

TDT4225 Very Large, Distributed Data Volumes

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Info

- Lecturer:
 - Svein Erik Bratsberg, office 209, email: sveinbra@ntnu.no
- Lectures:
 - Tuesdays 12.15-14.00 KJL5 (Kjelhuset)
 - Automical video recording using Panopto
- Exercises: Alexander Fredheim, Erling Moen, Simen Tengs
 - Time for guidance. Thursdays 11-12 KJL5 (Kjelhuset)
- Exam: 15. Des 2021. Hopefully normal university exam 50 %
Permitted examination support material: D – No written and handwritten examination support materials are permitted. A specified, simple calculator is permitted.
- Exercises:
 - 2 with evaluation (25 % each)
 - 2 compulsory.
- Partial evaluations (delvurderinger)

Curriculum (1)

- Martin Kleppmann: Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems
 - Chap 1-9. (383 pp).
- Håvard Dybvik: Evaluating the potential of LSM-trees to supersede B-trees in databases
 - Chapter 2.1 - 2.9 Survey of Storage, Indexing, and Database (31 pp)
- George Coulouris et al: Distributed Systems - Concepts and Design
 - Chapter 14 Time and global states (44 pp).

Curriculum (2)

- Ongaro/Ousterhout: In Search of an Understandable Consensus Algorithm (RAFT), USENIX 2014 (16 pp)
- Dynamo: Amazon's Highly Available Key-value Store, SOSP '07, (16 pp).
- Spanner: Google's Globally-Distributed Database, OSDI 2012, (14 pp) Video.
- Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging, SIGMOD 2018, (16 pp) Video

Lecture plan, temporary

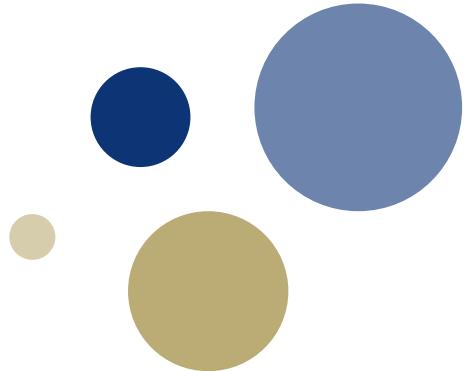
Date	Tue 12-14	Theme
24.Aug	KJL5	Intro + LSM/SSD
31.Aug	KJL5	Kleppmann. Chap 1 and 2.
7.Sep	KJL5	Kleppmann. Chap 3 and 4.
14.Sep	KJL5	Kleppmann. Chap 5 and 6.
21.Sep	KJL5	Kleppmann. Chap 7 and 8.
28.Sep	KJL5	Kleppmann. Chap 9.
5.Oct	KJL5	Coulouris. Chap 14.
12.Oct	KJL5	RAFT
19.Oct	KJL5	Dynamo
16.Oct	Video	Spanner
2.nov	Video	Dostoevsky
12.nov		???

Exercises

- Compulsory. All 4 must be approved. Done in groups of up to 3.
- Exercise 1: Theory - SSDs, LSM-trees and Kleppmann stuff (17 Sep)
- Exercise 2: Programming - MySQL (8 Oct) **(25 % eval)**
- Exercise 3: Programming - MongoDB (22 Oct) **(25 % eval)**
- Exercise 4: Theory - Kleppmann stuff, Coulouris stuff and systems (RAFT/Dynamo/Spanner/Dostoevsky) (5 Nov)



Kunnskap for en bedre verden



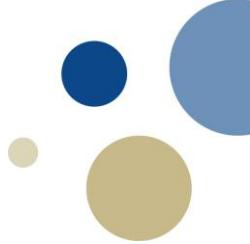
SSDs, LSM-trees and RocksDB

Håvard Dybvik

Svein Erik Bratsberg, IDI/NTNU

Contents

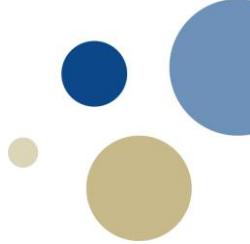
- HDDs
- SSDs
- Sequential writes
- B+-trees
- LSM-trees
- RocksDB
- MyRocks
- Write amplification
- Write stalls and write stops



HDDs

- Rotating, magnetic disks
- Have been developed since 1956
- Storing vast amounts of data at low costs
- Access time does not improve much with new disks
- 5-10 millisecs today, the first (1956) had 600 millisecs.
- Throughput (typical every day, desktop disk):
 - 160 MB/s (write)
 - 180 MB/s (read)
- Needs to be careful with layout of file system

SSDs (1)



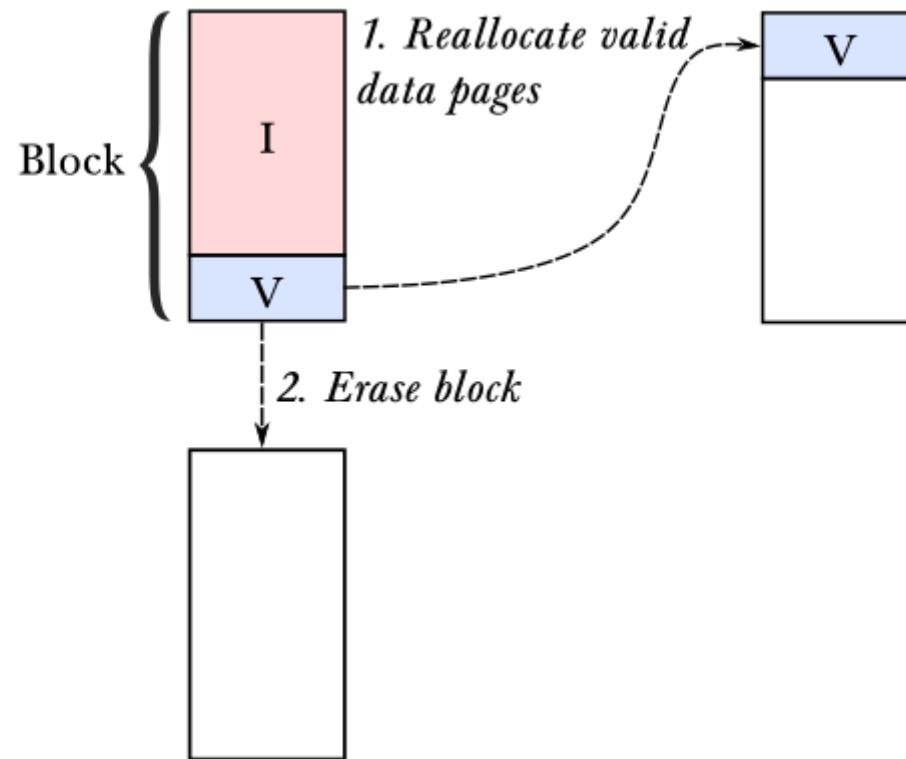
- Solid state drives – purely electronic devices
- Faster access times, lower latency, lower power consumption, completely silent operation, uniform random access speed
- NAND: Block: 64 or 128 *pages* of 2 Kbyte or 4 Kbyte
Block: 128 KB to 512 KB
- Reads and writes of *pages*
- Erase *complete blocks*
- Erase-before-write
- Throughput: Four times of HDD (at least), 550 MB/s

SSDs – wear leveling

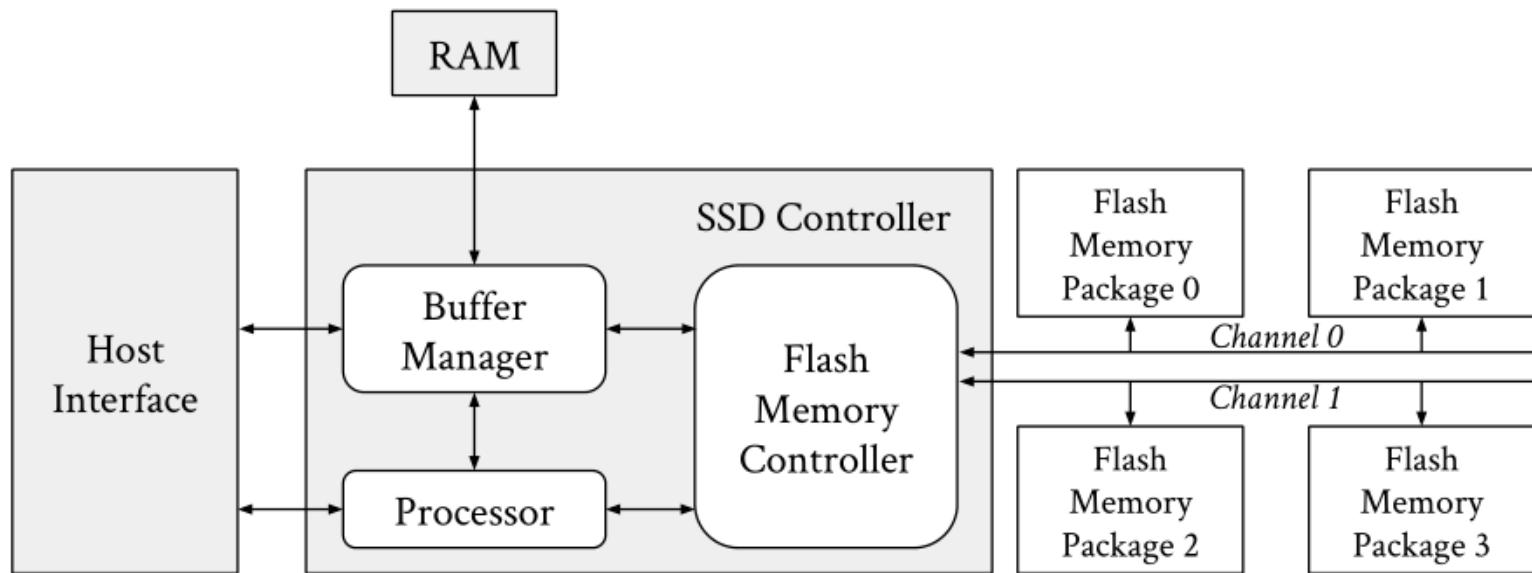
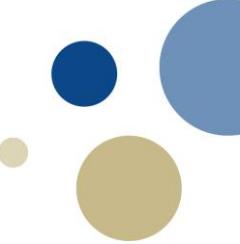


- Limited number of writes possible (wearing)
- Flash Translation Layer – firmware layer implementing wear leveling: Standardized by Intel.
 - *No wear leveling*: Fixed mapping from logical addresses to physical addresses
 - *Dynamic wear leveling*: At updates, the old block is marked as invalid and the block is relocated.
 - *Static wear leveling*: Also moves static blocks periodically. All parts of the disk will be worn out eventually.
- Garbage collection is important in SSDs. Done in units of blocks.
- Host based FTL (computer) and array-based FTL (disk)

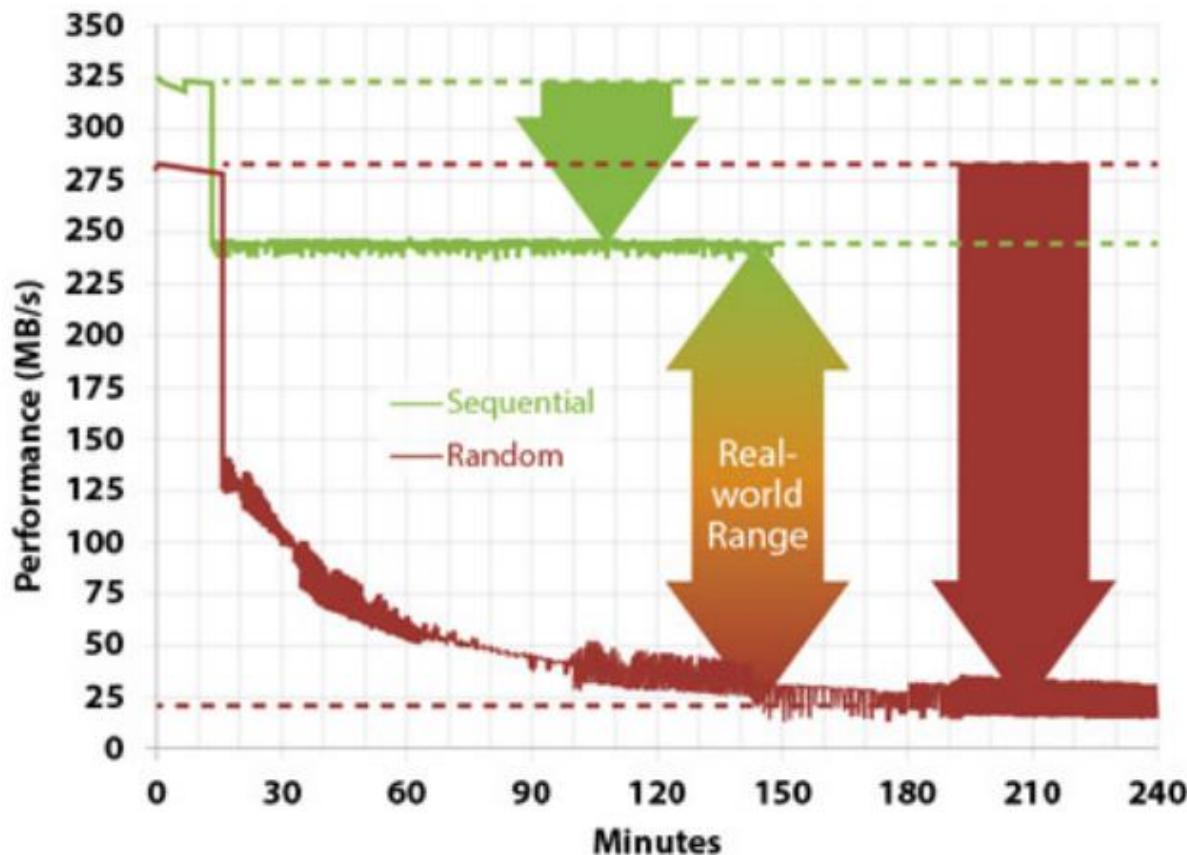
SSDs – garbage collection



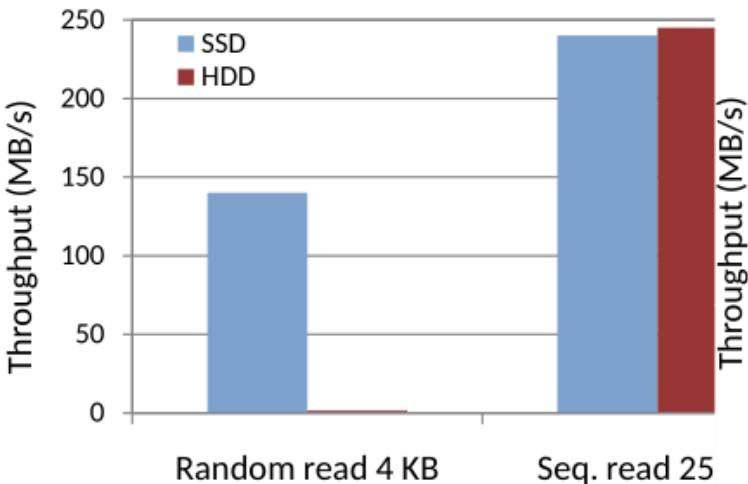
SSDs – parallel I/O



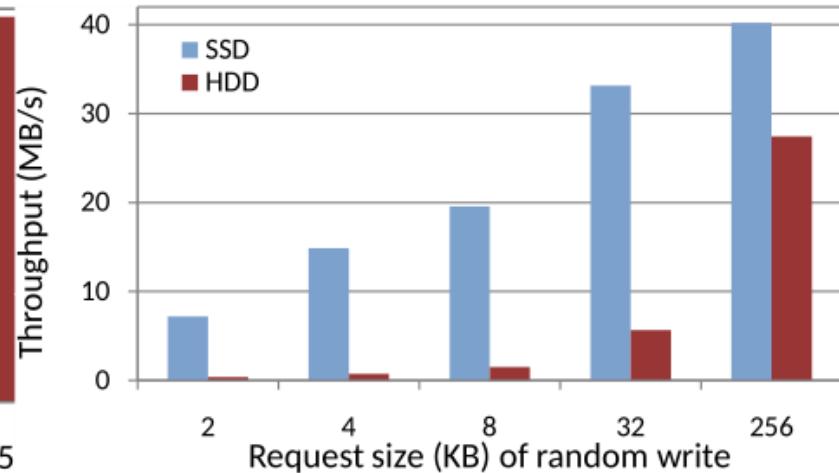
Sequential writes (1)



Sequential writes (2)



(a) Reads.



(b) Random writes.

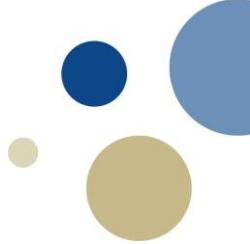
Sequential writes (3)

- One of the essential keys to great write performance is to organize data in such a way that each write request distributes the overhead of the write over multiple inserts batched together – large blocks are also good
- *Over-provisioning:* When the GC has too much to do, writes happen more often than erase operations -- The SSD has a reserved area where the garbage collector could put writes at high peaks periods. 7 – 28 % of disk.

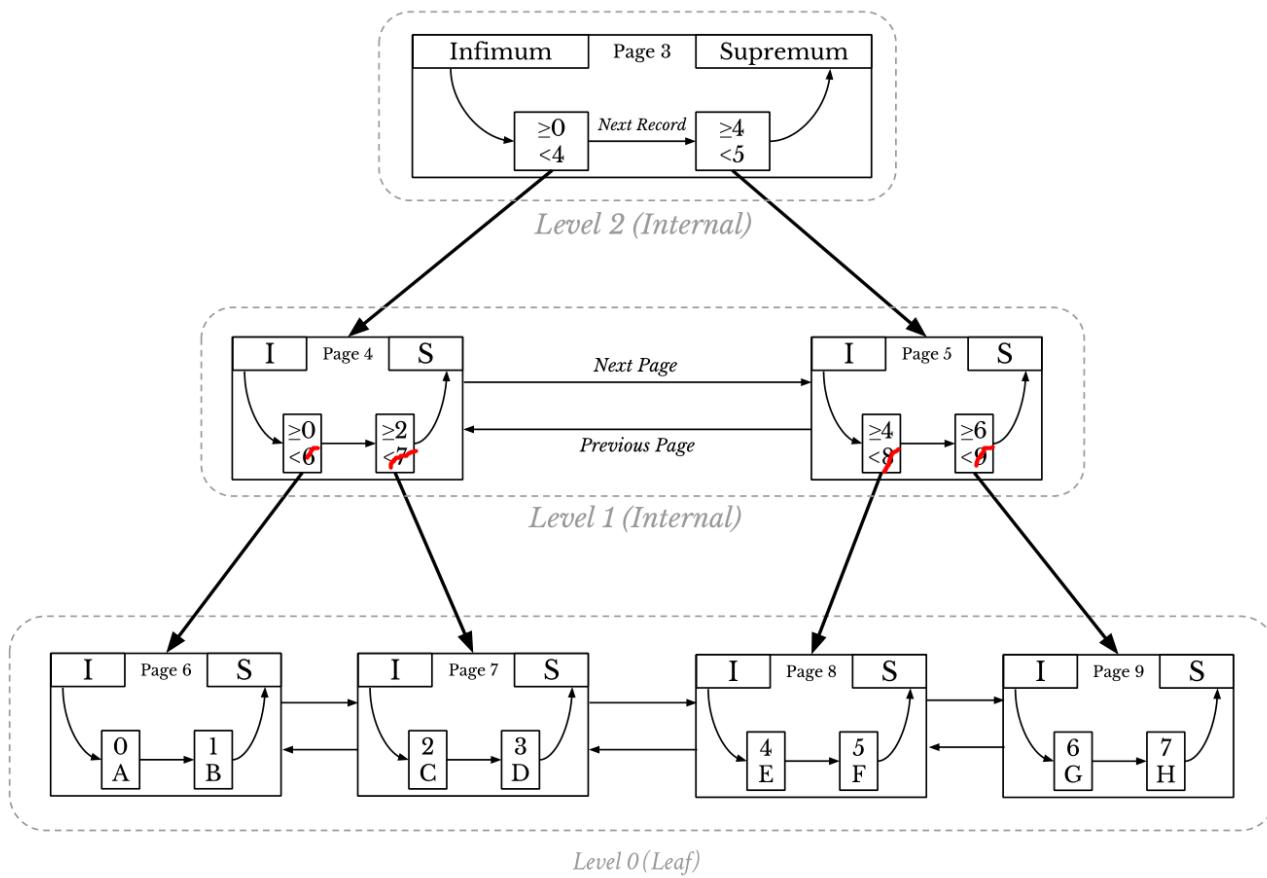
Writes/reads measured (old exercise)

	HDD/w	HDD/r	SSD/w	SSD/r
1GB	1976 MB/s	3835 MB/s		
2GB	2101 MB/s	3954 MB/s		
4GB	1766 MB/s	202 MB/s		
8GB	386 MB/s	157 MB/s		
16GB	221 MB/s	162 MB/s		
32GB	184 MB/s	163 MB/s	354 MB/s	463 MB/s

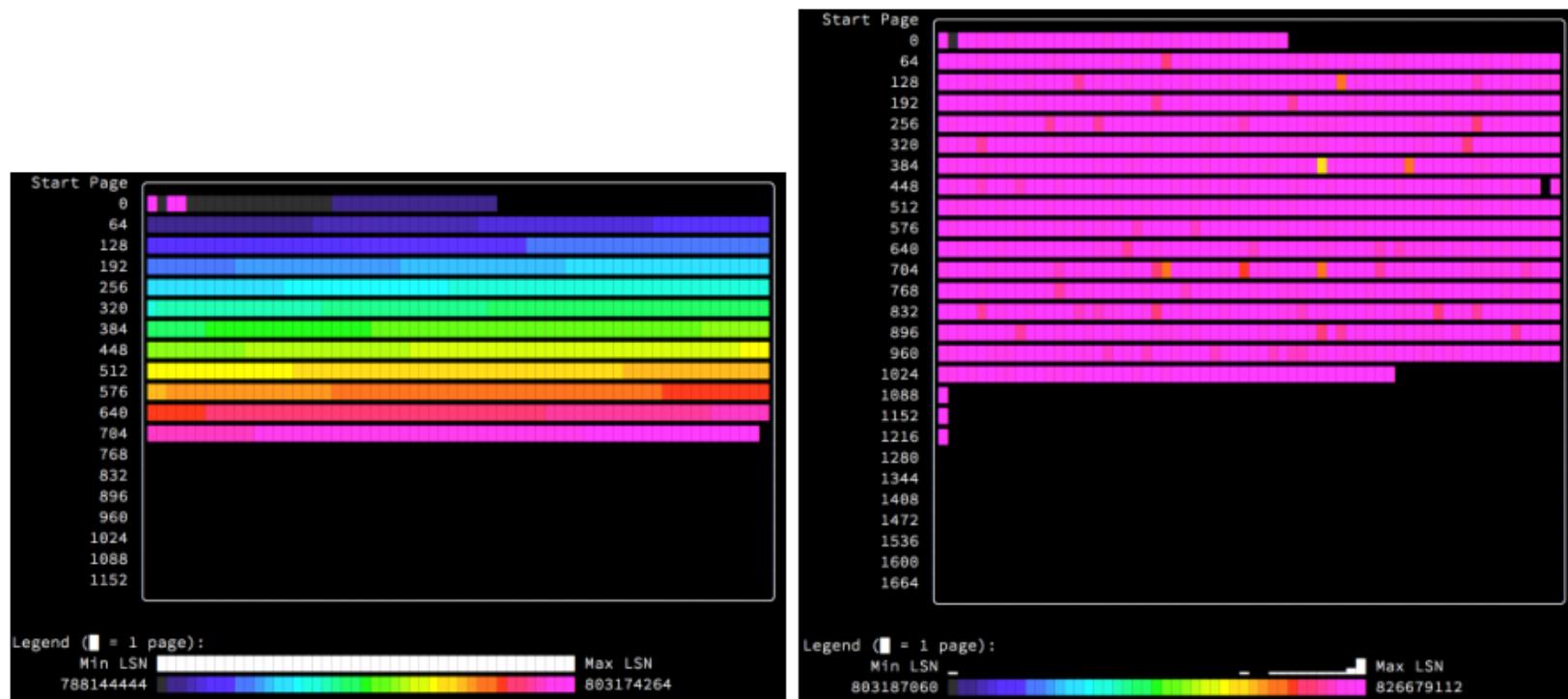
B+-trees (1)



B⁺-Tree Structure of InnoDB

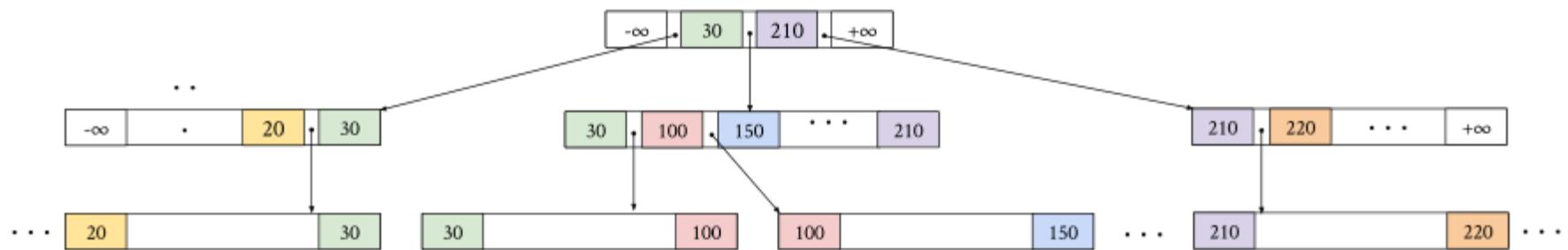


B+-trees (2) Freshness of PageLSNs

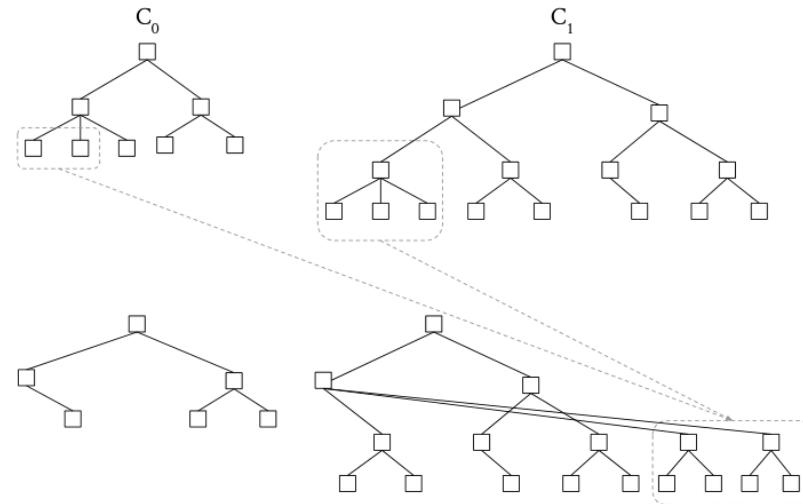
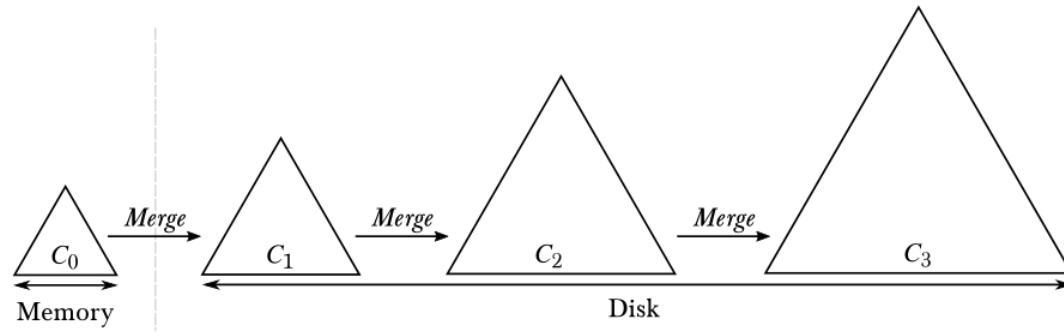


Variants of B+-trees

- *FB-tree*: Variable sized blocks simulating FTL's GC algorithm.
- *Copy-on-write B-tree*: Copy a block to a new location at updates. E.g. LMDB. BTRFS (file system on Linux).
- *Write-optimized B-tree*: No sideway pointers, index records contain lowKey and highKey of the leaf blocks. Prevents multiple writes due to pointers when moving a block.



LSM-trees



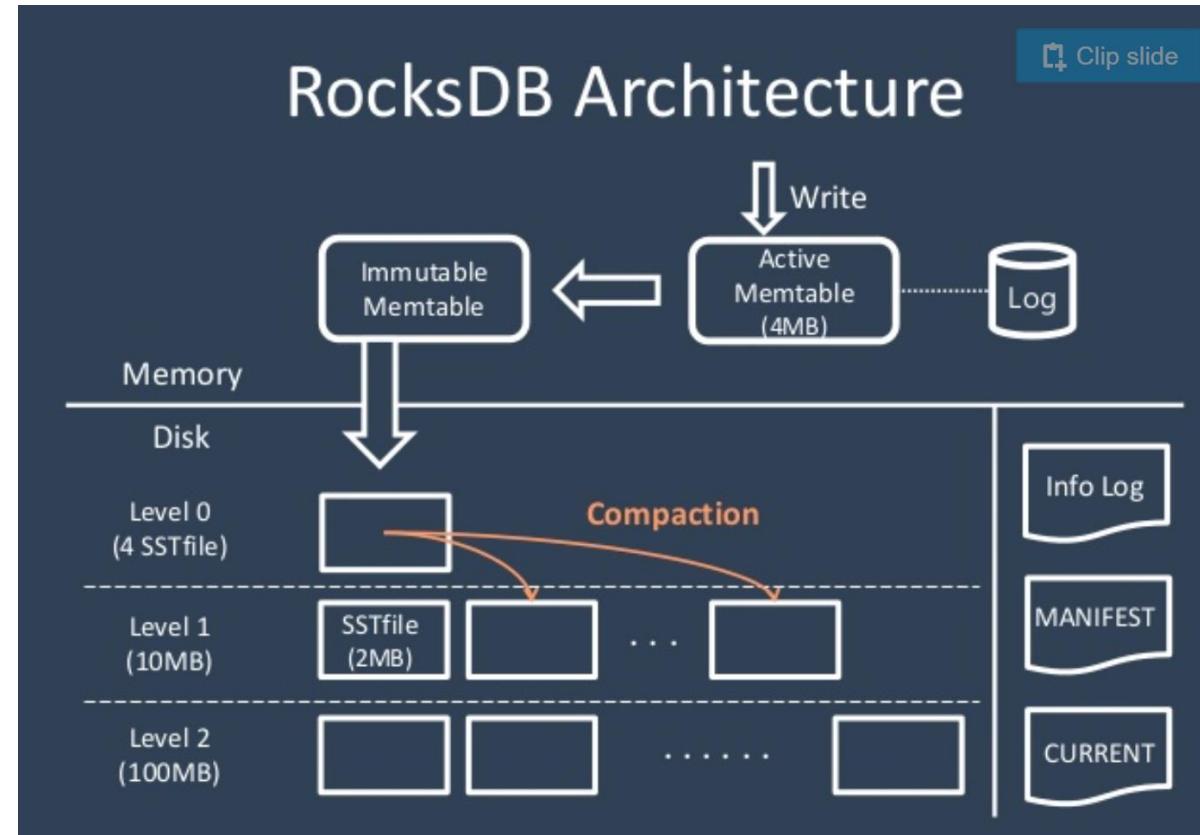
RocksDB

- Builds on Google's LevelDB
- Multithreaded compaction
- Multithreaded insertion into MemTable
- Extensive control over Bloom filters
- SSD support
- 10 x improved write performance compared to LevelDB



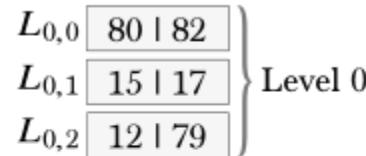
RocksDB (2)

- SSTable (sorted key ranges)
- Memtable
- Write Ahead Log

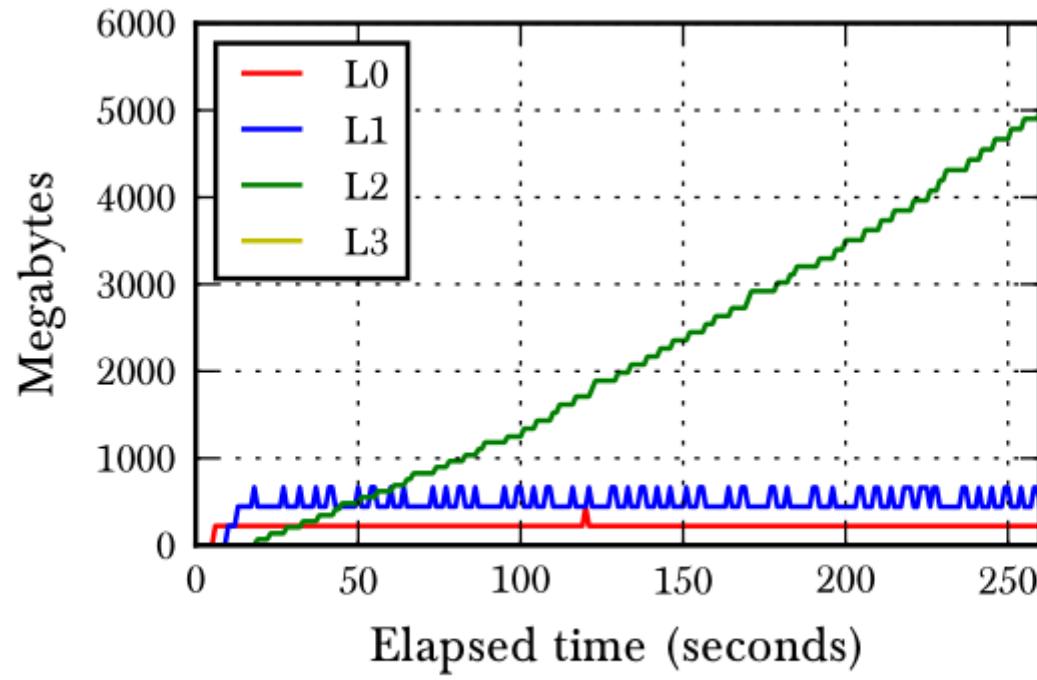


Leveled Compaction

- Original compaction style in LevelDB
- SSTable files stored in multiple levels
- Multiple overlapping SSTables at level 0 to get fast write of MemTables



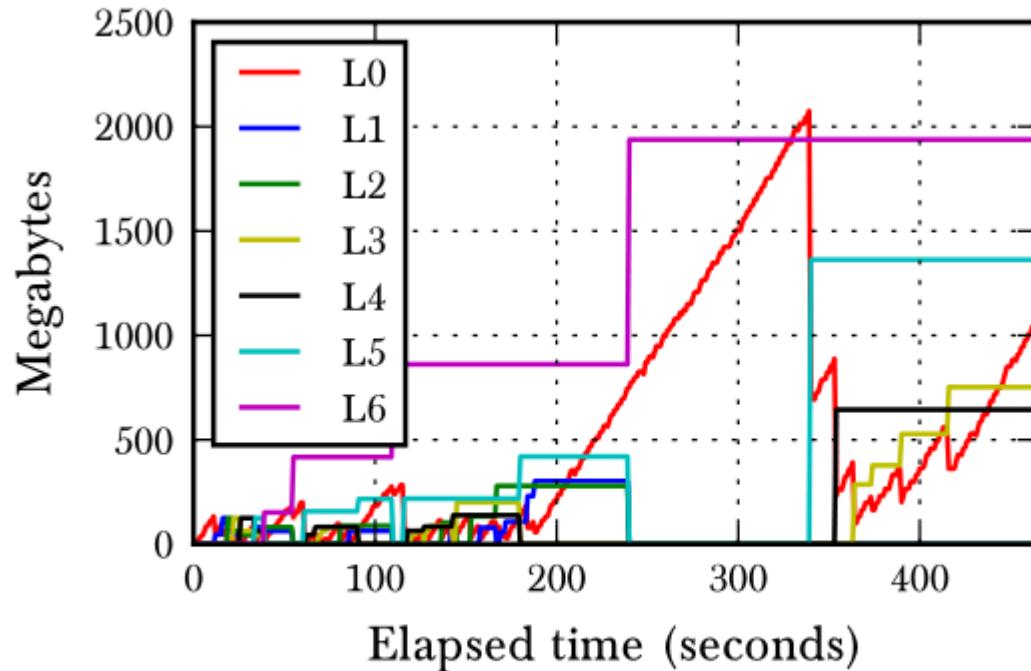
Leveled Compaction (2)



(a) Leveled compaction.

Universal Compaction

- SSTables overlap in key range, but not in time
- Merging SSTables to cover bigger time ranges



(b) Universal compaction.

Bloom filter

- Simple data structure to quickly check if a key *may* exist in a dataset
- Bitmap of m bits and k unique hash functions
- Either used on each block or the complete SSTables

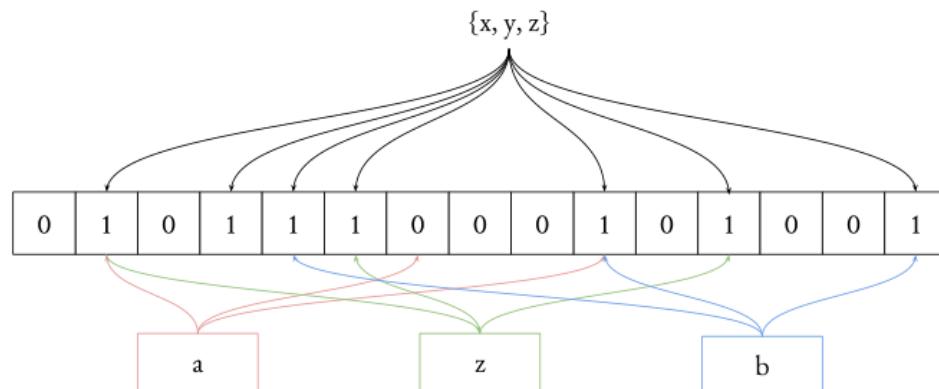
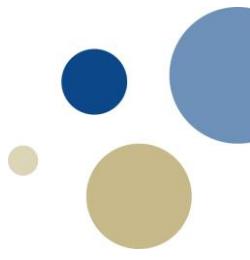


Figure 2.25: Example of a Bloom filter with the dataset x, y, z and $m = 15$ bits. There are $k = 3$ different hash functions.

MyRocks

- Facebook's integration of RocksDB as a storage engine with MySQL
- Created due to serious space and write amplification of MySQL's InnoDB
- MariaDB (MySQL competitor) is also integrating RocksDB as a storage engine

Write amplification



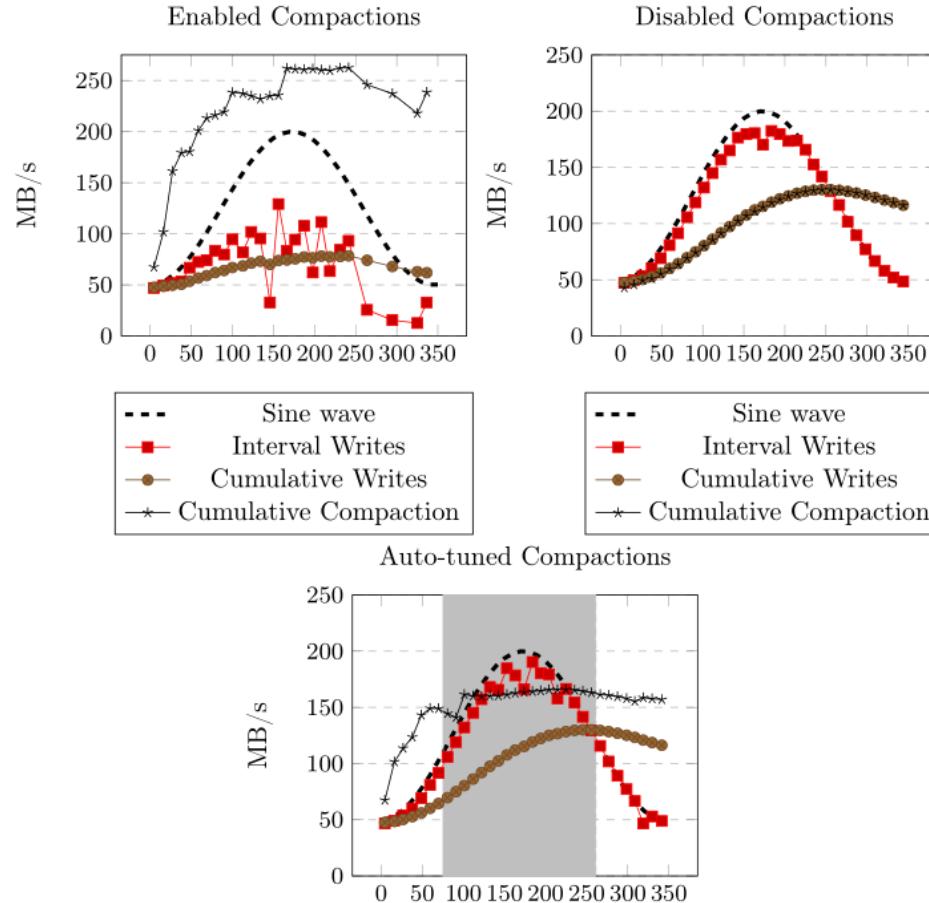
- #bytes written to api / #bytes written to database
- B-trees: write amp less with big records
- LSM-trees: less write amp due to big write chunks, but compaction require more writes
- LSM-trees: write amp independent on record size
- May be measured by insert throughput. RocksDB performs very well

Write stalls and write stops



- Write stalls: Slow the speed of inserts to cope with compaction
- Write stops: Stop inserts.
- RocksDB can configure parameters to control this:
 - Number of MemTables
 - Number of SSTables at level 0
 - Number of bytes awaiting compactions
- Hans-Wilhelm Kirsch Warlo developed an auto-tuner for compactations in RocksDB during his master thesis: Turns off compactations during high insert loads (spring 2018).

Warlo's auto-tuner



TDT4225

Chapter 1 – Reliable, Scalable and Maintainable Applications

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Technology and tasks

- *Data-intensive* applications as opposed to CPU-intensive
- *Databases*: Store data to be found later
- *Caches*: Remember the result of expensive operations to be read later
- *Search indexes*: Search data by a keyword
- *Stream processing*: Send messages asynchronously to another process
- *Batch processing*: Periodically crunch a large amount of accumulated data

Architecture

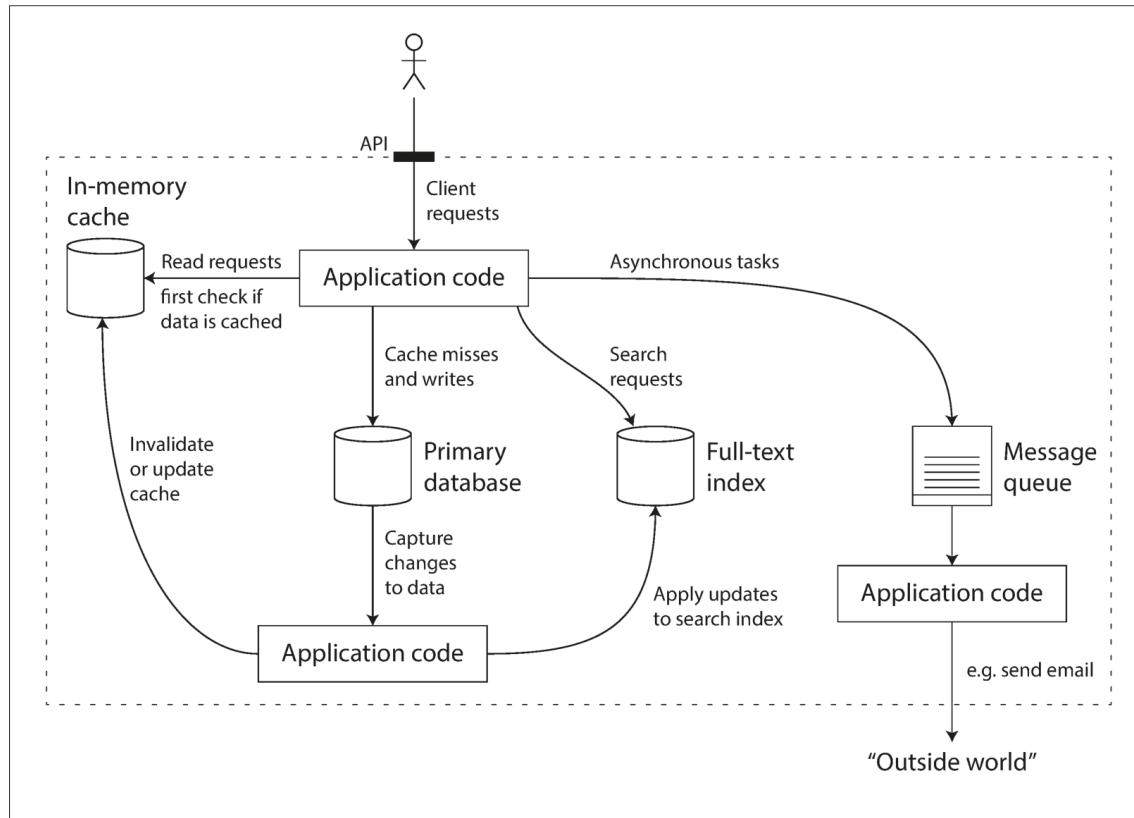


Figure 1-1. One possible architecture for a data system that combines several components.

Three desired properties

- *Reliability*: The system should work correctly even in the face of errors (software / hardware / human)
- *Scalability*: As the system grows (data volume/traffic volume/complexity), it should be dealt with
- *Maintainability*: Different people should be able to maintain and adapt the system over time

Reliability

- *Faults* – the things that can go wrong
- *Fault-tolerant* or *resilient*: Can tolerate certain types of faults
- *Failure*: When the whole system providing service fails
- *Exercise fault tolerance*: Randomly killing individual processes
- Prevention vs. cure of faults.
- Recovery-oriented computing (ROC) vs. replication.

Hardware faults

- Hard disks crash, faulty RAM, power outage, etc
- Hard disks MTTF: 10-50 years. Cluster of 10 000 disks, one disk dies every day
- High availability: Redundancy vs recovery?
- Systems that can tolerate loss of entire machine, as an alternative to redundant hardware. Clusters.

Software errors

- When all computers run the same software, errors may be correlated.
- N-version programming: Different software on each computer?
- Software faults may be dormant for long times, and may be triggered by a special situation?
- Systematic kill and restart may be a solution

Human errors

- Configuration errors by humans the biggest cause of outages
- Hardware faults 10-25 %
- Well-designed APIs and adm. interfaces
- Sandboxing for training using real data without destroying anything
- Testing: Whole system testing, unit testing
- Monitoring performance and errors
- Training and management of systems
- Autonomic computing is appearing slowly

Scalability

- Ability to cope with increased load, users and data
- Load parameters: Requests per second, reads/writes, #active users, etc
- Example twitter (2012)
 - (post) tweet: 12K requests/sec (peak)
 - view tweets: 300K requests/sec
- 1:

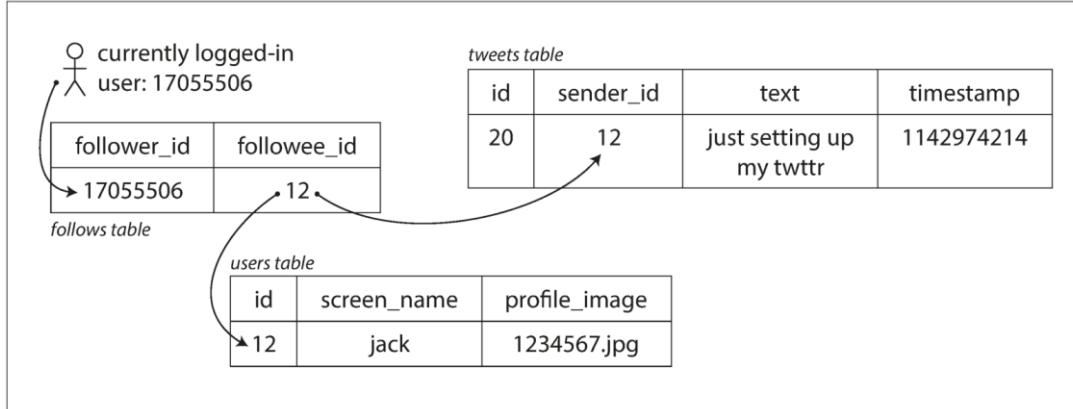


Figure 1-2. Simple relational schema for implementing a Twitter home timeline.

Scalability example - twitter

- 2:

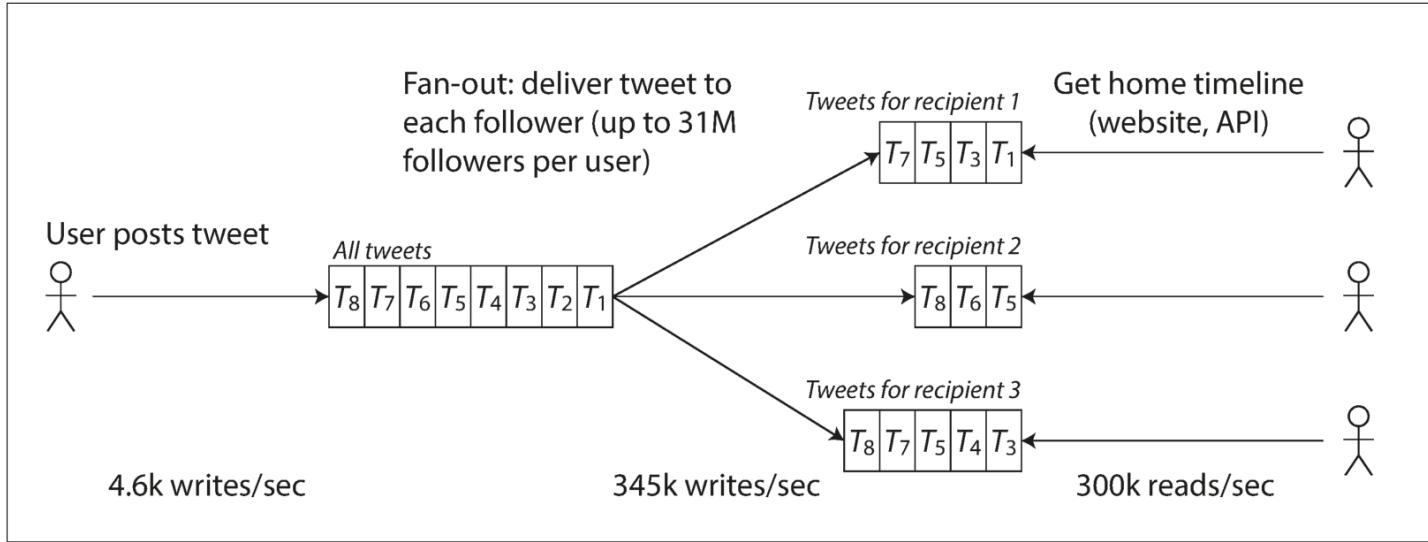


Figure 1-3. Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012 [16].

