NoSQL

Learning Objectives

- Understand the basic of NoSQL
- Four major types of NoSQL Database

Outline

1. Motivation
2. CAP Theorem
3. Category
4. Typical NoSQL API

1 Motivation

Why Cloud Data Stores

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Explosion of storage needs in large web sites such as Google, Yahoo
  - Much of the data is not files
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Shift to dynamically-typed data with frequent schema changes

Parallel Databases and Data Stores

- Web-based applications have huge demands on data storage volume and transaction rate
- Scalability of application servers is easy, but what about the database?
- Approach 1: memcache or other caching mechanisms to reduce database access
  - Limited in scalability
- Approach 2: Use existing parallel databases
  - Expensive, and most parallel databases were designed for decision support not OLTP
- Approach 3: Build parallel stores with databases underneath

Scaling RDBMS - Partitioning

- “Sharding”
  - Divide data amongst many cheap databases (MySQL/PostgreSQL)
  - Manage parallel access in the application
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware

Parallel Key-Value Data Stores

- Distributed key-value data storage systems allow key-value pairs to be stored (and retrieved on key) in a massively parallel system
  - E.g. Google BigTable, Yahoo! Sherpa/PNUTS, Amazon Dynamo, ..
- Partitioning, high availability etc completely transparent to application
- Sharding systems and key-value stores don't support many relational features
  - No join operations (except within partition)
  - No referential integrity constraints across partitions
  - etc.

Data Management

- Database Management System (DBMS) provides our application:
  efficient, reliable, convenient, and safe
  multi-user storage of and access to
  massive
  amounts of persistent data.
- Not every data management/analysis problem is best solved using a traditional DBMS

Why NoSQL

- Big data
- Scalability
- Data format
- Manageability

Big Data

- Collect
- Store
- Organize
- Analyze
- Share

Data growth outruns the ability to manage it so we need **scalable** solutions.

Scalability

- Scale up, Vertical scalability.
  - Increasing server capacity.
  - Adding more CPU, RAM.
  - Managing is hard.
  - Possible down times

Scalability

- Scale out, Horizontal scalability.
  - Adding servers to existing system with little effort, aka Elastically scalable
    - Bugs, hardware errors, things fail all the time.
    - It should become cheaper. Cost efficiency.
  - Shared nothing
  - Use of commodity/cheap hardware
  - Heterogeneous systems
  - Controlled Concurrency (avoid locks)
  - Symmetry, you don’t have to know what is happening. All nodes should be symmetric.

NoSQL: The Name

- "SQL" = Traditional relational DBMS
- Recognition over past decade or so:
  Not every data management/analysis problem is best solved using a traditional relational DBMS
- "NoSQL" = "No SQL" =
  Not using traditional relational DBMS
- "No SQL" ≠ Don't use SQL language
- "NoSQL" =? "Not Only SQL" => NOSQL

NoSQL databases

- The name stands for **N**ot **O**nly **SQL**
- Common features:
  - non-relational
  - usually do not require a fixed table schema
  - horizontal scalable
  - mostly open source
- More characteristics
  - relax one or more of the ACID properties (see CAP theorem)
  - replication support
  - easy API (if SQL, then only its very restricted variant)
- Do not fully support relational features
  - no join operations (except within partitions),
  - no referential integrity constraints across partitions.

Anecdote

- Johan Oskarsson wanted to organize an event to discuss open-source distributed databases in 2009
- Johan wanted a name for the meeting – something that would make a good Twitter hashtag: short, memorable, and without too many Google hits so that a search on the name would quickly find the meetup.
- He asked for suggestions on the #cassandra IRC channel and got a few, selecting the suggestion of “NoSQL” from Eric Evans, a developer at Rackspace.

Example #1: Web log analysis

Each record: UserID, URL, timestamp, additional-info

Task: Load into database system

Example #1: Web log analysis

Each record: UserID, URL, timestamp, additional-info

Task: Find all records for…

- Given UserID
- Given URL
- Given timestamp
- Certain construct appearing in additional-info

Example #1: Web log analysis

Each record: UserID, URL, timestamp, additional-info

Separate records: UserID, name, age, gender, …

Task: Find average age of user accessing given URL

Example #2: Social-network graph

Each record: $UserID_1$, $UserID_2$

Separate records: UserID, name, age, gender etc

Task: Find all friends of friends of friends of … friends of given user

Example #3: Blog Pages

Large collection of documents

Combination of structured and unstructured data

Task: Retrieve texts and images

More Programming and Less Database Design

Alternative to traditional relational DBMS

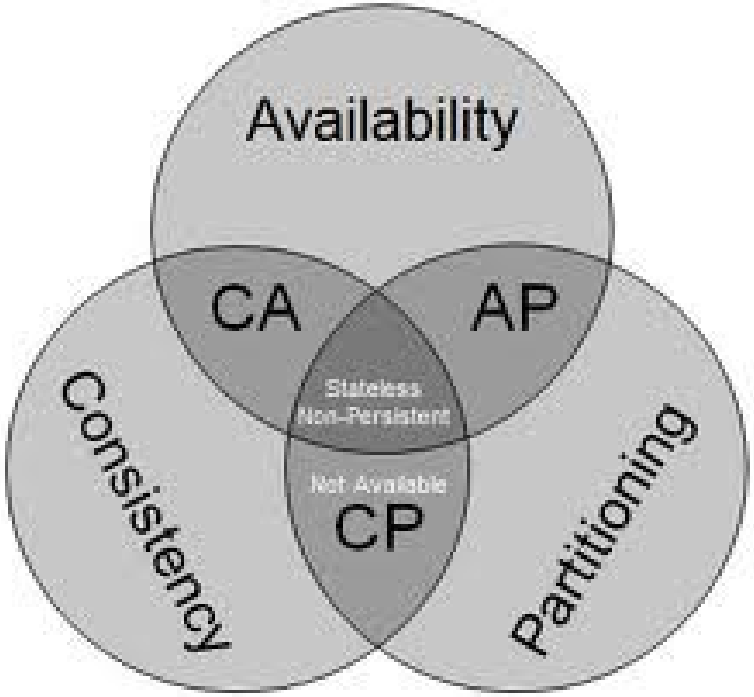
+ Flexible schema
+ Quicker/cheaper to set up
+ Massive scalability
+ Relaxed consistency → higher performance & availability

− No declarative query language → more programming
− Relaxed consistency → fewer guarantees

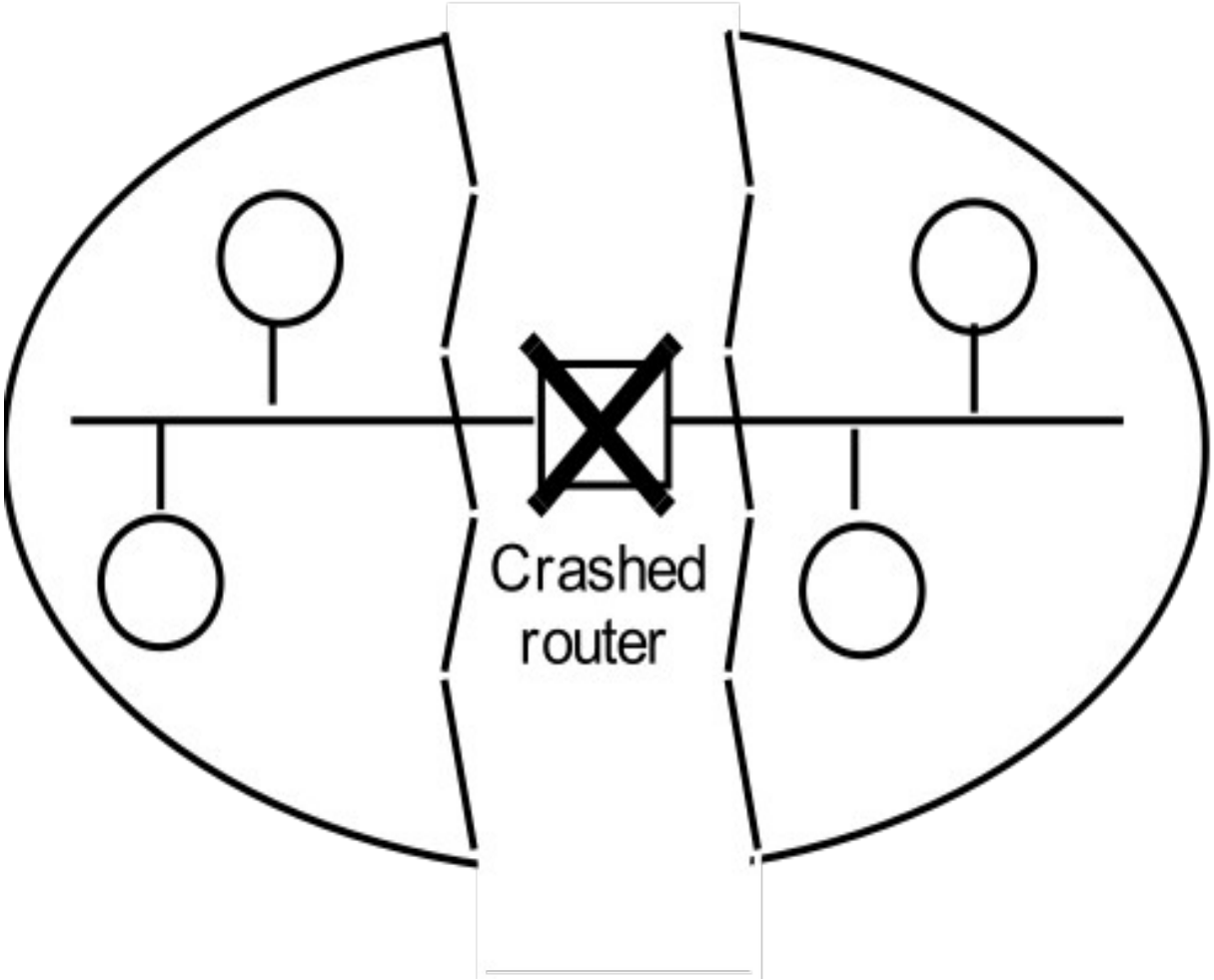
Challenge: Coordination

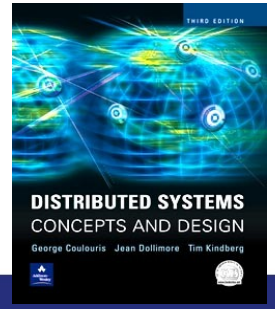
- The solution to availability and scalability is to decentralize and replicate functions and data…but how do we coordinate the nodes?
  - data consistency
  - update propagation
  - mutual exclusion
  - consistent global states
  - group membership
  - group communication
  - event ordering
  - distributed consensus
  - quorum consensus

2 CAP Theorem



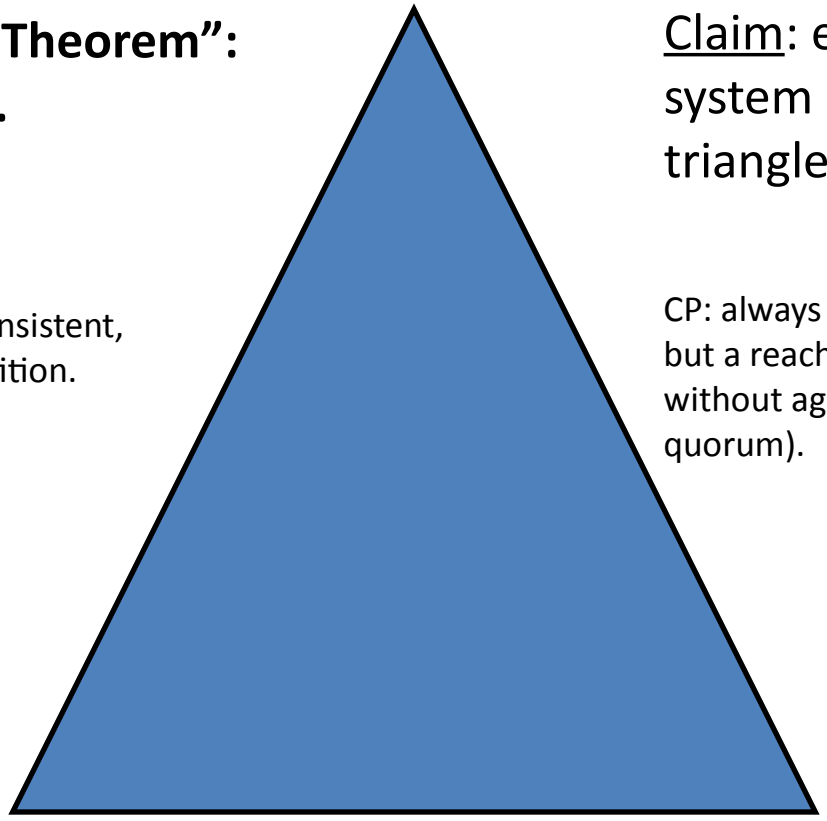
A network partition





consistency

Fox&Brewer "CAP Theorem":
C-A-P: choose two.



Claim: every distributed system is on one side of the triangle.

CA: available, and consistent, unless there is a partition.

