avoid run-time interpretation and planning
  - Register all transactions in advance

Strategies, cont.

- Serialize transactions

  Possible, since there aren't waits for disk I/O or user input

- Parallelize
  - Between transactions
  - Between parts of a single transaction
  - Between primary and secondary copies

H-Store & VoltDB

- H-Store is the academic project

  Brown/Yale/MIT

  http://hstore.cs.brown.edu/

- VoltDB is the company

  Velocity OnLine Transactions

  http://community.voltdb.com/documentation

  Community and Enterprise editions

VoltDB Techniques

Data in main memory

- 32-way cluster can have a terabyte of MM
- Don’t need a buffer manager
- No waiting for disk
- All in-use data generally resides in MM for OLTP systems anyway

VoltDB Techniques 2

Interact only via stored procedures

- No roundtrips to client during multi-query transactions
- No user waiting
- Can compile & optimize in advance
- (Might pre-analyze conflicts)

Need to structure applications carefully

Discussion Problem

Want to support on-line course reg.

1. Search for courses: number, time
2. User gets list of matching courses
3. User chooses a course
4. Show enrollment status of course
5. If not full, allow user to register

   Validate prerequisites

Tables

Offering(CRN, Course#, Days, Limit)

Registered(CRN, SID)

Student(SID, First, Last, Status)

Prereq(Course#, PCourse#, MinMark)

Transcript(SID, Course#, Grade)

- Don’t over-enroll course
- No user input in transaction
- Don’t turn student away if you’ve shown space in the course

VoltDB Techniques 3

Serial execution of transactions

- Avoids locking and latching
- Avoids thread or process switches
- Avoids some logging

  Still need undo buffer for rollback

VoltDB Techniques 4

Multiple copies for high availability

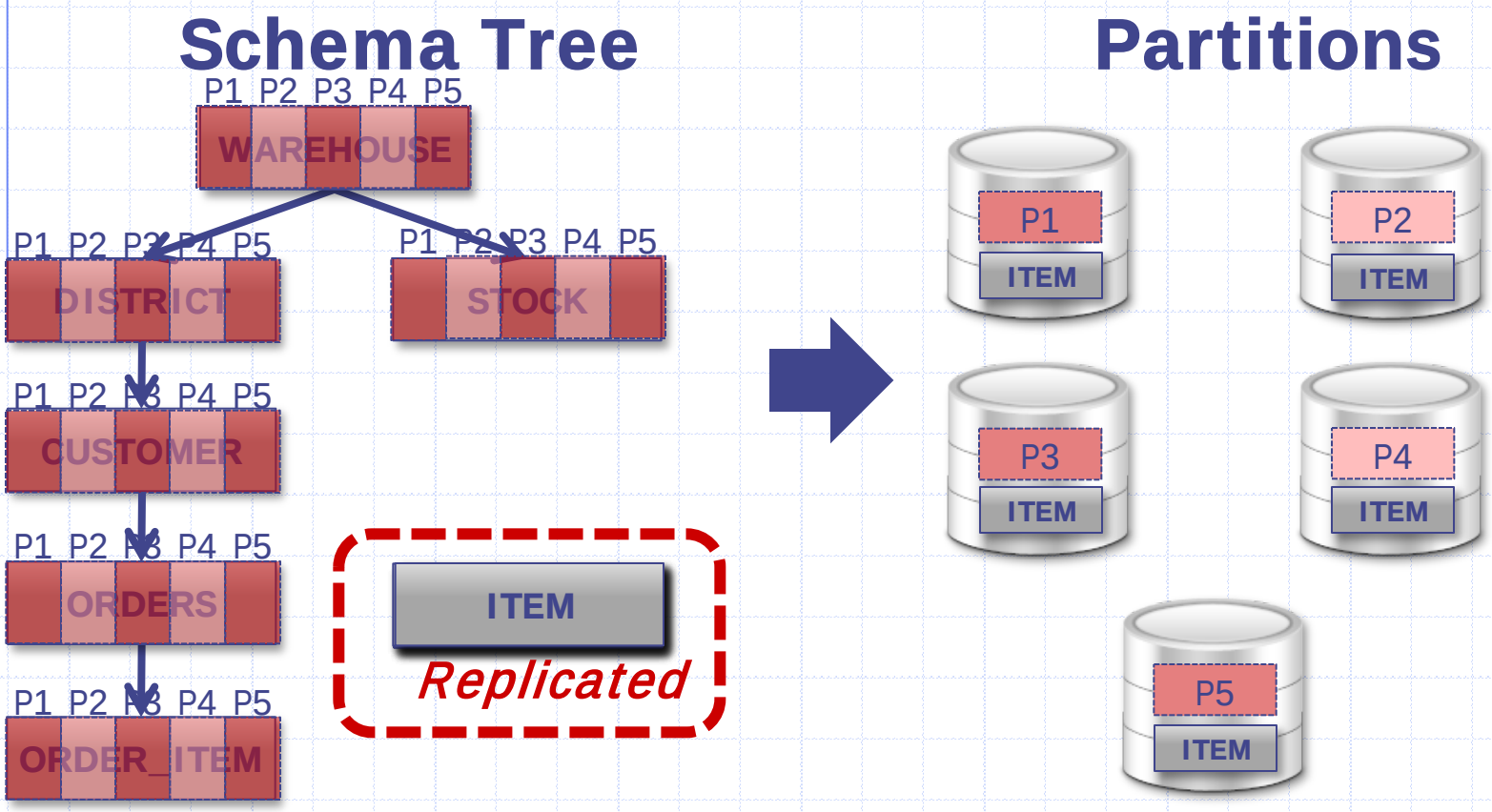
- Can specify k-factor for redundancy: can tolerate up to k node failures
- For complete durability:
  - Snapshot of DB state to disk
  - Log commands to disk

VoltDB Techniques 5

Shared-nothing parallelism: tables can be partitioned (or replicated) and spread across multiple sites.

