

Lecture 10: NoSQL

Wednesday, December 1st, 2011

RDBMS v.s. Map/Reduce

Common Wisdom:

- RDBMS load slowly, process fast
- MR load fast, process slower

Hadoop MR v.s. Parallel DBMS

Performance study done at the University of Wisconsin, by Stonebraker, DeWitt, and others, in 2009

- 100-nodes, shared nothing:
- Hadoop MR; DBMS-X; Vertica

- Grep: 10B records x 100bytes = 1TB
- Weblog: a group-by: 2TB = 155M records
- Joins: 2TB x 100GB

Benchmark performance on a 100-node cluster.

	Hadoop	DBMS-X	Vertica	Hadoop/DBMS-X	Hadoop/Vertica
Grep	284s	194s	108x	1.5x	2.6x
Web Log	1,146s	740s	268s	1.6x	4.3x
Join	1,158s	32s	55s	36.3x	21.0x

Discussion

- Repetitive record parsing: MR needs to parse during both M and R.
- Compression in DBMS delivered significant performance gain; unclear why Hadoop MR did not benefit equally from compression
- Pipelining
- Scheduling: static in DBMS (=query plan), dynamic in MR

Five Typical MR applications

- ETL and “read once” data sets; e.g. read logs, parse&clean, perform complex transformations, store information into DBMS – this is ETL
- Complex analytics: would require multipass SQL
- Semistructured data: usually this means key-value pairs, where the number of attributes per record varies
- Quick-and-dirty analysis: idea is that MR provides better “out of the box” experience than RDBMS: but lots of tuning needed
- Limited budget operations (Hadoop MR is free; but no free parallel RDBMS)

No-SQL

NoSQL: Overview

- Main objective: implement distributed state
 - Different objects stored on different servers
 - Same object replicated on different servers
- Main idea: give up some of the ACID constraints to improve performance
- Simple interface:
 - Write (=Put): needs to write all replicas
 - Read (=Get): may get only one
- Eventual consistency ← Strong consistency

NoSQL

“Not Only SQL” or “Not Relational”.

Six key features:

1. Scale horizontally “simple operations”
2. Replicate/distribute data over many servers
3. Simple call level interface (contrast w/ SQL)
4. Weaker concurrency model than ACID
5. Efficient use of distributed indexes and RAM
6. Flexible schema

Outline of this Lecture

- Main techniques and concepts:
 - Distributed storage using DHTs
 - Consistency: 2PC, vector clocks
 - The CAP theorem
- Overview of No-SQL systems (Cattell)
- Short case studies:
 - Dynamo, Cassandra, PNUTS
- Critique (c.f. Stonebraker)

Main Techniques and Concepts

Main Techniques, Concepts

- Distributed Hash Tables
- Consistency: 2PC, Vector Clocks
- The CAP theorem

A Note

- These techniques belong to a course on distributed systems, and not databases
- We will mention them because they are very relevant to NoSQL, but this is not an exhaustive treatment

Distributed Hash Table

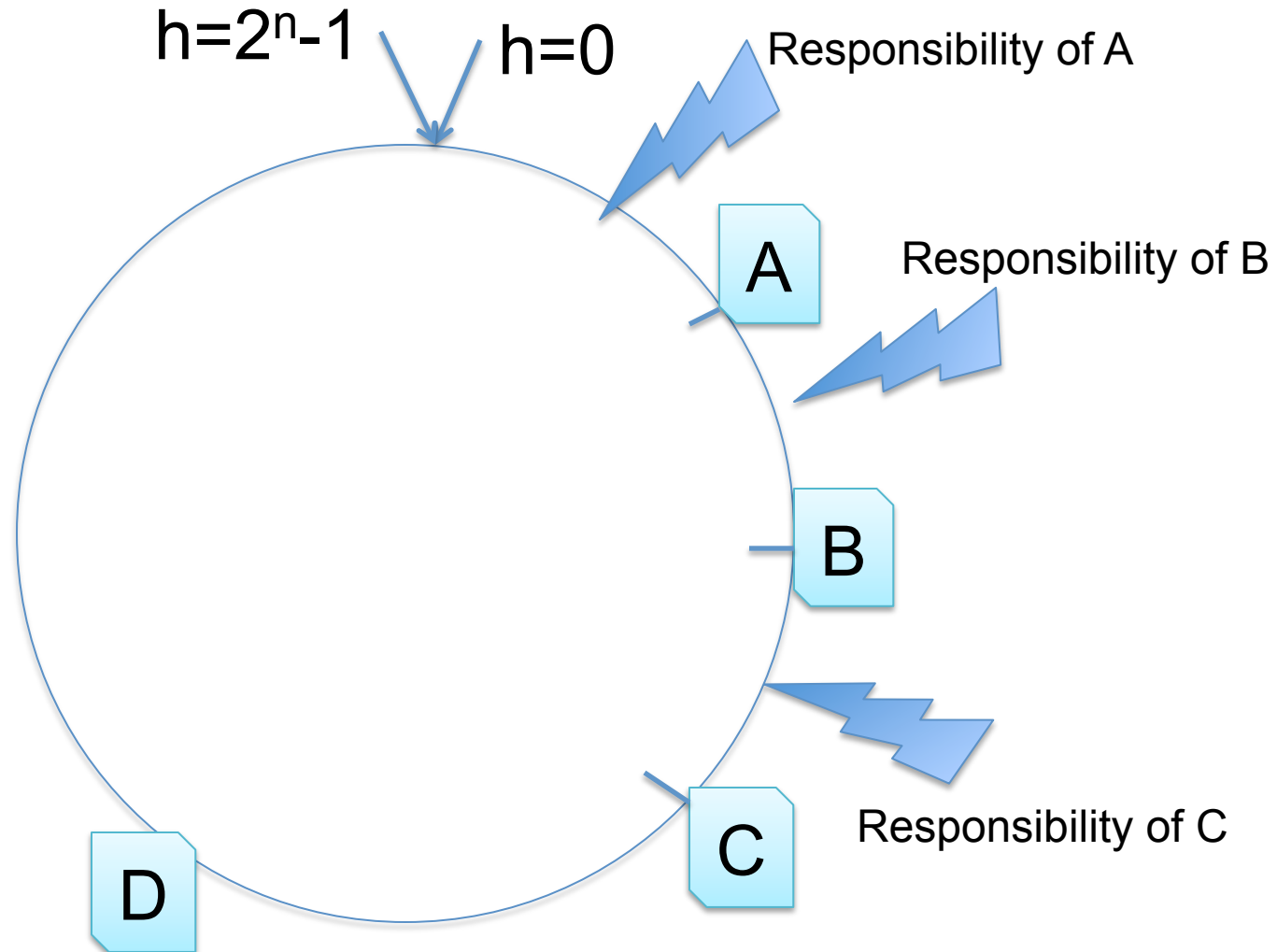
Implements a distributed storage

- Each key-value pair (k,v) is stored at some server $h(k)$
- API: $\text{write}(k,v)$; $\text{read}(k)$

