NoSQL Data Stores and Eventual Consistency

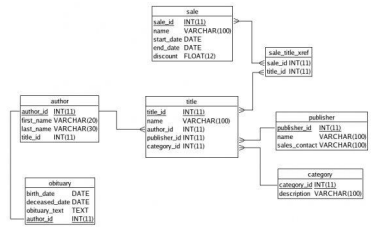
Dominic Duggan
Stevens Institute of Technology

Based on materials by Marin Dimitrov, Jonathan Ellis, Eric Evans, Eben Hewitt, Avinash Lakshman, Prashant Malik, Nati Shalom, Ran Tavory, Werner Vogels

1

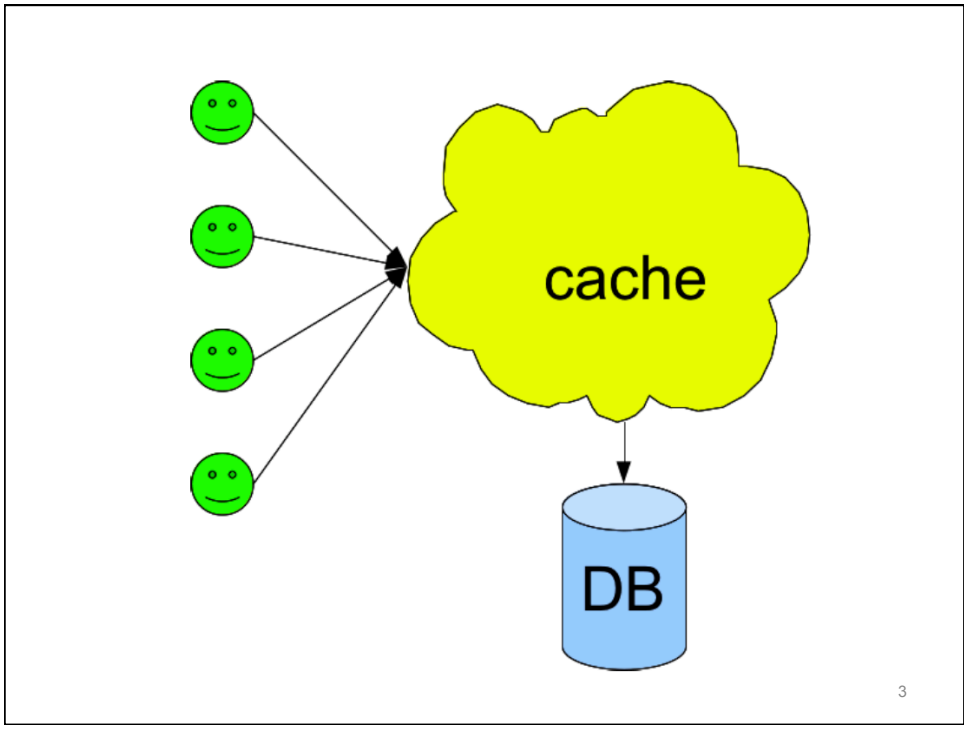
SQL

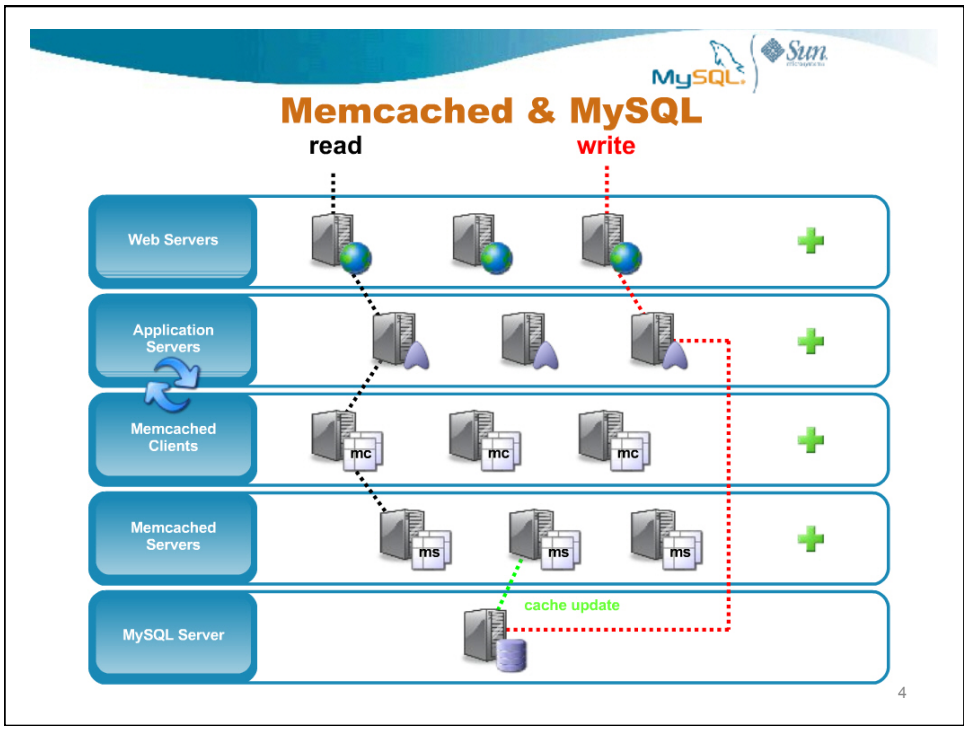
- Usually centralized
  - Transactional, consistent
  - Hard to scale
- Static, normalized schema
  - Don't duplicate, use FKs
- Ad-hoc query support
  - Model first, query later
- Standard
  - Rich ecosystem

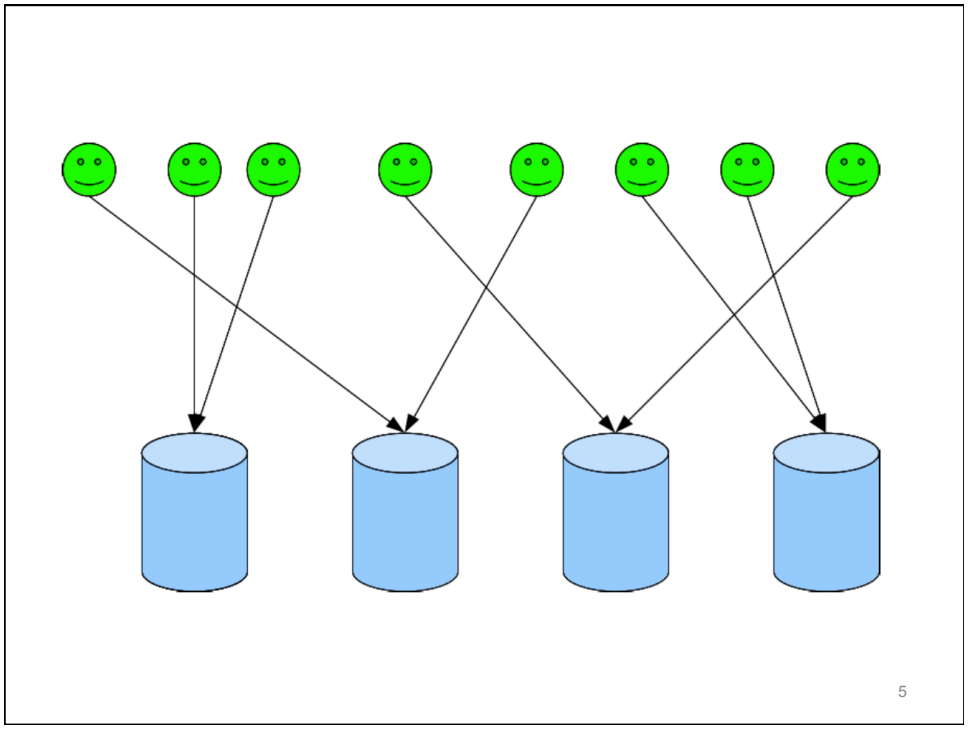



```
select users.user_id, users.email, count(*), max(classified_ads.posted)
from users, classified_ads
where users.user_id = classified_ads.user_id
group by users.user_id, users.email
order by upper(users.email);
```