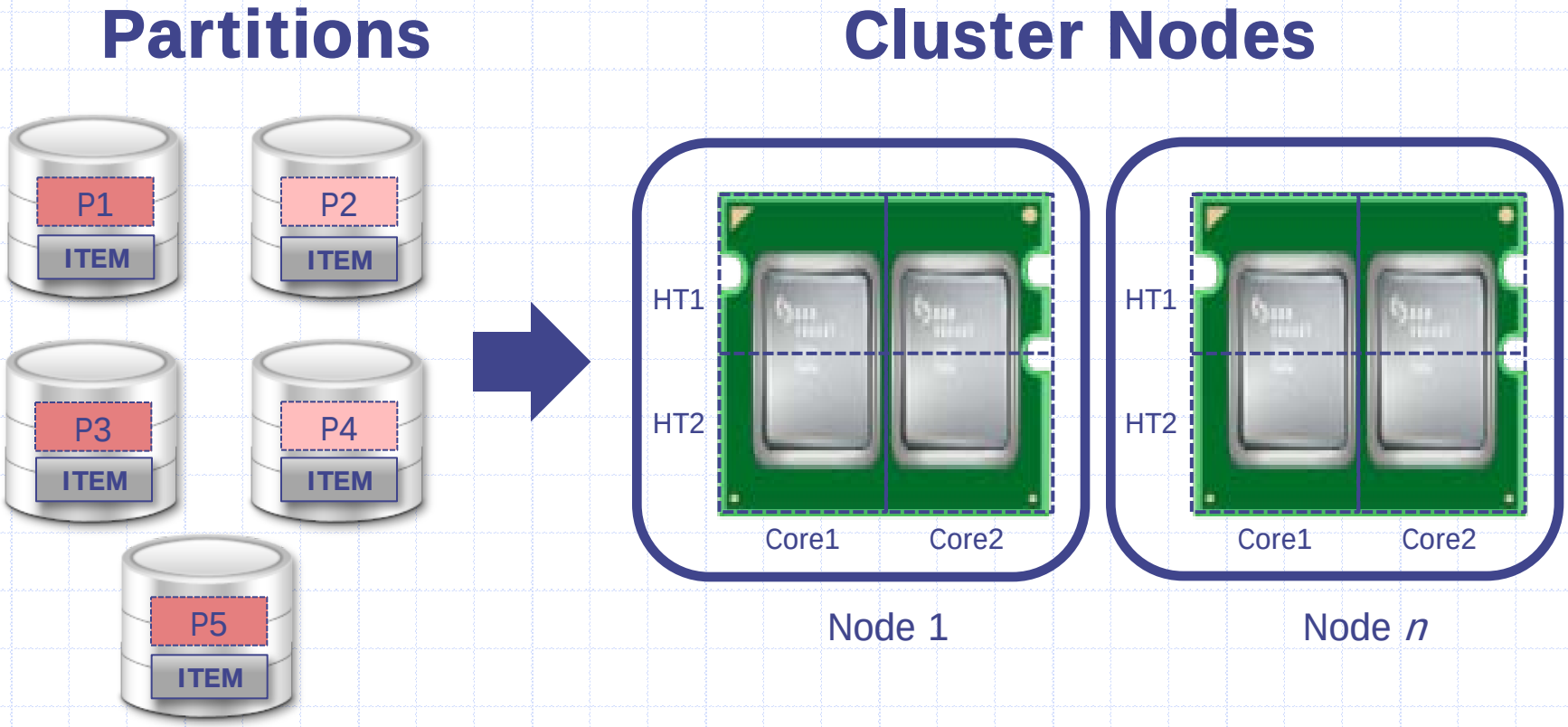
- Each site has its own execution engine and data structures
- No latching of shared structures
- Does incur some latency on multi-partition transactions

Can have partitions of several tables at each site



Data Placement

- Assign partitions to sites on nodes.



Results

- 45X conventional RDBMS
- 7X Cassandra on key-value workload
- Has been scaled to 3.3M (simple) transactions per second

What VoltDB Isn't Doing

- Reducing latency: aim is increased throughput
  - Might take a while to get results back
- All of SQL (e.g., no `NOT` in `WHERE`)
- Big aggregates
- Dynamic DDL
- Ad hoc queries (possible, not fast)

System Structure

- Hosts (nodes) each with several sites (< #cores)
- Each site has data (partitions), indexes, views, stored procedures
- Client can connect to any host
  - Encouraged for load balancing and availability
  - Also, request queue per host

In Operation

1. Client invokes stored procedures with parameters
2. Sent to some host
3. Rerouted to site with correct partition
4. SPs execute serially (need coordinator if more than one partition)
5. Partition forwards queries to redundant copies and waits
6. [Rollback if aborted]
7. Results come back in VoltTable (array)

Setting up a Database

- Schema definition
  - Tables (strings are stored out of line)
  - Indexes, views
- Select partitioning column (or replicate)
  - Can be different for different tables
  - Needn't be a key
  - But may want same column to keep transactions in one partition:
    Use `CRN` for `Offering` and `Registered`

Setting up a Database 2

- Stored procedures
  - In Java and a subset of SQL (some limits)
  - SQL can contain ‘?’ for parameters
  - Must be deterministic (don’t read system clock or do network I/O)
  - Can submit groups of SQL statements
  - Can declare that procedure runs in a single partition (fastest)
    - Multi-partition, multi-round can have waits and network delays

Stored Procedure Example

```
package fadvisor.procedures;
import org.voltdb.*;

@ProcInfo(
    singlePartition = true,
    partitionInfo = "Reservation.FlightID: 0”
)

public class HowManySeats extends VoltProcedure {

   public final SQLStmt GetSeatCount = new SQLStmt(
        "SELECT NumOfSeats, COUNT(ReserveID) " +
        "FROM Flight AS F, Reservation AS R " +
        "WHERE F.FlightID=R.FlightID AND R.FlightID=?;");
```

Stored Procedure Example cont.

```
public long run(int flightid)
        throws VoltAbortException {

        long numofseats;
        long seatsinuse;
        VoltTable[] queryresults;

        voltQueueSQL(GetSeatCount, flightid);
        queryresults = voltExecuteSQL();

        VoltTable result = queryresults[0];
        if (result.getRowCount() < 1) { return -1; }
        numofseats = result.fetchRow(0).getLong(0);
        seatsinuse = result.fetchRow(0).getLong(1);

        numofseats = numofseats - seatsinuse;
        return numofseats; // Return available seats
    }
}
```

Setting Up a Database 3

- Compile stored procedures and client apps
- Set up a Project Definition File
  - Schema
  - Stored Procedures
  - Partitioning
  - Groups & permissions

Project Definition File

```
<?xml version="1.0" ?>
<project>
  <database name="database”>
    <schemas>
      <schema path="flight.ddl" />
    </schemas>
    <procedures>
      <procedure class="procedures.LookupFlight"/>
      <procedure class="procedures.HowManySeats"/>
      <procedure class="procedures.MakeReservation"/>
      <procedure class="procedures.CancelReservation"/>
      <procedure class="procedures.RemoveFlight"/>
    </procedures>
    <partitions>
      <partition table="Reservation" column="FlightID"/>
      <partition table="Customer" column="CustomerID"/>
    </partitions>
  </database>
</project>
```

Starting a Database

- Need a configuration file

```
<?xml version="1.0"?>
<deployment>
   <cluster hostcount=”16”
            sitesperhost=”6”
            kfactor=”2”
   />
</deployment>
```

- Ask a “lead node” to start VoltDB
  - Lead becomes a peer after start up
- Start client apps

From the Client Side

- Connect to DB
- Call stored procedures

```
VoltTable[] results;
try { results = client.callProcedure("LookupFlight",
                                     origin,
                                     dest,
                                departtime).getResults();

} catch (Exception e) {
      e.printStackTrace();
      System.exit(-1);
}
```

- Can also be asynch. with callback

What Can You Change?