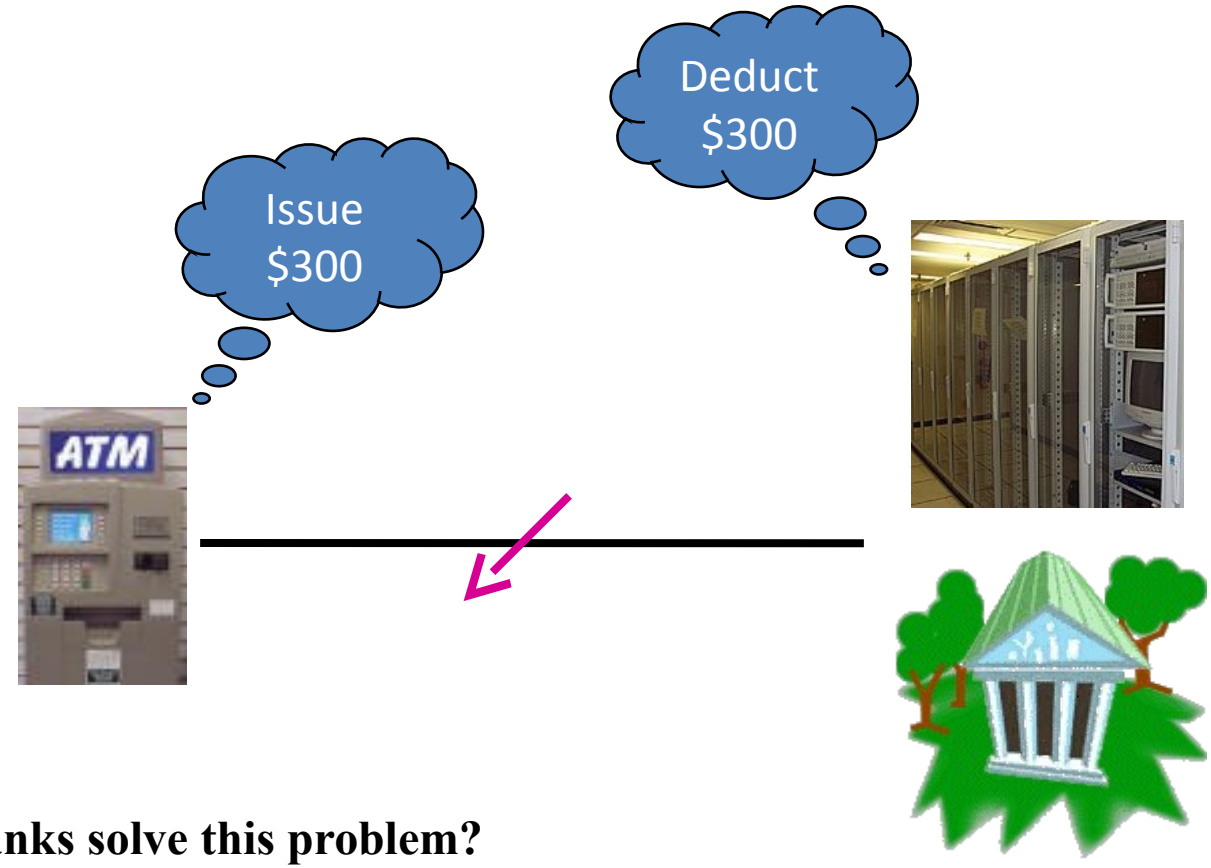
CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).

AP: a reachable replica provides service even in a partition, but may be inconsistent if there is a failure.

Availability

Partition-resilience

Two Generals in practice



How do banks solve this problem?

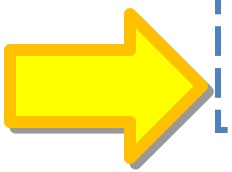
Careful ordering is limited

- Transfer $100 from Melissa's account to mine
  1. Deduct $100 from Melissa's accoun
  2. Add $100 to my account
- Crash between 1 and 2, we lose $
- Could reverse the ordering
  1. Add $100 to my account
  2. Deduct $100 from Melissa's accoun
- Crash between 1 and 2, we gain $
- What does this remind you of?

Transactions

- Fundamental to databases
  - (except MySQL, until recently)
- Several important properties
  - "ACID" (atomicity, consistent, isolated, durable)
  - We only care about atomicity (all or nothing)

Called "committing" the transaction



```
BEGIN
   disk write 1
   …
   disk write n
END
```

Transactions: logging

1. Begin transaction
2. Append info about modifications to a log
3. Append “commit” to log to end x-action
4. Write new data to normal database

- Single-sector write commits x-action (3)

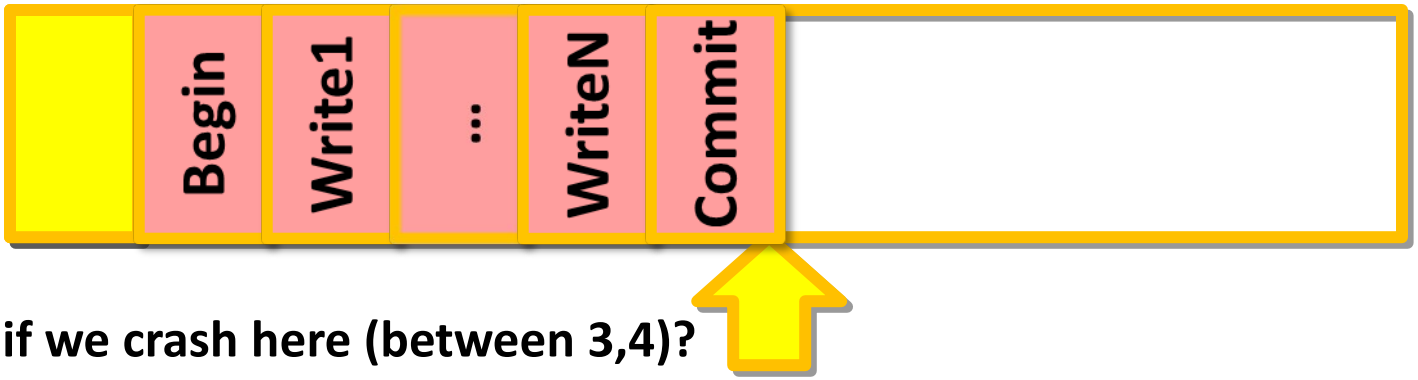
Invariant: append new data to log before applying to DB

Called “write-ahead logging”

Transactions: logging

1. Begin transaction
2. Append info about modifications to a log
3. Append “commit” to log to end x-action
4. Write new data to normal database

- Single-sector write commits x-action (3)

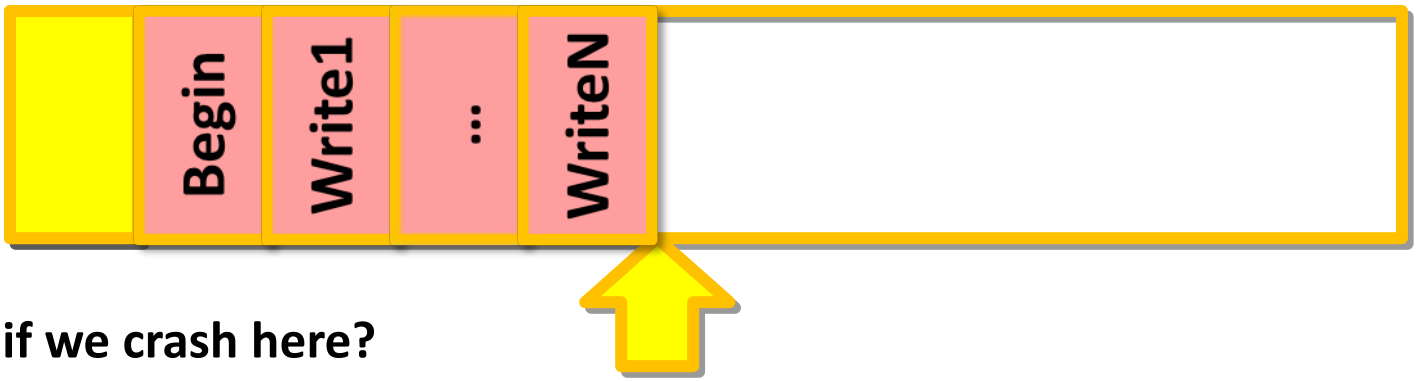


What if we crash here (between 3,4)?
On reboot, reapply committed updates in log order.

Transactions: logging

1. Begin transaction
2. Append info about modifications to a log
3. Append “commit” to log to end x-action
4. Write new data to normal database

- Single-sector write commits x-action (3)



What if we crash here?
On reboot, discard uncommitted updates.

Committing Distributed Transactions

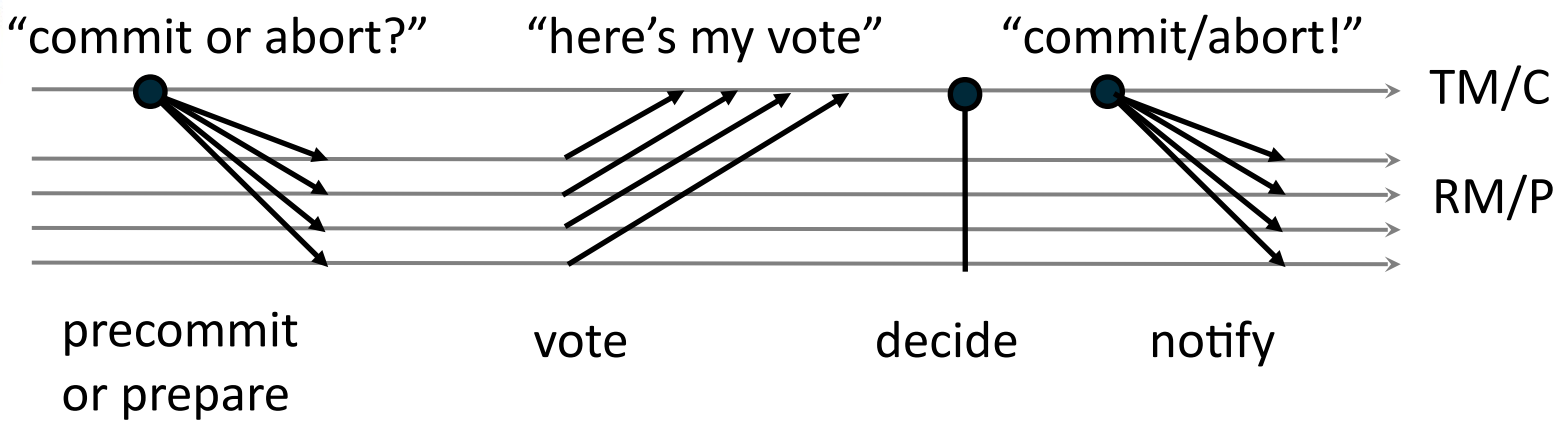
- Transactions may touch data at more than one site.
- Problem: any site may fail or disconnect while a commit for transaction T is in progress.
  - Atomicity says that T does not "partly commit", i.e., commit at some site and abort at another.
  - Individual sites cannot unilaterally choose to abort T without the agreement of the other sites.
  - If T holds locks at a site S, then S cannot release them until it knows if T committed or aborted.
  - If T has pending updates to data at a site S, then S cannot expose the data until T commits/aborts.

Commit is a Consensus Problem

- If there is more than one site, then the sites must agree to commit or abort.
- Sites (Resource Managers or RMs) manage their own data, but coordinate commit/abort with other sites.
  - "Log locally, commit globally."
- We need a protocol for distributed commit.
  - It must be safe, even if FLP tells us it might not terminate.
- Each transaction commit is led by a coordinator (Transaction Manager or TM).

Two-Phase Commit (2PC)

If unanimous to commit
decide to commit
else decide to abort



RMs validate Tx and prepare by logging their local updates and decisions

TM logs commit/abort (commit point)

2PC: Phase 1

- ✓ 1. **Tx** requests commit, by notifying coordinator (**C**)
  - **C** must know the list of participating sites/RMs.
- ✓ 2. Coordinator **C** requests each participant (**P**) to prepare.
- ✓ 3. Participants (RMs) validate, prepare, and vote.
  - Each **P** validates the request, logs validates updates locally, and responds to **C** with its vote to commit or abort.
  - If **P** votes to commit, **Tx** is said to be “prepared” at **P**.

2PC: Phase 2

- ✓ 4. Coordinator (TM) commits.
  - Iff all ***P*** votes are unanimous to commit
    - ***C*** writes a **commit** record to its log
    - ***Tx*** is committed.
  - Else **abort**.
- ✓ 5. Coordinator notifies participants.
  - ***C*** asynchronously notifies each ***P*** of the outcome for ***Tx***.
  - Each ***P*** logs the outcome locally
  - Each ***P*** releases any resources held for ***Tx***.

37

Handling Failures in 2PC

How to ensure consensus if a site fails during the 2PC protocol?

1. A participant **P** fails before preparing.

   Either **P** recovers and votes to abort, or **C** times out and aborts.

2. Each **P** votes to commit, but **C** fails before committing.

   - Participants wait until **C** recovers and notifies them of the decision to abort. The outcome is uncertain until **C** recovers.

