CLOUD DATABASES

CLOUD COMPUTING

Dr. Atm Shafiul Alam

a.alam@qmul.ac.uk

Queen Mary University of London

School of Electronic Engineering and Computer Science

Thanks to Dr. Gareth Tyson and Dr. Félix Cuadrado who created the original version of these slides

Yesterday we…

- Learnt about DNS
- Learnt about Content Delivery Networks (CNDs)
  - And how CDNs use DNS to redirect clients
- Learnt about real deployment of CDNs
  - E.g. Akamai, ChinaNet, ChinaCache

REWIND

Contents

Traditional Databases

- Most traditional databases are relational
- Relational data model has been developed to serve applications that insist on data consistency and reliability
- Data consistency and reliability are guaranteed by ACID properties
- ACID stands for **A**tomicity, **C**onsistency, **I**solation, and **D**urability

SQL Tables

orders

OID	CID	ODATE	SDATE
1	1	29/4/2008	NULL
2	2	20/3/2008	24/3/2008

clients

CID	NAME	COMPANY	ADDRESS
1	Miguel	Telefonica	C. Emilio V
2	Guillermo	UPM	Av Complut

products

PID	DESCRIPTION	PRICE
1	Intel P4	$200
2	Intel P3	$49
3	AthlonXP	$100
4	ASUS	$128
5	TYAN	$400

order_details

OID	PID	AMOUNT
1	1	2
1	5	2
2	2	1

Sample SQL Queries

- `INSERT INTO orders VALUES (1000, 1, ‘2014-04-29’, ‘2014-05-01’);`
- `UPDATE products SET PRICE=200, DESCRIPTION=‘Intel Pentium M 1.7GHz’ WHERE PID = ‘cpu-0001’;`
- `DELETE FROM orders WHERE ODATE < ‘2012-12-31’ and SDATE IS NOT NULL;`
- `SELECT DESCRIPTION, PRICE FROM products;`
- `SELECT * FROM products;`
- `SELECT * FROM products WHERE PRICE < 300;`
- `SELECT * FROM products WHERE PID = ‘cpu-0001’;`

ACID Database Properties

- **Atomicity**
  - Transactions must act as a whole, or not at all
  - No changes within a transaction can persist if any change within the transaction fails
- **Consistency**
  - Changes made by a transaction respect all database integrity constraints and the database remains in a consistent state at the end of a transaction
- **Isolation**
  - Transactions cannot see changes made by other concurrent transactions
  - Concurrent transactions leave the database in the same state as if the transactions were executed serially
- **Durability**
  - Database data are persistent,
  - Changes made by a transaction that completed successfully are stored permanently

NoSQL Database Systems

- Many cloud databases are not relational
  - E.g. BigTable, Dynamo, PNUTS/Sherpa, ..
- They are termed **NoSQL** databases (Not only SQL)
- Generally, NoSQL database systems:
  - Relax one or more of the ACID properties
  - Adopt the key-value data model for storing semi-structured or unstructured data
  - Are either scheme less, or their schemas lack rigidity and integrity constraints of relational schemes
  - Use data partitioning, replication and distribution to several independent network nodes made of commodity hardware
  - Do not support joins and use data sharding (horizontal partitioning) to avoid need for joins and achieve a good throughput

Relational vs. noSQL

Comparison criteria	**NoSQL database**	**Relational database**
Type of data handled	Mainly unstructured data	Only structured data
Volume of data	High Volume	Low Volume
Type of transactions handled	Simple	Complex
Single point of failure	No	Yes
Data arriving from	Many locations	A few locations

Common Advantages of NoSQL Systems

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - When data is written, the latest version is on at least one node and then replicated to other nodes
  - No single point of failure
- Easy to distribute
- Don't require a schema

Why change databases?

- Relational databases are powerful
  - SQL is a very expressive language
  - Relations between tables among to express rich queries
- … but there are issues as scale keeps growing
  - Vertical scaling works up to some extent
  - Horizontal scaling does not support ACID properties
    - Brewer's CAP Theorem
    - Relax need for ACID properties…

Contents

- Cloud Databases
- **CAP Theorem**
- CAP Consistency
- Data replication
- Cloud DB Services

Brewer's CAP Theorem

- Brewer's Conjecture (2000)
  - Symposium on Principles of Distributed Computing
  - Formally proven in 2002 (Gilbert and Lynch, MIT)
- **Brewer's CAP "Theorem"**: A distributed system cannot guarantee at the same time all three of:
  - Strong **Consistency** (all copies have same value)
  - **Availability** (system can run even if parts have failed)
  - Network **Partitioning** tolerance (network can break into two or more parts, each with active systems that can't talk to other parts)

CAP Theorem

- **Strong Consistency** - A service is considered to be ***strongly consistent*** if after an update operation of some writer, all readers see his updates in some shared data source (all nodes containing replicas of a datum have the same value)
- **Availability** - A service is considered to have a ***high availability*** if it allows read and write operations a high proportion of time (even in the case of a node crash or some hardware or software parts are down due to upgrades)
- **Network partition tolerance** is the ability of a service to perform expected operations in the presence of a network partition, unless the whole network crashes . A network partition occurs if two or more "islands" of network nodes cannot connect to each other. Dynamic addition and removal of nodes is also considered to be a network partition.)