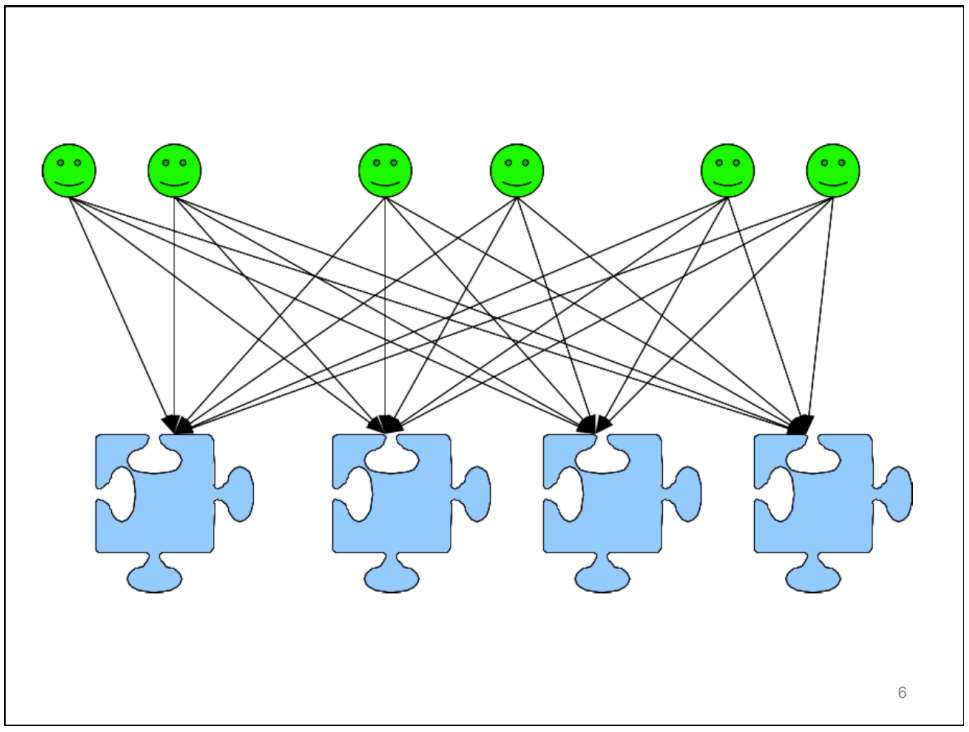
2


cache
DB
3

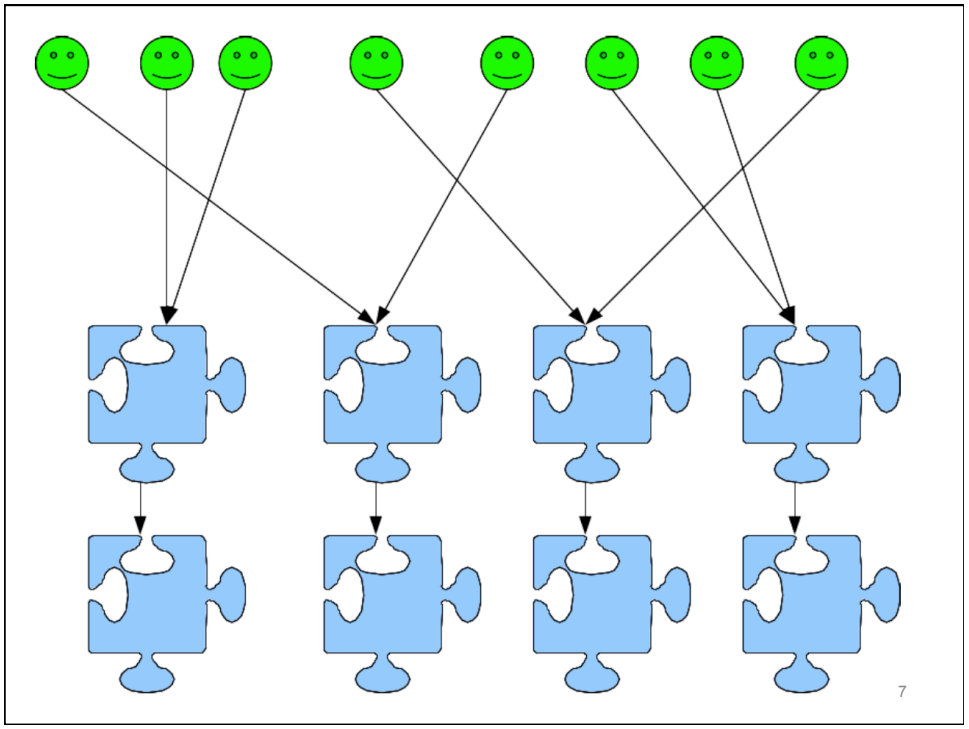

MySQL
Sun
Memcached & MySQL
read
write
Web Servers
Application Servers
Memcached Clients
Memcached Servers
MySQL Server
mc
ms
cache update
4

