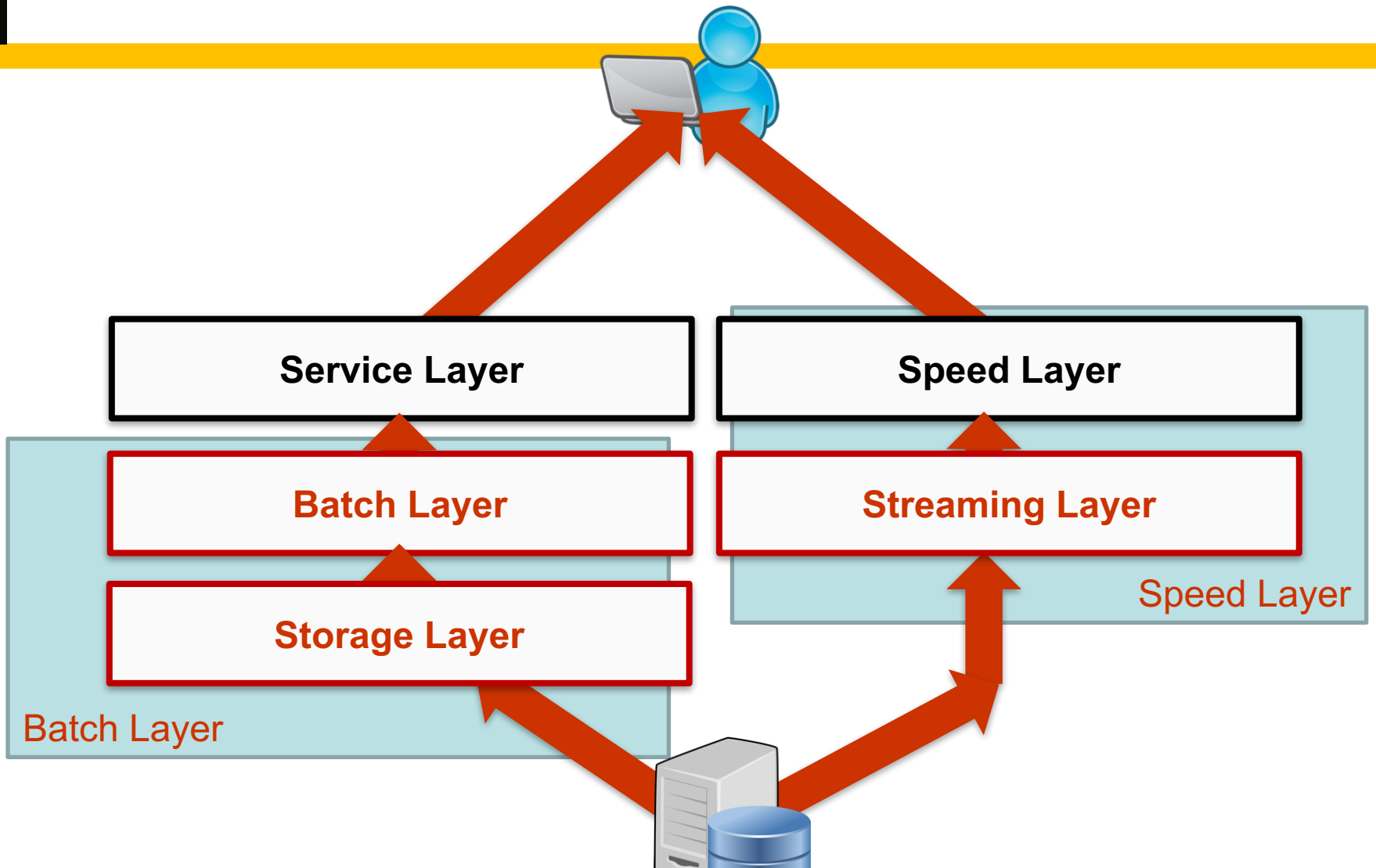
IT UNIVERSITY OF COPENHAGEN

Big Data Management
Serving Views

Björn Þór Jónsson

Framework
Lambda Architecture



Outline

- Service Layer
  - Requirements
  - Design
- Speed Layer (requirements, design)
  - NoSQL systems
  - CAP theorem and PACELC formulation
- Practical concerns

Service Layer

Input

- Views from Batch Layer
- Written in batches
  - No random writes!