- Hybrid approach used. Tweets from celebrities are handled separately

Performance

- Batch systems: Throughput
- Online systems: Response time
- Distributions: Percentiles vs. average

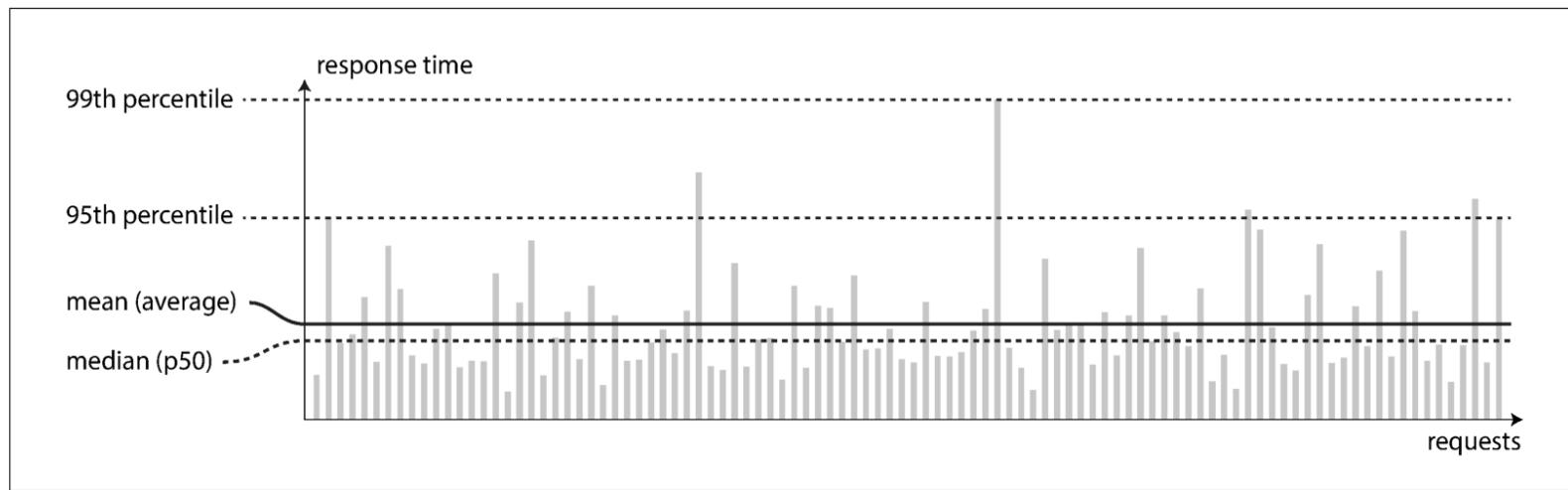


Figure 1-4. Illustrating mean and percentiles: response times for a sample of 100 requests to a service.

Performance and SLA

- SLA – Service level agreement
- Amazon's SLA measured in the 99.9 percentile (Dynamo paper, 2007)
- Hard due to events outside your control

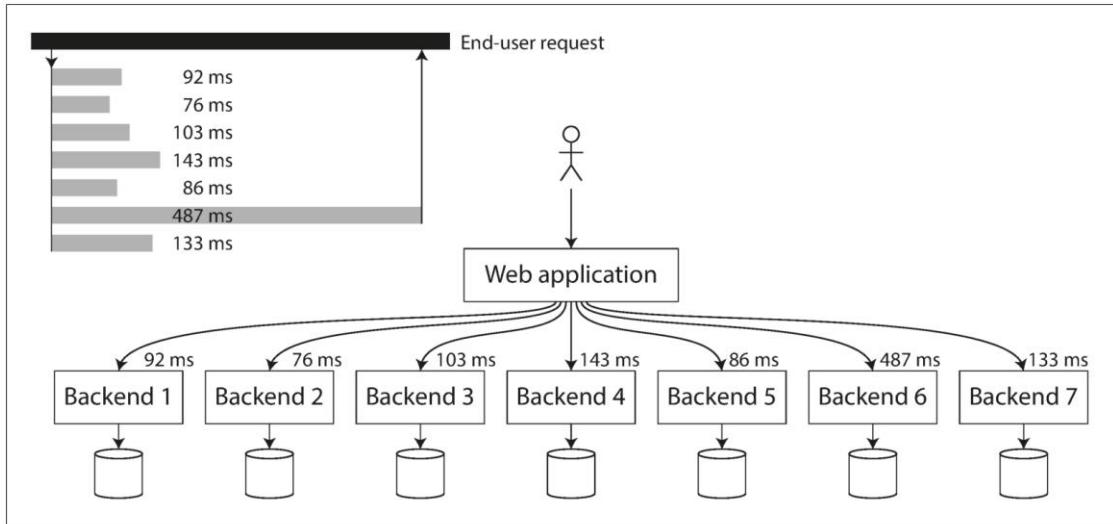


Figure 1-5. When several backend calls are needed to serve a request, it takes just a single slow backend request to slow down the entire end-user request.

Coping with load

- Scaling up (vertical scaling): More powerful machines
- Scaling out (horizontal scaling): More machines to do the work
- Stateless services: Easy to scale out
- Stateful services (e.g. databases): Complex to scale out.
- Sharding/partitioning – resharding/repartitioning

Maintainability

- Major cost of software is in ongoing maintenance, not in initial development
- Operability: Making life easy for operations
- Simplicity: Managing complexity
- Evolvability: Making change easy

TDT4225

Chapter 2 – Data Models and Query Languages

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Relational model vs. document model

- SQL is based on Ted Codd's relational model (1970)
- Relations and tuples and the relational model
- Tables, columns and rows in SQL
- Business data processing, transaction processing, and batch processing
- The big idea with SQL was to hide implementation detail
- Older alternatives: hierarchical model, network model
- Alternative models: object-oriented databases, XML databases

NoSQL

- Not Only SQL
 - Key/value stores
 - Document-oriented databases (JSON)
 - Graph databases
 - Extended relational databases
- Performance for simple operations
- Scalability (sharding)
- Free and open source
- Specialized query operations
- Frustration with SQL model. More dynamic and expressive model.

The object-relational mismatch

- Mismatch between application code and database model (objects vs rows)
- Object-Relational mapping (ORM, ActiveRecord and Hibernate) try to hide the difference
- Example LinkedIn resume (3 solutions in SQL):
 - Normalized: position, education, and contact information, using foreign keys
 - Structured datatypes/XML data/JSON inside rows. Querying and indexing these.
 - JSON/XML *documents* containing Position, Education and Contact Information.

Tables and foreign keys

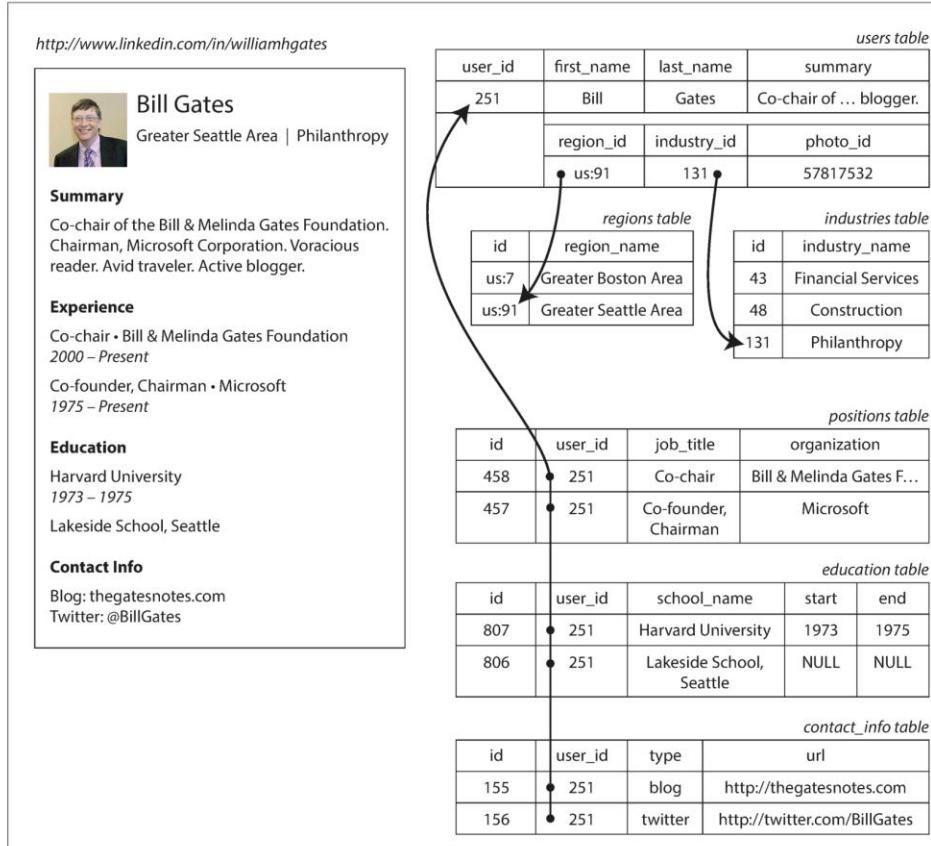


Figure 2-1. Representing a LinkedIn profile using a relational schema. Photo of Bill Gates courtesy of Wikimedia Commons, Ricardo Stuckert, Agência Brasil.

JSON

Example 2-1. Representing a LinkedIn profile as a JSON document

```
{  
    "user_id": 251,  
    "first_name": "Bill",  
    "last_name": "Gates",  
    "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",  
    "region_id": "us:91",  
    "industry_id": 131,  
    "photo_url": "/p/7/000/253/05b/308dd6e.jpg",  
  
    "positions": [  
        {"job_title": "Co-chair", "organization": "Bill & Melinda Gates Foundation"},  
        {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}  
    ],  
    "education": [  
        {"school_name": "Harvard University", "start": 1973, "end": 1975},  
        {"school_name": "Lakeside School, Seattle", "start": null, "end": null}  
    ],  
    "contact_info": {  
        "blog": "http://thegatesnotes.com",  
        "twitter": "http://twitter.com/BillGates"  
    }  
}
```

JSON viewed as trees

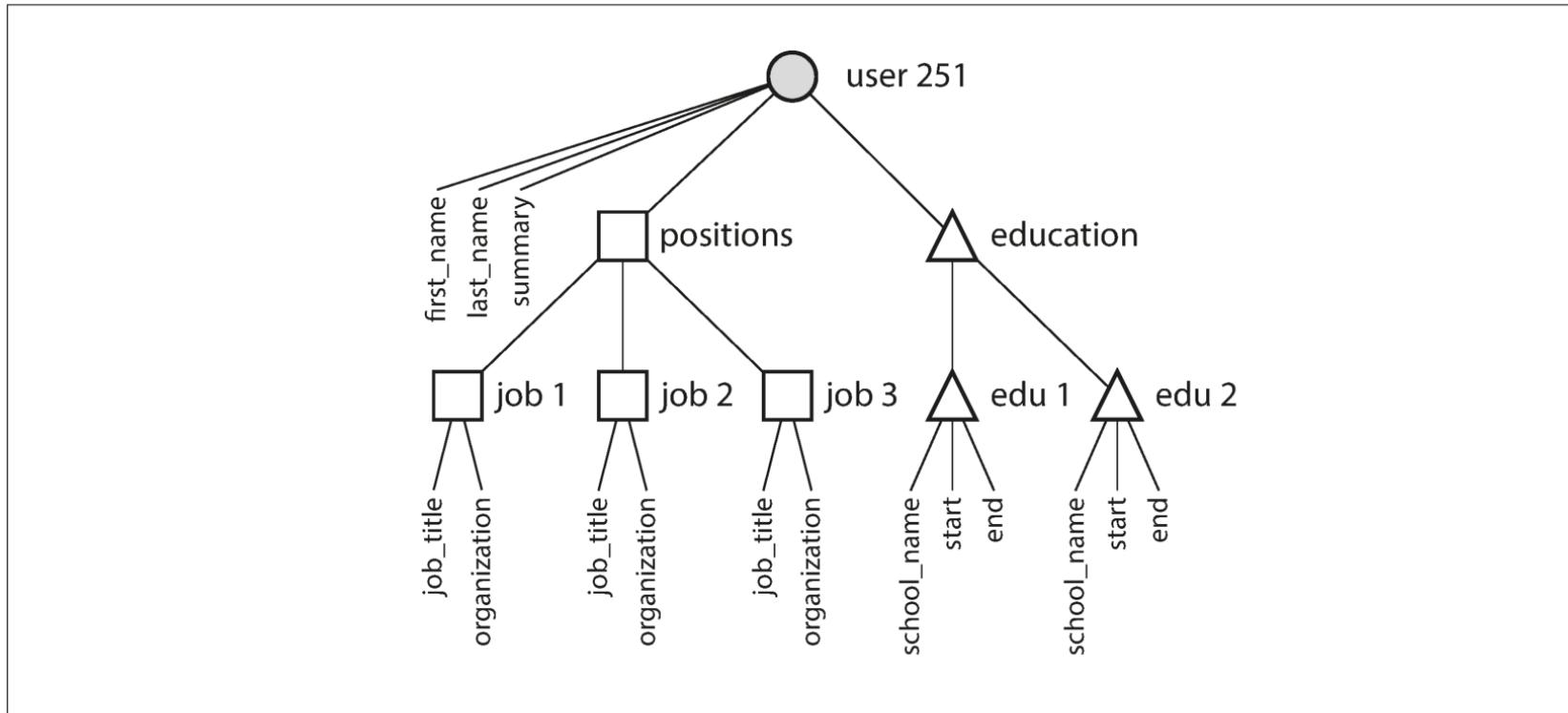


Figure 2-2. One-to-many relationships forming a tree structure.

Many-to-one and many-to-many relationships

- Standardized lists of geo regions, industries, etc.
- Use IDs
 - Consistent style and spelling
 - Avoiding ambiguity
 - Ease of updating
 - Localization support (when translating between languages)
 - Better search
- IDs have no meaning to humans and don't need to change
- Relationships supported by joins in SQL databases
- Weakly supported in document databases
(MongoDB/CouchDB)

Moving Organization to separate entity

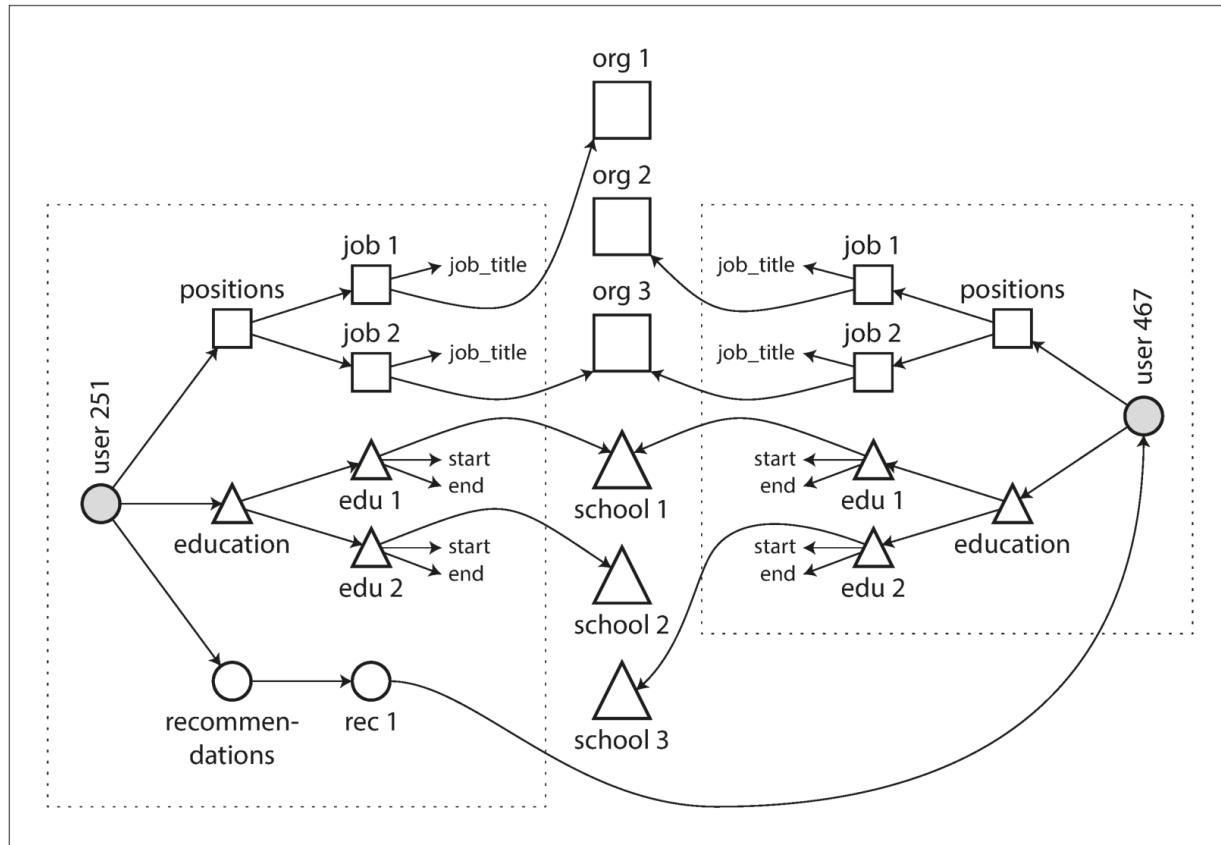


Figure 2-4. Extending résumés with many-to-many relationships.

Hierarchical vs. codasyl vs. SQL

- IBM's IMS system was hierarchical and had the same problems as the document model
- CODASYL, the network model, allowed pointers between records. Access paths and cursors iterated over collections and pointers.
- SQL uses a query compiler and optimizer to automatically decide how to execute a query.
- SQL allows arbitrary relationships using joins

Document model vs. SQL

- Schema flexibility in document model
 - Schema-on-read, not schemaless
 - May support certain schema changes easily

```
if (user && user.name && !user.first_name) {  
    // Documents written before Dec 8, 2013 don't have first_name  
    user.first_name = user.name.split(" ")[0];  
}  
  
ALTER TABLE users ADD COLUMN first_name text;  
UPDATE users SET first_name = split_part(name, ' ', 1);      -- PostgreSQL  
UPDATE users SET first_name = substring_index(name, ' ', 1);  -- MySQL
```

- Suits document-like structures
- Bad support for joins
- Bad support for many-to-many relationships

Schema-on-read and storage locality

- Good when there are many different types of objects
- Structure of objects determined by external systems (you have no control)
- When all objects are expected to have the same format, schema-on-read is not advantageous
- A document is usually stored as a single continuous string (JSON, BSON, XML).
- Gives locality when needing to access the whole document
- SQL databases have recently acquired support for XML and JSON. Making SQL DBs and document DBs similar.

Query languages for data

- Imperative query

```
function getSharks() {  
    var sharks = [];  
    for (var i = 0; i < animals.length; i++) {  
        if (animals[i].family === "Sharks") {  
            sharks.push(animals[i]);  
        }  
    }  
    return sharks;  
}
```

- Declarative query

```
SELECT * FROM animals WHERE family = 'Sharks';
```

- SQL gives room for optimizations and parallel execution, e.g. using indexes

Declarative queries on the web

- CSS

```
li.selected > p {  
    background-color: blue;  
}
```

- XSL

```
<xsl:template match="li[@class='selected']/p">  
    <fo:block background-color="blue">  
        <xsl:apply-templates/>  
    </fo:block>  
</xsl:template>
```

- Javascript
(DOM)

```
var liElements = document.getElementsByTagName("li");  
for (var i = 0; i < liElements.length; i++) {  
    if (liElements[i].className === "selected") {  
        var children = liElements[i].childNodes;  
        for (var j = 0; j < children.length; j++) {  
            var child = children[j];  
            if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {  
                child.setAttribute("style", "background-color: blue");  
            }  
        }  
    }  
}
```

MapReduce querying

- MapReduce is a query concept popularized by Google
- MongoDB also supports a form of MapReduce
- In SQL

```
SELECT date_trunc('month', observation_timestamp) AS observation_month,  
       sum(num_animals) AS total_animals  
  FROM observations  
 WHERE family = 'Sharks'  
 GROUP BY observation_month;
```

- MongoDB

```
db.observations.mapReduce(  
  function map() { ②  
    var year = this.observationTimestamp.getFullYear();  
    var month = this.observationTimestamp.getMonth() + 1;  
    emit(year + "-" + month, this.numAnimals); ③  
  },  
  function reduce(key, values) { ④  
    return Array.sum(values); ⑤  
  },  
  {  
    query: { family: "Sharks" }, ①  
    out: "monthlySharkReport" ⑥  
  }  
)
```

MapReduce querying (2)

- Map and reduce may only use data that is passed to them as input
- They cannot have side-effects
- «Low-level programming model» for distributed execution on a cluster of computers
- SQL may be run distributed and in parallel and may be optimized

Graph-like data models

- When many-to-many relationships are common, a graph model is appropriate
- Vertices and edges
 - Social graph
 - The web graph
 - Road and rail networks
- Multiple types of edge and nodes

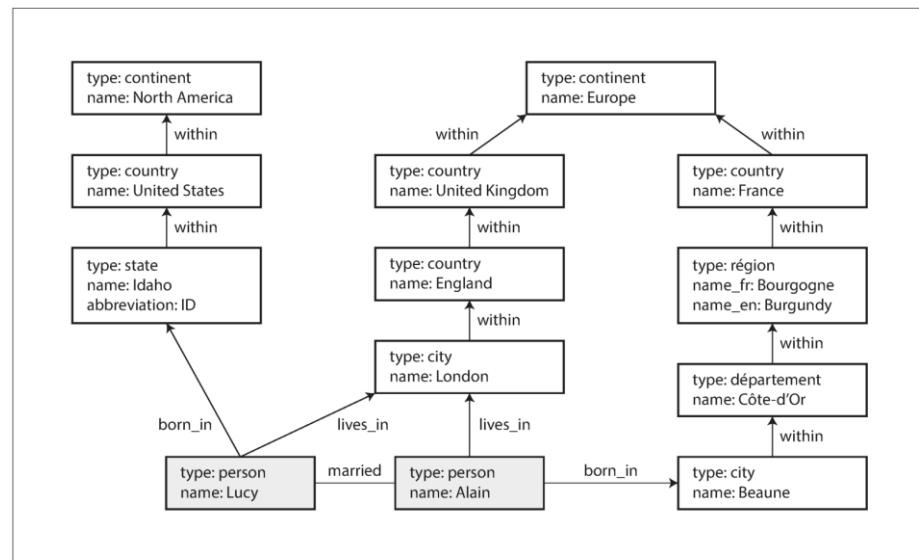


Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

Graph-like data models (2)

- Property graph: Neo4j, Titan and InfiniteGraph
- Triple store: Datomic, AllegroGraph
- Query languages: Cypher, SPARQL and Datalog

Property graphs

- Vertex (id, outgoing edges, incoming edges, properties)
- Edge (id, tail edge, head edge, label, properties)
- Edges between any type of vertex (no restrictions)
- Easy to traverse the graph
- Labels give a rich modeling framework

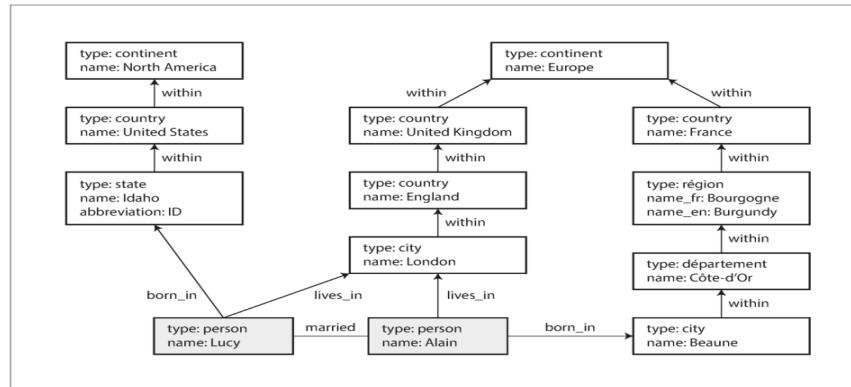


Figure 2-5. Example of graph-structured data (boxes represent vertices, arrows represent edges).

The Cypher query language

- Neo4j's query language

Example 2-3. A subset of the data in Figure 2-5, represented as a Cypher query

```
CREATE
  (NAmerica:Location {name:'North America', type:'continent'}),
  (USA:Location      {name:'United States', type:'country'   }),
  (Idaho:Location    {name:'Idaho',           type:'state'     }),
  (Lucy:Person        {name:'Lucy'            }),
  (Idaho) -[:WITHIN]-> (USA)  -[:WITHIN]-> (NAmerica),
  (Lucy)  -[:BORN_IN]-> (Idaho)
```

Example 2-4. Cypher query to find people who emigrated from the US to Europe

```
MATCH
  (person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
  (person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name
```

- Query may be executed in several ways

Graph queries in SQL

- Variable number of joins to traverse a graph?
- WITH RECURSIVE introduced in SQL:1999

Example 2-5. The same query as Example 2-4, expressed in SQL using recursive common table expressions

WITH RECURSIVE

```
-- in_usa is the set of vertex IDs of all locations within the United States
in_usa(vertex_id) AS (
    SELECT vertex_id FROM vertices WHERE properties->>'name' = 'United States' ①
    UNION
    SELECT edges.tail_vertex FROM edges ②
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'within'
),
-- in_europe is the set of vertex IDs of all locations within Europe
in_europe(vertex_id) AS (
    SELECT vertex_id FROM vertices WHERE properties->>'name' = 'Europe' ③
    UNION
    SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.label = 'within'
),
-- born_in_usa is the set of vertex IDs of all people born in the US
born_in_usa(vertex_id) AS ( ④
    SELECT edges.tail_vertex FROM edges
        JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
        WHERE edges.label = 'born_in'
),
```

```
-- lives_in_europe is the set of vertex IDs of all people living in Europe
lives_in_europe(vertex_id) AS ( ⑤
    SELECT edges.tail_vertex FROM edges
        JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
        WHERE edges.label = 'lives_in'
)
SELECT vertices.properties->>'name'
FROM vertices
-- join to find those people who were both born in the US *and* live in Europe
JOIN born_in_usa    ON vertices.vertex_id = born_in_usa.vertex_id ⑥
JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;
```

Triple-Stores and SPARQL

- (subject, predicate, object), e.g. (Jim, likes, bananas)
- Subject corresponds to a vertex
- Object corresponds to
 - A primitive datavalue or
 - Another vertex

Example 2-6. A subset of the data in Figure 2-5,

```
@prefix : <urn:example:>.  
_:lucy a :Person.  
_:lucy :name "Lucy".  
_:lucy :bornIn _:idaho.  
_:idaho a :Location.  
_:idaho :name "Idaho".  
_:idaho :type "state".  
_:idaho :within _:usa.  
_:usa a :Location.  
_:usa :name "United States".  
_:usa :type "country".  
_:usa :within _:namerica.  
_:namerica a :Location.  
_:namerica :name "North America".  
_:namerica :type "continent".
```

Semantic web and RDF

- Semantic web describes machine readable data of the web
- RDF – Resource Description Framework

Example 2-8. The data of Example 2-7, expressed using RDF/XML syntax

```
<rdf:RDF xmlns="urn:example:"  
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">  
  
  <Location rdf:nodeID="idaho">  
    <name>Idaho</name>  
    <type>state</type>  
    <within>  
      <Location rdf:nodeID="usa">  
        <name>United States</name>  
        <type>country</type>  
        <within>  
          <Location rdf:nodeID="namerica">  
            <name>North America</name>  
            <type>continent</type>  
          </Location>  
        </within>  
      </Location>  
    </within>  
  </Location>  
  <Person rdf:nodeID="lucy">  
    <name>Lucy</name>  
    <bornIn rdf:nodeID="idaho"/>  
  </Person>  
</rdf:RDF>
```

The SPARQL query language

- SPARQL is a query language for triple-stores using RDF

Example 2-9. The same query as Example 2-4, expressed in SPARQL

```
PREFIX : <urn:example:>

SELECT ?personName WHERE {
  ?person :name ?personName.
  ?person :bornIn / :within* / :name "United States".
  ?person :livesIn / :within* / :name "Europe".
}

(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location) # Cypher

?person :bornIn / :within* ?location.                      # SPARQL
```

Datalog

- Old model based on predicate logic
- Predicate(subject, object)

Example 2-10. A subset of the data in Figure 2-5

```
name(namerica, 'North America').  
type(namerica, continent).  
  
name(usa, 'United States').  
type(usa, country).  
within(usa, namerica).  
  
name(idaho, 'Idaho').  
type(idaho, state).  
within(idaho, usa).  
  
name(lucy, 'Lucy').  
born_in(lucy, idaho).
```

Example 2-11. The same query as Example 2-4, expressed in Datalog

```
within_recursive(Location, Name) :- name(Location, Name).      /* Rule 1 */  
  
within_recursive(Location, Name) :- within(Location, Via),      /* Rule 2 */  
                                 within_recursive(Via, Name).  
  
migrated(Name, BornIn, LivingIn) :- name(Person, Name),        /* Rule 3 */  
                                   born_in(Person, BornLoc),  
                                   within_recursive(BornLoc, BornIn),  
                                   lives_in(Person, LivingLoc),  
                                   within_recursive(LivingLoc, LivingIn).  
  
?- migrated(Who, 'United States', 'Europe').  
/* Who = 'Lucy'. */
```

Datalog (2)

- A rule applies if the system can find a match for all predicates on the righthand side

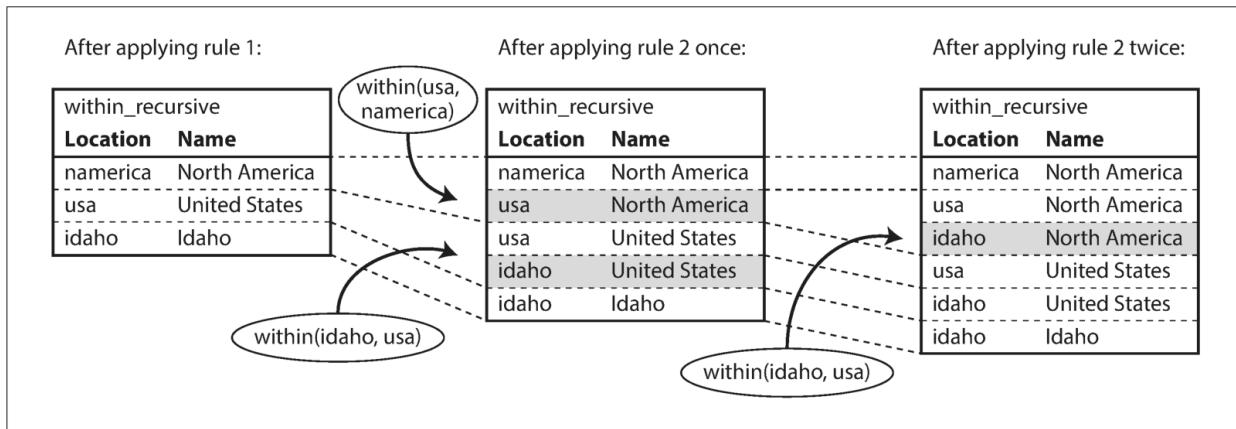


Figure 2-6. Determining that Idaho is in North America, using the Datalog rules from Example 2-11.

Document vs graph databases

- Document databases target use cases where data comes in self-contained documents and relationships between one document and another are rare.
- Graph databases go in the opposite direction, targeting use cases where anything is potentially related to everything.

TDT4225

Chapter 3 – Storage and Retrieval

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Data structures that power your database

- The basic function of a database is to store data and later you could retrieve it
- Basic storage + indexes (heap file + index)
- To speed up retrieval you use indexes
- Indexes speed up queries, but slow down writes
- Thus, the application developer or the database administrator choose which indexes to be created based on the application and query pattern

Hash indexes

- In-memory hash map + disk
- + compactions

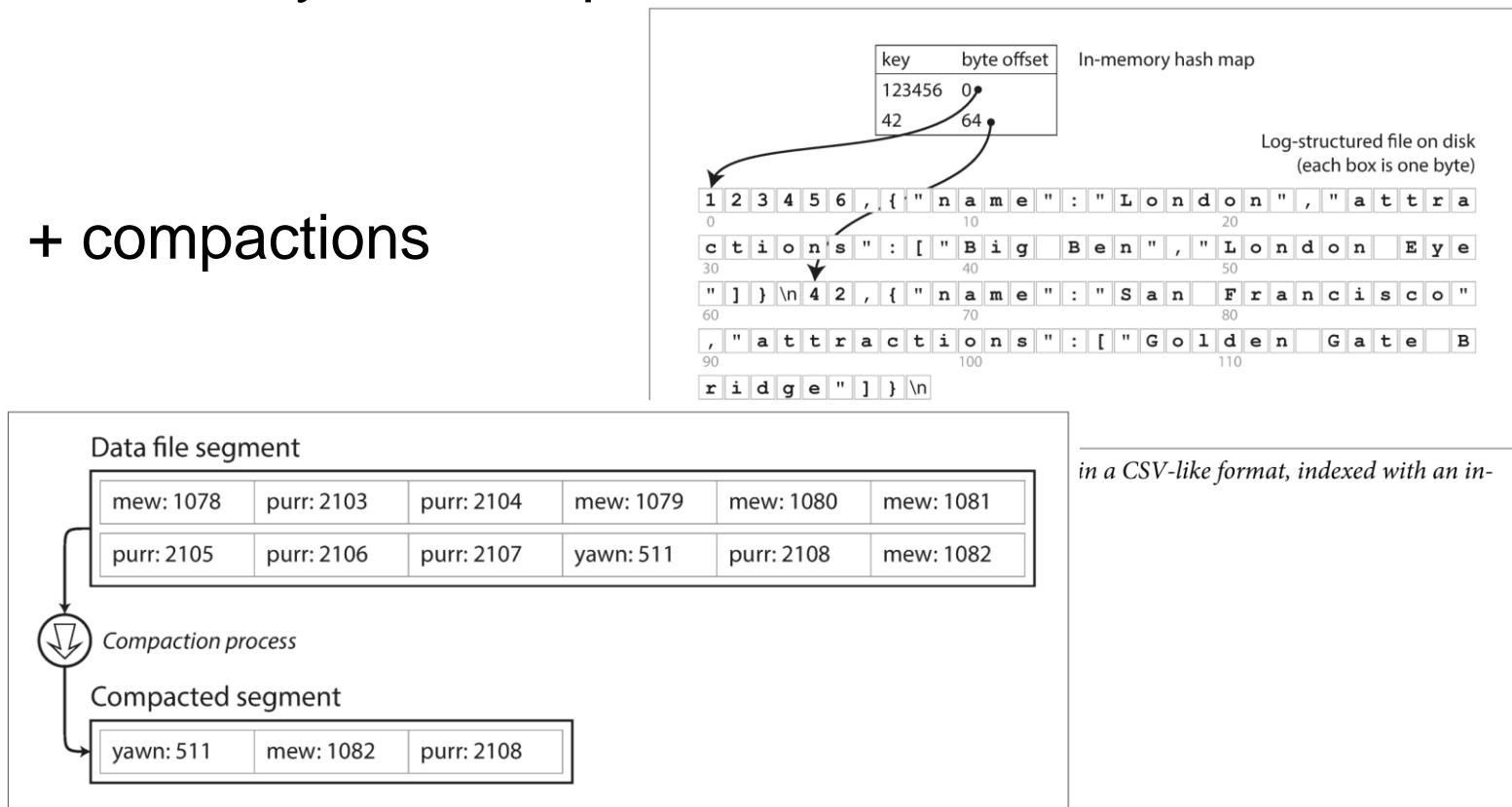


Figure 3-2. Compaction of a key-value update log (counting the number of times each cat video was played), retaining only the most recent value for each key.

Merging + compaction

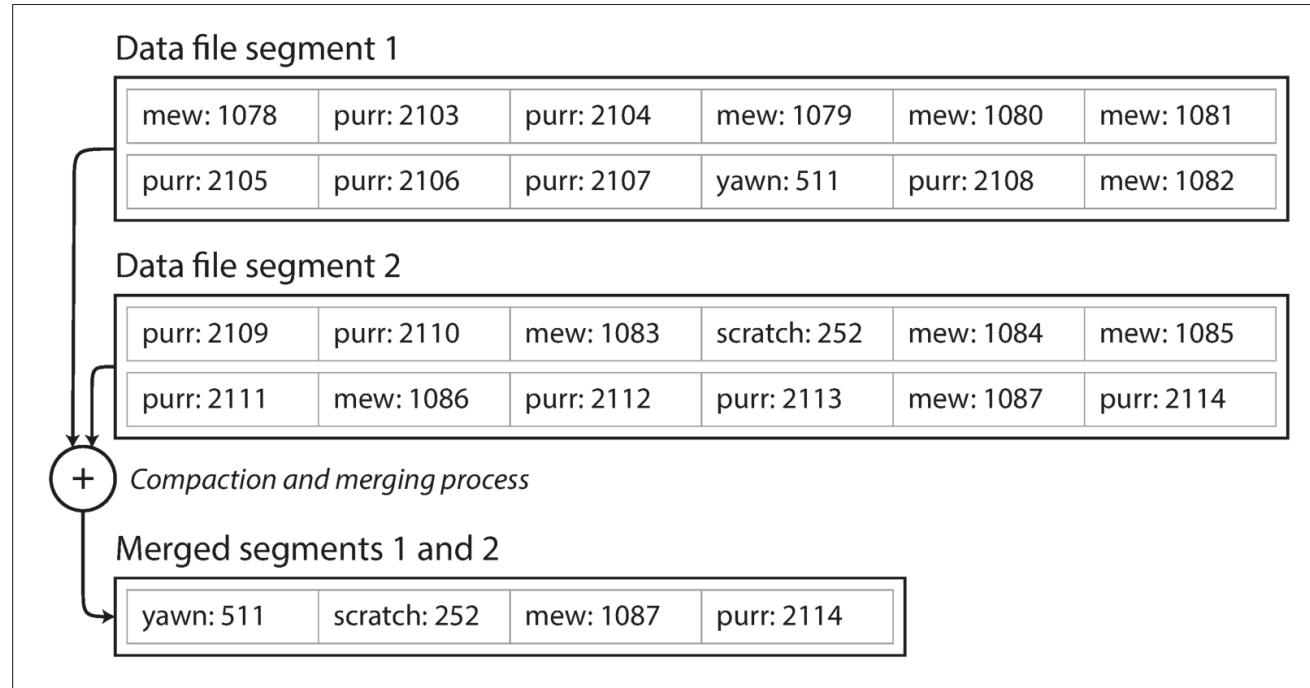


Figure 3-3. Performing compaction and segment merging simultaneously.

Details

- File format: binary formats? Better than csv.
- Deleting records (tombstones)
- Crash recovery: rebuild hash index. *Bitcask* stores a snapshot of the hashmap on disk
- Partially written records. Use checksums. Flip/flop on blocks.
- Concurrency control. Append only helps.
- Append only (vs in-place update)
 - Sequential writes
 - Concurrency and crash recovery become easier
 - Merging prevents fragmentation
- Hashmap must fit in-memory and range queries are problematic

SSTables and LSM-Trees

- Sorted String Table (SSTable)
- Sorted tables
- Efficient merge, like merge-sort

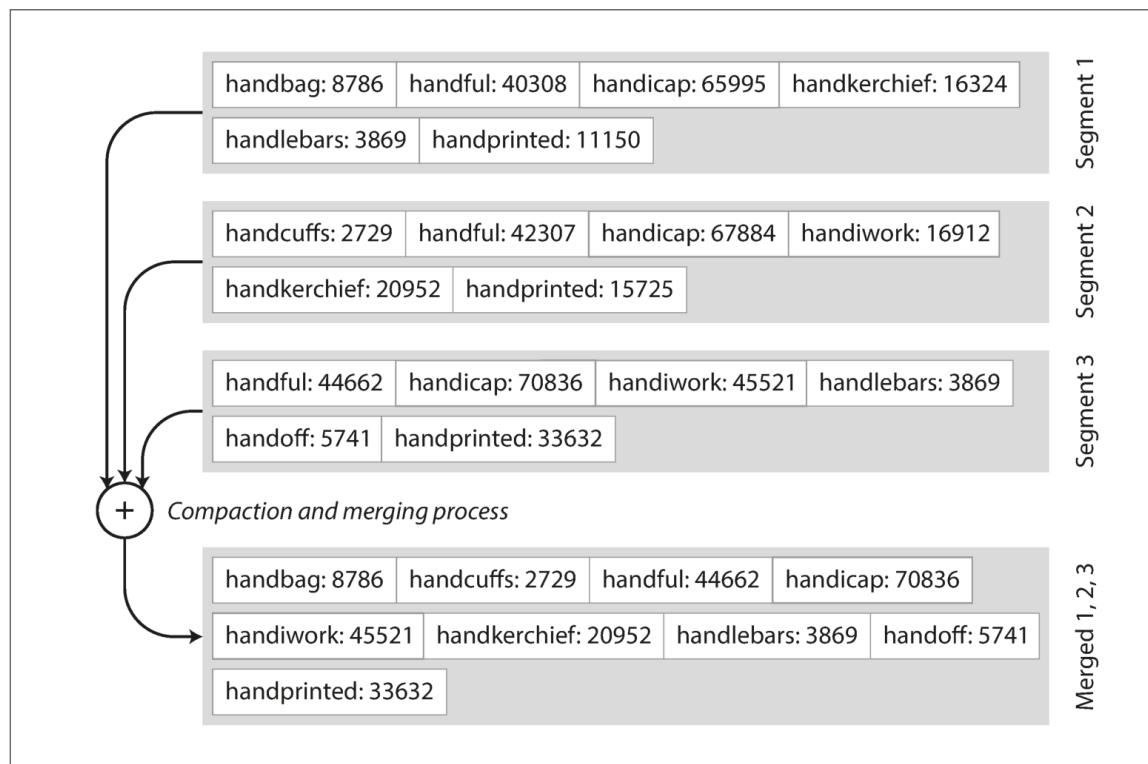


Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

SSTables and indexes

- An index within each SSTable, e.g. SkipLists
- Compression is used as well

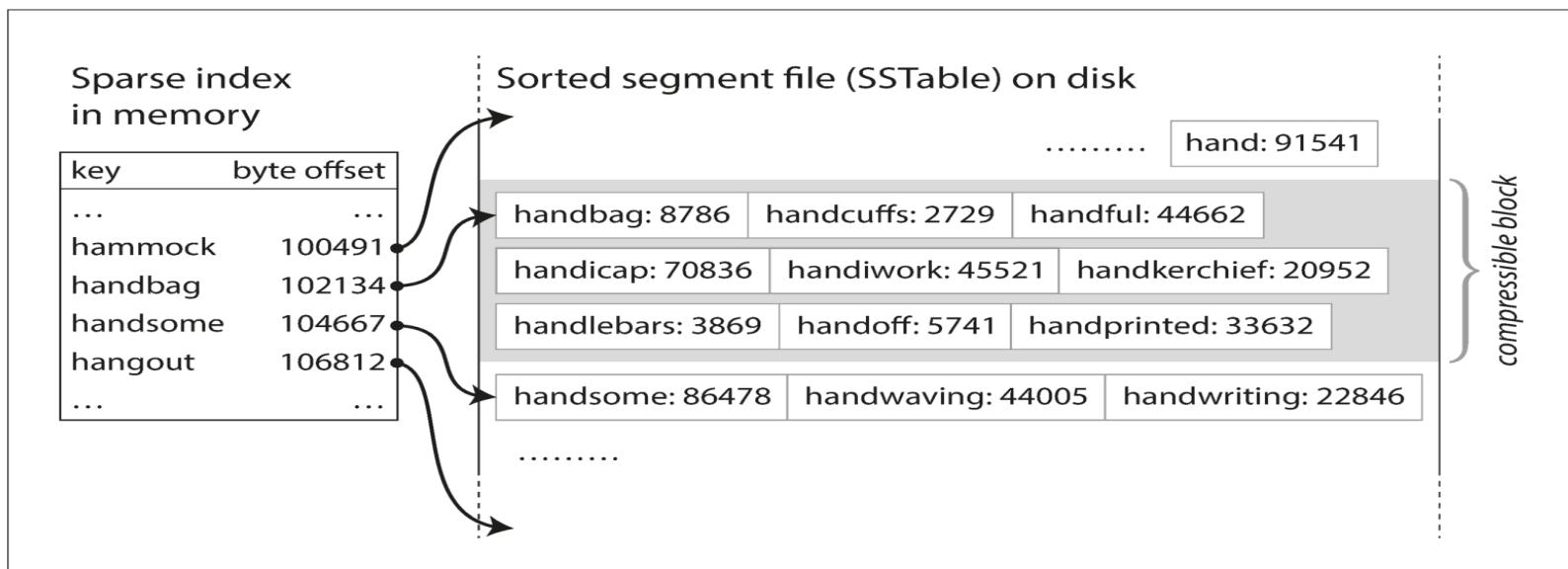


Figure 3-5. An SSTable with an in-memory index.

Constructing and maintaining SSTables

- Sort the records in MemTable using some type of tree, e.g. SkipLists as in LevelDB/RocksDB.
- When MemTable gets bigger than some threshold, write it as an SSTable to disk.
- While writing MemTable, a new MemTable is used for new inserts.
- At read, try MemTable, then SSTables at disk ...
- Merge and compact SSTables.
- Use a write-ahead log in case of crash (and MemTable is lost)
- Log may be discarded when MemTable is written to disk

LSM-trees

- LevelDB (Google) and RocksDB (Facebook) are LSM-based key-value storage engines
- Riak may use RocksDB as an alternative to Bitcask.
- Cassandra and HBase use LSM-trees
- Idea created by O'Neil & O'Neil in 1996 inspired by log-structured file systems (Ousterhout)
- Google made BigTable
- Bloom filters are used within each MemTable and SSTable to easily filter out requests to non-existing keys.
- Compaction and merge: Size-tiered and level-tiered

B+-trees

- The standard database method
- Height-balanced tree with blocks as nodes
- All «user records» are at leaf level («bottom»)
- Typical height: 2, 3 or 4.
- Minimum 50 % filldegree in blocks
- Average 67 % filldegree in blocks
- Records are sorted on the key, and the tree supports
 - Direct search on key
 - Range search
 - Sequential, sorted scans
 - Good for most uses, except heavy write (insert) load

B+-trees vs LSM-trees

- Writes faster for LSM-trees due to low write amplification
- Reads faster for B+-trees?
- B+-trees must write pages even if only a few bytes are updated
- LSM trees can provide a higher write throughput due to larger write units and sequential writes
- LSM trees can be compressed better due to larger units
- Compaction in LSM trees may interfere with ongoing reads and writes
- There is no quick and easy rule: You have to test which is best for you.

Secondary indexes, heap files and clustered indexes

- Secondary indexes may be created. Often, they are not unique. Must store multiple values per search key.
- A value here is either a pointer (RecordPtr) or another key.
- Heap files store records in insert order, and indexes are used to provide uniqueness and quick access
- Often, records are stored within the index itself, i.e. Clustered index
- MySQL InnoDB and tables in SQL Server use clustered B+-trees

Multi-column indexes

- Concatenated index, e.g. lastname + firstname
- Multiple dimensions (spatial index)

```
SELECT * FROM restaurants WHERE latitude > 51.4946 AND latitude < 51.5079  
AND longitude > -0.1162 AND longitude < -0.1004;
```

- Regular B-trees don't support spatial indexes
- Space-filling-curves (Z/Morton/Hilbert) could be stored within a B+-tree
- R-trees have multiple dimensions built-in
- Could be used for multiple dimensions in general, e.g. age and location

Full-text search and fuzzy indexes

- Inverted index to support full-text search
- Many search engines support database-like functionality, e.g. schemas
- They often have a very coarse-grained update policy, e.g. merge in a new index into a bigger one.
- Lucene allows searches with edit-distance, e.g. misspelled words and other linguistic tricks (synonyms etc.)

Keeping everything in-memory

- Disks require careful layout to become performant
- In-memory databases: Memcached, VoltDB, MemSQL, Oracle TimesTen.
- RAMCloud, Redis and Couchbase are memory-oriented
- They use a memory-based layout of data, and not encoded for disk
- Anti-caching: Store everything in memory, but evicting the least recently used data item (record) to disk
- Non-volatile memory (NVM) could change the architecture of databases

Transaction processing or analytics

- Traditionally, databases were used for business processing.
- OLTP -- Online transaction processing is used for any access pattern where you need answers quickly
- OLAP – Online analytic processing is used for access patterns which do analysis of large datasets

Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Data warehousing

- An enterprise may have dozens/hundreds of OLTP systems
- Most of these are operated by separate teams
- A data warehouse integrates such for analytic purposes

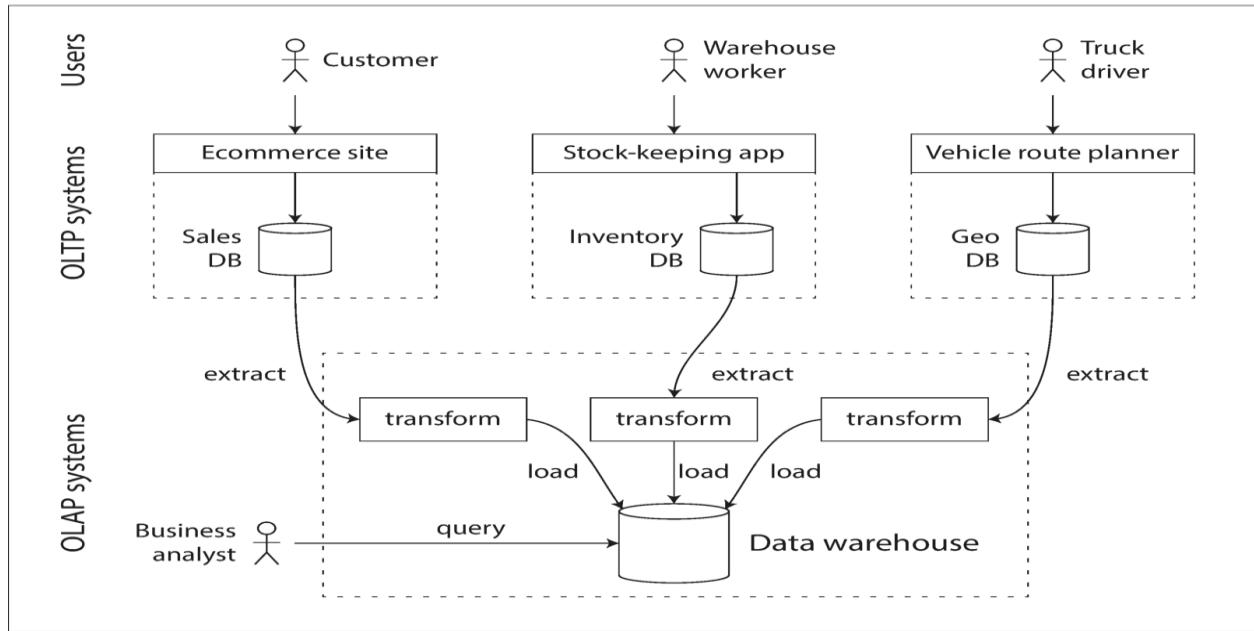


Figure 3-8. Simplified outline of ETL into a data warehouse.