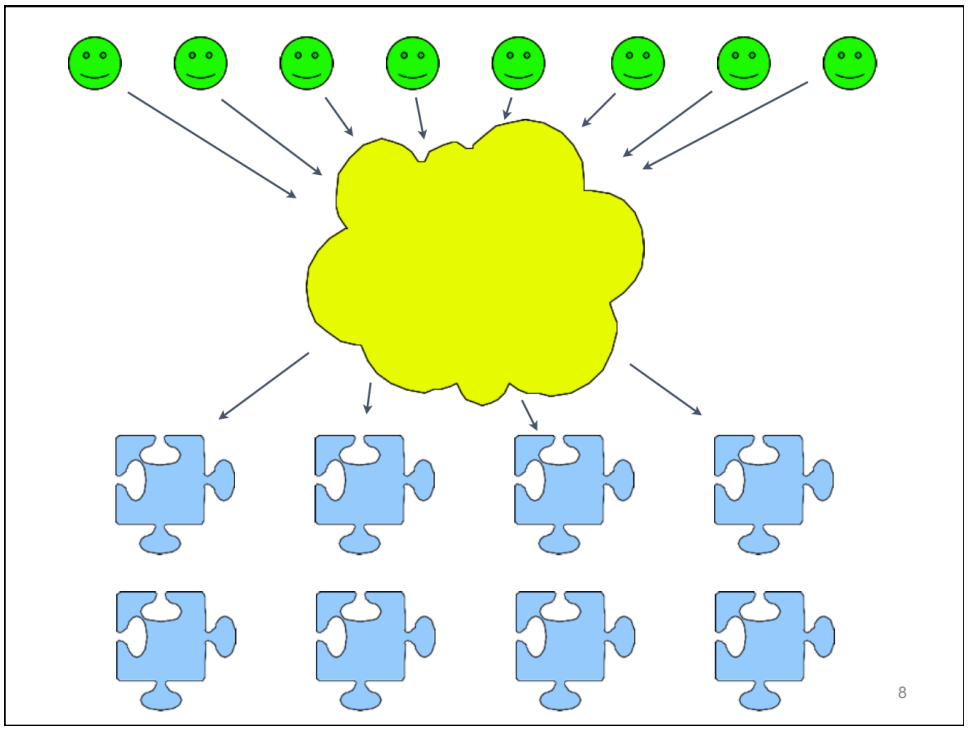


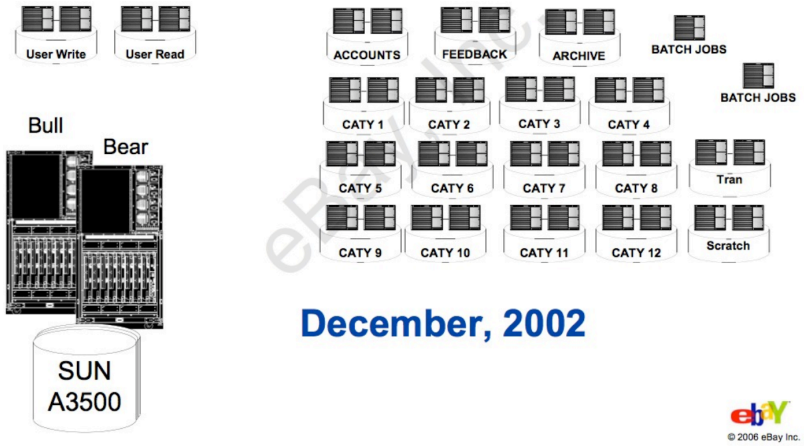
5



6

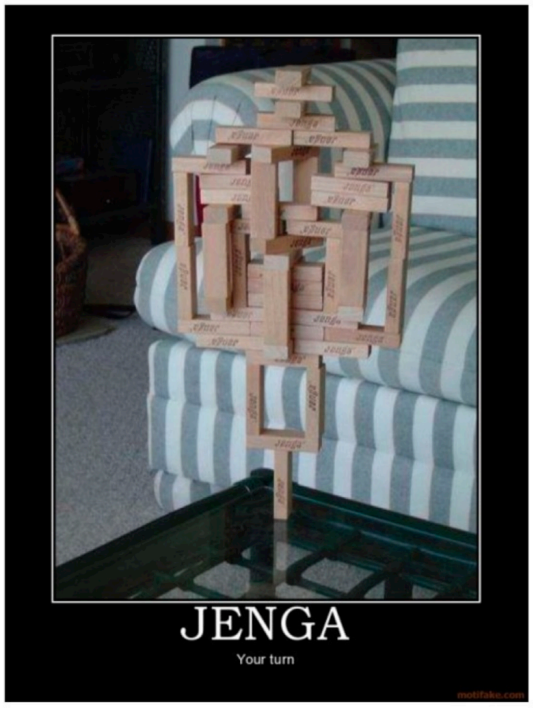

7


8



December, 2002

9



10

BASE: Basically Available, Soft-state, Eventually consistent

- "BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is quite manageable and leads be obtained with ACID."
  - "BASE: An Acid Alternative," Dan Pritchett, eBay

11

12

Why NoSQL?

- ACID doesn't scale well
- Web app needs
  - Low and predictable response time (latency)
  - Scalability and elasticity (at low cost)
  - High availability
  - Flexible schemas / semi-structured data
  - Geographic distribution (multiple data centers)
- Not so much
  - Transactions / strong consistency / integrity
  - Complex queries

13

NoSQL Use Cases

- Massive data volumes
  - Geographically distributed due to size
  - Google, Amazon, Yahoo, Facebook: 10-100K servers
- Extreme query workload
  - Joining columns in RDBMS does not scale
- Schema evolution
  - Migration at large scale
  - Key is gradual change

14

NoSQL pros & cons

- Advantages
  - Scalability
  - Availability
  - Low cost
  - Predictable elasticity
  - Schema flexibility, sparse and semi-structured data
- Disadvantages
  - Limited query capabilities
  - **Eventual consistency**
  - Standardization
  - Access control

15

CAP Theorem

- Requirements of distributed system
  - Consistency
    - All clients see same data
    - Strong (ACID) vs eventual (BASE)
  - Availability
    - Node failure
    - Upgrades
  - Partition tolerance
    - Reads **and** writes
- CAP Theorem: Pick 2 out of 3!

Conjectured by Brewer 2000.
Proved by Gilbert and Lynch 2002.

16

CAP Theorem

- Requirements of distributed system
  - Consistency
    - All clients see same data
    - Strong (ACID) vs eventual (BASE)
  - Availability
    - Node failure
    - Upgrades
  - Partition tolerance
    - Reads **and** writes
- ~~CAP Theorem: Pick 2 out of 3!~~

17

CAP Theorem

- CA:
  - Single site clusters e.g. 2PC
  - Partition blocks the system
- CP:
  - Some data inaccessible e.g. sharded database
- AP:
  - Some data may be inaccurate
  - E.g. DNS, caches, master/slave replication
  - Conflict resolution

18

CAP Positioning

19

CAE Requirements

- Requirements of distributed system
  - Cost-efficiency
    - Over-provisioning?
  - High Availability
    - Make the client wait?
  - Elasticity
    - Book resources in advance?
- Everyone wants HA, so provide E in a CE way

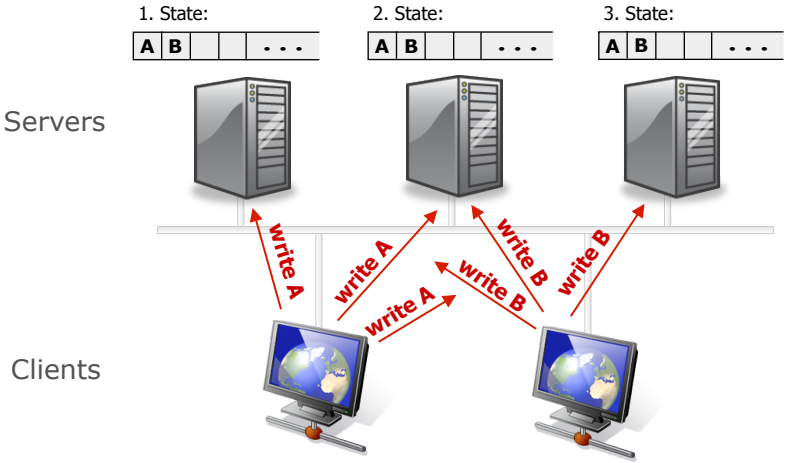
20

RELAXED CONSISTENCY

21

Data Consistency

- Strong consistency



22

Data Consistency

- Strong consistency
  - One-copy serializability
  - For replicated transactions in fixed networks

23

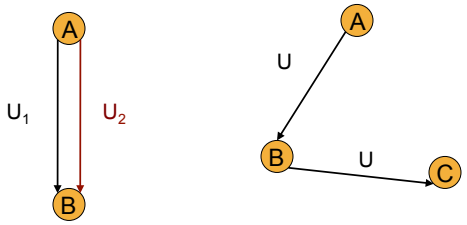
Data Consistency

- Best effort consistency
  - Ex: Grapevine email transport
  - Replicas may unintentionally fail to converge:
  - Ex: Update operations are not deterministic
  - Ex: Updates performed in different orders on replicas

24

Eventual Consistency

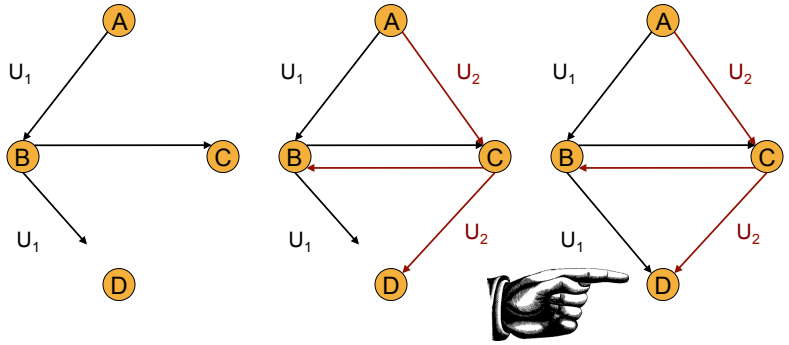
- Eventual consistency
  - Two properties:
    - Total propagation
    - Consistent ordering
  - Ordering of updates important



25

Eventual Consistency

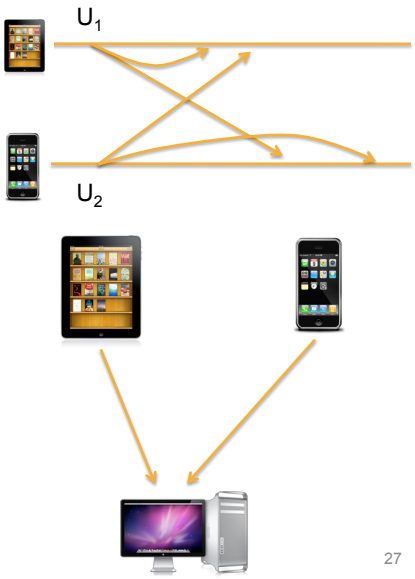
- Eventual consistency
  - Ordering of updates important



26

Out-of-order updates?