Use standard hash function: service key k by server $h(k)$

- Problem 1: a client knows only one server, doesn't know how to access $h(k)$
- Problem 2. if new server joins, then $N \rightarrow N+1$, and the entire hash table needs to be reorganized
- Problem 3: we want replication, i.e. store the object at more than one server

Distributed Hash Table



Problem 1: Routing

A client doesn't know server $h(k)$, but some other server

- Naive routing algorithm:
 - Each node knows its neighbors
 - Send message to nearest neighbor
 - Hop-by-hop from there
 - Obviously this is $O(n)$, So no good
- Better algorithm: “finger table”
 - Memorize locations of other nodes in the ring
 - $a, a + 2, a + 4, a + 8, a + 16, \dots a + 2^n - 1$
 - Send message to closest node to destination
 - Hop-by-hop again: this is $\log(n)$

Problem 1: Routing

$h(k)$ handled by server G

Read(k)

$h=2^n-1$ $h=0$

Client only "knows" server A

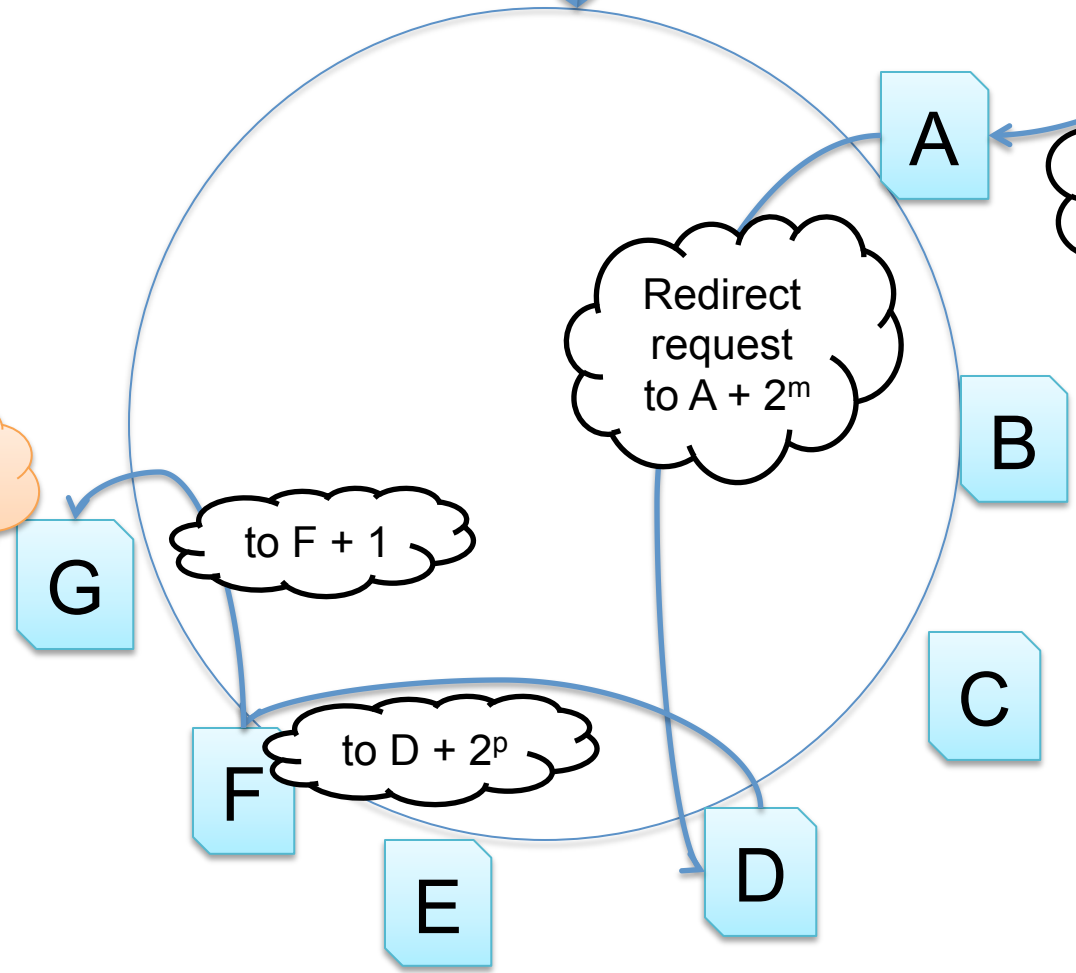
Redirect request to $A + 2^m$

Found Read(k)!

