CSEP 544: Lecture 10

Column-Oriented Databases and NoSQL

Announcement

Take home final: 3/15-16

- Online Webquiz
  - Need your UW NET ID, check that it works!
  - I will also email the final in pdf form (e.g. to print)
- Opens Friday night, closes Sunday night
- No time limits:
  - Work, save, take a break, return later…
- No need to run code
- Questions?
  - Email me and cc Priya
- Watch your email
  - E.g. corrections
- No discussion of the final with colleagues
- When you are done:
  - Submit and receive confirmation code!

Today's Agenda

- Column-oriented databases
- No-SQL
- Evaluation forms

Column-Oriented Databases

Brief discussion of the paper:
The Design and Implementation of Modern Column-Oriented Database Systems

Column-Oriented Databases

- Main idea:
  - Physical storage: complete vertical partition; each column stored separately: R.A, R.B, R.A
  - Logical schema: remains the same R(A,B,C)

- Main advantage:
  - Improved transfer rate: disk to memory, memory to CPU, better cache locality
  - Other advantages (next)

Key Architectural Trends (Sec.1)

- Virtual IDs
- Block-oriented and vertical processing
- Late materialization
- Column-specific compression

Key Architectural Trends (Sec.1)

- Virtual IDs
  - Offsets (arrays) instead of keys
- Block-oriented and vertical processing
  - Iterator model: one tuple→one block of tuples
- Late materialization
  - Postpone tuple reconstruction in query plan
- Column-specific compression
  - Much better than row-compression (why?)

Discussion

- What are “covering indexes” (pp. 204)
  And what is their connection to column-oriented databases?

- What is the main takeaway from Fig. 1.2?

Discussion

- What are “covering indexes” (pp. 204) And what is their connection to column-oriented databases?
  - A set of indexes that can completely answer the query; one index ≈ one column
- What is the main takeaway from Fig. 1.2?
  - Column-oriented databases don’t work! Unless you really optimize them well

Compression (Sec. 4)

- What is the advantage of compression in databases?

- Discuss main column-at-a-time compression techniques

Compression (Sec. 4)

- What is the advantage of compression in databases?

- Discuss main column-at-a-time compression techniques
  - Row-length encoding: F,F,F,F,M,M→4F,2M
  - Bit-vector (see also bit-map indexes)
  - Dictionary. More generally: Ziv-Lempel

Late Materialization (Sec. 4)

- What is it?

- Discuss $\Pi_C(\sigma_{A='a' \wedge B='b'}(R(A,B,C,D,\ldots))$

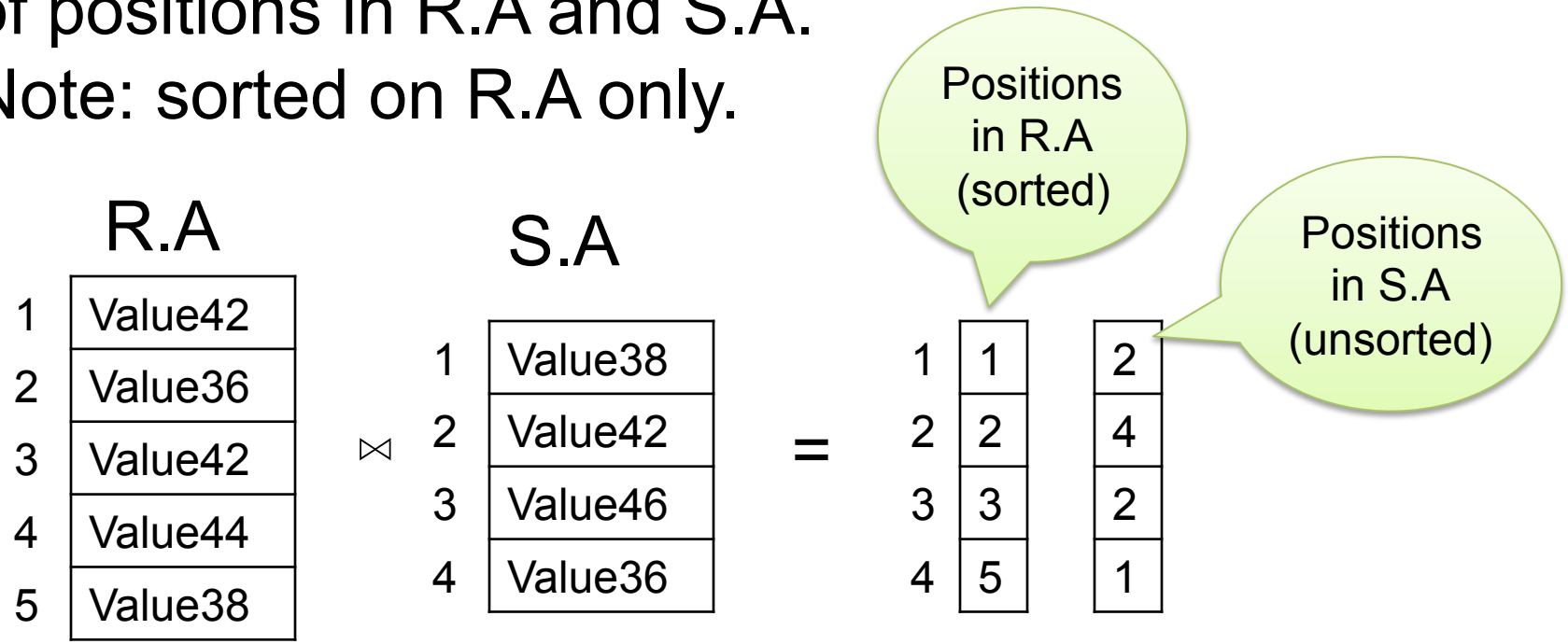
Late Materialization (Sec. 4)

- What is it?
  - The result is an array of positions

- Discuss $\Pi_C(\sigma_{A='a' \wedge B='b'}(R(A,B,C,D,\ldots))$
  - Retrieve positions in column A: 2, 4, 5, 9, 25…
  - Retrieve positions in column B: 3, 4, 7, 9,12,..
  - Intersect: 4, 9, …
  - Lookup values in column C: C[4], C[9], …

Joins (Sec. 4)

The result of a join R.A ⋈ S.A is an array of positions in R.A and S.A.
Note: sorted on R.A only.