OLTP vs data warehouses

- Data warehouses use SQL, but may have other storage backends using other types of indexes.
- Column stores vs. row stores
- MS SQL server and SAP Hana support both OLTP and OLAP
- Terradata, Vertica, ParAccel, SAP Hana sell commercial data warehouses
- Apache Hive, Spark SQL, Cloudera Impala, Facebook presto, etc. are open source alternatives

Star Schemas and Snowflakes

- *Star schemas* or dimensional modeling
- *Snowflakes* are schemas, where dimensions may be normalized

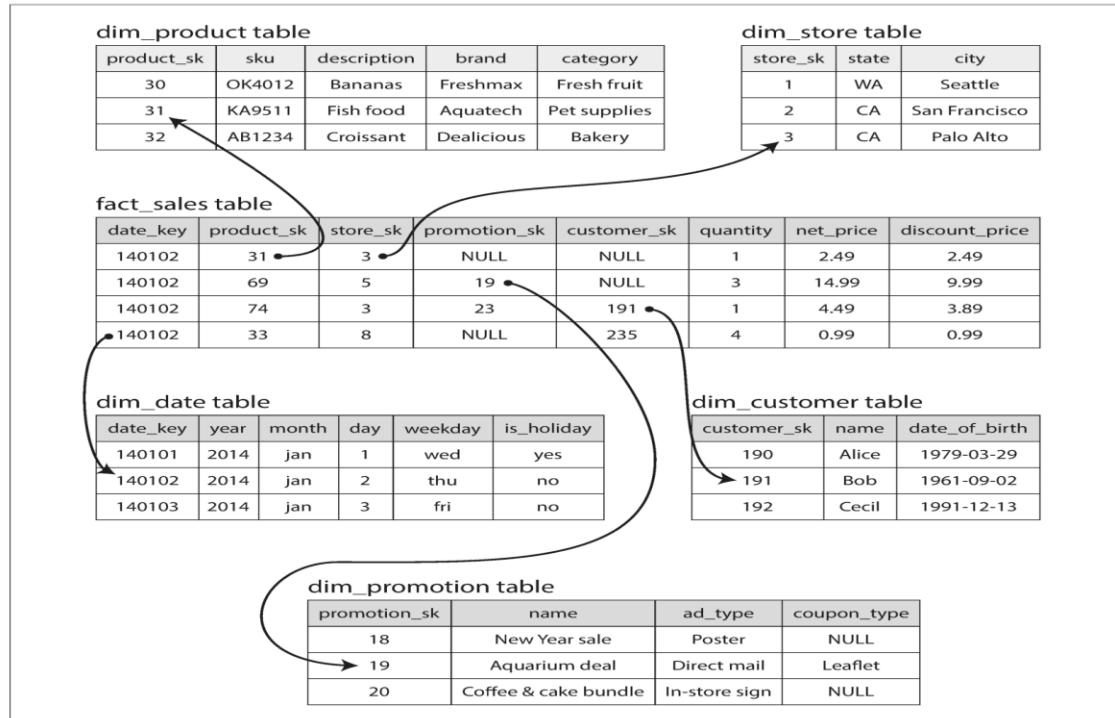


Figure 3-9. Example of a star schema for use in a data warehouse.

Column stores (1)

- Row storage for OLTP and column storage for OLAP

Example 3-1. Analyzing whether people are more inclined to buy fresh fruit or candy, depending on the day of the week

```
SELECT
    dim_date.weekday, dim_product.category,
    SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
    JOIN dim_date      ON fact_sales.date_key      = dim_date.date_key
    JOIN dim_product  ON fact_sales.product_sk = dim_product.product_sk
WHERE
    dim_date.year = 2013 AND
    dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
    dim_date.weekday, dim_product.category;
```

Column stores (2)

- Store all columns of the same table in same (row) order

fact_sales table							
date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents: 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents: 69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents: 4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents: NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents: NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents: 1, 3, 1, 5, 1, 3, 1, 1
net_price file contents: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 3-10. Storing relational data by column, rather than by row.

Column compression

- Bitmap encoding: Well suited for «where c in (69, 31)»

Column values:

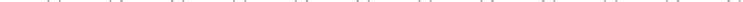
product_sk: 69 69 69 69 74 31 31 31 31 29 30 30 31 31 31 68 69 69

Bitmap for each possible value:

product_sk = 29: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]

product_sk = 30: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0]

product_sk = 31: 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0

product_sk = 68: 

product_sk = 69: 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

product_sk = 74: 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Run-length encoding:

product_sk = 29: 9, 1 (9 zeros, 1 one, rest zeros)

product_sk = 30: 10, 2 (10 zeros, 2 ones, rest zeros)

`product_sk = 31: 5, 4, 3, 3 (5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)`

product_sk = 68: 15, 1 (15 zeros, 1 one, rest zeros)

product_sk = 69: 0, 4, 12, 2 (0 zeros, 4 ones, 12 zeros, 2 ones)

product_sk = 74: 4, 1 (4 zeros, 1 one, rest zeros)

Figure 3-11. Compressed, bitmap-indexed storage of a single column.

Various data warehouse techniques

- Vectorized processing: Avoid branching, Looping over L1 cache. Bitwise operations, etc.
- Sorting rows according to e.g. date? Typical for queries.
- Store the same data in different sort orders (Vertica)
- Writing to a column store: Use a write store, which further distributes data to columns through merge operations
- Materialized aggregates: COUNT, SUM, AVG, MAX, etc
- Materialized views: Used a lot in read-heavy environments

Data cubes / OLAP cubes

- Aggregates in several dimensions: Making some queries faster

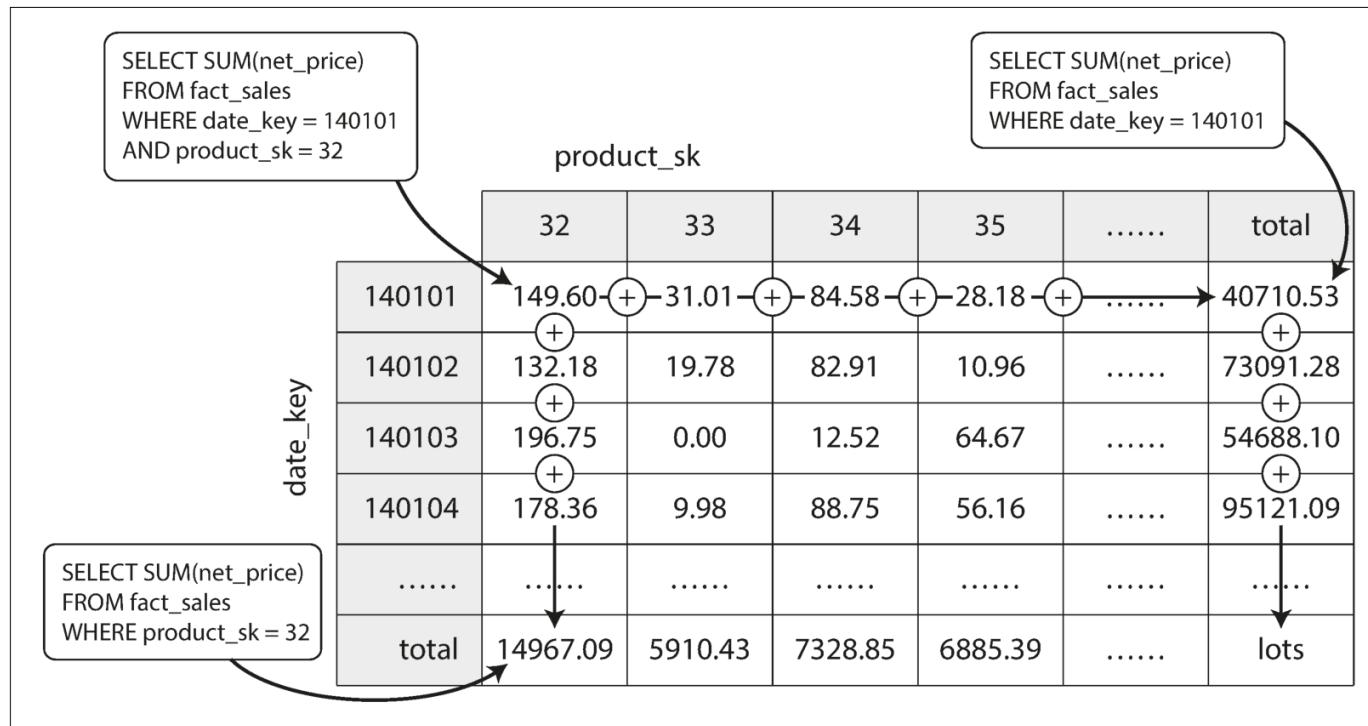


Figure 3-12. Two dimensions of a data cube, aggregating data by summing.

Summary

- OLTP databases: Online users
 - Log structured: LSM trees
 - Update in place: B+-trees
- OLAP databases: System analytics
 - Column stores
 - Data cubes

TDT4225

Chapter 4 – Encoding and Evolution

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Evolvability - Basics

- SQL databases assume a schema – may be changed with ALTER TABLE
- Schema-on-read databases don't enforce a schema – a database may contain older and newer formats
- Server-side applications – rolling upgrades
- Client-side applications – at the mercy of the user who may not upgrade for some time?
- *Backward compatibility*: Newer code can read old data
- *Forward compatibility*: Older code can read new data.
Tricky.

Formats for encoding data

- In memory, data is kept in objects, structs, lists, arrays, hash tables, trees, etc. Use pointers.
- When writing to a file or sending it over the network: Self-contained sequence of bytes. No pointers.
- Encoding/decoding, marshalling/unmarshalling, serialization/deserialization: sending and receiving data
- Built-in language support
 - Java has `java.io.Serializable`
 - Ruby has `Marshal`
 - Python has `pickle`
- Tied to a particular programming language
- Decoding may cause security problems due to arbitrary code has to be installed
- Versioning?
- Performance?

Data formatting languages (textual)

- JSON – Supported by browsers and Javascript
- XML – complex data formats
- CSV – simple data format
- Ambiguity in encoding numbers/strings and int/float
- Support for UNICODE, but no support for binary strings
(encode binary streams using text)
- Schema support for JSON and XML. Used in XML, but skipped in JSON. Hard-coded interpretation in the apps.
- CSV has no schema support
- JSON, XML, and CSV are good enough for many purposes
- “The difficulty in getting organizations to agree on anything, outweighs most others concerns.”

Binary encoding

- Used for internal data between programs
- Compacter and faster than textual encoding
- Binary encodings for JSON: MessagePack, BSON, BJSON, UBJSON, BISON, and Smile.
- For XML: WBXML and Fast Infoset.

Example 4-1. Example record which we will encode in several binary formats in this chapter

```
{  
  "userName": "Martin",  
  "favoriteNumber": 1337,  
  "interests": ["daydreaming", "hacking"]  
}
```

MessagePack

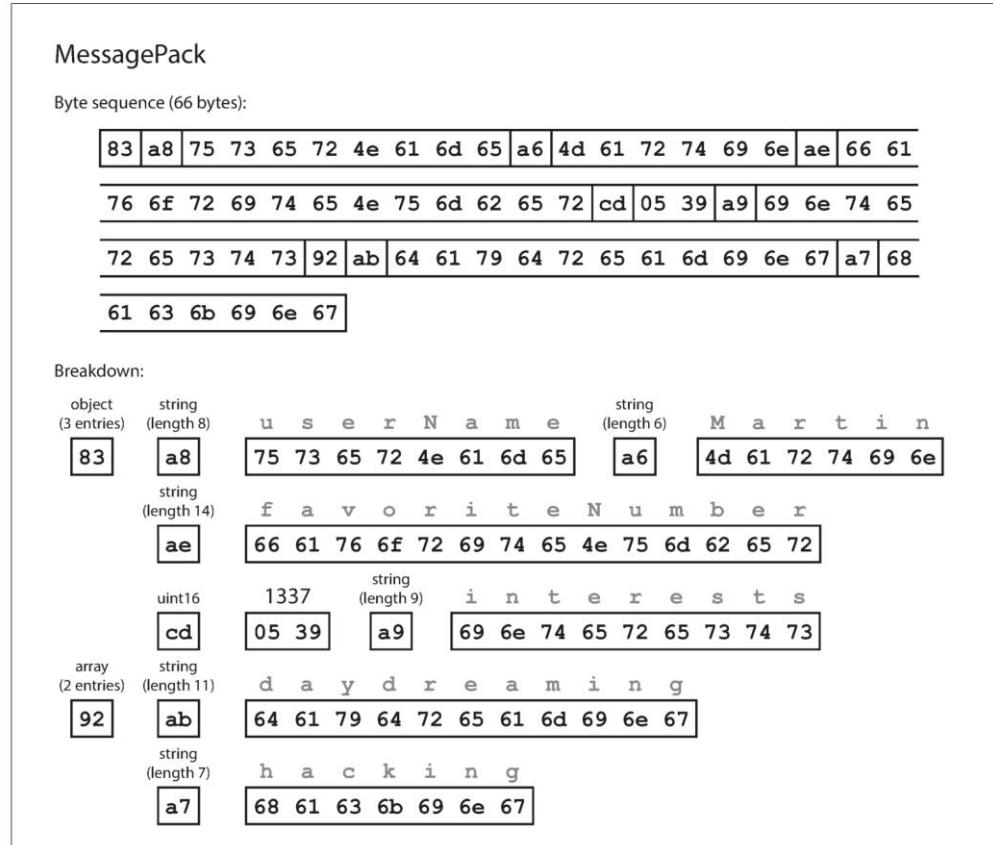


Figure 4-1. Example record (Example 4-1) encoded using MessagePack.

Thrift and Protocol Buffers

- Apache Thrift (Facebook) and Protocol Buffers (Google) are binary encodings
- Both require a schema, with tools to generate code

```
struct Person {  
    1: required string      userName,  
    2: optional i64         favoriteNumber,  
    3: optional list<string> interests  
}
```

```
message Person {  
    required string user_name      = 1;  
    optional int64  favorite_number = 2;  
    repeated string interests       = 3;  
}
```

Thrift's BinaryProtocol

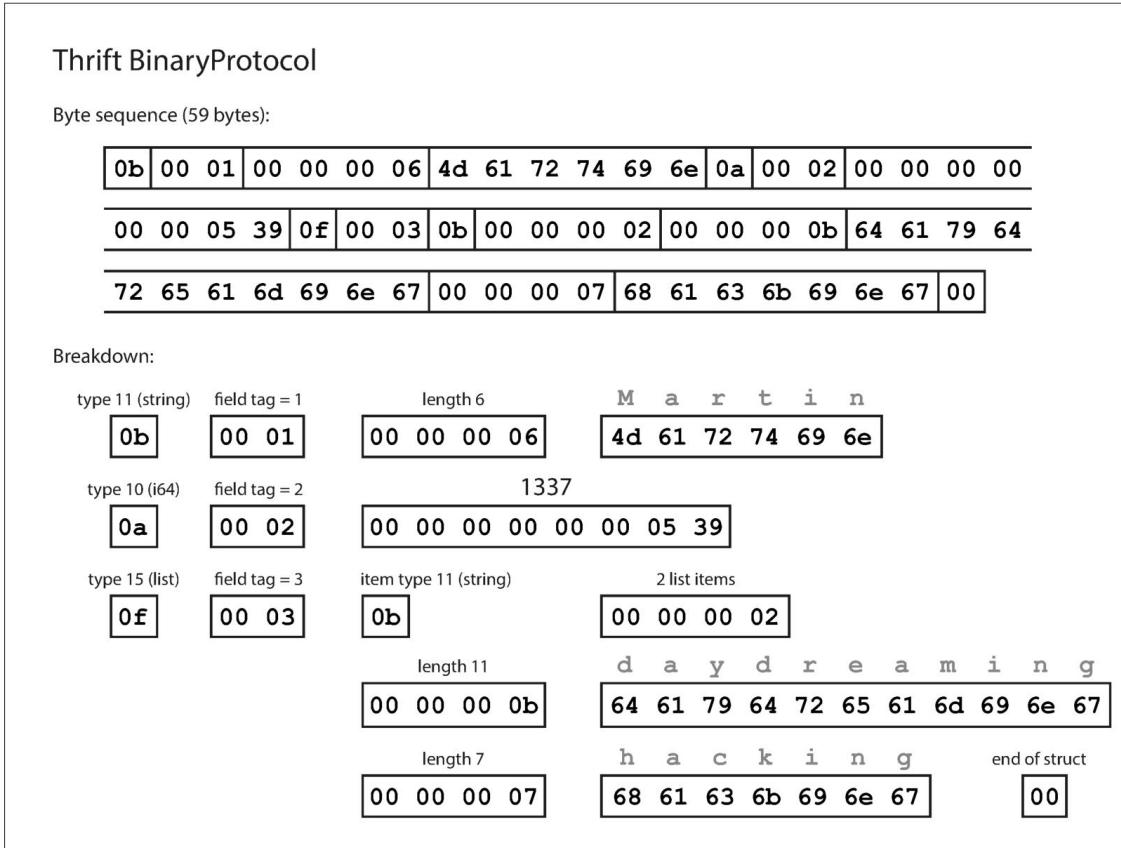


Figure 4-2. Example record encoded using Thrift's BinaryProtocol.

Thrift CompactProtocol

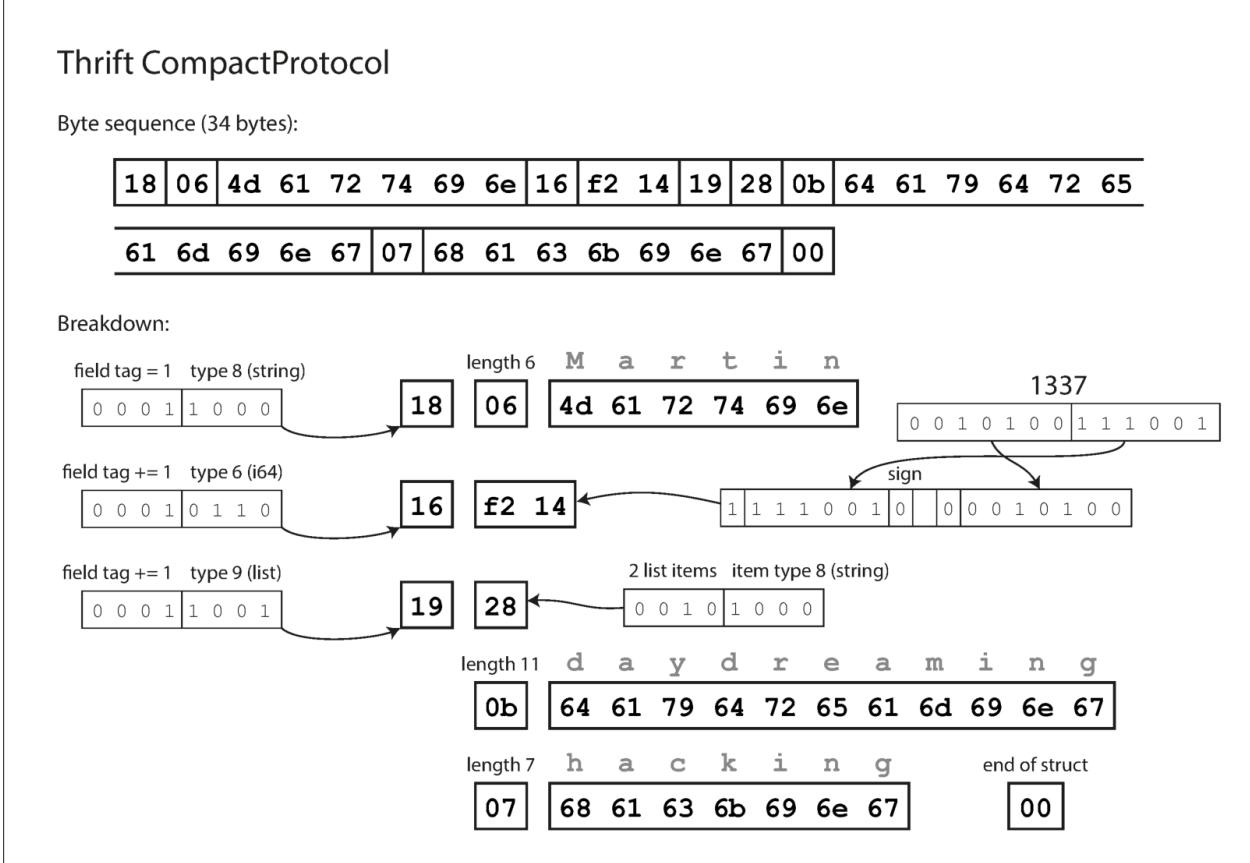
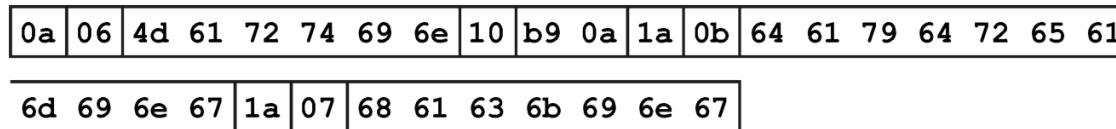


Figure 4-3. Example record encoded using Thrift's CompactProtocol.

Protocol Buffers

Protocol Buffers

Byte sequence (33 bytes):



Breakdown:

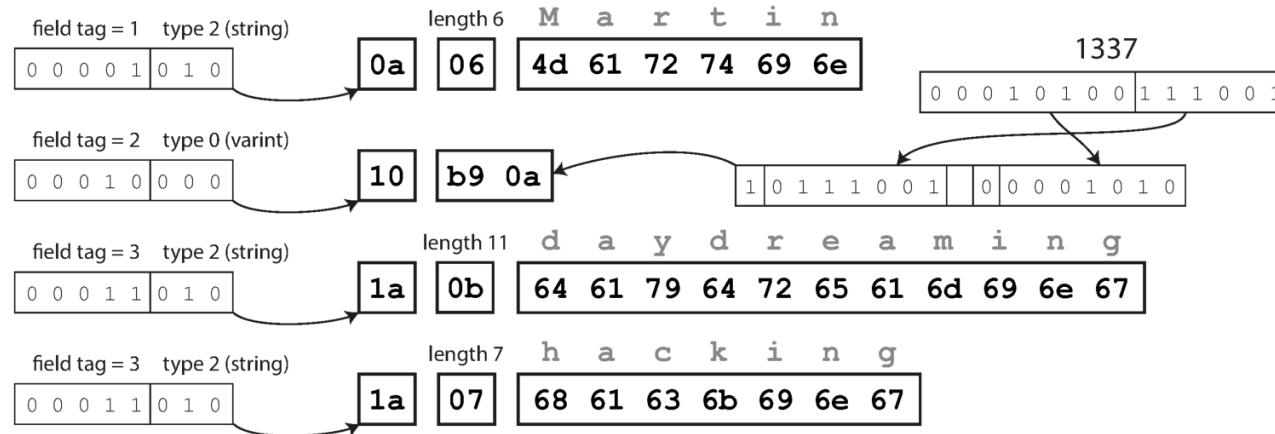


Figure 4-4. Example record encoded using Protocol Buffers.

Schema evolution

- Thrift/Protocol Buffers use schemas and fields are identified with tag numbers
- Field names may change and new fields may be added, but they must be optional or have a default value
- Optional fields may be removed
- Changing datatypes may be possible, check the documentation
- Protocol buffers has a repeated field instead of arrays

Avro

- Apache Avro is another binary encoding format that is interestingly different

```
record Person {  
    string              userName;  
    union { null, long } favoriteNumber = null;  
    array<string>      interests;  
}
```

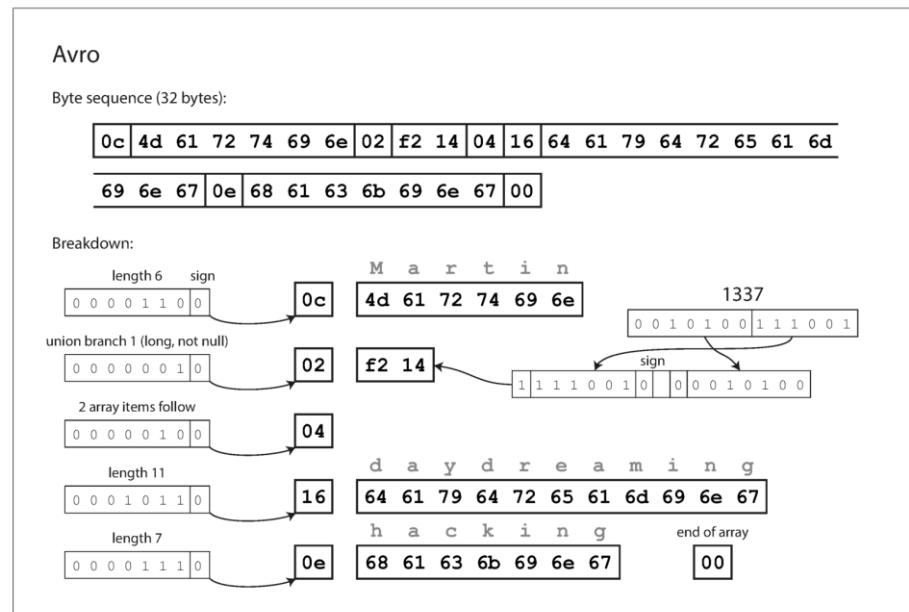


Figure 4-5. Example record encoded using Avro.

Reader's and writer's schema

- Avro resolves differences automatically. When reading you know both schemas.

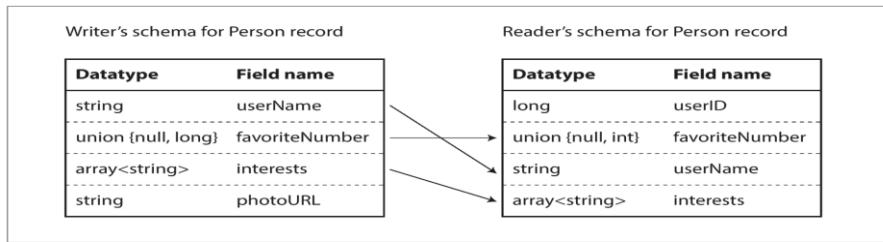


Figure 4-6. An Avro reader resolves differences between the writer's schema and the reader's schema.

- You may add or remove fields which have default values
- Schemas stored with large files
- In databases schema version number included with each record
- On network messages, schema versions are negotiated at startup
- Avro is friendlier to dynamically generated schemas
- Also better for dynamically typed language since the data is self-describing (in contrast to statically typed languages)

Advantages of using a schema

- More compact since they don't use field names
- Schemas are documentation
- Database of schemas keep version history for backward and forward compatibility
- For users of statically typed languages, the use of schemas make compile-time type checking possible

Dataflow through databases

- Store data in databases for later use or exchange
- Data outlives code: old data may exist
- Some problems may appear

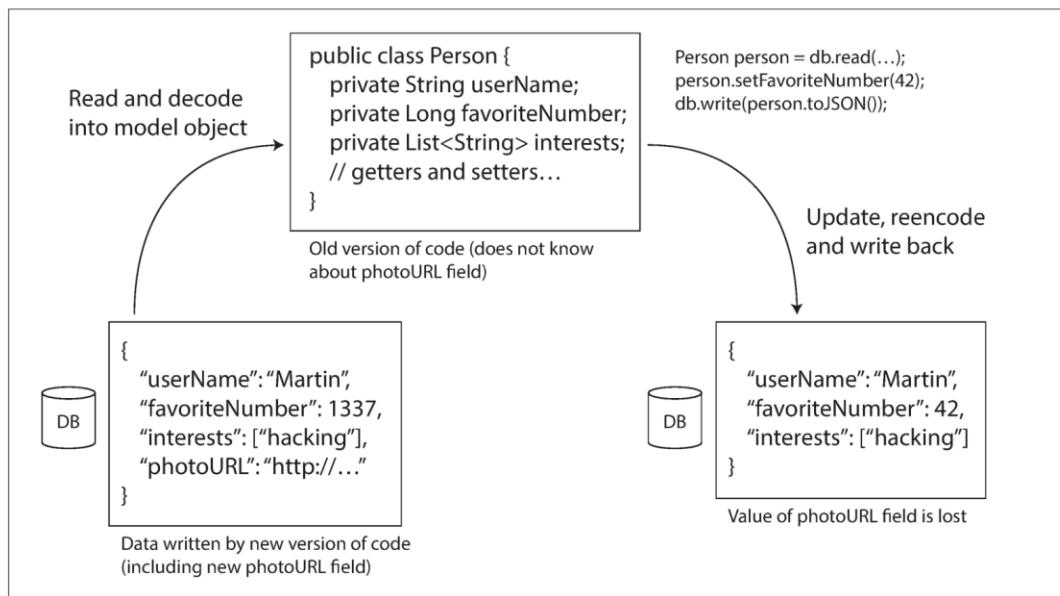


Figure 4-7. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you're not careful.

Dataflow through services: REST / RPC

- Clients and servers: Servers expose services using API
- Web browser and servers:
 - GET to download HTML, CSS, JavaScript, images, etc.
 - POST to submit data
- API uses standard formats: HTTP, URLs, SSL/TLS, HTML, etc
- Applications may also use these protocols and standards (e.g. AJAX)
- Microservice architectures: Decompose a large application into smaller services by area of functionality

Web services

- Is when HTTP is used as the underlying protocol for talking to the service.
 - An app uses HTTP to access a server over the Internet
 - Services within an organization uses HTTP to communicate
 - Services used between organizations, e.g. credit card processing
- REST – a design philosophy -- RESTful interfaces
 - Using HTTP for cache control, authentication, etc.
 - URLs to describe resources
 - Focuses on code generation and use of tools
- SOAP – based on XML and its API is described by WSDL
 - Complex and not human-readable, thus tools are used
 - Used in and between large organizations

Problems with RPCs

- RPC – Remote procedure calls
- EJB – Enterprise Java Beans, Java RMI, Corba, DCOM, Sun RPC, etc.
- Make a request to a remote network service look the same as calling a function or method
 - Unpredictable behavior – error handling necessary
 - Timeouts (no answer)
 - Idempotence? Retries.
 - Response time widely variable
 - Parameters must be marshalled and complex objects are problematic
 - Different programming languages at both ends? Problems with data types?

The future for RPC

- Thrift, Avro, gRPC (Protocol Buffers), Finagle, Rest.li are examples of RPCs.
- Using *futures* simplifies error handling and parallel requests
- gRPC uses *streams (multiple calls and replies)*
- Custom RPC using binary encoding may achieve good performance
- JSON over REST may provide good flexibility and is supported by many tools and programming languages

Message passing dataflow

- Asynchronous message-passing systems
- Message broker / message queue / message-oriented middleware
- Buffer – more reliable communication
- Redeliver message to failed servers
- The sender does not need to know the addresses of receivers
- Send to more recipients
- Decouples senders and receivers
- Asynchronous – send and forget
- One-way
- Systems: TIBCO, WebSphere, Kafka, NATS, RabbitMQ

Distributed actor frameworks

- Programming model with independent actors and messages
- Message may be lost and automatically resent
- The same messages used internally or distributed
- Akka (java), Erlang OTP, Orleans
- Must solve schema evolution without much help from the system.

Summary

- Data encoding should support rolling upgrades
- Must ensure backward compatibility
- Programming language solutions – efficient, but locked
- Textual formats like JSON, XML and CSV support schemas, but are vague about datatypes
- Binary schema–driven formats like Thrift, Protocol Buffers, and Avro allow compact, efficient encoding
- Modes of dataflow
 - Through databases
 - RPC and REST API
 - Message passing

TDT4225

Chapter 5 – Replication

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Distributed data (chap 5-9)

- *Scalability* – partitioning – spread the load across multiple machines
- *Fault tolerance* – replication – takeover when one machine fails
- *Latency* – locality to the application when data is distributed globally
- *Shared nothing* – every computer has all it needs
- *Shared disk* – many computers share disk through network
- *NUMA* – many CPUs, but *non-uniform memory architecture* – each CPU has its «local» memory

Shared-nothing architecture

- Each machine is a node having CPU, memory, disks and network interface
- Coordination between nodes done through network messages
- No special hardware needed
- You need to be aware of the constraints and trade-offs that occur in such a distributed system
- The database cannot magically hide these from you
- Beware: In some cases, a simple single-threaded program can perform significantly better than a cluster with over 100 CPU cores

Replication vs. partitioning

- **Replication:** Keeping a copy of the same data on several different nodes, potentially in different locations.
- Provides redundancy (and fault tolerance)
- **Partitioning:** Splitting a big database into smaller subsets called partitions so that different partitions can be assigned to different nodes (also known as **sharding**). Provides scalability

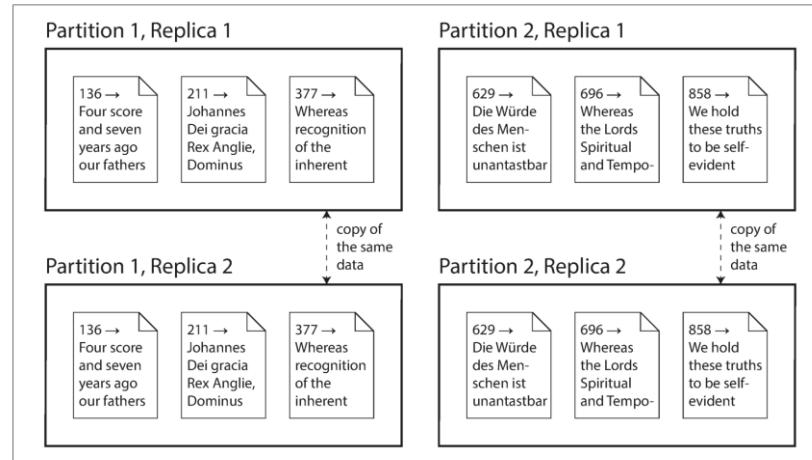


Figure II-1. A database split into two partitions, with two replicas per partition.

Replication

- To keep data geographically close to your users
- Fault tolerance – increased availability
- Scale out for read queries
- If the data that you're replicating does not change over time, then replication is easy
- Replicating changes between nodes:
 - single-leader
 - multi-leader
 - leaderless replication.
- Synchronous vs. asynchronous replication

Leader-based replication

- Also named *active/passive*, *master–slave*, and *primary/backup*
- Writes sent to leader which writes the changes to disk
- Followers (read replicas, slaves, secondaries, or hot standbys) are sent the changes through a replication stream / change log
- Must apply all writes in the same order as they were processed on the leader
- Any replica may be read, but only master accepts writes

Leader-based replication (2)

- PostgreSQL, MySQL, Oracle Data Guard, and SQL Server's AlwaysOn Availability Groups, MongoDB, etc

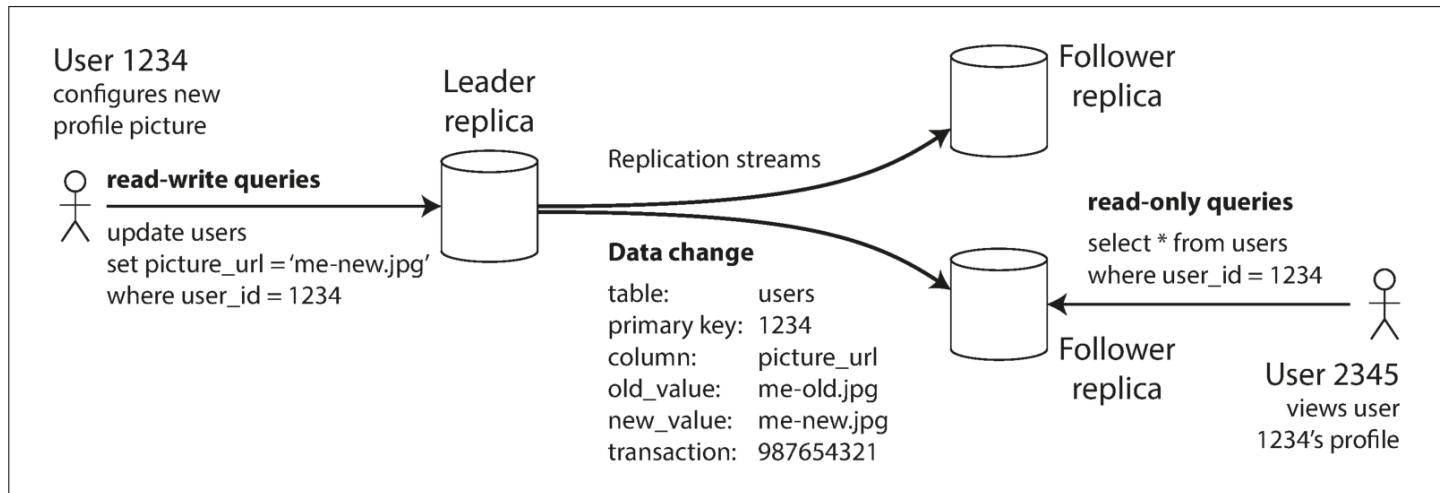


Figure 5-1. Leader-based (master-slave) replication.

Synchronous vs. Asynchronous Replication

- Is the replica update included in the client response (synchronous)
- More complicated failure scenarios for asynchronous

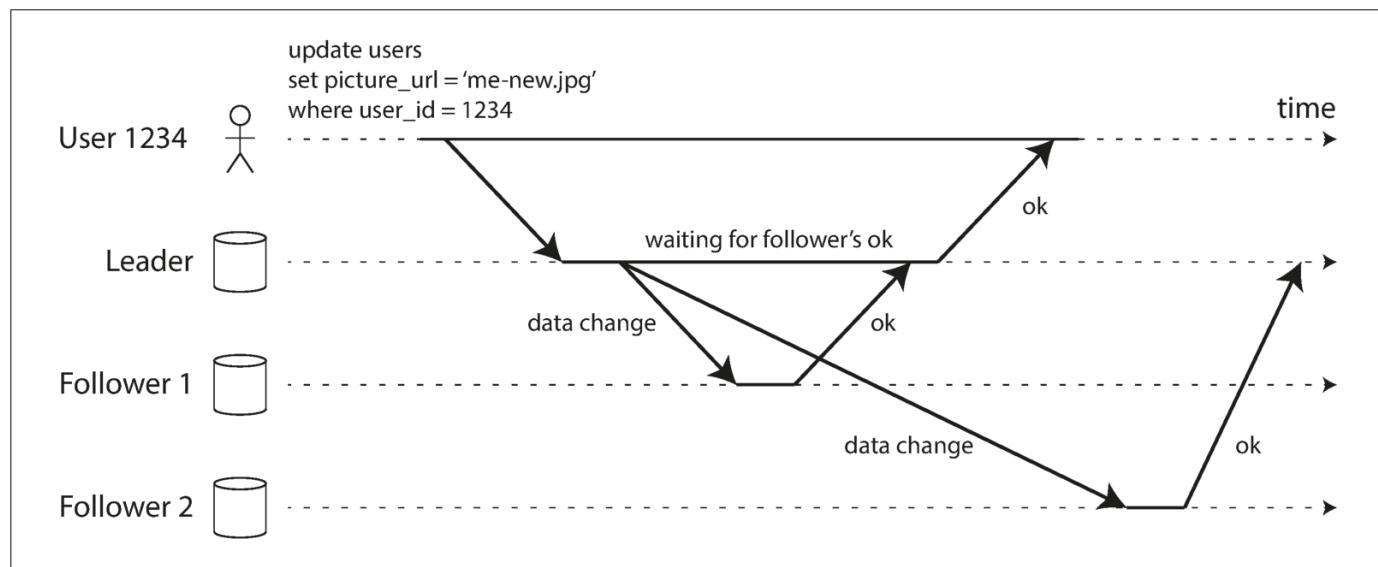


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

Asynchronous replication

- Often, leader-based replication is configured to be completely asynchronous
- May cause lost updates in case of crash of leader
- The leader can continue processing writes, even if all of its followers have fallen behind
- Weakened durability. But widely used anyway.
- Chain replication is a new method applied by Microsoft (but created at Cornell, Rob van Renesse): Updates at head, reads at tail of replication chain

Adding new replicas (repair?)

- Take a consistent snapshot of the leader's database at some point in time
- Copy the snapshot to the new follower node
- Copy the log: all the data changes that have happened since the snapshot was taken (all alive transaction log)
- Caught up when all backlog processed
- Fully automated by some systems, but manual adm commands by some others

Takeover / Failure handling

- Failure detection – I-am-alive protocol?
- Election of new leader, or pre-determined
- Takeover – the new one is the leader
- Problems:
 - Some updates weren't received before takeover. The old leader may have «lost updates» when recovering
 - The new leader should «bump-up» anything used for identification (auto-increment keys and log sequence numbers)
 - Split-brain: 2 new nodes both think they are the leader
 - What timeout to use?
 - Too small -> false failures may be detected.
 - Too big -> long time without any leader (unavailability)

Replication logs – Statement based

- For SQL: INSERT, UPDATE, or DELETE statement is forwarded to followers
- Problems
 - nondeterministic function, NOW() or RAND()
 - autoincrementing columns and concurrent updates
 - statements with side effects (triggers etc.)
- Used in MySQL prior to v 5.1, but now row-based replication is used
- VoltDB uses this, but transactions must be deterministic

Write-Ahead Log-shipping

- Log: append-only sequence of bytes containing all writes (and commits/aborts)
- When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.
- Used in PostgreSQL and Oracle and others
- Low-level details being storage-specific. Copies must be physically equal.
- If the database changes its storage format, typically, it is not possible to run different database versions at different servers
- May be used for online upgrades to database software if the follower is allowed to upgrade its software

Logical (row-based) log replication

- Different formats on log and local storage (e.g. primary keys and not BlockIds in log)
- Inserts, deletes and updates generate different log records containing before and/or after image.
- MySQL's binlog uses this approach and also ships commit log records
- Leader and follower may have different storage formats and storage engines
- Could provide better fault tolerance

Trigger-based replication

- When flexibility is needed
- Replicate a subset of the data
- Need conflict resolution
- Triggers and stored procedures let «application code» be run when updates appear
- The trigger may log the update into a separate change table, which again may be read by a replication process
- General approach with higher runtime cost than special purpose replication approaches

Problems with replication lag

- Leader-based replication OK for loads having mostly reads and a few writes
- And using asynchronous replication
- You may read out-of-date info from a follower: Eventual consistency
- Ranging from milliseconds out-of-date to minutes at high loads

Reading-your-writes consistency

- Write-then-read-what you've written
- Problems when reading a not-updated replica
- Need *read-after-write consistency* / *read-your-writes consistency*

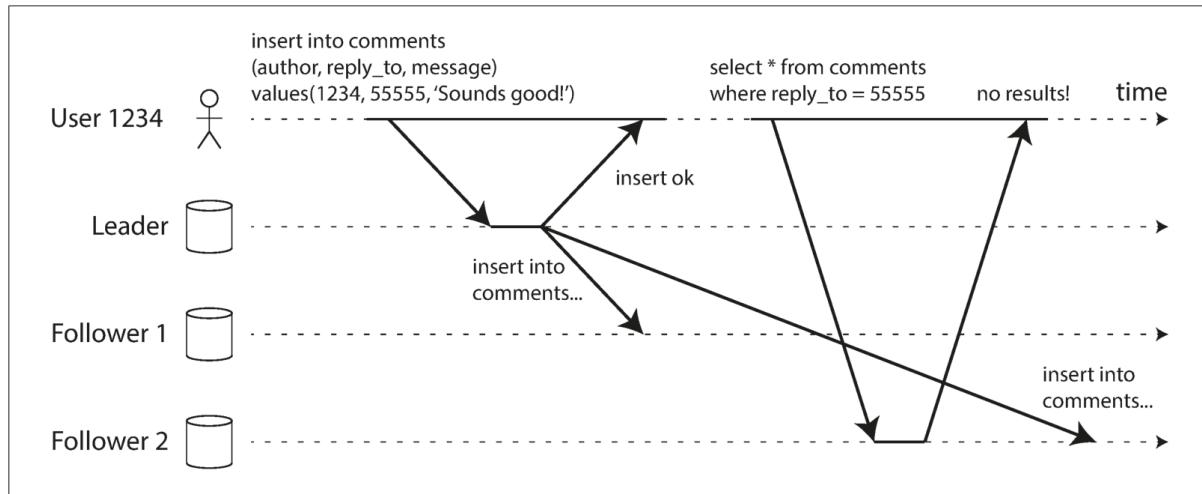


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

Reading your write consistency (2)

- Write/Read your profile to/from *the leader*, and may read everything else from a *follower* (social network example)
- When you have to write everywhere, use a one minute wait and read from the leader. Observe the lag.
- Using logical timestamps: Register write timestamps and use this when reading replicas
- Multi-datacenters. Route to the leader of the correct datacenter
- Complicated: Routing may change over time. Different devices may connect to different datacenters.

Monotonic Reads

- Monotonic reads is a guarantee that you never read older versions of data
- May be ensured by making reads always to the same replica by hashing on the userId.

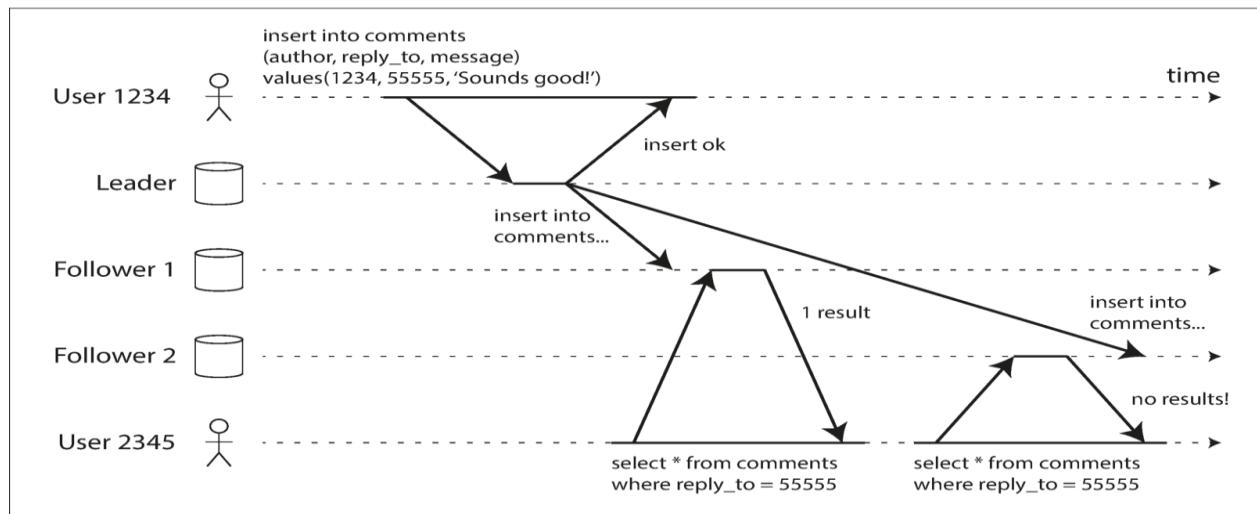


Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

Consistent Prefix Reads

- This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order: Related data go to the same machine.

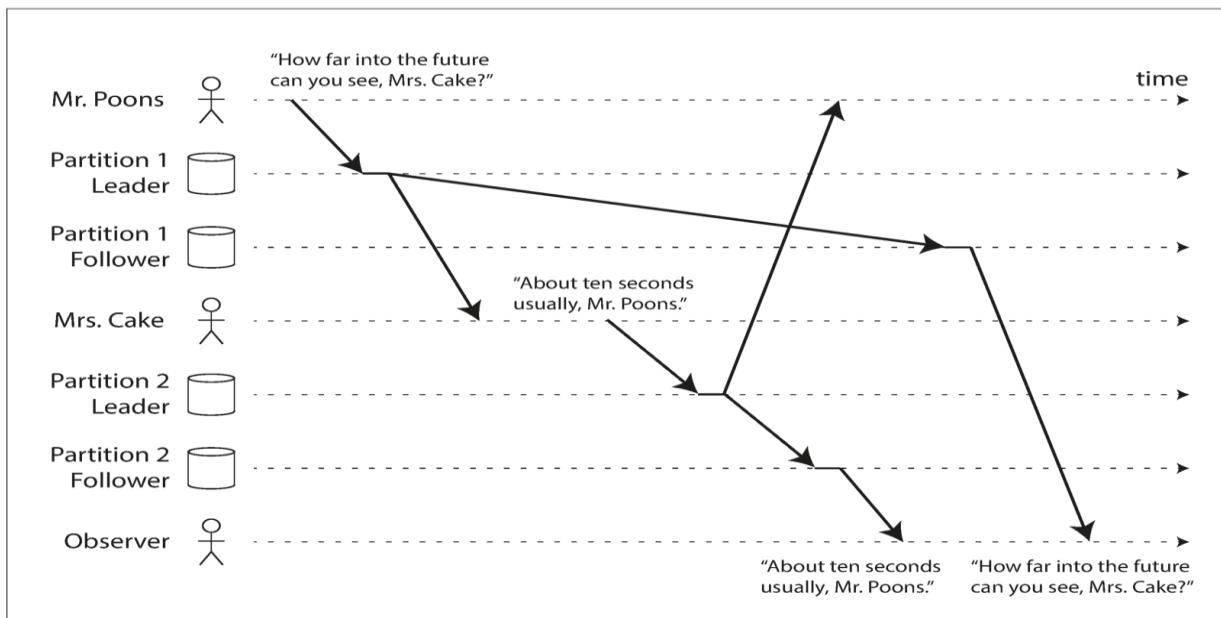


Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

Solutions for replication lag

- Is replication lag a real problem?
- Use better guarantees: E.g. read-your-writes
- Use transactions
- Some NoSQL databases have skipped transactions:
They're too expensive
- NewSQL databases often support distributed
transactions

Multi-Leader Replication

- Allow more than one node to accept writes
- Multi-Leader, master/master or active/active replication
- A leader in each datacenter

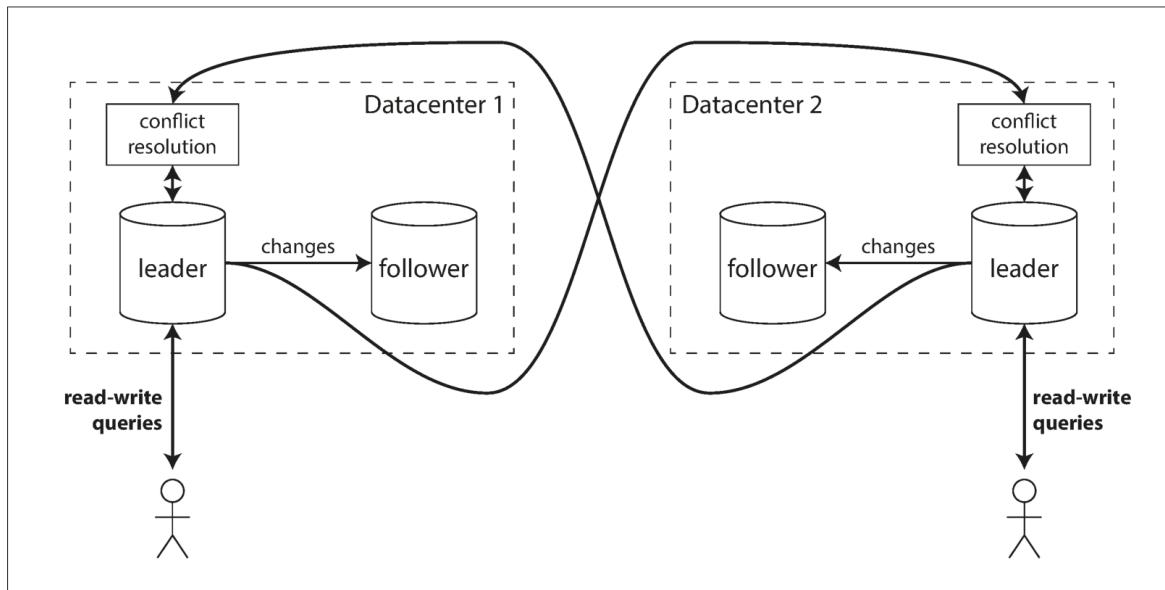


Figure 5-6. Multi-leader replication across multiple datacenters.