to $F + 1$

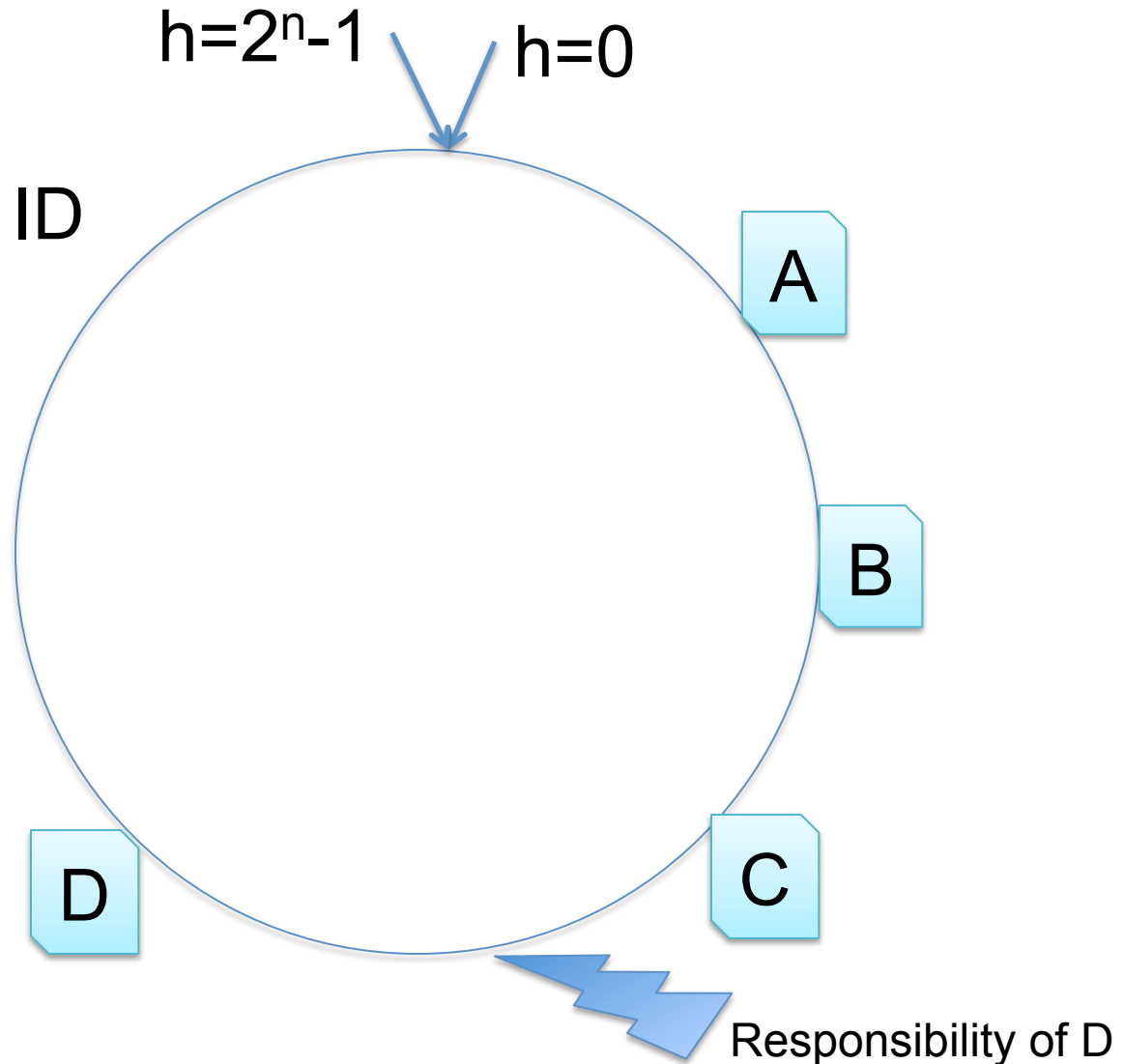
to $D + 2^p$

$O(\log n)$



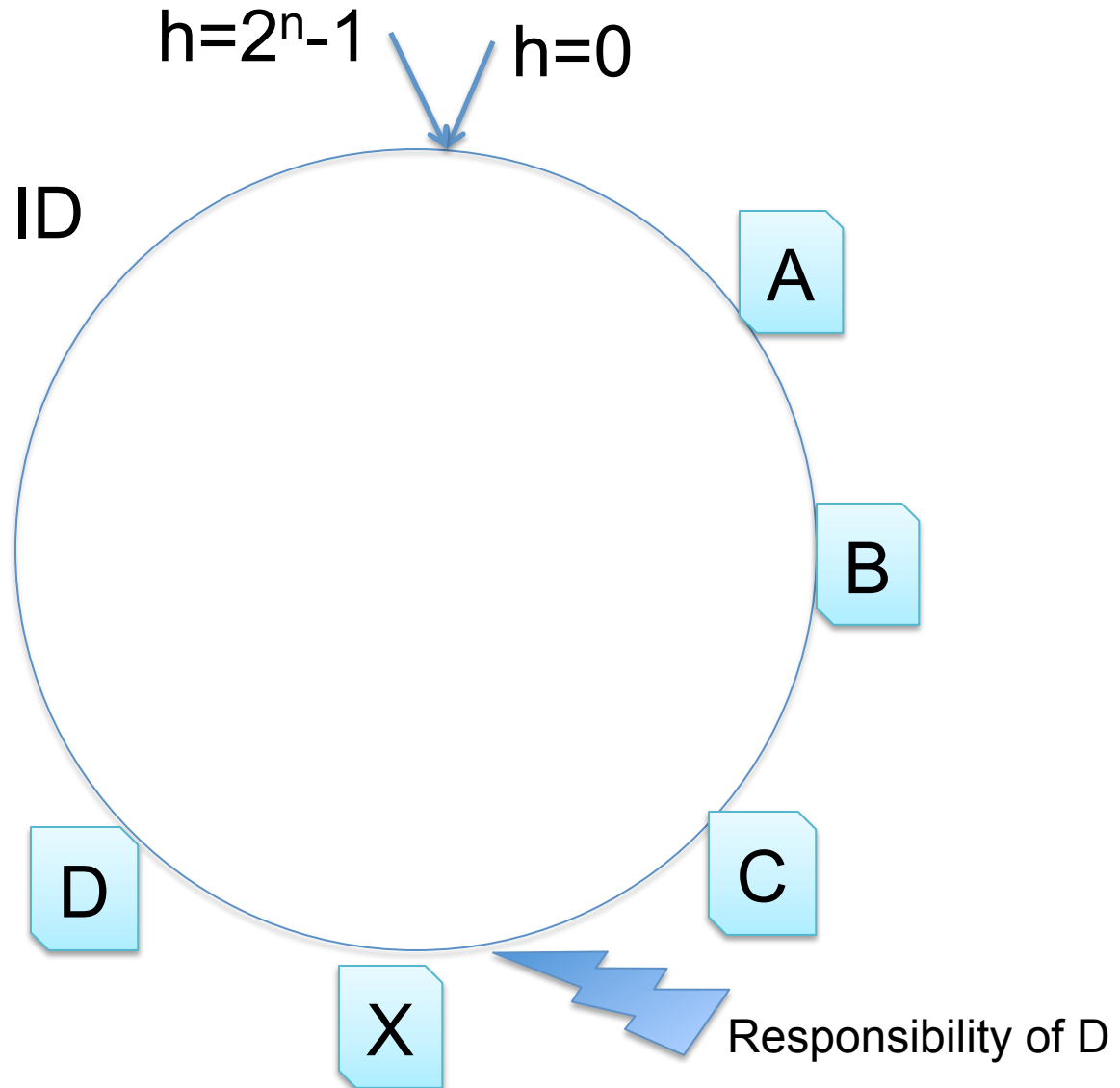
Problem 2: Joining

When X joins:
select random ID



Problem 2: Joining

When X joins:
select random ID

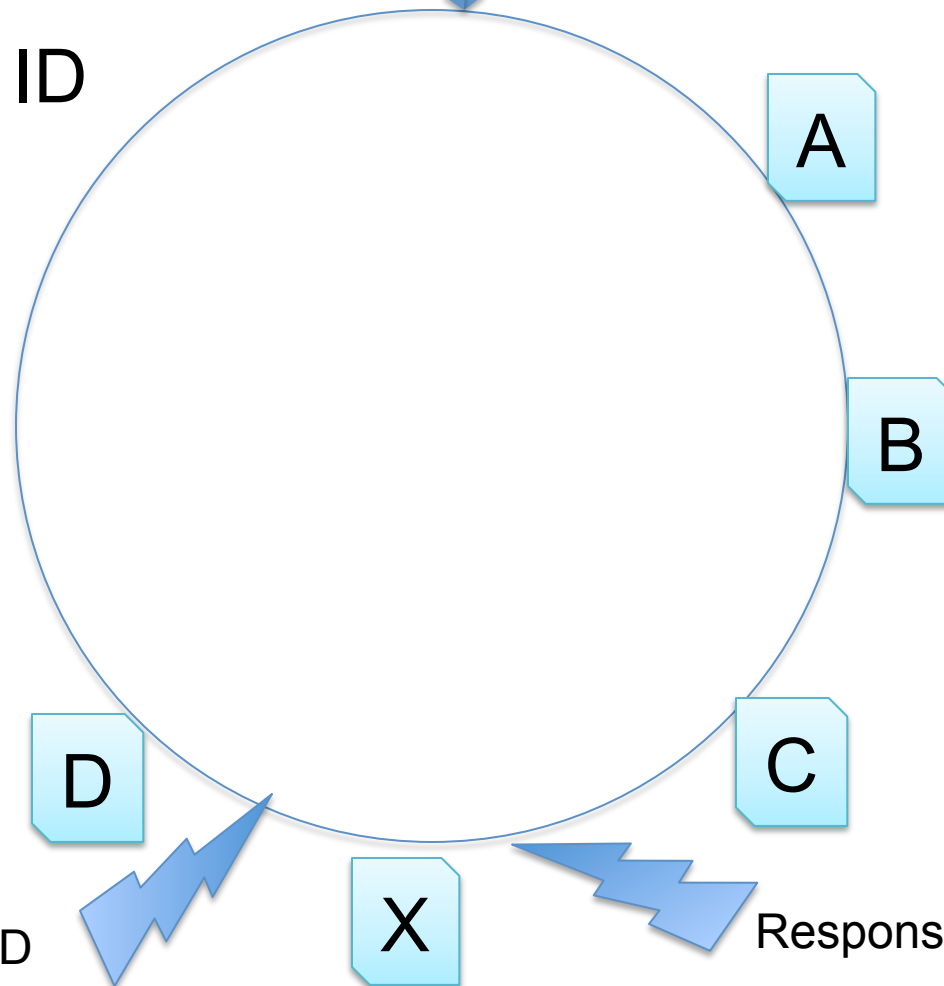


Problem 2: Joining

$h=2^n-1$ $h=0$

When X joins:
select random ID

Redistribute
the load at D



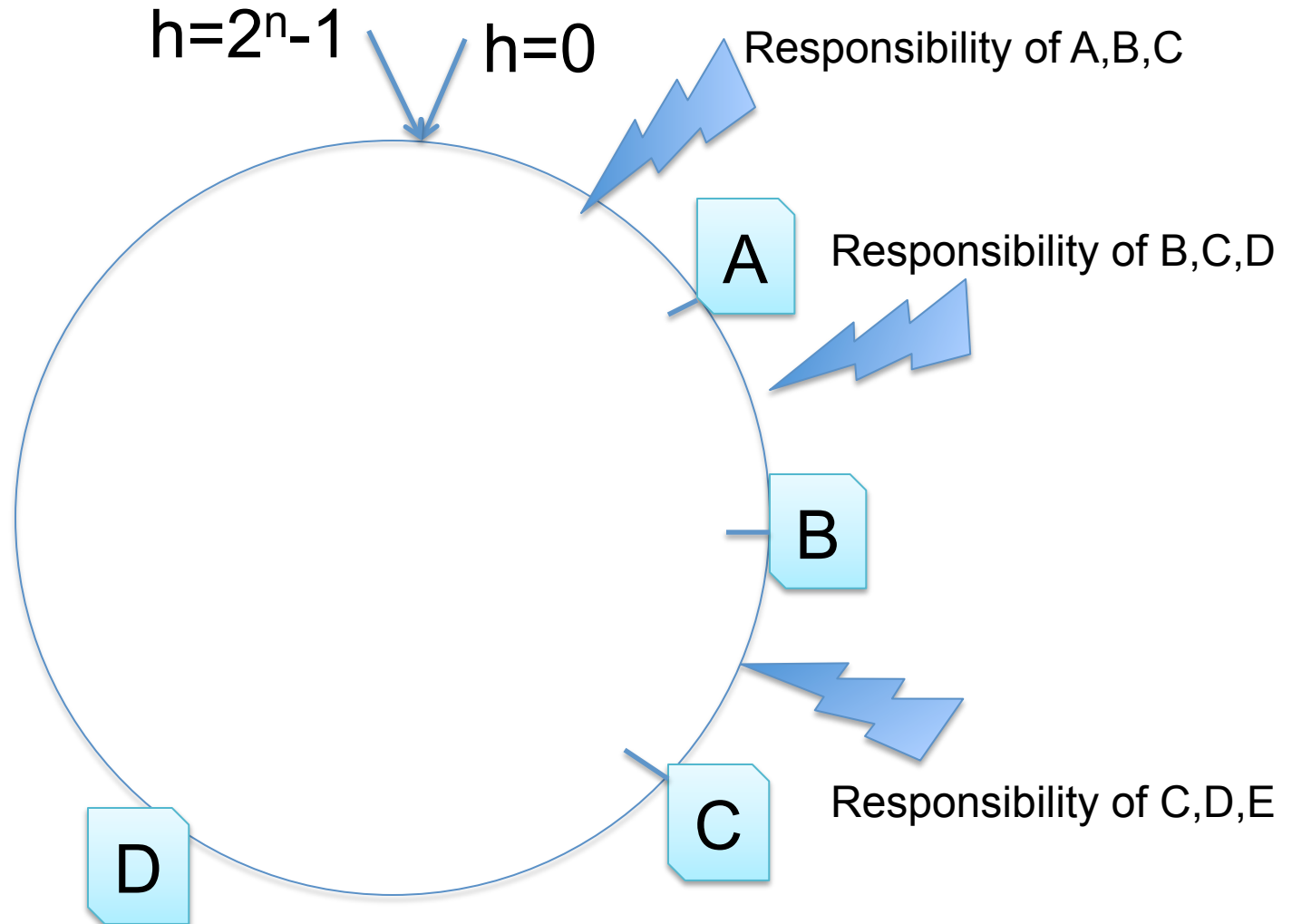
Responsibility of D

Responsibility of X

Problem 3: Replication

- Need to have some degree of replication to cope with node failure
- Let N =degree of replication
- Assign key k to $h(k), h(k)+1, \dots, h(k)+N-1$

Problem 3: Replication



Consistency

- ACID
 - Two phase commit
 - Paxos (will not discuss)
- Eventual consistency
 - Vector clocks

Two Phase Commit

- Multiple servers run parts of the same transaction
- They all must commit, or none should commit
- Two-phase commit is a complicated protocol that ensures that
- 2PC can also be used for WRITE with replication: commit the write at all replicas before declaring success

Two Phase Commit

Assumptions:

- Each site logs actions at that site, but there is no global log
- There is a special site, called the *coordinator*, which plays a special role
- 2PC involves sending certain messages: as each message is sent, it is logged at the sending site, to aid in case of recovery

Two-Phase Commit

Book, Sec. 21.13.1

1. Coordinator sends prepare message
2. Subordinates receive prepare statement; force-write **<prepare>** log entry; answers yes or no
3. If coordinator receives only yes, force write **<commit>**, sends commit messages;
If at least one no, or timeout, force write **<abort>**, sends abort messages
4. If subordinate receives abort, force-write **<abort>**, sends ack message and aborts; if receives commit, force-write **<commit>**, sends ack, commits.
5. When coordinator receives all ack, writes **<end log>**