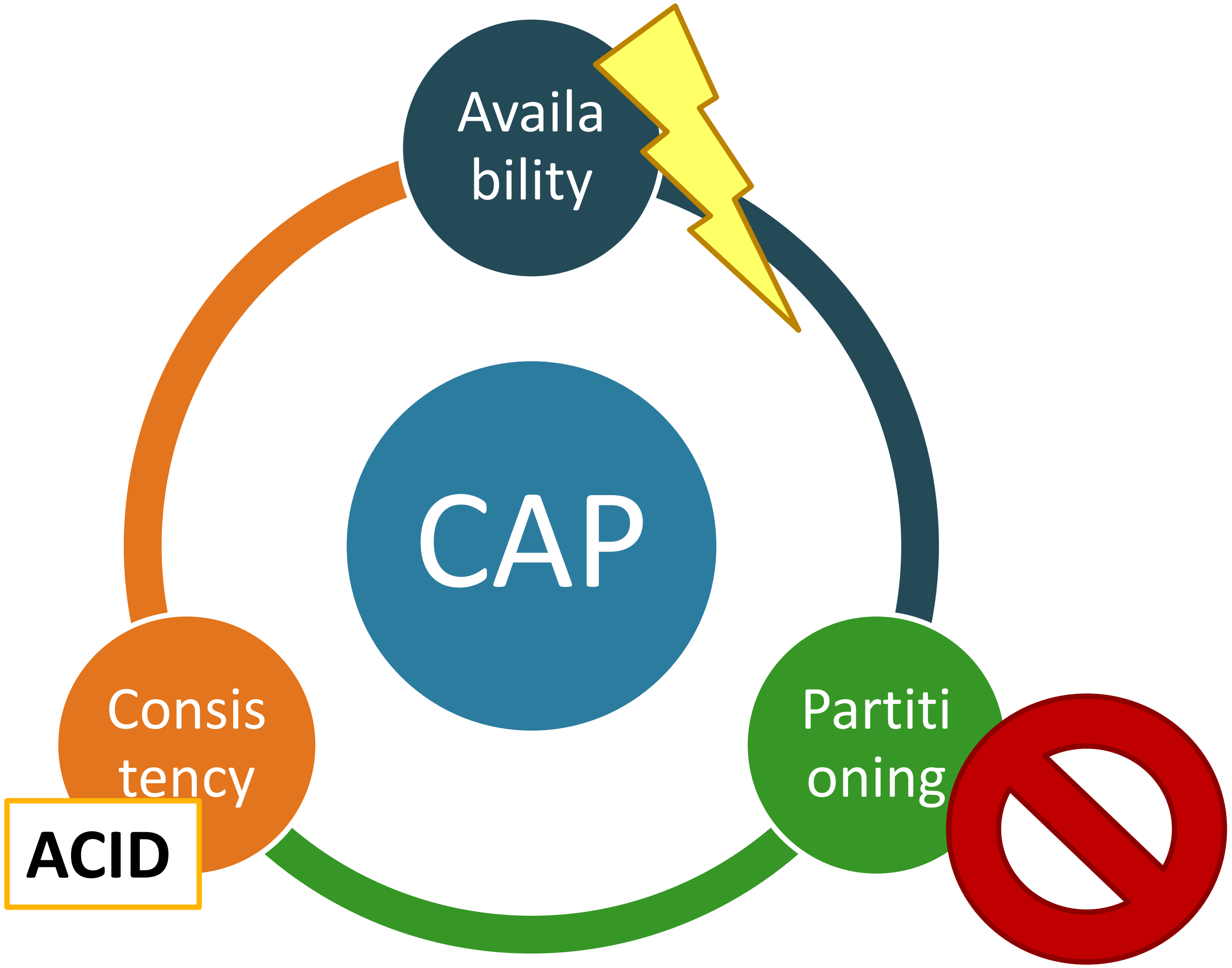
Comments on CAP Properties & Cloud DBs

- Contrary to traditional databases, many cloud databases are not expected to satisfy any **integrity** constraints
- High availability of a service is often characterized by small latency
  - Amazon: 0.1 secs more response time will cost them 1% in sales,
  - Google: 0.5 secs in latency caused traffic to drop by 20%
- It is hard to draw a clear border between availability and network partitions
  - Network partitions may induce delays in operations of a service, and hence a greater latency

Comments on CAP Properties

- Imagine you replicate your database for high **availability**
  - What happens if the network crashes (**partition**) between them?
- Imagine you make it more resilient by not requiring immediate updates in the case of a **partition**
  - Data might become **inconsistent**
- Therefore, it is impossible to build a system that has:
  - Strong **consistency**
  - High **availability**
  - **Partition** tolerance

Traditional SQL databases



CAP-based tradeoffs

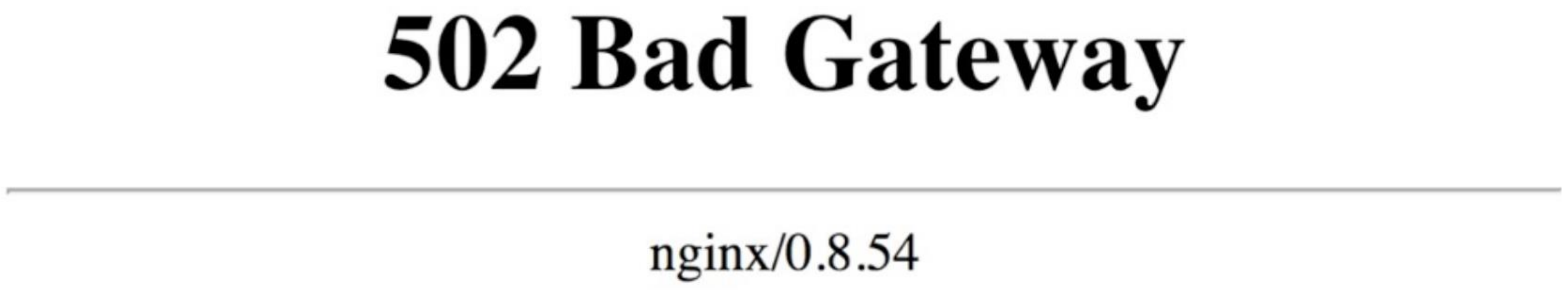
- SQL databases provide **strong consistency** and **availability** at the expense of **partition tolerance**
- Different NoSQL databases make different tradeoffs
- Dynamo systems provide **availability** and **partition tolerance** at the expense of **strong consistency**

Amazon Dynamo

- Amazon analyzed their visitors purchasing habits
- Determined that `High latency == lost revenue`
  - 1% loss every 100ms
- Researched how to provide low latency & high availability for their data
  - Paper in 2007
- Rise of eventual consistency

What about foregoing availability?