Multi-Leader Replication (2)

- Performance (local writes)
- Tolerance of datacenter outages
- Tolerance of network problems
- May cause concurrent update to the same data: Thus, needs conflict resolution
- Disconnected databases, e.g. calendar, have the same problem
- Collaborative editing (Google docs, Office365, etc)

Handling write conflicts

- The biggest problem with multi-leader replication

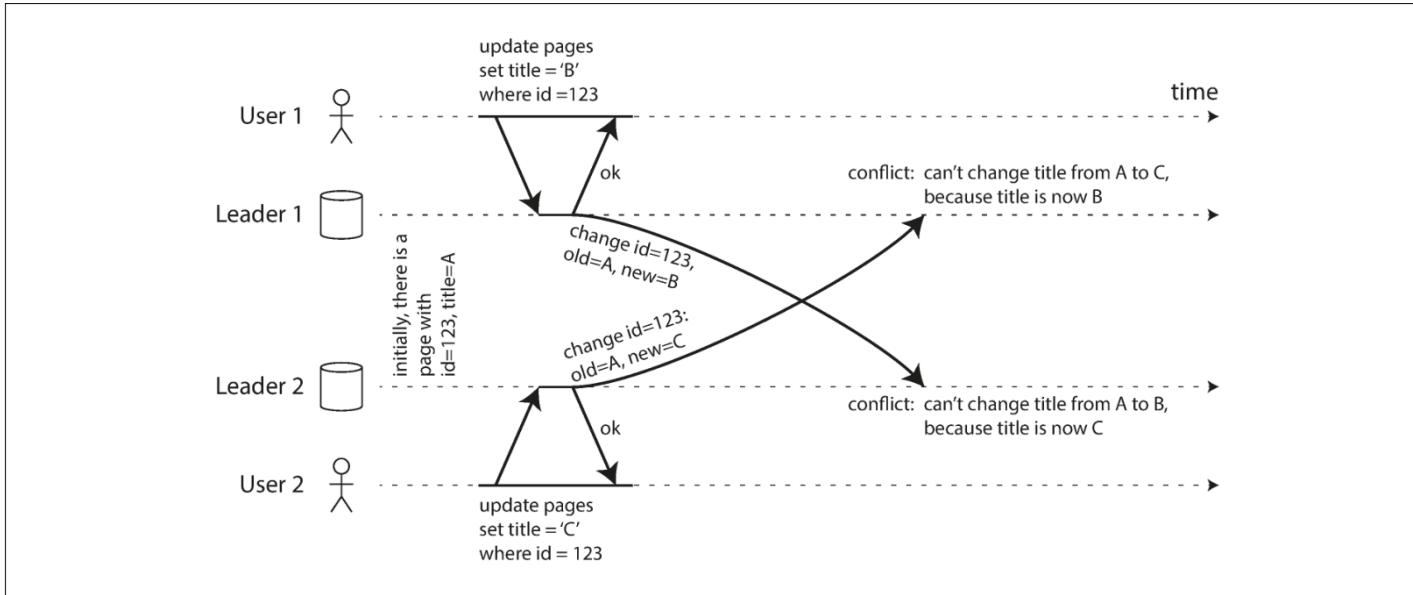


Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.

Handling write conflicts

- Synchronous versus asynchronous conflict detection
- Conflict avoidance: Allow updates only at one site for each data item
- Converging to a consistent state:
- There is no «correct» values when you have write conflicts, however, consistent values are the goal:
 - Each write has a unique ID, e.g., a timestamp, last write wins (lww)
 - Each replica has a unique ID, and higher ID “wins”
 - Merge values and keep all of them (B and C in the example)
 - Apply conflict resolution to them (automatic or manual)

Conflict resolution

- *On write:* As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler.
- *On read:* When a conflict is detected, all the conflicting writes are stored. All are read and a flag is set and the user resolves and writes back the correct value (CouchDB)
- *Conflict-free replicated datatypes* (CRDTs): Standard data structures to be used concurrently without locking
- *Mergeable persistent data structures:* Utilizes history and uses 3-way merge
- *Operational transformation:* Concurrent editing of documents (Google docs)

Multi-Leader Replication Topologies

- A replication topology describes the communication paths along which writes are propagated
- Stop propagating a message when it is received at the sender

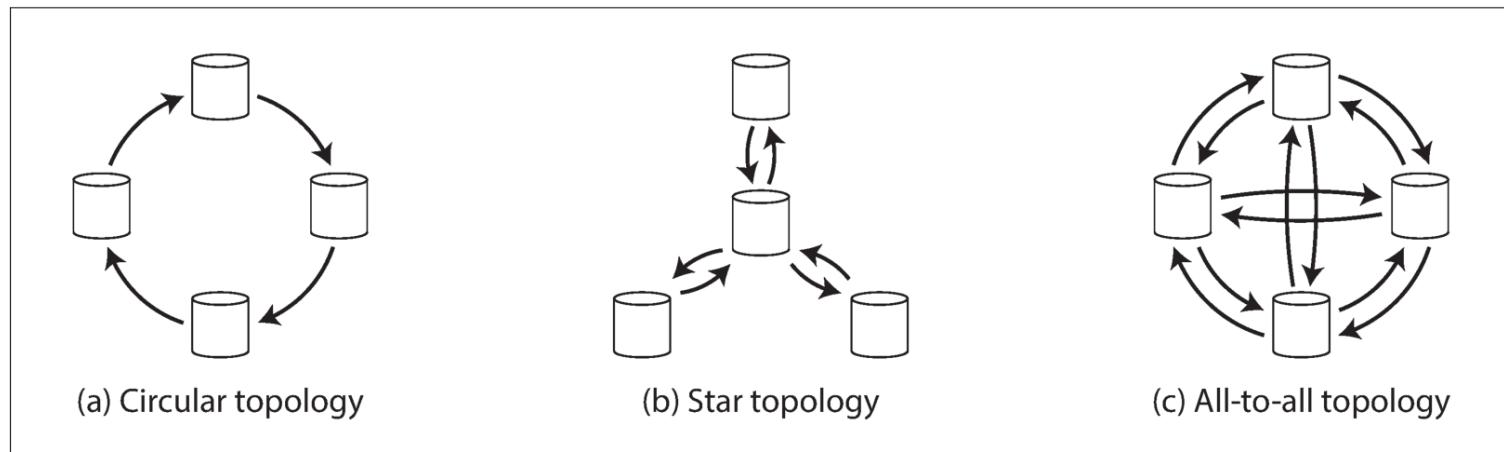


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

Multi-Leader Replication Topologies (2)

- Wrong order of messages received
- May be «solved» by version vectors

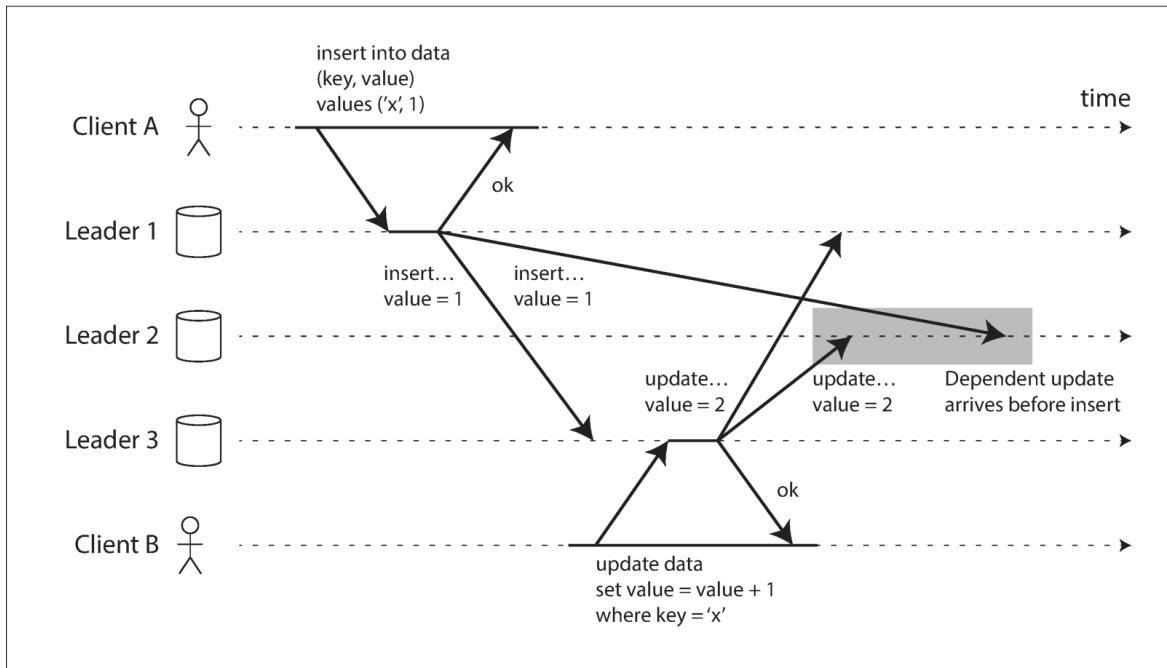


Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.

Leaderless replication

- Dynamo-style of replication
- Quorums, read repair and anti-entropy

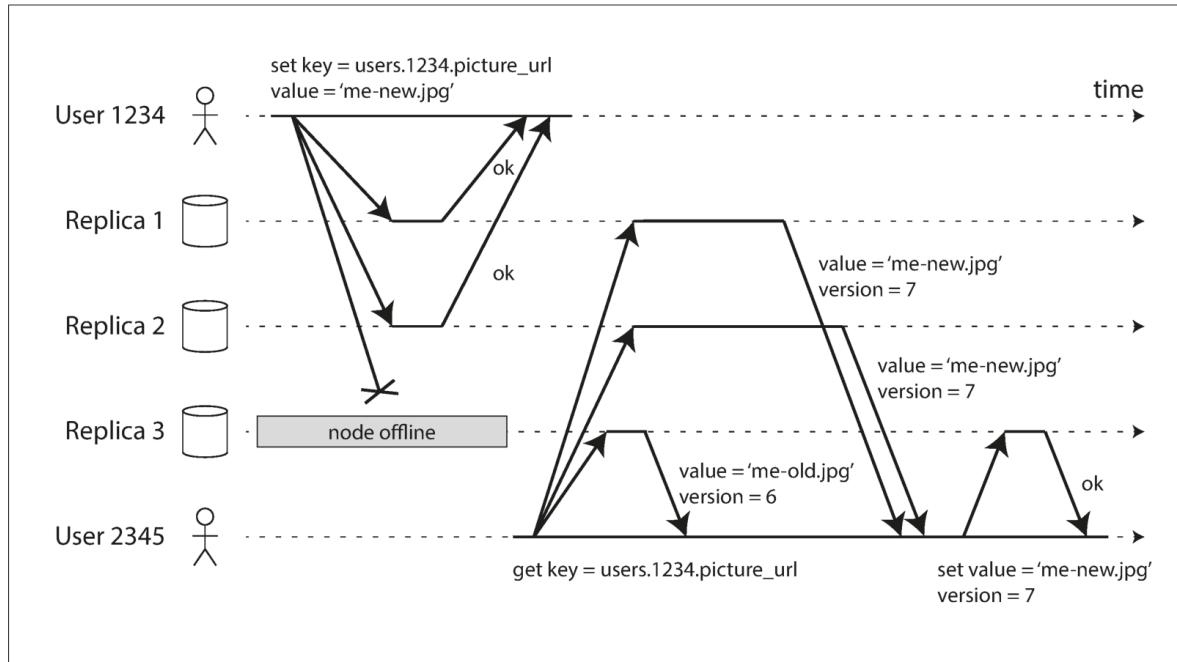


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

Quorums

- n replicas, every write must be confirmed by w nodes
- Query at least r nodes for each read
- As long as $w + r > n$, we expect to get an up-to-date value
- Dynamo-style databases, the parameters n , w , and r are typically configurable
- $w = r = (n + 1) / 2$ (rounded up): Common approach
- $w = n$ and $r = 1$: Fast reads but all nodes are written

Quorums (2)

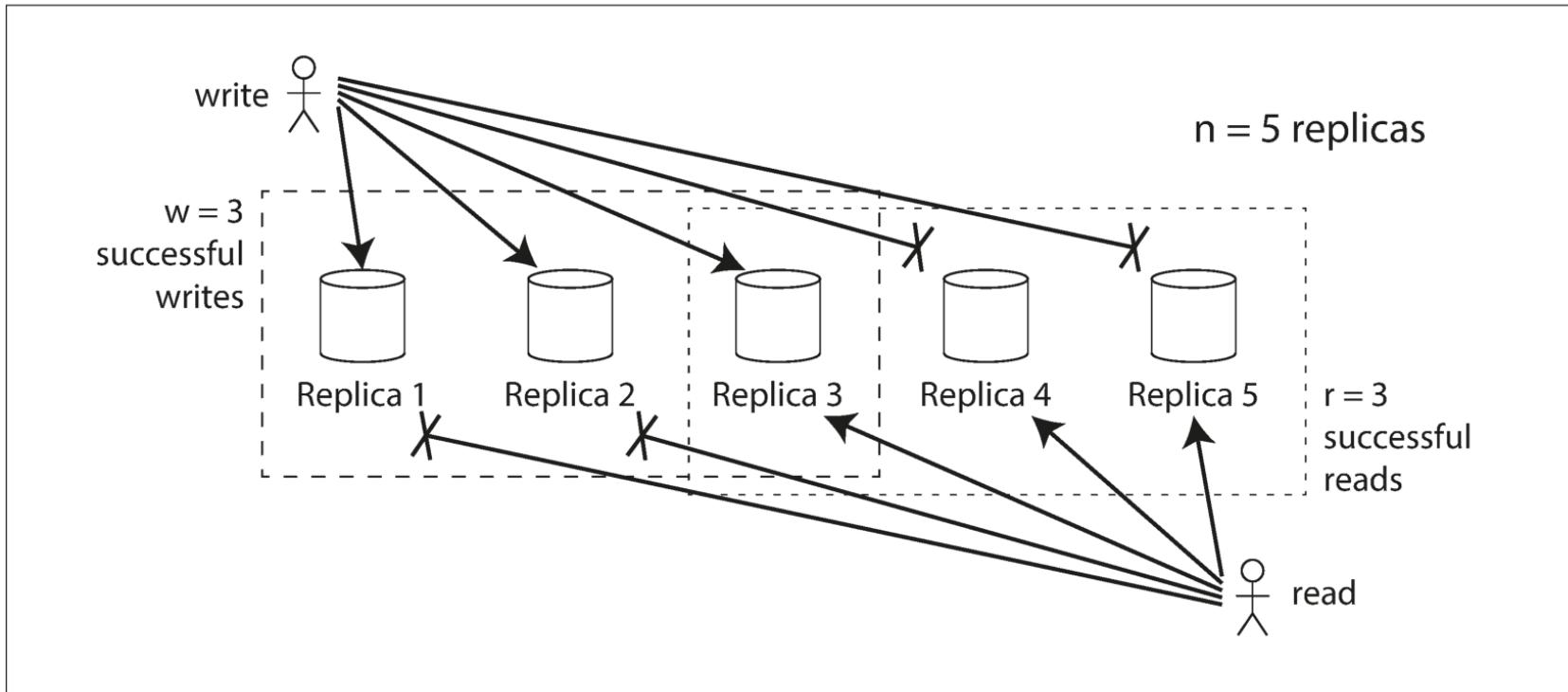


Figure 5-11. If $w + r > n$, at least one of the r replicas you read from must have seen the most recent successful write.

Quoroms (3)

- $W + R < N$ may also be valid, giving better response times and less overhead, but you may read stale values
- Sloppy quoroms may give that read and write quoroms may be different nodes (no overlap)
- Concurrent writes may have ordering problems, they may be in conflict. Merge conflicts? Last-Write-Wins?
- Concurrent writes and reads may return new or old values
- It should be a pre-defined order of the replicas, primary-backup1-backup2 etc (not leaderless)
- Monitoring staleness? Quantify «eventual»?

Sloppy Quorums and Hinted Handoff

- Dynamo-style systems, with some unreachable nodes
- We should accept writes anyway, and write them to some nodes that are reachable but aren't among the n nodes on which the value usually lives.
- Writes and reads still require w and r successful responses, but those may include nodes that are not among the designated n “home” nodes for a value
- A sloppy quorum is a durability assurance
- Sloppy quoroms are optional in Riak, Voldemort and Cassandra

Multi-datacenter support

- Multi-leader replication is used in multi-datacenters.
Why?
- Cassandra and Voldemort: Writes to all replicas, but awaits only response from local replicas
- Riak replicates locally, but uses asynch replication for multi-datacenters

Detecting Concurrent Writes

- Problem: Events may arrive in a different order at different nodes, due to variable network delays and partial failures.

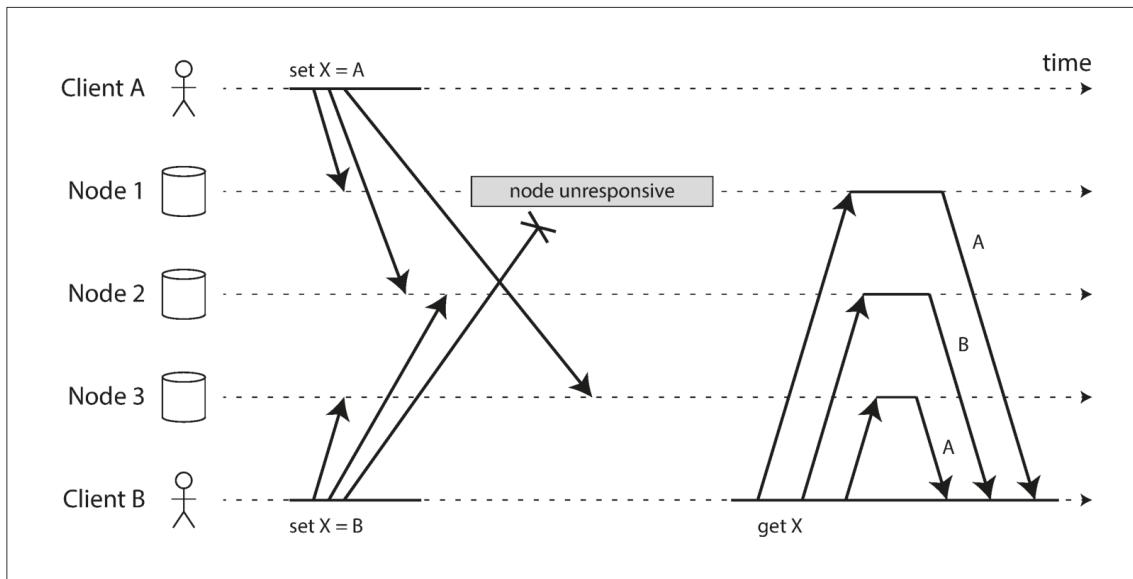


Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

Solutions to concurrent writes

- Last Write Wins - LWW
- Some way of unambiguously determining which write is more “recent”
- We can attach a timestamp to each write, pick the biggest timestamp as the most “recent,” and discard any writes with an earlier timestamp.

Causality: Happens before and concurrent

- An operation B depends on another operation A, when the user/transaction has read A's value.
- A **happens-before** B, or B is causally dependent on A.
- An operation A happens before another operation B if B knows about A, or depends on A, or builds upon A in some way.
- Two operations are **concurrent** if neither happens before the other.
- Exact time doesn't matter: we simply call two operations concurrent if they are both unaware of each other

Capturing the happens-before relationship

- Single-node case: Two clients adding items to the same shopping cart

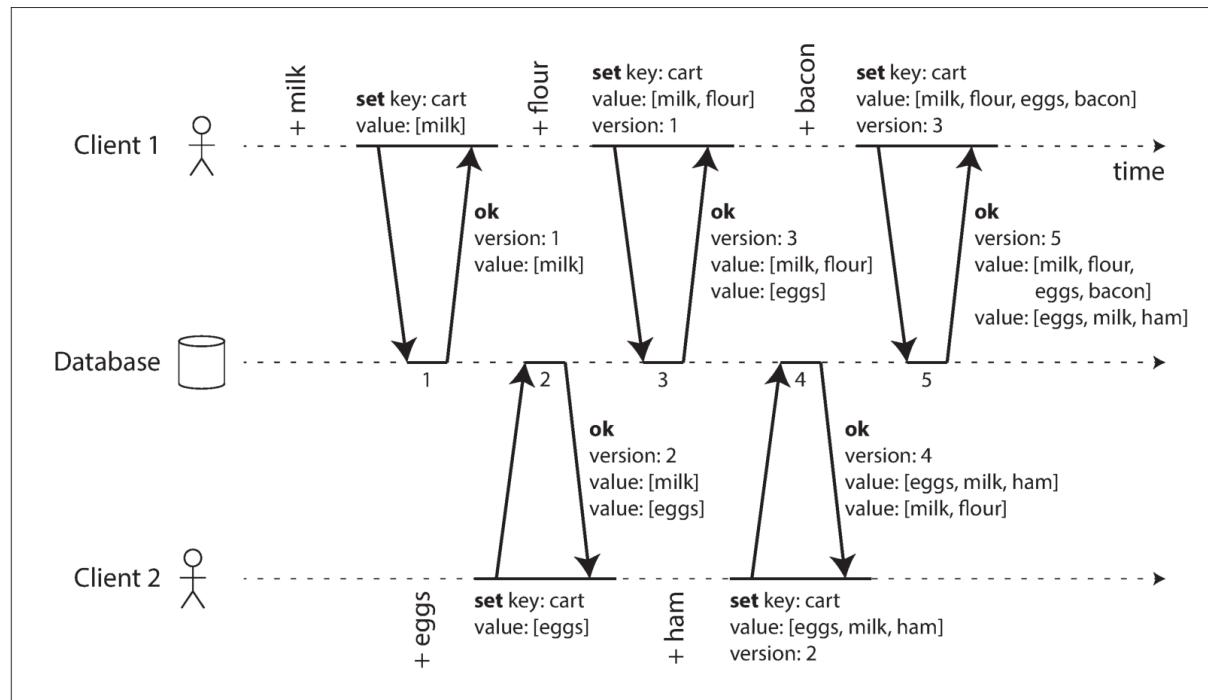


Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.

Dataflow in the example

- The database returns multiple values (*siblings*), which the client may merge in.

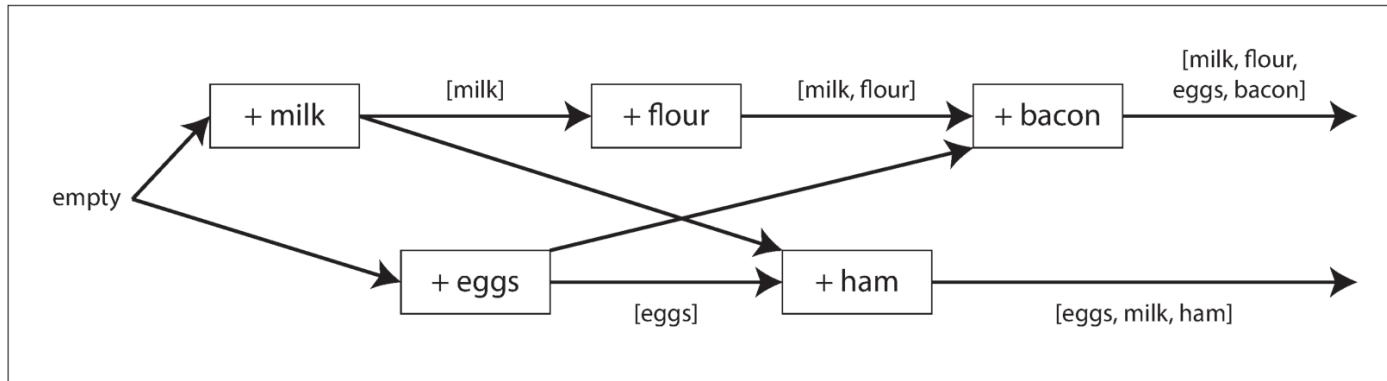


Figure 5-14. Graph of causal dependencies in Figure 5-13.

- Must use *tombstones* for deletes
- Version vectors* (*multiple replicas*) and *vector clocks* (*multiple nodes*)

Algorithm to decide overwrite or «concurrent»

- Server maintains version number for every key, increments version number at writes
- At read, the version number is returned. A client must read before writing.
- When writing, it must include the version number of the read.
- When the server receives a write with a (read) version number, it can overwrite all values with that version number and lower.
But it must keep all values with a higher version number (they are «concurrent»).

Summary – Chapter 5

- Why replication?
 - High availability
 - Disconnected operation
 - Latency
 - Scalability
- Types of leadership
 - Single-leader replication
 - Multi-leader replication
 - Leaderless replication
- Types of replication
 - Synchronous
 - Asynchronous

TDT4225

Chapter 6 – Partitioning

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Partitioning concepts

- For very large datasets, or very high query throughput:
- We need to break the data up into **partitions**
 - Shards (MongoDB, ElasticSearch)
 - Region (HBase)
 - Vnode (Cassandra, Riak)
 - vBucket (CouchBase)
- **Scalability** is the main reason for partitioning

Query execution and Partitioning

- For single partition queries (simple queries), each node can independently execute the queries
- Large, complex queries can potentially be parallelized across many nodes, and by shipping SQL and/or data across the network (**data shipping vs function shipping**)
- TerraData and Tandem NonStop SQL pioneered some of this
- HyperKuben and Clustra came from NTNU and used hash partitioning (and replication)

Partitioning and Replication

- Also called *declustering*

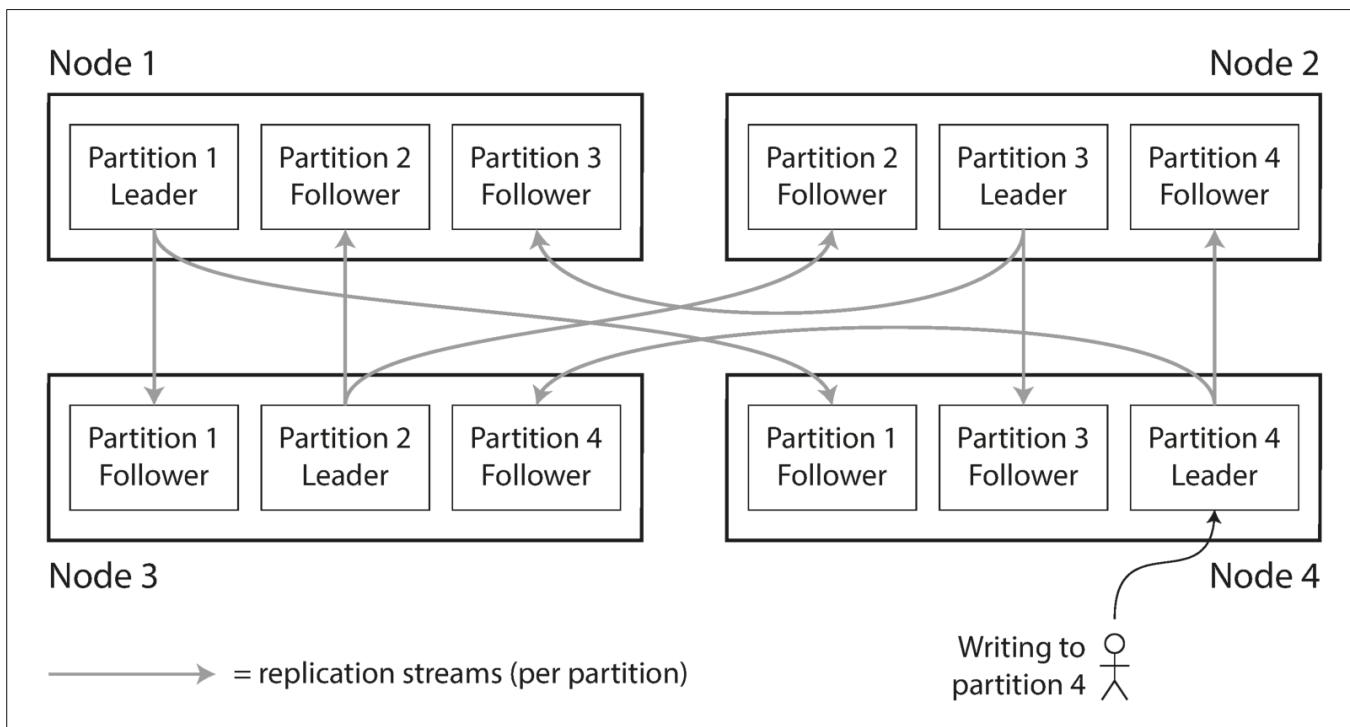


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

Range Partitioning

- Partitioned by the order of the search key

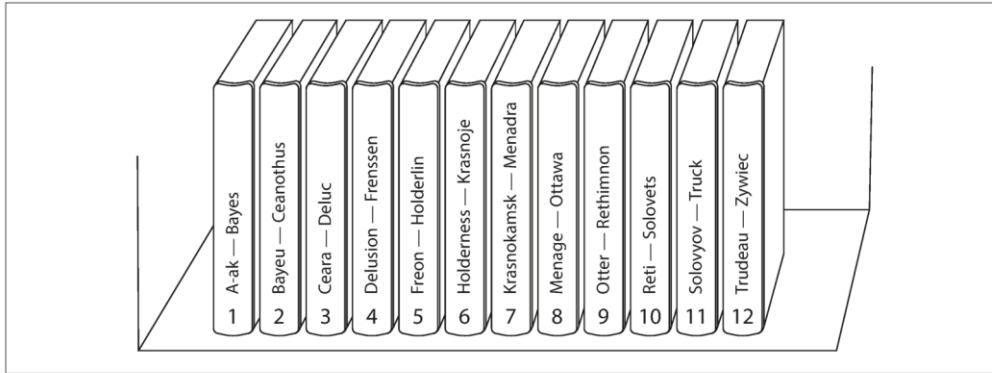


Figure 6-2. A print encyclopedia is partitioned by key range.

- May lead to some hot spots (e.g. newest items)
- Repartitioning may be necessary
- Don't use timestamps as first key, e.g., use «sensor name» (example in text book)

```

algorithm fnv-1 is
    hash := FNV_offset_basis do

        for each byte_of_data to be hashed
            hash := hash × FNV_prime
            hash := hash XOR byte_of_data

    return hash

```

Hash Partitioning

- A good hash function takes skewed data and makes it uniformly distributed
 - Cassandra and MongoDB use MD5
 - Voldemort uses the Fowler–Noll–Vo function
- Gives good load balance, but no range scans. You have to do range scans on all partitions (MongoDB)
 - Cassandra may use *compound keys*, where only the first part is hashed

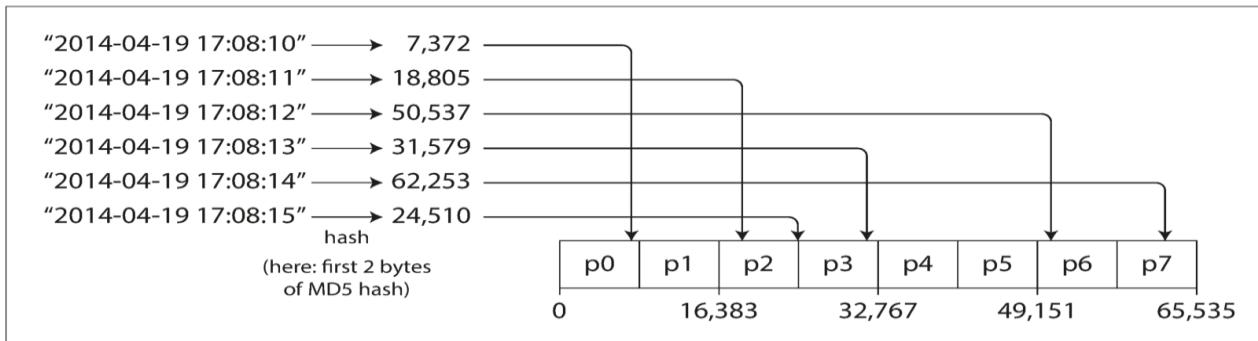


Figure 6-3. Partitioning by hash of key.

Skewed Workloads and Hot Spots

- Sometimes hashing doesn't load balance well
- All writes happen to the same key being hashed
- A technique to solve this is to let the application use an additional random key to the start or the end of the shared key
- It is also possible to use a different hash function if the current hash function doesn't work very well.

Document-Based Indexing

- Document-based indexing is also called *local indexing*
- Used in MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud, and VoltDB

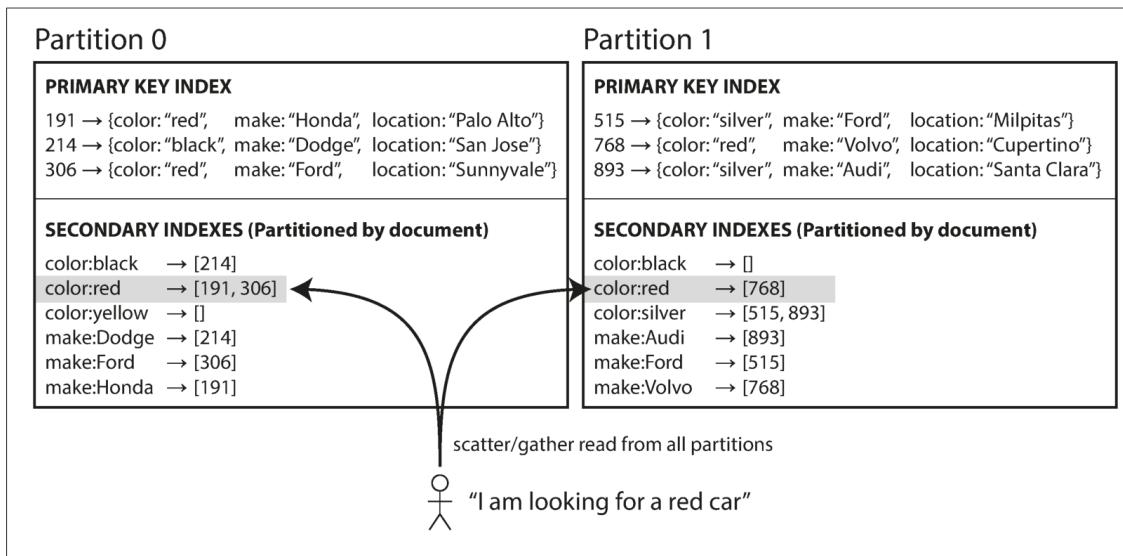


Figure 6-4. Partitioning secondary indexes by document.

Term-Based Indexing

- Term-Based Indexing (term partitioned) is also called *global indexing*
- Reads may become faster, but writes may become slower (they're not local). Updates may be done asynchronously.

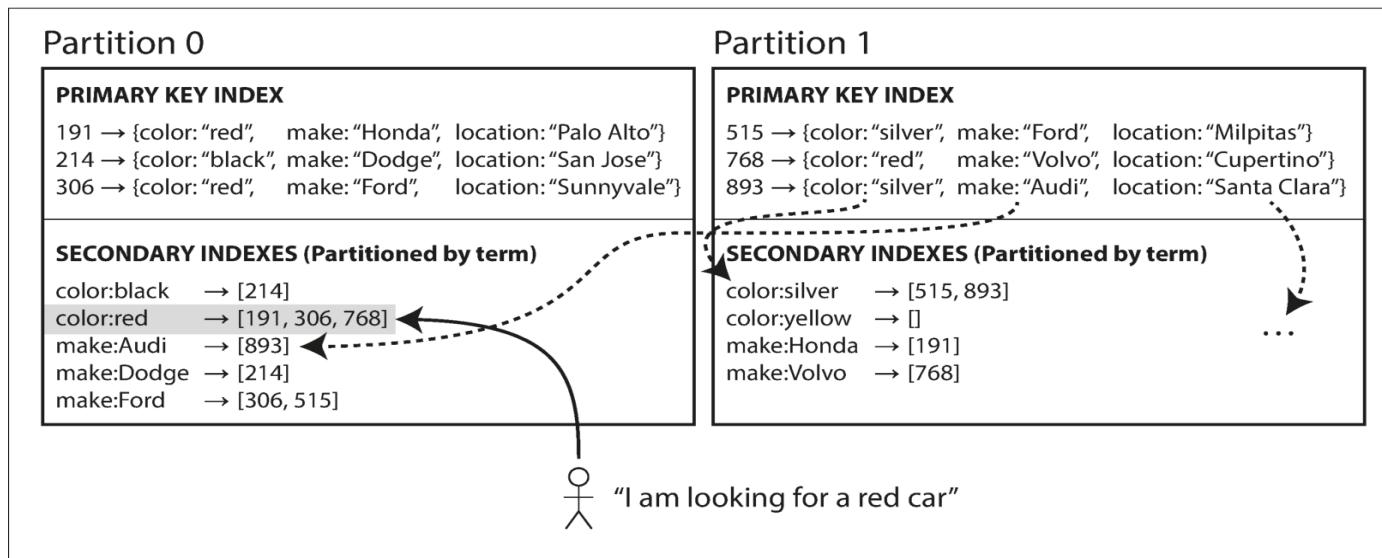
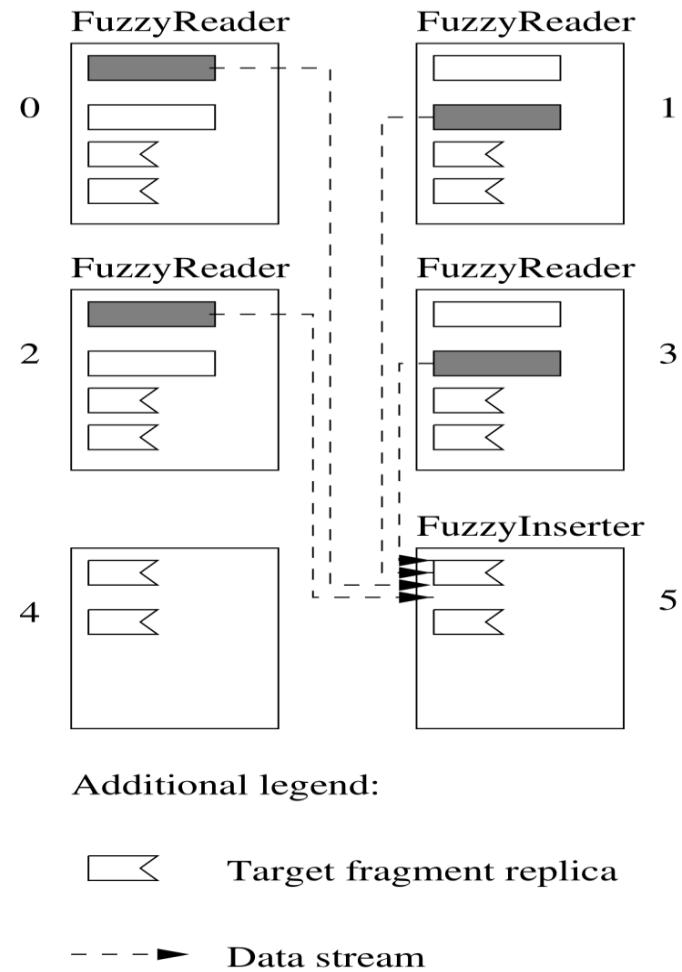


Figure 6-5. Partitioning secondary indexes by term.

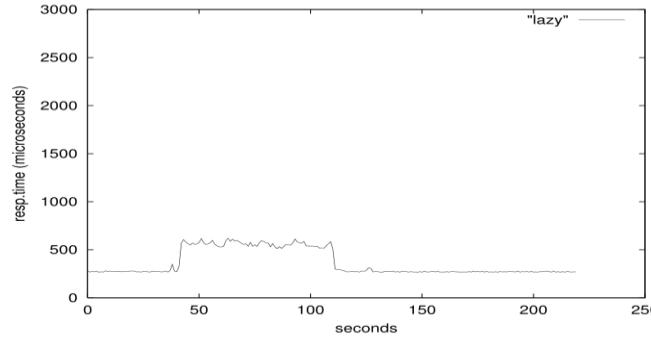
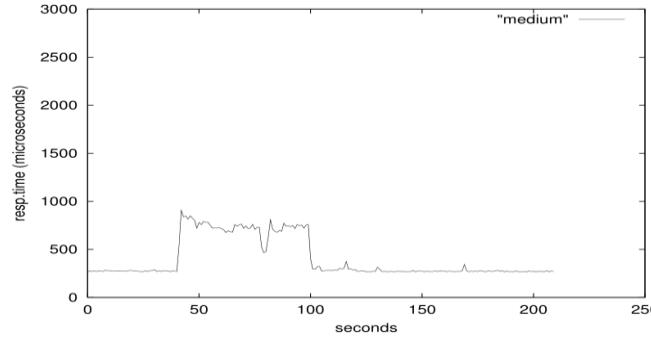
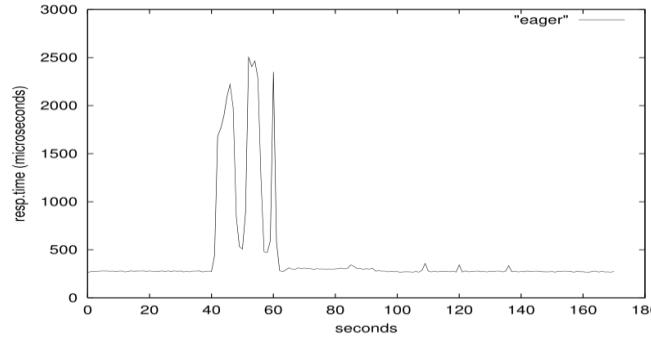
Rebalancing Partitions

- Reasons for rebalancing / repartitioning
 - Increased load
 - Increased data volume
 - Failed nodes need to be replaced
- Hashing with mod N is not a good choice
 - Need to move most records
 - All to all copy
 - Done in Clustra



Rebalancing partitions (2)

- Done in Clustra
- Will slow down concurrent queries



Rebalancing Partitions (3)

- Use a fixed, high number of partitions (Riak, Voldemort, Elasticsearch, Couchbase, ...)

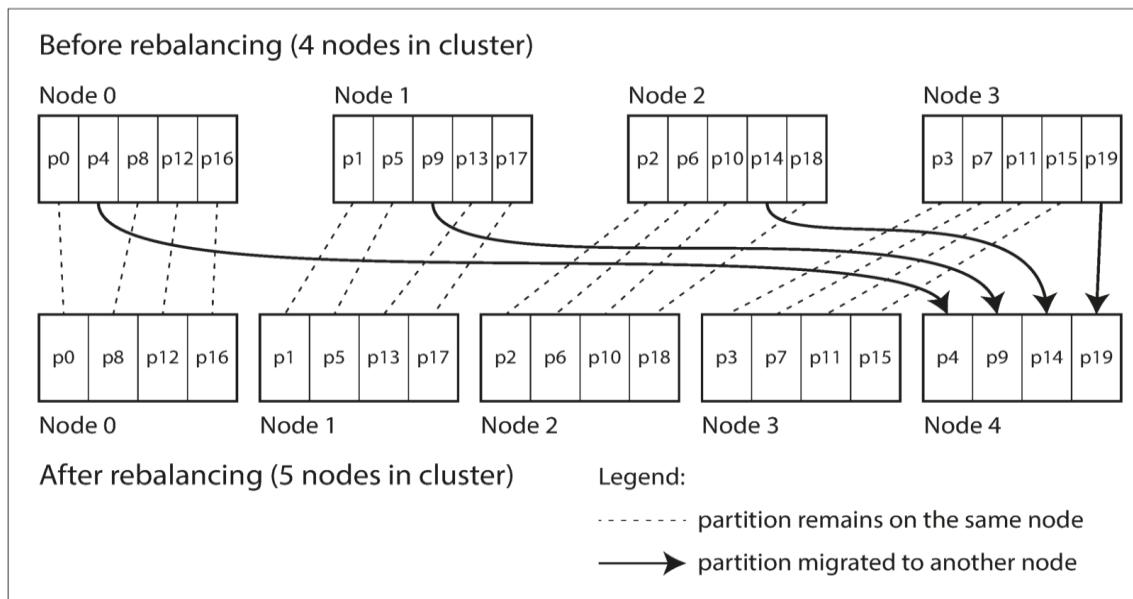


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

Dynamic Partitioning (range part.)

- When a partition grows to exceed a configured size, it is split into two partitions
- HBase and RethinkDB create partitions dynamically
- HBase and MongoDB allow an initial set of partitions to be configured on an empty database (*pre-splitting*)
- May also be used on hash partitioning (MongoDB)
- Proportional to the size of data volume
- Proportional to the size of the number of nodes (Cassandra and Ketama)
- Operations: Automatic or manual rebalancing

Request Routing

- Three different approaches
 1. Allow clients to contact any node
 2. Send all requests from clients to a routing tier first
 3. Require that clients be aware of the partitioning and the assignment of partitions

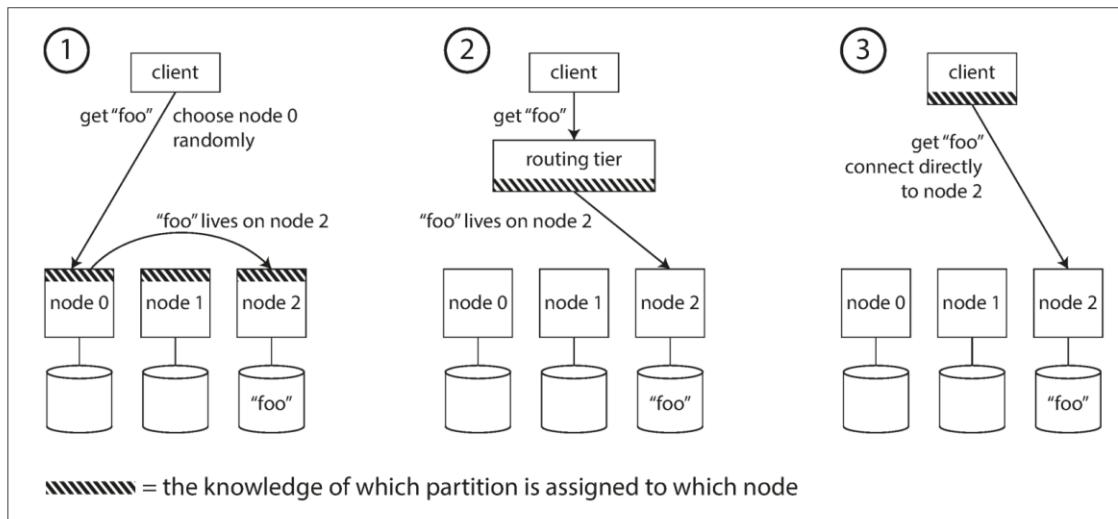


Figure 6-7. Three different ways of routing a request to the right node.

Coordination of services

- How do clients know about changes?
- ZooKeeper

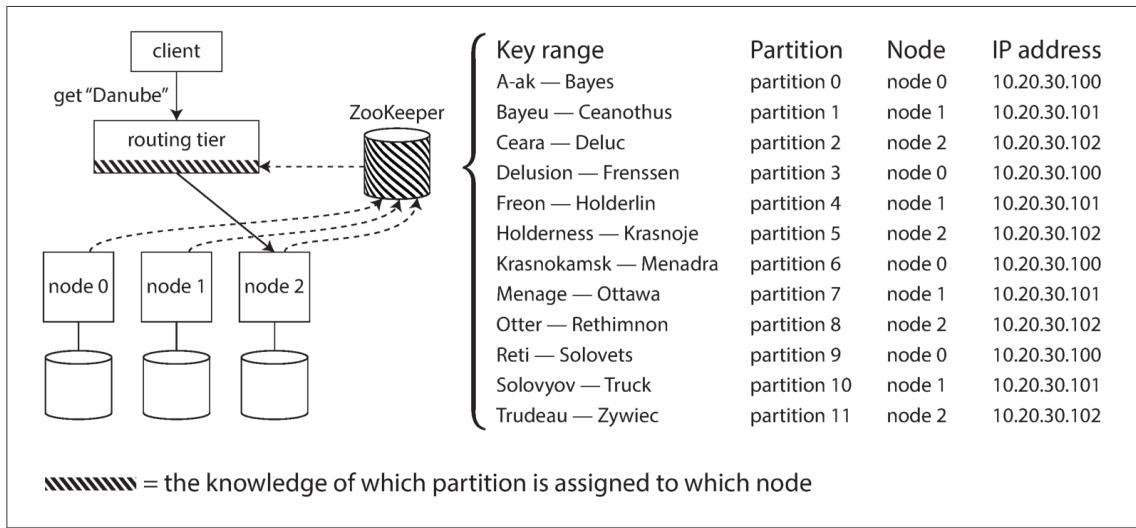


Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

- Gossip protocol (Cassandra and Riak)
- Virtual partition protocol (UCSB) used in Clustra

Summary

- Hash partitioning
- Key range partitioning
- Document-partitioned indexes (local indexes)
- Term-partitioned indexes (global indexes)
- Rebalancing
- Request routing
- Coordination of services

TDT4225

Chapter 7 – Transactions

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Transactions are necessary because

- The database software or hardware may fail at any time
- The application may crash at any time
- Interruptions in the network cause apps to be lost from db
- Several clients may have concurrent, conflicting writes
- A client may read data that doesn't make sense due to partial updates
- Race conditions between clients can cause surprising bugs

Transactions

- Used to group several reads and writes together into a logical unit
- Commits or aborts as a unit
- May be retried in case of abort
- Used to simplify the programming model for applications accessing a database
- Some applications don't need transactions
- Concepts to learn: Read committed, snapshot isolation, and serializability.

History of transactions

- Started with System R (IBM) in the 70s
- Supported almost without change since then in SQL databases (MySQL, PostgreSQL, Oracle, SQL Server, DB2)
- NoSQL skipped transactions
 - Partitioning / Sharding
 - Replication
 - CRUD
- NewSQL added transactions to NoSQL (Spanner, CockroachDB, YugabyteDB, TiDB)

ACID – properties of a transaction

Transaction: A sequence of operations on DB which are

- **A** – atomic: completely run or not
- **C** – consistency: (primary key, references, check, etc)
- **I** – isolation: does not notice other transactions.
- **D** – durability: nothing lost after commit.

A transaction is usually a logical operation or task.

Kleppmann: The high-level idea is sound, but the devil is in the details. Today, when a system claims to be “ACID compliant,” it’s unclear what guarantees you can actually expect. ACID has unfortunately become mostly a marketing term.

- BASE is «not ACID»

Atomicity

- Atomic refers to something that cannot be broken down into smaller parts.
- In ACID, atomicity is not about concurrency
- ACID atomicity describes what happens if a client wants to make several writes (and reads)
- Atomicity simplifies the problem of half-done operations: If a transaction was aborted, the application can be sure that it didn't change anything, so it can safely be retried.
- *Kleppmann*: Abortability is a better word.