Goals

- Low latency
  - … and high throughput
  - Careful IO management
- Linear scale-out
  - Distributed processing

No Random Writes

Random Writes (Updates)

- Consistency
  - Locking
  - Versions
- Recovery
  - Logging
- Compaction
  - Removing versions/logs

Service Layer

- Immutable data → no inconsistency
- Storage Layer → keeps all versions
- Batch Layer → recover views

Low Latency

Sources of Latency?

- Data on disk?
  - Random reads
- Distributed computation
  - Slowest worker

Solutions?

- Organize data for sequential read
  - Indexes to avoid full scans
- Organize data for single worker access
  - Redundancy can help load balancing

Apache Drill

- One tool for service layer
- Reads data from HDFS, HBase, …
  - Ships computation to data
  - Uses functionality of data source
  - Very small code-base!
- Provides schema-free SQL support
  - Exercise today...

Outline

- Service layer (requirements, design)
- Speed layer
  - Requirements, Design
  - NoSQL systems
  - CAP theorem and PACELC formulation
- Practical concerns

Speed Layer

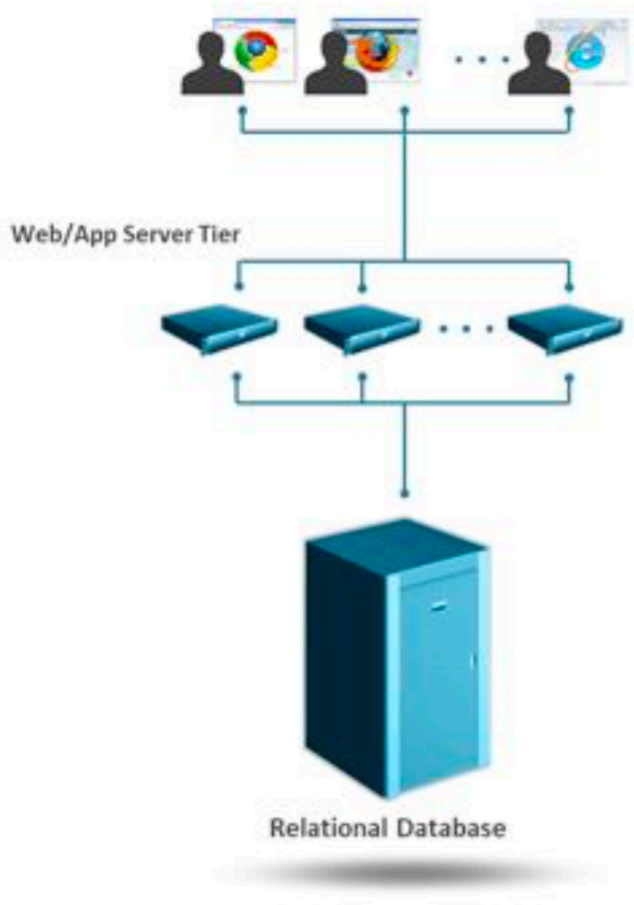
Input

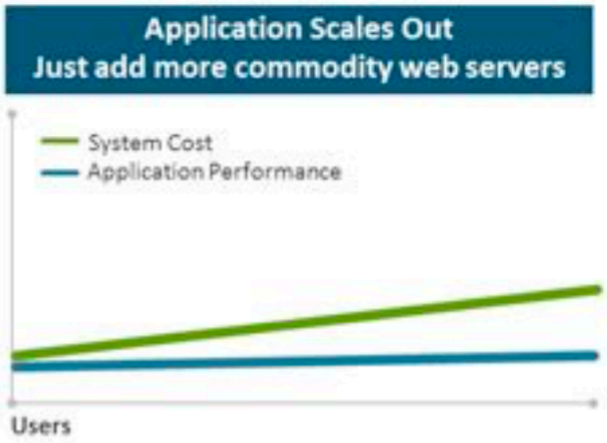
- New data
- Written incrementally
  - Random writes!
- **Small data set**
  - Data deleted once in Service Layer