What about foregoing availability?



502 Bad Gateway

nginx/0.8.54

Contents

What is (Strong) Consistency?

- Strong Consistency: Replicas update linearly in the same total order
  - No apparent conflicts
  - This is what we are used to as developers
  - ACID databases (SQL) are Strongly Consistent
- Distributed + ACID = “**Consensus**”
  - Lots of messages to synchronize replica state
  - Well known limitations

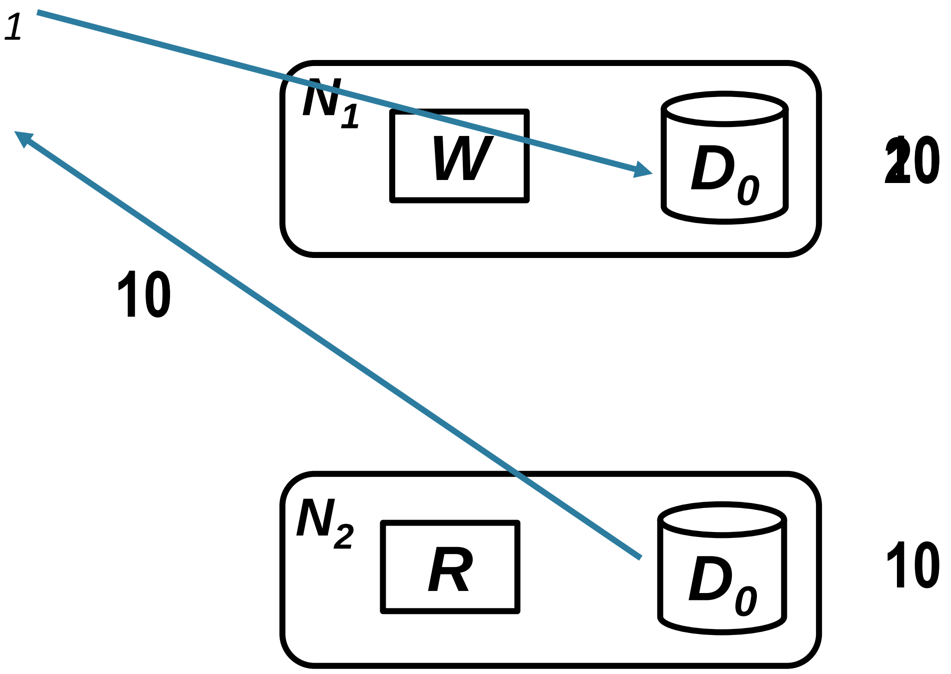
Eventual Consistency

- “Replicas Update in the background. May not converge to the same total order”
- Update is accepted by local node
- Local node propagates update to replica nodes
- No synchronization phase
  - Eventually, all replicas are updated
  - Data can diverge! Arbitrate? Rollback?
- Simply…

  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

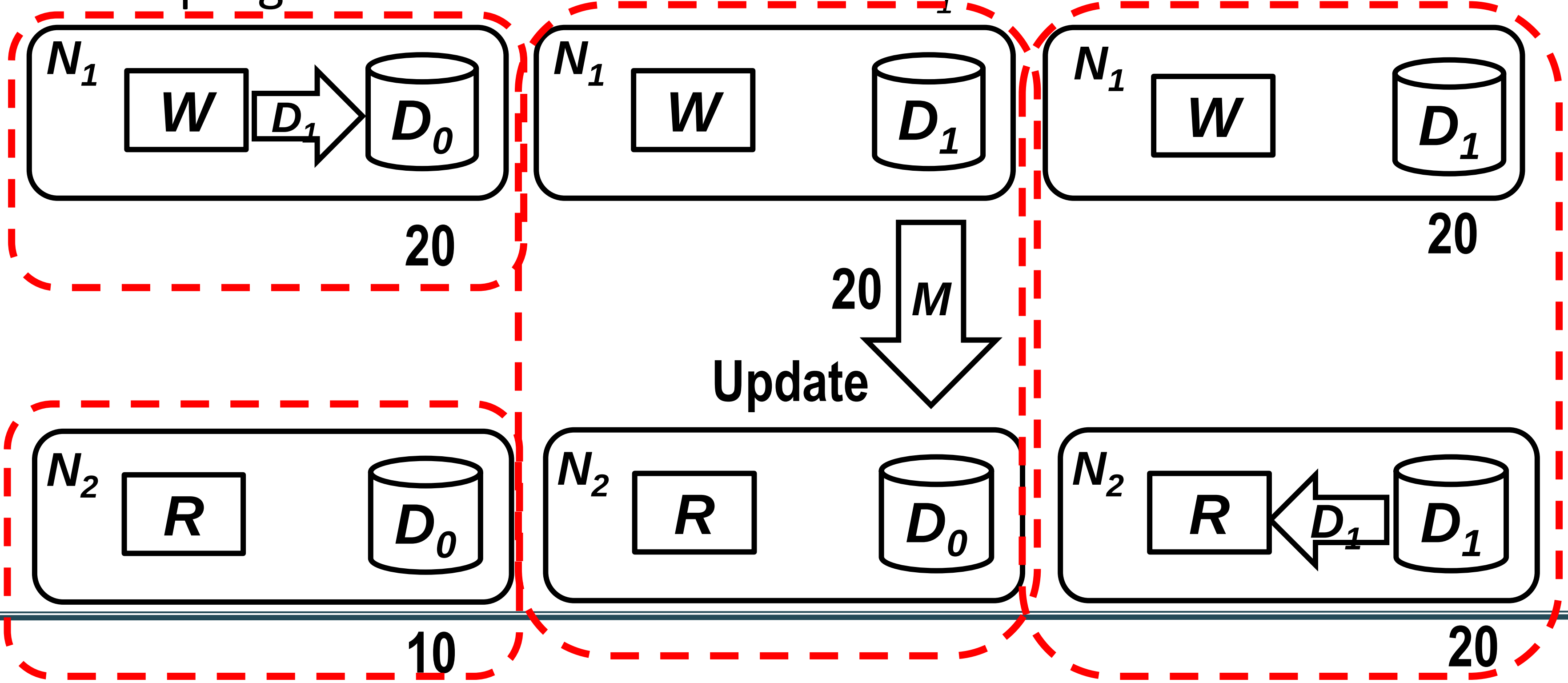
Consistency Example according to CAP

- A Highly Available Partitioned system can be inconsistent
- Let us consider two nodes N_1 and N_2
  - Both nodes contain the same copy of data D having a value D_0
  - A program W (write) runs on N_1
  - A program R (read) runs on N_2