Consistency

- Consistency is an overloaded word.
- In ACID, consistency refers to an application-specific notion of the database being in a “good state.”
- Consistency depends on the application’s notion of invariants
- Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application. The application may rely on the database’s atomicity and isolation properties in order to achieve consistency, but it’s not up to the database alone. Thus, the letter C doesn’t really belong in ACID. (Kleppmann)

Isolation

- Isolation in ACID means that concurrently executing transactions are isolated from each other.
- Serializability: Each transaction can pretend that it is the only transaction running on the entire database.
- The database ensures that when the transactions have committed, the result is the same as if they had run serially

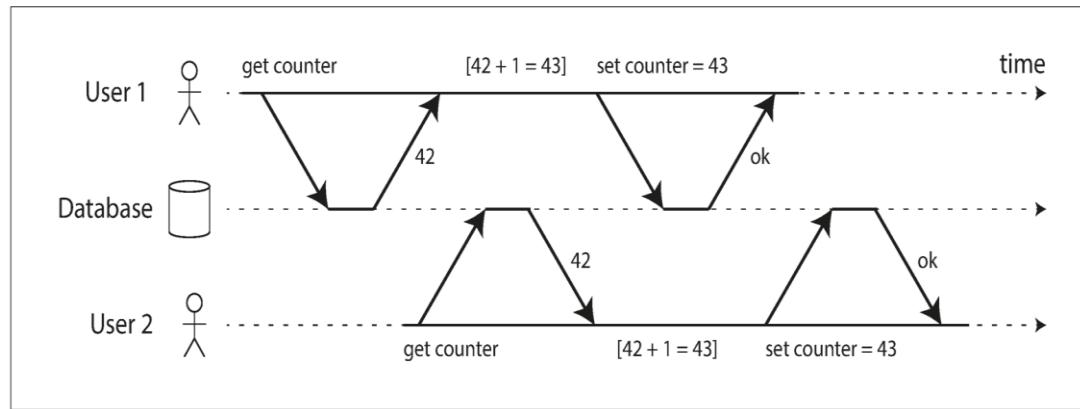


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

Durability

- Durability is the promise that once a transaction has committed successfully, any data it has written will not be forgotten.
- Usually done by WAL – Write Ahead Logging (force-log-at-commit). Some oldtype databases forces data to disk.
- Force/Steal classification (Theo Harder)
- Force data or force redo log
- Steal slot in buffer (force undo log) or no-steal

Single-Object and Multi-Object Operations

- Multiple writes could be rolled back (atomicity)
- Concurrent transactions may see all writes or none (isolation)
- `SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND unread_flag = true`
- Slow, so use a cached counter: Anomaly may appear

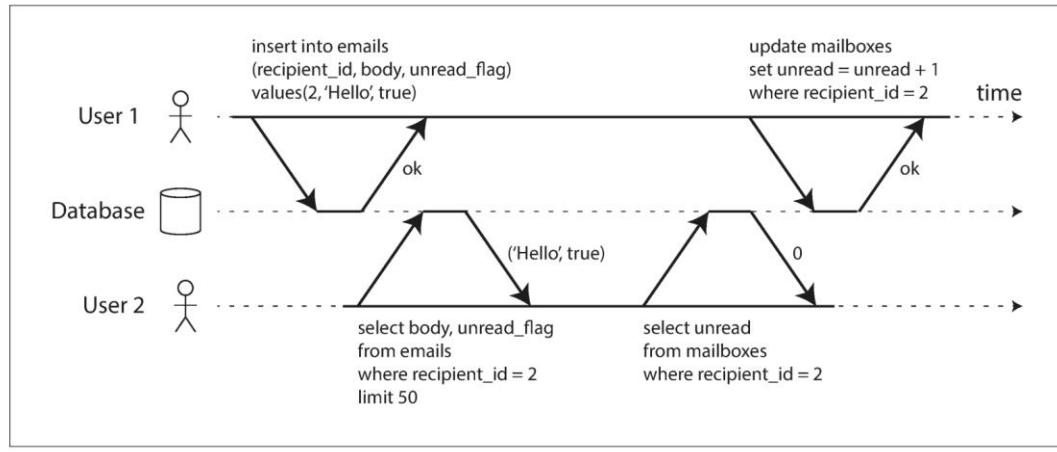


Figure 7-2. Violating isolation: one transaction reads another transaction's uncommitted writes (a “dirty read”).

Transactions and Atomicity

- If an error occurs somewhere over the course of the transaction: Abort.
- NoSQL databases usually don't support this

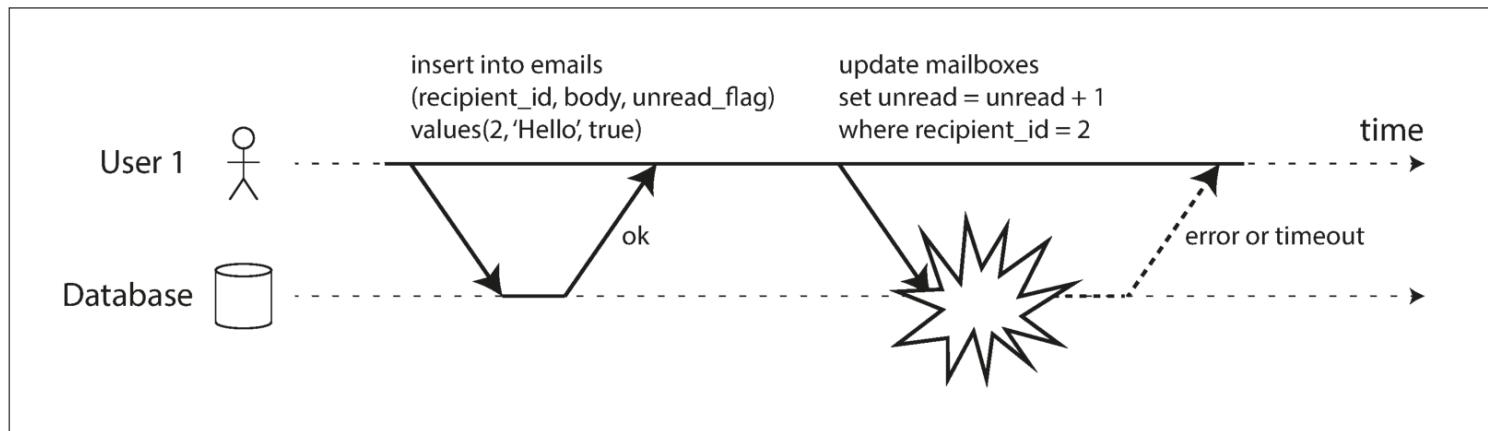


Figure 7-3. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid an inconsistent state.

Single Object Writes

- All databases try to make single object writes atomic.
- E.g. writing large JSON strings may fail halfway
- Use Logging & concurrency control
- Support increment instead of using read-modify-write cycle (some DBs do)
- Compare-and-set and other single-object operations have been dubbed “lightweight transactions” or even “ACID” for marketing purposes
- A transaction is usually understood as a mechanism for grouping multiple operations on multiple objects into one unit of execution.

The need for multi-object transactions

- Rows may contain foreign keys, thus need to update several rows in different tables
- Document databases using normalization may have the same problem, documents referencing other documents
- Secondary indexes require multiple writes belonging to the same logical operation
- Operations belonging to the same logical task (of course)

Handling errors and aborts

- *Leaderless replication*: “the database will do as much as it can, and if it runs into an error, it won’t undo something it has already done”—so it’s the application’s responsibility to recover from errors”
- Retries are not always good:
 - The transaction succeeded, but the network failed
 - Overload: retrying the transaction will make the problem worse
 - It is only worth retrying after transient errors (deadlocks, concurrency, etc)
 - If the transaction also has side effects outside of the database, those side effects may happen even if the transaction is aborted
 - If the client process fails while retrying, any data it was trying to write to the database is lost

Weak Isolation Levels (1)

- Concurrency issues (race conditions) when one transaction reads data that is concurrently modified by another transaction
- Concurrency bugs are hard to find by testing
- Thus, transaction isolation: Transactions should feel that they are the sole users running.
- We need to develop a good understanding of the kinds of concurrency problems that exist, and how to prevent them.
- Isolation levels and their problems and properties follow
...

Read Committed

1. When reading from the database, you will only see data that has been committed (no *dirty reads*).
2. When writing to the database, you will only overwrite data that has been committed (no *dirty writes*).

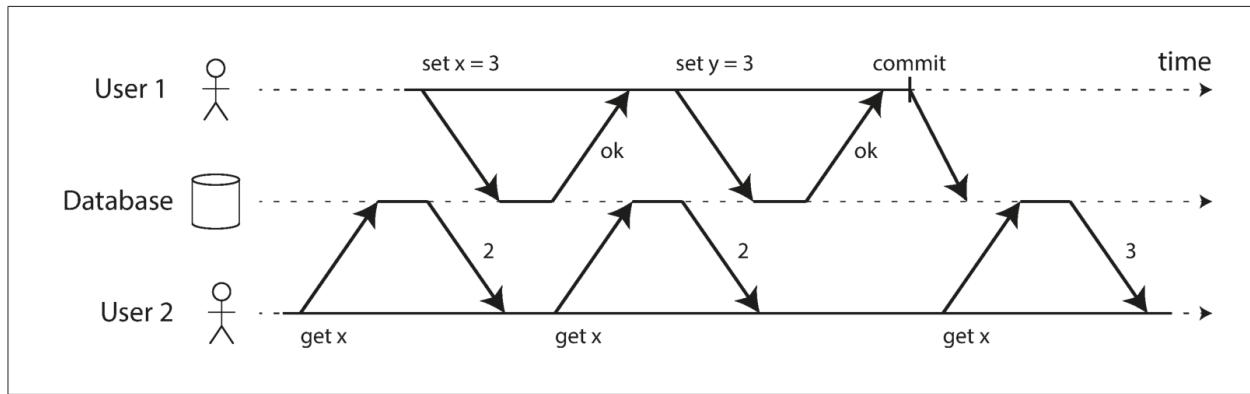


Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

Read Committed (2)

- If a transaction needs to update several objects, a dirty read means that another transaction may see some of the updates but not others
- If the database allows dirty reads, that means a transaction may see data that is later rolled back

No Dirty Writes

- The earlier write is part of a transaction that has not yet committed, so the later write overwrites an uncommitted value.

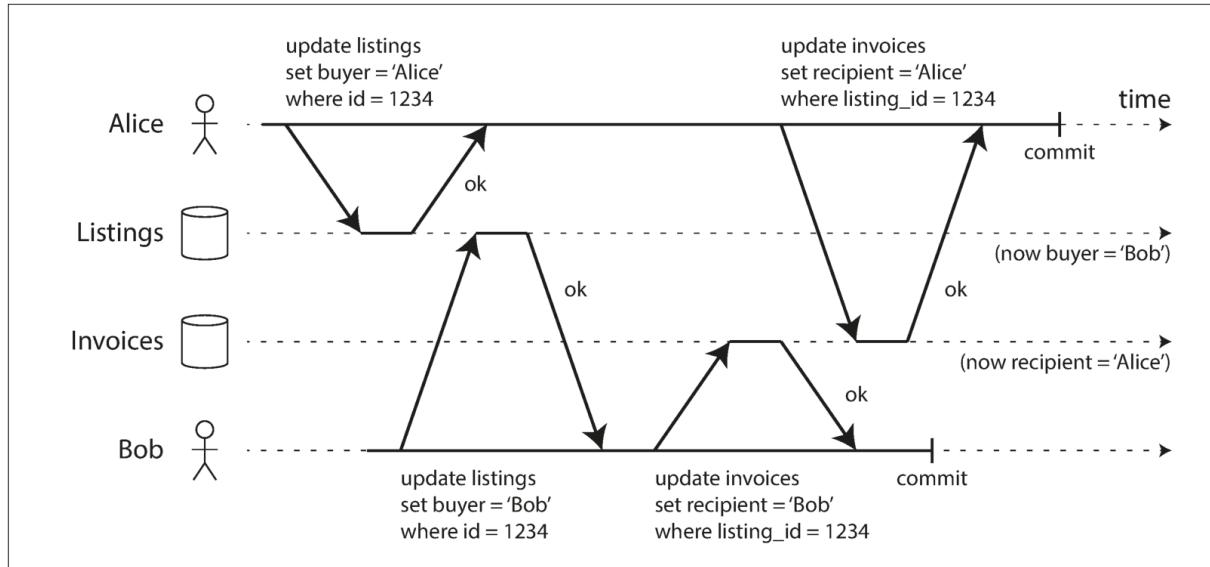


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Read Committed

- **Read committed** is a very popular isolation level. It is the default setting in Oracle 11g, PostgreSQL, SQL Server 2012, MemSQL, ...
- Most databases prevent dirty reads by keeping old values for writes until the transactions commit. Read transactions may read the old value.
- Only when the new value is committed do transactions switch over to reading the new value.
- To keep single record locks would cost too much, since one writer may cause multiple readers to wait

Snapshot Isolation and Repeatable Read

- Problems with Read Committed: This anomaly is called a nonrepeatable read or read skew

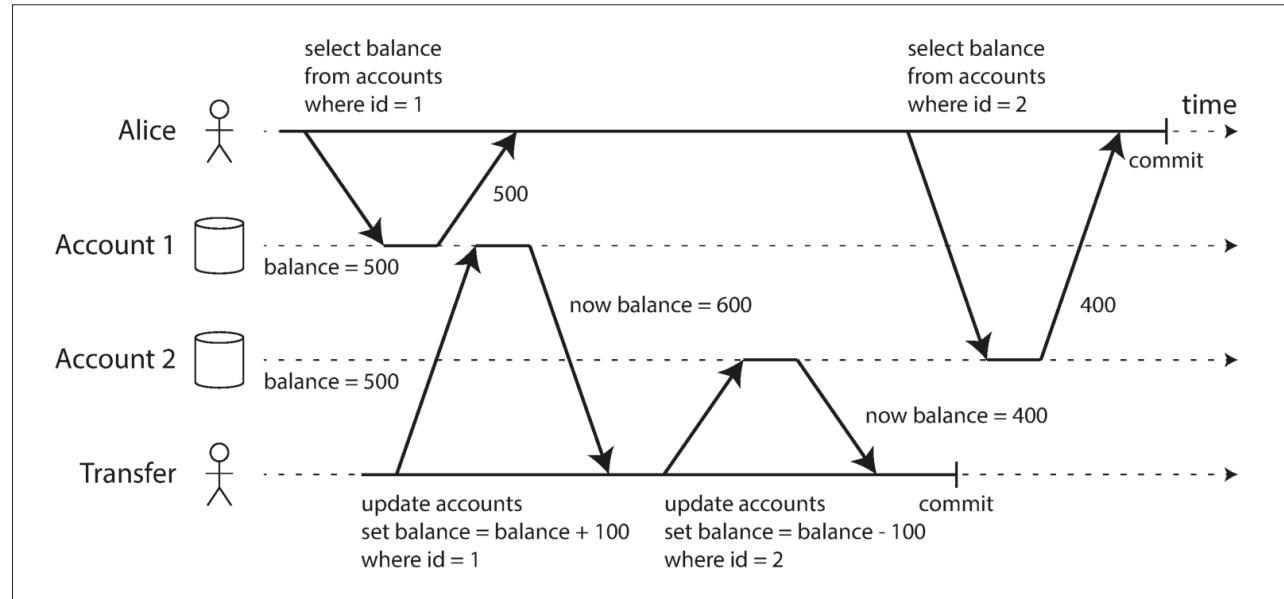


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

Snapshot Isolation and Repeatable Read (2)

- Snapshot isolation is the most common solution to this problem. Supported by PostgreSQL, MySQL/InnoDB, Oracle, SQL Server, and others.
- Use write locks, but reads do not require locks
- Readers never block writers, writers never block readers
- Multiversion concurrency control (MVCC). Store several versions of an updated record. One for each snapshot.
- Storage engines that support snapshot isolation typically use MVCC for their read committed isolation level.

PostgresSQL's MVCC

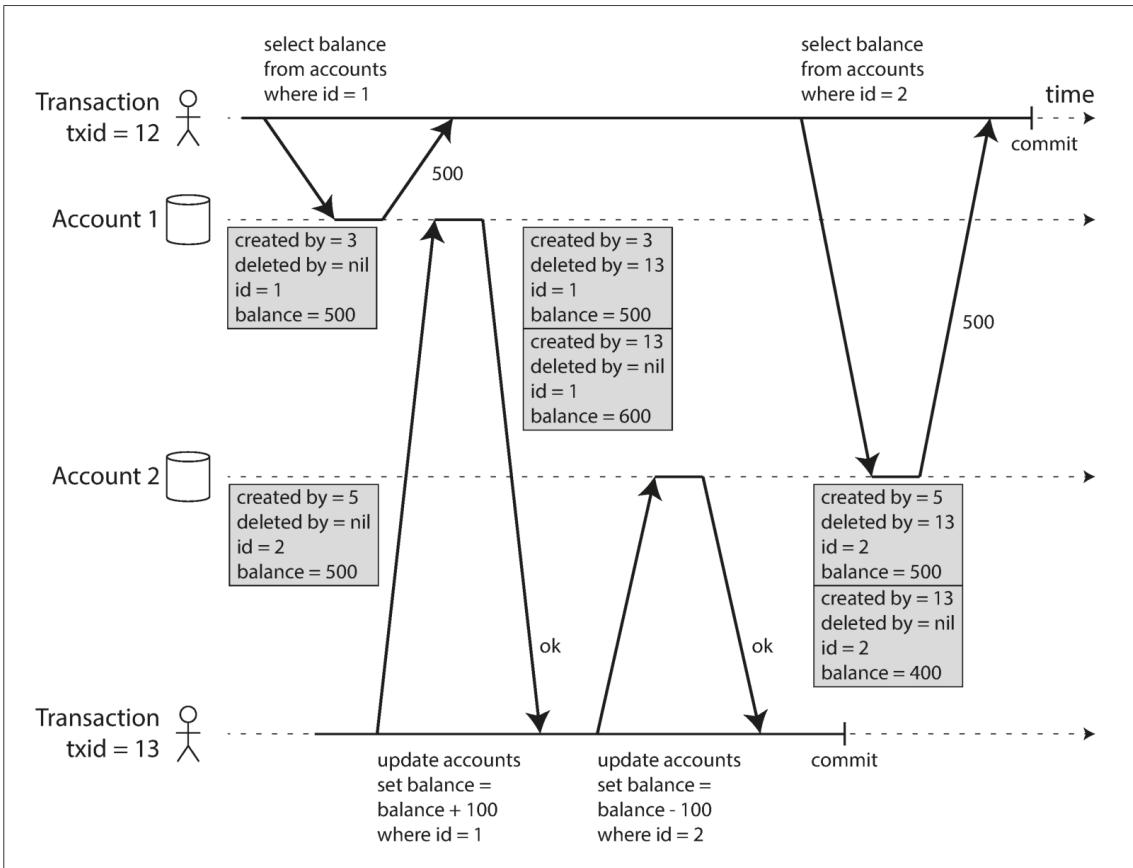


Figure 7-7. Implementing snapshot isolation using multi-version objects.

Visibility rules for observing a consistent snapshot

- TransactionIds are used to decide visibility of objects
 1. Existing, active transactions' writes are not visible to a new transaction
 2. Writes by aborted transactions are ignored
 3. Writes by newer transactions are not visible
 4. All other writes are visible
- Alternative rules (skipped)

MVCC, Indexes and COW-B+-trees

- Should the index point to all versions of an object?
- PostgreSQL has optimizations when the object versions are in the same page (avoids updates to index)
- CouchDB, Datomic, and LMDB: Copy-on-write B+-trees.
- All writes make copies of pages which are propagated up in the tree.
- And a new root is visible at commit
- Creates need for compaction and garbage collection
- BTRFS – File system on Linux based on copy-on-write B+-trees.

Repeatable Read and the SQL «standard»

- Real confusion on use of Repeatable Read in SQL databases
- Oracle: calls it Serializable
- PostgreSQL and MySQL calls snapshot isolation for Repeatable Read
- IBM DB2 uses «Repeatable Read» to refer to Serializability
- SQL standard based on System R's use of isolation levels, and snapshot isolation wasn't implemented at that time

Preventing Lost Updates

- Snapshot isolation: what a read-only transaction can see in the presence of concurrent writes
- Atomic write (update) operations:

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

- Implemented by exclusive lock
- Object-relational mapping frameworks make it easy to accidentally write code that performs unsafe read-modify-write cycles instead of using atomic operations provided by the database

Explicit locking

- SELECT FOR UPDATE provides explicit locking

Example 7-1. Explicitly locking rows to prevent lost updates

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
    FOR UPDATE; ①

-- Check whether move is valid, then update the position
-- of the piece that was returned by the previous SELECT.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

- PostgreSQL's repeatable read, Oracle's serializable, and SQL Server's snapshot isolation levels automatically detect when a lost update has occurred and abort the offending transaction.

Compare-and-Set

- Used by some databases without transactions, often called *lightweight transactions (LWT)*
- The purpose of this operation is to avoid lost updates by allowing an update to happen only if the value has not changed since you last read it.

-- This may or may not be safe, depending on the database implementation
`UPDATE wiki_pages SET content = 'new content'
WHERE id = 1234 AND content = 'old content';`

- If the content has changed and no longer matches 'old content' , this update will have no effect, so you need to check whether the update took effect and retry.

Conflict Resolution and Replication

- In multi-leader replication you may have conflicts when parallel updates happen
- Use application code to resolve conflicts
- You need to detect the conflicts and this must be provided by the database system using vector clocks or version vectors
- Last Write Wins (LWW) is the default solution used in many database systems

Write Skew and Phantoms

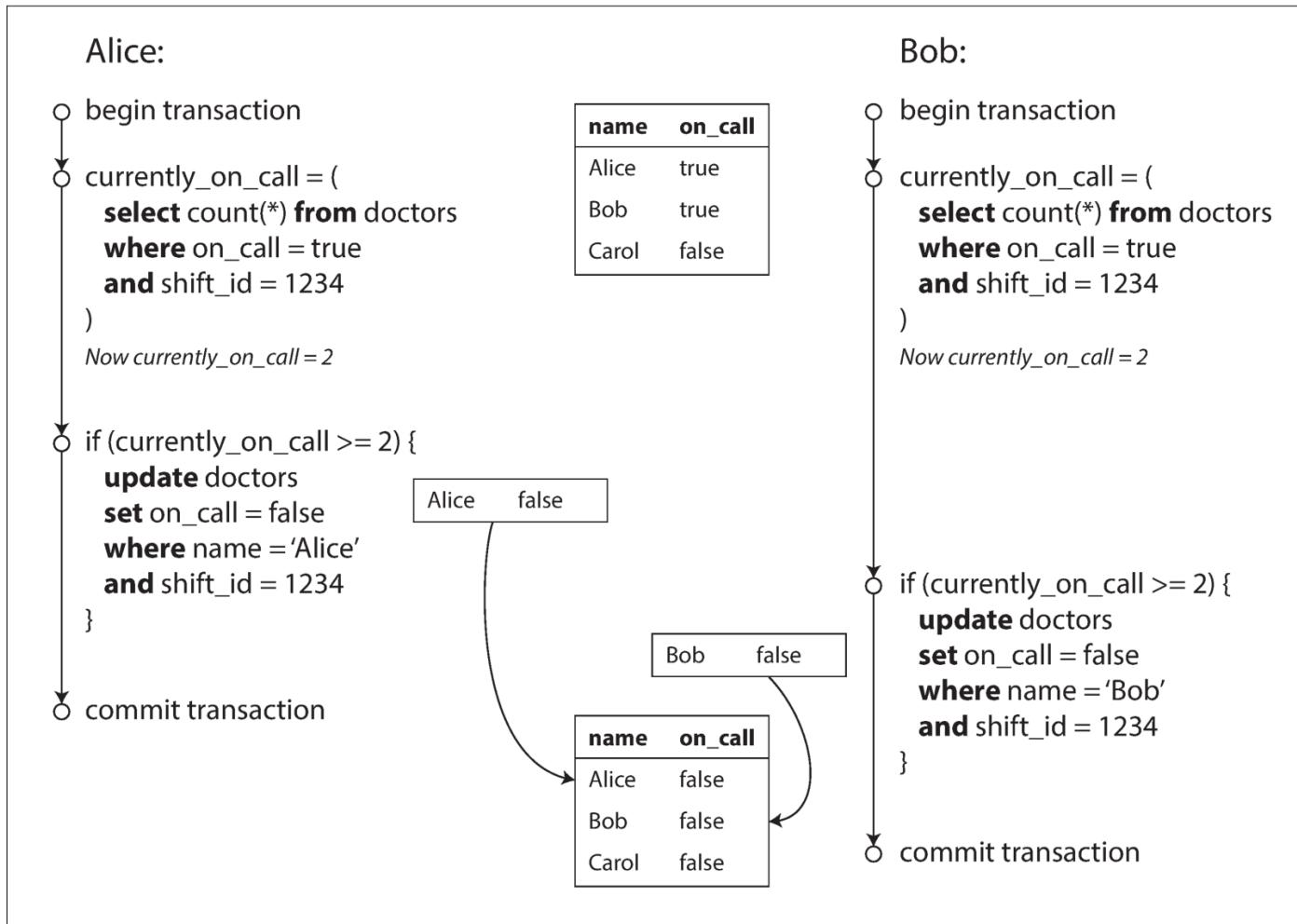


Figure 7-8. Example of write skew causing an application bug.

Write Skew and Phantoms (2)

- Write skew can occur if two transactions read the same objects, and then update some of those objects
- *Serializable isolation level* should solve this (treated later)
- Explicit locking:

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = true
    AND shift_id = 1234 FOR UPDATE; ①

UPDATE doctors
  SET on_call = false
  WHERE name = 'Alice'
    AND shift_id = 1234;

COMMIT;
```

As before, FOR UPDATE tells the database to lock all rows returned by this query.

Write Skew and Phantoms (3)

- Pattern for problem:
 1. A SELECT query checks some conditions
 2. Depending on the result of the first query, the application code decides how to continue
 3. If the application decides to go ahead, it makes a write to the database and commits the transaction.
- The effect, where a write in one transaction changes the result of a search query in another transaction, is called a phantom
- E.g. insert a new row into a table being counted by another transaction

Serializability (1)

- Isolation levels are hard to understand, and inconsistently implemented in different databases.
- If you look at your application code, it's difficult to tell whether it is safe to run at a particular isolation level.
- There are no good tools to help us detect race conditions.
- **Serializable isolation** is usually regarded as the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, serially, without any concurrency.

Serializability (2)

- How do databases provide serializability?
 1. Literally executing transactions in a serial order
 2. Two-phase locking
 3. Optimistic concurrency control techniques
- Why run serially? (VoltDB/H-Store, Redis, Datomic)
 1. RAM became cheap enough that for many use cases is now feasible to keep the entire active dataset in memory
 2. OLTP transactions are usually short and only make a small number of reads and writes

Serializability (3)

- Each transaction is through one HTTP request.
- Executed by a stored procedure
- No interaction

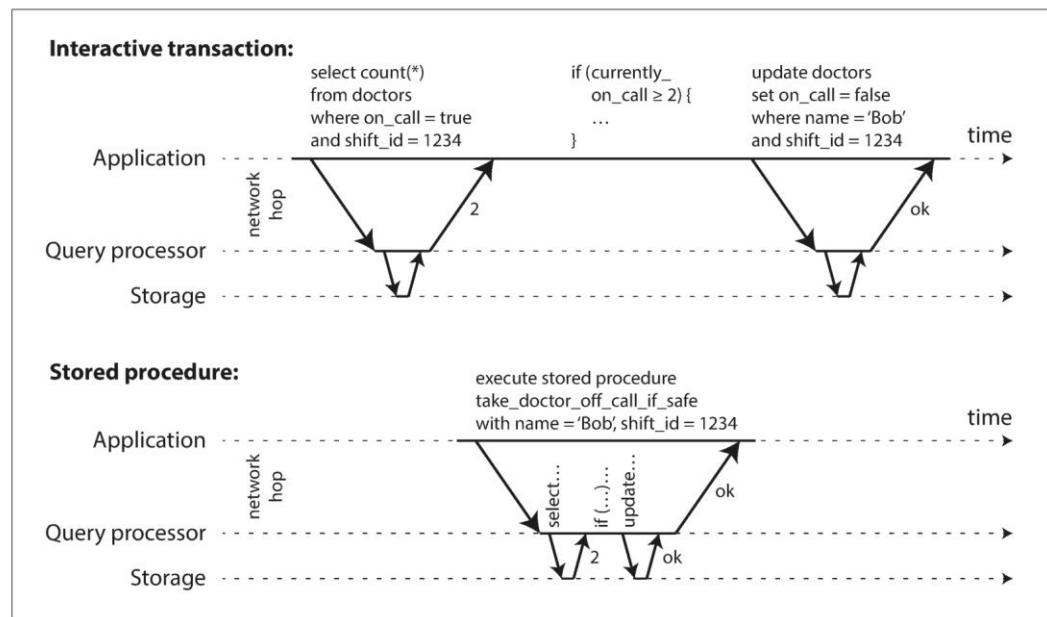


Figure 7-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 7-8).

Pros and cons of stored procedures

- Each database vendor has its own language for stored procedures
- Code running in a database is difficult to manage
- A database is often much more performance-sensitive than an application server
- Modern implementations of stored procedures have abandoned PL/SQL and use existing general-purpose programming languages instead (Java, Clojure, Lua, ..)
- VoltDB also uses stored procedures for replication
- Difficult to support this efficiently using partitioning. Geo-partitioning may be an exception.

Summary of serial execution

- Every transaction must be small and fast.
- The active dataset must fit in memory.
- Write throughput must be low enough to be handled on a single CPU core.
- Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.

Two-phase locking (2PL)

- Several transactions are allowed to concurrently read the same object as long as nobody is writing to it.
- If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue.
- If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue.
- 2PL is used by the serializable isolation level in MySQL (InnoDB) and SQL Server, and the repeatable read isolation level in DB2

Two-Phase Locking (2PL) (2)

- T wants to read an object: T must acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously.
- T wants to write to an object: T must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time.
- T first reads and then writes an object, it may upgrade its shared lock to an exclusive lock.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort).

Performance of two-phase locking

- Overhead of acquiring and releasing all those locks
- Reduced concurrency
- It may take just one slow transaction, or one transaction that accesses a lot of data and acquires many locks, to cause the rest of the system to grind to a halt.
- Deadlocks, abortions and retries

Predicate locks

- Locking a search condition, so that you can have a consistent read

```
SELECT * FROM bookings  
WHERE room_id = 123 AND  
end_time > '2018-01-01 12:00' AND  
start_time < '2018-01-01 13:00';
```

- Transaction A must acquire a shared-mode predicate lock on the conditions of the query.
- If transaction B currently has an exclusive lock matching those conditions, A must wait until B releases its lock.
- Predicate locks apply even to objects that do not yet exist

Index-range locks

- Also called next-key locking
- Simplified approximation of predicate locking
- Lock index entries to the data set you are searching
- Shared locks
- E.g. room_id or start_time / end_time in the room booking example

Serializable Snapshot Isolation (SSI)

- 2PL is a pessimistic concurrency control
- SSI is an optimistic concurrency control
- Detecting stale MVCC reads

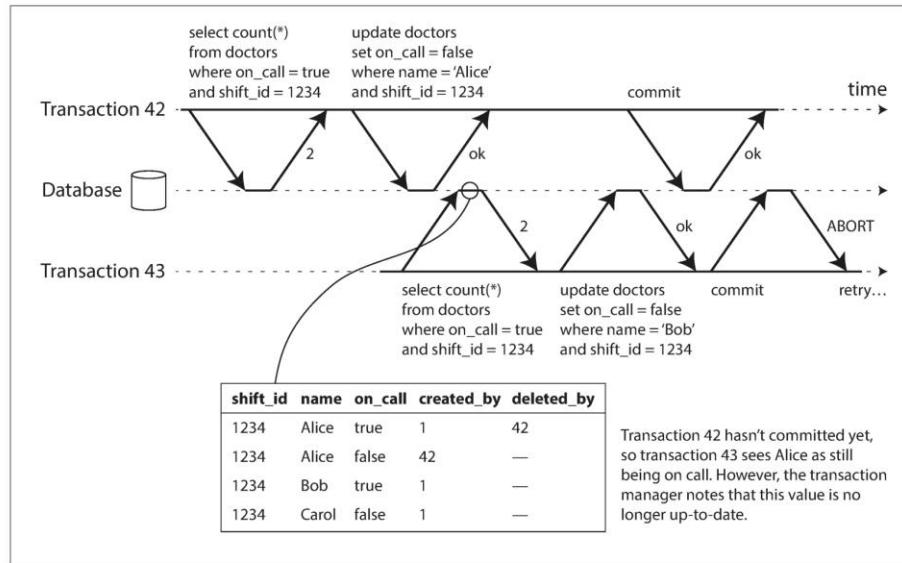


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

Serializable Snapshot Isolation (SSI) (2)

- Detecting writes that affect prior reads

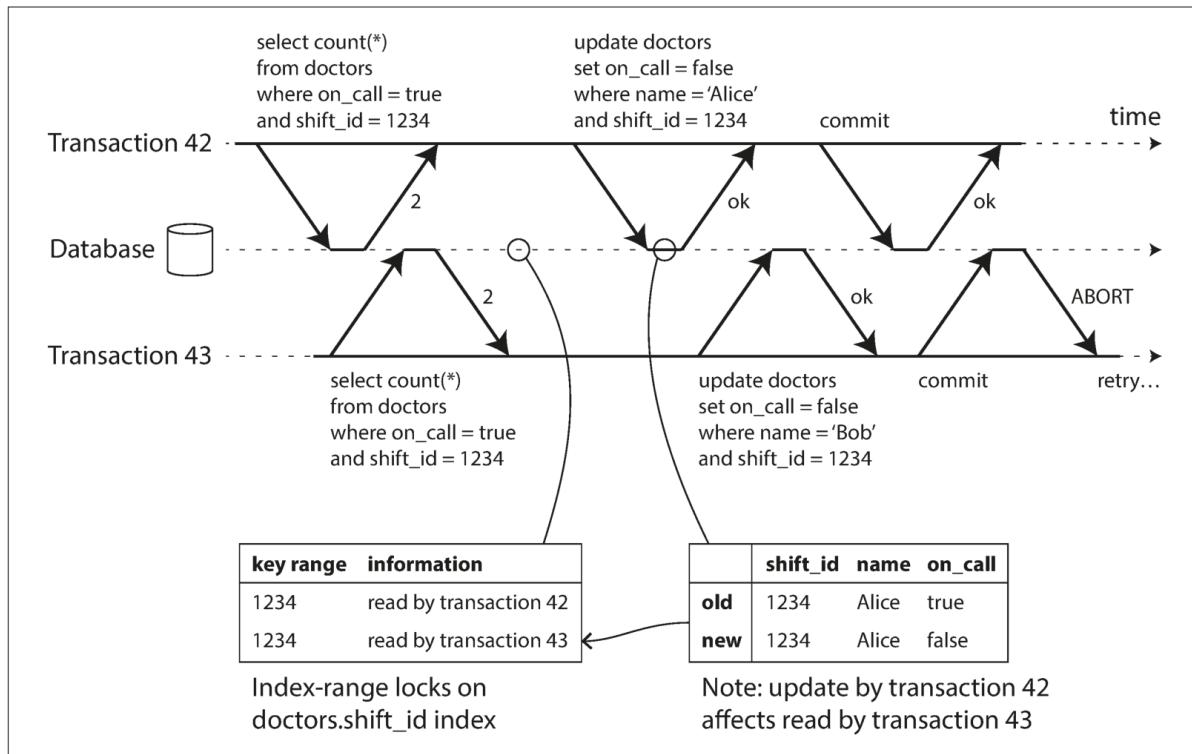


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

SSI – Advantages and performance

- Compared to two-phase locking, the big advantage of serializable snapshot isolation is that one transaction doesn't need to block waiting for locks held by another transaction.
- Compared to serial execution, serializable snapshot isolation is not limited to the throughput of a single CPU core
- FoundationDB distributes the detection of serialization conflicts across multiple machines

Summary

- Transactions are abstractions to simplify error handling
- Isolation levels: read committed, snapshot isolation (repeatable read), and serializable
- Problems
 - Dirty reads
 - Dirty writes
 - Read skew (non-repeatable reads)
 - Lost updates
 - Write skew
 - Phantom reads

Summary (2)

- Solutions to serializable
 - 1. Serial execution
 - 2. Two-Phase locking
 - 3. Serializable snapshot isolation (SSI)

TDT4225

Chapter 8 – The Trouble with Distributed Systems

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Problems in Distributed Systems

- Working with distributed systems is fundamentally different from writing software on a single computer.
- Get a taste of the problems that arise in practice.
- Get an understanding of the things we can and cannot rely on.
- Software running on a single computer has predictable behaviour, unless there are bugs in the software
- Some parts of the system that are broken: partial failure
- This together with non-determinism makes distributed systems hard to work with

Cloud Computing and Supercomputing

- **High-performance computing (HPC).** Supercomputers with thousands of CPUs are typically used for computationally intensive scientific computing tasks
- **Cloud computing:** Multi-tenant datacenters, commodity computers connected with an IP network, elastic/on-demand resource allocation, and metered billing.
- These are different approaches to computing. HPC is much more controlled and is usually batch-oriented

HPC vs Cloud Computing

- CC: online, in the sense that they need to be able to serve users with low latency at any time
- HPC: Specialized hardware, where each node is quite reliable, and nodes communicate through shared memory and remote direct memory access (RDMA)
- CC: networks are often based on IP and Ethernet, arranged in Clos topologies
- CC: it is reasonable to assume that something is always broken. Advantageous if you can run with something broken.
- Network slow on geographically distributed systems
- Building a reliable system from unreliable components

Asynchronous packet networks, no response?

- Your request may have been lost
- Your request may be waiting in a queue and will be delivered later
- The remote node may have failed
- The remote node may have temporarily stopped responding
- The remote node may have processed your request, but the response is lost
- The remote node may have processed your request, but the response has been delayed

No response?

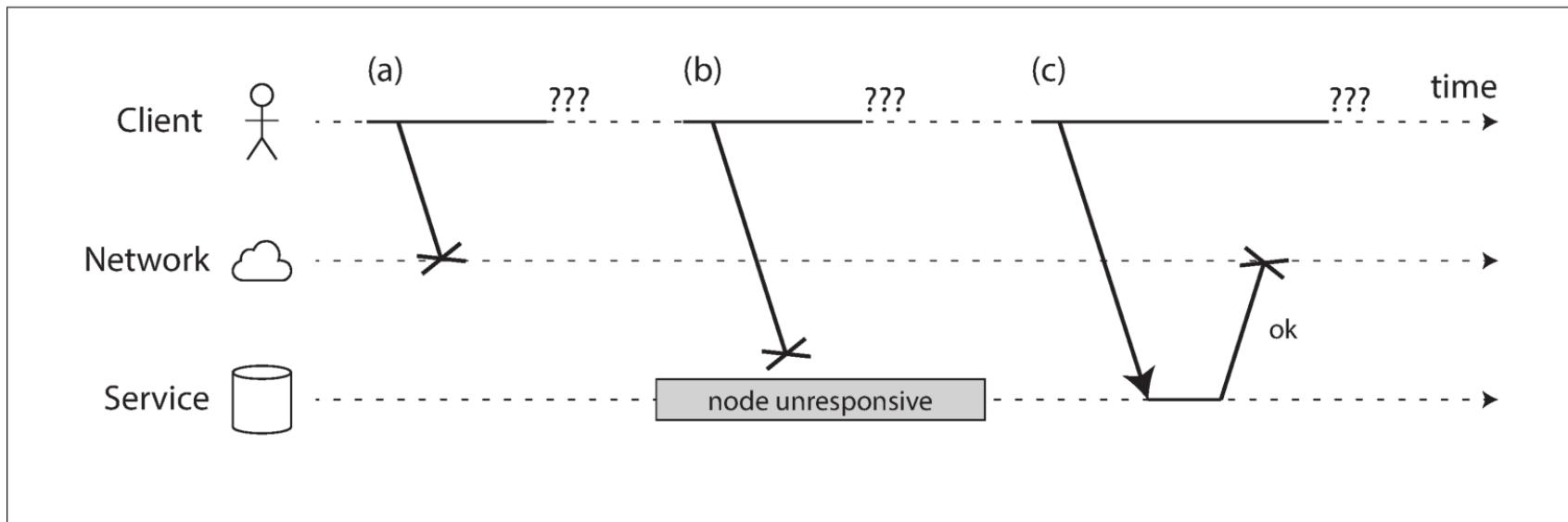


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

Network Faults in Practice

- Networks are unreliable and all sorts of faults may occur
- Human misconfigurations is the biggest cause
- Fault detection may be necessary
 - A load balancer needs to stop sending requests to a node that is dead
 - In a distributed database with single-leader replication, if the leader fails, one of the followers needs to be promoted
- Some feedback is possible
 - On crash of server process: The OS will helpfully close or refuse TCP connections by sending a RST or FIN reply
 - If a node's process crashed but the node's OS is still running, a script can notify other nodes about the crash

Timeouts and Unbounded Delays

- If a timeout is the only sure way of detecting a fault, then how long should the timeout be?
- A long timeout means a long wait until a node is declared dead
- A short timeout detects faults faster, but carries a higher risk of incorrectly declaring a node dead
- When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network.
- Most systems we work with have no guarantees: asynchronous networks have unbounded delays

Network congestion and queueing

- Many computers send packets to the same destination, the network switch must queue them up and feed them into the destination network link one by one

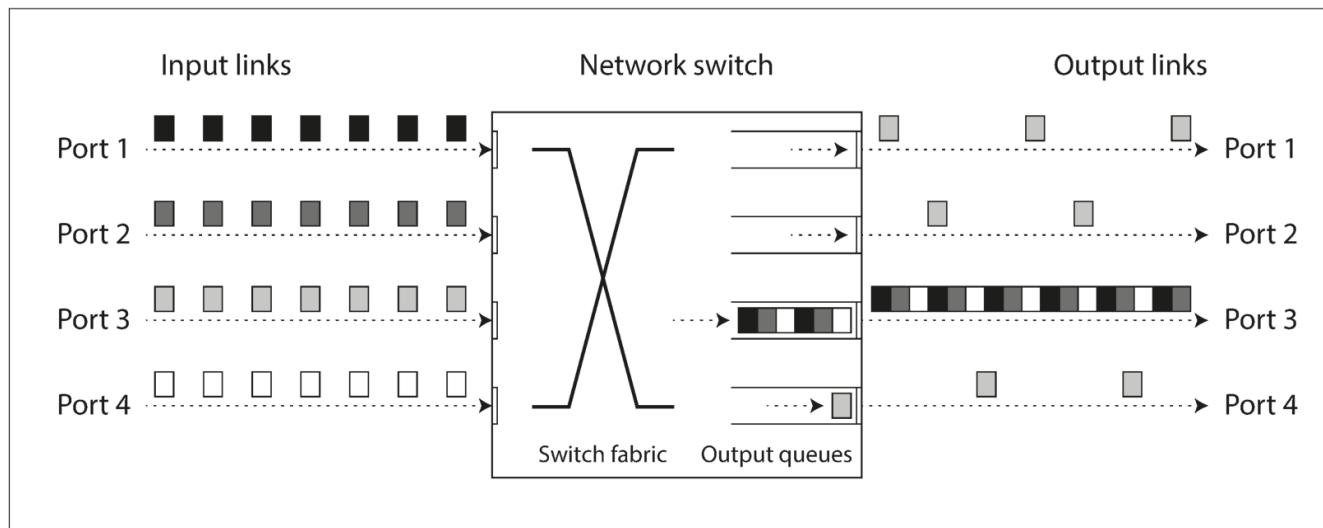


Figure 8-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.

Network congestion and queueing (2)

- If all CPU cores are currently busy, the incoming request from the network is queued by the operating system until the application is ready to handle it.
- In virtualized environments, a running operating system is often paused for tens of milliseconds
- TCP/UDP may add delays at sender node (to wait for more data, or to limit sending data)
- **Timeout:** Systems can continually measure response times, and automatically adjust timeouts according to the observed response time distribution

Predictable networks

- Like the fixed phone lines?
- Packet switching? Optimized for bursty traffic.
- Hard to estimate traffic: If you guess too low (bandwidth), the transfer is unnecessarily slow, leaving network capacity unused. If you guess too high, the circuit cannot be set up.
- Variable delays in networks are not a law of nature, but simply the result of a cost / benefit trade-off.

Unreliable Clocks

- Clocks used for duration
- Clocks used for point in time
- The time when a message is received is always later than the time when it is sent, but due to variable delays in the network, we don't know how much later.
- Makes it difficult to determine the order in which things happened when multiple machines are involved.
- Network Time Protocol (NTP) to synchronize clocks
- Servers may use GPS receivers.

Monotonic vs. Time-of-Day Clocks

- Time-of-day clocks
 - Wall-clock time
 - `System.currentTimeMillis()`
 - `clock_gettime(CLOCK_REALTIME)`
- Monotonic clocks
 - Suitable for measuring a duration
 - `clock_gettime(CLOCK_MONOTONIC)`
 - `System.nanoTime()`
 - Used for measurements
 - Cannot be compared between computers

Clock Synchronization and Accuracy

- The quartz clock in a computer: Drift up to 17 seconds a day
- Local clock with difference from NTP clock: Refuse to synchronize or applications need to accept clock adjustments
- NTP synchronization can only be as good as the network delay
- Some NTP servers are wrong or misconfigured
- Leap seconds
- In virtual machines, the hardware clock is virtualized
- If you run software on devices that you don't fully control, you cannot trust the clock

Relying on Synchronized Clocks

- Clocks work quite well most of the time, but robust software needs to be prepared to deal with incorrectness.
- Incorrect clocks easily go unnoticed.
- If you use software that requires synchronized clocks, it is essential that you carefully monitor the clock offsets.

Timestamps for ordering events

- A dangerous use of time-of-day clocks in a database with multileader replication: last write wins (LWW)

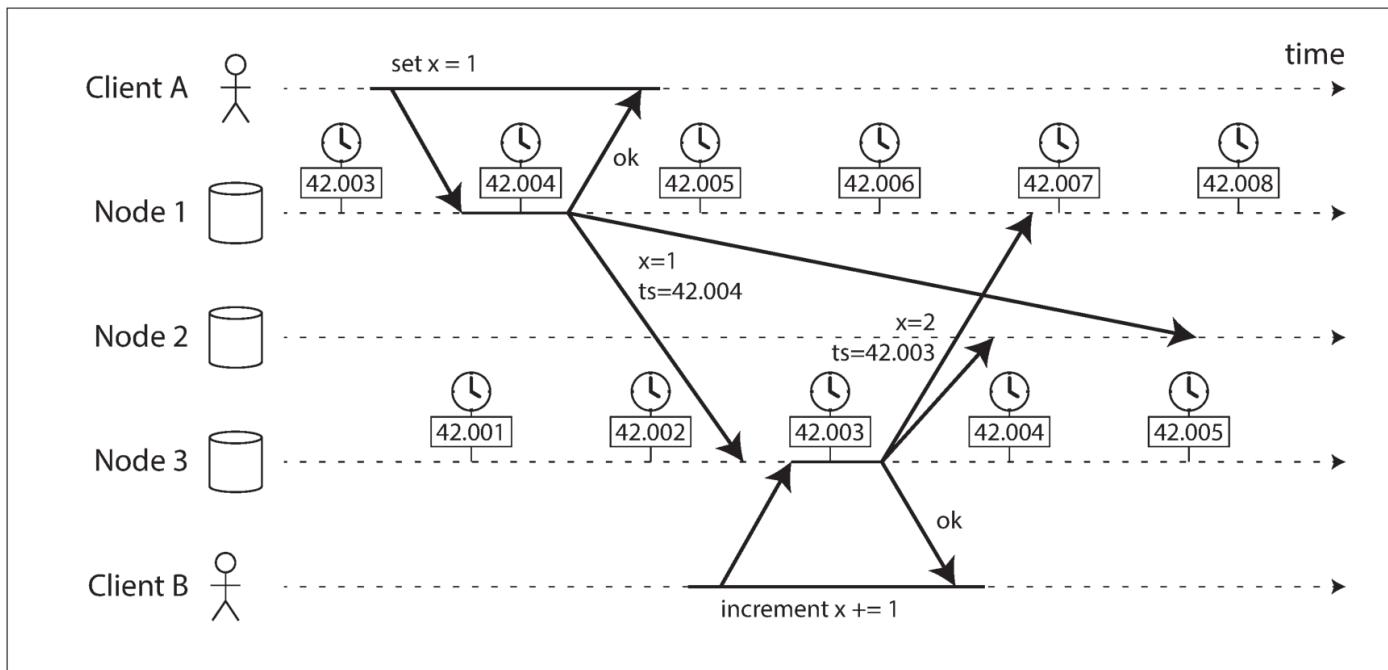


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

Accuracy of last write wins

- Could NTP synchronization be made accurate enough that such incorrect orderings cannot occur?
- Probably not, because NTP's synchronization accuracy is itself limited by the network round-trip time.
- Logical clocks based on incrementing counters are a safer alternative for ordering events.
- Clock readings have a confidence interval
- NTP server gives best possible accuracy is tens of milliseconds, and the error may easily spike to over 100 ms when there is network congestion
- Most systems don't expose clock uncertainty

Synchronized clocks for global snapshots and Google Spanner

- Snapshot isolation: Allows read-only transactions to see the database in a consistent state at a particular point in time
- Distribution: A global, monotonically increasing transaction ID is difficult to generate.
- With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.
- Spanner implements snapshot isolation across datacenters using its TrueTime API.
- Using overlapping intervals as undefined order.

Process Pauses

- Leaders obtain a lease with a timeout from the other nodes

```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

- It's relying on synchronized clocks.
- The code assumes little time passes between the point that it checks the time and the time when the request is processed.

Response time guarantees

- Hard real-time systems: There is a specified deadline
- A real-time operating system (RTOS) that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed.
- C programs
- Developing real-time systems is very expensive, and they are most commonly used in safety-critical embedded devices
- Garbage collection is a challenge in real-time

Knowledge, Truth, and Lies

- Distributed systems: No shared memory, only message passing via an unreliable network with variable delays, and the systems may suffer from partial failures, unreliable clocks, and processing pauses.
- The truth is defined by the majority
- A node cannot necessarily trust its own judgment of a situation
- Quorum: voting among the nodes. Consensus algorithms.

The leader and the lock

- Some situations require only one node to be the leader, hold a lock to a particular resource

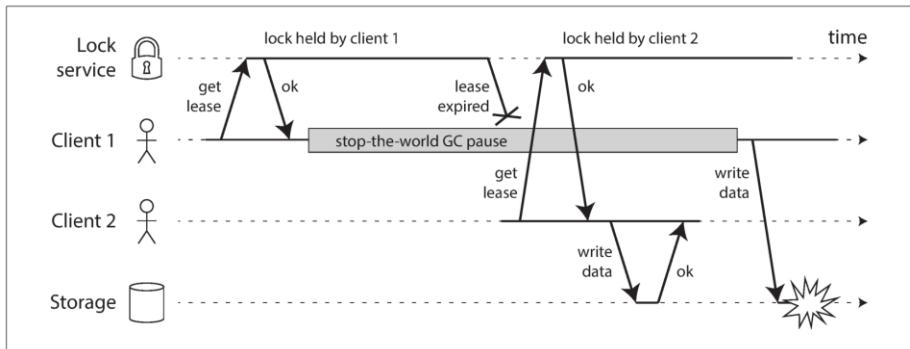


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

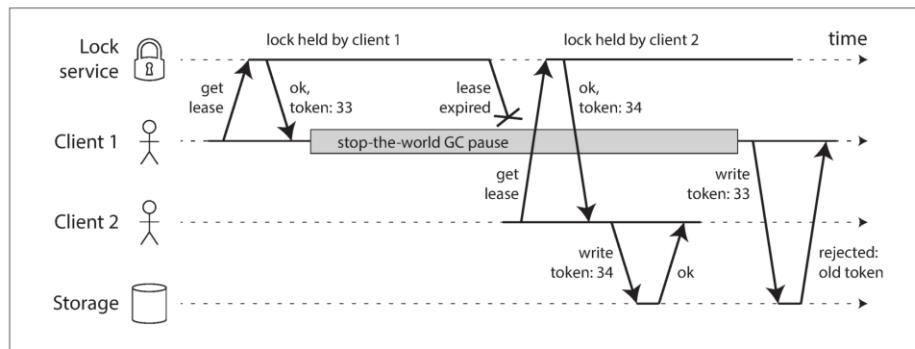


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

- Fencing tokens: An increasing number is get every time a lease is given
- ZooKeeper offers such properties

Byzantine faults

- We assume that nodes are unreliable, but honest: they may be slow or never respond, and their state may be outdated. But they never “lie”.
- If a node tells a “lie”, everything becomes harder: Byzantine faults.
- Aerospace applications need to tolerate Byzantine faults (cosmic radiation)
- Bitcoin / blockchains need to tolerate Byzantine faults (fraud)
- A bug in the software could be regarded as a Byzantine fault, but if you deploy the same software to all nodes, then a Byzantine fault-tolerant algorithm cannot save you.