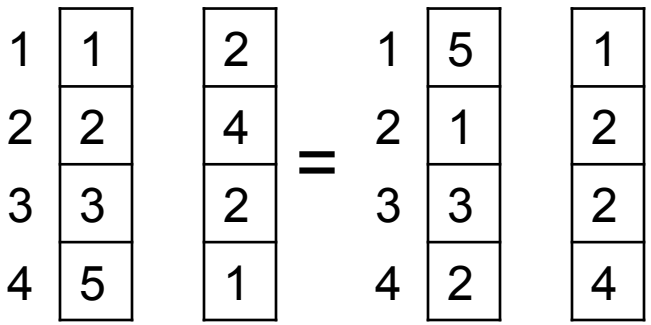


Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$

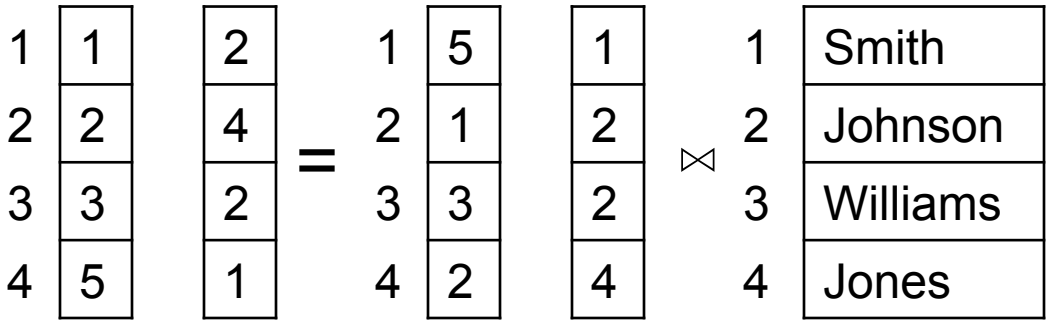


Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$



Sort on positions in S.B

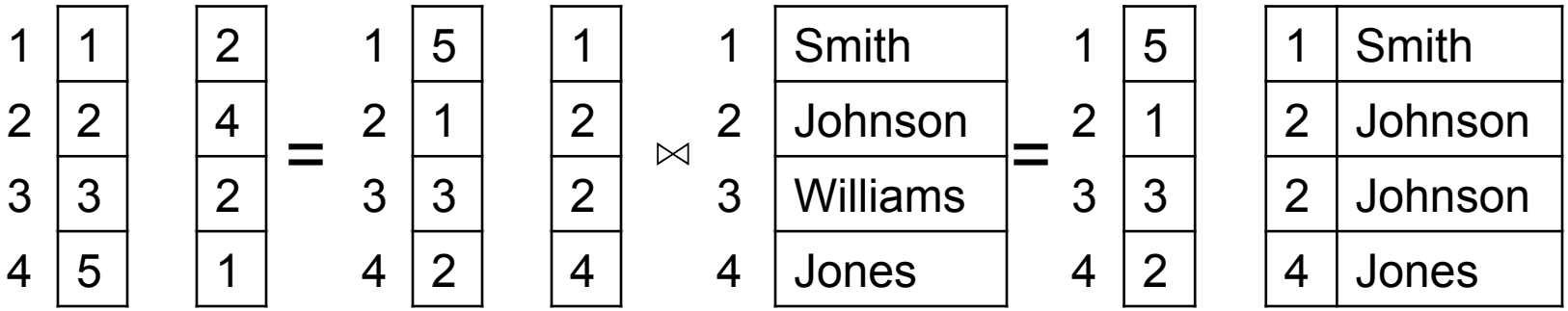
Lookup S.C (this is a merge-join; why?)

Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$

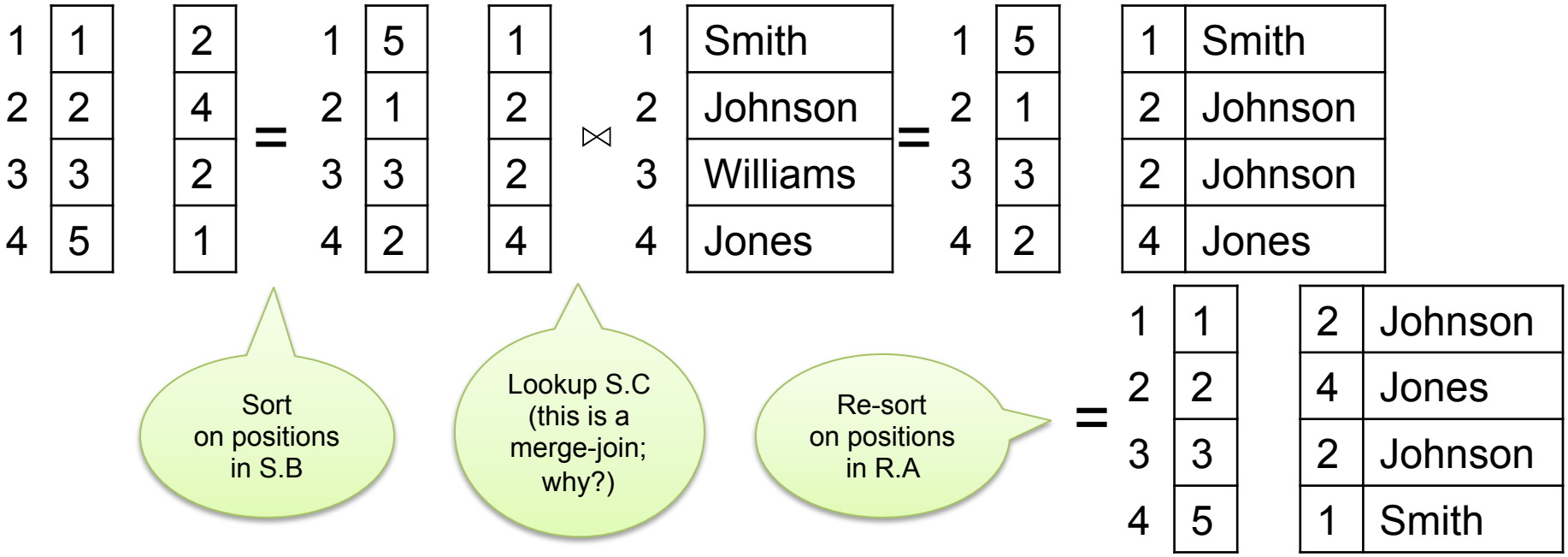


Jive-Join (Sec. 4)

Problem: accessing the values in the second table has poor memory locality

Solution: re-sort by the second coljun, fetch, sort back

E.g. $\Pi_{S.C}(R(A,\ldots) \bowtie S(B,C,\ldots))$



NoSQL Databases

Based on paper by Cattell, in SIGMOD Record 2010

NoSLQ: 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

Cattell, SIGMOD Record 2010

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: write(k,v); read(k)