- Can add or modify stored procedures while DB is running
  - Need to coordinate change with client apps
- Add columns, tables
  - Need to snapshot DB, stop, restart, restore
- Add nodes, change partitions
  - Same drill

High Availability

- If a site is unavailable, use a redundant copy
- A node can rejoin a cluster, rebuild the partitions it has

  Partition being copied is locked for duration
- Can specify on a cluster split, only the larger group keeps running

Snapshots

Can make a consistent copy of snapshot to disk

- Manual or on a schedule
- Each node stores a file locally
- Transaction consistent: will maintain multiple versions of data temporarily
- Can restore with changes
  - New column
  - Different partitioning

Command Logging

Can log commands to disk, then play back from last snapshot

- Don’t need to log SELECTs
- Can be synchronous, will delay client responses
- Snapshot + synchronous command logging shouldn’t lose anything

Views

- Views are materialized
- Must have group-by and return all grouping columns
- Aggregates are COUNT and SUM (??)

Export

- VoltDB can be the front end to a warehouse or map-reduce engine
- Export-only tables
  - Can only insert into them (but will undo)
  - Contents are spooled to a Connector
  - Export client polls the Connector
  - Export data overflows to disk
- Have an export client that uses Sqoop to populate HDFS

Languages

- C#
- C++
- Erlang
- Java
- JDBC
- JSON (HTTP from PHP, Python, Perl, C#)
- PHP
- Python
- Ruby

Minimal Configuration

- OS: 64-bit Linux
- Dual-core, 64-bit proc. (4-8 cores better)
- 4 Gbytes memory minimum
- Sun Java SDK 6
- Network Time Protocol (NTP)

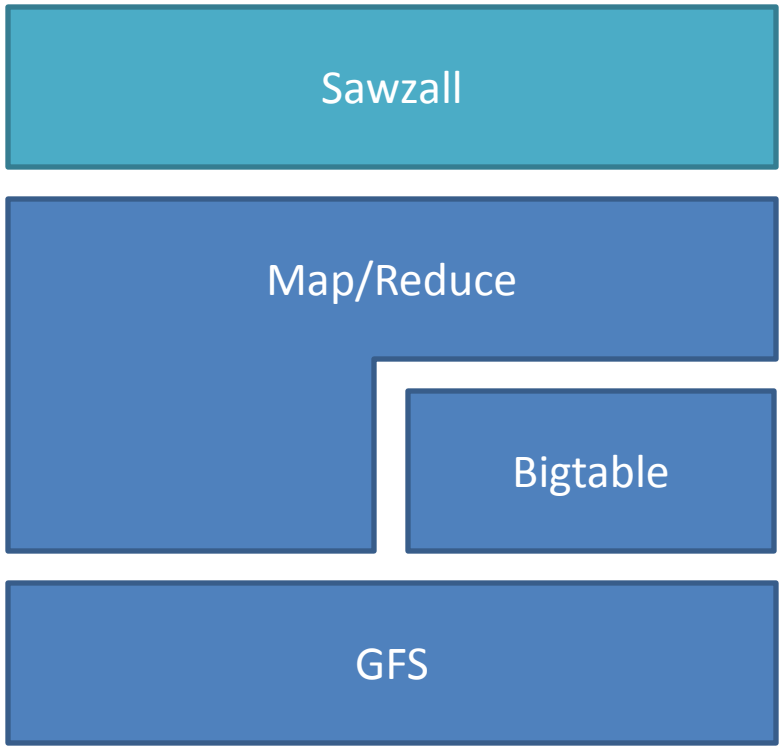
Ongoing Work

- VoltDB uses 2-phase commit on multi-partition procedures
- Considering speculative execution of transactions at sites waiting for commit/abort
- Would require multi-transaction rollback