Two-Phase Commit

- ACID properties, but expensive
- Relies on central coordinator: both performance bottleneck, and single-point-of-failure
- Solution: Paxos = distributed protocol
 - Complex: will not discuss at all

Vector Clocks

- An extension of Multiversion Concurrency Control (MVCC) to multiple servers
- Standard MVCC:
each data item X has a timestamp t :
 $X_4, X_9, X_{10}, X_{14}, \dots, X_t$
- Vector Clocks:
 X has set of [server, timestamp] pairs
 $X([s_1, t_1], [s_2, t_2], \dots)$

Vector Clocks

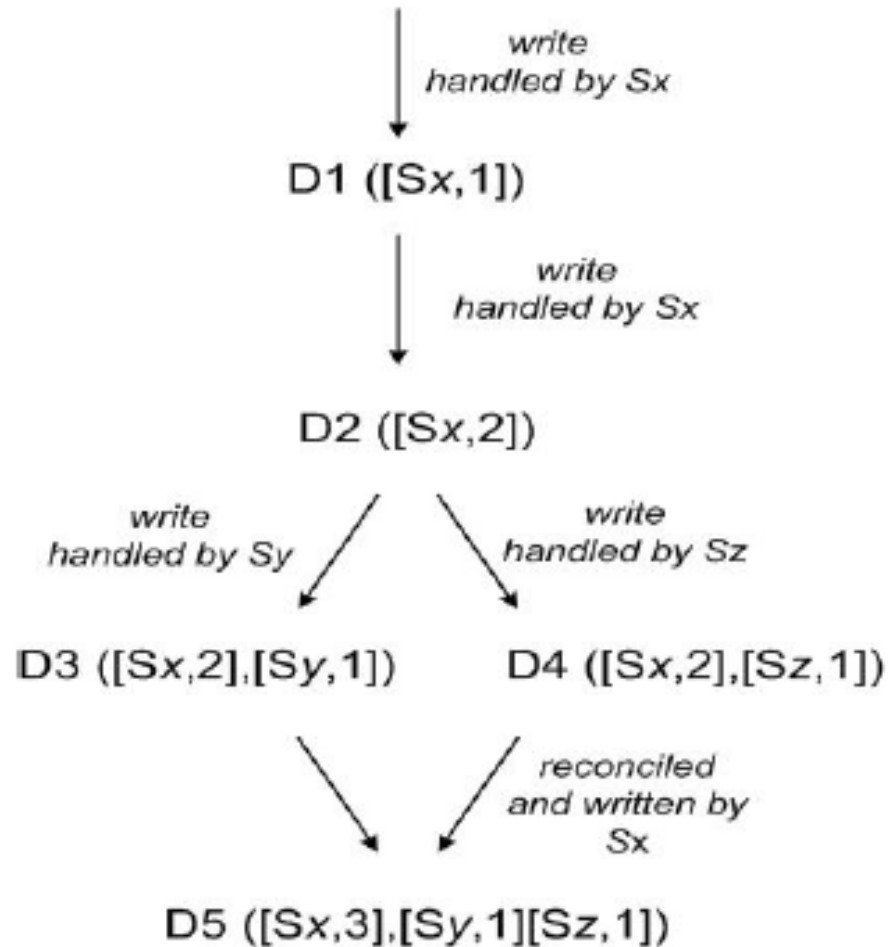


Figure 3: Version evolution of an object over time.

Vector Clocks: Example

- A client writes D1 at server SX:
D1 ([SX,1])
- Another client reads D1, writes back D2; also handled by server SX:
D2 ([SX,2]) (D1 garbage collected)
- Another client reads D2, writes back D3; handled by server SY:
D3 ([SX,2], [SY,1])
- Another client reads D2, writes back D4; handled by server SZ:
D4 ([SX,2], [SZ,1])
- Another client reads D3, D4: CONFLICT !

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
$([SX,3],[SY,6])$	$([SX,3],[SZ,2])$	Yes
$([SX,3])$	$([SX,5])$	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	Yes

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	Yes
([SX,3],[SY,10])	([SX,3],[SY,20],[SZ,2])	

Vector Clocks: Conflict or not?

Data 1	Data 2	Conflict ?
([SX,3],[SY,6])	([SX,3],[SZ,2])	Yes
([SX,3])	([SX,5])	No
([SX,3],[SY,6])	([SX,3],[SY,6],[SZ,2])	No
([SX,3],[SY,10])	([SX,3],[SY,6],[SZ,2])	Yes
([SX,3],[SY,10])	([SX,3],[SY,20],[SZ,2])	No

CAP Theorem

Brewer 2000:

You can only have two of the following three:

- Consistency
- Availability
- Tolerance to Partitions

CAP Theorem: No Partitions

- CA = Consistency + Availability
- Single site database
- Cluster database
- Need 2 phase commit
- Need cache validation protocol

CAP Theorem: No Availability

- CP = Consistency + tolerance to Partitions
- Distributed databases
- Majority protocols
- Make minority partitions unavailable

CAP Theorem: No Consistency

- AP = Availability + tolerance to Partitions
- DNS
- Web caching

CAP Theorem: Criticism

- Not really a “theorem”, since definitions are imprecise: a real theorem was proven a few years later, but under more limiting assumptions
- Many tradeoffs possible
- D.Abadi: “CP makes no sense” because it suggest *never* available. A, C asymmetric!
 - No “C” = *all the time*
 - No “A” = *only when the network is partitioned*

Overview of No-SQL systems

Early “Proof of Concepts”

- Memcached: demonstrated that in-memory indexes (DHT) can be highly scalable
- Dynamo: pioneered *eventual consistency* for higher availability and scalability
- BigTable: demonstrated that persistent record storage can be scaled to thousands of nodes

ACID v.s. BASE

- ACID = Atomicity, Consistency, Isolation, and Durability
- BASE = Basically Available, Soft state, Eventually consistent

Terminology

- **Simple operations** = key lookups, read/writes of one record, or a small number of records
- **Sharding** = horizontal partitioning by some key, and storing records on different servers in order to improve performance.
- **Horizontal scalability** = distribute both data *and* load over many servers
- **Vertical scaling** = when a dbms uses multiple cores and/or CPUs

Not exactly same as horizontal partitioning

Definitely different from vertical partitioning

Data Model

- **Tuple** = row in a relational db
- **Document** = nested values, extensible records (think XML or JSON)
- **Extensible record** = families of attributes have a schema, but new attributes may be added
- **Object** = like in a programming language, but without methods

1. Key-value Stores

Think “file system” more than “database”

- Persistence,
- Replication
- Versioning,
- Locking
- Transactions
- Sorting

1. Key-value Stores

- Voldemort, Riak, Redis, Scalaris, Tokyo Cabinet, Memcached/Membrain/Membase
- Consistent hashing (DHT)
- Only primary index: lookup by key
- No secondary indexes
- Transactions: single- or multi-update TXNs
 - locks, or MVCC