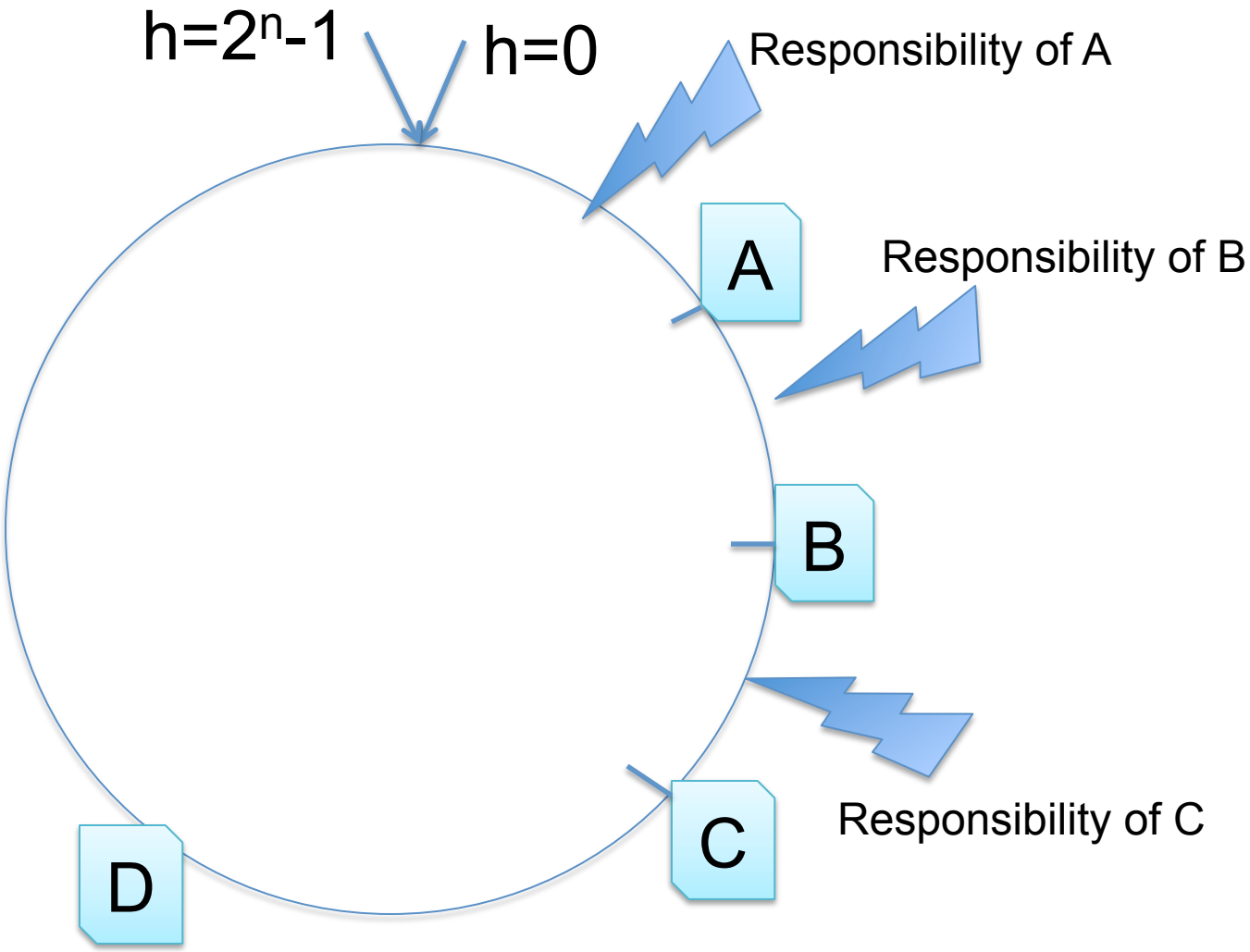
Use standard hash function: service key k by server h(k)

- Problem 1: a client knows only one server, does't know how to access h(k)

- Problem 2. if new server joins, then N → 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