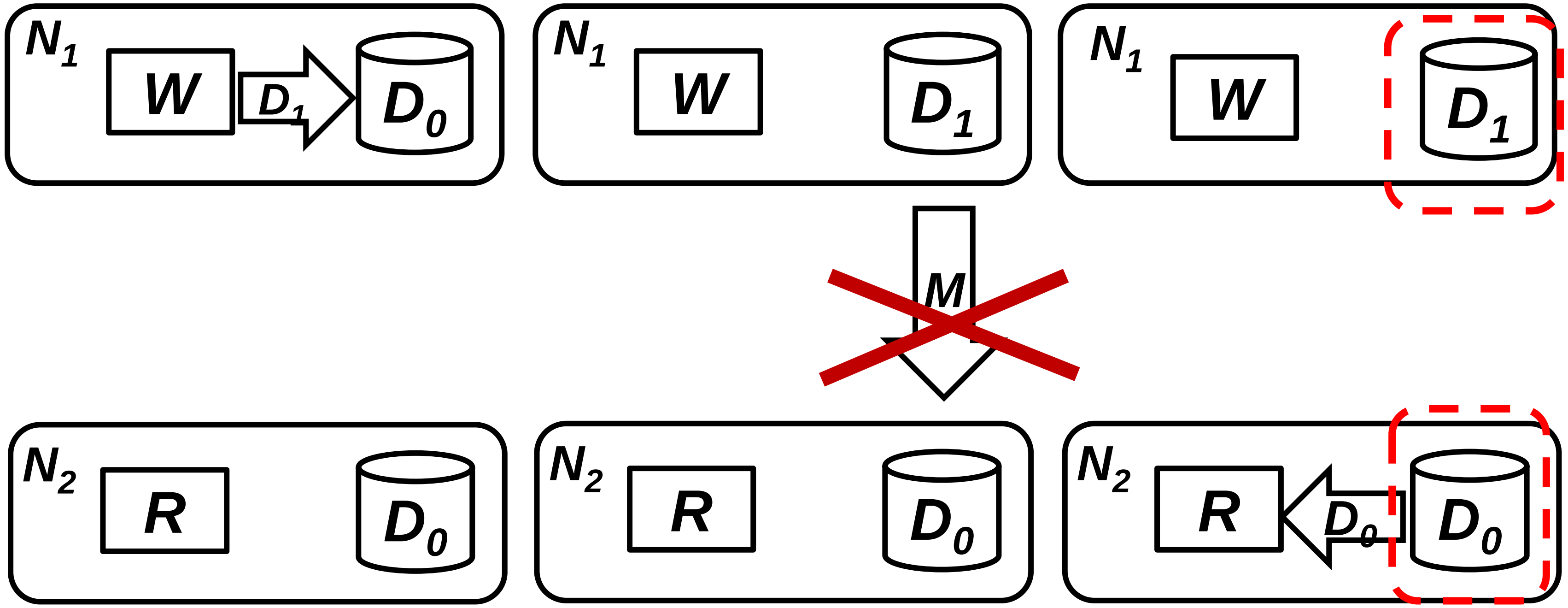
Sequence I: Apparent Consistency

- In a regular situation:
  - The program *W* writes a new value of *D*, say D_1 on N_1
  - Then, N_1 sends a message *M* to N_2 to update the copy of *D* to D_1,
  - The program *R* reads *D* and returns D_1



Sequence 2: Inconsistency appears

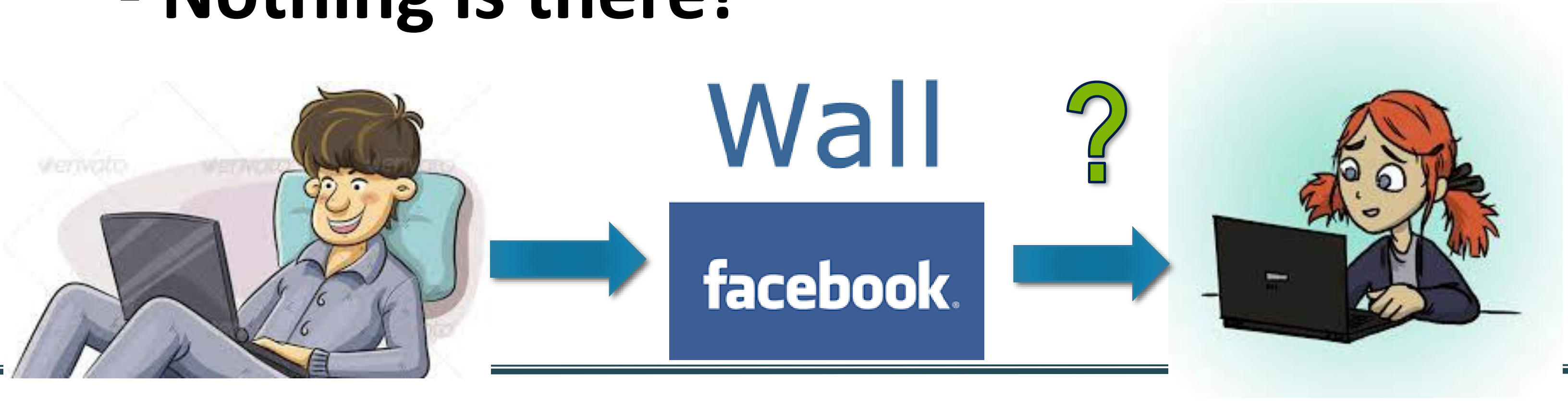
- Assume now the network partitions and the message from N_1 to N_2 is not delivered
  - The program *R* reads *D* and returns D_0