2. Document Stores

- A "document" = a pointerless object = e.g. JSON = nested or not = schema-less
- In addition to KV stores, may have secondary indexes

2. Document Stores

- SimpleDB, CouchDB, MongoDB, Terrastore
- Scalability:
 - Replication (e.g. SimpleDB, CouchDB – means entire db is replicated),
 - Sharding (MongoDB);
 - Both

3. Extensible Record Stores

- Typical Access: Row ID, Column ID, Timestamp
- Rows: sharding by primary key
 - BigTable: split table into tablets = units of distribution
- Columns: "column groups" = indication for which columns to be stored together (e.g. customer name/address group, financial info group, login info group)
- HBase, HyperTable, Cassandra, PNUT, BigTable

4. Scalable Relational Systems

- Means RDBS that are offering sharding
- Key difference: NoSQL make it difficult or impossible to perform large-scope operations and transactions (to ensure performance), while scalable RDBMS do not *preclude* these operations, but users pay a price only when they need them.
- MySQL Cluster, VoltDB, Clusterix, ScaleDB, Megastore (the new BigTable)

Application 1

- Web application that needs to display lots of customer information; the users data is rarely updated, and when it is, you know when it changes because updates go through the same interface. Store this information persistently using a KV store.

Key-value store

Application 2

- Department of Motor Vehicle: lookup objects by multiple fields (driver's name, license number, birth date, etc); "eventual consistency" is ok, since updates are usually performed at a single location.

Document Store

Application 3

- eBay style application. Cluster customers by country; separate the rarely changed "core" customer information (address, email) from frequently-updated info (current bids).

Extensible Record Store

Application 4

- Everything else (e.g. a serious DMV application)

Scalable RDBMS

Short Case Studies

Case Study 1: Dynamo

- Developed at Amazon, published 2007
- It is probably in SimpleDB today, I couldn't confirm
- Was the first to demonstrate that eventual consistency can work

Case Study 1: Dynamo

Key features:

- Service Level Agreement (SLN): at the 99th percentile, and not on mean/median/variance (otherwise, one penalizes the heavy users)
 - “Respond within 300ms for 99.9% of its requests”

Case Study 1: Dynamo

Key features:

- DHT with replication:
 - Store value at $k, k+1, \dots, k+N-1$
- Eventual consistency through vector clocks
- Reconciliation at read time:
 - Writes never fail (“poor customer experience”)
 - Conflict resolution: “last write wins” or application specific

Case Study 2: Cassandra

- Cassandra stores semi-structured rows that belong to column families
 - Rows are accessed by a **key**
 - Rows are replicated and distributed by hashing keys
- Multi-master replication for each row
 - Enables Cassandra to run in multiple data centers
 - Also gives us partition tolerance

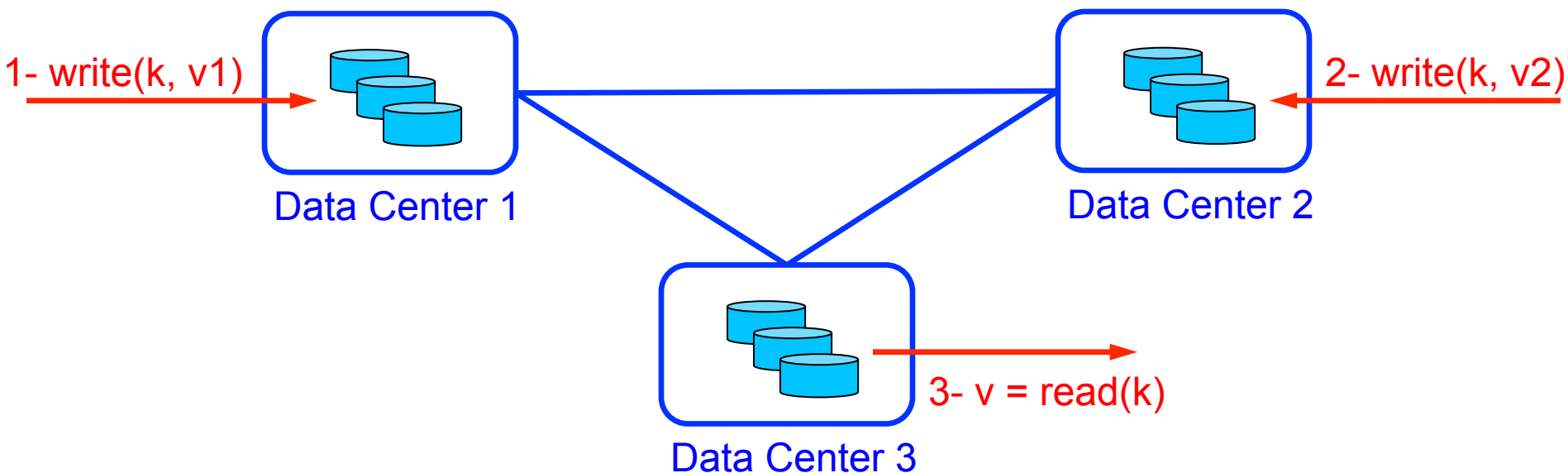
Case Study 2: Cassandra

- Client controls the consistency vs. latency trade-off for each read and write operation
 - write(1)/read(1) – fast but not necessarily consistent
 - write(ALL)/read(ALL) – consistent but may be slow
- Client decides the serialization order of updates
- Scalable, elastic, highly available
 - Like many other cloud storage systems!