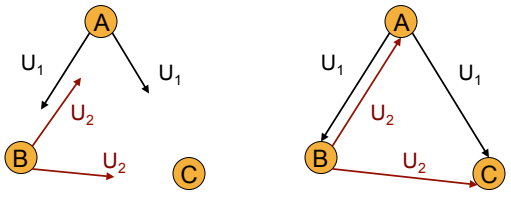
- Delay delivery
  - Ordered multicast
  - Delay even for local replica!
  - Not practical for disconnected
- Tentative update
  - Resolve with other updates later
    - Undo, perform missing updates, redo
  - Information about non-conflicting updates
    - API to specify



27

Causal Consistency

- Causal consistency



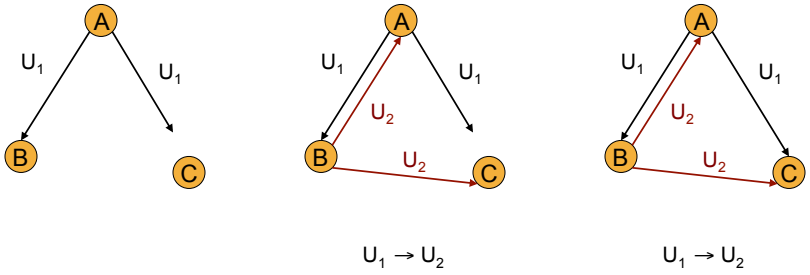
$U_1 \parallel U_2$ $U_1 \parallel U_2$

Does order of delivery matter at C?
Strong consistency: ✓
Causal consistency: ✗

28

Causal Consistency

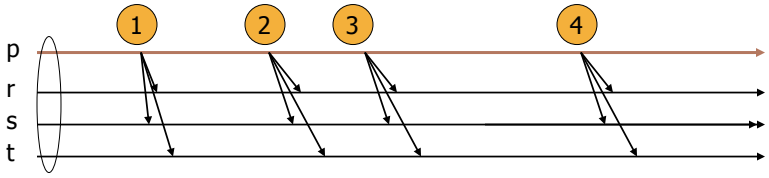
- Causal consistency



29

Single updater, FIFO ordering

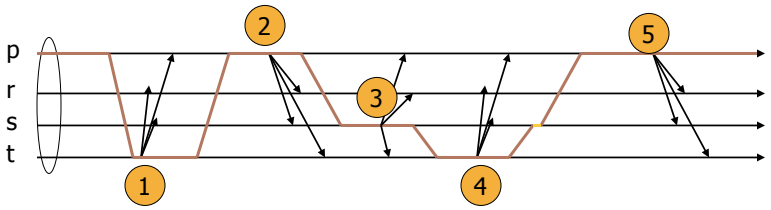
- If p is the only update source, the need is a bit like the TCP “fifo” ordering



30

Causally ordered updates

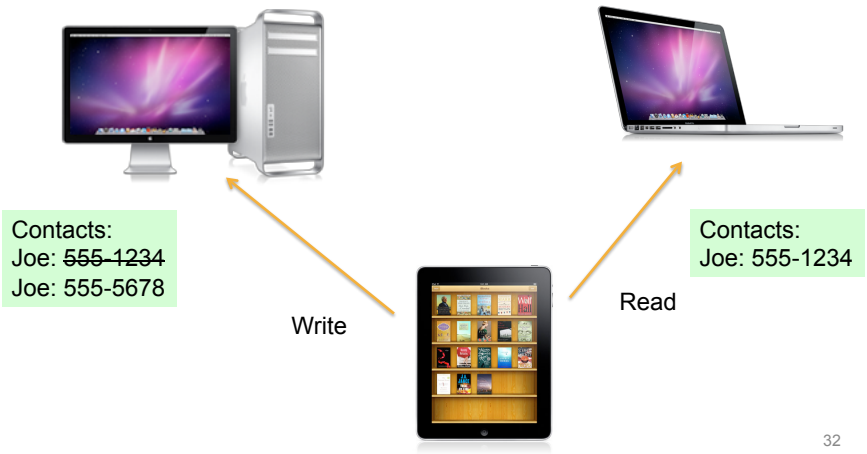
- Events occur on a "causal thread" but multicasts have different senders



31

Session Consistency

- Session consistency



32

Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads
  - Monotonic writes

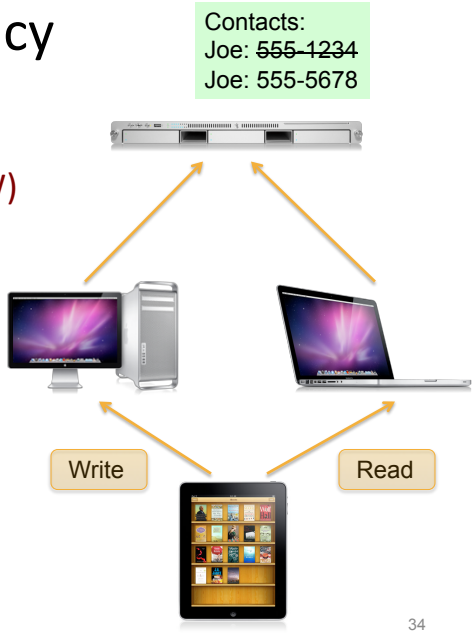
Pick one! (per session)

  - Assume global eventual consistency property

33

Session Consistency

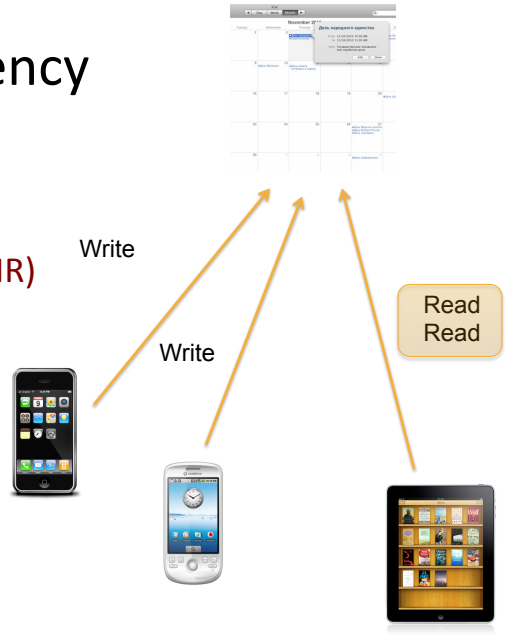
- Session consistency
  - Read your writes (RYW)
  - Monotonic reads
  - Writes follow reads
  - Monotonic writes



34

Session Consistency

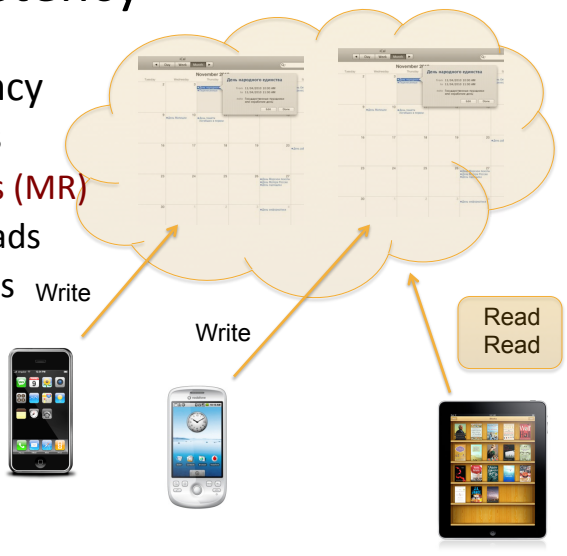
- Session consistency
  - Read your writes
  - Monotonic reads (MR)
  - Writes follow reads
  - Monotonic writes



35

Session Consistency

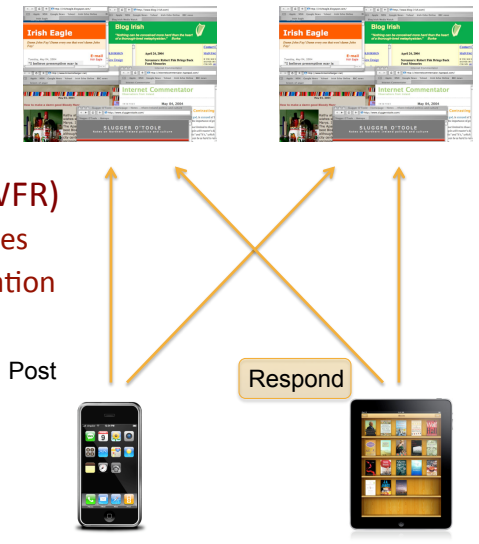
- Session consistency
  - Read your writes
  - Monotonic reads (MR)
  - Writes follow reads
  - Monotonic writes