Eventual Consistency – A Facebook Example

- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:

  - **Nothing is there!**



Eventual Consistency – A Facebook Example

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back:
  - **She finds the story Bob shared with her!**

Eventual Consistency – A Facebook Example

- Reason: it is possible because Facebook uses an **eventual consistent model**
- Why Facebook chooses eventual consistent model over the strong consistent one?
  - Facebook has more than 2.7 billion active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to **reduce the load and improve availability**

Eventual Consistency – A Dropbox Example

- Dropbox enabled immediate consistency via synchronization in many cases.
- However, what happens in case of a network partition?

Eventual Consistency – A Dropbox Example

- Let’s do a simple experiment here:
  - Open a file in your drop box
  - Disable your network connection (e.g., WiFi, 4G)
  - Try to edit the file in the drop box: can you do that?
  - Re-enable your network connection: what happens to your dropbox folder?

Eventual Consistency – A Dropbox Example

- Dropbox embraces eventual consistency:
  - Immediate consistency is impossible in case of a network partition
  - Users will feel bad if their word documents freeze each time they hit Ctrl+S, simply due to the large latency to update all devices across WAN
  - Dropbox is oriented to **personal syncing**, not on collaboration, so it is not a real limitation.

Eventual Consistency – An ATM Example

- In design of automated teller machine (ATM):
  - Strong consistency appear to be a nature choice
  - However, in practice, **A beats C**
  - Higher availability means **higher revenue**
  - ATM will allow you to withdraw money *even if the machine is partitioned from the network*
  - However, it puts **a limit** on the amount of withdraw (e.g., $200)
  - The bank might also charge you a fee when an overdraft happens

Strong Consistency

- The definition of eventual consistency invokes the definition of its counterpart – **strong consistency**
  - All read operations must return data from the latest completed write operation, regardless of which replica the operation went to
  - It is very hard to achieve the strict consistency in a highly available replicated database implemented on commodity servers

Contents

- Cloud Databases
- CAP Theorem
- CAP Consistency
- **Data replication**
- Cloud DB Services

Data Partitioning and Replication

- There are three reasons for storing a database on a multiple machines (nodes)
  - The amount of data exceeds the capacity of a single machine,
  - To allow scaling for load balancing, and
  - To ensure reliability and availability by replication
- NoSQL DBs tend to maximise availability when partitioning
- There are a number of techniques to achieve data partitioning and replication:
  - Memory caches,
  - Separating reads from writes,
  - Clustering, sharding

Caches

- ***Memory Caches*** can be seen as transient, partly partitioned and replicated in-memory databases, since they replicate most frequently requested parts of a database to main memories of a number of servers
  - Fast response to clients
  - Off-loading database servers

Separating Reads from Writes

- **Separating reads from writes**:
  - One or more servers are dedicated to writes (master(s)),
  - A number of replica servers are dedicated to satisfy reads (slaves),
  - The master replicates the updates to slaves
    - If the master crashes before completing replication to at least one slave, the write operation is lost
    - Otherwise the most up to date slave undertakes the role of a master

Clustering

- ***High-availability clusters*** (also known as **HA clusters** or **failover clusters**) are groups of computers that support server applications
- They operate by harnessing redundant computers in groups to provide a continued service when system components fail
- When a HA cluster detects a hardware/software fault, it immediately restarts the application on another system without requiring administrative intervention, a process known as ***failover***
- HA cluster use redundancy to eliminate **single points of failure** (SPOF)