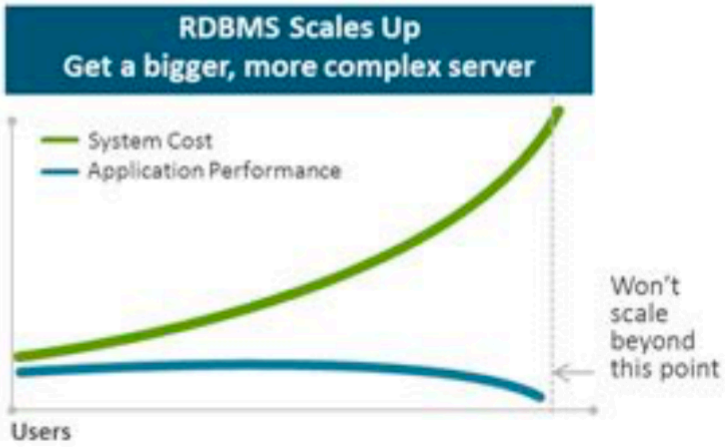
Goals

- Low latency
  - … and high throughput
  - Careful IO management
  - Replicas for load-balancing
- Linear scale-out
  - Distributed processing
- **NoSQL solution!**

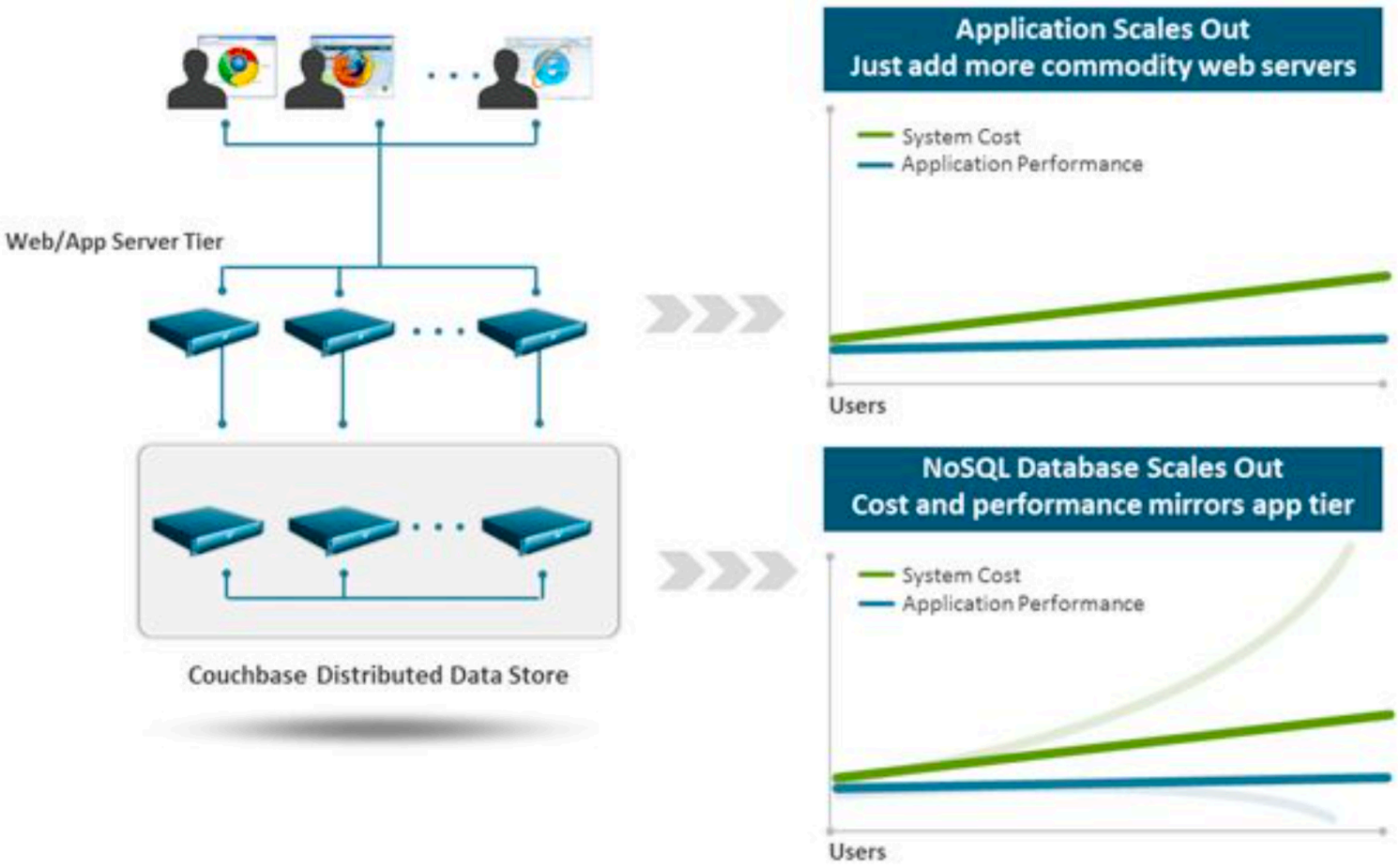
NoSQL Motivation







Distributed DBMS



NoSQL

- Data model
  - **Not relational**
  - No formally described structure (schema-less)
- Interface
  - Not only SQL
  - Proprietary, REST, CQL etc.
- Architecture
  - Usually distributed

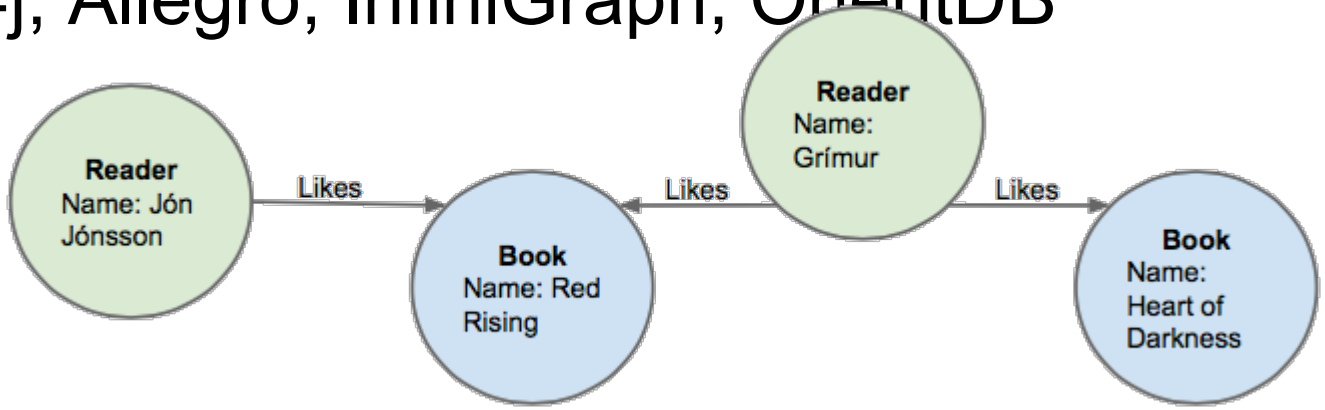
- Mostly not ACID compliant
- Mostly open source
- Consistency/Availability tradeoff

Partial NoSQL timeline