Handling Failures in 2PC

3. **P** or **C** fails during phase 2, after the outcome is determined.
    - Carry out the decision by reinitiating the protocol on recovery.
    - Again, if **C** fails, the outcome is uncertain until **C** recovers.

consistency

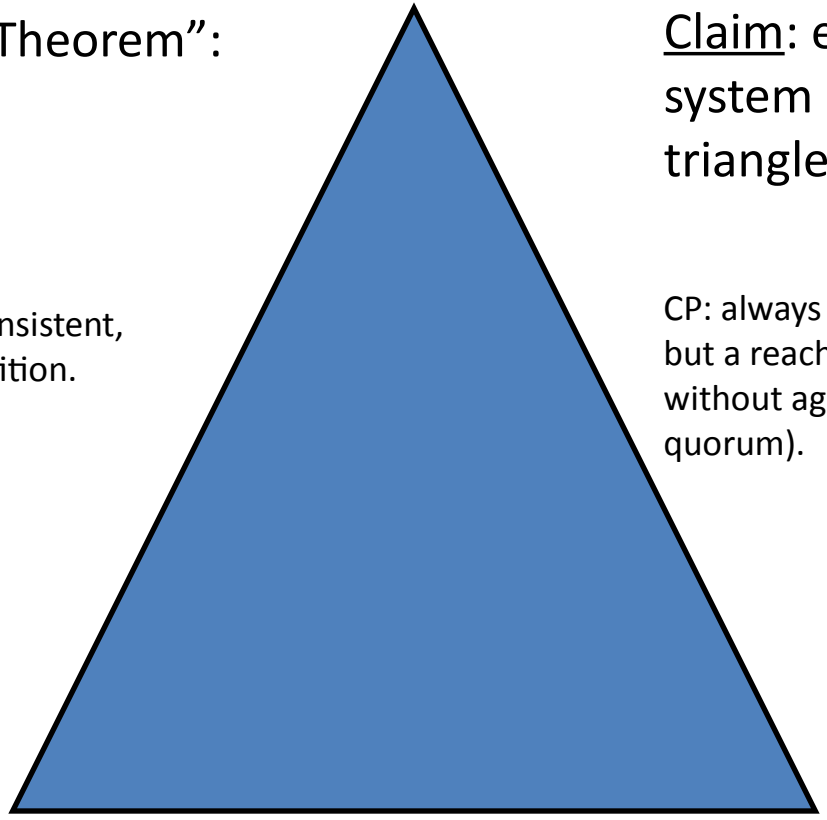
C

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

<u>Claim</u>: every distributed system is on one side of the triangle.

CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).



A

Availability

AP: a reachable replica provides service even in a partition, but may be inconsistent.

P

Partition-resilience

Google GFS: Assumptions

- Design a Google FS for Google's distinct needs
- High component failure rates
  - Inexpensive commodity components fail often
- "Modest" number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads
- High sustained throughput favored over low latency

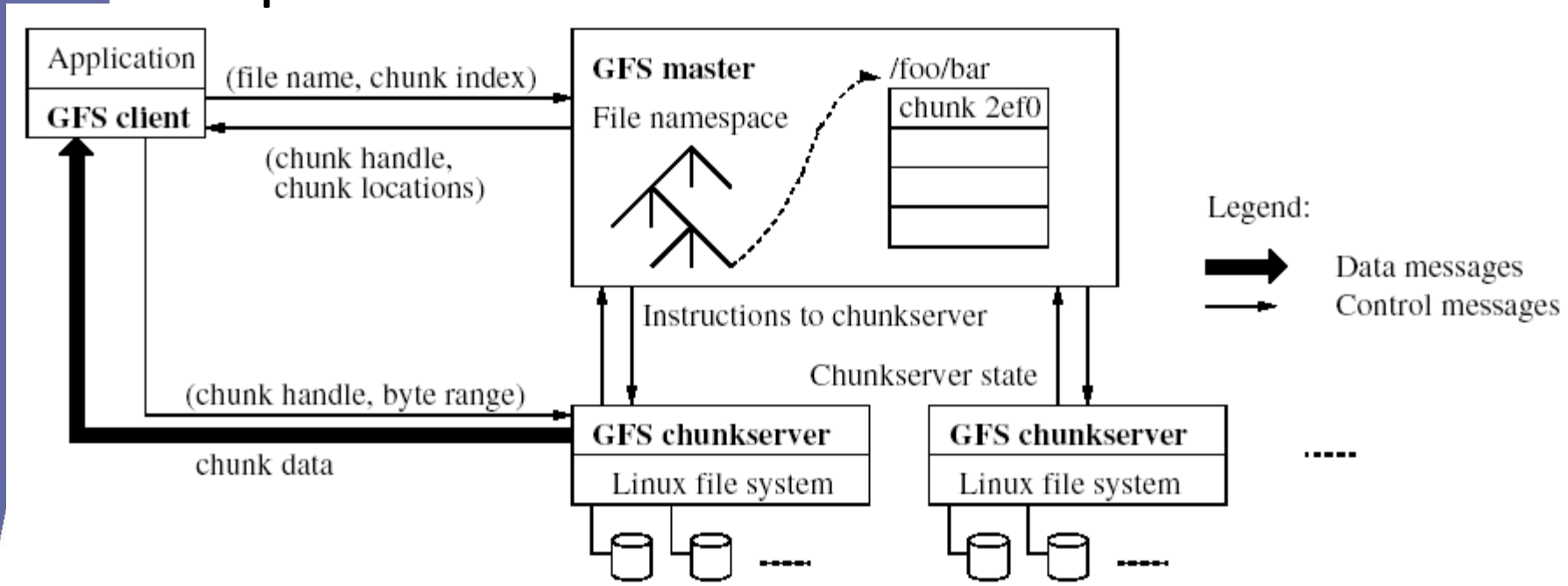
[Alex Moschuk]

GFS Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ *chunkservers*
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
  - Simplify the problem; focus on Google apps
  - Add *snapshot* and *record append* operations

GFS Architecture

- Single master
- Mutiple chunkservers



…Can anyone see a potential weakness in this design?

[Alex Moschuk]

Single master

- From distributed systems we know this is a:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - Shadow masters
  - Minimize master involvement
    - never move data through it, use only for metadata
      - and cache metadata at clients
    - large chunk size
    - master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough!

Fault Tolerance

- High availability
  - fast recovery
    - master and chunkservers restartable in a few seconds
  - chunk replication
    - default: 3 replicas.
  - shadow masters
- Data integrity
  - checksum every 64KB block in each chunk

What is the consensus problem here?

Google Ecosystem

- Google builds and runs services at massive scale.
  - More than half a million servers
- Services at massive scale must be robust and adaptive.
  - To complement a robust, adaptive infrastructure
- Writing robust, adaptive distributed services is hard.
- Google Labs works on tools, methodologies, and infrastructures to make it easier.
  - Conceive, design, build
  - Promote and transition to practice
  - Evaluate under real use

Google Systems

- Google File System (GFS) [SOSP 2003]
  - Common foundational storage layer
- MapReduce for data-intensive cluster computing [OSDI 2004]
  - Used for hundreds of google apps
  - Open-source: Hadoop (Yahoo)
- BigTable [OSDI 2006]
  - a spreadsheet-like data/index model layered on GFS
- Sawzall
  - Execute filter and aggregation scripts on BigTable servers
- Chubby [OSDI 2006]
  - Foundational lock/consensus/name service for all of the above
  - Distributed locks
  - The “root” of distributed coordination in Google tool set

What Good is “Chubby”?

- Claim: with a good lock service, lots of distributed system problems become “easy”.
  - Where have we seen this before?
- Chubby encapsulates the algorithms for consensus.
  - Where does consensus appear in Chubby?
- Consensus in the real world is imperfect and messy.
  - How much of the mess can Chubby hide?
  - How is “the rest of the mess” exposed?
- What new problems does such a service create?

Chubby Structure

- Cell with multiple participants (replicas and master)
  - replicated membership list
  - common DNS name (e.g., DNS-RR)
- Replicas elect one participant to serve as Master
  - master renews its Master Lease periodically
  - elect a new master if the master fails
  - all writes propagate to secondary replicas
- Clients send “master location requests” to any replica
  - returns identity of master
- Replace replica after long-term failure (hours)

Master Election/Fail-over

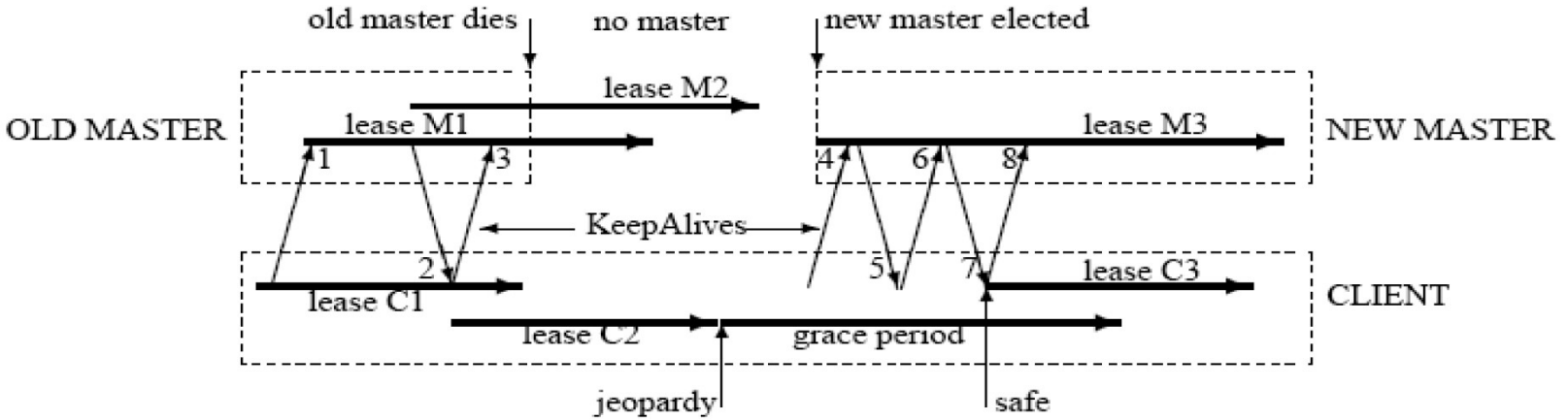
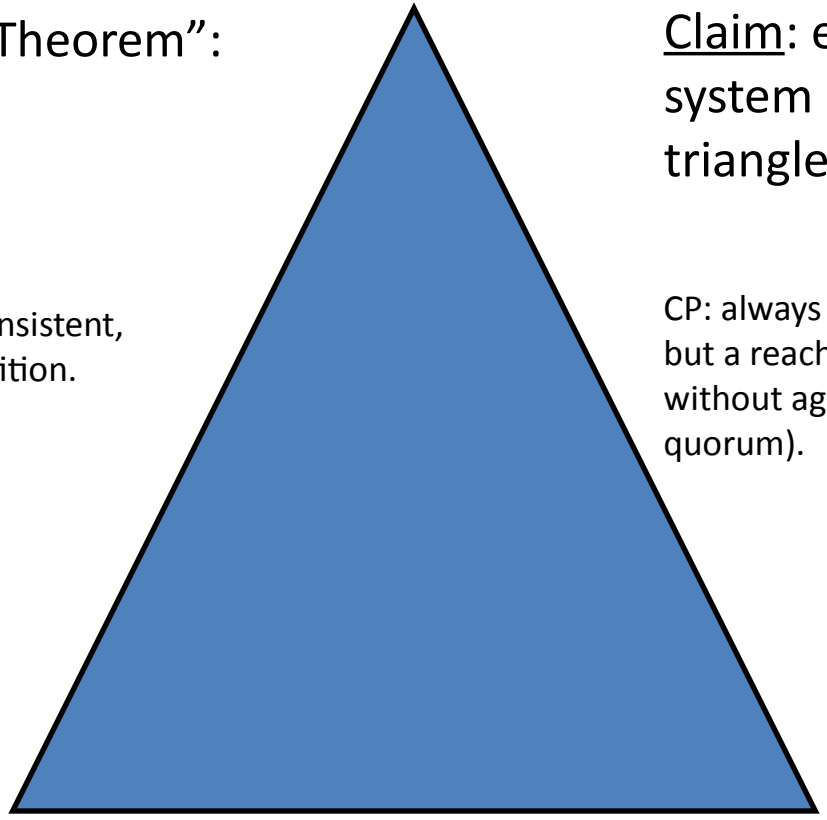


Figure 2: The role of the grace period in master fail-over

Fox&Brewer “CAP Theorem”:
C-A-P: choose two.

consistency

C



A

Availability

P

Partition-resilience

<u>Claim</u>: every distributed system is on one side of the triangle.

CA: available, and consistent, unless there is a partition.

CP: always consistent, even in a partition, but a reachable replica may deny service without agreement of the others (e.g., quorum).

AP: a reachable replica provides service even in a partition, but may be inconsistent.