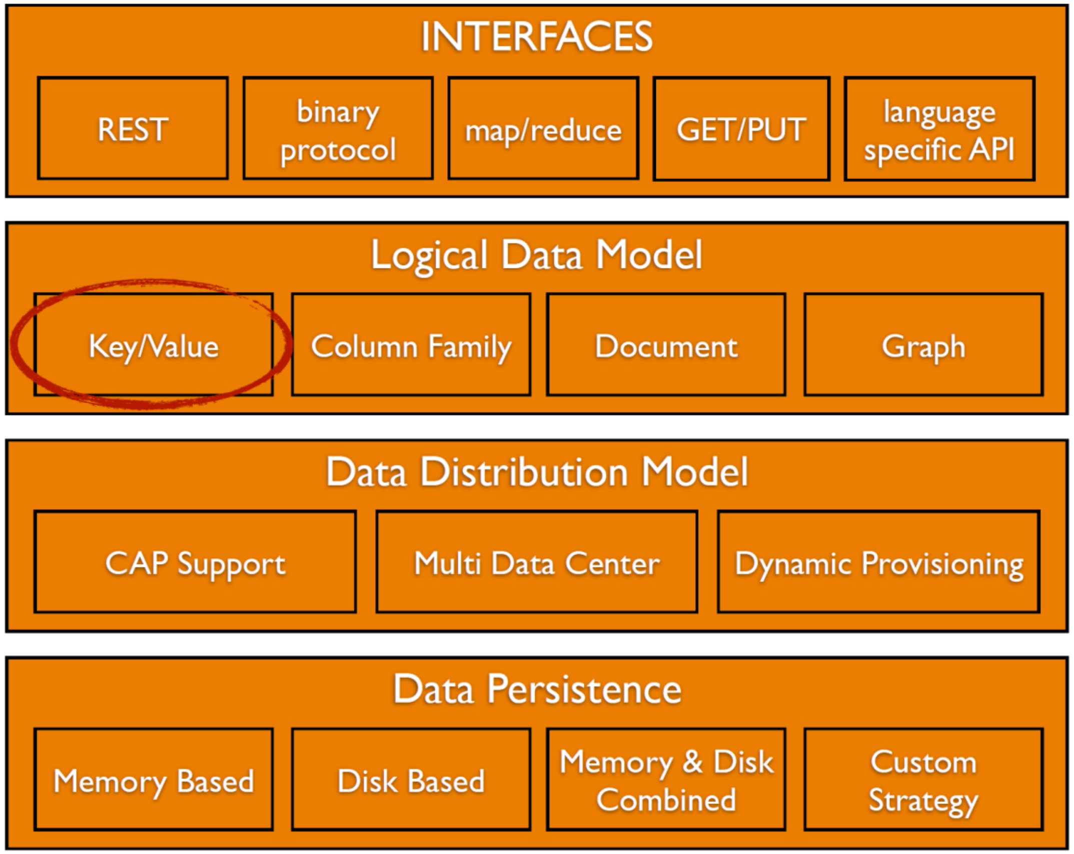
Sharding

- **Data Sharding** is data partitioning in such a way that:
  - Data typically requested and updated ***together*** reside on the same node, and
  - The work load and storage volume are roughly evenly distributed among servers
- Data sharding is not an appropriate technique for partitioning relational databases, due to their normalized structure
- Data sharding is very popular with cloud non relational databases, since they are designed having partitioning in mind

Contents

- Cloud Databases
- CAP Theorem
- CAP Consistency
- Data replication
- **Cloud DB Services**

NoSQL Databases landscape

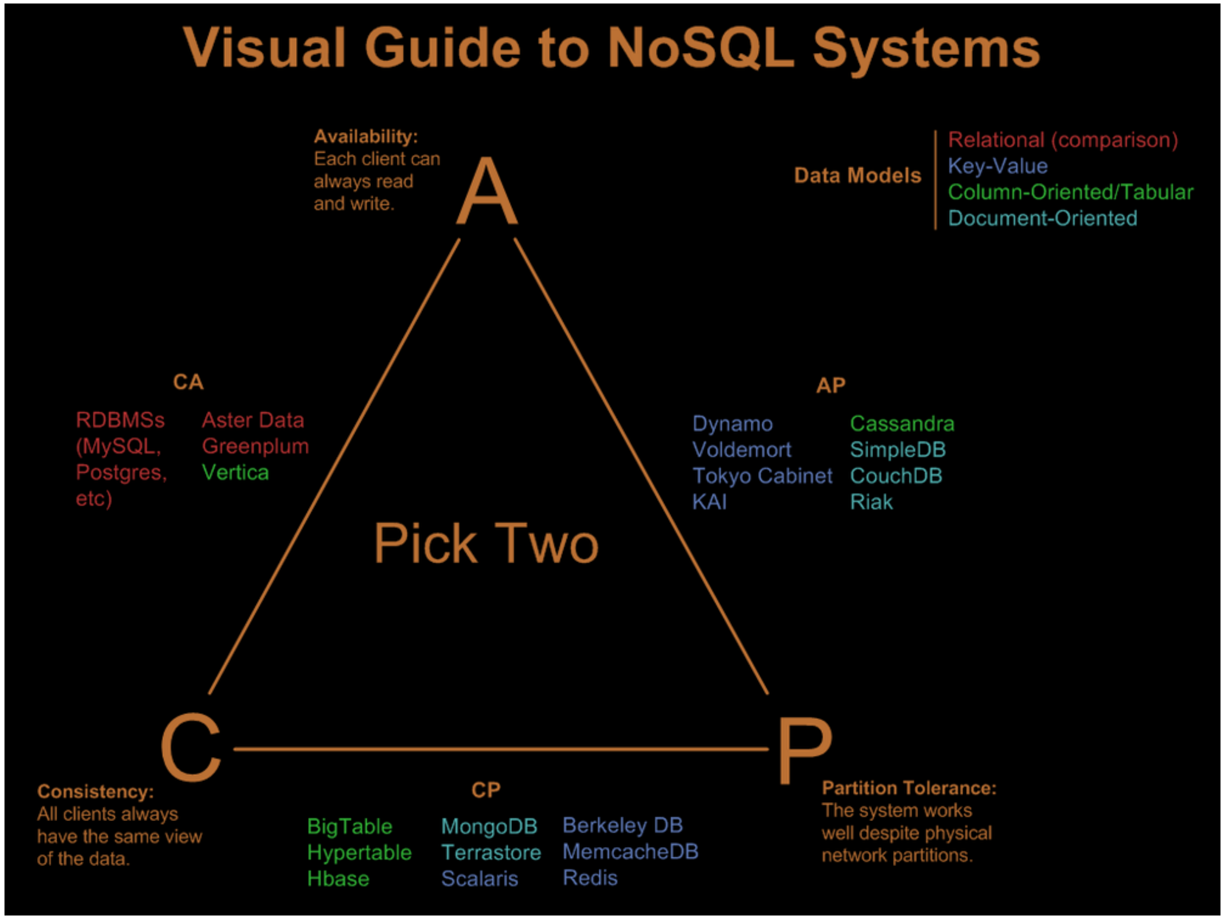


NoSQL database types and examples

- **Key/value Databases**
  - These manage a simple value or row, indexed by a key
  - e.g. Voldemort, Vertica, memcached
- **Big table Databases**
  - “a sparse, distributed, persistent multidimensional sorted map”
  - e.g. Google BigTable, Azure Table Storage, Amazon SimpleDB, Apache Cassandra
- **Document Databases**
  - Multi-field documents (or objects) with JSON access
  - e.g. MongoDB, RavenDB (.NET specific), CouchDB
- **Graph Databases**
  - Manage nodes, edges, and properties
  - e.g. Neo4j, sones