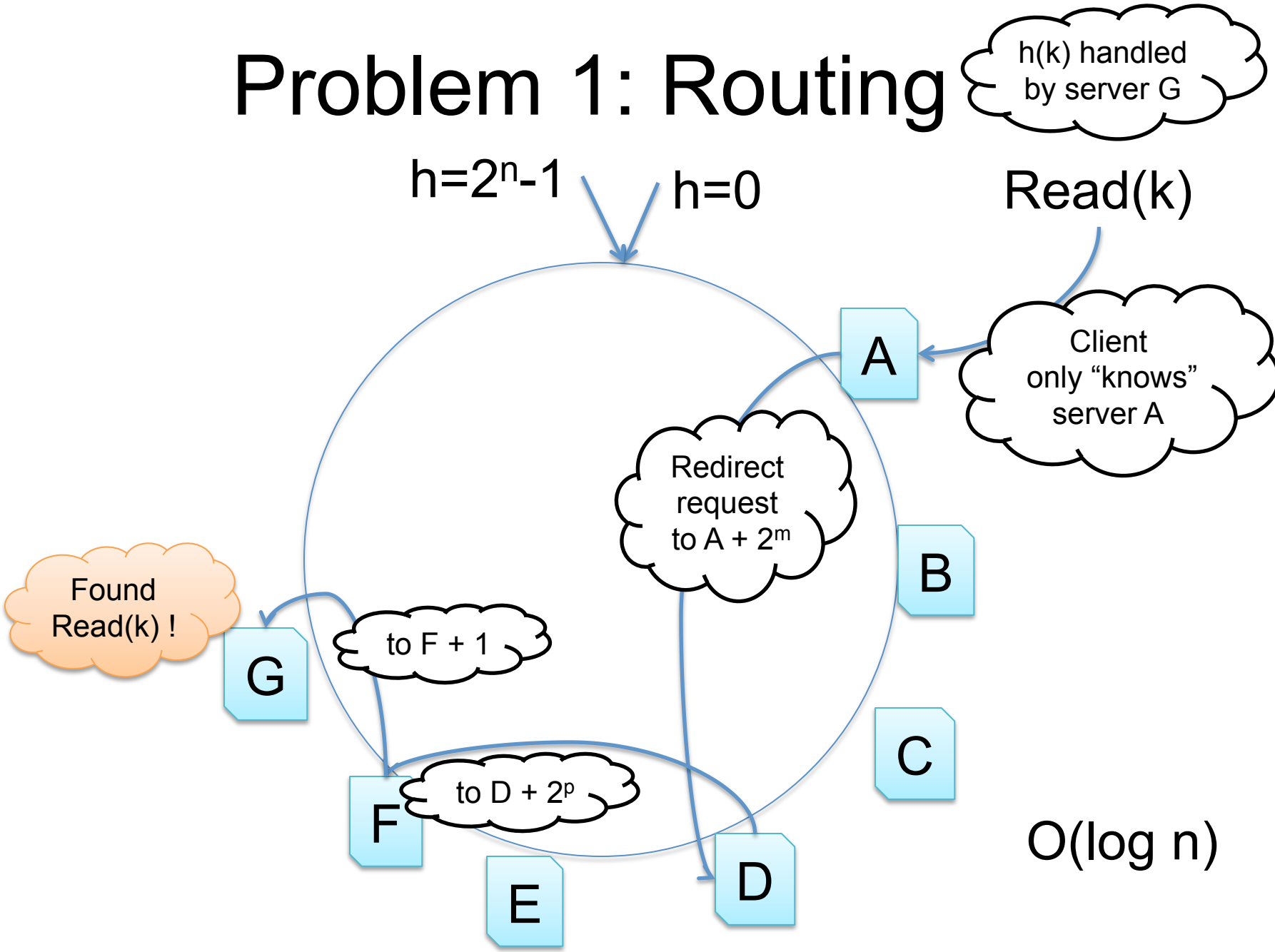
Problem 1: Routing

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

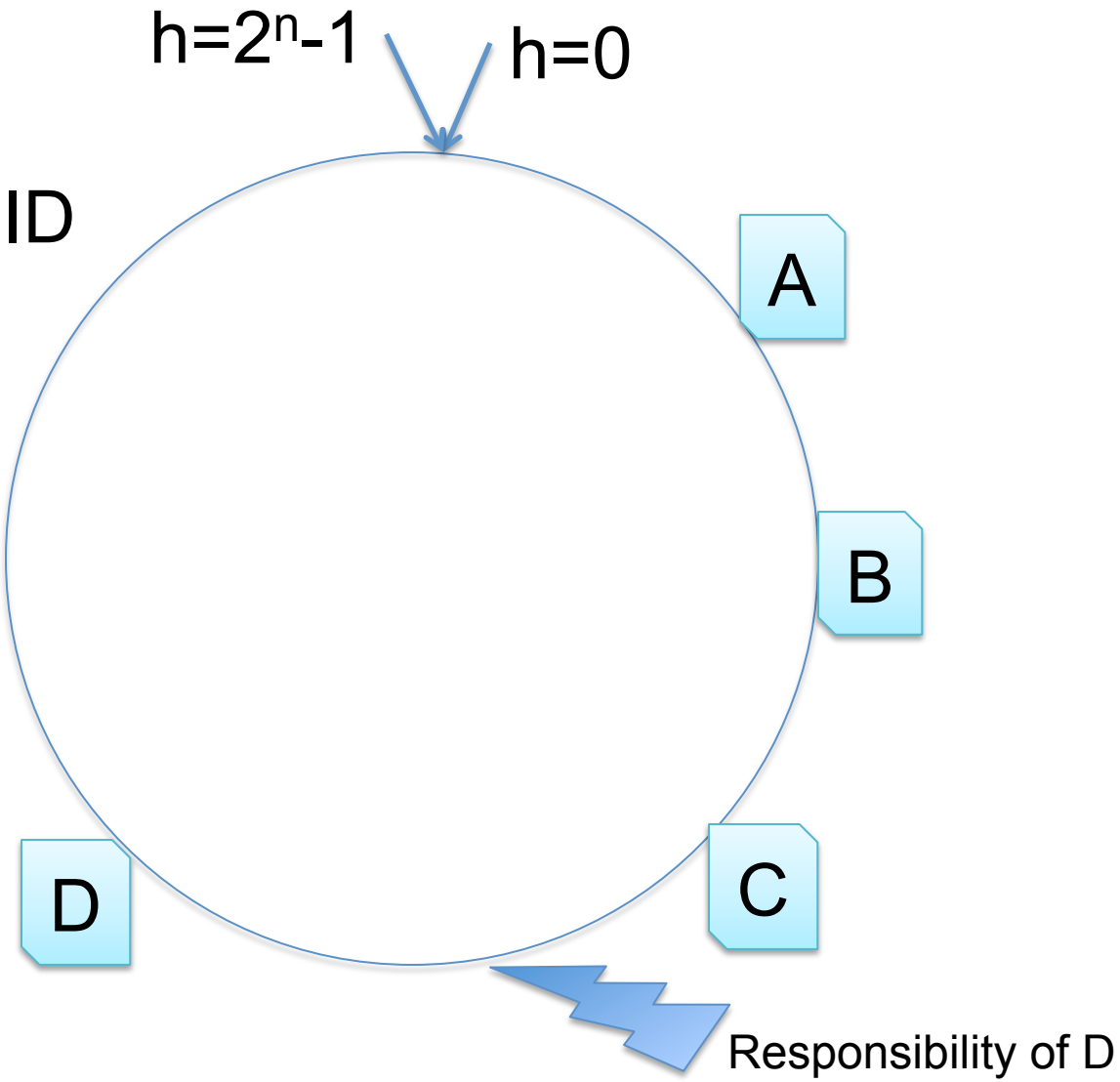
- Better algorithm: "finger table"
  - Memorize locations of other nodes in the ring
  - a, a + 2, a + 4, a + 8, a + 16, ... a + 2^n – 1
  - Send message to closest node to destination
  - Hop-by-hop again: this is log(n)