- Pre 1970s - noSQL-like databases
  - network model, hierarchical model
- 1964 - MultiValue data model developed at TRW (PICK)
- 1979 - DBM released (AT&T, Ken Thompson)
- 1989 - Lotus Notes released (IBM)
- 1998 - noSQL name (Carlo Strozzi)
- 2000 - neo4j released
- 2004 - Google BigTable development starts
- 2005 - CouchDB released
- 2007 - Amazon's Dynamo paper released
- 2008 - Facebook's Cassandra open sourced
- 2012 - Amazon's DynamoDB released

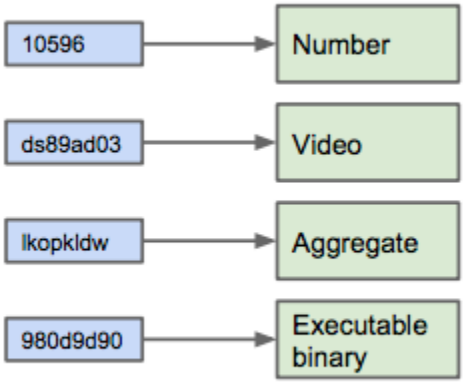
Graph Stores

- Nodes = Entities
- Edges = Relationships, directional
- Properties = Entity descriptors
- Examples
  - neo4j, Allegro, InfiniGraph, OrientDB