Implications of CAP theorem

- Choose a database whose priorities match the application



http://blog.nahurst.com/visual-guide-to-nosql-systems

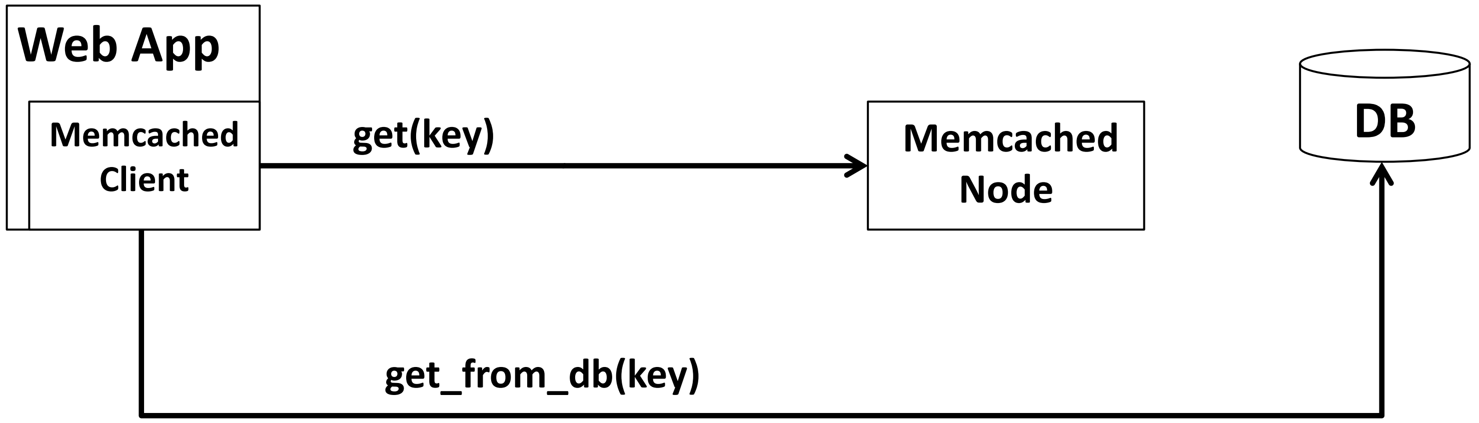
Let’s look at two examples

- **Memcached**: In-memory caching subsystem
- **Cassandra**: Distributed database with replication and tunable consistency

Memcached

- High performance distributed in-memory caching service that manages "objects"
  - Very similar to a Hashtable
- Key-value API has become an accepted standard
  - `Get` and `set` methods
- Goal: attend majority of requests for read-heavy workloads
- Used in Facebook, LinkedIn, Flickr…

Memcached use with standard DB



```
public String getFoo(String fooId)
        String foo = memcached.get(fooId)
        if foo != null return foo
                foo = fetchFooFromDatabase(fooId)
                memcached.set(fooId, foo)
        return foo
        end
```

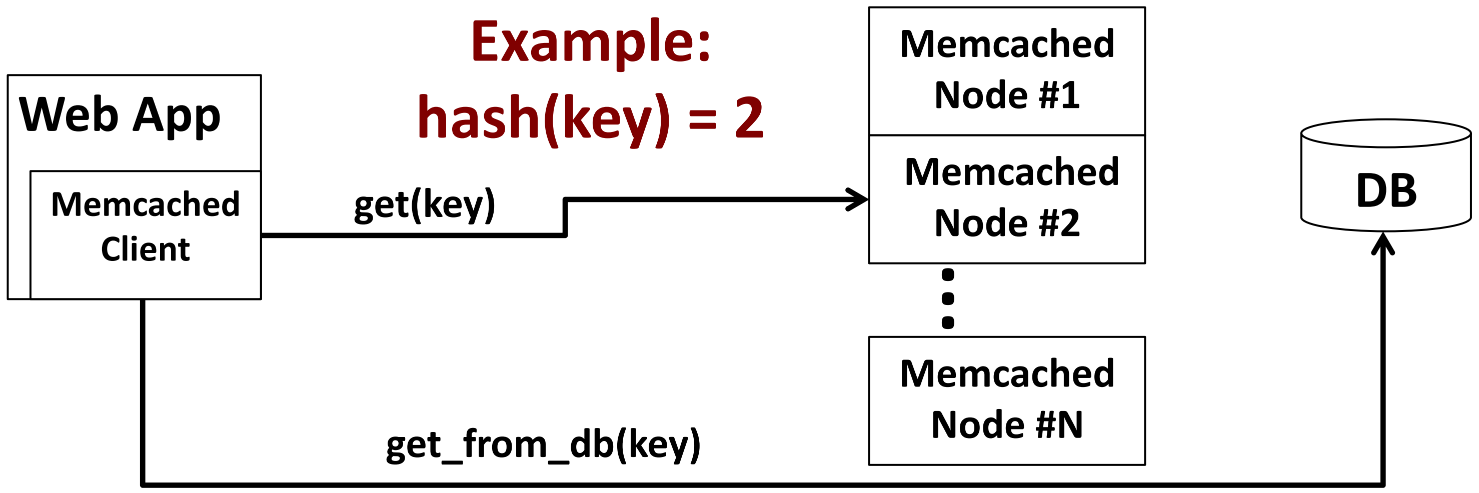
Memcached size

- A Memcached node has limited capacity
  - Set number of keys, each no more than 128 bits
  - Each value no more than 1MB
- If a new key is added to a full Memcached, it takes the place of an old one
  - Key Replacement based on LRU policy
  - Least Recently Used item goes away
  - This way, frequently accessed items are in memcached regardless of capacity

Memcached clusters

- Multiple memcached nodes
  - Each one holds different keys
- Smart Clients/Dumb Servers
  - Simple client hash function maps keys→servers
  - Servers are not aware there are multiple of them

Scaling Horizontally Memcached (II)



Apache Cassandra

- Open Source Distributed Database
  - Based on two influential NoSQL systems: Amazon Dynamo, and Google BigTable
  - Released in 2008
  - Used by Facebook, eBay and many other Companies