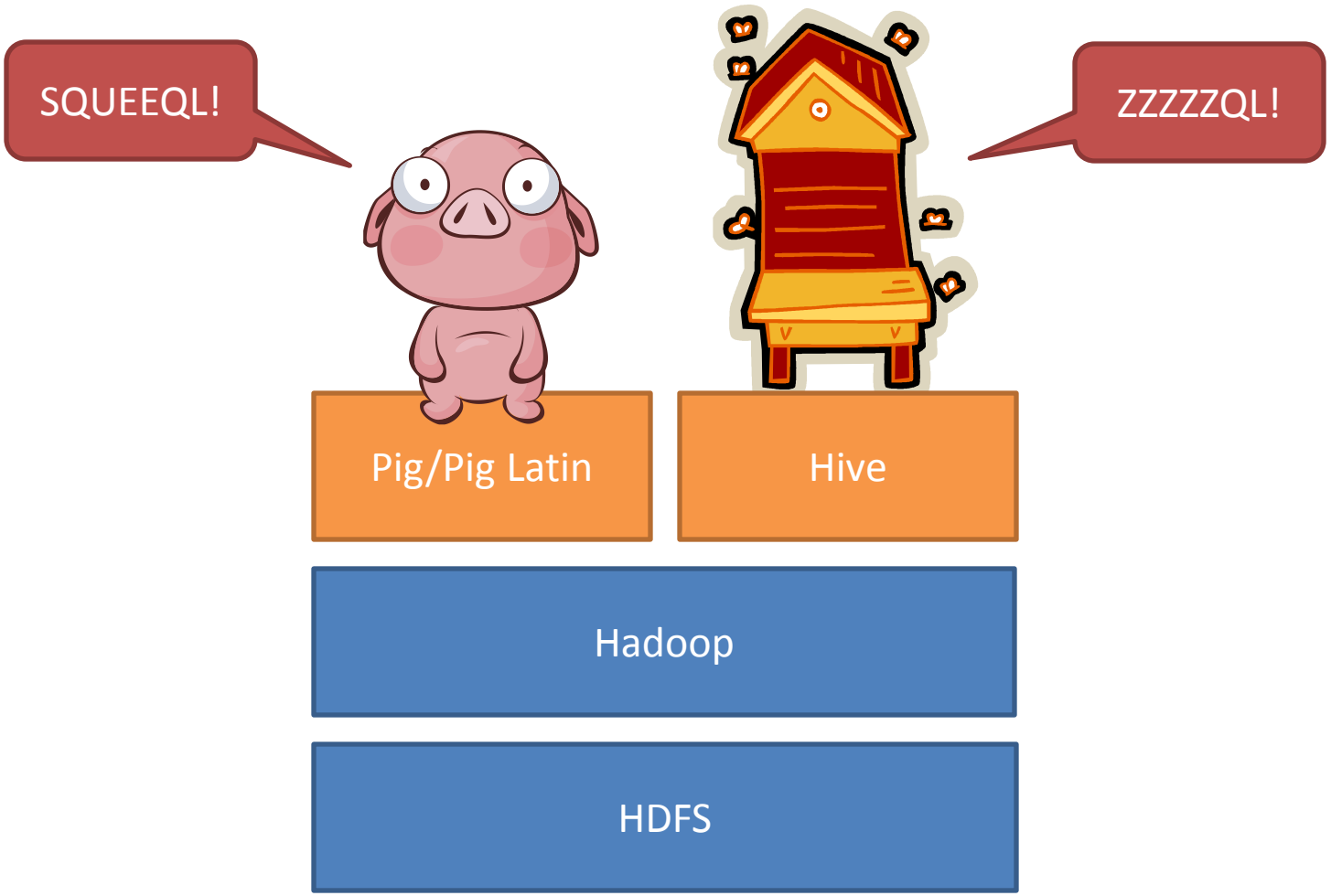
Data Management in the Cloud

PIG LATIN AND HIVE

The Google Stack



The Hadoop Stack



Motivation for Pig Latin

- Disadvantages of parallel database products
  - prohibitively expensive at Web scale
  - programmers like to write scripts to analyze data
  - SQL is “unnatural” and overly restrictive in this context
- Limitations of Map/Reduce
  - one-input two-stage data flow is extremely rigid
  - custom code has to be written for common operations such as projection and filtering
  - opaque nature of map and reduce function impedes ability of system to perform optimization
- Pig Latin combines “best of both worlds”
  - high-level declarative querying in SQL
  - low-level procedural programming of Map/Reduce

A First Example

- Find pages in sufficiently large categories with a high page rank
- SQL

```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 10^6
```

- 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(good_urls.pagerank);
```

Pig Latin Programs

- Embody “best of both worlds” approach
- Sequence of steps
  - similar to imperative language
  - each step carries out a single data transformation
  - appealing to many developers
- High-level transformations
  - similar to SQL
  - high-level operations render low-level manipulations unnecessary
  - potential for optimization
- Similar to specifying a query execution plan
  - “automatic query optimization has its limits, especially with uncatalogued data, prevalent user-defined functions, and parallel execution”

Pig Latin Features

- "Unconventional features that are important for [..] casual ad-hoc data analysis by programmers"
  - flexible, fully nested data model
  - extensive support for user-defined functions
  - ability to operate over plain input files without any schema
  - debugging environment to deal with enormous data sets
- Pig Latin programs are executed using Pig
  - compiled into (ensembles of) map-reduce jobs
  - executed using Hadoop
- Pig is an open-source project in the Apache incubator

Dataflow Language

- "While the SQL approach is good for non-programmers and/or small data sets, experienced programmers who must manipulate large data sets [..] prefer the Pig Latin approach."
  - "I much prefer writing in Pig [Latin] versus SQL. The step-by-step method of creating a program in Pig [Latin] is much cleaner and simpler to use than the single block method of SQL. It is easier to keep track of what your variables are, and where you are in the process of analyzing your data." – Jasmine Novak, Engineer, Yahoo!

Optimizations

- Pig Latin programs supply explicit sequence of operations, but are not necessarily executed in that order
- High-level relational-algebra-style operations enable traditional database optimization
- Example

```
spam_urls = FILTER urls BY isSpam(url);
culprit_urls = FILTER spam_urls BY pagerank > 0.8;
```

  - if **`isSpam`** is an expensive function and the **`FILTER`** condition is selective, it is more efficient to execute the second statement first

Optional Schemas

- Traditional database systems require importing data into system-managed tables
  - transactional consistency guarantees
  - efficient point lookups (physical tuple identifiers)
  - curate data on behalf of the user: schema enables other users to make sense of the data
- Pig only supports read-only data analysis of data sets that are often temporary
  - stored schemas are strictly optional
  - no need for time-consuming data import
  - user-provided function converts input into tuples (and vice-versa)

Nested Data Model

- Motivation
  - programmers often think in nested data models
    `term = Map<documentId, Set<positions>>`
  - in a traditional database, data must be normalized into flat table
    `term(termId, termString, …)`
    `term_position(termId, documentId, position)`
- Pig Latin has a flexible, fully nested data model
  - closer to how programmers think
  - data is often already stored in nested fashion in source files on disk
  - expressing processing tasks as sequences of steps where each step performs a single transformation requires a nested data model, e.g. **GROUP** returns a non-atomic result
  - user-defined functions are more easily written

User-Defined Functions

- Pig Latin has extensive support for user-defined functions (UDF) for custom processing
  - analysis of search logs
  - crawl data
  - click streams
  - …
- Input and output of UDF follow flexible, nested data model
  - non-atomic input and output
  - only one type of UDF that can be used in all construct
- UDFs are implemented in Java

Parallelism

- Pig Latin is geared towards Web-scale data
  - requires parallelism
  - does not make sense to consider non-parallel evaluation
- Pig Latin includes a small set of carefully chosen primitives that can easily be parallelized
  - "language primitives that do not lend themselves to efficient parallel evaluation have been deliberately excluded"
  - no non-equi-joins
  - no correlated sub-queries
- Backdoor
  - UDFs can be written to carry out tasks that require this functionality
  - this approach makes user aware of how efficient their programs will be and whether they will be parallelized

Data Model

- **Atom**: contains a simple atomic value
  - string, number, …
- **Tuple**: sequence of fields
  - each field can be any of the data types
- **Bag**: collection of tuple with possible duplicates
  - schema of constituent tuples is flexible
- **Map**: collection of data items, where each data item has a key
  - data items can be looked up by key
  - schema of constituent tuples is flexible
  - useful to model data sets where schemas change over time, i.e. attribute names are modeled as keys and attribute values as values

Data Model

$$t = \left(\text{'alice'}, \left\{ \begin{array}{c} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20]\right)$$

Let fields of tuple t be called f1, f2, f3

Expression Type	Example	Value for t
Constant	'bob'	Independent of t
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{('lakers'), ('iPod')}
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Data Loading

- First step in a Pig Latin program
  - what are the data files?
  - how are file contents deserialized, i.e. converted into Pig's data model
  - data files are assumed to contain a bag of tuples
- Example