Relaxing ACID properties

- ACID is hard to achieve, moreover, it is not always required, e.g. for blogs, status updates, product listings, etc.
- Availability
  - Traditionally, thought of as the server/process available 99.999 % of time
  - For a large-scale node system, there is a high probability that a node is either down or that there is a network partitioning
- Partition tolerance
  - ensures that write and read operations are redirected to available replicas when segments of the network become disconnected

Eventual Consistency

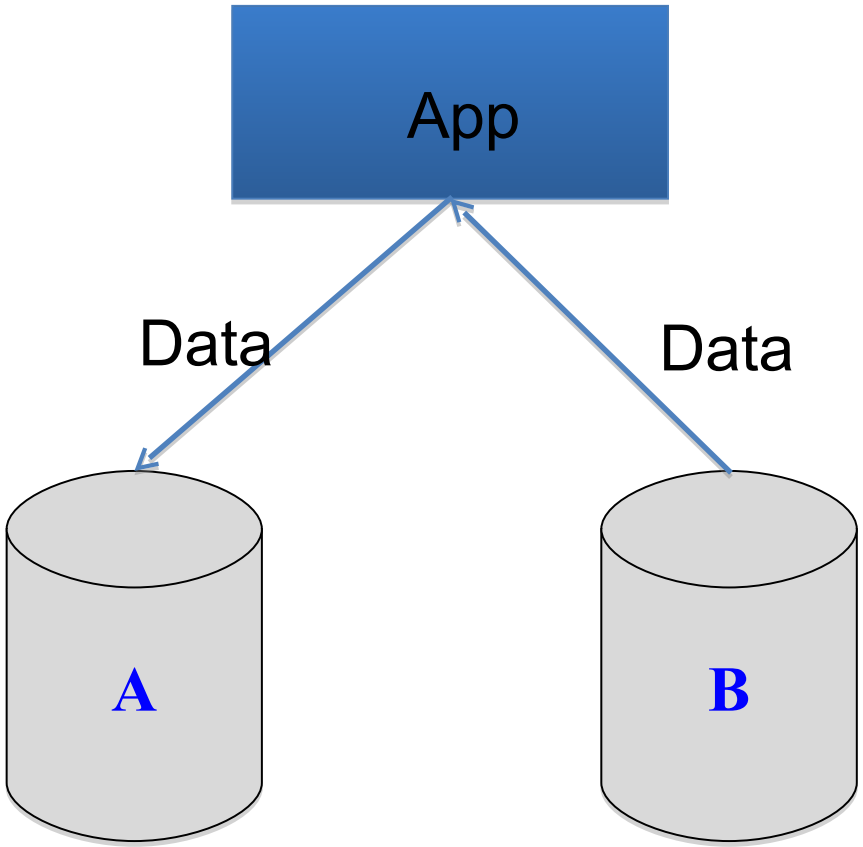
- Eventual Consistency
  - When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
  - For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency) properties, as opposed to ACID
  - Soft state: copies of a data item may be inconsistent
  - Eventually Consistent – copies becomes consistent at some later time if there are no more updates to that data item
  - Basically Available – possibilities of faults but not a fault of the whole system

CAP Theorem

- Suppose three properties of a system
  - Consistency (all copies have same value)
  - Availability (system can run even if parts have failed)
  - Partitions (network can break into two or more parts, each with active systems that can not influence other parts)
- Brewer's CAP "Theorem": for any system sharing data it is impossible to guarantee simultaneously all of these three properties
- Very large systems will partition at some point
  - it is necessary to decide between C and A
  - traditional DBMS prefer C over A and P
  - most Web applications choose A (except in specific applications such as order processing)

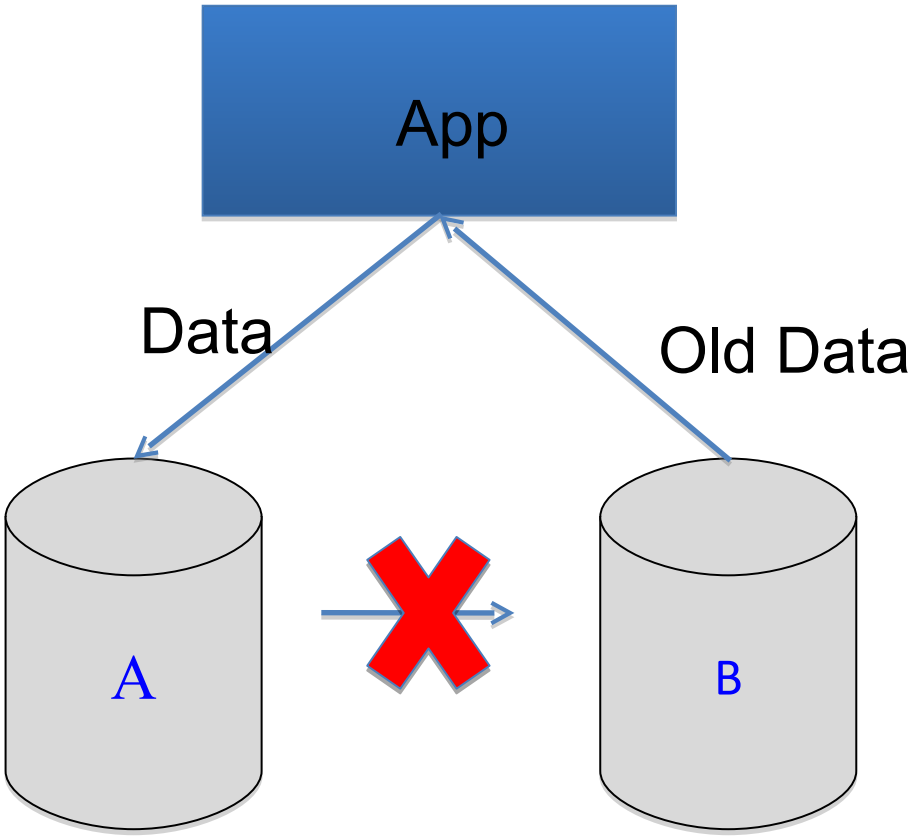
An Elaboration

Consistent and available
No partition.



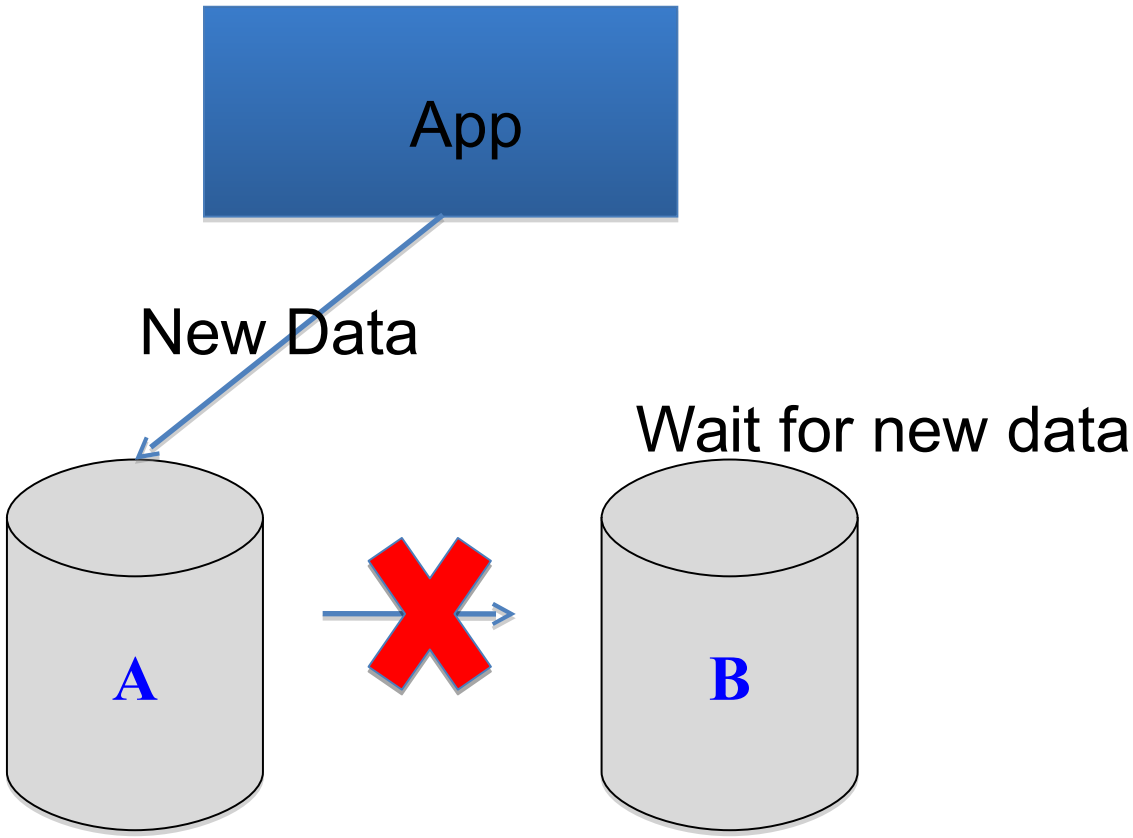
An Elaboration

Available and partitioned
Not consistent, we get back old data.



An Elaboration

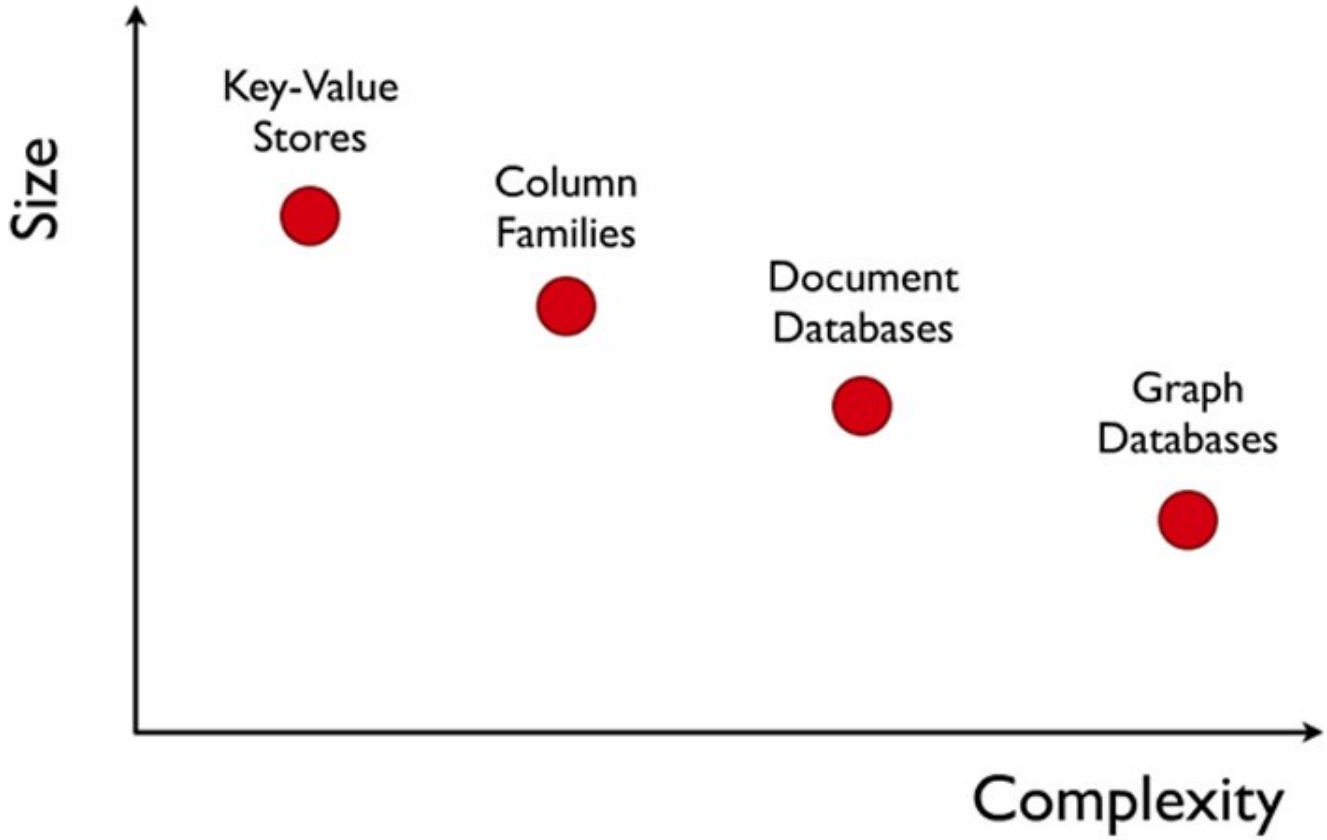
Consistent and partitioned
Not available, waiting…



CAP Theorem

- Drop A or C of ACID
  - relaxing C makes replication easy, facilitates fault tolerance,
  - relaxing A reduces (or eliminates) need for distributed concurrency control.

4 Category


Size
Key-Value Stores
Column Families
Document Databases
Graph Databases
Complexity

Quote

"NoSQL marketing is confusing... Everything does everything and at a small scale everything works."

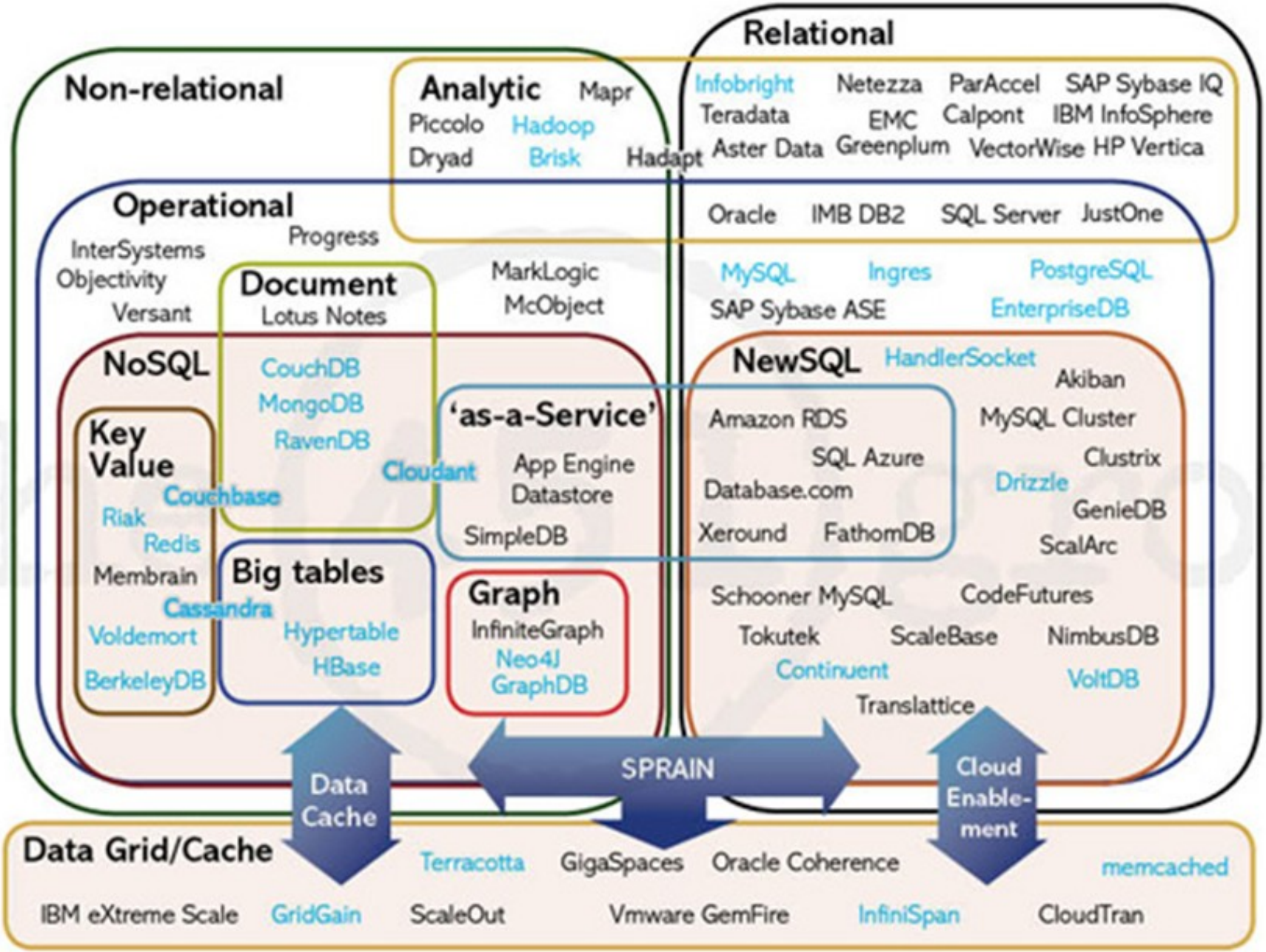
"If you're evaluating Mongo vs. Riak or Couch vs. Cassandra you don't understand either your problem or the technologies."