Handling «weak forms of lying»

- *Errors due to software or hardware bugs:*
- Checksums on messages
- Checking input values
- NTP clients must connect to multiple servers to estimate errors in time

System Model and Reality - timing

- Synchronous model: Bounded network time, bounded clock error, bounded process pauses (not used because assumptions do not hold)
- Partially synchronous model: Usually synchronous, but may experience and tolerate glitches. Mostly used.
- Asynchronous model: No timing assumptions.

System Model and Node failures

- Crash-stop faults: Node fails by crashing
- Crash-recovery faults: Node fails, but recovers from persistent medium
- Byzantine faults: Nodes may do absolutely anything.

Correctness of an algorithm

- **Example:** Lock using fencing token. Properties:
- *Uniqueness:* No two requests for a fencing token return the same value.
- *Monotonic sequence:* If request x returned token tx , and request y returned token ty , and x completed before y began, then $tx < ty$
- *Availability:* A node that requests a fencing token and does not crash, eventually receives a response.

Safety and liveness (math properties)

- **Safety:** Nothing bad happens
- **Liveness:** Something good eventually happens.
- If a safety property is violated, we can point at a particular point in time at which it was broken. After a safety property has been violated, the violation cannot be undone—the damage is already done.
- A liveness property may not hold at some point in time, but may eventually hold
- For distributed algorithms, it is common to require that safety properties always hold
- Liveness: A request needs to receive a response only if a majority of nodes have not crashed.

TDT4225

Chapter 9 – Consistency and Consensus

Svein Erik Bratsberg

Department of Computer Science (IDI), NTNU

Consistency Guarantees

- Replicated systems will have inconsistent values, however,
- Eventual consistency: convergence, as we expect all replicas to eventually converge to the same value.
- The edge cases of eventual consistency only become apparent when there is a fault in the system (e.g., a network interruption) or at high concurrency.
- Systems with stronger guarantees may have worse performance or be less fault-tolerant (available) than systems with weaker guarantees.
- Distributed consistency is mostly about coordinating the state of replicas in the face of delays and faults.

Linearizability

- Aka atomic consistency, strong consistency, immediate consistency, or external consistency.
- The basic idea is to make a system appear as if there were only one copy of the data, and all operations on it are atomic.

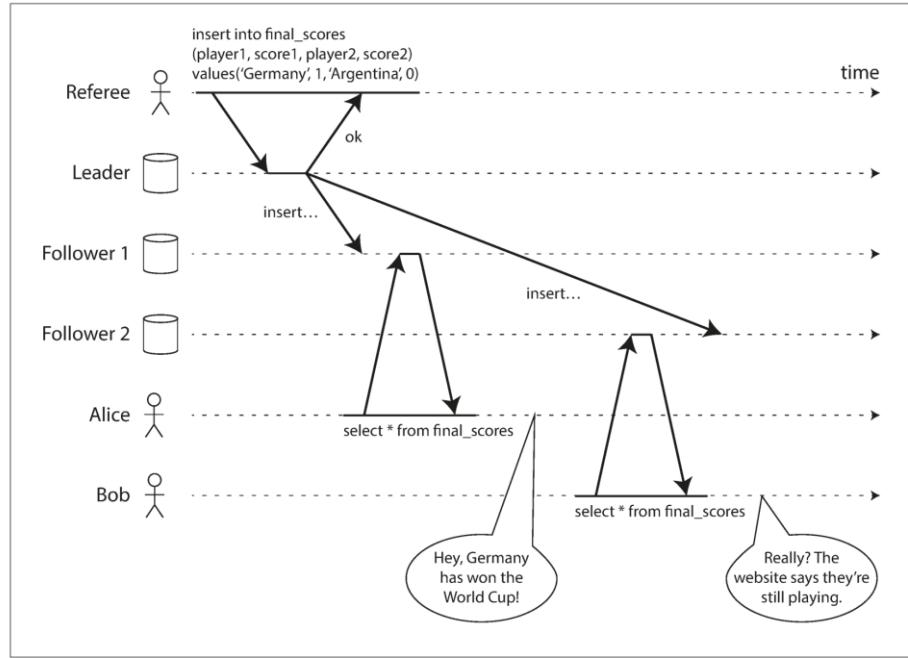


Figure 9-1. This system is not linearizable, causing football fans to be confused.

What Makes a System Linearizable?

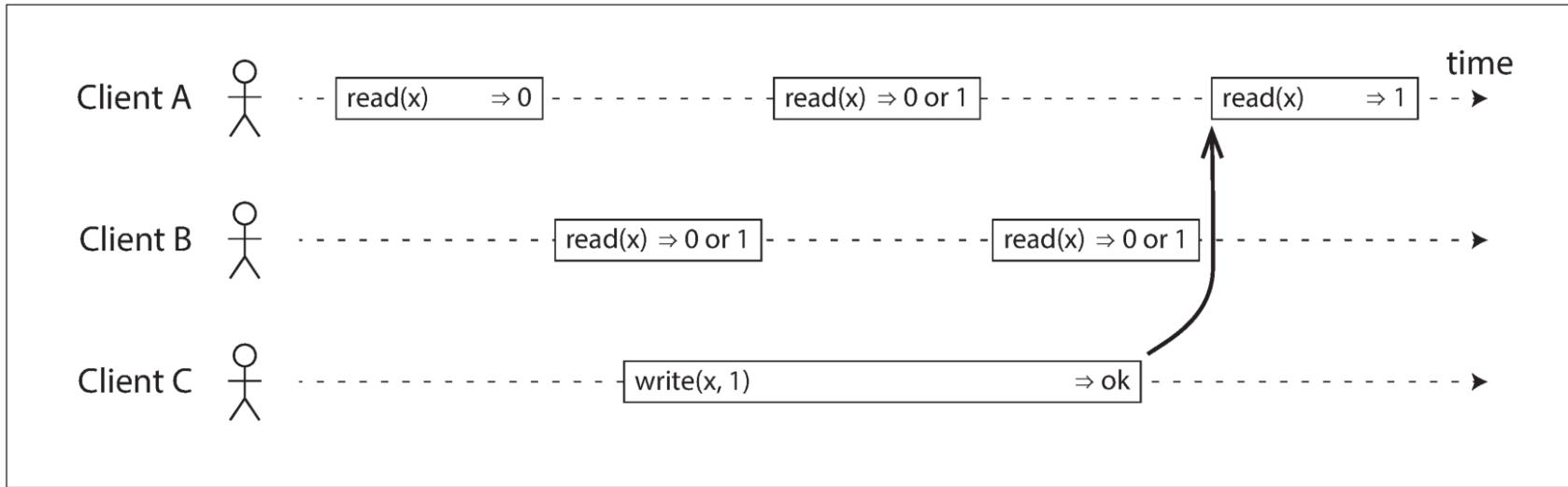


Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

Linearizable, Another Constraint

- An atomic point in time for the flip?

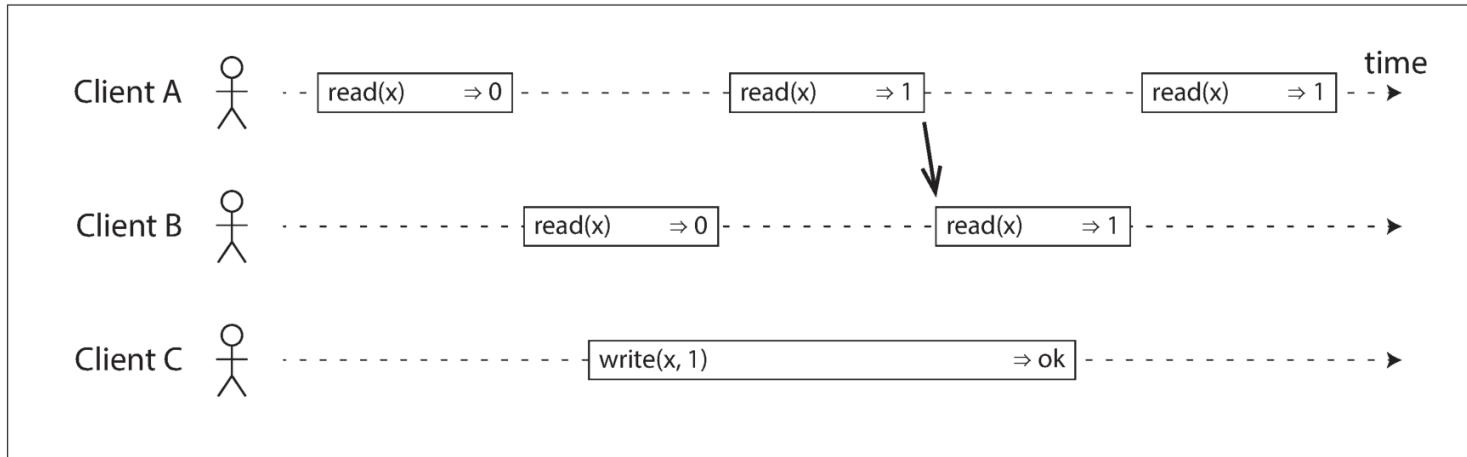


Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

Linearizable, CAS – Compare-and-Set

- $\text{cas}(x, v_{\text{old}}, v_{\text{new}}) \Rightarrow r$ means the client requested an atomic compare-and-set
- An atomic compare-and-set (cas) operation can be used to check the value hasn't been concurrently changed by another client
- The requirement of linearizability is that the lines joining up the operation markers always move forward in time (from left to right), never backward.

Linearizable, CAS – Compare-and-Set (2)

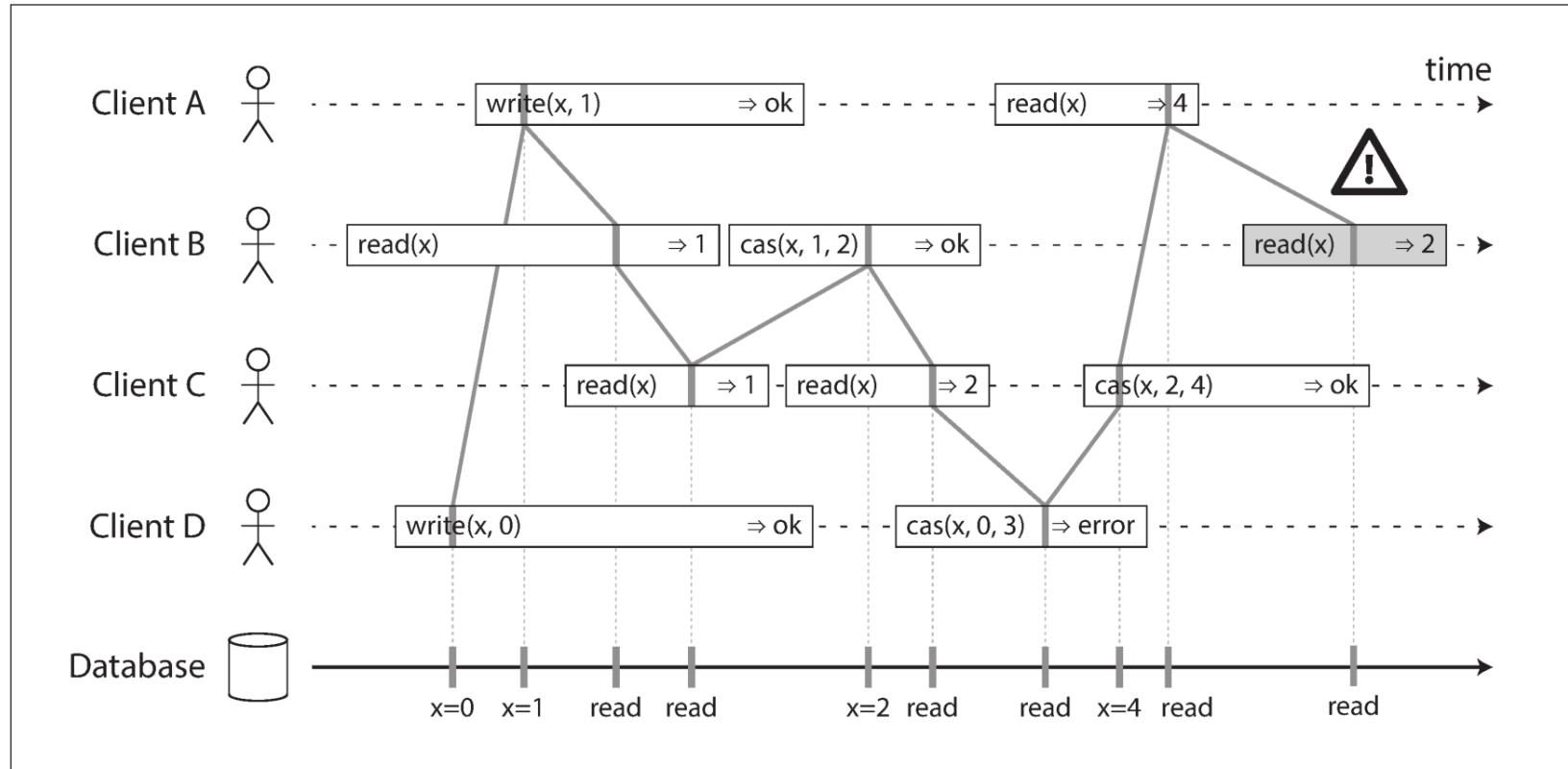


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

Linearizable vs. serializable

- **Serializability:** Guarantees that transactions behave the same as if they had executed in some serial order.
- **Linearizability:** Recency guarantee on reads and writes.
- Implementations of serializability based on two-phase locking or actual serial execution are linearizable.
- Serializable snapshot isolation is by design not linearizable. It hides concurrent writes.

Relying on Linearizability

- Locking and leader election: All nodes must agree which node owns the lock; otherwise it is useless.
- Constraints and uniqueness guarantees: A hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability.
- Cross-channel timing dependencies:

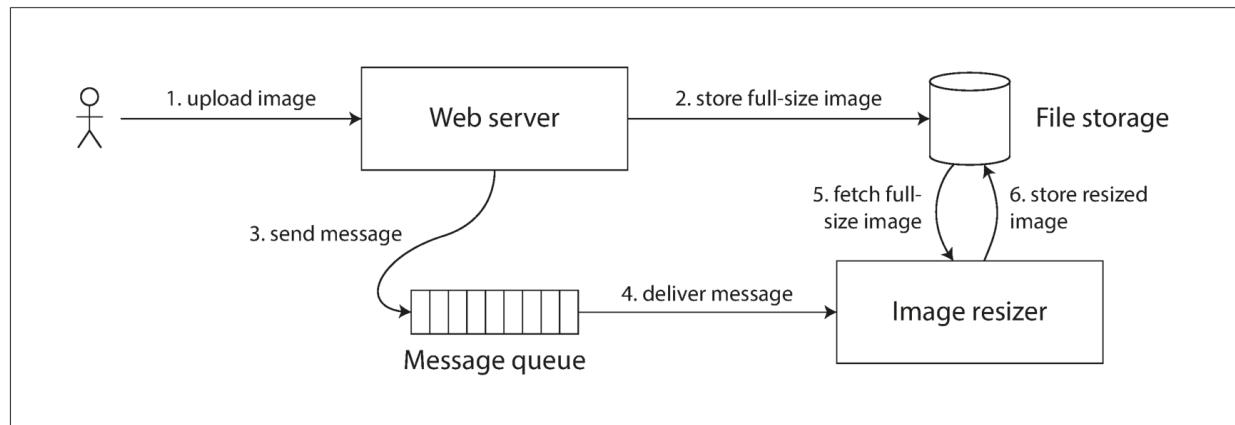


Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

Implementing Linearizable Systems

- Replication methods:
 - Single-leader replication (potentially linearizable)
 - Consensus algorithms (linearizable): ZooKeeper and etcd
 - Multi-leader replication (not linearizable)
 - Leaderless replication (probably not linearizable)

Linearizability and quorums

- It seems as though strict quorum reads and writes should be linearizable, it may not when we have variable network delays:

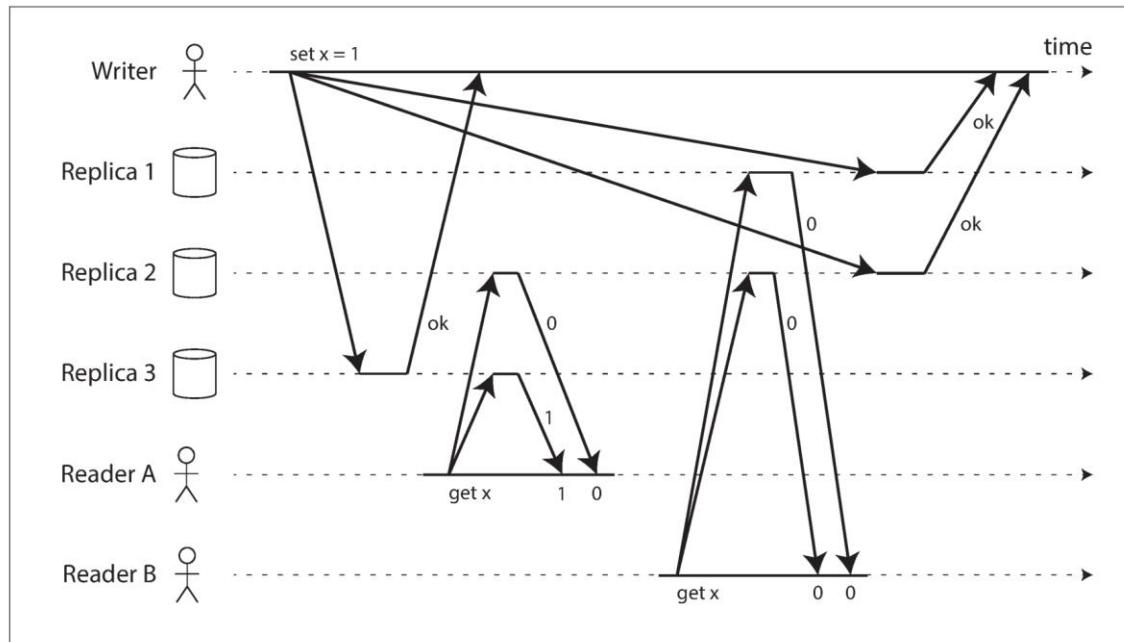


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

The Cost of Linearizability

- If you need linearizability: Must be made unavailable

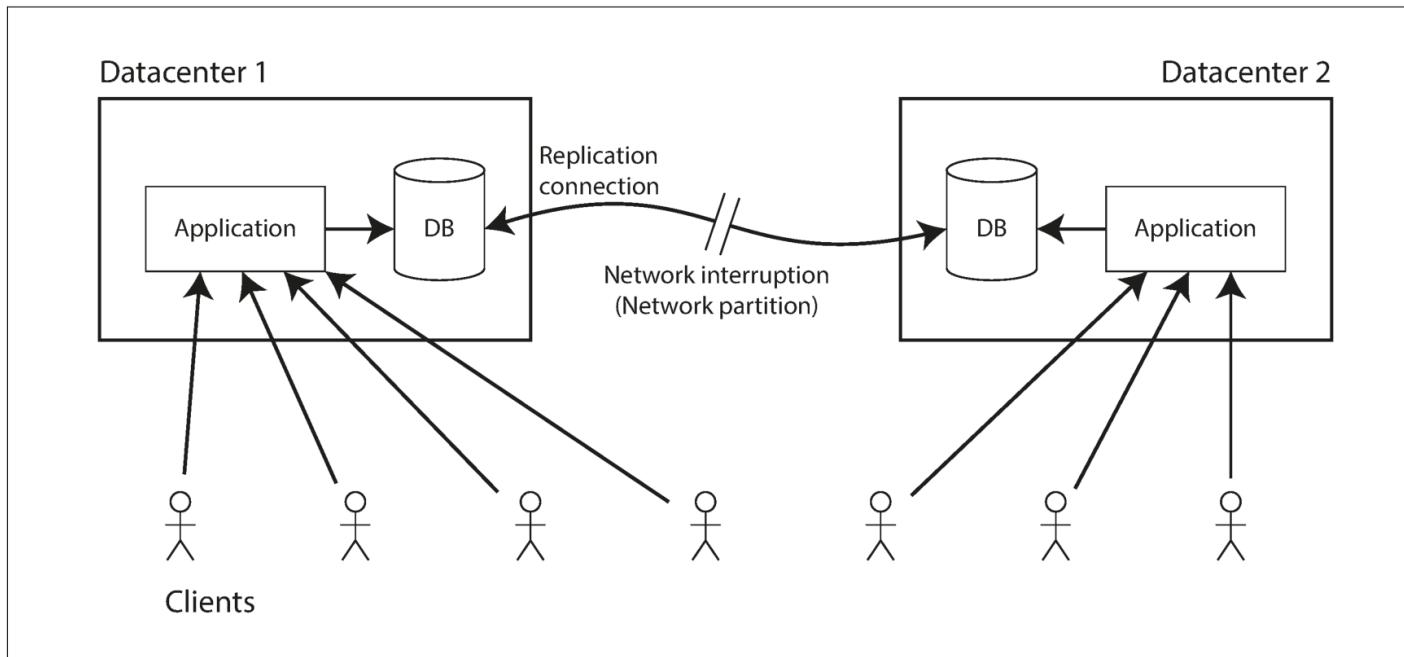


Figure 9-7. A network interruption forcing a choice between linearizability and availability.

The CAP theorem

- Eric Brewer, 2000: Consistency, Availability and Partition Tolerance. Choose any two.
- Applications that don't require linearizability can be more tolerant of network problems.
- Either Consistent or Available when Partitioned.
- Partitioned isn't something you choose
- CAP awaked the interest in other solutions than consistency
- CAP isn't used so much anymore

Linearizability and network delays

- Every CPU core has its own memory cache and store buffer. Memory access first goes to the cache by default, and any changes are asynchronously written out to main memory.
- The reason for dropping linearizability is performance, not fault tolerance.
- The same is true of many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance.
- If you want linearizability, the response time of read and write requests is at least proportional to the uncertainty of delays in the network.

Ordering Guarantees

- There are deep connections between ordering, linearizability, and consensus.
- Ordering helps preserve causality, e.g. causal dependency between the question and the answer.
- E.g. a row must first be created before it can be updated.
- If a system obeys the ordering imposed by causality, we say that it is **causally consistent**.
- Linearizability: total order of operations
- Causality: defines a partial order, some operations are incomparable

Linearizability is stronger than causal consistency

- Linearizability implies causality.
- Performance considerations: Some distributed data systems have abandoned linearizability.
- Causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures.
- We can use sequence numbers or timestamps to order events.
- However, physical clocks cannot be used

Lamport timestamps (1978)

- Every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request: Every causal dependency results in an increased timestamp.

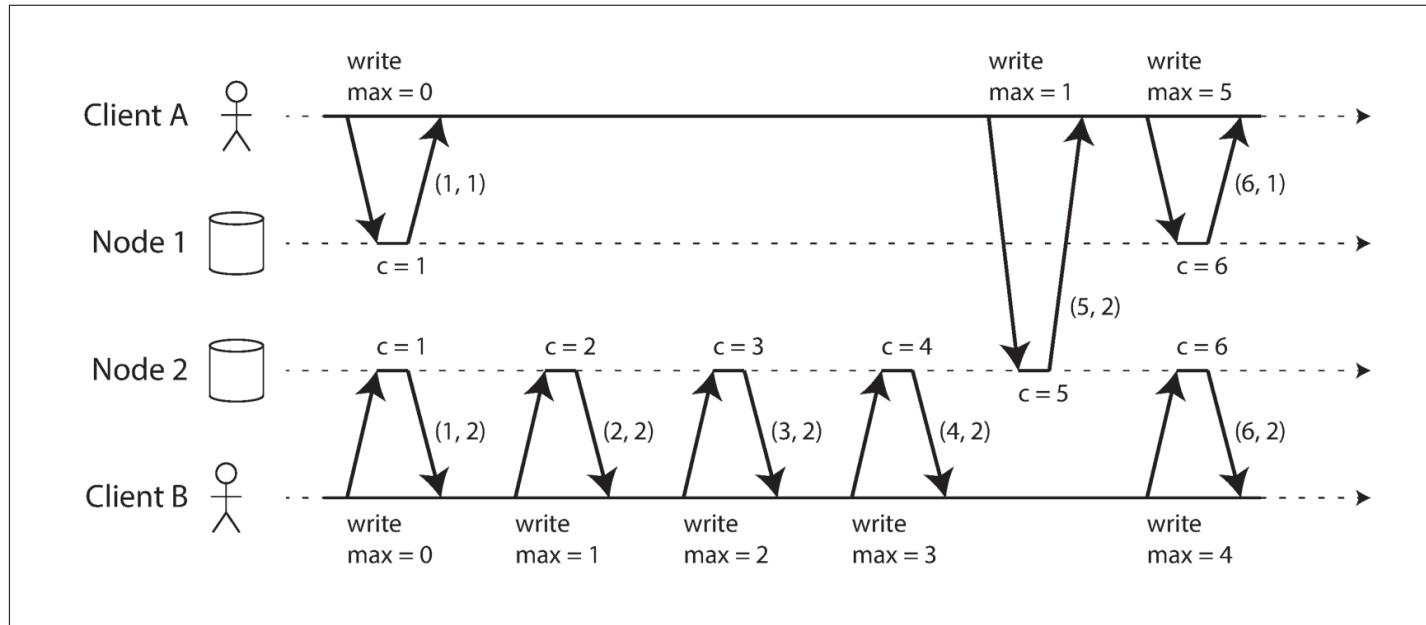


Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

Timestamp ordering is not sufficient

- To implement a uniqueness constraint for user-names, it's not sufficient to have a total ordering of operations – you need to know when that order is finalized among all nodes.
- Partitioned databases with a single leader per partition often maintain ordering only per partition, which means they cannot offer consistency guarantees across partitions
- *Total order broadcast*: a protocol for exchanging messages between nodes.
 - Reliable delivery
 - Totally ordered delivery: Every node receives the messages in the same order
 - May be used for state-machine replication
 - May be seen as a replication log, where all nodes can read the same sequence of messages

Distributed Transactions and Consensus

- Informally, the goal is simply to get several nodes to agree on something.
- Leader election
- Atomic commit
- On a single node, transaction commitment crucially depends on the order in which the log is durably written to disk: operations + commit log record
- A transaction commit must be irrevocable—you are not allowed to change your mind and retroactively abort a transaction after it has been committed.

Two-phase commit (2PC)

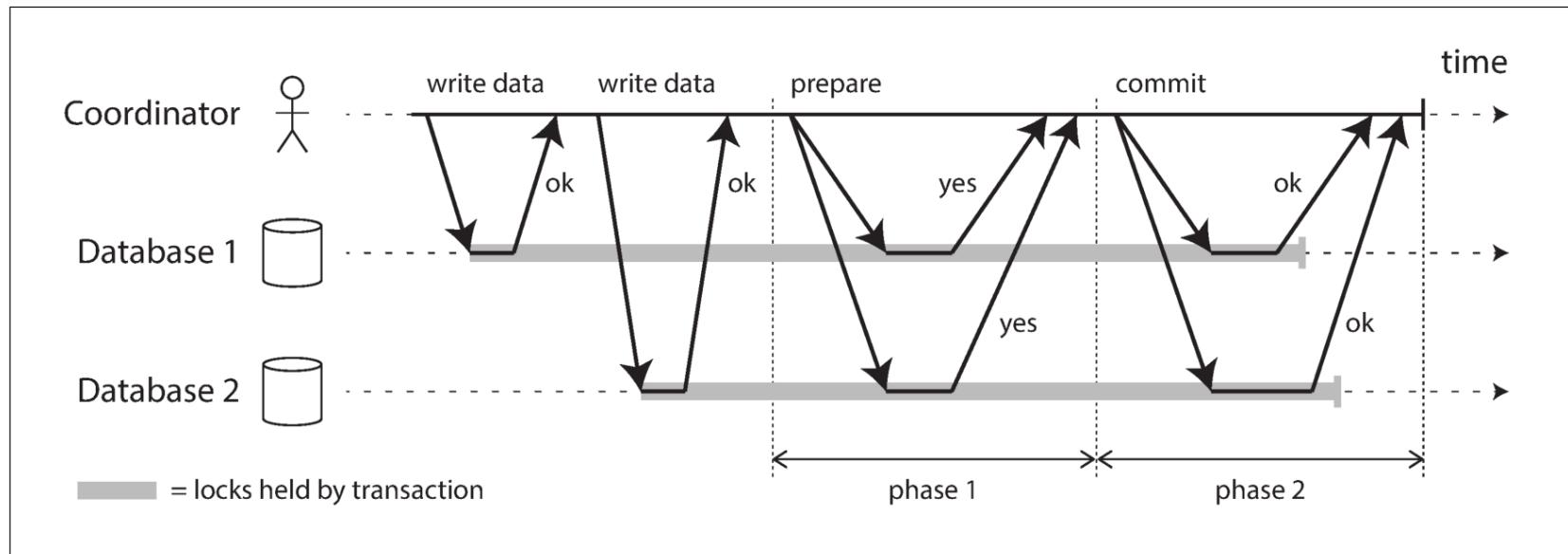


Figure 9-9. A successful execution of two-phase commit (2PC).

2PC (2)

- Coordinator
- Participants
- If all participants reply “yes,” indicating they are ready to commit, then the coordinator sends out a commit request in phase 2.
- If any of the participants replies “no,” the coordinator sends an abort request to all nodes in phase 2.
- Commitment:
 - It’s like bacon and eggs
 - The chicken participates
 - The pig is committed

2PC (3)

- Request a transaction ID from the coordinator
- Start execution on all nodes. Any node may decide to abort.
- When ready-to-commit, the coordinator sends a prepare-to-commit to all participants
- Participants receive PTC, and replies yes or no.
- Coordinator receives all, and decides. Logs the decision.
- The commit or abort message is sent to all participants
- Done is returned from all participants

Coordinator failure

- If any of the prepare requests fail or time out, the coordinator aborts the transaction.
- If any of the commit or abort requests fail, the coordinator retries them indefinitely.
- **Uncertain / in doubt:** If the coordinator crashes, the participant can do nothing but wait until the coordinator recovers.

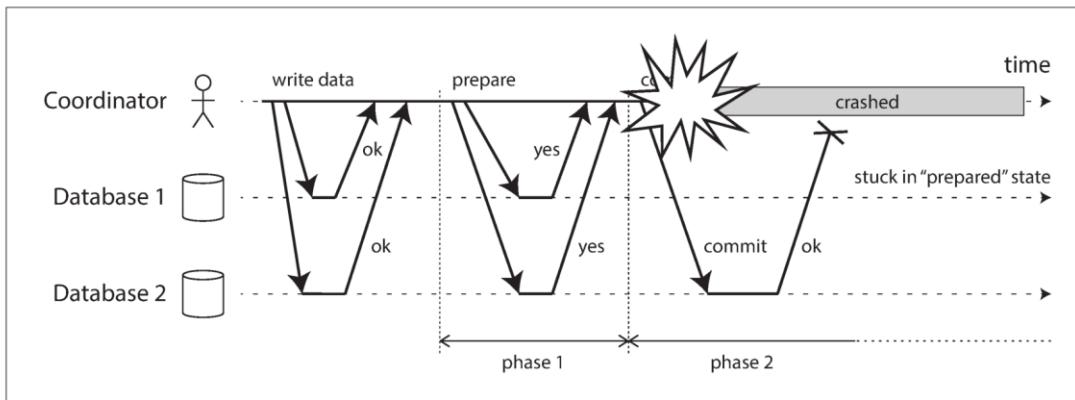


Figure 9-10. The coordinator crashes after participants vote “yes.” Database 1 does not know whether to commit or abort.

Performance of 2PC

- Too costly?: Due to the additional disk forcing (`fsync`) that is required for crash recovery, and the additional network round-trips.
- Two types of distributed transactions
 - Database-internal distributed transactions
 - Heterogeneous distributed transactions (TP monitors)
- XA Protocol: a C API for interfacing with a transaction coordinator. Standard 2PC support.
- The coordinator is usually a library that is loaded into the same process as the application issuing the transaction.

Holding locks while in doubt

- Problem when prepared-to-commit, but coordinator dies

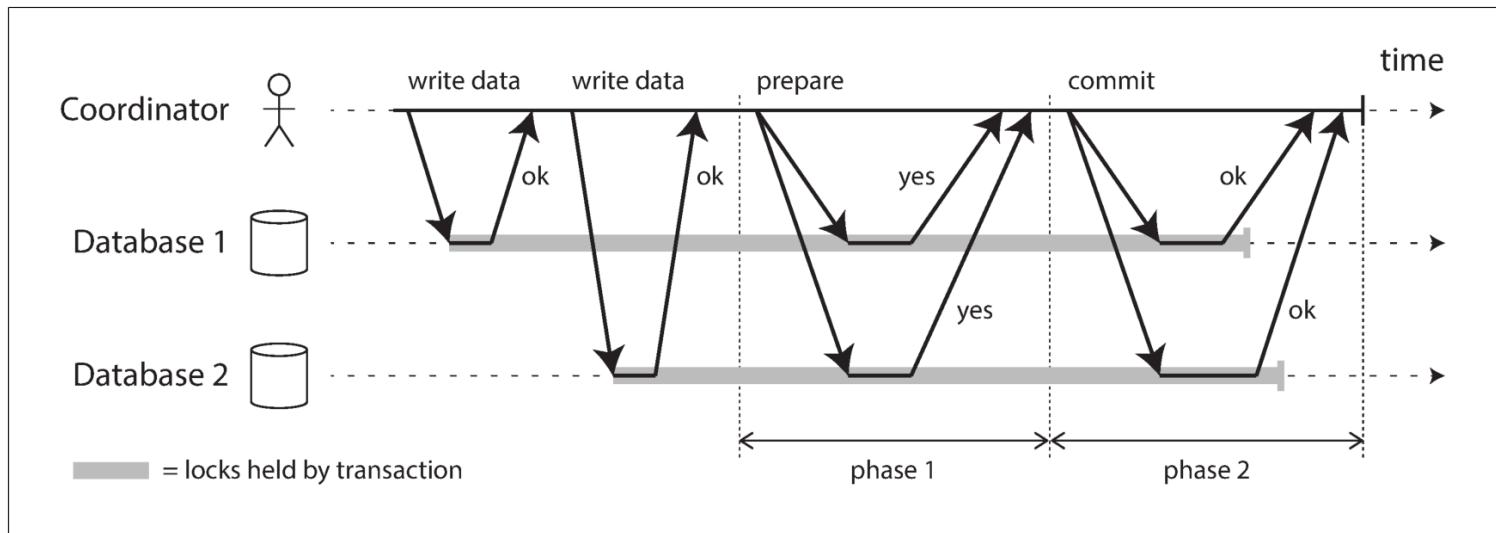


Figure 9-9. A successful execution of two-phase commit (2PC).

- Orphaned transactions must often be manually GCed (removing problematic locks after a long period)

Limitations of distributed transactions (XA)

- The coordinator must also be replicated (hot standby controller)
- XA cannot detect deadlocks in-between heterogenous participants.
- Problem when applications become parts of the transaction execution.
- If any part of the system is broken, the transaction also fails. Distributed transactions thus have a tendency of amplifying failures.

Fault-Tolerant Consensus

- One or more nodes may propose values, and the consensus algorithm decides on one of those values.
- Everyone decides on the same outcome, and once you have decided, you cannot change your mind.
- *Uniform agreement*: No two nodes decide differently.
- *Integrity*: No node decides twice.
- *Validity*: If a node decides value v , then v was proposed by some node.
- *Termination*: Every node that does not crash eventually decides some value.

Consensus algorithms and total order broadcast

- Viewstamped Replication (VSR)
- Paxos (and multi-Paxos)
- RAFT
- Zab
- They decide on a sequence of values: Total order broadcast algorithms.
- Messages to be delivered exactly once, in the same order, to all nodes.

Epoch numbering and quorums

- Epoch number: The algorithms guarantee that within each epoch, the leader is unique.
- Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader.
- The new leader must collect votes from a quorum of nodes to see that it has the highest epoch number.
- If a vote on a proposal succeeds, at least one of the nodes that voted for it must have also participated in the most recent leader election.

Limitations of consensus

- Consensus algorithms were a breakthrough in distributed systems, however they are costly.
- Asynchronous replication: Some operations may be lost
- Consensus systems always require a strict majority to operate. Network failure may be a problem.
- Consensus systems generally rely on timeouts to detect failed nodes. Variable network time may be a problem.
- RAFT: If one network link is problematic, the algorithm may be fluctuating between multiple new leaders.
(sensitive to network problems)

Membership and Coordination Services

- ZooKeeper and etcd are designed to hold small amounts of data that can fit entirely in memory
- ZooKeeper is modeled after Google's Chubby lock service
 - Linearizable atomic operations
 - Total ordering of operations
 - Failure detection
 - Change notification
- ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients.
- Service discovery does not require consensus, leader election does.
- A membership service determines which nodes are currently active and live members of a cluster.

Summary

- Consistency and consensus.
- Similar solutions / problems
 - Linearizable compare-and-set registers
 - Atomic transaction commit
 - Total order broadcast
 - Locks and leases
 - Membership / coordination service
 - Uniqueness constraint



NTNU

Det skapende universitet



TDT 4225 Very Large, Distributed Data Volumes

Chapter 14
Time and global states

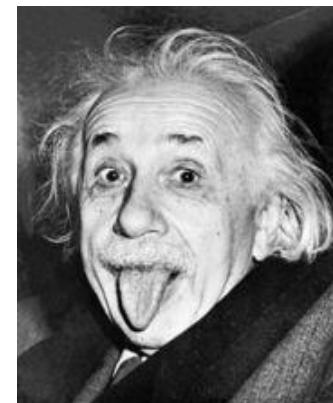
Jon Olav Hauglid, Svein Erik Bratsberg

Part 1: Time

- Why is time important?
- Physical time
 - Skew, drift, UTC
 - External and internal synchronization
 - NTP: Network Time Protocol
- Logical time
 - Definitions
 - Logical clocks
 - Vector clocks

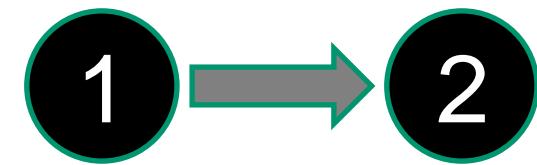
Why is time important?

- Time is used «everywhere»
 - When did something happen?
 - What happened first and what happened later?
 - Bank transactions, e-mail, ...
- Trivial for a single computer
- More difficult in a distributed system
 - Communication takes time
 - What is the time?
 - No universally correct time



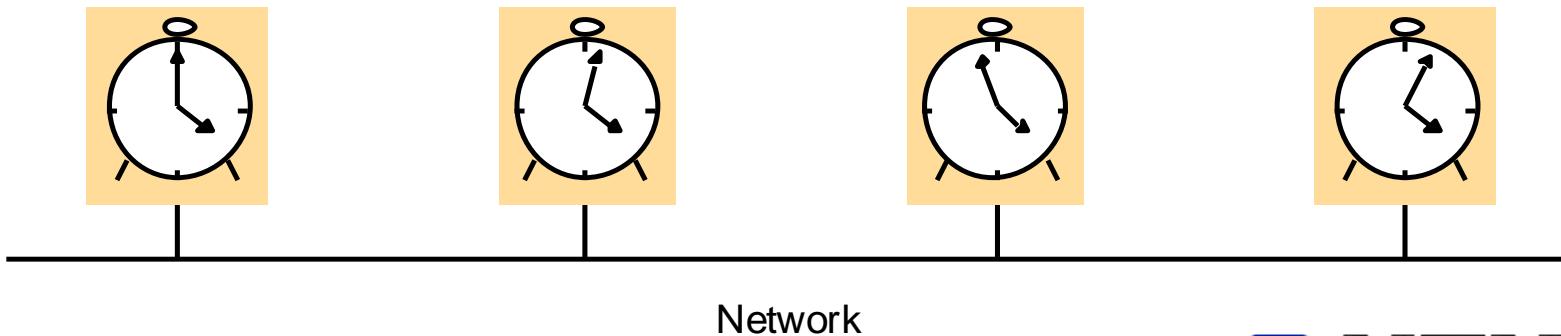
Physical and logical time

- Physical time
 - Timestamp of event
 - Can derive order of events
 - Requires synchronized clocks
- Logical time
 - Order of events
 - Focused on cause and effect
 - Typically a counter incremented for each event



Physical time

- Demand for physical time
- Ordering of events based on timestamp
- Can we trust clocks?
 - «Skew» - difference between clocks at a point in time
 - «Drift» – skew changes over time



Network

UTC – Coordinated Universal Time

- Highly accurate international time standard
- Based on atomic clocks
- An extra second is sometimes inserted due to Earth's rotation slowing down
- Time zones are relative to UTC
 - We are at UTC+1 (UTC+2 in the summer)
- Transmitted using
 - Ground based stations (~1 ms accuracy)
 - Satellites GPS (~1 μ s accuracy)

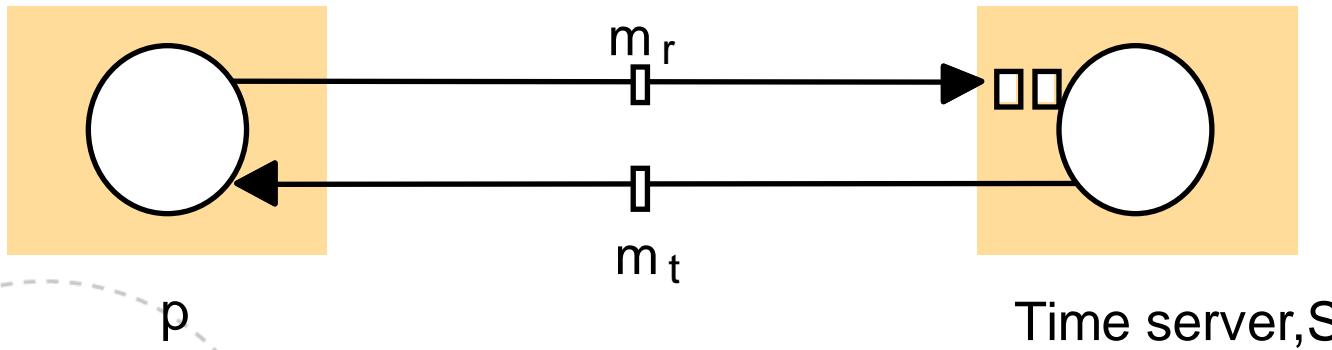
Physical clock synchronization

- External synchronization
 - Clocks synchronized against an external time source
 - Christian's algorithm
- Internal synchronization
 - Synchronization of clocks internally in a distributed system
 - Not necessarily the «correct» time
 - The Berkeley algorithm
- Basic problem: Communication takes time

Christian's algorithm

External time server, S (UTC)

1. p sends message m_r to S
2. S replies with its time t in message m_t
3. When p receives m_t , it sets its clock to $t + \text{half of the time passed since } m_r \text{ was sent}$



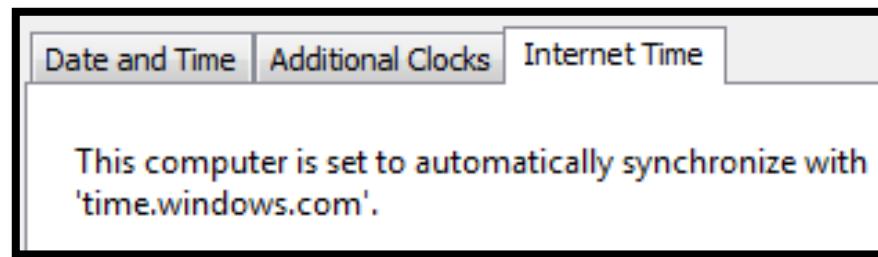
The Berkeley algorithm

One node selected as *master*

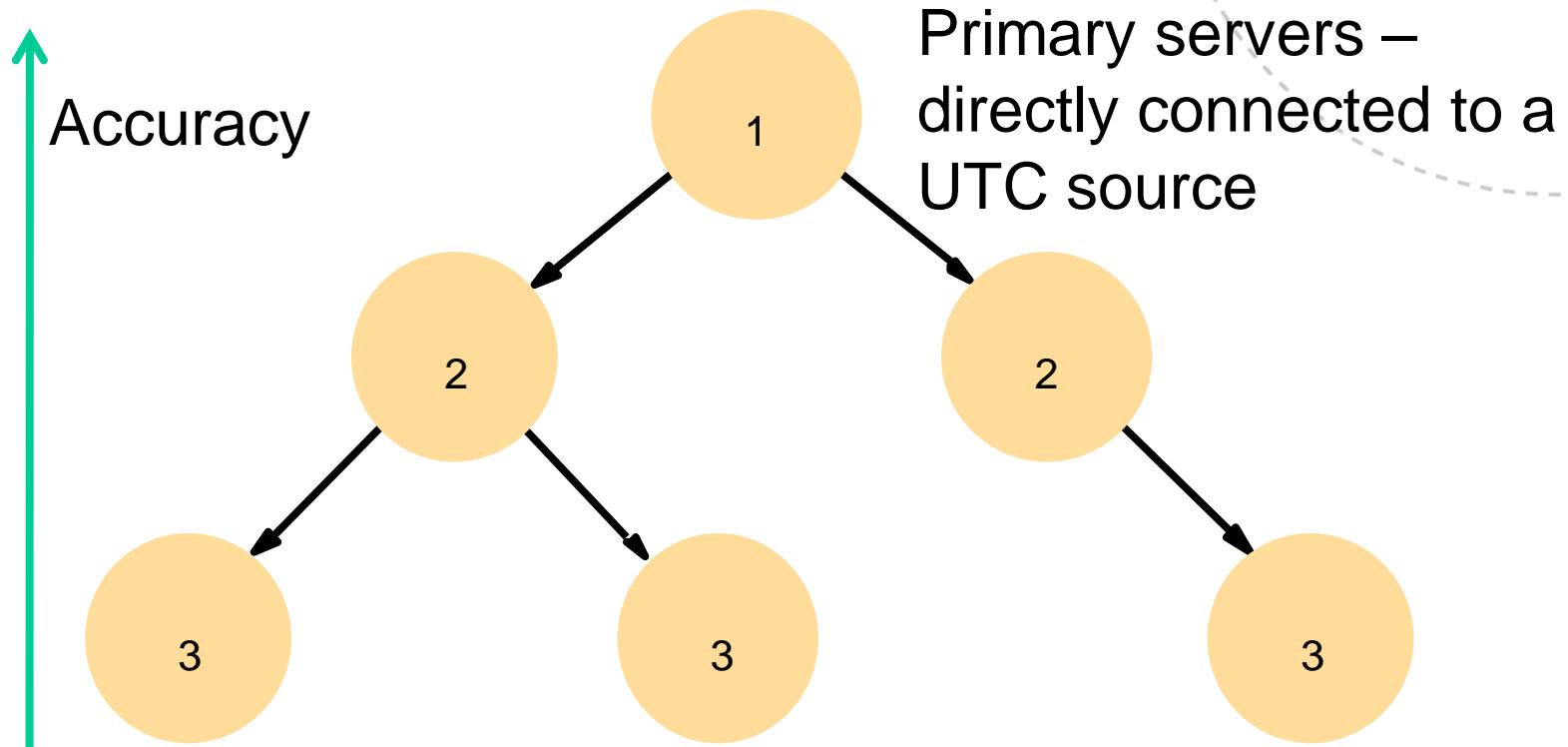
1. The master polls all other nodes (*slaves*)
2. Slaves reply with their local time
3. Master calculates average time
 - Message latency considered
 - Ignores outliers
4. Master sends individual differences to each slave (why differences?)

NTP: Network Time Protocol

- Protocol for synchronizing clocks connected to the Internet
- Uses UTC
- Focus on:
 - Scalability – hierarch of servers
 - Correctness – handle clock drift
 - Reliability – dynamic reconfiguration
 - Security – authentication etc.