Consistency vs. Latency

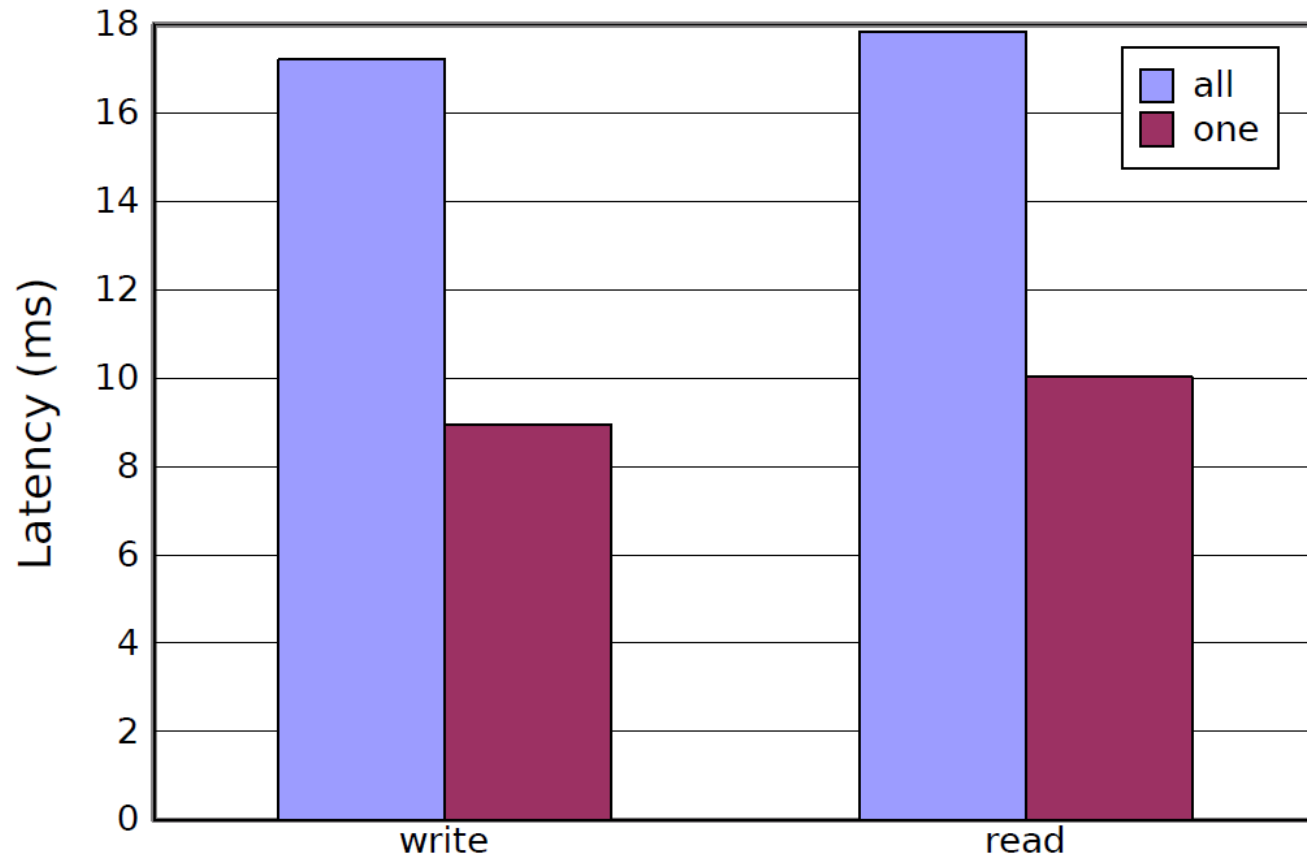
- `value = read(1, key, column)`
 - Send read request to all replicas of the row (based on key)
 - Return first response received to client
 - Returns quickly but may return stale data
- `value = read(ALL, key, column)`
 - Send read request to all replicas of the row (based on key)
 - Wait until all replicas respond and return ***latest version*** to client
 - Consistent but as slow as the slowest replica
- `write(1) vs. write(ALL)`
 - Send write request to all replicas
 - ***Client provides a timestamp for each write***
- Other consistency levels are supported

Consistency vs. Latency



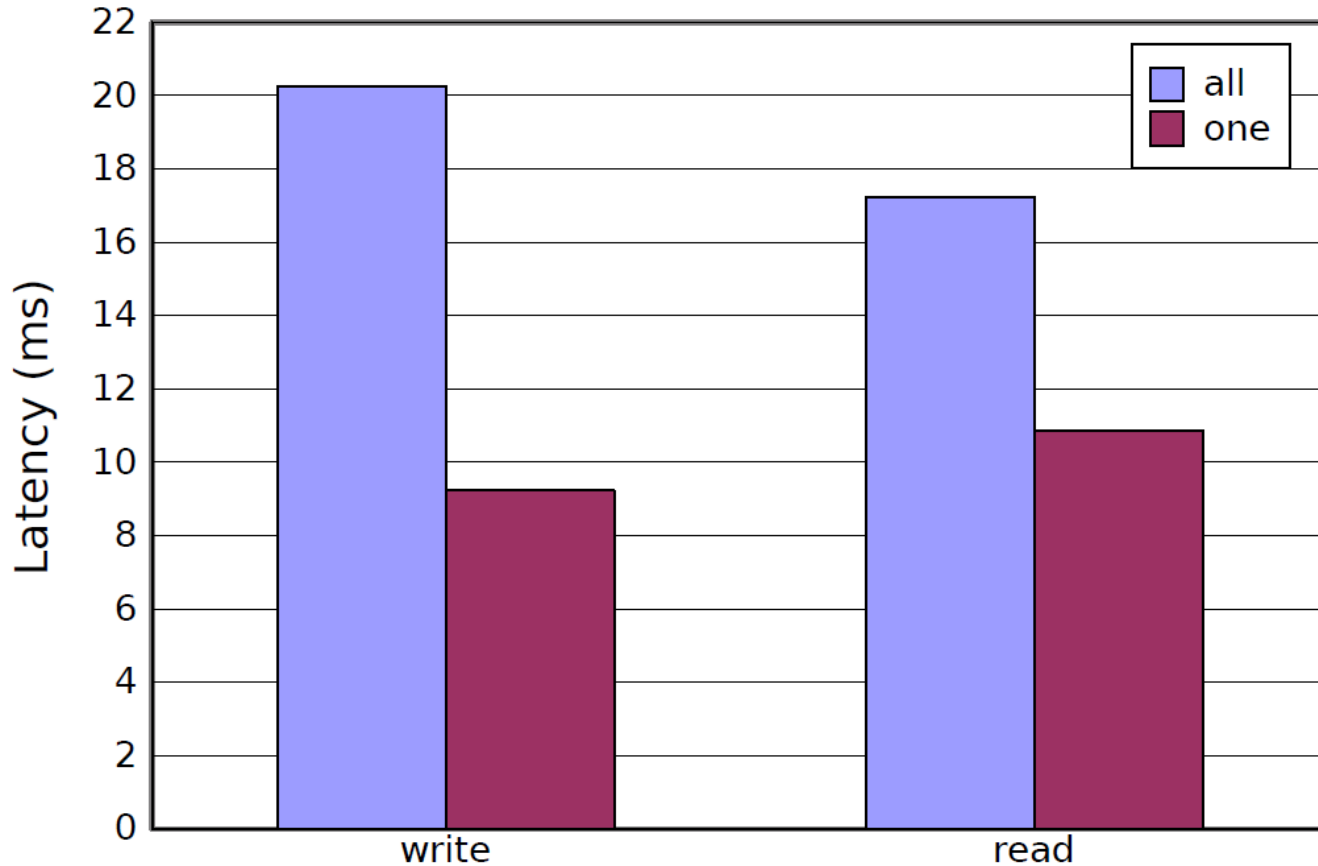
- Which v is returned to the read()?
 - write(1)/read(1): possibly $v1$, and **eventually** $v2$
 - write(ALL)/read(1): guaranteed to return $v2$ if successful
 - write(1)/read(ALL): guaranteed to return $v2$ if successful