Problem 1: Routing



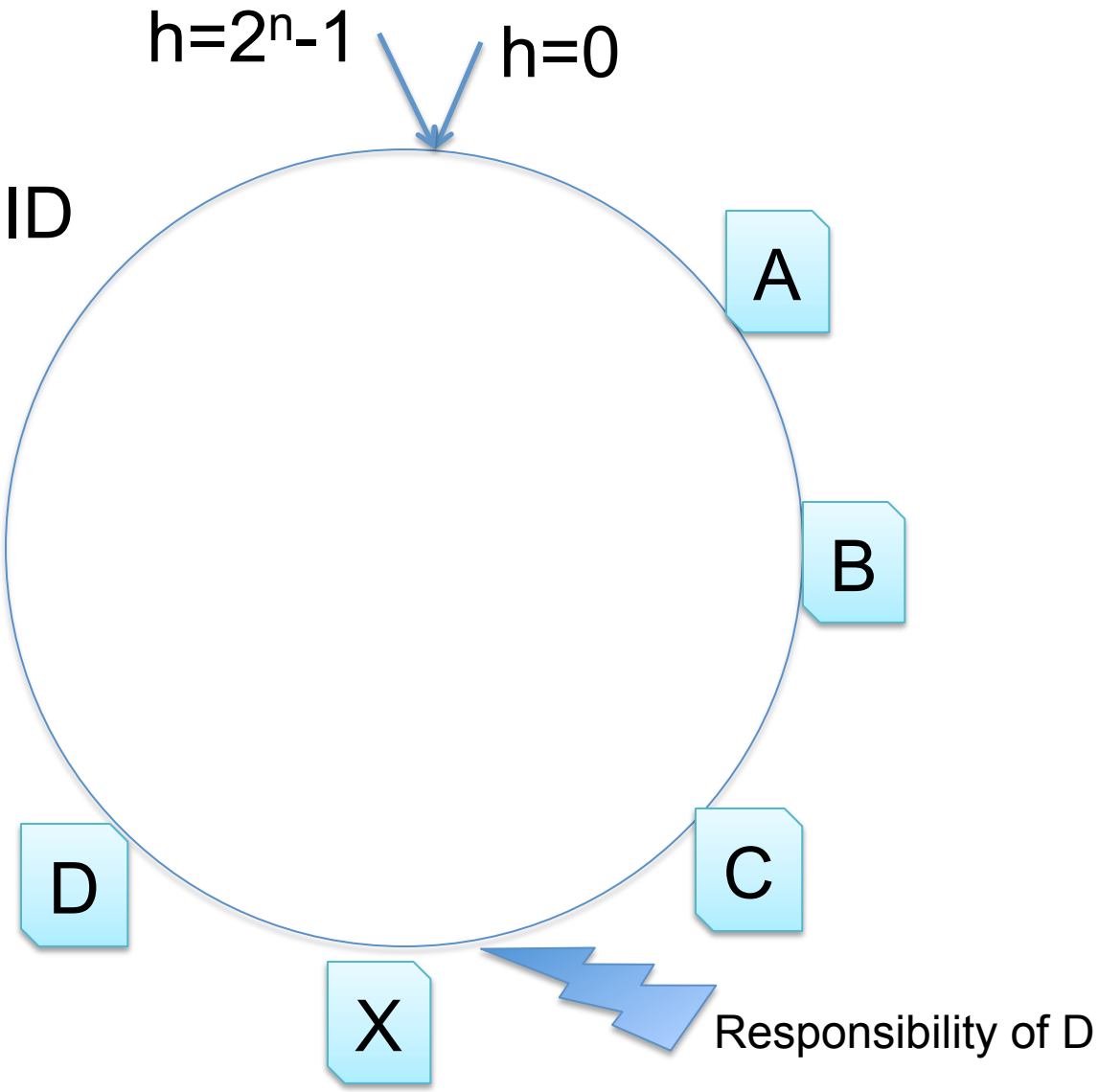
Problem 2: Joining

When X joins:
select random ID



Problem 2: Joining

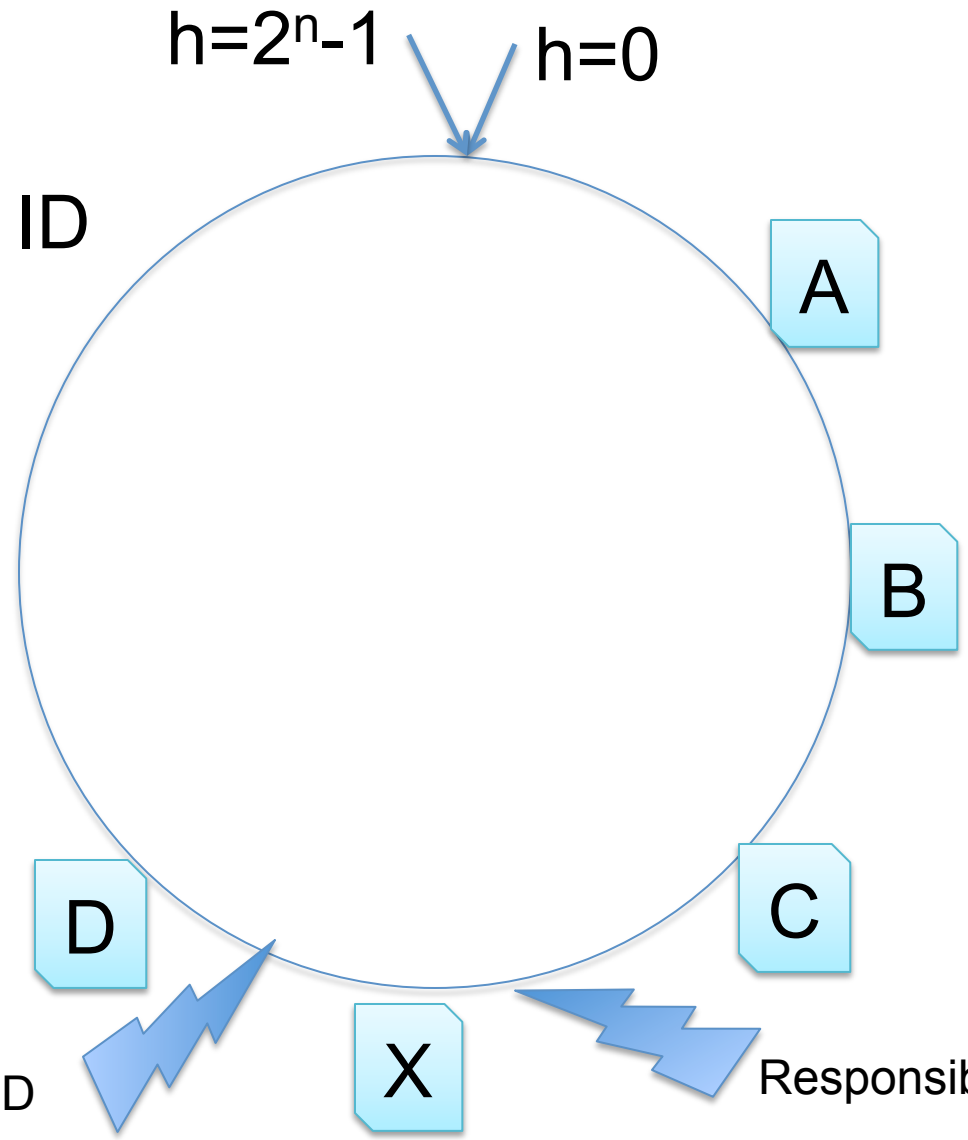
When X joins:
select random ID



Problem 2: Joining

When X joins:
select random ID

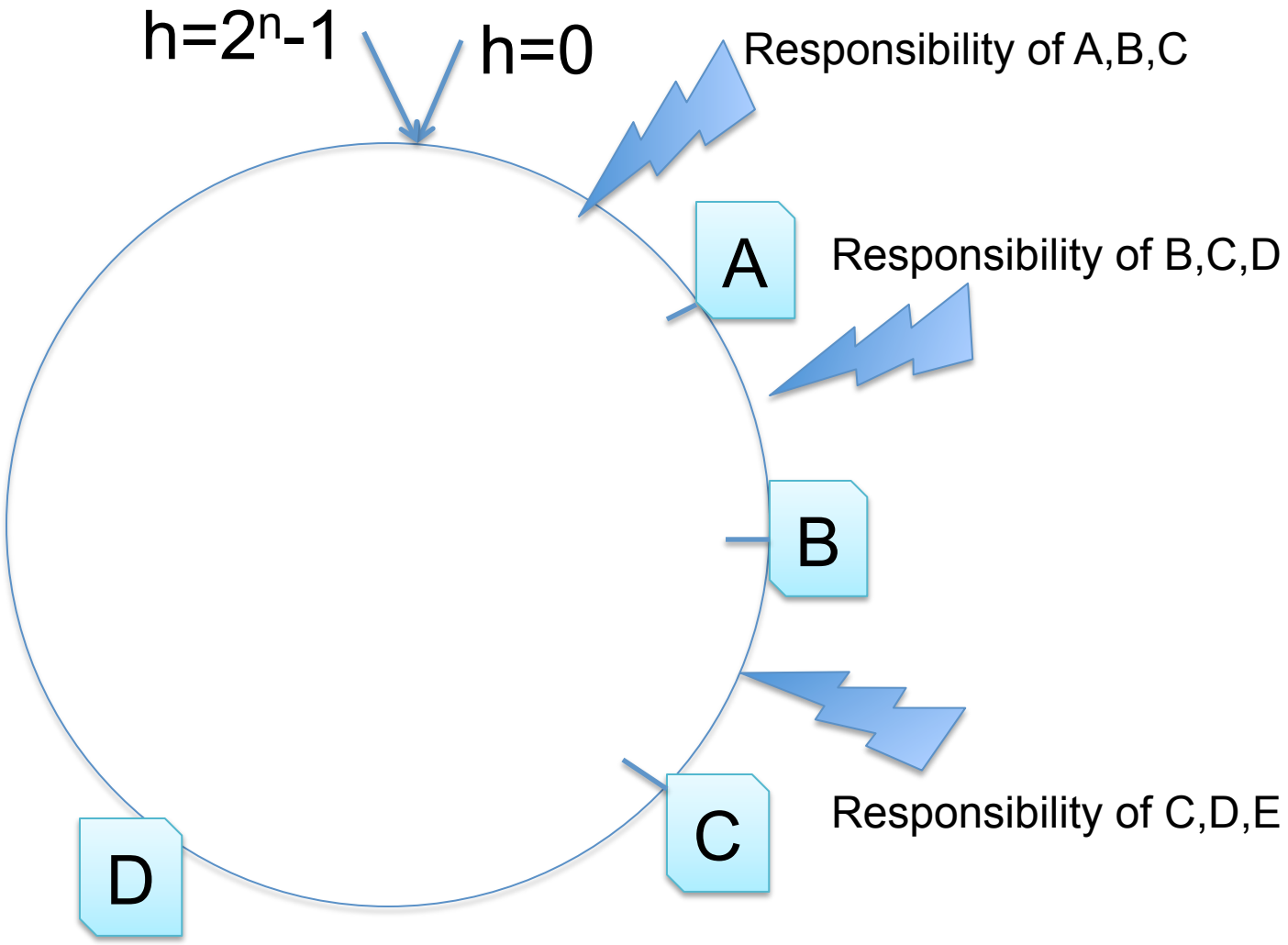
Redistribute
the load at D



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, …, 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}, …, X_t

- Vector Clocks:
  X has set of [server, timestamp] pairs
  X([s1,t1], [s2,t2],…)

Dynamo:2007

Vector Clocks

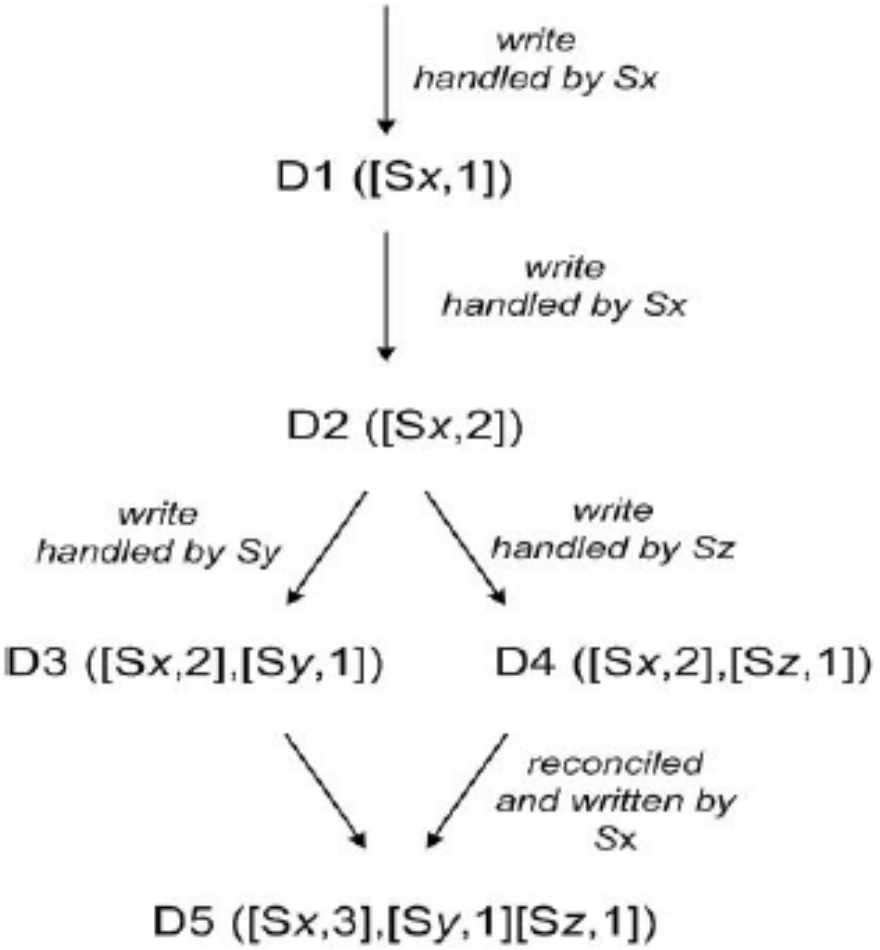


Figure 3: Version evolution of an object over time.

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)
-
-
-

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)
-
-
-

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])
-
-

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])
-

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: Meaning

- A data item D[(S1,v1),(S2,v2),…] means a value that represents version v1 for S1, version v2 for S2, etc.
- If server Si updates D, then:
  - It must increment vi, if (Si, vi) exists
  - Otherwise, it must create a new entry (Si,1)

Vector Clocks: Conflicts

- A data item D *is an ancestor* of D' if for all (S,v)∈D there exists (S,v')∈D' s.t. v ≤ v'

- Otherwise, D and D' are on parallel branches, and it means that they have a conflict that needs to be reconciled semantically

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

Brewer 2000

CAP Theorem: No Availability

- CP = Consistency + tolerance to Partitions

- Distributed databases
- Majority protocols

- Make minority partitions unavailable

Brewer 2000

CAP Theorem: No Consistency

- AP = Availability + tolerance to Partitions

- DNS
- Web caching

Brewer 2000

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*

Cattell, SIGMOD Record 2010

Overview of No-SQL systems

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

ACID v.s. BASE

- ACID = Atomicity, Consistency, Isolation, and Durability

- BASE = Basically Available, Soft state, Eventually consistent

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

1. Key-value Stores

Think “file system” more than “database”

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

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

2. Document Stores

- SimpleDB, CouchDB, MongoDB, Terrastore

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

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

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)

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

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

Cattell, SIGMOD Record 2010

Application 3

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

Extensible Record Store

Cattell, SIGMOD Record 2010

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, …, 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

Aboulnaga'2011

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

Aboulnaga'2011

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!