- Andy Gross, VP Engineering at Basho Technologies

(a*pproximate paraphrasing, hopefully not grossly misquoted*)

Categories of NoSQL databases

- Key-value stores
- Column NoSQL databases
- Document-based
- Graph database (neo4j, InfoGrid)
- XML databases (myXMLDB, Tamino, Sedna)


Relational
Non-relational
Analytic
Mapr
Infobright
Netezza
ParAccel
SAP Sybase IQ
Piccolo
Hadoop
Teradata
EMC
Calpont
IBM InfoSphere
Dryad
Brisk
Hadapt
Aster Data
Greenplum
VectorWise
HP Vertica
Operational
Oracle
IMB DB2
SQL Server
JustOne
Progress
InterSystems
Objectivity
Versant
Document
Lotus Notes
MarkLogic
McObject
MySQL
Ingres
PostgreSQL
SAP Sybase ASE
EnterpriseDB
NoSQL
CouchDB
MongoDB
RavenDB
Cloudant
Couchbase
Riak
Redis
Membrain
Cassandra
Voldemort
BerkeleyDB
'as-a-Service'
App Engine
Datastore
SimpleDB
Big tables
Hypertable
HBase
Graph
InfiniteGraph
Neo4J
GraphDB
NewSQL
HandlerSocket
Akiban
Amazon RDS
MySQL Cluster
SQL Azure
Clustrix
Drizzle
Database.com
GenieDB
Xeround
FathomDB
ScalArc
Schooner MySQL
CodeFutures
Tokutek
ScaleBase
NimbusDB
Continuent
VoltDB
Translattice
Data Cache
SPRAIN
Cloud Enable-ment
Data Grid/Cache
Terracotta
GigaSpaces
Oracle Coherence
memcached
IBM eXtreme Scale
GridGain
ScaleOut
Vmware GemFire
InfiniSpan
CloudTran

Key-Value Stores

Extremely simple interface

- Data model: (key, value) pairs
- Operations: Insert(key,value), Fetch(key), Update(key), Delete(key)

Implementation: efficiency, scalability, fault-tolerance

- Records distributed to nodes based on key
- Replication
- Single-record transactions, “eventual consistency

Key-Value Data Stores

- Example: SimpleDB
  - Based on Amazon's Single Storage Service (S3)
  - items (represent objects) having one or more pairs (name, value), where name denotes an attribute.
  - An attribute can have multiple values.
  - items are combined into domains.

Riak

Shell

Close

```
~$
~$
~$
~$
~$

~$ curl -XPUT "http://localhost:8098/riak/bucket/key" \
>
-H "Content-Type: application/json" \
>
-d '{"course":"comp5323"}'
~$
```

Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://rstudio.comp....098/riak/bucket/key × RStudio

rstudio.comp.polyu.edu.hk:8098/riak/bucket/key

```
{"course":"comp5323"}
```

Example

```
~$ curl -XPUT "http://localhost:8098/riak/bucket/kimman" \
>
-H "Content-Type: application/json" \
>
-d '{"course leader":"kimman"}'
~$ curl http://localhost:8098/riak/bucket/kimman
{"course leader":"kimman"}~$
```

Key-Value Stores

Extremely simple interface

- Data model: (key, value) pairs
- Operations: Insert(key,value), Fetch(key), Update(key), Delete(key)

Implementation: efficiency, scalability, fault-tolerance

- Records distributed to nodes based on key
- Replication
- Single-record transactions, “eventual consistency”

Suitable Use Cases

- Storing Session Information
- User Profiles, Preferences: Almost every user has a unique userID as well as preferences such as language, color, timezone, which products the user has access to , and so on.

Shopping Cart Data

- As we want the shopping carts to be available all the time, across browsers, machines, and sessions, all the shopping information can be put into value where the key is the userID

Not to Use

- Relationships among data
- Multi-operation Transactions
- Query by Data
- Operations by Sets: since operations are limited to one key at a time, there is no way to operate upon multiple keys at the same time. If you need to operate upon multiple keys, you have to handle this from the client side

Document Stores

Like Key-Value Stores except value is document

- Data model: (key, document) pairs
- Document: JSON, XML, other semistructured formats
- Basic operations: Insert(key,document), Fetch(key), Update(key), Delete(key)
- Also Fetch based on document contents

Example systems

- CouchDB, MongoDB, SimpleDB etc

Document-Based

- based on JSON format: a data model which supports lists, maps, dates, Boolean with nesting
- Really: *indexed* semistructured documents
- Example: Mongo
  - {Name:"Jaroslav",

    Address:"Malostranske nám. 25, 118 00 Praha 1“

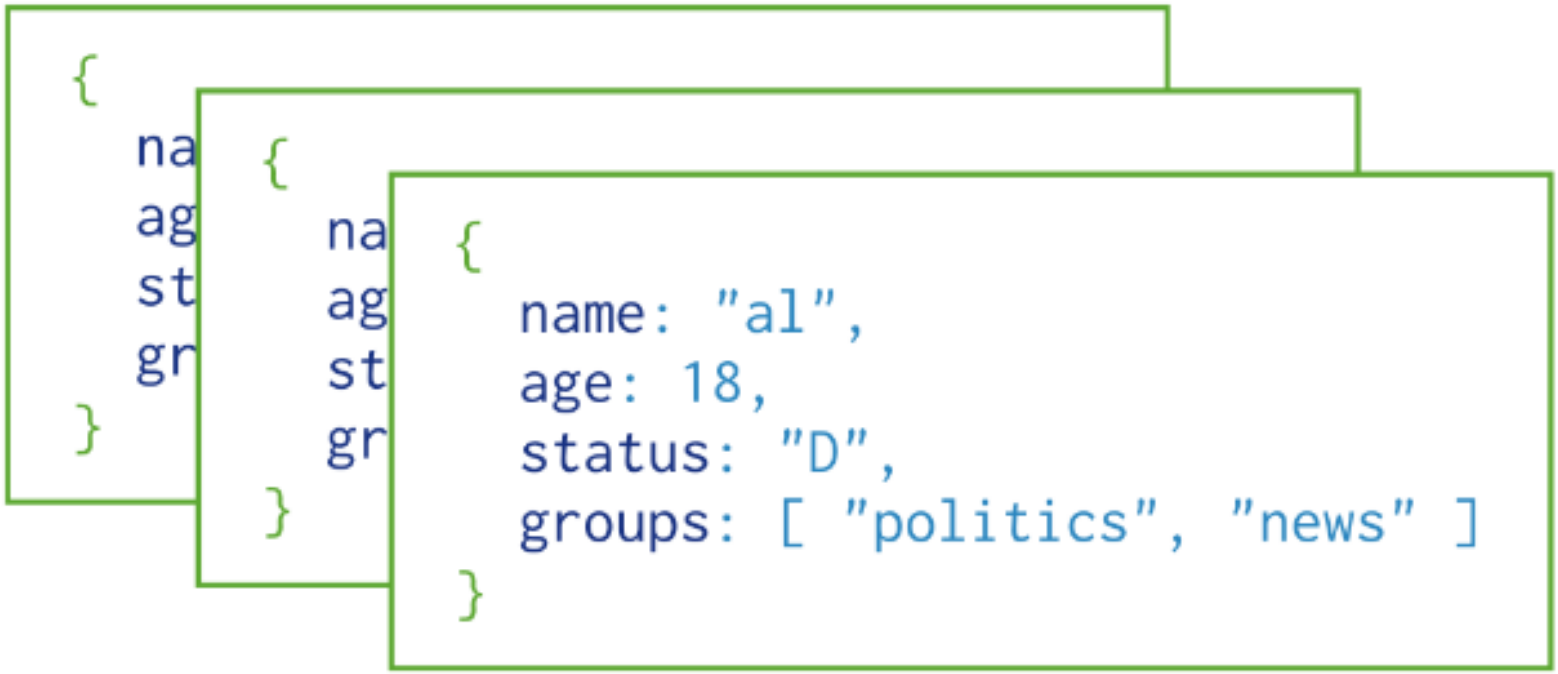
    Grandchildren: [Claire: "7", Barbara: "6", "Magda: "3",
    "Kirsten: "1", "Otis: "3", Richard: "1"]

    }

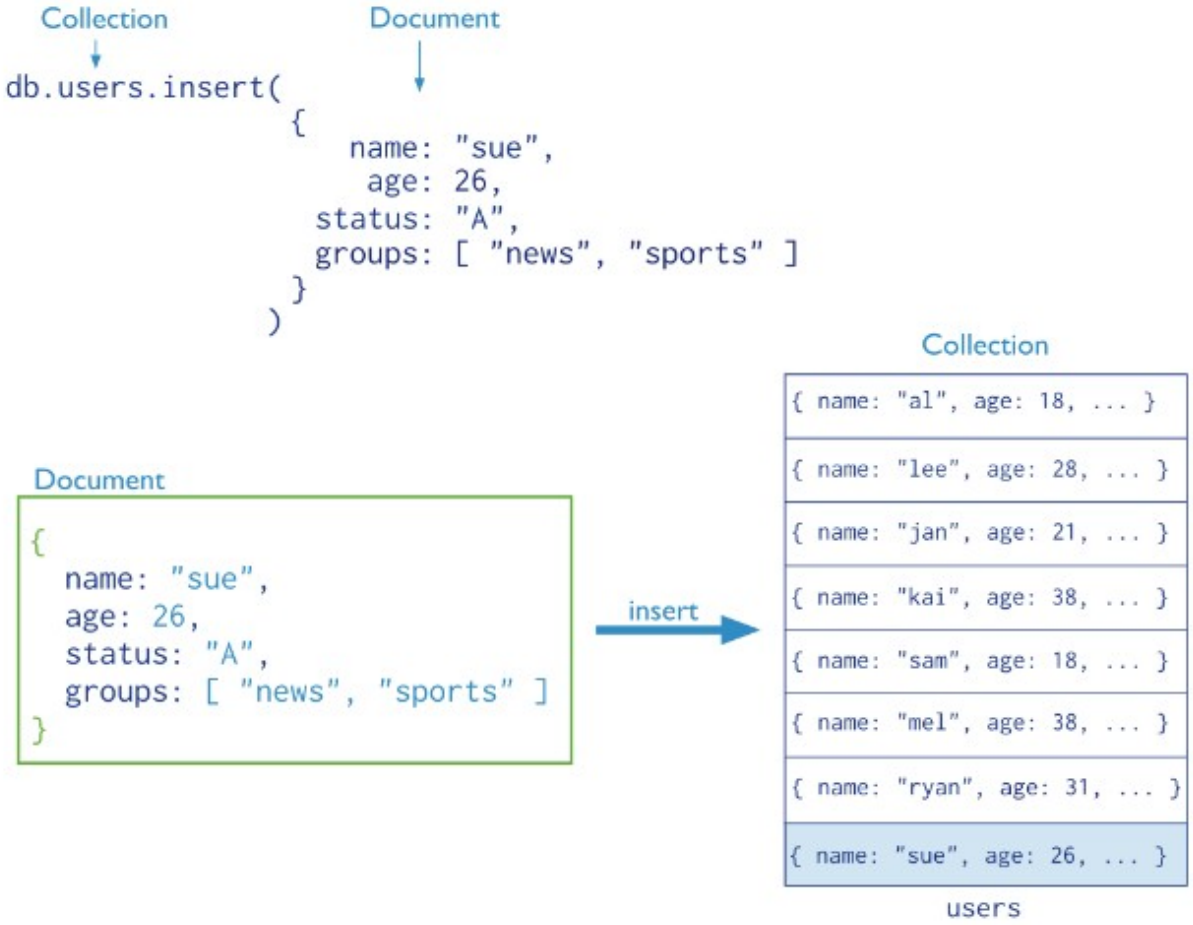
MongoDB CRUD operations

- CRUD stands for create, read, update, and delete
- MongoDB stores data in the form of documents, which are JSON-like field and value pairs.