36

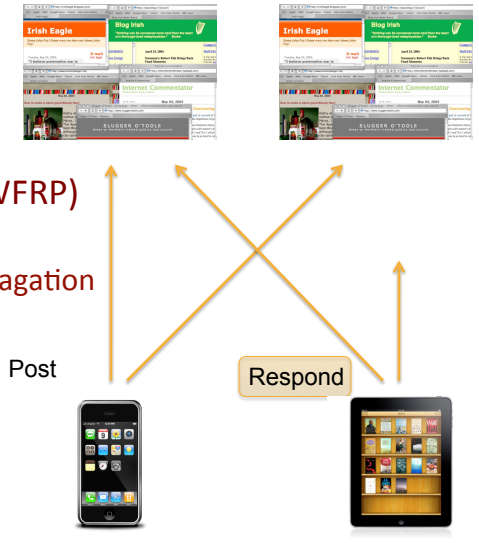
Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads (WFR)
    - Global ordering of writes
    - Order of write propagation
  - Monotonic writes



Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads (WFRP)
    - Ordering of write propagation
  - Monotonic writes

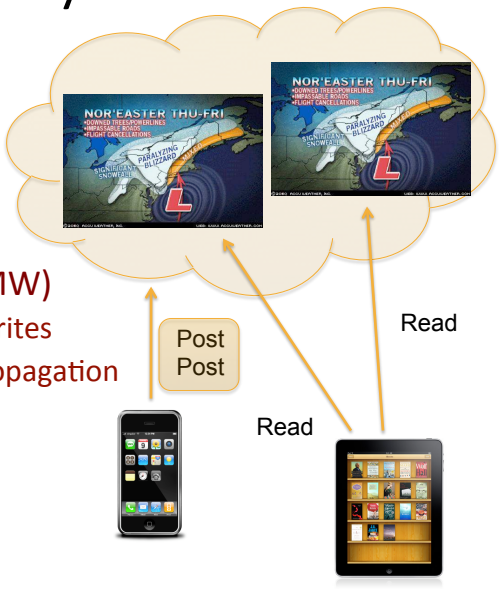


Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads (WFRO)
    - Global ordering of writes
  - Monotonic writes

Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads
  - Monotonic writes (MW)
    - Global ordering of writes
    - Ordering of write propagation



Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads
  - Monotonic writes (MW)
    - MW = RYW + WFR?

41

Session Consistency

- Session consistency
  - Read your writes
  - Monotonic reads
  - Writes follow reads
  - Monotonic writes (MW)
    - MW ≠ RYW + WFR

42

Data Consistency

- Bounded consistency
  - Ex: No data more than one hour old
  - Requires connectivity
- Hybrid consistency for Mobile
  - Strong consistency: read over Wifi or 3/4G
  - Weak consistency: read local copy
  - Consistency vs latency

43

NOSQL

44

NoSQL taxonomy

- Key-Value stores (DHT)
- Column stores
- Document stores
- Graph databases
  - Can't satisfy HA and E very well

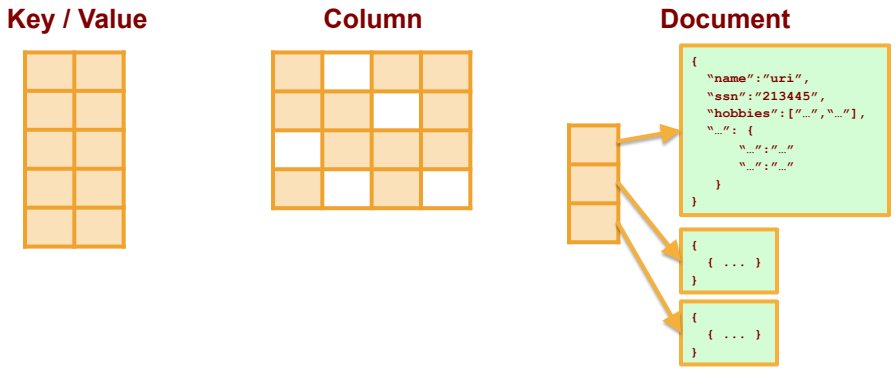
45

NoSQL taxonomy

- Key-Value stores (DHT) Amazon Dynamo
- Column stores Google Bigtable, Cassandra
- Document stores CouchDB, MongoDB, SimpleDB
- Graph databases Neo4j
  - Can't satisfy HA and E very well

46

A Universe of Data Models



47

Key/Value

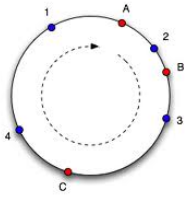
- Have the key? Get the value
  - Map/Reduce (sometimes)
  - Good for
    - cache aside (e.g. Hibernate 2nd level cache)
    - Simple, id based interactions (e.g. user profiles)
- In most cases, values are opaque

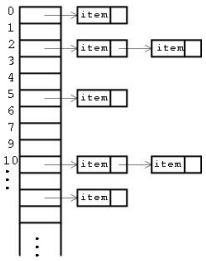
K1	V1
K2	V2
K3	V3
K4	V1

48

Key/Value

- Scaling out is relatively easy
  - just hash the keys
- Some will do that automatically for you
- Fixed vs. consistent hashing

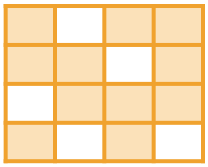




49

Column Based

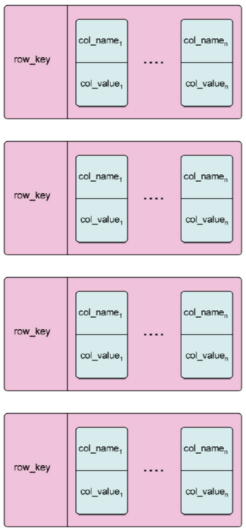
- Mostly derived from Google's BigTable papers
- One giant table of rows and columns
  - Column == pair (name and a value, sometimes timestamp)
  - Each row can have a different number of columns
  - Table is sparse:
    `(#rows) × (#columns) ≥ (#values)`



50

Column Based

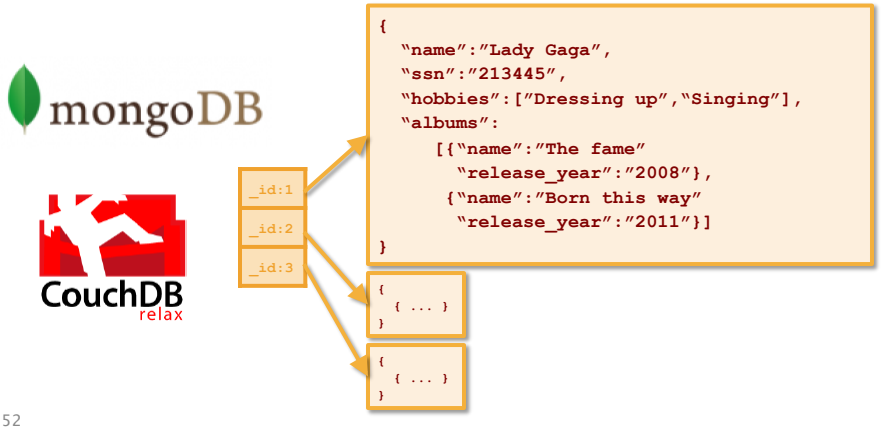
- Query on row key
  - Or column value (aka secondary index)
- Good for a constantly changing, (albeit flat) domain model



51

Document

- Think JSON (or BSON, or XML)