NTP: Server hierarchy (logical)



NTP: Synchronization

Three synchronization modes:

1. Multicast (LAN)

- Assumes fixed message latency
- Periodic multicast (not on demand)

2. Procedure-call (e.g. Christian's algorithm)

3. Symmetric mode (high accuracy)

- Servers communicate in pairs
- One server can be in multiple pairs
- Estimates *offset* and *delay*

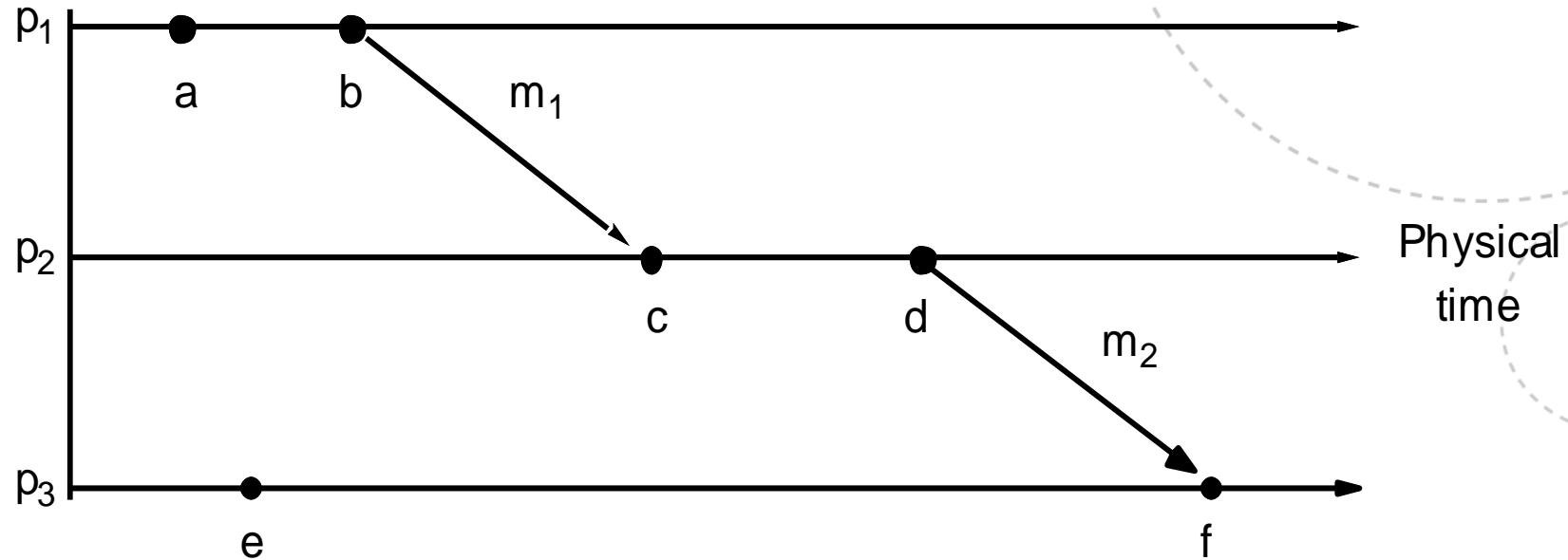
Logical time

- If we only need to order events, physical time is overkill
- Also, perfect synchronization of physical clocks is impossible!
- Local time – focus on event order
 - Two local events happened in the order observed by the process executing them
 - A message must be sent before it can be received
 - I.e. cause → effect

Definitions

- Collection of processes p_i , $i = 1, 2, \dots N$
- Every process p_i has a state s_i
 - E.g. the values of all variables
- Processes communicate using messages
- Events
 - State change
 - Sending or receiving a message

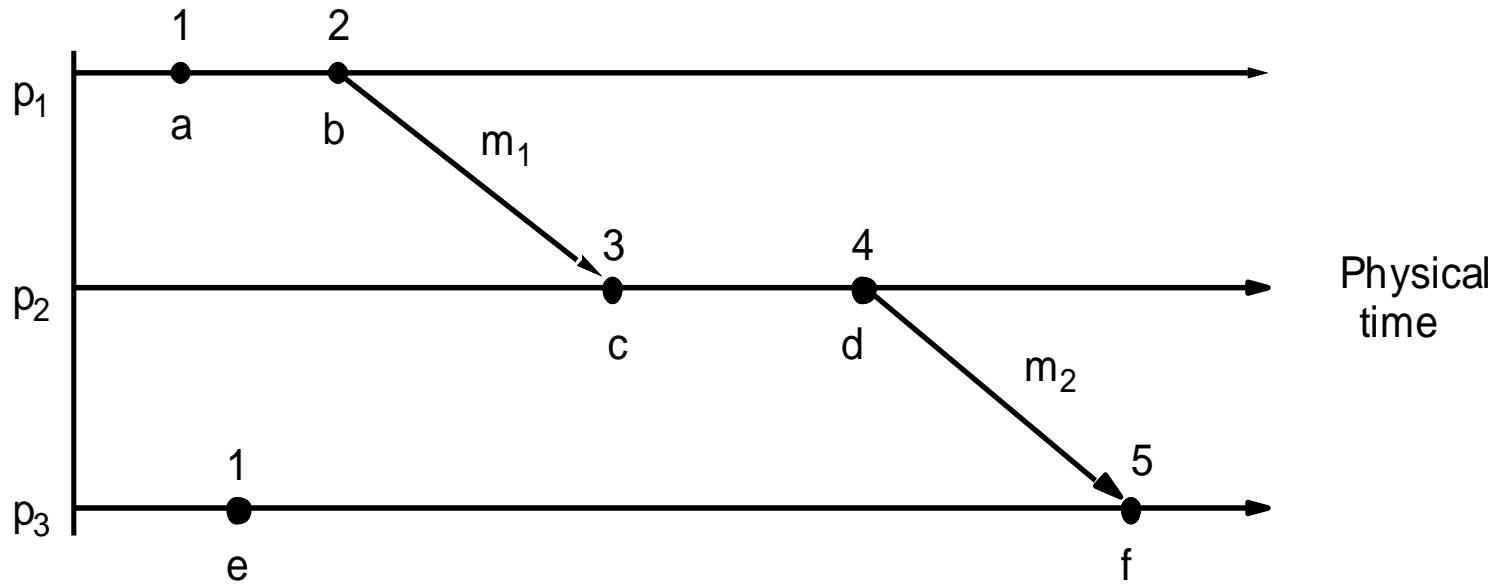
Happened-before (\rightarrow)



- Local events: $a \rightarrow b$; $c \rightarrow d$; $e \rightarrow f$
- Messages: $b \rightarrow c$; $d \rightarrow f$
- Derived: $a \rightarrow c$; $a \rightarrow f$; $b \rightarrow d$
- Concurrent: $a \parallel e$; $b \parallel e$; $c \parallel e$; $d \parallel e$

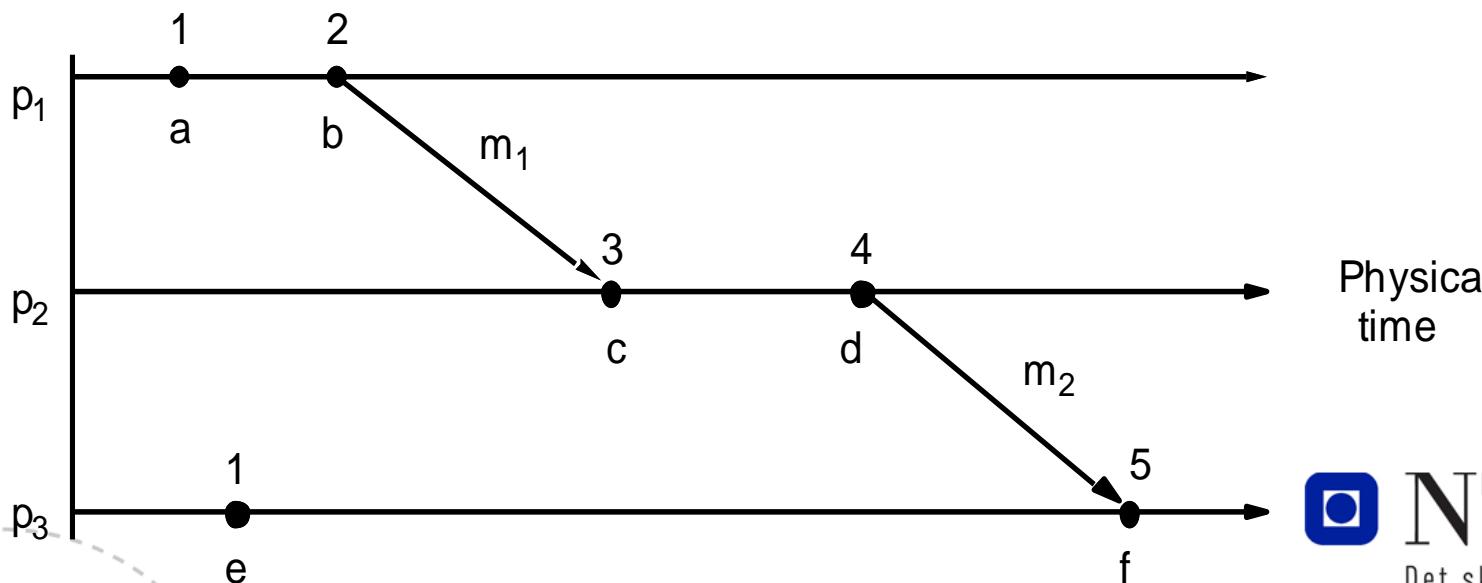
Logical clocks (Lamport) (1/2)

- Every process has a logical clock (counter) L_i ,
- Before every event: $L_i = L_i + 1$
- Attach clock value to every message, $t = L_i$
- When receiving, $L_j = \max(L_j, t) + 1$

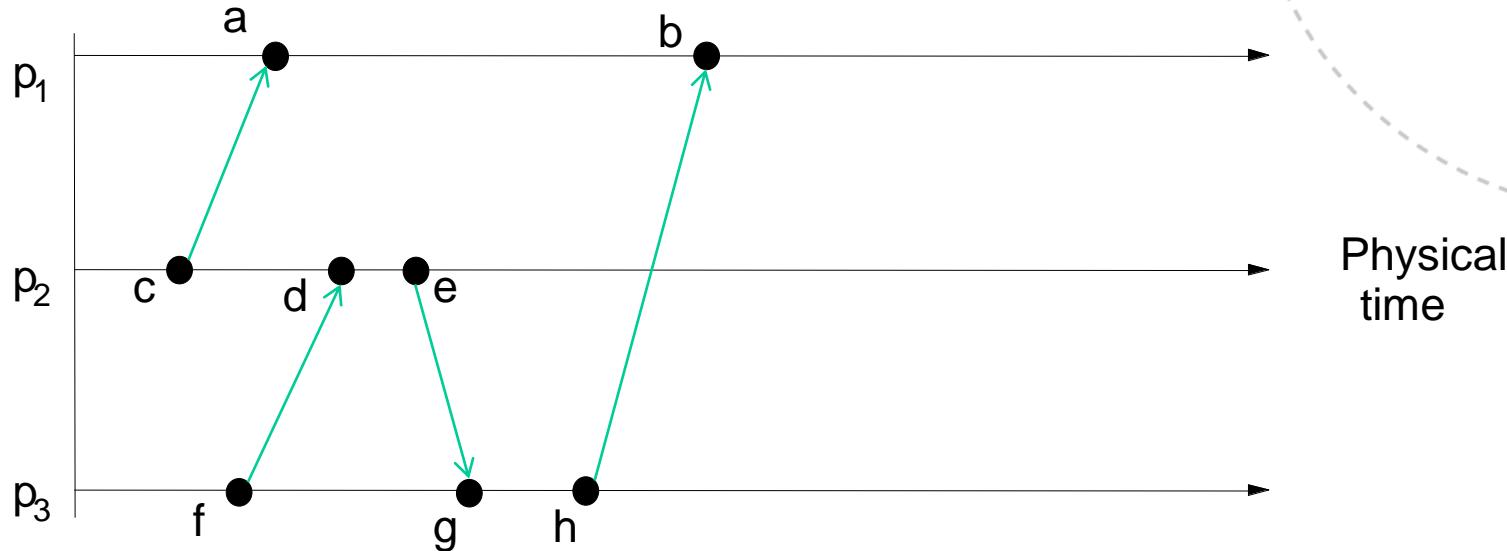


Logical clocks(Lamport) (2/2)

- If $e \rightarrow e'$ then $L(e) < L(e')$
- But the reverse is not true!
 - $L(e) < L(b)$ without $e \rightarrow b$
 - But $L(e') > L(e) \rightarrow (\text{not } e' \rightarrow e)$



Try yourselves!



- What are the logical clock values?
- Is $d \rightarrow g$?
- Is $a \rightarrow g$?

Vector clocks (1/3)

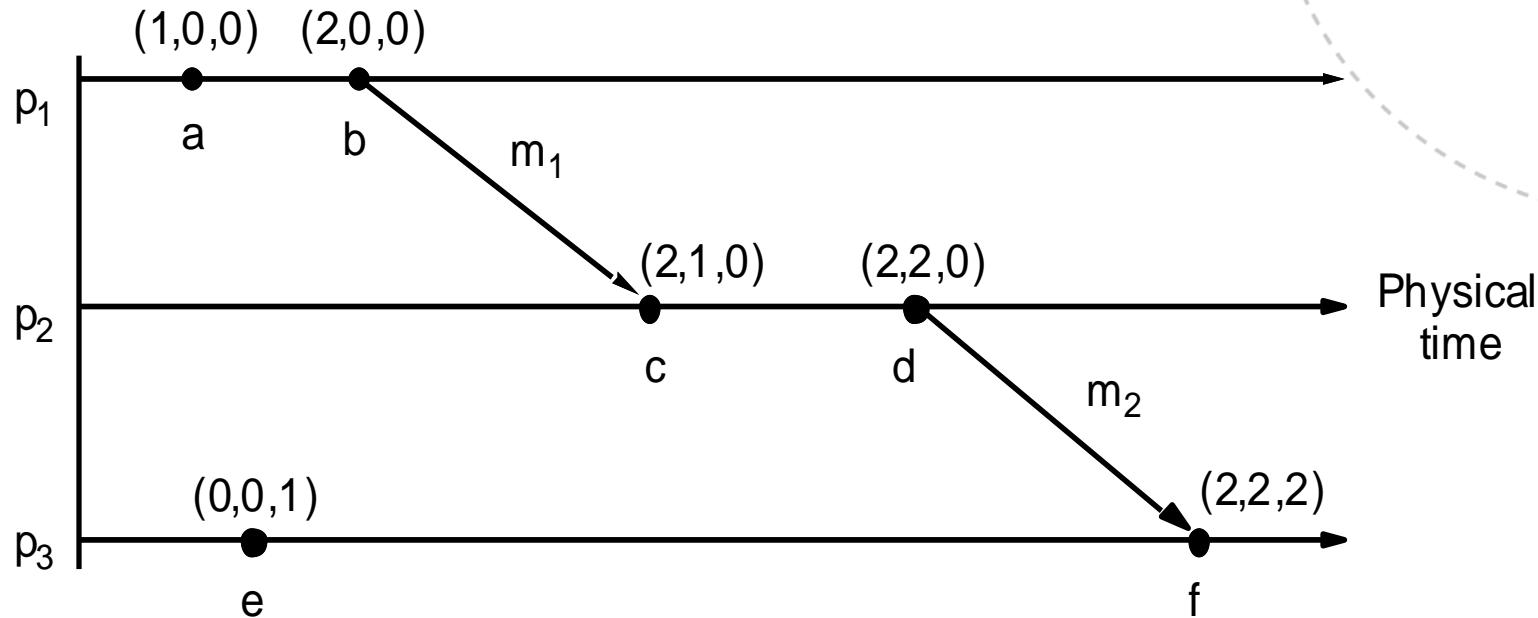
- Logical clocks only get you so far
- To figure out more about event order, we need to store/transfer more info
- Vector clock, given N processes
 - Every process has a vector of N elements
 - Contains the number of events from each process that the given process can have been affected by

Vector clocks (2/3)

Definitions:

- V_i – vector at process i
- Initially all vector elements = 0
- Before each event at p_i :
 $V_i[i] = V_i[i] + 1$
- p_i attaches $t = V_i$ to all messages
- When p_i receives a message,
 $V_i[j] = \max(V_i[j], t[j])$, for $j = 1, 2\dots N$
(Also $V_i[i] = V_i[i] + 1$)

Vector clocks (3/3)



- If $e \rightarrow e'$ then $V(e) < V(e')$
- If $V(e) < V(e')$ then $e \rightarrow e'$
 - $V < V'$ iff $V \leq V'$ and $V \neq V'$
 - \leq and $=$ must hold for all pairs of vector elements

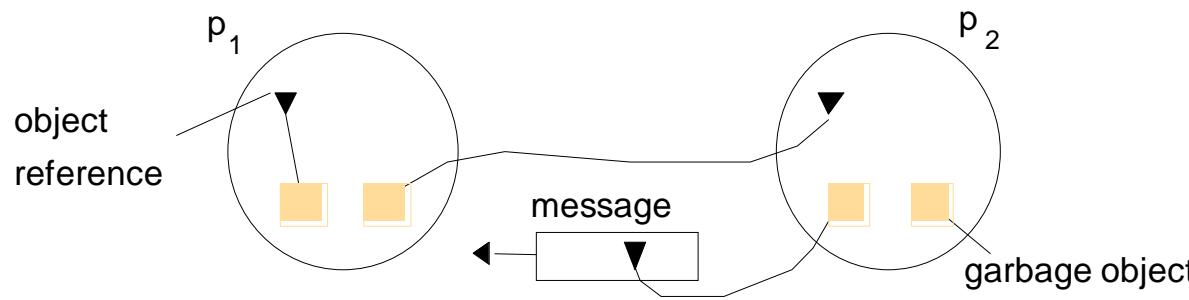
As before
New!

Part 2: Global states

- What and why?
 - Distributed garbage collection
 - Distributed deadlock detection
 - Distributed debugging
- How?
 - Cuts and globally consistent states

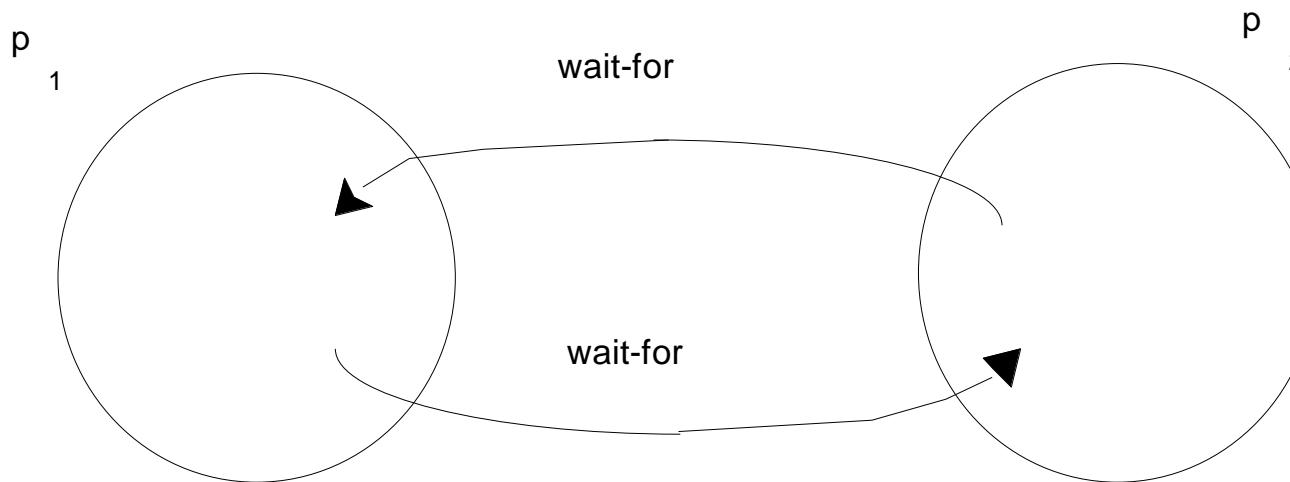
Distributed garbage collection

- Garbage: Objects without active references
- References can be:
 - Local
 - At other processes/nodes (new)
 - In messages (new)
- Need global state (including messages in transit)



Distributed deadlock detection

- Distributed waits-for cycle
- Need global state to detect this



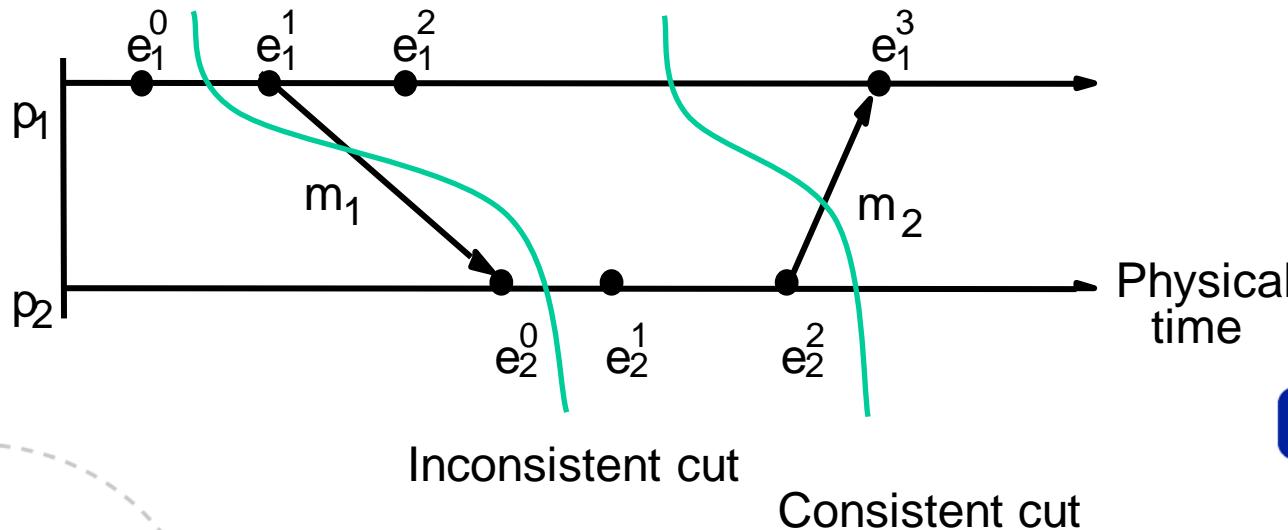
Distributed debugging

- How did variables change at runtime?
 - Example: Has $|x_1 - x_2|$ ever been > 50 ?
- Variables can be located at different processes/nodes
- Problem: Global consistent view of variable values



Cuts (1/2)

- Local history: Events at one process
- Global history: Union of all local histories
- Cut: Subset of global history (local prefix)
- What is the problem?
 - How to find consistent cuts without global time?



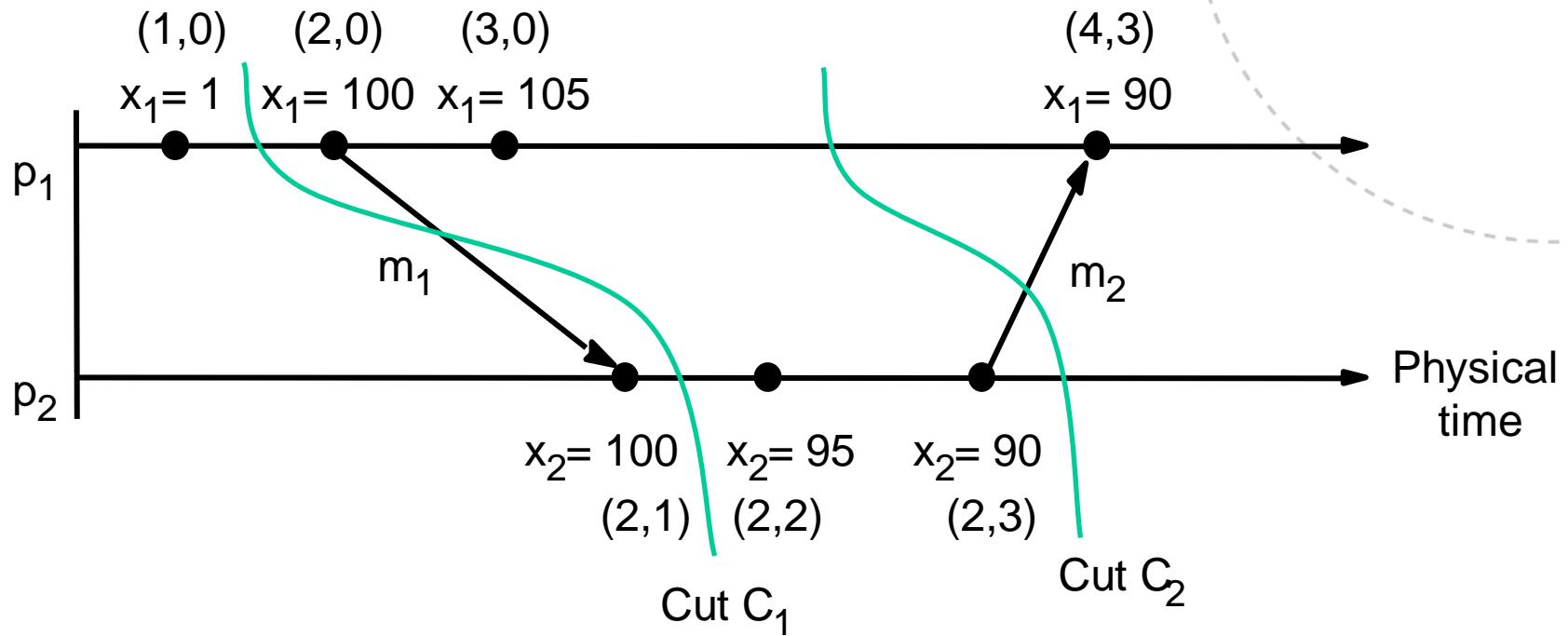
Cuts (2/2)

- A cut C is consistent if
 - For all events $e \in C$, $f \rightarrow e \Rightarrow f \in C$
 - Inconsistent: The system could never have been in this state
 - Consistent \Rightarrow Global consistent state
- Run
 - Global history where the order satisfies all local histories
- Consistent run / linearization
 - All global states passed through are consistent
- Reachable
 - S' is reachable from S if there is a consistent run between them

Distributed debugging

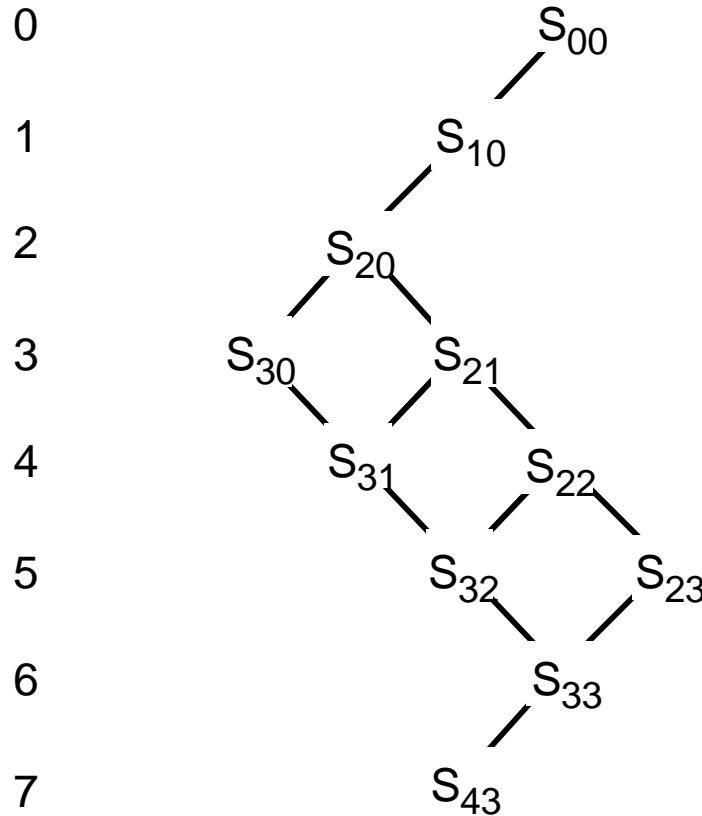
- After the fact: Was a condition true during execution?
- All participants send information about state changes to an external observer
- Global state predicate ϕ
 - Possibly ϕ
 - At least one consistent run passes through a global consistent state where $\phi = \text{true}$
 - Definitely ϕ
 - All consistent runs pass through a global consistent state where $\phi = \text{true}$

Observing global state



- The observer is notified about all state changes
- Uses vector clocks to find global consistent states
- Has $|x_1 - x_2|$ ever been > 50 ?

Alternative consistent runs



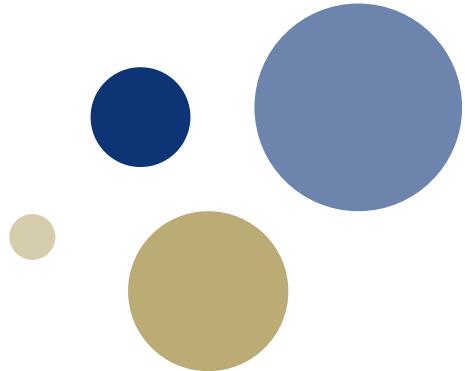
S_{ij} = global state after i events at process 1
and j events at process 2

Possibly ϕ
Definitely ϕ

Passes S where $\phi = \text{true}$
Can't avoid S where $\phi = \text{true}$



Kunnskap for en bedre verden



Dynamo: Amazon's Highly Available Key-value Store

DeCandia et al.

Svein Erik Bratsberg, IDI/NTNU

What is Dynamo?



“Dynamo is a highly available key-value storage system that some of Amazon’s core services use to provide an “always-on” experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.”

Amazon's Services' Requirements



- Strict operational requirements on performance, reliability and efficiency
- Needs to be highly scalable

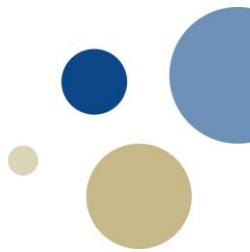
“Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust.”

Why not use a traditional SQL-DBMS?



- Most of Amazon services only store and retrieve data by primary key and do not require the complex querying and management functionality offered by SQL
- Isolation and strong consistency is not the most important properties
- Replication technologies typically choose consistency over availability, making data unavailable instead of inconsistent
 - leading to: Network partitions make data unavailable

System requirements and assumptions



- Simple read and write queries
- Data items uniquely identified by keys
- No operation span multiple data items

Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

Efficiency and Other Assumptions



- Commodity hardware infrastructure.
- Measures in the 99.9th percentile of the distribution.
- Services must be able to configure Dynamo to meet requirements in performance, cost efficiency, availability, and durability guarantees.
- Non-hostile environment: No security related requirements.
- Heterogeneous machines

Amazon's SLA

- A single page request can include requests to over 150 services
- Measure performance by average, median and expected variance (traditionally)
- Amazon wants to build a system that is good for ALL customers, so they focus on the 99.9th percentile.

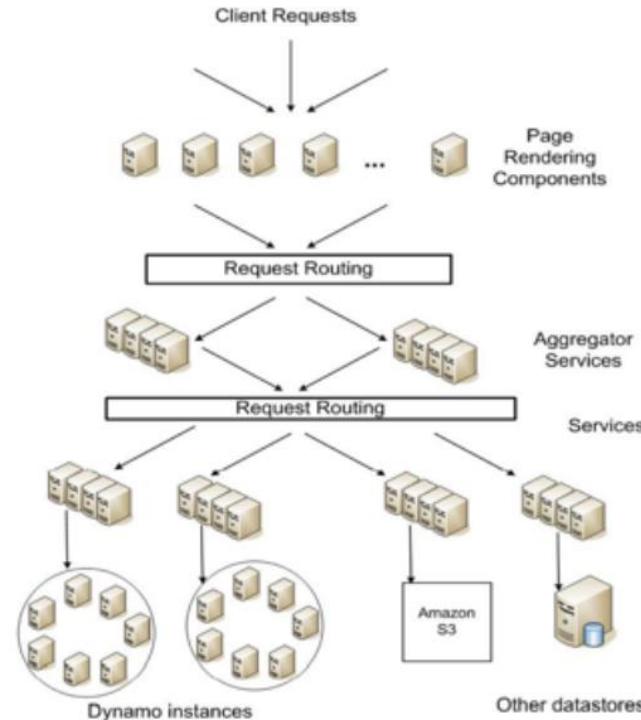


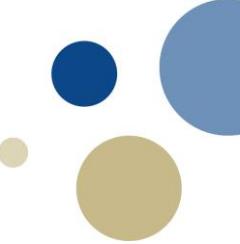
Figure 1: Service-oriented architecture of Amazon's platform

Design Considerations



- Give services control over their system properties, such as durability and consistency, and to let services make their own tradeoffs between functionality, performance and cost-effectiveness.
- Eventually consistent data store.
- “Always writeable” data store - conflict handling happens on read!
- Both client application and data store can resolve conflicts.
- Principles
 - Incremental scalability
 - Symmetry
 - Decentralization
 - Heterogeneity

Key Concepts in Dynamo



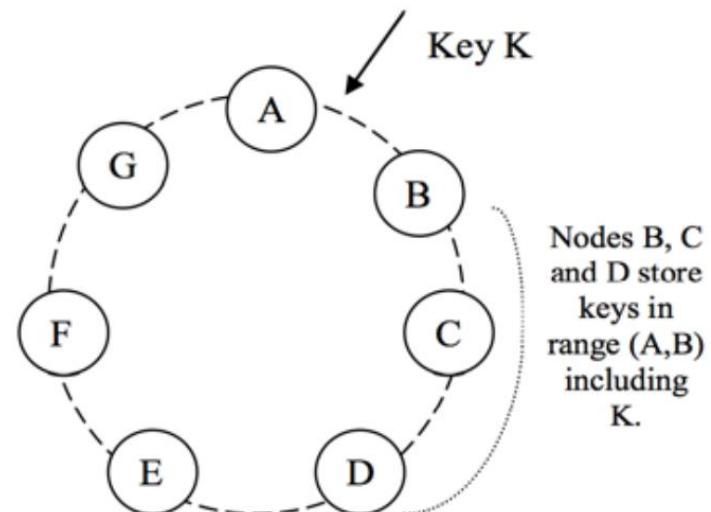
Interface:

- `get(key)`
- `put(context, key, object)`

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

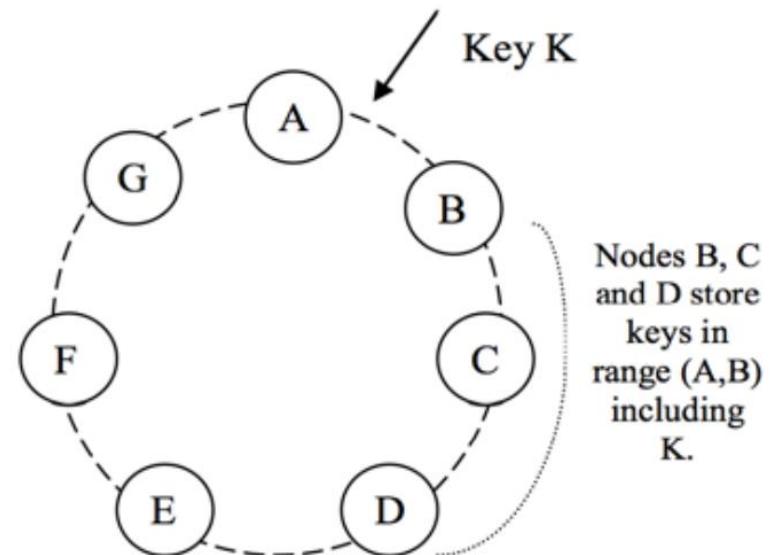
Consistent Hashing

- Departure or arrival of a node only affects its immediate neighbors!
- Assign nodes a random responsible area of the hash ring (circular keyspace)
- Challenges:
 - non-uniform data and load distribution.
 - oblivious to the heterogeneity of hardware
- Solutions:
 - virtual nodes - more powerful computers take responsibility of more virtual nodes



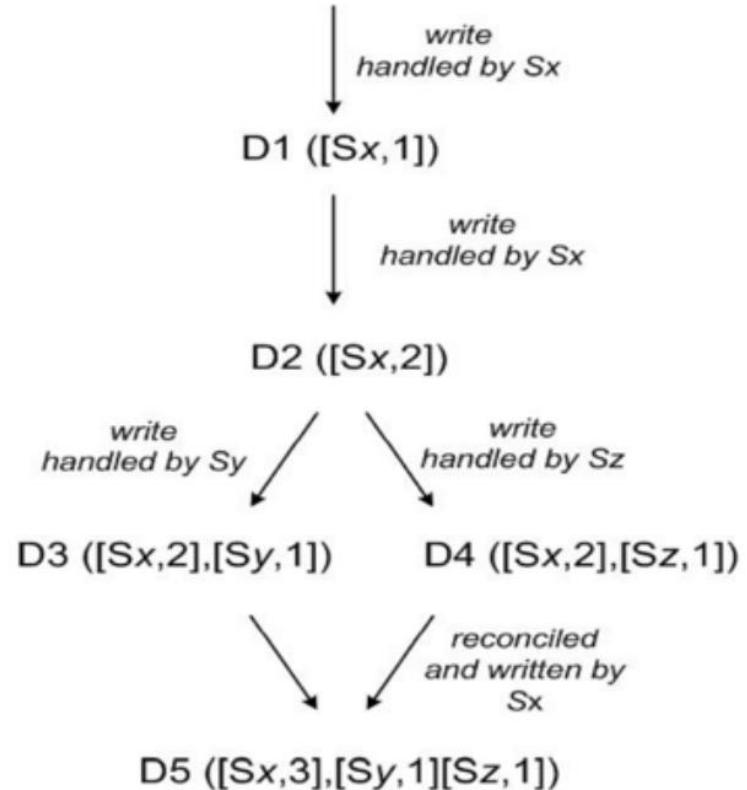
Replication

- Each data item is replicated at N hosts, where N is a parameter configured “per-instance”
- Each key, k, is assigned to a coordinator node
- The coordinator is in charge of the replication of the data items that fall within its range.
- A preference list store the nodes responsible for a key
- The preference list is constructed so the top N nodes are N physical nodes



Data Versioning

- Eventual consistency
- Each modification is a new and immutable version of the data.
- Usually the system handles version branching, but in some cases multiple versions are returned for the client for reconciliation.
- Vector clocks to keep track between different versions



Execution of get() and put()

Generic load-balancer

- Sends request to a random node, which then forwards it to the first of the top N nodes in its preference list
- Client does not have to link any code specific to Dynamo in its application

Partition aware client library

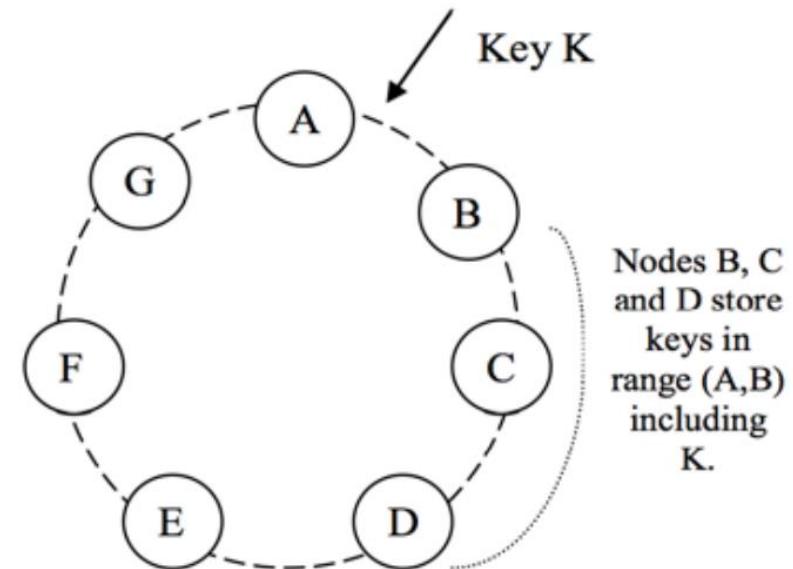
- Sends request to one of the top N nodes in the preference list
- Lower latency

Durability Tuning

- Dynamo uses two key configurable values **R** and **W**.
- **R** is the minimum number of nodes that must participate in a successful read operation.
- **W** is the minimum number of nodes that must participate in a successful write operation. E.g.: if coordinator gets a write, it writes a new version locally then sends the new version to N healthy nodes. If $W-1$ nodes respond, the write is successful.
- **R** and **W** does not necessarily need to be the top N nodes, rather the top N healthy nodes.

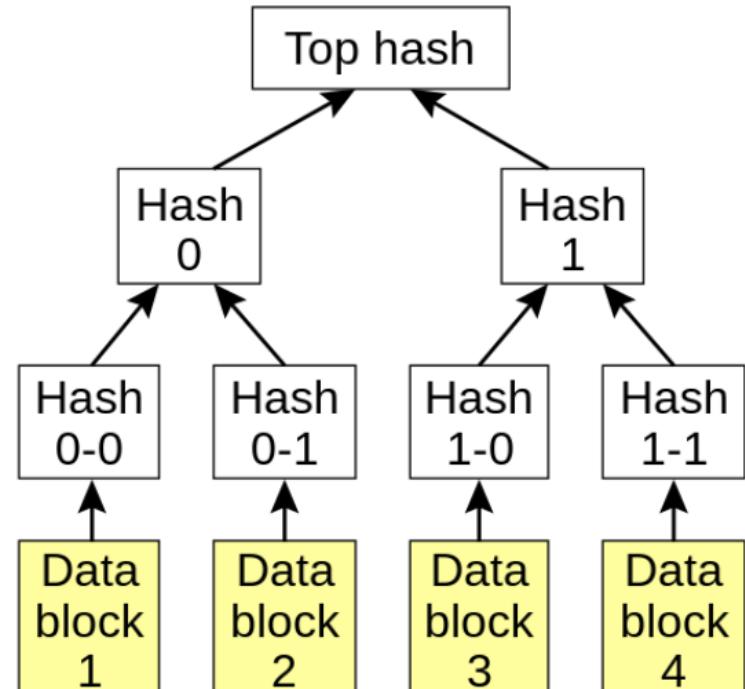
Handling failure - Hinted handoff

- Node A is temporary down during a write operation.
- Node D will receive the replica and a hint in its metadata that the object originally belongs to A.
- This will be kept in a separate local database and once A recovers, the object is transferred back to A and deleted.
 - Works best in low churn environments



Permanent failure – Replica reconstruction

- In some scenarios replicas become unavailable
- To detect inconsistencies between replicas Dynamo uses **Merkle Trees**.
- An entire branch can be checked independently for consistency without transferring entire data set or tree
 - Each node maintains a separate Merkle tree for each key range it hosts.
 - Two nodes exchange the root of the Merkle tree corresponding to the key ranges that they host in common.
 - Use tree traversal to determine if there is need for synchronization



Membership and failure detection

A gossip-based protocol

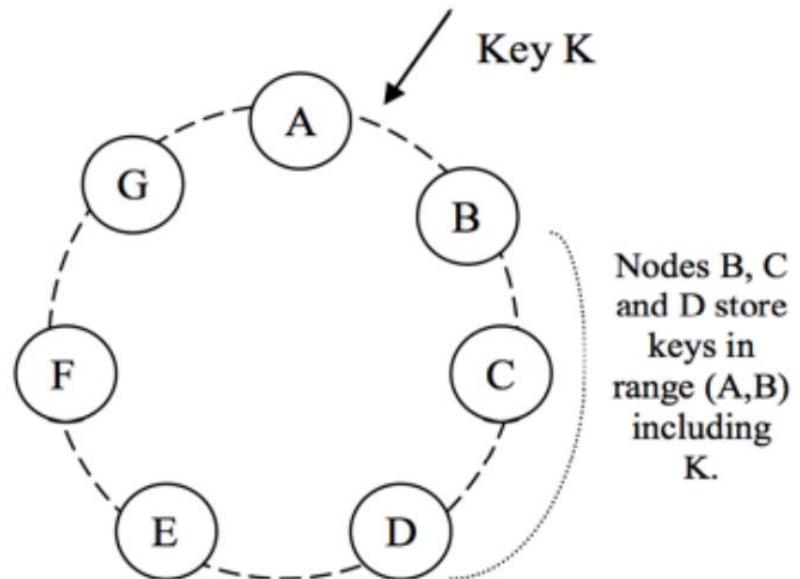
- Each node contacts a peer chosen at random every second
- The two nodes efficiently reconcile their persisted membership change histories.
- Seeds (nodes known to all nodes)
- Local notion of failure detection

Joining the ring:

- When a node starts for the first time, it chooses its set of tokens and maps nodes to their respective token sets.
- Reconciled using the same gossip-based protocol.
- Partitioning and placement information also propagates and each storage node is aware of the token ranges handled by its peers.

Adding and removing nodes

- Node X is added between A and B
- When X is added to the system, it is in charge of storing keys in the ranges $(F, G]$, $(G, A]$ and $(A, X]$.
- This relieves Node B, C and D of some responsibility.
- These nodes will then offer to transfer the appropriate set of keys
- Removing a Node will lead to the process happening in reverse.



Software Architecture

In Dynamo, each storage node has three main software components:

1. request coordination

- The coordinator executes the read and write requests on behalf of clients by collecting data from one or more nodes (in the case of reads) or storing data at one or more nodes (for writes).
- Uses a state machine for each request.
- Handles the request and performs read repairs if needed.
- read-your-writes consistency: don't read stale data

2. membership and failure detection

3. local persistence engine

- Dynamo's local persistence component allows for different storage engines to be plugged in
- determined by access pattern and object size distribution

Experiences

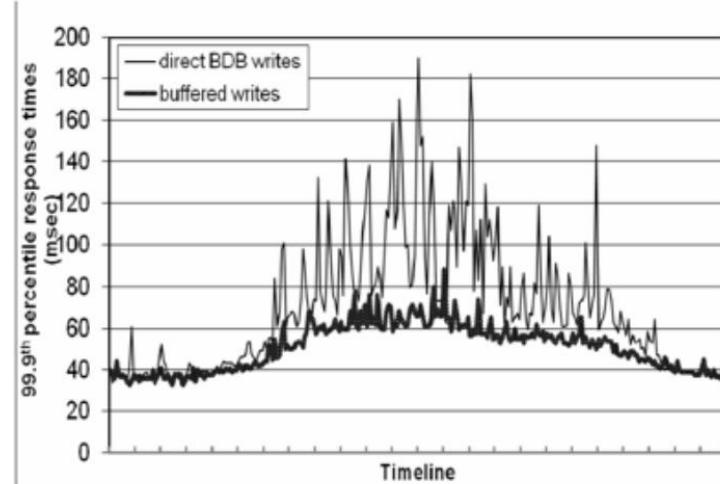
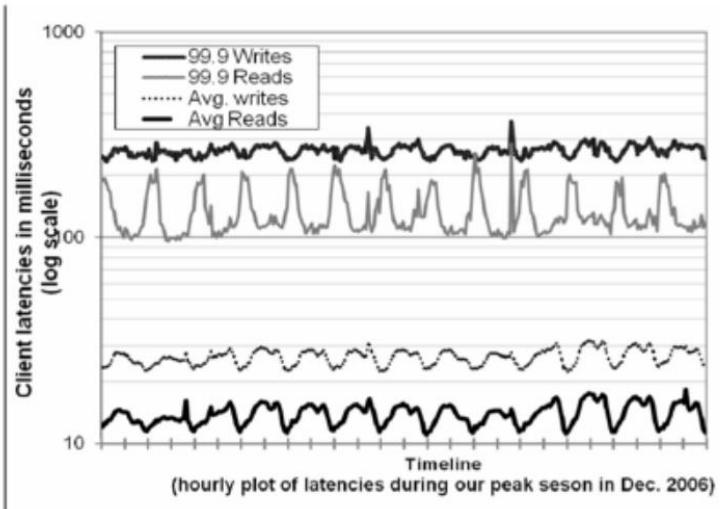
Reconciliation done in different ways:

- Business logic specific
 - The client application handles divergent versions
- Timestamp based
 - Last write wins
- High performance read engine with good durability
 - $R = 1, W = N$

“The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability.”

“The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2).”

Performance



Load distribution

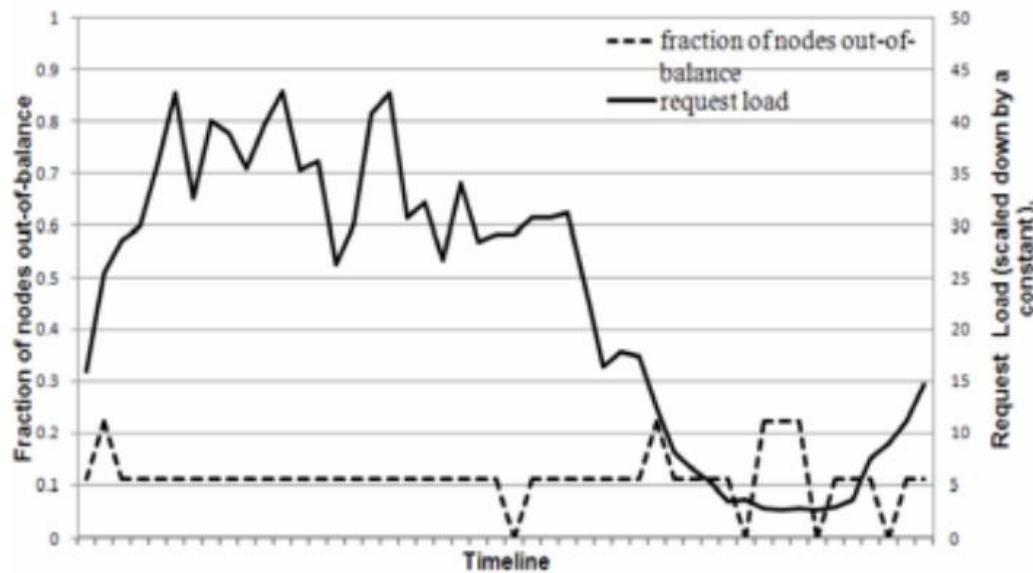
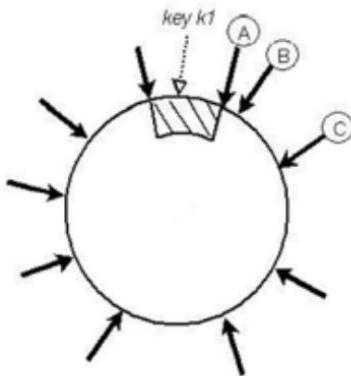


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

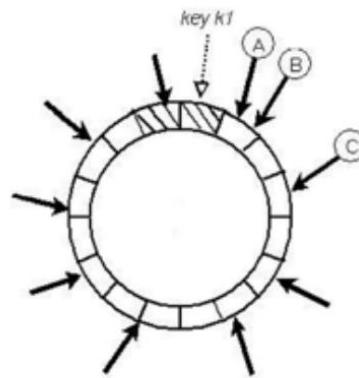
Partitioning strategies



Strategy 1

Random tokens and partitions by token value:

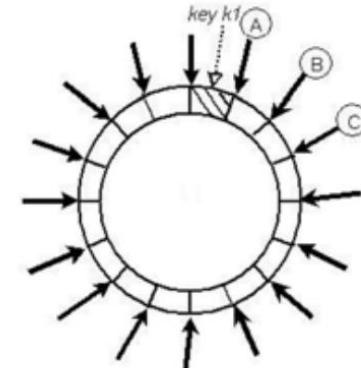
- Ranges vary
- Very resource intensive to add and remove nodes



Strategy 2

Random tokens but fixed partitions:

- Q equally sized partitions
- Consistent ranges
- Decoupling partitioning and partition placement

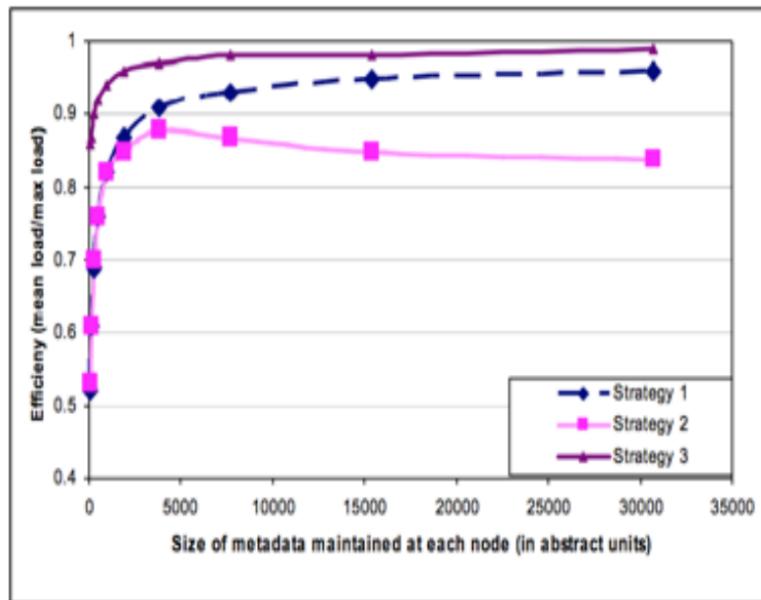


Strategy 3

Q/S tokens per node, fixed partitions:

- S is number of nodes in the system
- Decoupling partitioning and partition placement

Performance and divergent versions



So how big of an issue is divergent versions?

24h shopping cart application:

- 99.94% of requests saw 1 version
- 0.00057% of requests saw 2 versions
- 0.00047% of requests saw 3 versions
- 0.00009% of requests saw 4 versions

This shows that divergent versions are created rarely! Experience also shows that it is increased numbers of concurrent writes that contributes to versioning. However, these are triggered by busy robots and not humans!