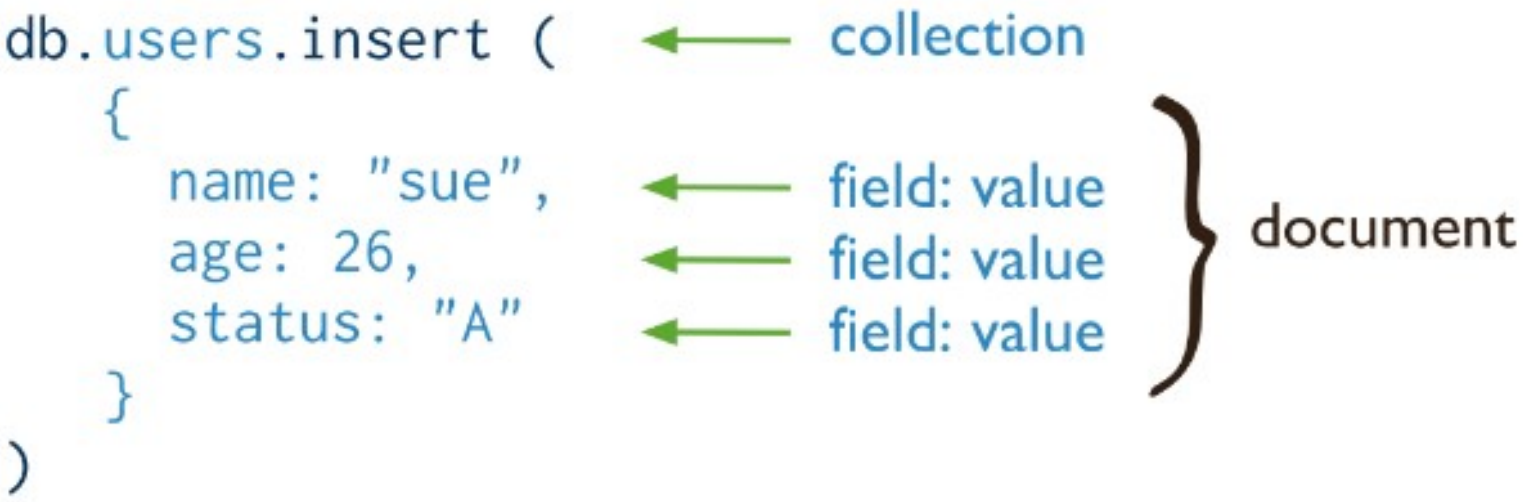
A collection of MongoDB documents



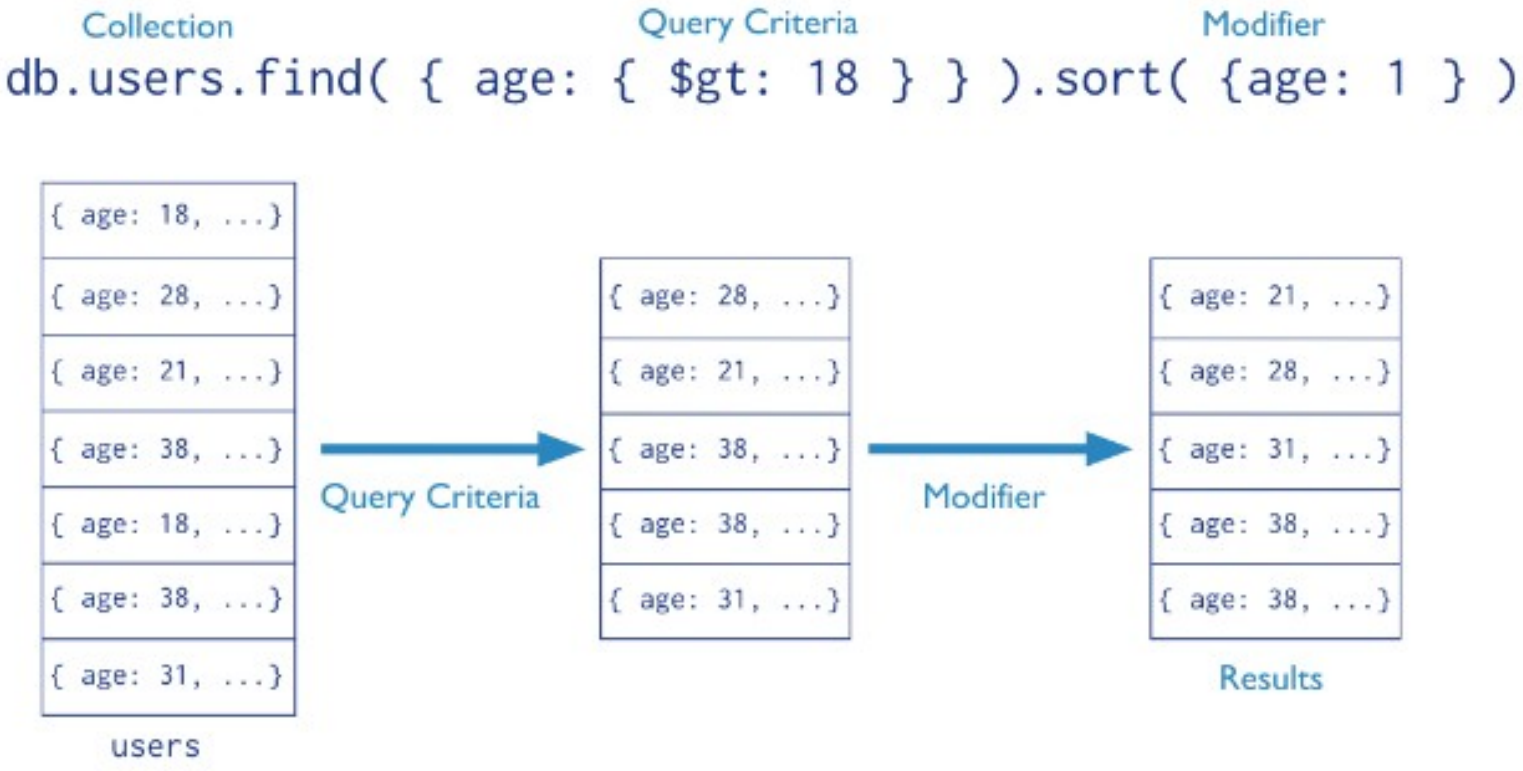
Insert Operation



Insert Operation



Query Opeation



Update Operation

```
db.inventory.update(
    { type : "book" },
    { $inc : { qty : -1 } },
    { multi: true }
)
```

Delete Operation

- db.inventory.remove()
- db.inventory.remove({ type : "food“ })

- Try
- http://try.mongodb.org/

Suitable Use Cases

- Event Logging
- Content Management Systems
- Web Analytics or Real time Analysis
- E-commerce Applications

Not to use

- Complex Transaction Spanning Different Operations
- Queries against Varying Aggregate Sturcture

Column-oriented

- Store data in column order
- Allow key-value pairs to be stored (and retrieved on key) in a massively parallel system
  - data model: families of attributes defined in a schema, new attributes can be added
  - storing principle: big hashed distributed tables
  - properties: partitioning (horizontally and/or vertically), high availability etc. completely transparent to application

Cassandra

"Fortuneteller of Doom"

from Greek Mythology. Tried to warn others about future disasters, but no one listened. Unfortunately, she was 100% accurate.

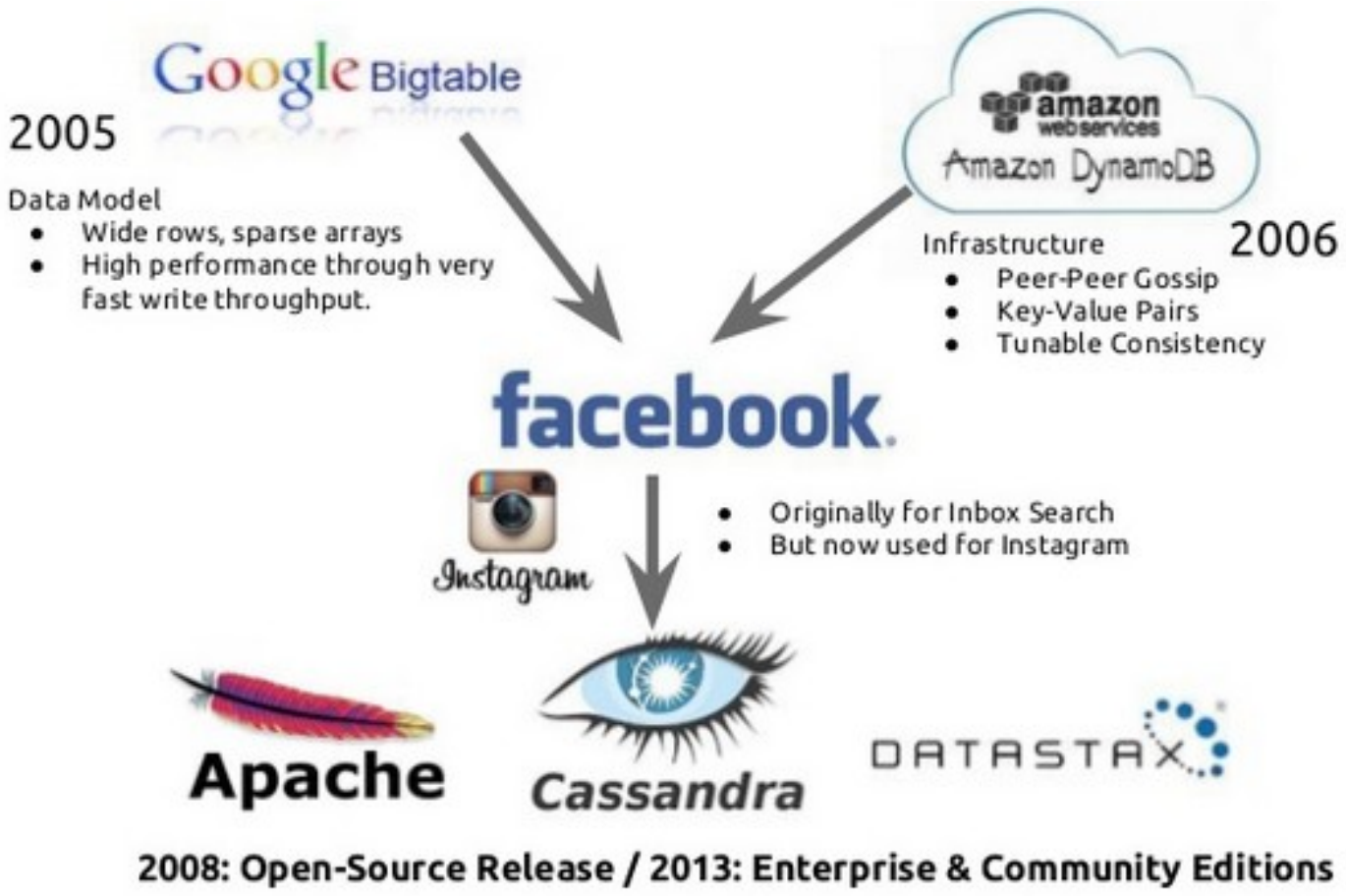
NoSQL Distributed DB

- **Consistency** - A__ID
- **Availability** - High
- **Point of Failure** - none
- **Good for Event Tracking & Analysis**
  - Time series data
  - Sensor device data
  - Social media analytics
  - Risk Analysis
  - Failure Prediction

Cassandra

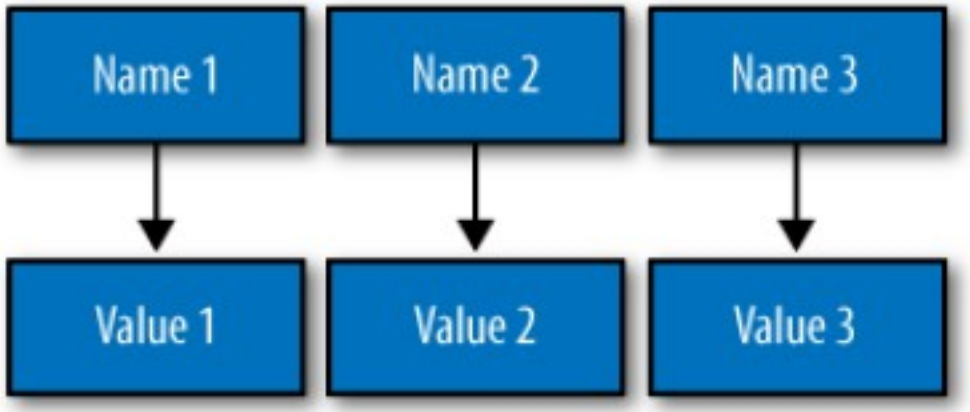
- keyspace: Usually the name of the application; e.g., 'Twitter', 'Wordpress‘.
- column family: structure containing an unlimited number of rows
- column: a tuple with name, value and time stamp
- key: name of record
- super column: contains more columns