52

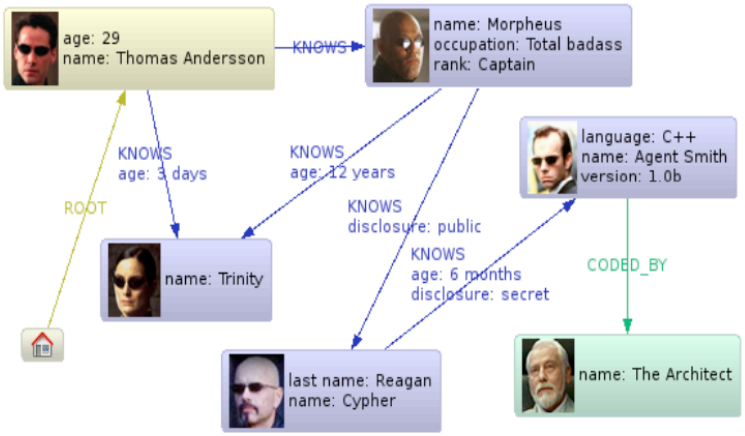
Document

- Model is not flat, data store is aware of it
  - Arrays, nested documents
- Better support for ad hoc queries
  - MongoDB excels at this
- Very intuitive model
- Flexible schema

```
> db.people.find({age: {$gt: 27}})
{ "_id" : ObjectId("4bed80b20b4acd070c593bac"), "name" : "John", "age" : 28 }
{ "_id" : ObjectId("4bed80bb0b4acd070c593bad"), "name" : "Steve", "age" : 29 }
```

53

Graph



54

DYNAMO: KEY-BASED STORE

55

Motivation

- Build a distributed storage system:
  - Scale
  - Simple: key-value
  - *Highly available*
  - *Guarantee Service Level Agreements (SLA)*

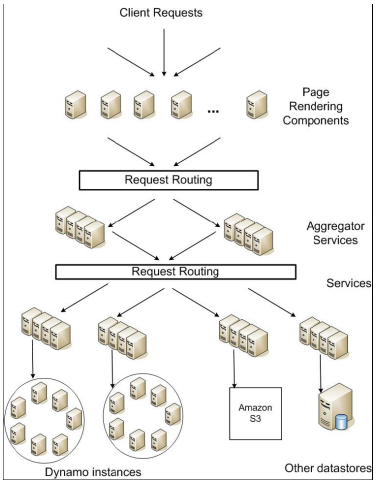
56

System Assumptions and Requirements

- Query Model:
  - key-based read and write
- ACID Properties:
- Efficiency:
  - latency at the 99.9th percentile
- Other Assumptions:
  - non-hostile (trusted!) environment

57

Service Level Agreements (SLA)



Service-oriented architecture of Amazon's platform

- Time bounds on delivery.
- Example: response within 300ms for 99.9% of requests, for a peak client load of 500 requests per second.

58

Design Consideration

- *Sacrifice strong consistency for availability*
- Conflict resolution during read instead of write, i.e. "always writeable"
- Other principles:
  - Incremental scalability.
  - Symmetry.
  - Decentralization.
  - Heterogeneity.

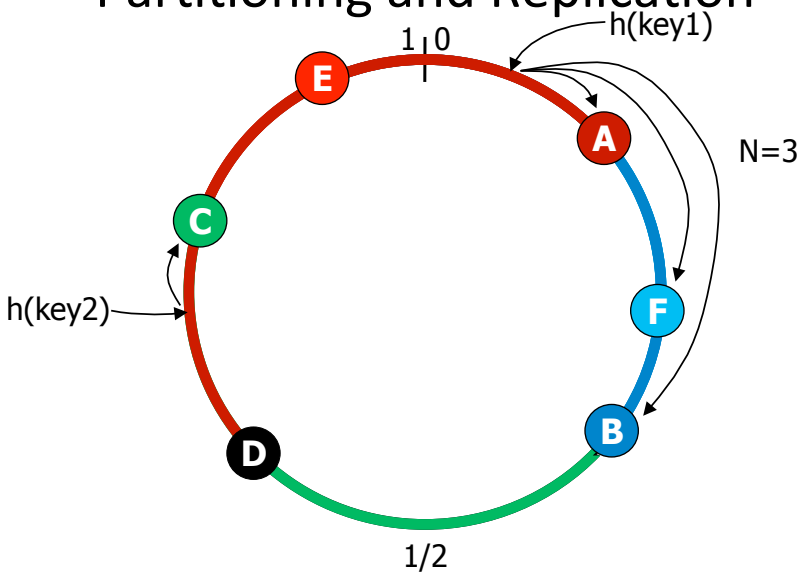
59

Dynamo Techniques

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.

60

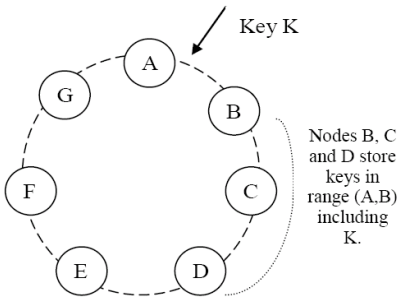
Partitioning and Replication



61

61

Partition Algorithm



- Consistent hashing
- "Virtual Nodes"
  - Nodes back up other nodes
  - Load balancing
- "Preference list"
  - e.g. B, C, D for key K

62

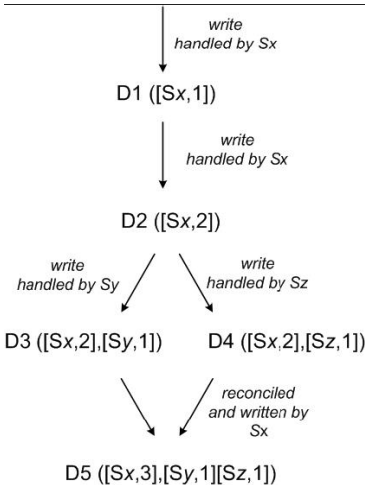
Data Versioning

- put(): return before update applied at all replicas
- get(): return many versions of the same object

- *Challenge:* object has distinct version sub-histories
- *Solution:* vector clocks for reconciliation

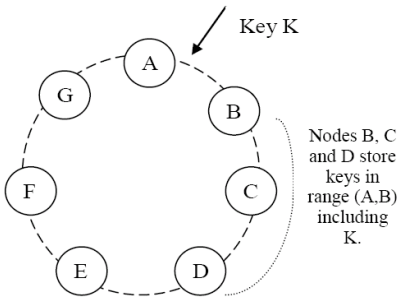
63

Vector clock example



64

Hinted handoff

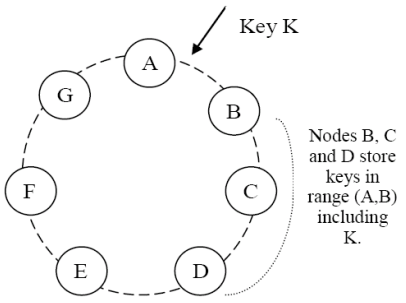


- Assume N = 3
- A down ⇒ send replica to D
- D is hinted that the replica belongs to A
- D will deliver to A when A is recovered
- "Always writeable"

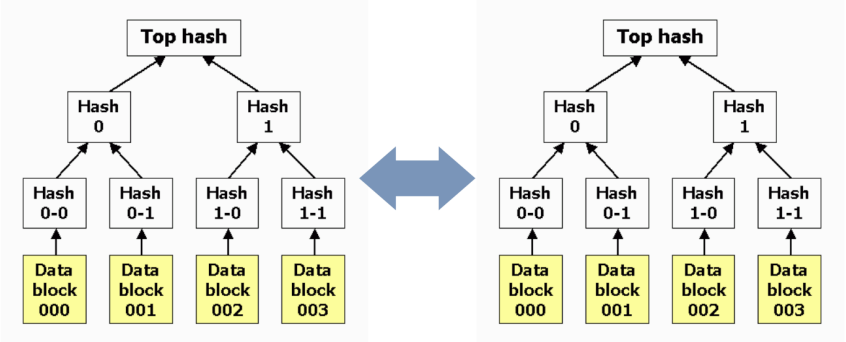
65

Sloppy Quorum

- R/W read and write quora
- R + W > N
  - Latency dictated by slowest of R (or W) replicas
  - …but which N?
  - Pref List (N=3) ⇒ consistent
  - Sloppy Quorum: **any** N (hinted handoff)
- R + W < N
  - better latency