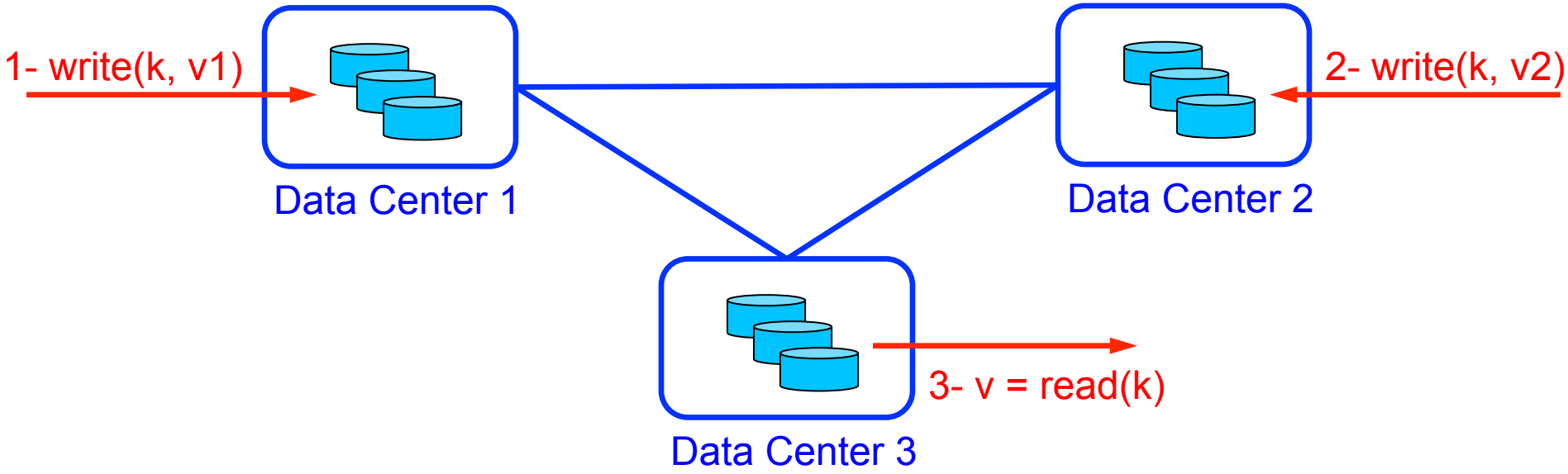
Aboulnaga'2011

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

Aboulnaga'2011

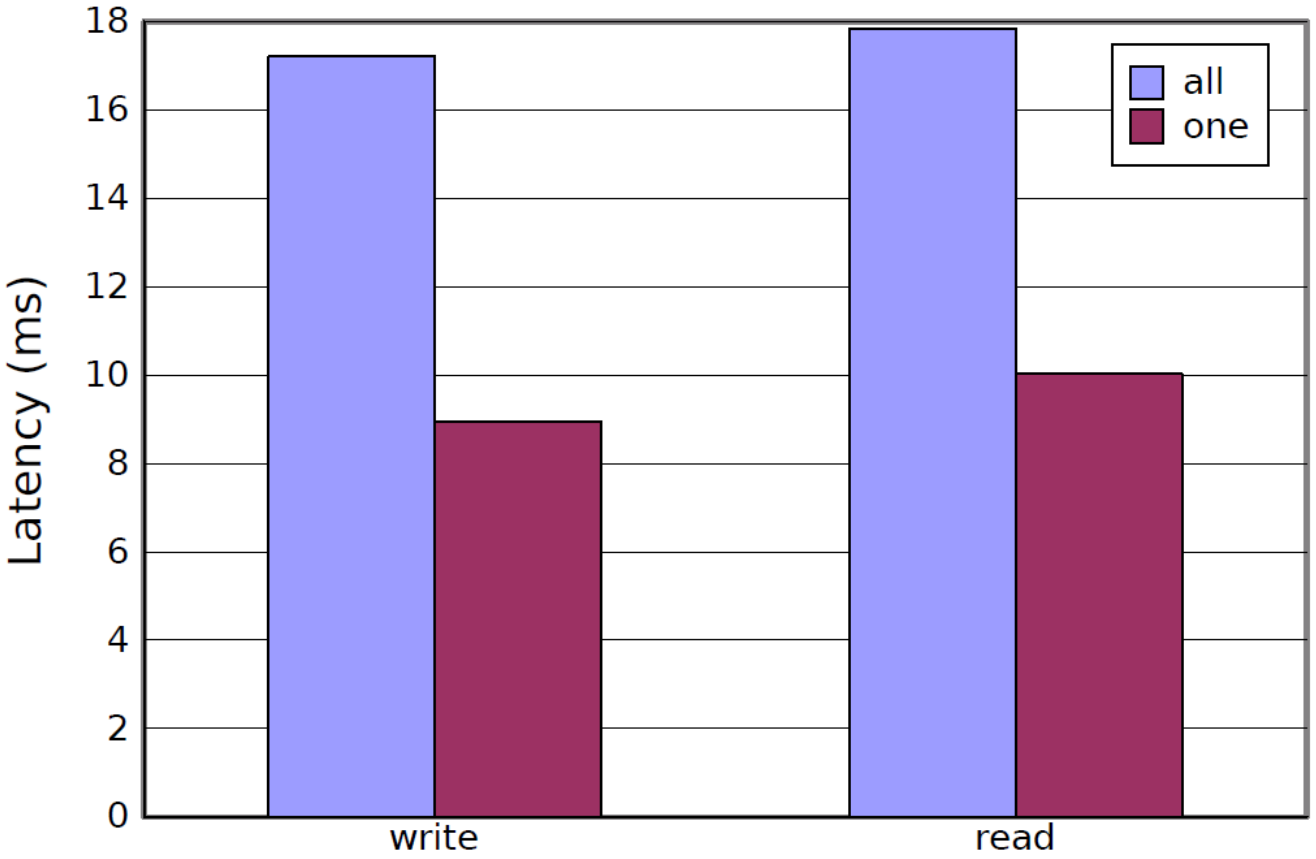
Consistency vs. Latency



- 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

Aboulnaga'2011

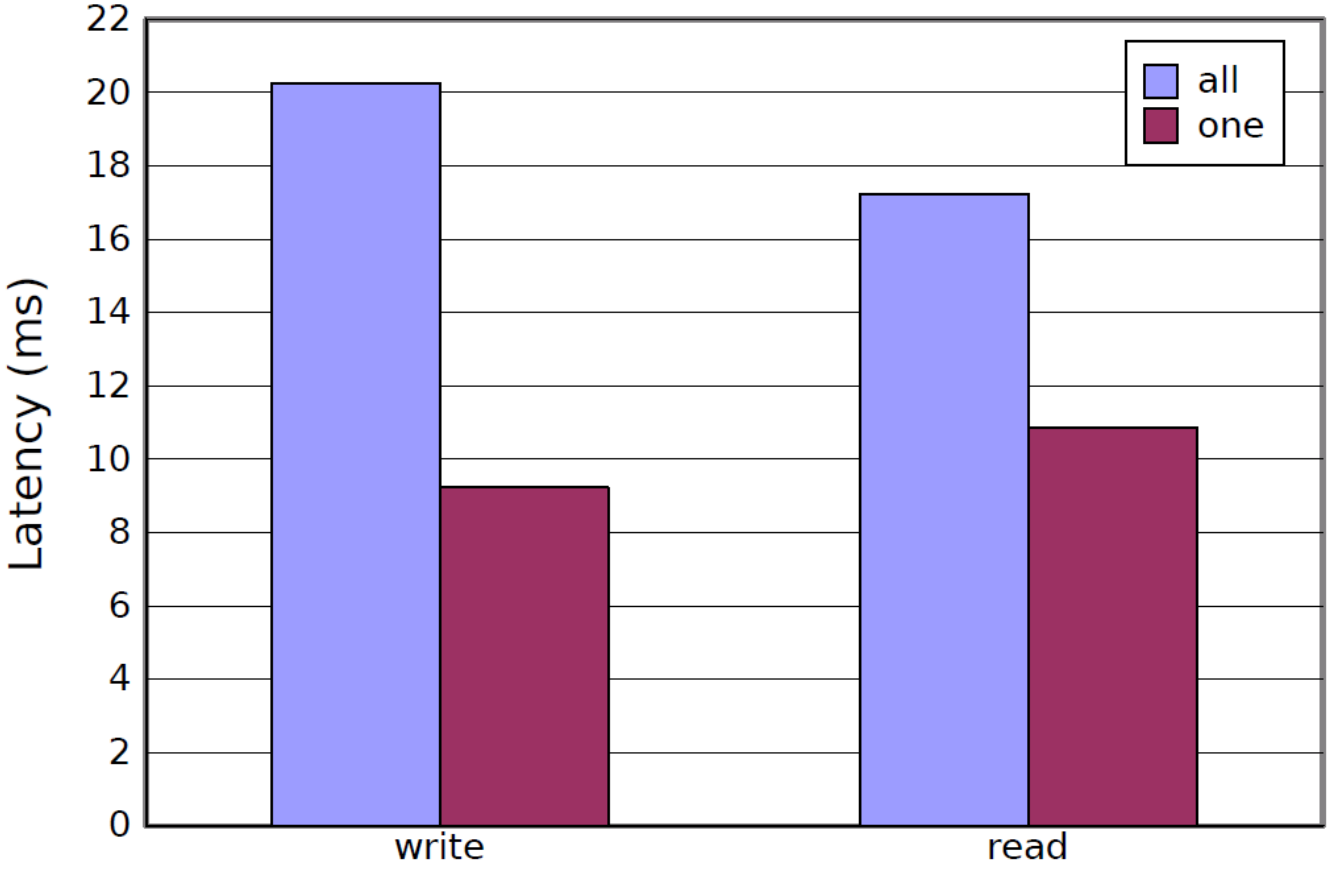
Consistency vs. Latency



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

Aboulnaga'2011

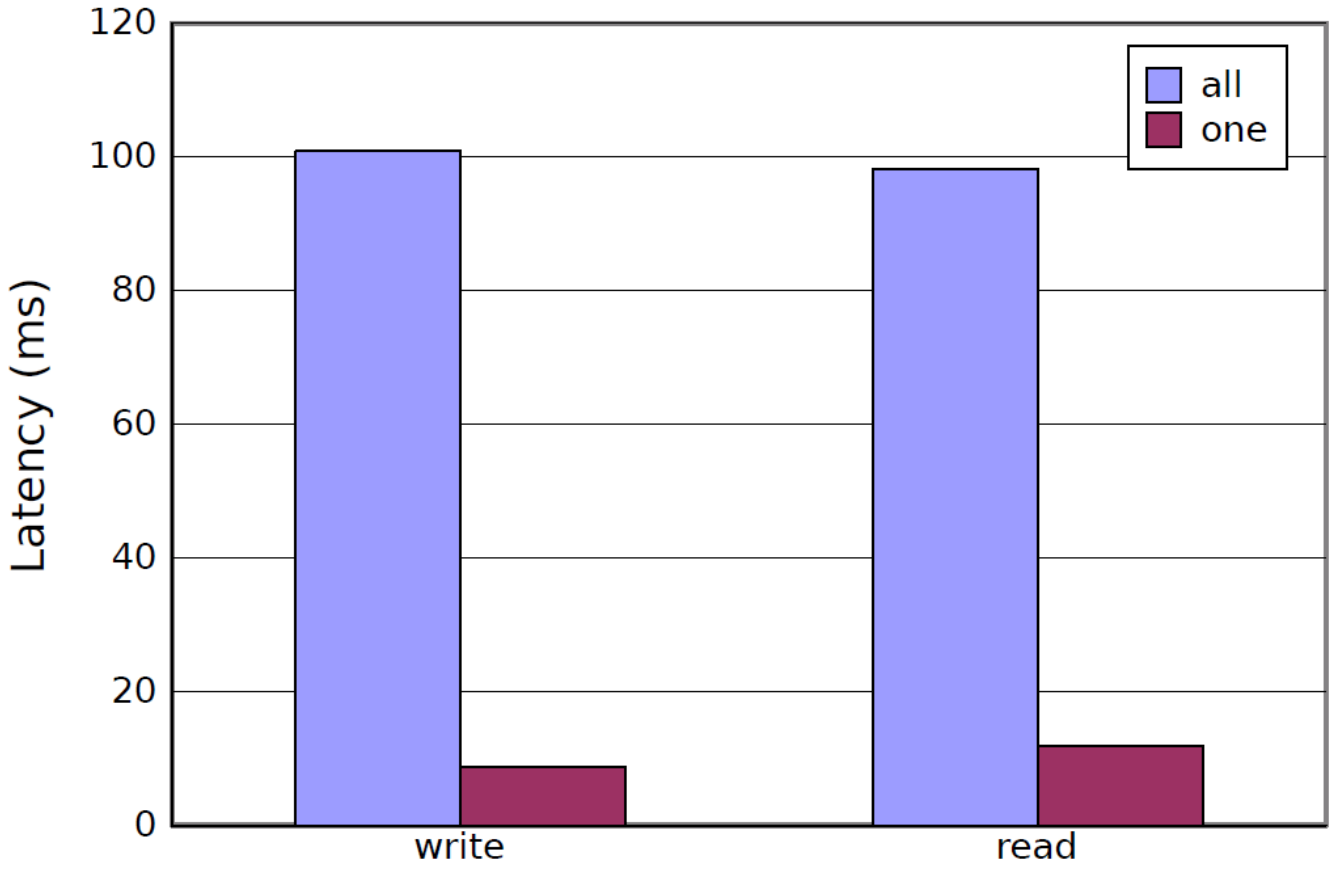
Consistency vs. Latency