Cassandra

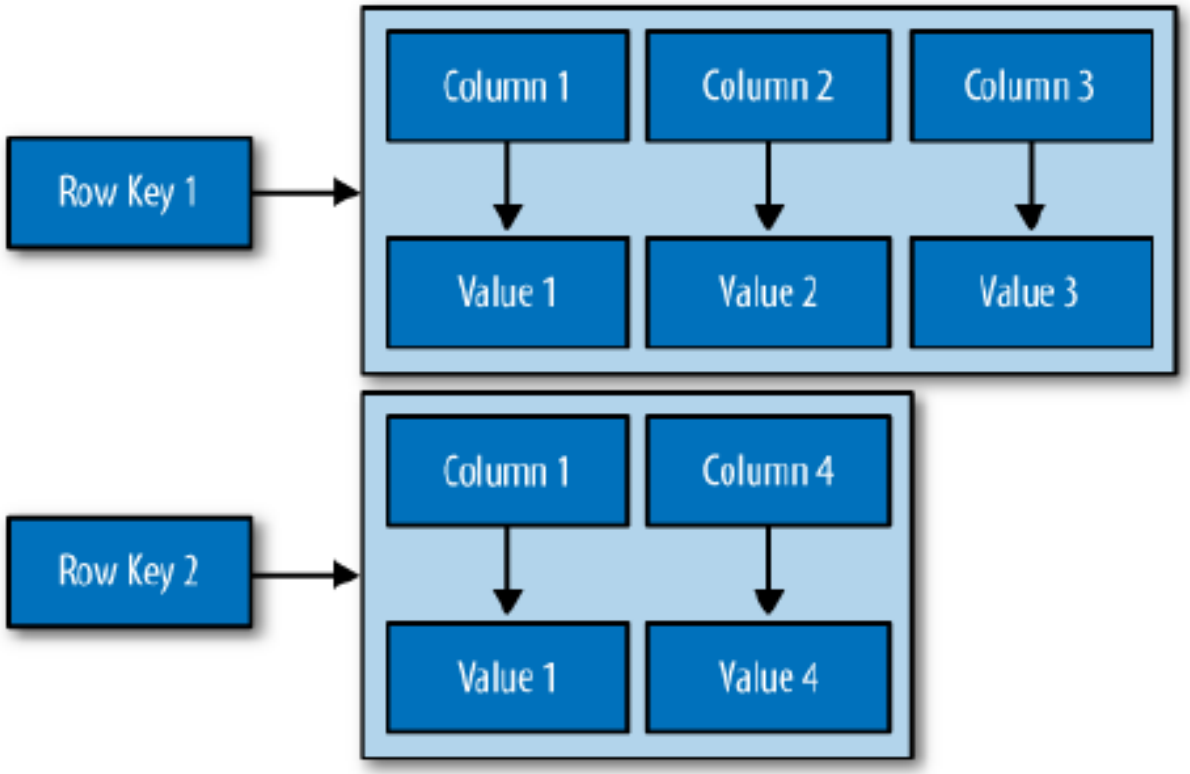

Google Bigtable
2005
Data Model
Wide rows, sparse arrays
High performance through very fast write throughput.
amazon web services
Amazon DynamoDB
2006
Infrastructure
Peer-Peer Gossip
Key-Value Pairs
Tunable Consistency
facebook
Instagram
Originally for Inbox Search
But now used for Instagram
Apache
Cassandra
DATASTAX
2008: Open-Source Release / 2013: Enterprise & Community Editions

list of values

- A map of name/value pairs



Column Family

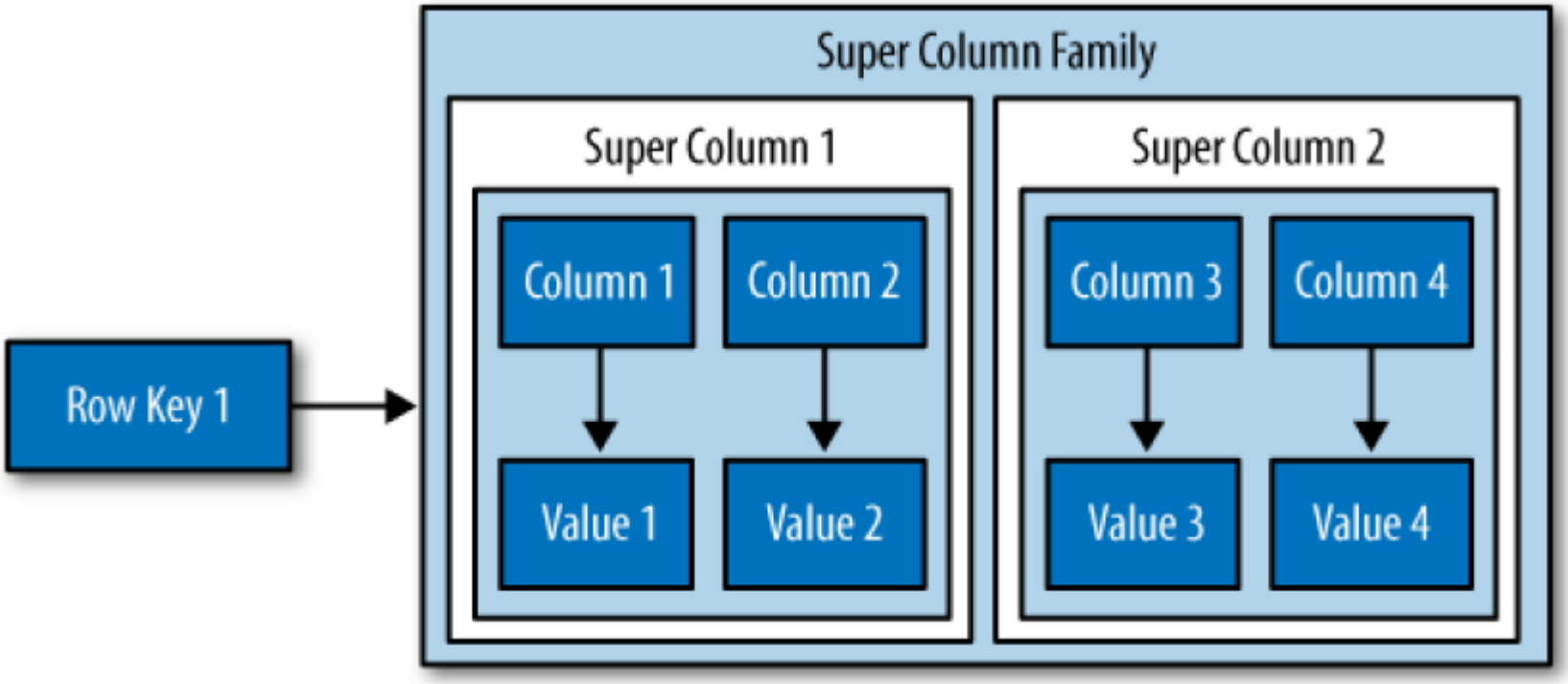


Example

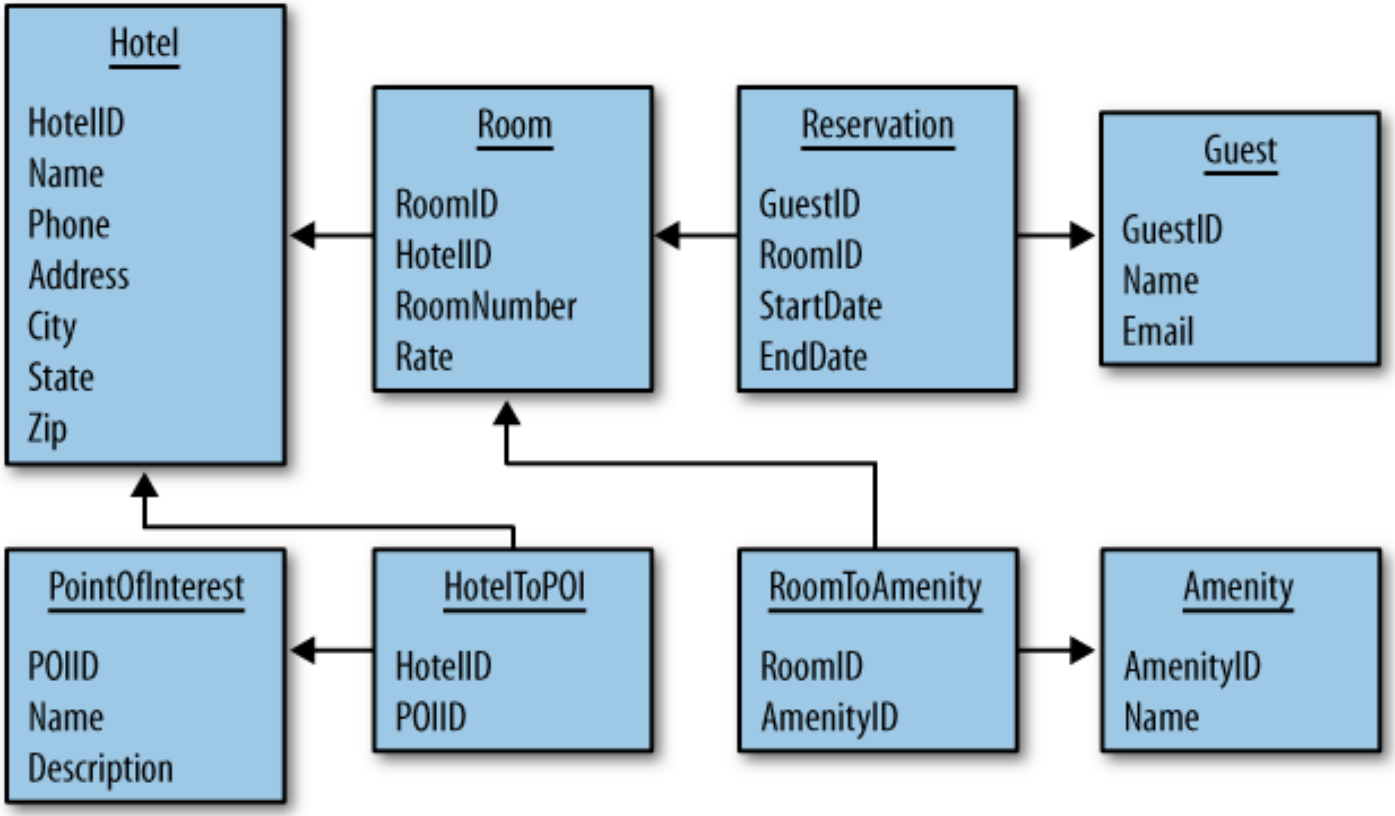
```
Musician:                                ColumnFamily 1
    bootsy:                              RowKey
        email: bootsy@pfunk.com,         ColumnName:Value
        instrument: bass                 ColumnName:Value
    george:                              RowKey
        email: george@pfunk.com          ColumnName:Value

Band:                                    ColumnFamily 2
    george:                              RowKey
        pfunk: 1968-2010                 ColumnName:Value
```

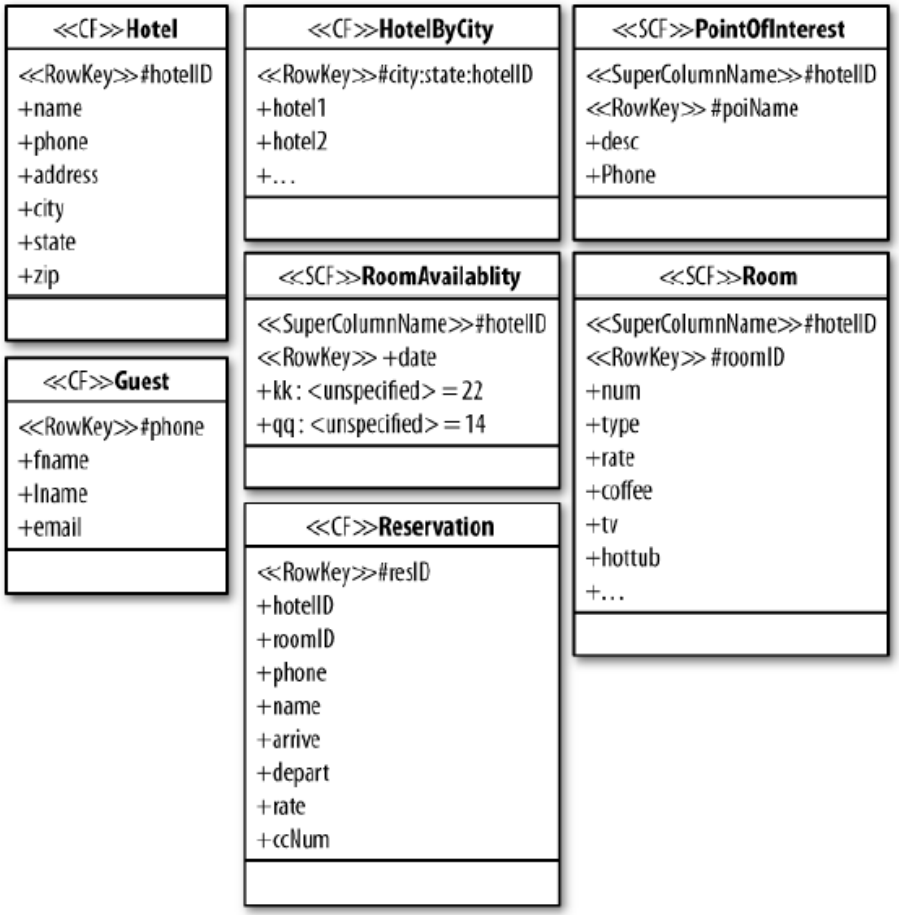
Super Column Family



Simple Hotel Search System (RDBMS)