Key-Value Stores

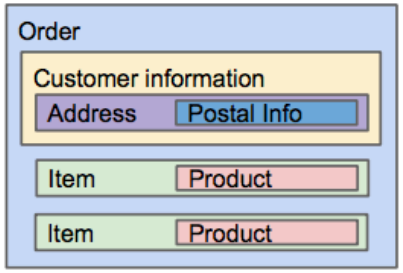
- Associative Array
  - Unique key points to a value
  - Value contents unknown
- Can not be queried
  - GET / PUT only
  - Value can be an aggregate structure
- Examples
  - Riak, Voldemort, MemcacheDB, redis

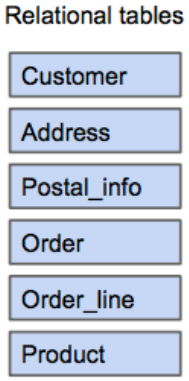


Document Stores

- Each value is a document
  - Most often JSON
  - Unique keys used for retrieval
- You can query into the document
  - More transparent than key-value stores
- The document is an aggregate structure

```
{
"order_id" : 3294,
"customer" : {
        "ssn" : "1111111119",
        "name" : "Jón Jónsson",
        "address" : {
                "street" : "Skúlagata 18",
                "postal_code" : "101"
        }
},
"line-items" : [
        {"product" : "Sófasett", "price" : 300000, "discount" : 45000},
        {"product" : "Lampi", "price" : 40000 }
]
}
```

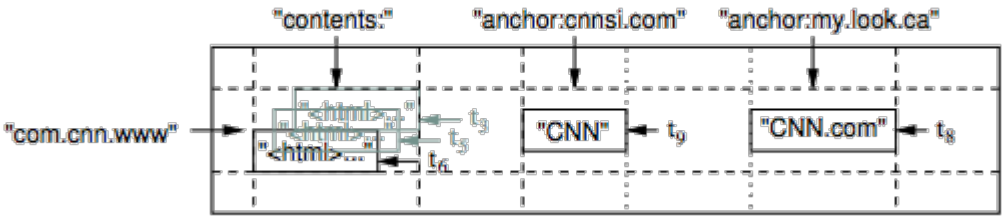




- Examples
  - Lotus Notes, CouchDB, MongoDB

Wide Column Stores

- Similar to Key-Value stores BUT the key is multi-dimensional
- “It uses tables, rows, and columns, but unlike a relational database, the names and format of the columns can vary from row to row in the same table”
- Examples
  - BigTable, Cassandra, …



https://en.wikipedia.org/wiki/Wide_column_store

Speed Layer

Input

- New data
- Written incrementally
  - Random writes!
- **Small data set**
  - Data deleted once in Service Layer

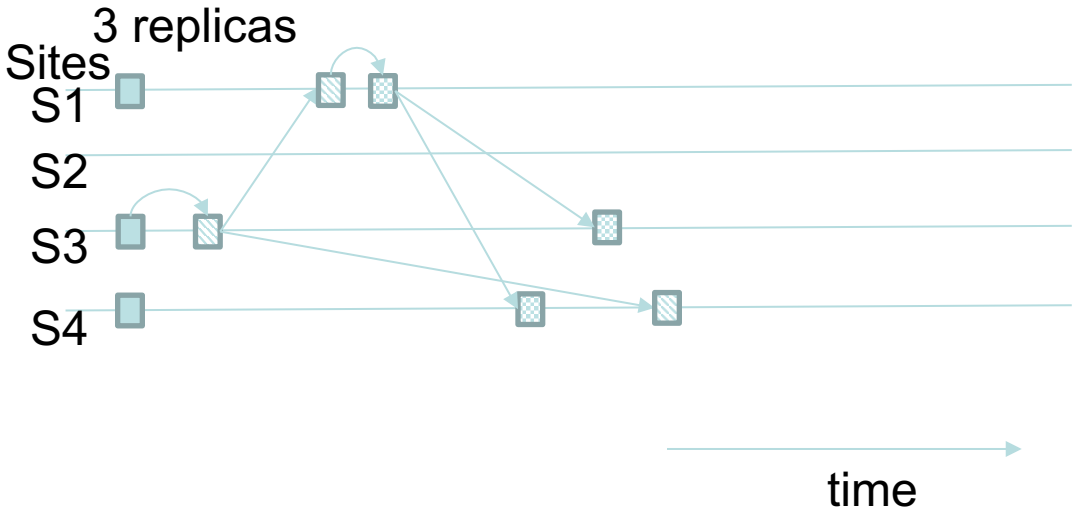
Goals

- Low latency
  - … and high throughput
  - Careful IO management
  - **Replicas for load-balancing**
- Linear scale-out
  - Distributed processing
- NoSQL solution!