Two EC2 Availability Zones
Same EC2 Geographic Region

Aboulnaga'2011

Consistency vs. Latency

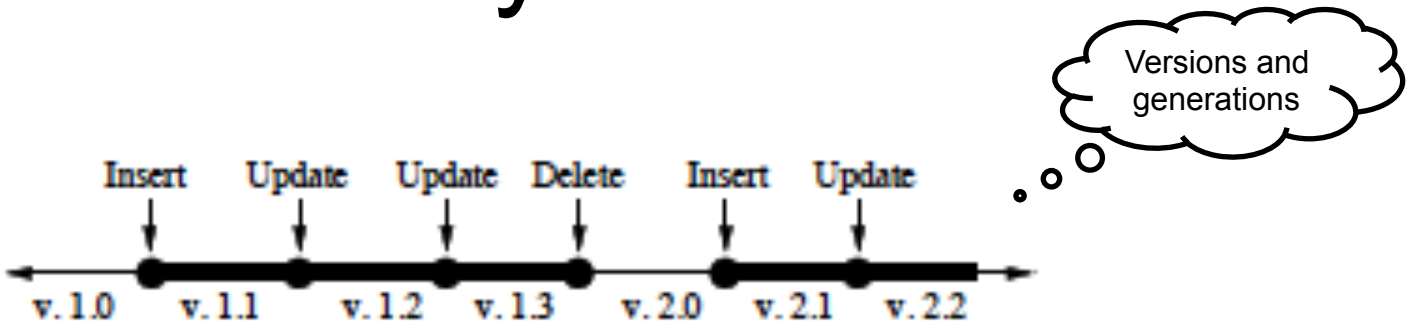


Two EC2 Regions
(US East and US West)

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

Stonebraker, CACM'2010 (blog 1)

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: abut 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

Stonebraker, CACM'2011 (blog 2)

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