Cassandra

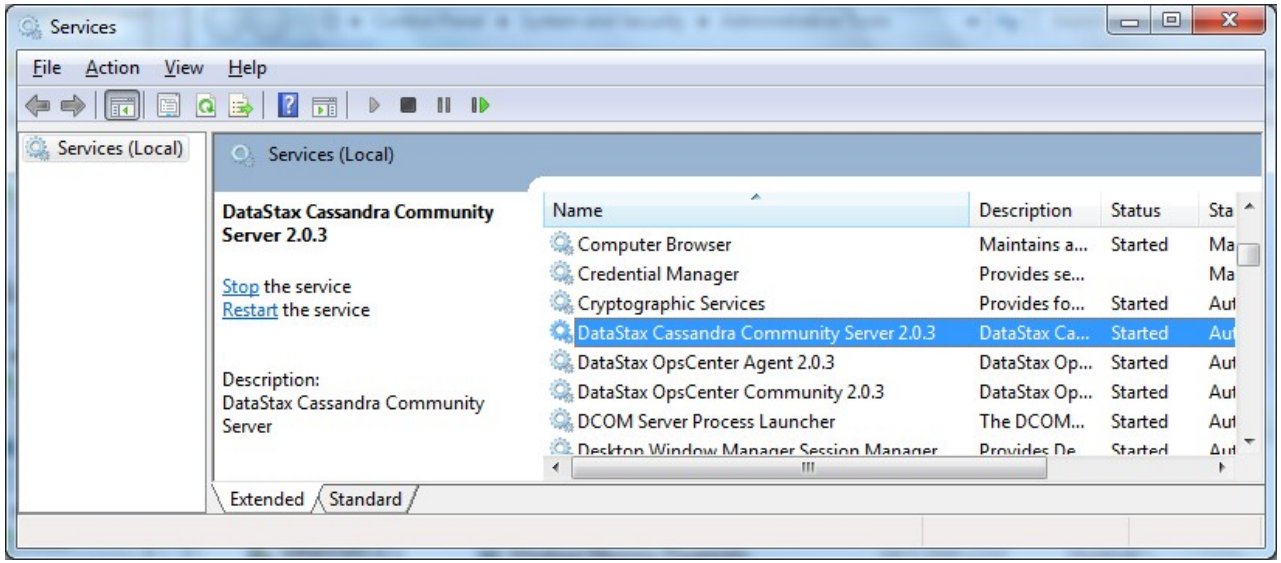

≪CF≫Hotel
≪RowKey≫#hotelID
+name
+phone
+address
+city
+state
+zip
≪CF≫HotelByCity
≪RowKey≫#city:state:hotelID
+hotel1
+hotel2
+…
≪SCF≫PointOfInterest
≪SuperColumnName≫#hotelID
≪RowKey≫ #poiName
+desc
+Phone
≪CF≫Guest
≪RowKey≫#phone
+fname
+lname
+email
≪SCF≫RoomAvailablity
≪SuperColumnName≫#hotelID
≪RowKey≫ +date
+kk : <unspecified> = 22
+qq : <unspecified> = 14
≪SCF≫Room
≪SuperColumnName≫#hotelID
≪RowKey≫ #roomID
+num
+type
+rate
+coffee
+tv
+hottub
+…
≪CF≫Reservation
≪RowKey≫#resID
+hotelID
+roomID
+phone
+name
+arrive
+depart
+rate
+ccNum

Cassandra Query Language

- The thrift API has historically confuse people coming from the relational world with the fact that it uses the terms “rows” and “columns”, but with a different meaning than in SQL.
- CQL3 fixes that since in the model it exposes, row and columns have the same meaning than in SQL.
- We believe this to be an improvement for newcomers, but unfortunately, in doing so, it creates some temporary confusion when you want to switch from thrift to CQL3, as a “thrift” row doesn’t always map to a “CQL3″ row, and a “CQL3″ column doesn’t always map to a “thrift” column.

Cassandra


DATASTAX DOCUMENTATION
Planet Cassandra Community
Getting Started
DOWNLOAD
Getting started
DataStax Enterprise
Cassandra
OpsCenter
CQL
Drivers
Upgrade
Documentation > Home
Search document
Getting started with Cassandra and DataStax Enterprise
Introducing Cassandra
Introducing DataStax Enterprise
Installing Cassandra
Installing DataStax Community on Windows
Install Cassandra and OpsCenter on 32- or 64-bit Windows 7 or Windows Server 2008.


Services
File
Action
View
Help
Services (Local)
Services (Local)
DataStax Cassandra Community Server 2.0.3
Stop the service
Restart the service
Description:
DataStax Cassandra Community Server
Name
Description
Status
Computer Browser
Maintains a...
Started
Credential Manager
Provides se...
Cryptographic Services
Provides fo...
Started
DataStax Cassandra Community Server 2.0.3
DataStax Ca...
Started
DataStax OpsCenter Agent 2.0.3
DataStax Op...
Started
DataStax OpsCenter Community 2.0.3
DataStax Op...
Started
DCOM Server Process Launcher
The DCOM...
Started
Extended
Standard

Suitable Use Cases

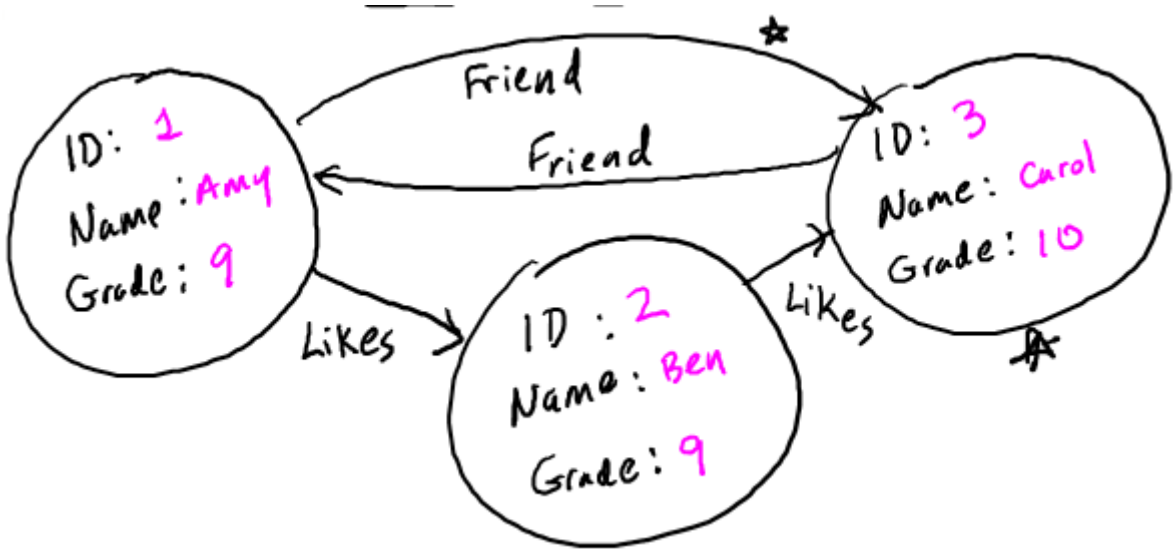
- Event Logging
- Content management Systems, blogging platforms

Not to use

- There are problems for which column-family databases are not best solutions, such as systems that require ACID transactions for writes and reads.
- If you need the database to aggregate the data using queries (such as SUM or AVG), you have to do this on the client side using data retrieved by the client from all the rows.

Graph Database

- Data model: nodes and edges
- Nodes may have properties (including ID)
- Edges may have labels or roles

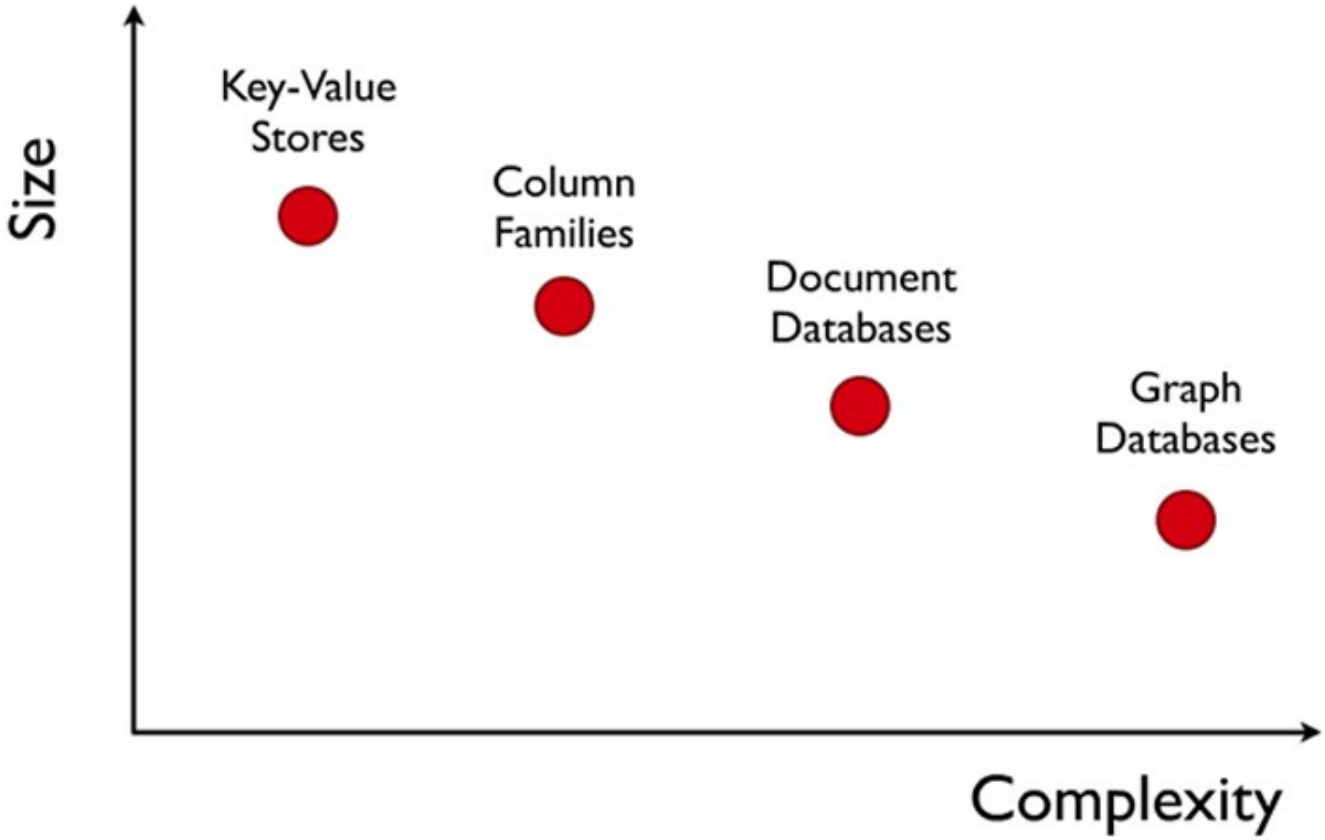


Graph Database Systems

- Interfaces and query languages vary
- Single-step versus “path expressions” versus full recursion
- Example systems

  Neo4j, FlockDB, Pregel, …
- RDF “triple stores” can map to graph databases

5 Summary


Size
Key-Value Stores
Column Families
Document Databases
Graph Databases
Complexity

Typical NoSQL API

- Basic API access:
  - get(key) -- Extract the value given a key
  - put(key, value) -- Create or update the value given its key
  - delete(key) -- Remove the key and its associated value
  - execute(key, operation, parameters) -- Invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc).

Representatives of NoSQL Databases (key-valued)

Name	Producer	Data model	Querying
SimpleDB	Amazon	set of couples (key, {attribute}), where attribute is a couple (name, value)	restricted SQL; select, delete, GetAttributes, and PutAttributes operations
Redis	Salvatore Sanfilippo	set of couples (key, value), where value is simple typed value, list, ordered (according to ranking) or unordered set, hash value	primitive operations for each value type
Dynamo	Amazon	like SimpleDB	simple get operation and put in a context
Voldemort	LinkeId	like SimpleDB	similar to Dynamo

Representatives of NoSQL Databases (column-oriented)

Name	Producer	Data model	Querying
BigTable	Google	set of couples (key, {value})	selection (by combination of row, column, and time stamp ranges)
HBase	Apache	groups of columns (a BigTable clone)	JRUBY IRB-based shell (similar to SQL)
Hypertable	Hypertable	like BigTable	HQL (Hypertext Query Language)
CASSANDRA	Apache (originally Facebook)	columns, groups of columns corresponding to a key (supercolumns)	simple selections on key, range queries, column or columns ranges
PNUTS	Yahoo	(hashed or ordered) tables, typed arrays, flexible schema	selection and projection from a single table (retrieve an arbitrary single record by primary key, range queries, complex predicates, ordering, top-k)

Representatives of NoSQL Databases (document-based)

Name	Producer	Data model	Querying
MongoDB	10gen	object-structured documents stored in collections; each object has a primary key called ObjectId	manipulations with objects in collections (find object or objects via simple selections and logical expressions, delete, update,)
Couchbase	Couchbase[1]	document as a list of named (structured) items (JSON document)	by key and key range, views via Javascript and MapReduce

[1] after merging Membase and CouchOne

Summary

- NoSQL database cover only a part of data-intensive cloud applications (mainly Web applications).
- Problems with cloud computing:
  - SaaS applications require enterprise-level functionality, including ACID transactions, security, and other features associated with commercial RDBMS technology, i.e. NoSQL should not be the only option in the cloud.
  - Hybrid solutions:
    - Voldemort with MySQL as one of storage backend
    - deal with NoSQL data as semistructured data

      $\Rightarrow$ integrating RDBMS and NoSQL via SQL/XML

Summary

- Next generation of highly scalable and elastic RDBMS: NewSQL databases (from April 2011)
  - they are designed to scale out horizontally on shared nothing machines,
  - still provide ACID guarantees,
  - applications interact with the database primarily using SQL,
  - the system employs a lock-free concurrency control scheme to avoid user shut down,
  - the system provides higher performance than available from the traditional systems.
- Examples: MySQL Cluster (most mature solution), VoltDB, Clustrix, ScalArc, …

Summary

- New buzzword: SPRAIN – 6 key factors for alternative data management:
  - Scalability
  - Performance
  - relaxed consistency
  - Agility
  - Intracacy
  - Necessity

Reference

- A Little Riak book http://littleriakbook.com/
- https://www.mongodb.org/
- http://cassandra.apache.org/
- http://nosql-database.org/