Replica Synchronization



Merkle Hash Trees

67

Cluster Membership and Failure Detection

- Gossip protocol
- State disseminated in O(log(N)) rounds
- Every T seconds, each member increments its heartbeat counter and selects one other member to send its list to
- A member merges the list with its own list

68

Accrual Failure Detector

- PHI = suspicion level
- Failure detector outputs PHI for each process
- Applications:
  - set threshold
  - trigger suspicions
  - perform appropriate actions
- PHI threshold = 5 ⇒ average time taken to detect a failure is 10-15 seconds

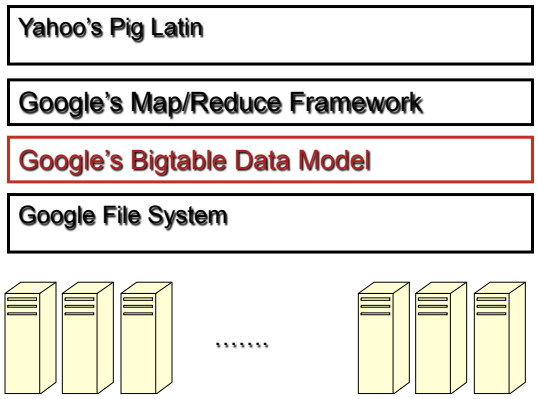
69

BIGTABLE: COLUMN-BASED STORE

70

Overall Architecture

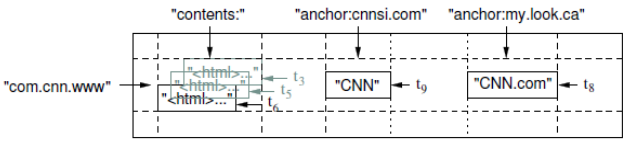
- Shared-nothing architecture consisting of thousands of nodes!



71

Bigtable

- A sparse, distributed persistent multi-dimensional sorted map.
- The map is indexed by a row key, column key, and a timestamp.
- Output value is array of bytes.
  - (row: byte[ ], column: byte[ ], time: int64) → byte[ ]



72

Rows

- Read/write of data under a single row is atomic
- Data ordered by row key
- Row range dynamically partitioned
- **Tablet**: partition (row range)
  - Unit of distribution and load-balancing
- Objective: make read operations single-sited!
  - E.g., In Webtable: com.google.maps instead of maps.google.com.

73

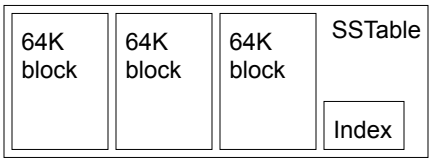
Building Blocks

- Google File System
  - High availability
- SSTable
  - A key/value database
- Chubby
  - Name space
  - Paxos!

74

SSTable

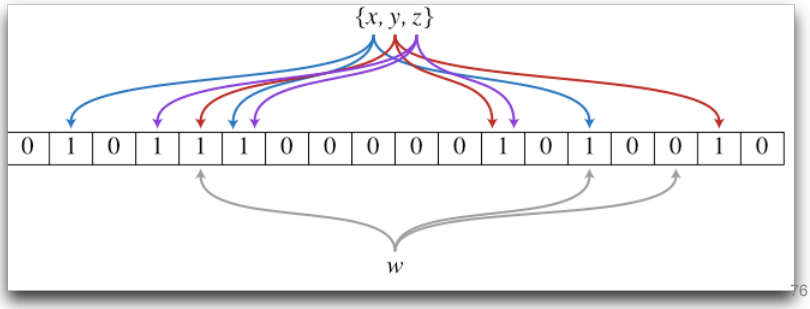
- *Immutable*, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is of block ranges, not values
  - Binary search (in memory) to find element location
  - Bloom filter to reduce number of unneeded binary searches.



75

Bloom Filters

- Space efficient probabilistic data structure
- Test whether an element is a member of a set
- Allow false positive, but not false negative
- k hash functions
- Union and intersection implemented as bitwise OR, AND



76

Tablets: Hybrid Range Partitioning [VLDB'90]

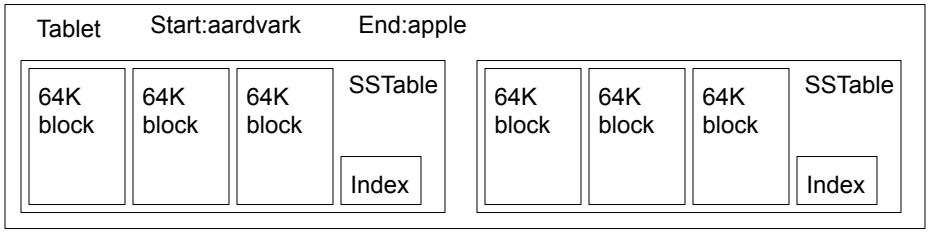
- To minimize the impact of load imbalance, construct more (HN) ranges than (N) nodes, e.g., 10 ranges for a 5 node system; H = 2.

- H is higher in practice e.g. 10
- A range is named a **tablet**. A tablet is represented as:
  - A set of SSTable files.
  - A set of redo points which are pointers into any commit logs that may contain data for the tablet.

77

Tablet

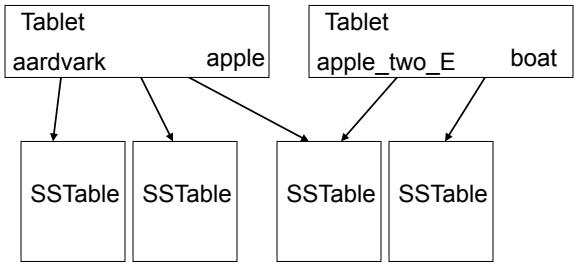
- Contains some range of rows of the table
- Built out of multiple SSTables



78

Table

- Multiple tablets make up the table
- SSTables can be shared
- Tablets do not overlap, SSTables can overlap



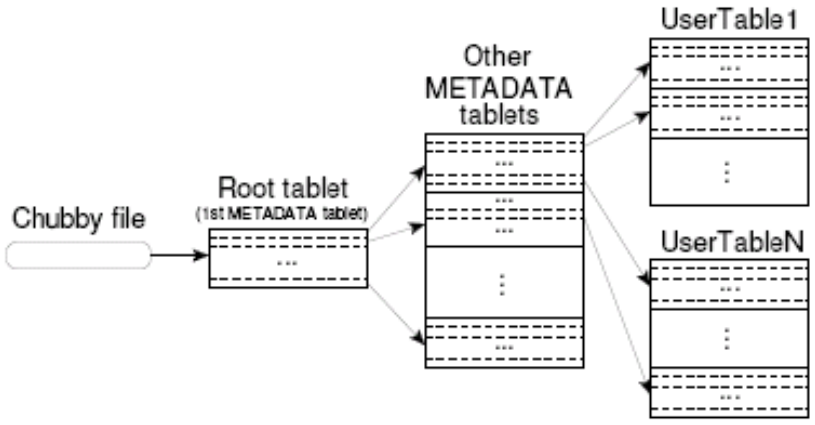
79

Servers

- Tablet servers manage tablets
  - Each tablet is 100-200 megs
  - Each tablet lives at only one server
  - Tablet server splits tablets that get too big
- Master server responsible for load balancing and fault tolerance
  - Chubby: monitor tablet servers, restart failed servers
  - GFS replicates data
  - Start tablet server on same machine that holds data

80

Finding a tablet



81

Bigtable and Chubby

- Bigtable uses Chubby to:
  - Ensure at most one active master
  - Store bootstrap location of root tablet
  - Discover tablet servers, finalize server deaths
  - Store Bigtable column family information
  - Store access control list
- Chubby unavailable ⇒ Bigtable unavailable