Consistency vs. Latency



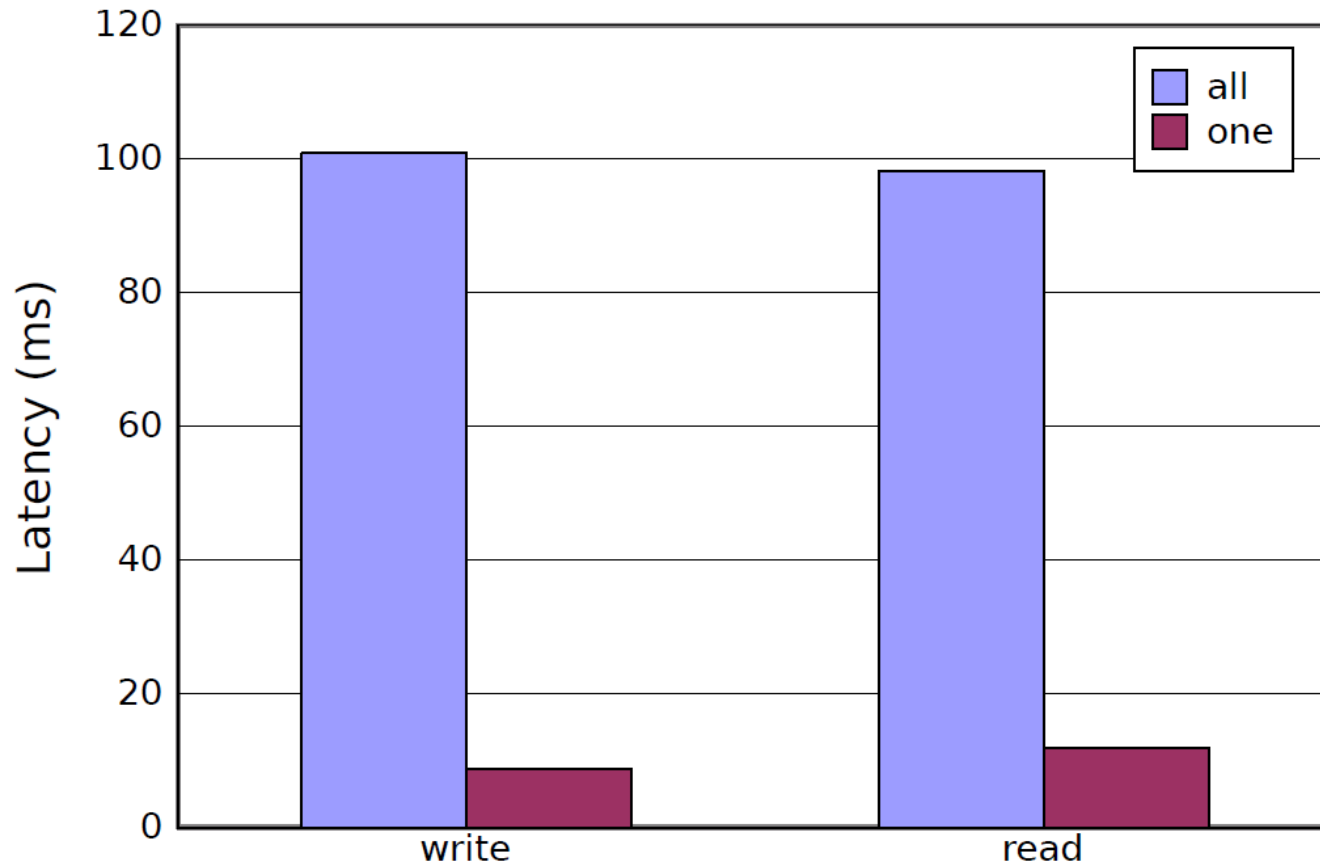
*Experiment on Amazon EC2 – Yahoo! Cloud Serving Benchmark (YCSB) – 4 Cassandra Nodes
Same EC2 Availability Zone*

Consistency vs. Latency



*Two EC2 Availability Zones
Same EC2 Geographic Region*

Consistency vs. Latency

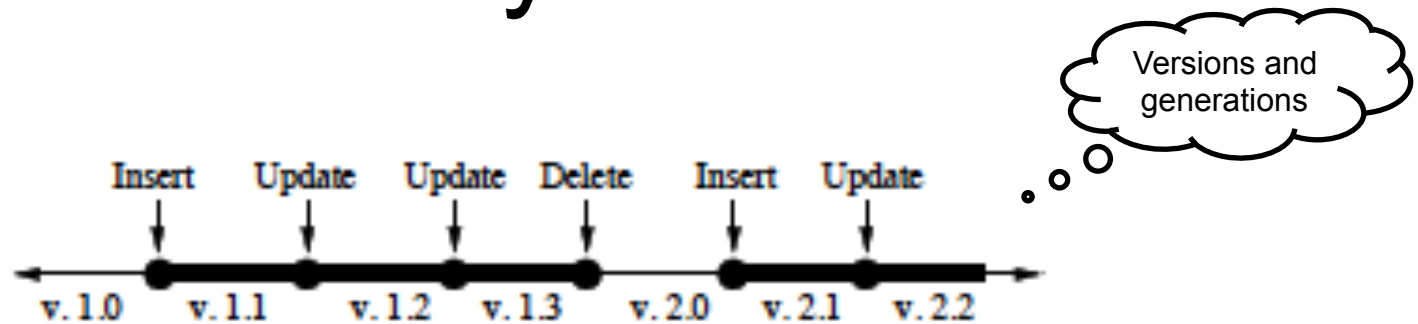


*Two EC2 Regions
(US East and US West)*

Case Study 3: PNUTS

- Yahoo; the only system that has a benchmark, and thorough experimental evaluation

Case Study 3: PNUTS



- Read-any = returns any stable version
- Read-critical(required_version) = reads a version that is strictly newer
- Read-latest = reads absolute latest
- Test-and-set-write(required_version) = writes only if current version is the required one

Criticism

Criticism

- Two ways to improve OLTP performance:
 - Sharding over shared-nothing
 - Improve per-server OLTP performance
- Recent RDBMs do provide sharding: Greenplum, Aster Data, Vertica, ParAccel
- Hence, the discussion is about single-node performance

Criticism (cont'd)

- Single-node performance:
- Major performance bottleneck: communication with DBMS using ODBC or JDBC
 - Solution: stored procedures, OR embedded databases
- Server-side performance (next slide)

Criticism (cont'd)

Server-side performance: about 25% each

- Logging
 - Everything written twice; log must be forced
- Locking
 - Needed for ACID semantics
- Latching
 - This is when the DBMS itself is multithreaded; e.g. latch for the lock table
- Buffer management

Criticism (cont'd)

Main take-away:

- NoSQL databases give up 1, or 2, or 3 of those features
- Thus, performance improvement can only be modest
- Need to give up all 4 features for significantly higher performance
- On the downside, NoSQL give up ACID

Criticism (cont'd)

Who are the customers of NoSQL?

- Lots of startups
- Very few enterprises. Why? most applications are traditional OLTP on structured data; a few other applications around the “edges”, but considered less important

Criticism (cont'd)

- No ACID Equals No Interest
 - Screwing up mission-critical data is no-no-no
- Low-level Query Language is Death
 - Remember CODASYL?
- NoSQL means NoStandards
 - One (typical) large enterprise has 10,000 databases. These need accepted standards