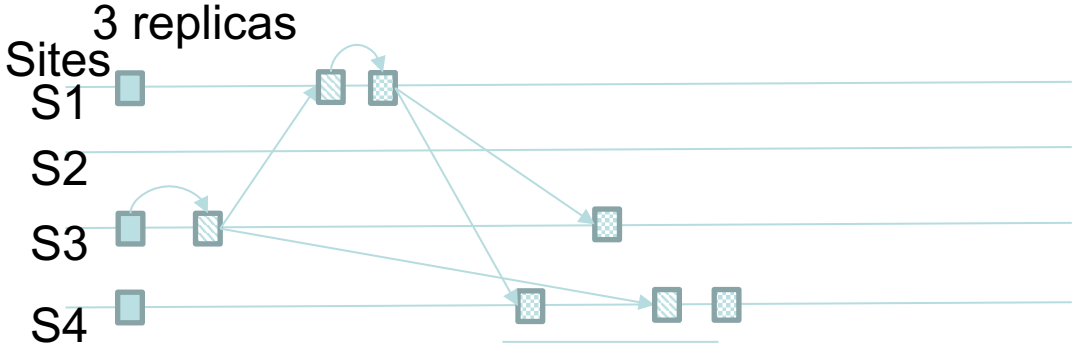
Replica Consistency

- **Sequential (or strong) consistency**: All updates are seen by all processes in the same order. As a result, the effects of an update are seen by all observers. There is no inconsistency.
  - **Atomic (or external or strict) consistency:** The effects of updates are "immediately" seen by all processes in the same sequence. There is no inconsistency, but reads of a replica might be delayed.
- **Weak consistency**: Observers might see inconsistencies among replicas
  - **Eventual consistency**: A form of weak consistency, where at some point, in case there is no failure, all replicas will reflect the last update.

Replica Consistency – Weak Consistency



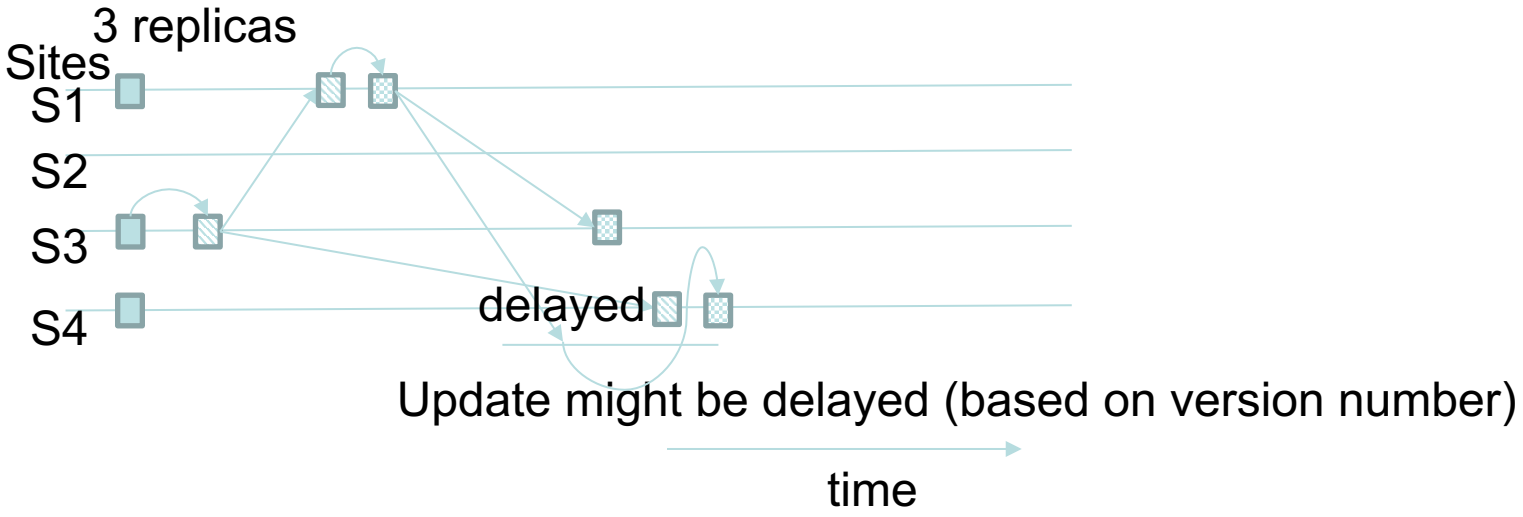
Replica Consistency – Eventual Consistency