82

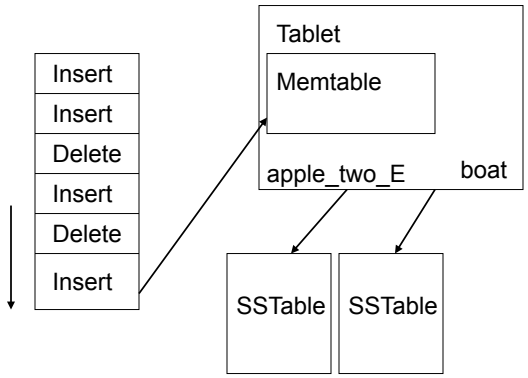
Placement of Tablets

- Master maintains:
  - Live tablet servers,
  - Assignment of tablets to tablet servers
- Chubby maintains tablet servers:
  - **Server directory**: Tablet server acquires Chubby lock on tablet
  - Lock lost (network partition) ⇒ cannot process requests
  - Master monitors server directory to discover tablet server

83

Editing a table

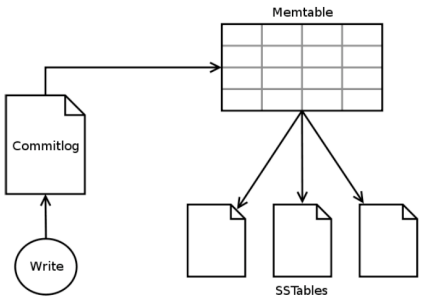
- Mutations are logged, then applied to an in-memory version (memtable)
- Log file stored in GFS



84

Memtable

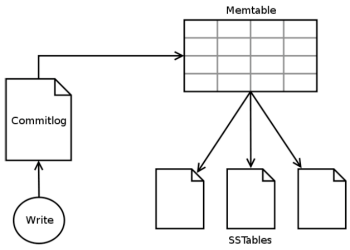
- In-memory recently written data
- Table full ⇒ sorted and flushed to disk (**SSTable**)



85

Client Write & Read Operations

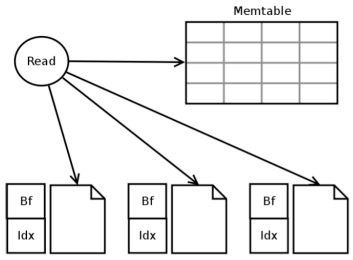
- Write operation arrives at a tablet server:
  - Client has sufficient privileges (Chubby)?
  - Log record written to the commit log file
  - Committed write contents are inserted into the memtable.



86

Client Write & Read Operations

- Read operation arrives at a tablet server:
  - Client has sufficient privileges (Chubby)?
  - Read performed on a merged view of (a) the SSTables for the tablet, and (b) the memtable.



87

Compactions

- Minor compaction – convert memtable into SSTable
  - Reduce memory usage
  - Reduce log traffic on restart
- Merging compaction
  - Reduce number of SSTables
  - Apply policy "keep only N versions"
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data

88

Write Properties

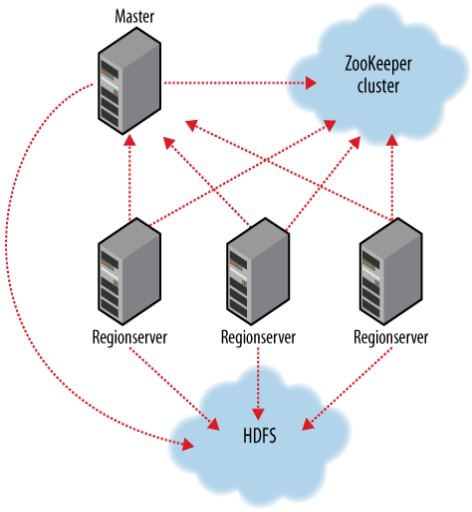
- No locks in the critical path
- Sequential disk access
- Behaves like a write back cache
- Append support without read ahead
- Atomicity guarantee for a key (row)

89

HBASE

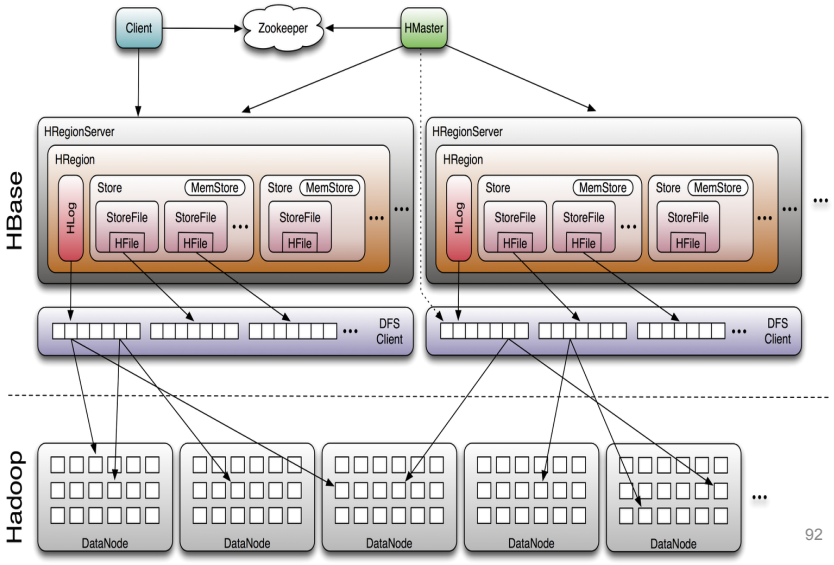
90

HBase: An OSS Bigtable Clone



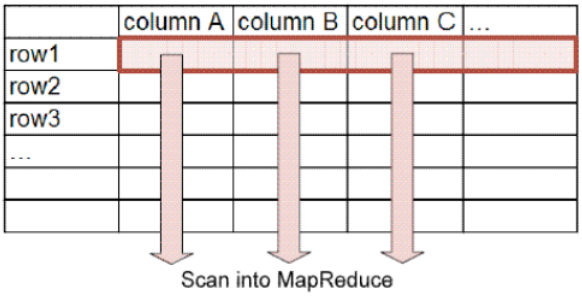
91

HBase Architecture



92

Hbase as MapReduce input



- Each row is an input record to MapReduce
- MapReduce jobs can sort/search/index/query data in bulk

93

CASSANDRA

94

Cassandra = Dynamo + Bigtable

95

Data Model

- Same as Bigtable
- Composite Columns
  - Secondary indexes
- Column order in a Column Family can be specified (name, time)

96

Dynamic Partitioning

- Consistent hashing
- Ring of nodes
- Nodes can be "moved" on the ring for load balancing

97

Operations

- Writes (= Bigtable…almost)
  - Client sends request to a random node, that node determines the actual node responsible for storing the data (cluster awareness)
  - Data replicated to N nodes (configurable)
  - Data first added to the commit log, then to the memtable, eventually flushed to SSTable

98

Operations

- Reads (= Bigtable…almost)
  - Send request to a random node, which forwards it to all the N node having the data
    - Single read: return first response
    - Quorum read: value that the majority of nodes agree on
  - First read from memtable, if info is incomplete then read the SSTables
  - Bloom filters for efficient SSTable lookups

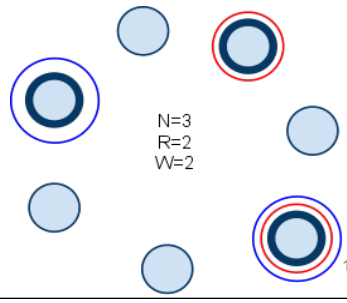
99

Consistency Choices

- W=1 ⇒ Block until first node written successfully
- W=N ⇒ Block until all nodes written successfully
- W=0 ⇒ Async writes

- R=1 ⇒ Block until the first node returns an answer
- R=N ⇒ Block until all nodes return an answer
- R=0 ⇒ Doesn't make sense

- QUORUM:
  - R = N/2+1
  - W = N/2+1
  - ⇒ Fully consistent



100

Consistency Choices

Write	
Level	**Description**
ZERO	Hail Mary
ANY	1 replica (HH)
ONE	1 replica
QUORUM	(N / 2) +1
ALL	All replicas

Read	
Level	**Description**
ZERO	N/A
ANY	N/A
ONE	1 replica
QUORUM	(N / 2) +1
ALL	All replicas

R + W > N

101

Conclusions

- NoSQL Data Stores
  - Scalable
  - Available
- Rethink assumptions
  - Eventual consistency
  - Semi-structured data
  - Design for queries (MR)

102