```
queries = LOAD 'query_log.txt'
          USING myLoad()
          AS (userId, queryString, timestamp);
```

  - **query_log.txt** is the input file
  - file contents are converted into tuples using the custom **myLoad** deserializer
  - the tuples have three attributes named **userId, queryString, timestamp**

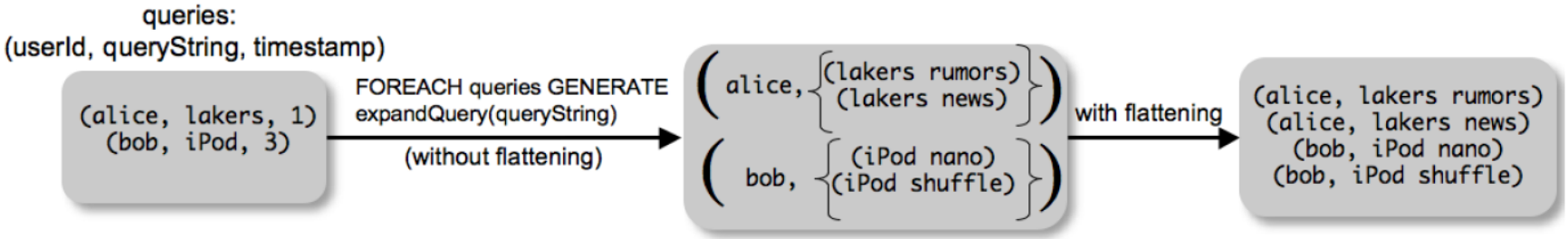
Per-Tuple Processing

- Command **FOREACH** applies some processing to every tuple of the data sets
- Example

```
expanded_queries = FOREACH queries GENERATE
                   userId, expandQuery(queryString);
```

  - every tuple in the **queries** bag is processed independently
  - attribute **userId** is projected
  - UDF **expandQuery** is applied to the **queryString** attribute
- Since there can be no dependence between the processing of different tuples, **FOREACH** can be easily parallelized

Per-Tuple Processing



- **GENERATE** clause is followed by a list of expressions as supported by Pig Latin's data model
- For example, **FLATTEN** can be used to unnest data

```
expanded_queries = FOREACH queries GENERATE
userId, FLATTEN(expandQuery(queryString));
```

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Selection

- Tuples are selected using the FILTER command
- Example

```
real_queries = FILTER queries BY userId neq 'bot';
```

- Filtering conditions involve a combination of expressions
  - **equality**: **==** (numeric), **eq** (strings)
  - **inequality**: **!=** (numeric), **neq** (strings)
  - **logical connectors**: **AND**, **OR**, and **NOT**
  - **user-defined functions**
- Example

```
real_queries =
                FILTER queries BY NOT isBot(userId);
```

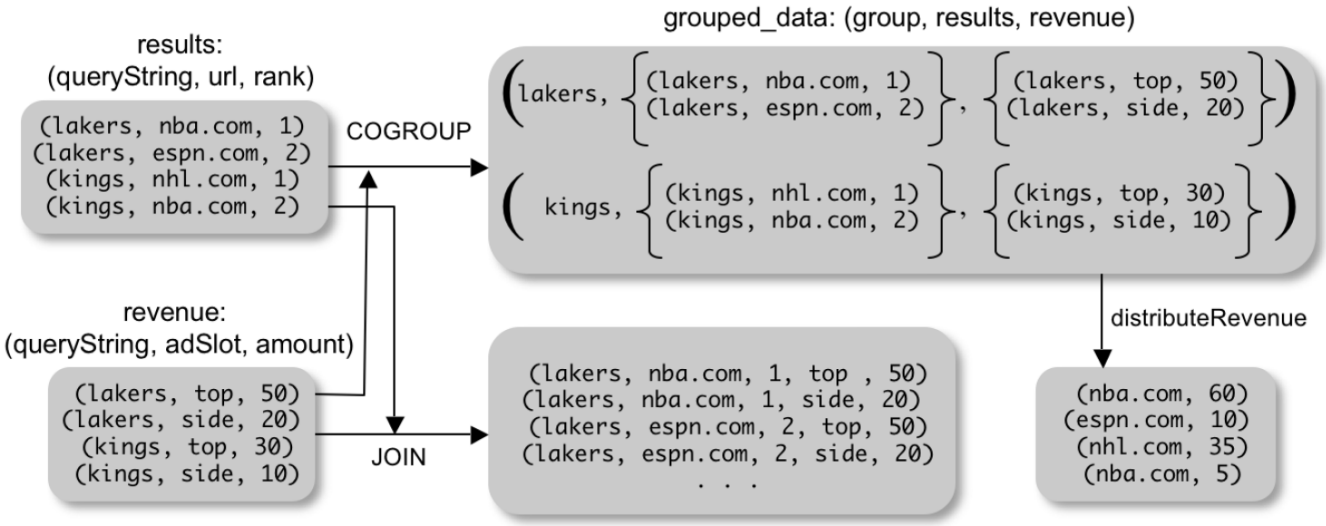
Grouping

- Command **COGROUP** groups tuples from one or more data sets
- Example

```
grouped_data = COGROUP results BY queryString,
                       revenue BY queryString;
```

  - assume **(queryString, url, rank)** for **results**
  - assume **(queryString, adSlot, amount)** for **revenue**
  - **gouped_data** will be **(queryString, results, revenue)**
- Difference to **JOIN**
  - **JOIN** is equivalent to **COGROUP** followed by taking the cross-product of the tuples in the nested bags
  - **COGROUP** gives access to “intermediate result” (example on next slide)
- Nested data model enables **COGROUP** as independent operation

Grouping versus Joining



- Example

```
url_revenues = FOREACH grouped_data GENERATE
FLATTEN(distributeRevenue(results, revenue));
```

  - **distributeRevenue** attributes revenue from top slot entirely to first result, while revenue from side slot is attributed to all results
  - Since this processing task is difficult to express in SQL, **COGROUP** is a key difference between Pig Latin and SQL

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Syntactic Sugar

- Special case of **COGROUP** is when only one data set is involved
  - can use more intuitive keyword **GROUP**
  - similar to typical group-by/aggregate queries
- Example

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue GENERATE
                 queryString,
                 SUM(revenue.amount) AS totalRevenue;
```

  - **revenue.amount** refers to a projection of the nested bag in the tuples of **grouped_revenue**

More Syntactic Sugar

- Pig Latin provides a **JOIN** key word for equi-joins
- Example

```
join_result = JOIN results BY queryString,
                   revenue BY queryString;
```

is equivalent to

```
temp_var    = COGROUP results BY queryString,
              revenue BY queryString;
join_result = FOREACH temp_var GENERATE
              FLATTEN(results), FLATTEN(revenue);
```

We Gotta Have Map/Reduce!