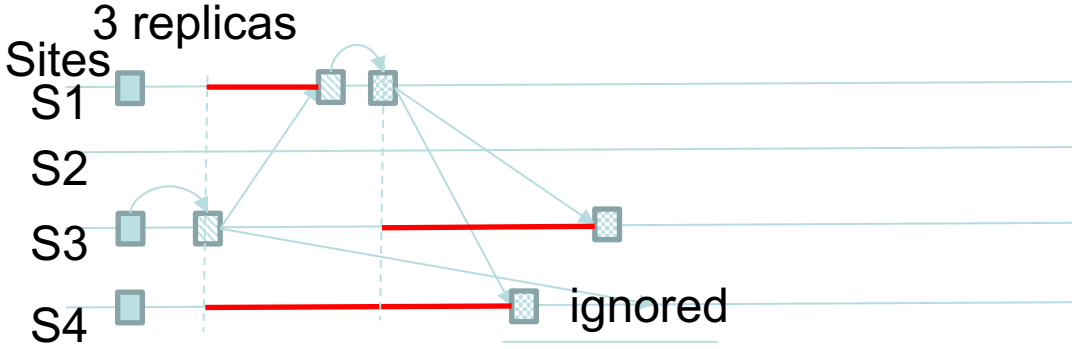
Conflict resolution: latest win (based on version number)

time

Replica Consistency – Strong Consistency



Replica Consistency – Strict Consistency



Reads not allowed on outdated version of a replica
Only latest update accepted

time

How to ensure synchronized decision on availability of a new value?

Paxos Algorithm

- Distributed consensus algorithm
  - Peer-to-peer architecture
  - Messages can be lost
  - Machines can fail

- A Paxos node
  - Proposer: proposes a value it wants agreement upon (e.g., new value)
  - Acceptor: chooses a value – it might receive several proposals and sends its decision to learners
  - Learner: determine whether a value has been accepted.

- In Paxos, a value is accepted if:
  a **majority** of acceptors **choose** the same value