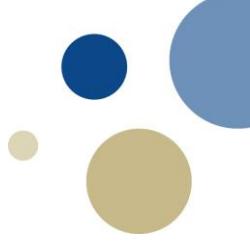
Client- vs server-driven Coordination

Client based:

- Move state machine to client nodes.
- Poll membership information from random node every 10 seconds
- Pull based approach scales better, but in worst case a node can be exposed to stale membership for 10 secs

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

Dynamo's conclusion



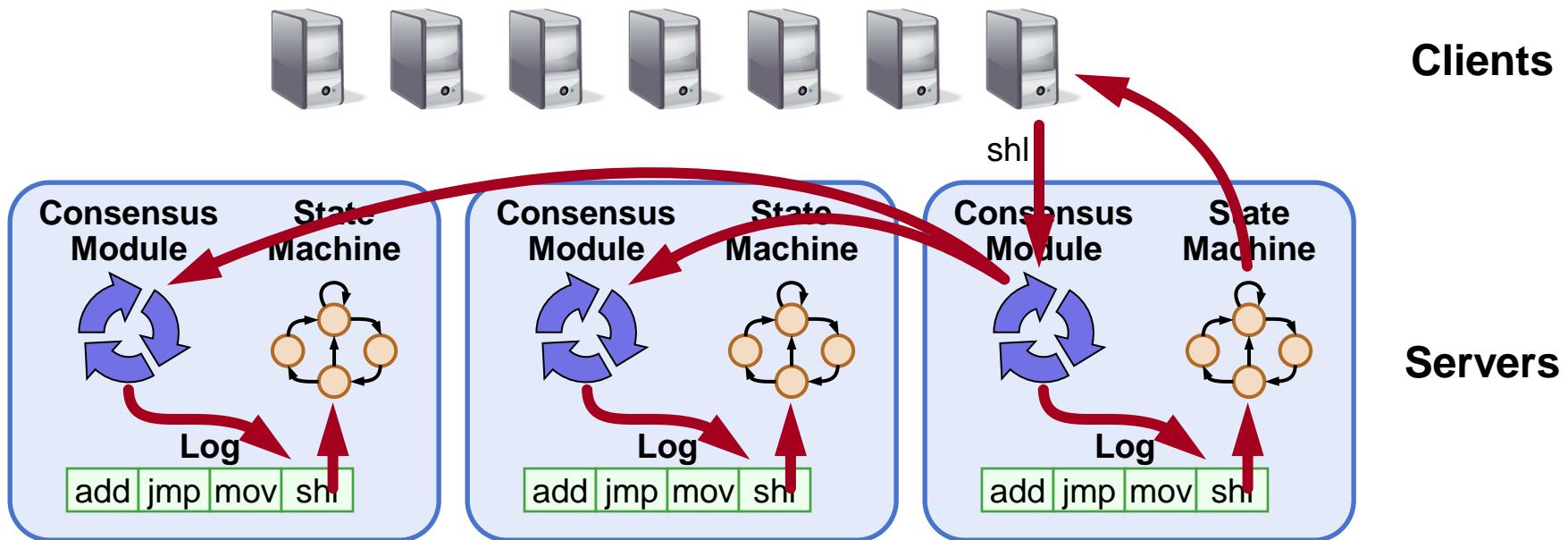
- Dynamo provides a highly available single-key storage system for smaller sized objects.
- Provides easy developer customization by needs in regards to availability, performance and durability.
- Does not scale into tens of thousands!
 - each node has full list of routing info - lots of gossiping!
- May require application logic for data reconciliation
- Dynamo is the model for multiple NoSQL databases
- Available as a cloud service from Amazon

Raft: A Consensus Algorithm for Replicated Logs

Diego Ongaro and John Ousterhout
Stanford University



Goal: Replicated Log



- **Replicated log => replicated state machine**
 - All servers execute same commands in same order
- **Consensus module ensures proper log replication**
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

Approaches to Consensus

Two general approaches to consensus:

- **Symmetric, leader-less:**

- All servers have equal roles
- Clients can contact any server

- **Asymmetric, leader-based:**

- At any given time, one server is in charge, others accept its decisions
- Clients communicate with the leader

- **Raft uses a leader:**

- Decomposes the problem (normal operation, leader changes)
- Simplifies normal operation (no conflicts)
- More efficient than leader-less approaches

Raft Overview

1. Leader election:

- Select one of the servers to act as leader
- Detect crashes, choose new leader

2. Normal operation (basic log replication)

3. Safety and consistency after leader changes

4. Neutralizing old leaders

5. Client interactions

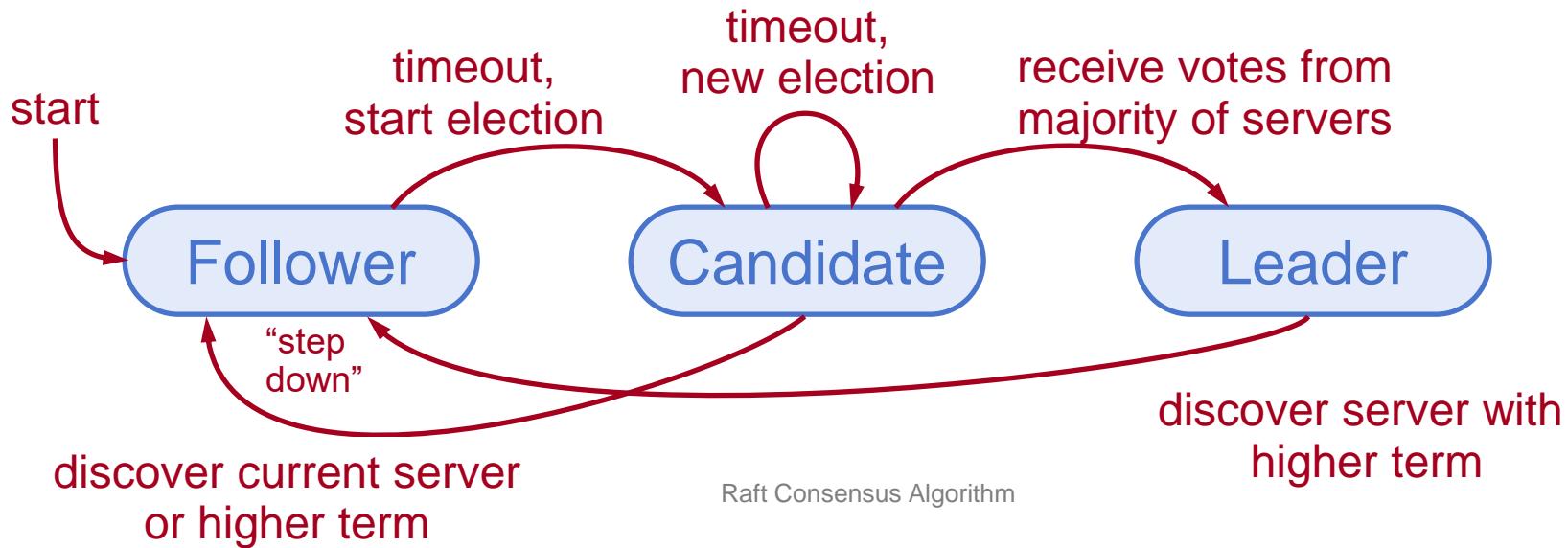
- Implementing linearizable semantics

6. Configuration changes:

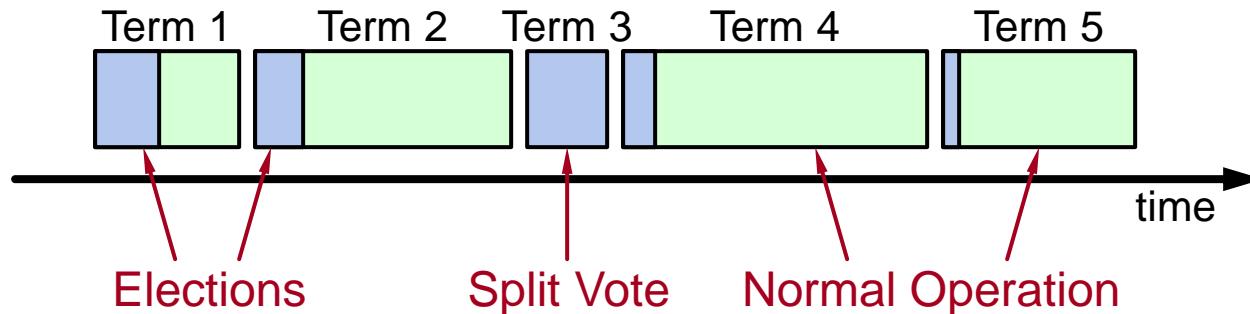
- Adding and removing servers

Server States

- At any given time, each server is either:
 - Leader: handles all client interactions, log replication
 - At most 1 viable leader at a time
 - Follower: completely passive (issues no RPCs, responds to incoming RPCs)
 - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers



Terms



- **Time divided into terms:**
 - Election
 - Normal operation under a single leader
- **At most 1 leader per term**
- **Some terms have no leader (failed election)**
- **Each server maintains **current term** value**
- **Key role of terms: identify obsolete information**

Raft Protocol Summary

Followers <ul style="list-style-type: none">Respond to RPCs from candidates and leaders.Convert to candidate if election timeout elapses without either:<ul style="list-style-type: none">Receiving valid AppendEntries RPC, orGranting vote to candidate	RequestVote RPC <p>Invoked by candidates to gather votes.</p> <p>Arguments:</p> <table><tr><td>candidateId</td><td>candidate requesting vote</td></tr><tr><td>term</td><td>candidate's term</td></tr><tr><td>lastLogIndex</td><td>index of candidate's last log entry</td></tr><tr><td>lastLogTerm</td><td>term of candidate's last log entry</td></tr></table> <p>Results:</p> <table><tr><td>term</td><td>currentTerm, for candidate to update itself</td></tr><tr><td>voteGranted</td><td>true means candidate received vote</td></tr></table> <p>Implementation:</p> <ol style="list-style-type: none">If term > currentTerm, currentTerm \leftarrow term (step down if leader or candidate)If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout	candidateId	candidate requesting vote	term	candidate's term	lastLogIndex	index of candidate's last log entry	lastLogTerm	term of candidate's last log entry	term	currentTerm, for candidate to update itself	voteGranted	true means candidate received vote				
candidateId	candidate requesting vote																
term	candidate's term																
lastLogIndex	index of candidate's last log entry																
lastLogTerm	term of candidate's last log entry																
term	currentTerm, for candidate to update itself																
voteGranted	true means candidate received vote																
Candidates <ul style="list-style-type: none">Increment currentTerm, vote for selfReset election timeoutSend RequestVote RPCs to all other servers, wait for either:<ul style="list-style-type: none">Votes received from majority of servers: become leaderAppendEntries RPC received from new leader: step downElection timeout elapses without election resolution: increment term, start new electionDiscover higher term: step down																	
Leaders <ul style="list-style-type: none">Initialize nextIndex for each to last log index + 1Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeoutsAccept commands from clients, append new entries to local logWhenever last log index \geq nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successfulIf AppendEntries fails because of log inconsistency, decrement nextIndex and retryMark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of serversStep down if currentTerm changes	AppendEntries RPC <p>Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .</p> <p>Arguments:</p> <table><tr><td>term</td><td>leader's term</td></tr><tr><td>leaderId</td><td>so follower can redirect clients</td></tr><tr><td>prevLogIndex</td><td>index of log entry immediately preceding new ones</td></tr><tr><td>prevLogTerm</td><td>term of prevLogIndex entry</td></tr><tr><td>entries[]</td><td>log entries to store (empty for heartbeat)</td></tr><tr><td>commitIndex</td><td>last entry known to be committed</td></tr></table> <p>Results:</p> <table><tr><td>term</td><td>currentTerm, for leader to update itself</td></tr><tr><td>success</td><td>true if follower contained entry matching prevLogIndex and prevLogTerm</td></tr></table> <p>Implementation:</p> <ol style="list-style-type: none">Return if term < currentTermIf term > currentTerm, currentTerm \leftarrow termIf candidate or leader, step downReset election timeoutReturn failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTermIf existing entries conflict with new entries, delete all existing entries starting with first conflicting entryAppend any new entries not already in the logAdvance state machine with newly committed entries	term	leader's term	leaderId	so follower can redirect clients	prevLogIndex	index of log entry immediately preceding new ones	prevLogTerm	term of prevLogIndex entry	entries[]	log entries to store (empty for heartbeat)	commitIndex	last entry known to be committed	term	currentTerm, for leader to update itself	success	true if follower contained entry matching prevLogIndex and prevLogTerm
term	leader's term																
leaderId	so follower can redirect clients																
prevLogIndex	index of log entry immediately preceding new ones																
prevLogTerm	term of prevLogIndex entry																
entries[]	log entries to store (empty for heartbeat)																
commitIndex	last entry known to be committed																
term	currentTerm, for leader to update itself																
success	true if follower contained entry matching prevLogIndex and prevLogTerm																
Persistent State <p>Each server persists the following to stable storage synchronously before responding to RPCs:</p> <table><tr><td>currentTerm</td><td>latest term server has seen (initialized to 0 on first boot)</td></tr><tr><td>votedFor</td><td>candidateId that received vote in current term (or null if none)</td></tr><tr><td>log[]</td><td>log entries</td></tr></table>	currentTerm	latest term server has seen (initialized to 0 on first boot)	votedFor	candidateId that received vote in current term (or null if none)	log[]	log entries											
currentTerm	latest term server has seen (initialized to 0 on first boot)																
votedFor	candidateId that received vote in current term (or null if none)																
log[]	log entries																
Log Entry <table><tr><td>term</td><td>term when entry was received by leader</td></tr><tr><td>index</td><td>position of entry in the log</td></tr><tr><td>command</td><td>command for state machine</td></tr></table>	term	term when entry was received by leader	index	position of entry in the log	command	command for state machine											
term	term when entry was received by leader																
index	position of entry in the log																
command	command for state machine																

Heartbeats and Timeouts

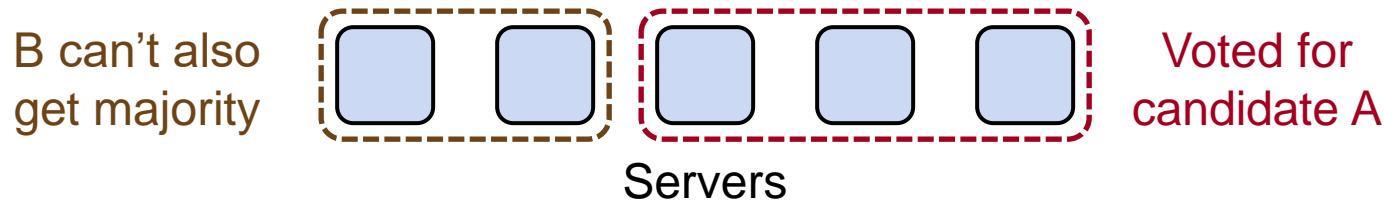
- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send **heartbeats** (empty AppendEntries RPCs) to maintain authority
- If **electionTimeout** elapses with no RPCs:
 - Follower assumes leader has crashed
 - Follower starts new election
 - Timeouts typically 100-500ms

Election Basics

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs to all other servers, retry until either:
 1. Receive votes from majority of servers:
 - Become leader
 - Send AppendEntries heartbeats to all other servers
 2. Receive RPC from valid leader:
 - Return to follower state
 3. No-one wins election (election timeout elapses):
 - Increment term, start new election

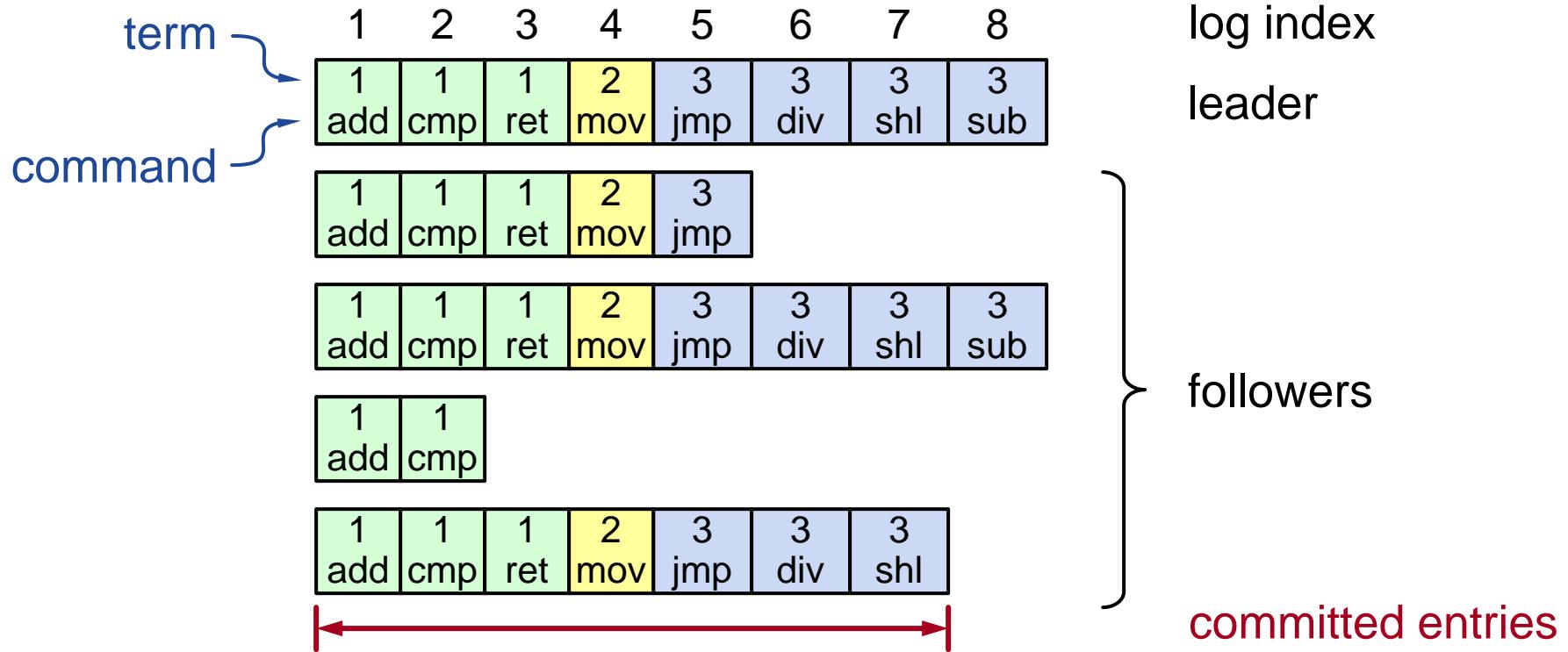
Elections, cont'd

- **Safety:** allow at most one winner per term
 - Each server gives out only one vote per term (persist on disk)
 - Two different candidates can't accumulate majorities in same term



- **Liveness:** some candidate must eventually win
 - Choose election timeouts randomly in $[T, 2T]$
 - One server usually times out and wins election before others wake up
 - Works well if $T \gg$ broadcast time

Log Structure



- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes
- Entry **committed** if known to be stored on majority of servers
 - Durable, will eventually be executed by state machines

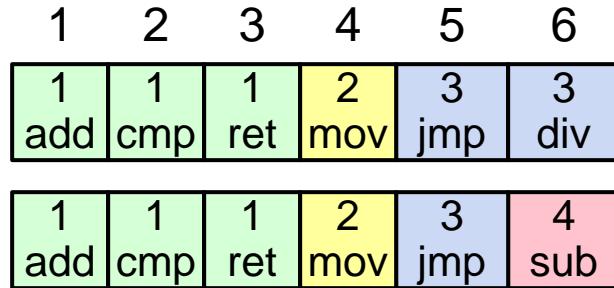
Normal Operation

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
 - Leader passes command to its state machine, returns result to client
 - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
 - Followers pass committed commands to their state machines
- Crashed/slow followers?
 - Leader retries RPCs until they succeed
- Performance is optimal in common case:
 - One successful RPC to any majority of servers

Log Consistency

High level of coherency between logs:

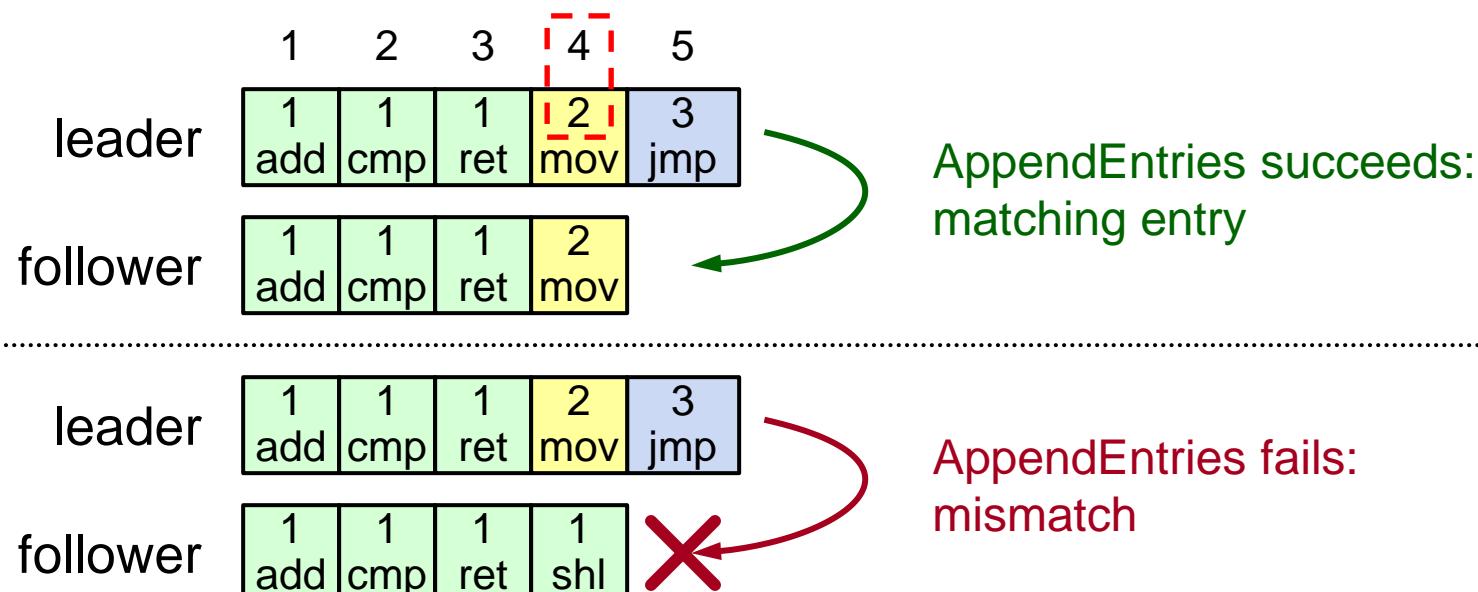
- **If log entries on different servers have same index and term:**
 - They store the same command
 - The logs are identical in all preceding entries



- **If a given entry is committed, all preceding entries are also committed**

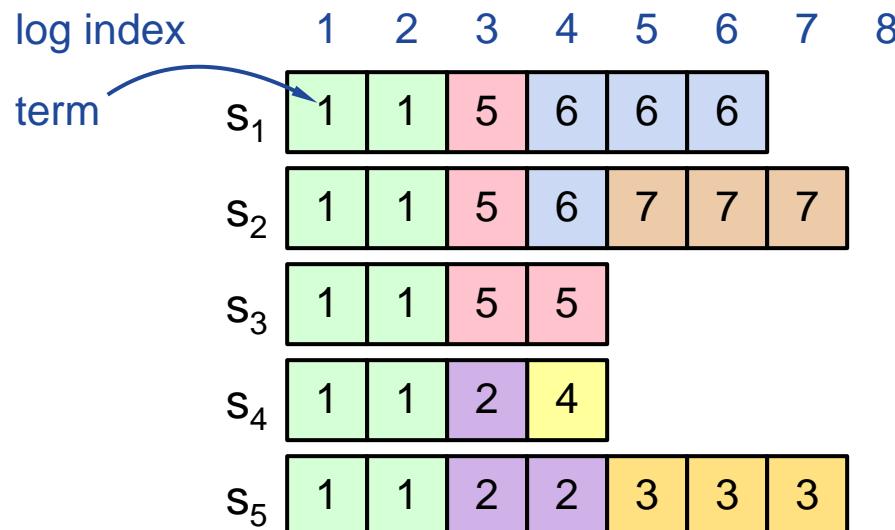
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an **induction step**, ensures coherency



Leader Changes

- At beginning of new leader's term:
 - Old leader may have left entries partially replicated
 - No special steps by new leader: just start normal operation
 - Leader's log is “the truth”
 - Will eventually make follower's logs identical to leader's
 - Multiple crashes can leave many extraneous log entries:



Safety Requirement

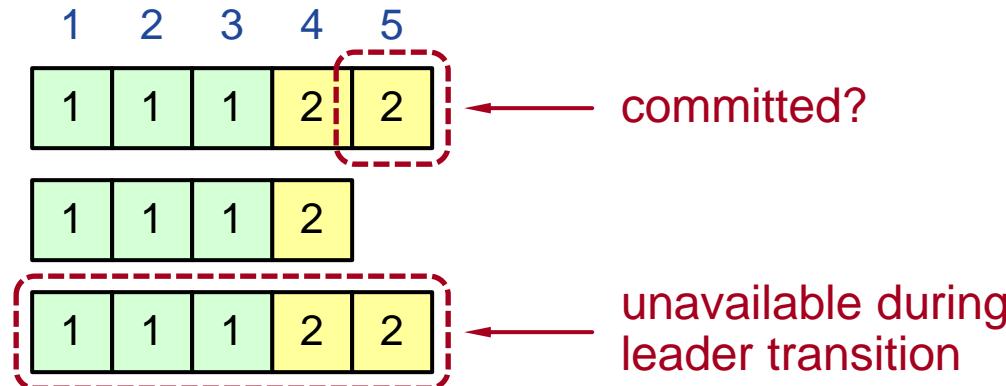
Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry

- **Raft safety property:**
 - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
- **This guarantees the safety requirement**
 - Leaders never overwrite entries in their logs
 - Only entries in the leader's log can be committed
 - Entries must be committed before applying to state machine



Picking the Best Leader

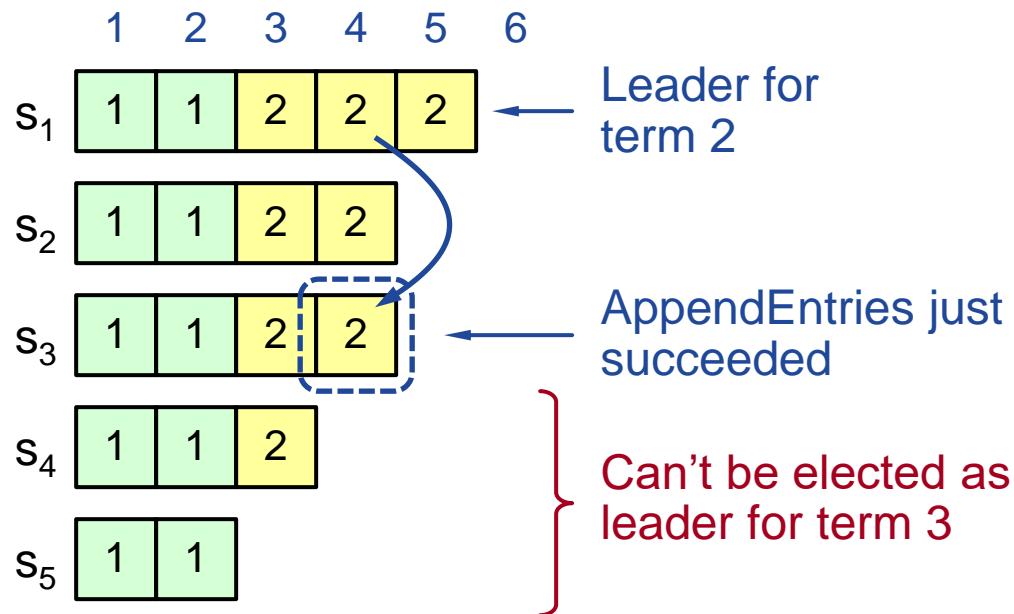
- Can't tell which entries are committed!



- During elections, choose candidate with log most likely to contain all committed entries
 - Candidates include log info in RequestVote RPCs (index & term of last log entry)
 - Voting server V denies vote if its log is “more complete”:
 $(\text{lastTerm}_V > \text{lastTerm}_C) \text{ || } (\text{lastTerm}_V == \text{lastTerm}_C) \text{ && } (\text{lastIndex}_V > \text{lastIndex}_C)$
 - Leader will have “most complete” log among electing majority

Committing Entry from Current Term

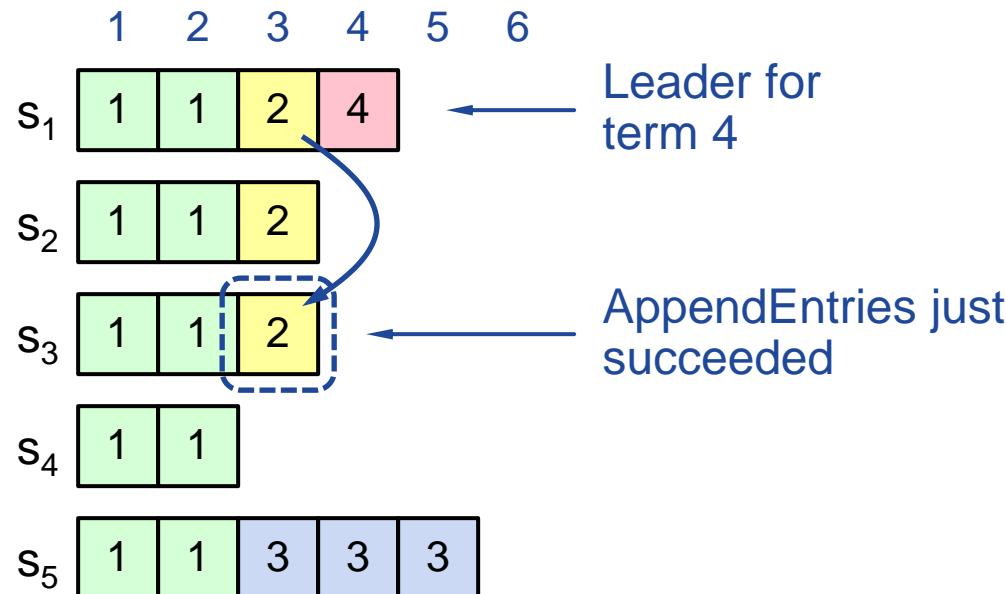
- Case #1/2: Leader decides entry in current term is committed



- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term

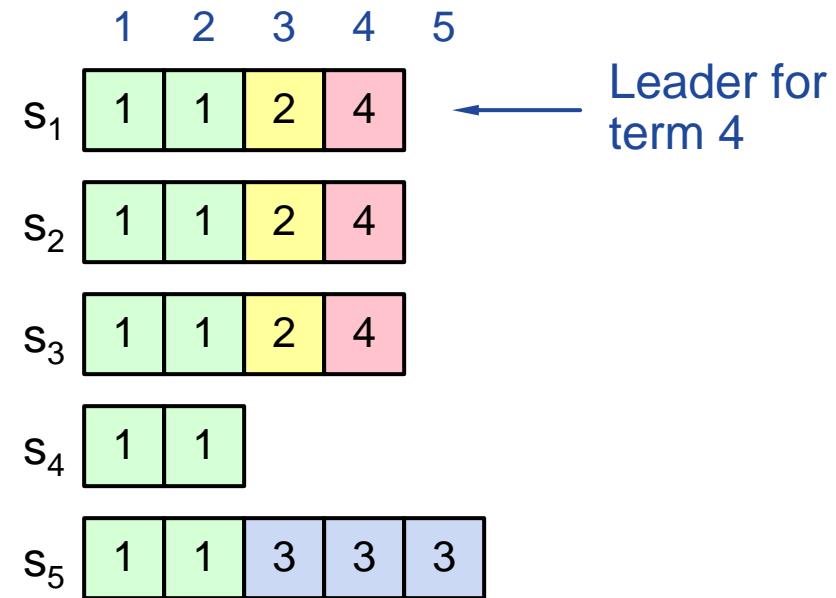
- Case #2/2: Leader is trying to finish committing entry from an earlier term



- Entry 3 **not safely committed**:
 - s₅ can be elected as leader for term 5
 - If elected, it will overwrite entry 3 on s₁, s₂, and s₃!

New Commitment Rules

- For a leader to decide an entry is committed:
 - Must be stored on a majority of servers
 - At least one new entry from leader's term must also be stored on majority of servers
- Once entry 4 committed:
 - s_5 cannot be elected leader for term 5
 - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe

Log Inconsistencies

Leader changes can result in log inconsistencies:

log index

1 2 3 4 5 6 7 8 9 10 11 12

leader for
term 8

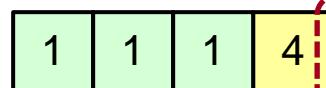


possible
followers

(a)



(b)



(c)



(d)



(e)



(f)

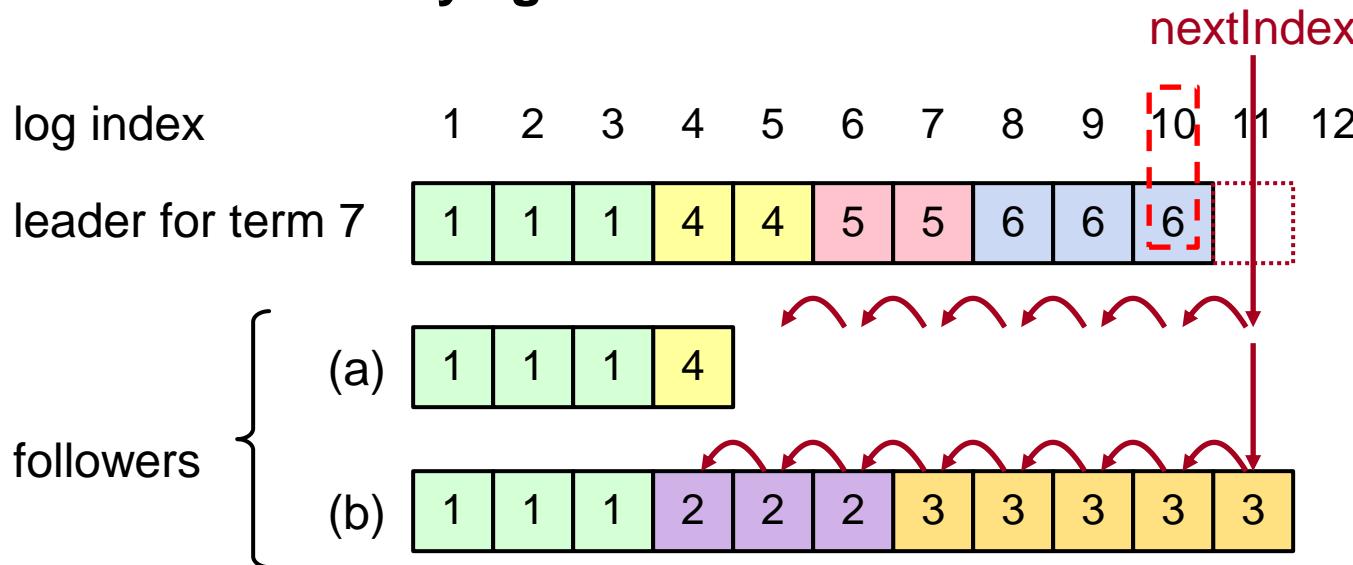


Missing
Entries

Extraneous
Entries

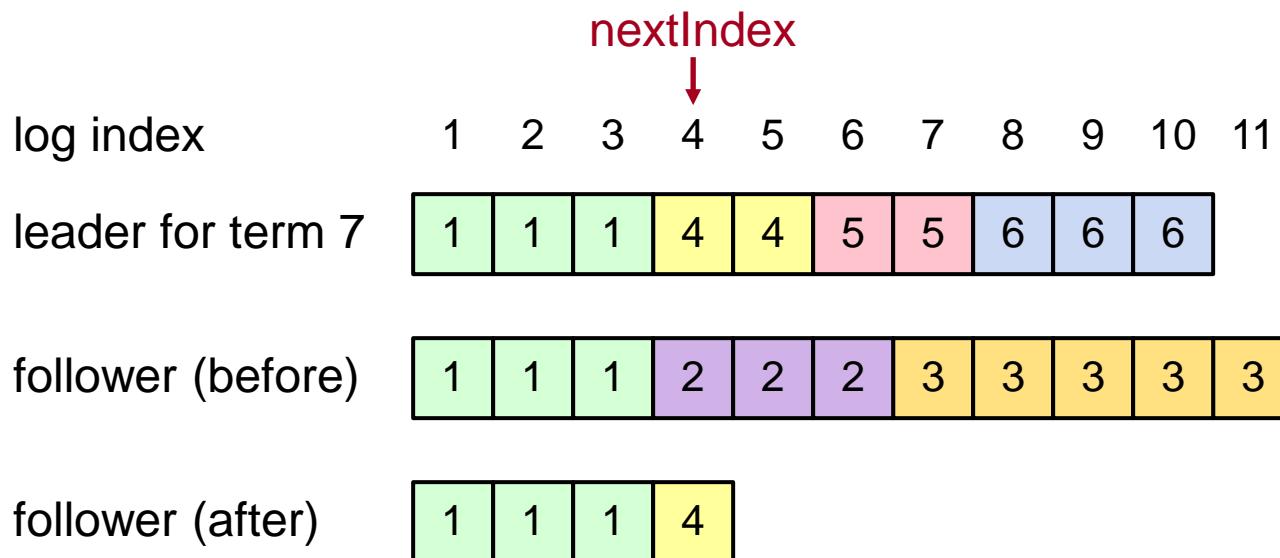
Repairing Follower Logs

- New leader must make follower logs consistent with its own
 - Delete extraneous entries
 - Fill in missing entries
- Leader keeps **nextIndex** for each follower:
 - Index of next log entry to send to that follower
 - Initialized to ($1 + \text{leader's last index}$)
- When AppendEntries consistency check fails, decrement **nextIndex** and try again:



Repairing Logs, cont'd

- When follower overwrites inconsistent entry, it deletes all subsequent entries:



Neutralizing Old Leaders

- **Deposed leader may not be dead:**
 - Temporarily disconnected from network
 - Other servers elect a new leader
 - Old leader becomes reconnected, attempts to commit log entries
- **Terms used to detect stale leaders (and candidates)**
 - Every RPC contains term of sender
 - If sender's term is older, RPC is rejected, sender reverts to follower and updates its term
 - If receiver's term is older, it reverts to follower, updates its term, then processes RPC normally
- **Election updates terms of majority of servers**
 - Deposed server cannot commit new log entries

Client Protocol

- **Send commands to leader**
 - If leader unknown, contact any server
 - If contacted server not leader, it will redirect to leader
- **Leader does not respond until command has been logged, committed, and executed by leader's state machine**
- **If request times out (e.g., leader crash):**
 - Client reissues command to some other server
 - Eventually redirected to new leader
 - Retry request with new leader

Client Protocol, cont'd

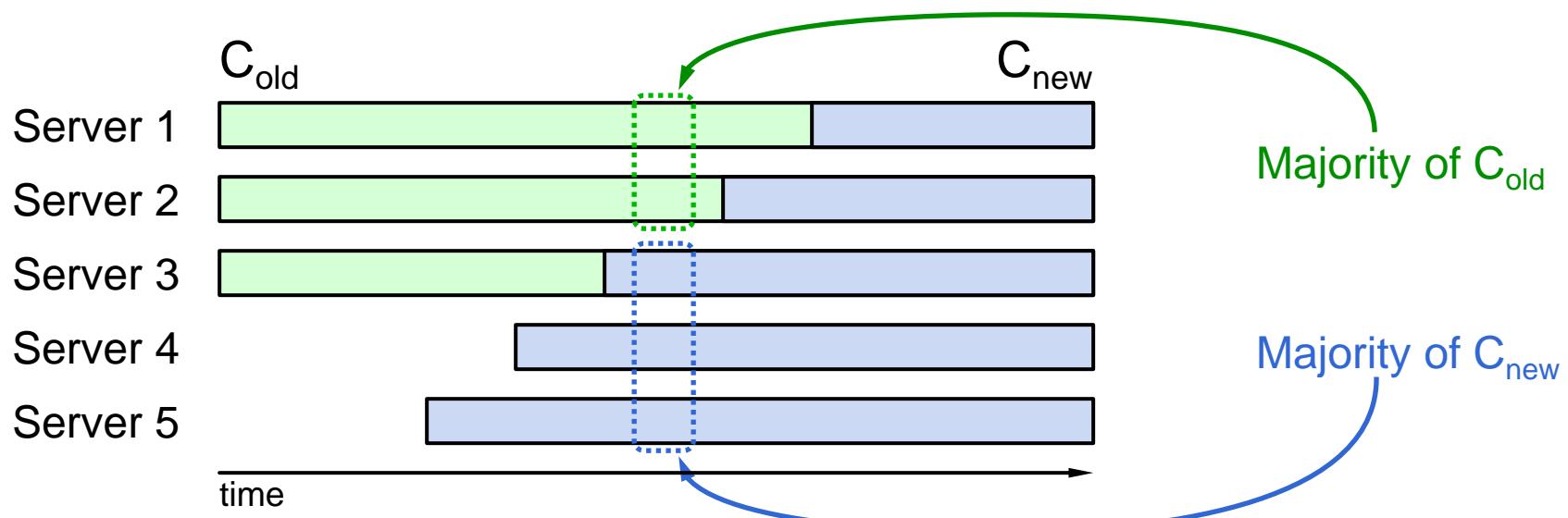
- **What if leader crashes after executing command, but before responding?**
 - Must not execute command twice
- **Solution: client embeds a unique id in each command**
 - Server includes id in log entry
 - Before accepting command, leader checks its log for entry with that id
 - If id found in log, ignore new command, return response from old command
- **Result: exactly-once semantics as long as client doesn't crash**

Configuration Changes

- **System configuration:**
 - ID, address for each server
 - Determines what constitutes a majority
- **Consensus mechanism must support changes in the configuration:**
 - Replace failed machine
 - Change degree of replication

Configuration Changes, cont'd

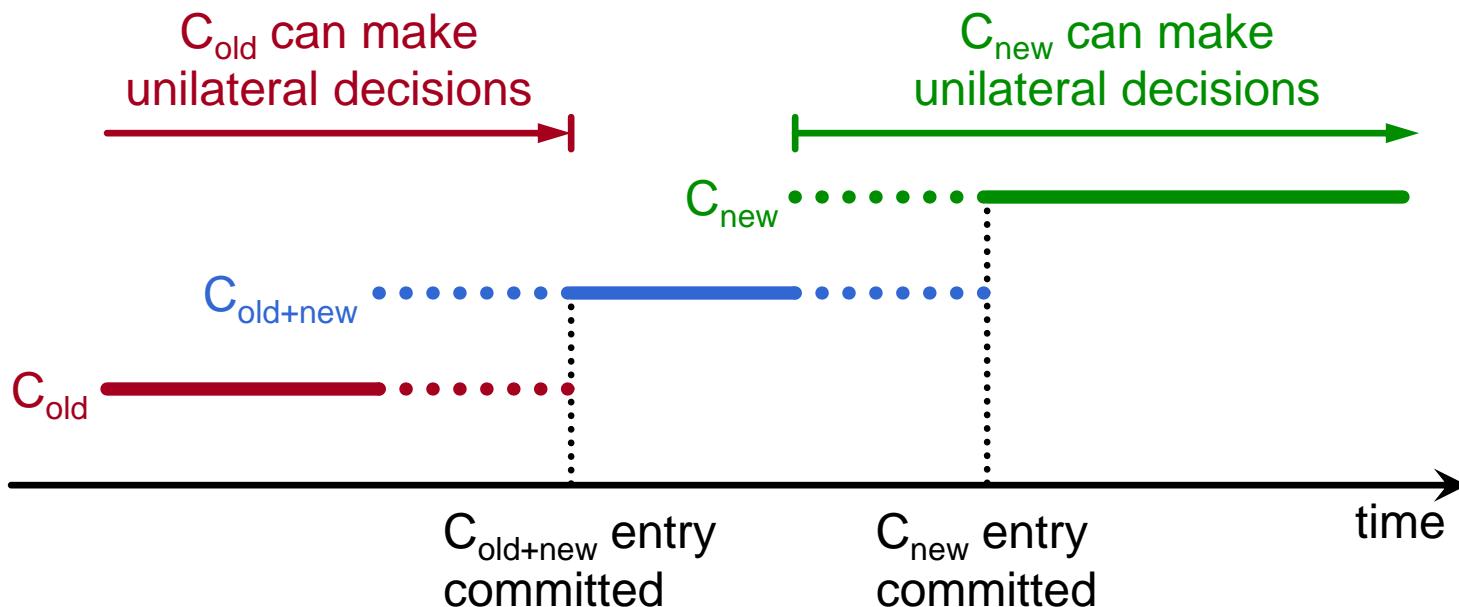
Cannot switch directly from one configuration to another: **conflicting majorities** could arise



Joint Consensus

- Raft uses a 2-phase approach:

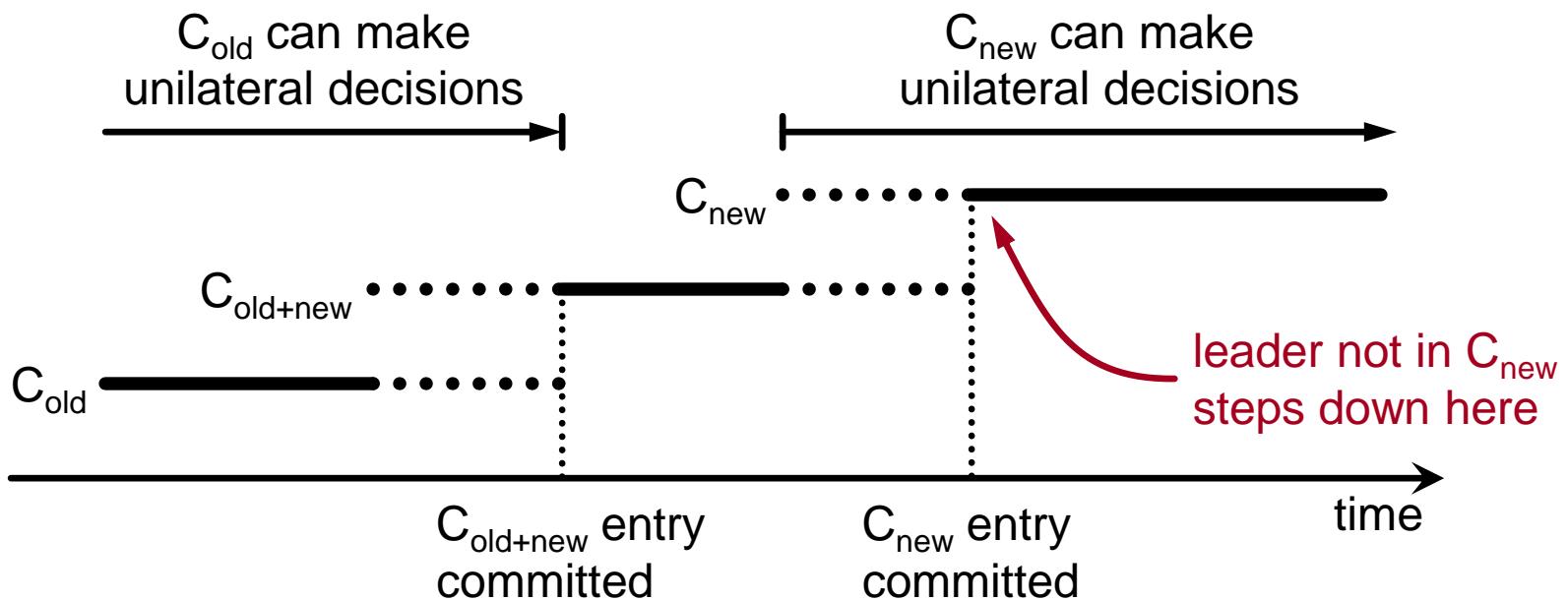
- Intermediate phase uses **joint consensus** (need majority of both old and new configurations for elections, commitment)
- Configuration change is just a log entry; applied immediately on receipt (committed or not)
- Once joint consensus is committed, begin replicating log entry for final configuration



Joint Consensus, cont'd

- Additional details:

- Any server from either configuration can serve as leader
- If current leader is not in C_{new} , must step down once C_{new} is committed.



Raft Summary

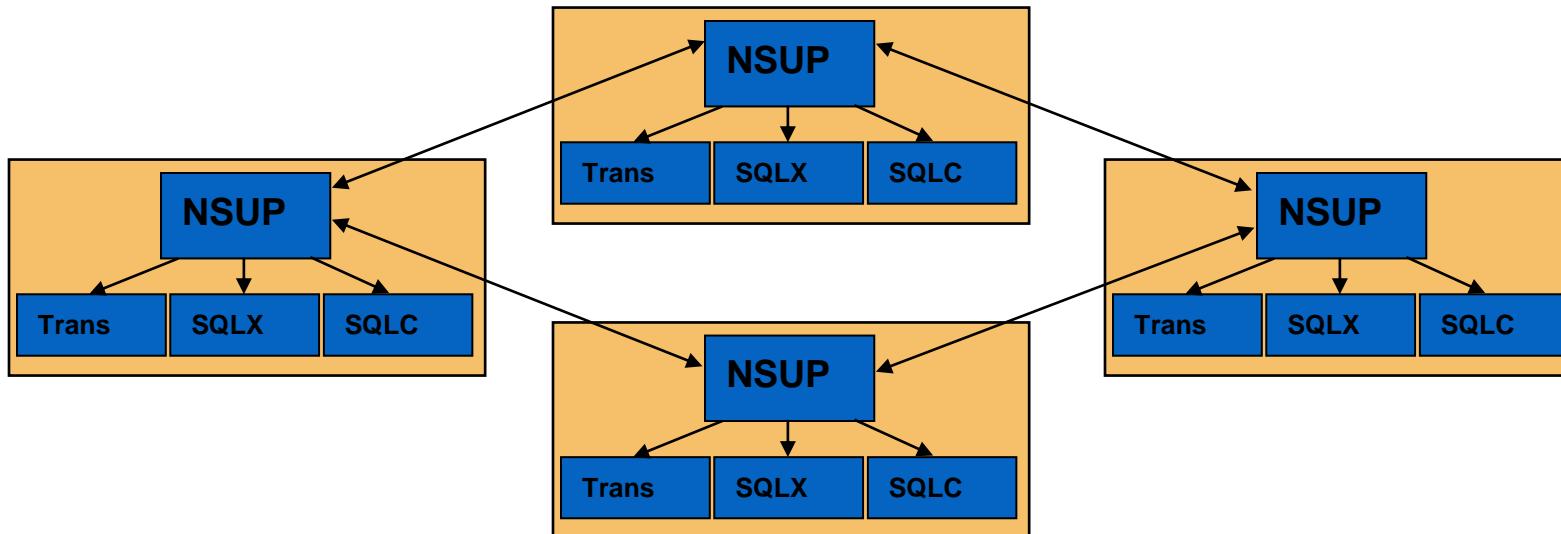
- 1. Leader election**
- 2. Normal operation**
- 3. Safety and consistency**
- 4. Neutralize old leaders**
- 5. Client protocol**
- 6. Configuration changes**

Clustra's Virtual Partition Protocol

Svein Erik Bratsberg

Clustra's Virtual Partition Protocol

- Node supervisors are responsible for
 - I am alive
 - Virtual partition protocol
 - Starting processes



Virtual Partition Protocol

- El Abbadi, Skeen, Cristian: PODS 1985
- Only nodes assigned to the same VP may communicate: messages tagged w/currVP