- Based on **FOREACH**, **GROUP**, and UDFs, map-reduce programs can be expressed
- Example

```
map_result = FOREACH input
             GENERATE FLATTEN(map(*));
key_groups = GROUP map_result BY $0;
output = FOREACH key_groups GENERATE reduce(*);
```

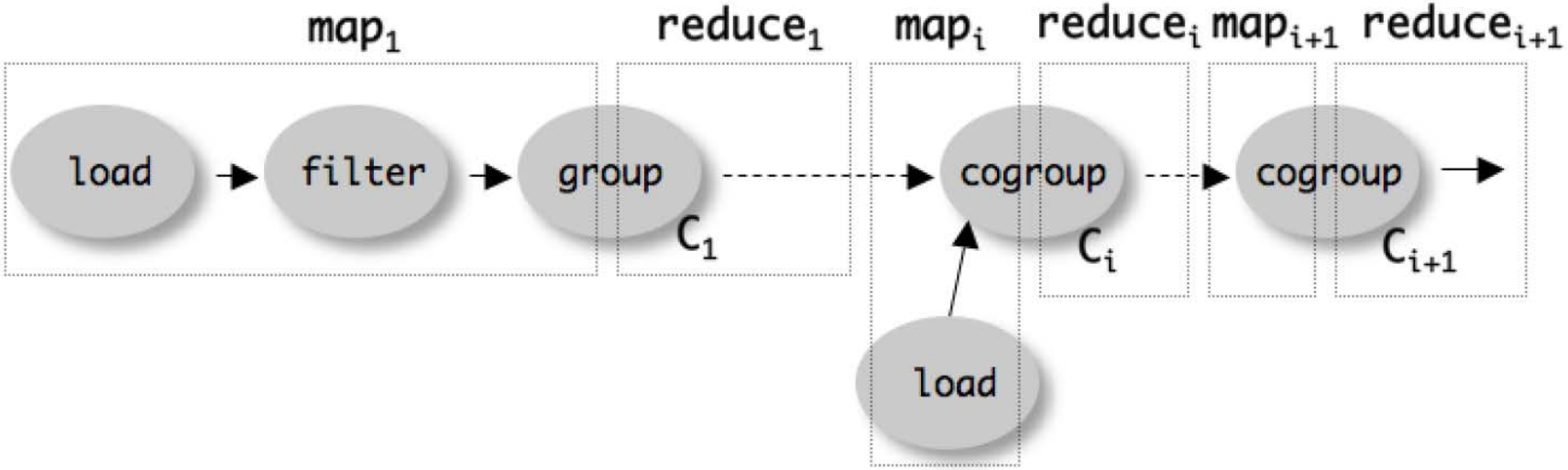
More Pig Latin Commands

- Pig Latin commands that are similar to SQL counterparts
  - UNION: returns the union of two or more bags
  - CROSS: returns the cross-product of two or more bags
  - ORDER: orders a bag by the specified fields
  - DISTINCT: eliminates duplicate tuples in the bag (syntactic sugar for grouping the bag by all fields and projecting out the groups)
- Nested operations
  - process nested bags within tuples
  - **`FILTER`**, **`ORDER`**, and **`DISTINCT`** can be nested within **`FOREACH`**
- Output
  - command **`STORE`** materializes results to a file
  - as in **`LOAD`**, default serializer can be replaced in the **`USING`** clause

Implementation

- Pig is the execution platform of Pig Latin
  - different systems can be plugged in as data processing backend
  - currently implemented using Hadoop
- Lazy execution
  - processing is only triggered when **STORE** command is invoked
  - enables in-memory pipelining and filter reordering across multiple Pig Latin commands
- Logical query plan builder
  - checks that input files and bags being referred to are valid
  - builds a plan for every bag the user defines
  - is independent of data processing backend
- Physical query plan compiler
  - compiles a Pig Latin program into map-reduce jobs (see next slide)

Mapping Pig Latin to Map/Reduce



- Each (**CO**)**GROUP** command is converted into a separate map-reduce job, i.e. a dedicated map and reduce function
- Commands between (**CO**)**GROUP** commands are appended to the preceding reduce function
- For (**CO**)**GROUP** commands with more than one data set, the map function adds an extra attribute to identify the data set

Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

More Nuts and Bolts

- Two map-reduce jobs are required for the **ORDER** command
  - first job samples input to determine quantiles of sort key
  - map of second job range partitions input according to quantiles
  - reduce of second job performs the sort
- Parallelism
  - **LOAD**: parallelism due to data residing in HDFS
  - **FILTER** and **FOREACH**: automatic parallelism due to Hadoop
  - (**CO**)**GROUP**: output from multiple map instances is repartitioned in parallel to multiple reduce instances

Hadoop as a Data Processing Backend

- Pros: Hadoop comes with free
  - parallelism
  - load-balancing
  - fault-tolerance
- Cons: Map-reduce model introduces overheads
  - data needs to be materialized and replicated between successive jobs
  - additional attributes need to be inserted to identify multiple data sets
- Conclusion
  - overhead is often acceptable, given the Pig Latin productivity gains
  - Pig does not preclude use of an alternative data processing backend

Debugging Environment

- Using an iterative development and debugging cycle is not efficient in the context of long-running data processing tasks
- Pig Pen
  - interactive Pig Latin development
  - sandbox data set visualizes result of each step
- Sandbox data set
  - must meet objectives of realism, conciseness, and completeness
  - generated by random sampling, synthesizing “missing” data, and pruning redundant tuples