  https://angus.nyc/2012/paxos-by-example/

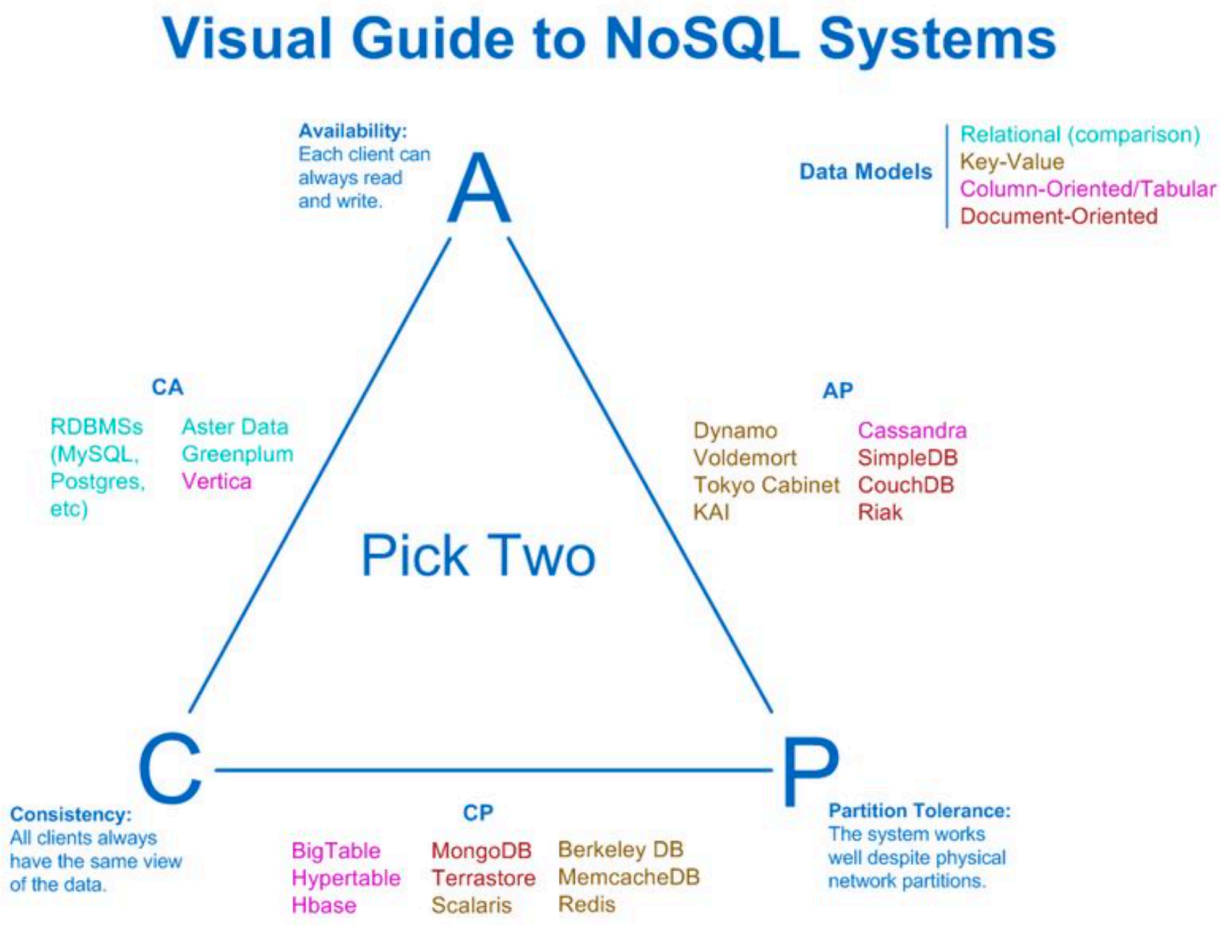
CAP Theorem

- C = Consistency
  - Readers read most recent update (PAXOS)
- A = Availability
  - An answer is always returned
- P = Partitions
  - The network becomes disconnected

Incorrect (but Typical) Formulation

- You can only get **two** of **Consistency** **Availability** and **Partition Tolerance**

Resulting Classification



Correct (but Useless) Formulation

- In a **partitioned** network, choose between **Consistency** and **Availability**

- Proof: Simple thought experiment

Seth Gilbert and Nancy Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, ACM SIGACT News, Volume 33 Issue 2 (2002)

PACELC (useful!) Formulation

- In a **partitioned** network, choose between **Availability** and **Consistency**
- **Else** (regular operation), choose between **Latency** and **Consistency**

Daniel Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”, IEEE Computer 45(2): 37-42 (2012)

Tunable Consistency

- Not a binary decision
  - N replicas, R reads, W writes
  - R = W = 1 gives eventual consistency
  - R + W > N (+Paxos) gives strong consistency
    - Reduced likelihood of non-availability

Outline

- Service layer (requirements, design)
- Speed layer (requirements, design)
  - NoSQL systems
  - CAP theorem and PACELC formulation
- Practical concerns
  - Maintaining Speed Layer
  - View Generality vs. Performance

Maintaining Speed Layer

- When can we remove data?
  - When it is reflected in the Service Layer
- How can we remove data **instantaneously**?
  - Duplicate storage and (stream) processing

View Generality vs. Performance

- Specific views
  - Good performance for each query
  - Answer limited queries
- General views
  - Processing required to answer query
  - Flexible query answering
- Need to strike a good balance!

Take Away Points

- Service Layer
  - Input: Sequential writes
  - Output: Sequential reads
  - Main goal: Low latency
- Speed Layer
  - Main goal: Low latency
  - Input: Random writes
  - CAP (PACELC) impacts design choices