Debugging Environment

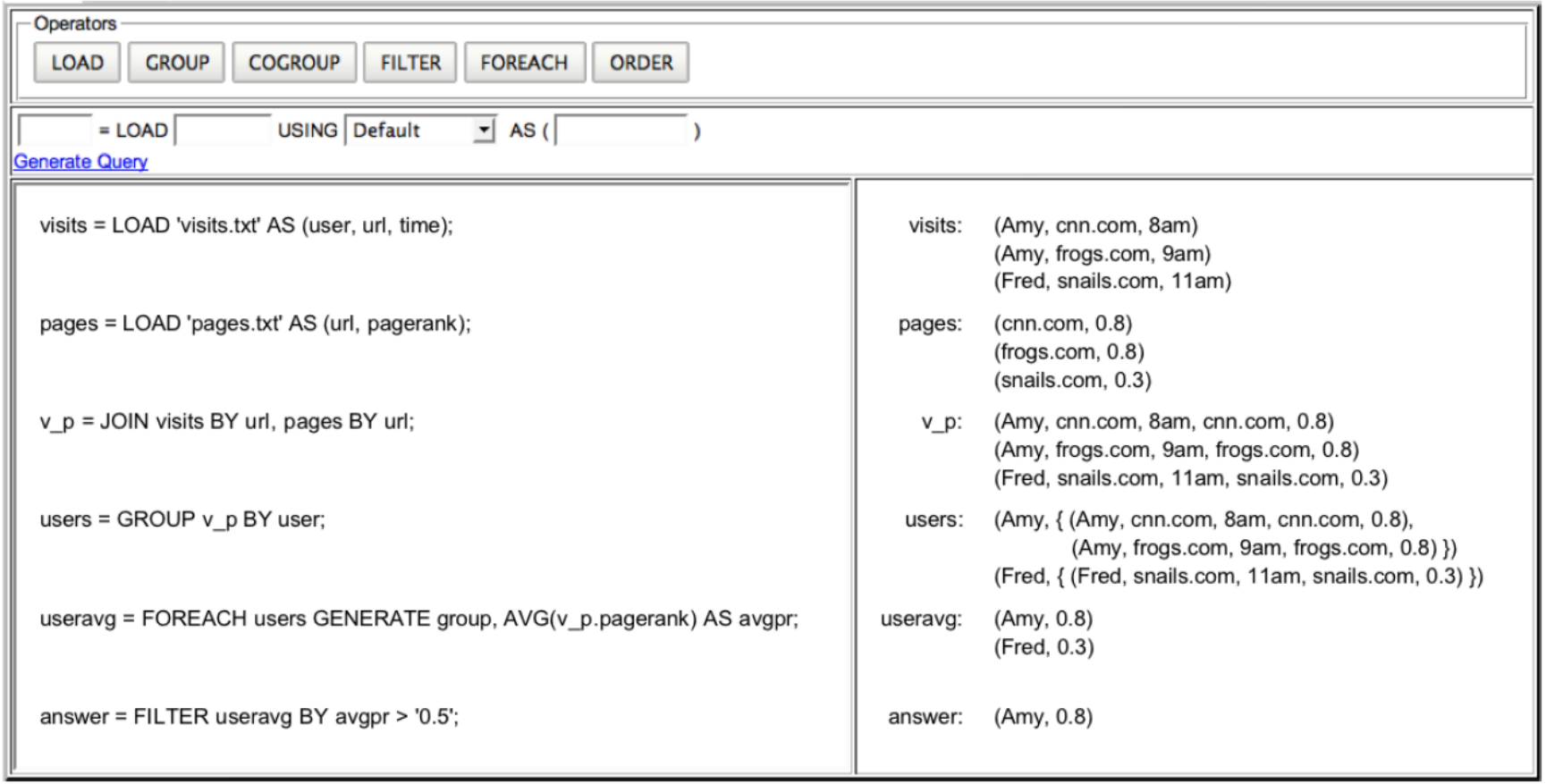


Figure Credit: "Pig Latin: A Not-So-Foreign Language for Data Processing" by C. Olston et al., 2008

Use Cases at Yahoo!

- Rollup aggregates
  - frequency of search terms aggregated over days, week, or months, and also geographical location
  - number of searches per user and average number of searches per user
  - **Pig Point**: data is too big and transient to justify curation in database
- Temporal analysis
  - how do search query distributions change over time?
  - **Pig Point**: good use case for the **COGROUP** command
- Session analysis
  - how long is the average user session?
  - how many links does a user click before leaving a page?
  - how do click patterns vary over time?
  - **Pig Point**: sessions are easily expressed in the nested data model

Motivation for Hive

- Growth of the Facebook data warehouse
  - 2007: 15TB of net data
  - 2010: 700TB of net data
- Original Facebook data processing infrastructure
  - built using a commercial RDBMS prior to 2008
  - became inadequate as daily data processing jobs took longer than a day
- Hadoop was selected as a replacement
  - pros: petabyte scale and use of commodity hardware
  - cons: using it was not easy for end user not familiar with map-reduce
  - "Hadoop lacked the expressiveness of [..] query languages like SQL and users ended up spending hours (if not days) to write programs for even simple analysis."
- Hive is intended to address this problem by bridging the gap

Data Model

- Unlike Pig Latin, schemas are **not** optional in Hive
- Hive structures data into well-understood database concepts like tables, columns, rows, and partitions
- Primitive types
  - **Integers**: bigint (8 bytes), int (4 bytes), smallint (2 bytes), tinyint (1 byte)
  - **Floating point**: float (single precision), double (double precision)
  - **String**
- Complex types
  - **Associative arrays**: map<key-type, value-type>
  - **Lists**: list<element-type>
  - **Structs**: struct<field-name: field-type, …>
- Complex types are templated and can be composed to create types of arbitrary complexity

Creating Tables

- Example

```
CREATE TABLE t1(
    st string,
    fl float,
    li list<map<string, struct<p1:int, p2:int>>
);
```

  - Query expressions can access fields using the dot operator
  - **`t1.li[0].key`** gives the struct associated with **`key`** of the first element of the list **`li`**

- Tables are serialized and deserialized using serializers and deserializers provided by Hive

Creating Tables

- Legacy data or data from other applications is supported through custom serializers and deserializers
  - SerDe framework
  - **`ObjectInspector`** interface
- Example

```
ADD JAR /jars/myformat.jar
CREATE TABLE t2
ROW FORMAT SERDE 'com.myformat.MySerDe';
```

Query Language

- HiveQL is a subset of SQL plus some extensions
  - from clause sub-queries
  - various types of joins: inner, left outer, right outer and outer joins
  - Cartesian products
  - group by and aggregation
  - union all
  - create table as select
  - useful functions on primitive and complex types
- Limitations
  - only equality joins
  - joins need to be written using ANSI join syntax
  - not support for inserts in existing table or data partition
  - all inserts overwrite existing data

Inserting Data

- Example

```
INSERT OVERWRITE TABLE t2
SELECT t3.c2, COUNT(1)
FROM t3
WHERE t3.c1 <= 20
GROUP BY t3.c2;
```

  - **OVERWRITE** (instead of **INTO**) keyword to make semantics of insert statement explicit
- Lack of **INSERT INTO**, **UPDATE**, and **DELETE** enable simple mechanisms to deal with reader and writer concurrency
- At Facebook, these restrictions have not been a problem
  - data is loaded into warehouse daily or hourly
  - each batch is loaded into a new partition of the table that corresponds to that day or hour

Inserting Data

- Hive also supports inserting data into HDFS, local directories, or directly into partitions (more on that later)
- Inserting into HDFS

```
INSERT OVERWRITE DIRECTORY '/output_dir'
SELECT t3.c2, AVG(t3.c1)
FROM t3
WHERE t3.c1 > 20 AND t3.c1 <= 30
GROUP BY t3.c2;
```

- Inserting into local directory

```
INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'
SELECT t3.c2, SUM(t3.c1)
FROM t3
WHERE t3.c1 > 30
GROUP BY t3.c2;
```

We Gotta Have Map/Reduce!

- HiveQL has extensions to express map-reduce programs
- Example

```
FROM (
  MAP doctext USING 'python wc_mapper.py'
              AS (word, cnt)
  FROM docs CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

- Distribution criteria between mappers and reducers can be fine tuned using **DISTRIBUTE BY** and **SORT BY**
- interchangeable order of **FROM**, **SELECT**, **MAP**, and **REDUCE**
- **MAP** can be used without **REDUCE** clause

Data Storage

- Table metadata associates data in a table to HDFS directories
  - **tables**: represented by a top-level directory in HDFS
  - **table partitions**: stored as a sub-directory of the table directory
  - **buckets**: stores the actual data and resides in the sub-directory that corresponds to the bucket's partition, or in the top-level directory if there are no partitions
- Partitioned table are created using the **`PARTITIONED BY`** clause in the **`CREATE TABLE`** statement

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int);
```

- New partitions can be created through an **`INSERT`** statement or an **`ALTER`** statement that adds a partition to a table

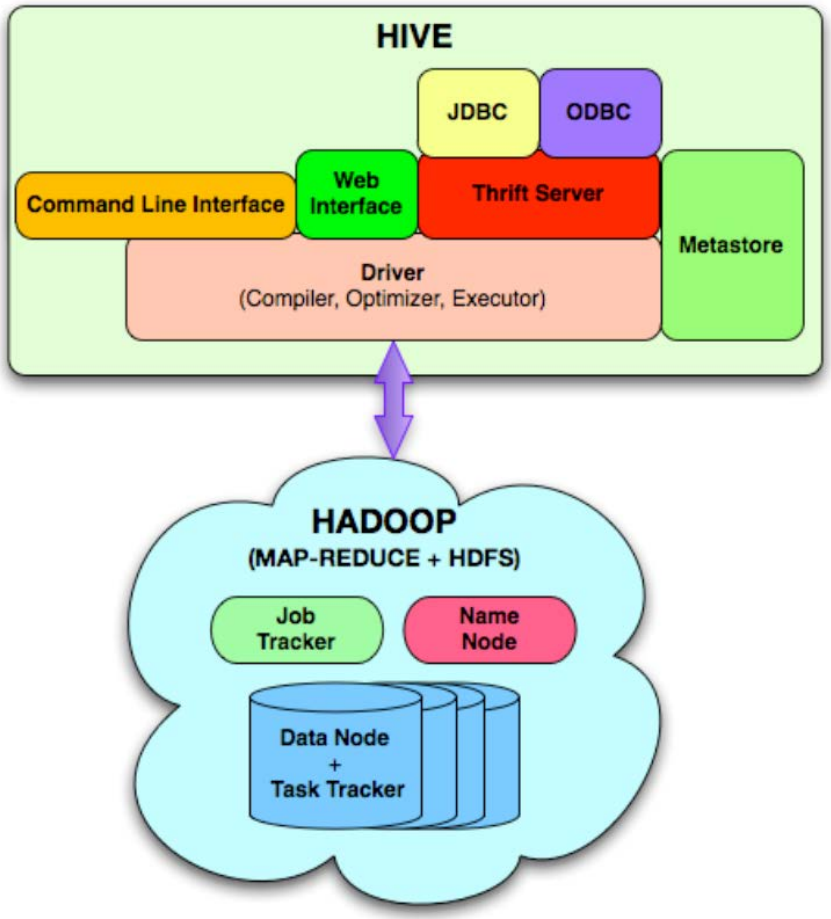
Partition Example

```
INSERT OVERWRITE TABLE test_part
PARTITION(ds='2009-01-01', hr=12)
SELECT * FROM t;

ALTER TABLE test_part
ADD PARTITION(ds='2009-02-02', hr=11);
```

- Each of these statements creates a new directory
  - /…/test_part/ds=2009-01-01/hr=12
  - /…/test_part/ds=2009-02-02/hr=11
- Note that partitioning columns are not part of the table data
- HiveQL compiler uses this information to prune directories that need to be scanned to evaluate a query

```
SELECT * FROM test_part WHERE ds='2009-01-01';

SELECT * FROM test_part
WHERE ds='2009-02-02' AND hr=11;
```

Hive Architecture



- **Driver** manages lifecycle of HiveQL statement as it moves through Hive
- **HiveServer** provides Thrift and JDBC/ODBC interfaces
- **Clients** use command line interface, Web UI, or JDBC/ODBC driver
- **Extensibility interfaces** include SerDe, user defined functions (UDF), user defined aggregate functions (UDAF)

Figure Credit: "Hive – A Petabyte Scale Data Warehouse Using Hadoop" by A. Thusoo et al., 2010

Hive Architecture

- Metastore
  - system catalog and metadata about tables, columns, partitions, etc.
  - uses a traditional RDBMS “as this information needs to be served fast”
  - backed up regularly
  - needs to be able to scale with the number of submitted queries
  - only plan compiler talks to Metastore
- Query Compiler
  1. parsing HiveQL using Antlr to generate an abstract syntax tree
  2. type checking and semantic analysis based on Metastore information
  3. naïve rule-based optimization: column pruning, predicate pushdown, partition pruning, mapping of “side joins”, and join reordering
  4. generation of the physical plan by splitting it into multiple map-reduce and HDFS tasks… magic!

Hive Usage at Facebook as of 2010

- Data stored
  - 700TB of data in warehouse (2.1PB with three-way replication)
  - 5TB of compressed data added daily (15TB after replication)
  - typical compression ratio is 1:7 and more
- Data processed
  - 75TB of data processed each day
  - 7500 jobs submitted to cluster per day
- Data processing task
  - more than 50% of the workload are ad-hoc queries
  - remaining workload produces data for reporting dashboards
  - range from simple summarization tasks to generate rollups and cubes to more advanced machine learning algorithms
- Hive is used by novice and expert users

Hive and Pig Latin

Feature	Hive	Pig
Language	SQL-like	PigLatin
Schemas/Types	Yes (explicit)	Yes (implicit)
Partitions	Yes	No
Server	Optional (Thrift)	No
User Defined Functions (UDF)	Yes (Java)	Yes (Java)
Custom Serializer/Deserializer	Yes	Yes
DFS Direct Access	Yes (implicit)	Yes (explicit)
Join/Order/Sort	Yes	Yes
Shell	Yes	Yes
Streaming	Yes	Yes
Web Interface	Yes	No
JDBC/ODBC	Yes (limited)	No

Source: Lars George (http://www.larsgeorge.com)

References

- C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins: **Pig Latin: A Not-So-Foreign Language for Data Processing.** *Proc. Intl. Conf. on Management of Data (SIGMOD), pp. 1099-1110, 2008.*
- A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, R. Murthy: **Hive – A Petabyte Scale Data Warehouse Using Hadoop.** *Proc. Intl. Conf. on Data Engineering (ICDE), pp. 996-1005, 2010.*

That's All Folks!

"It was much nicer before people started storing all their data in the Cloud."