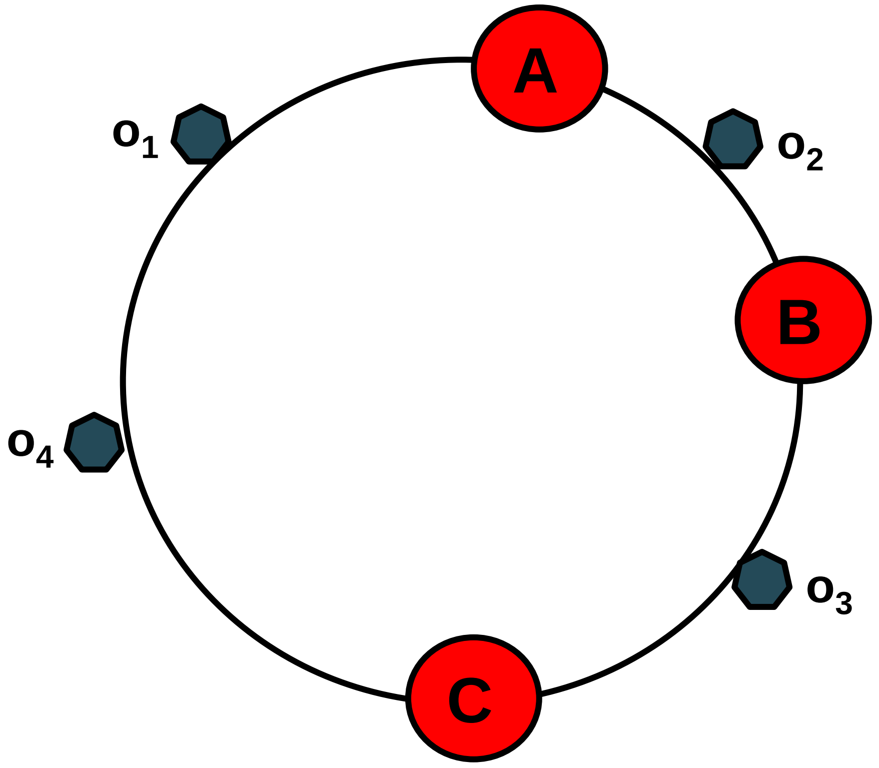
Cassandra features

1. Elastic: Read and write throughput increases linearly as new machines are added
   - High performance for write operations
2. Supports content replication, per node and per datacenter
3. Decentralised: Fault tolerant with **no single point of failure**; no “master” node
4. Data model: Column based, multi dimensional hashtable, no join capabilities
5. Tunable consistency, per operation

Consistent Hashing

- Each database object is mapped to a point on a circle by hashing its key value
- Each available machine (node) is mapped to a point on the same circle
- To find the node to store an object, Cassandra:
  - Hashes the object’s key to a point on the edge of the circle,
  - Walks clockwise around the circle until it encounters the first node
- Consistent hashing is a special type of hashing function that reduces the impact of nodes entering and leaving the ring

Consistent Hashing (Example 1)



Objects o_1 and o_4 are stored on the node A

Object o_2 is stored on the node B

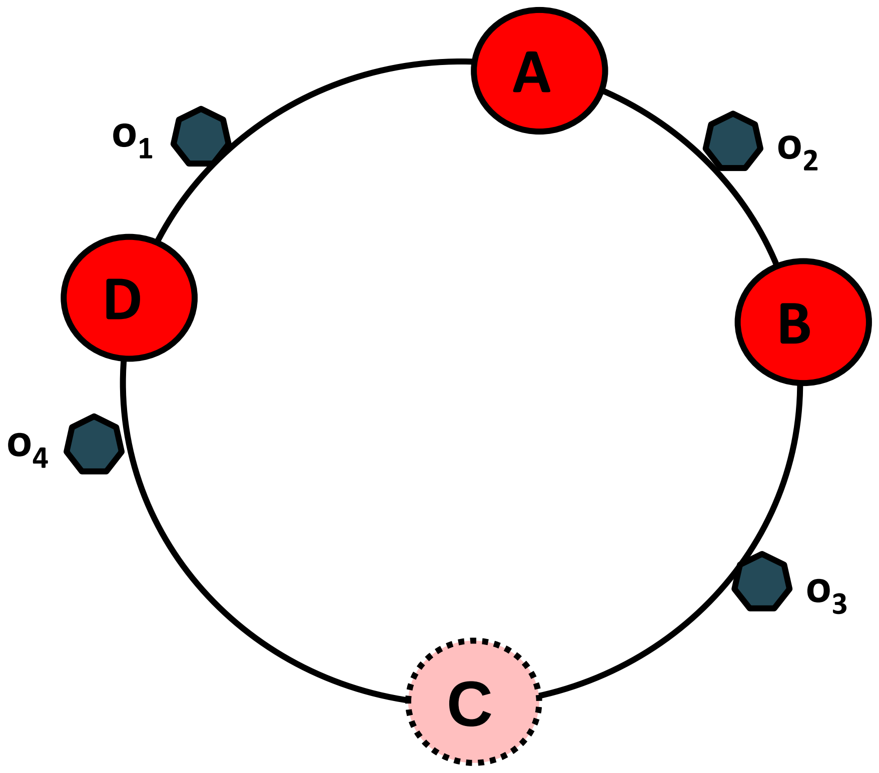
Object o_3 is stored on the node C

Consistent Hashing: node changes

- If a node leaves the network, the node in the clockwise direction stores all new objects that would belong to the failed node
- The existing data objects of the failed node have to be redistributed to remaining nodes
- If a node is added to the network, it is mapped to a point
  - All objects that map between the point of the new node and the first counter clock wise neighbour, map to the new node

Consistent Hashing (Example 2)

The node C has left and the node D has entered the network



Object o_1 is stored on the node A

Object o_2 is stored on the node B

Objects o_3 and o_4 are stored on the node D

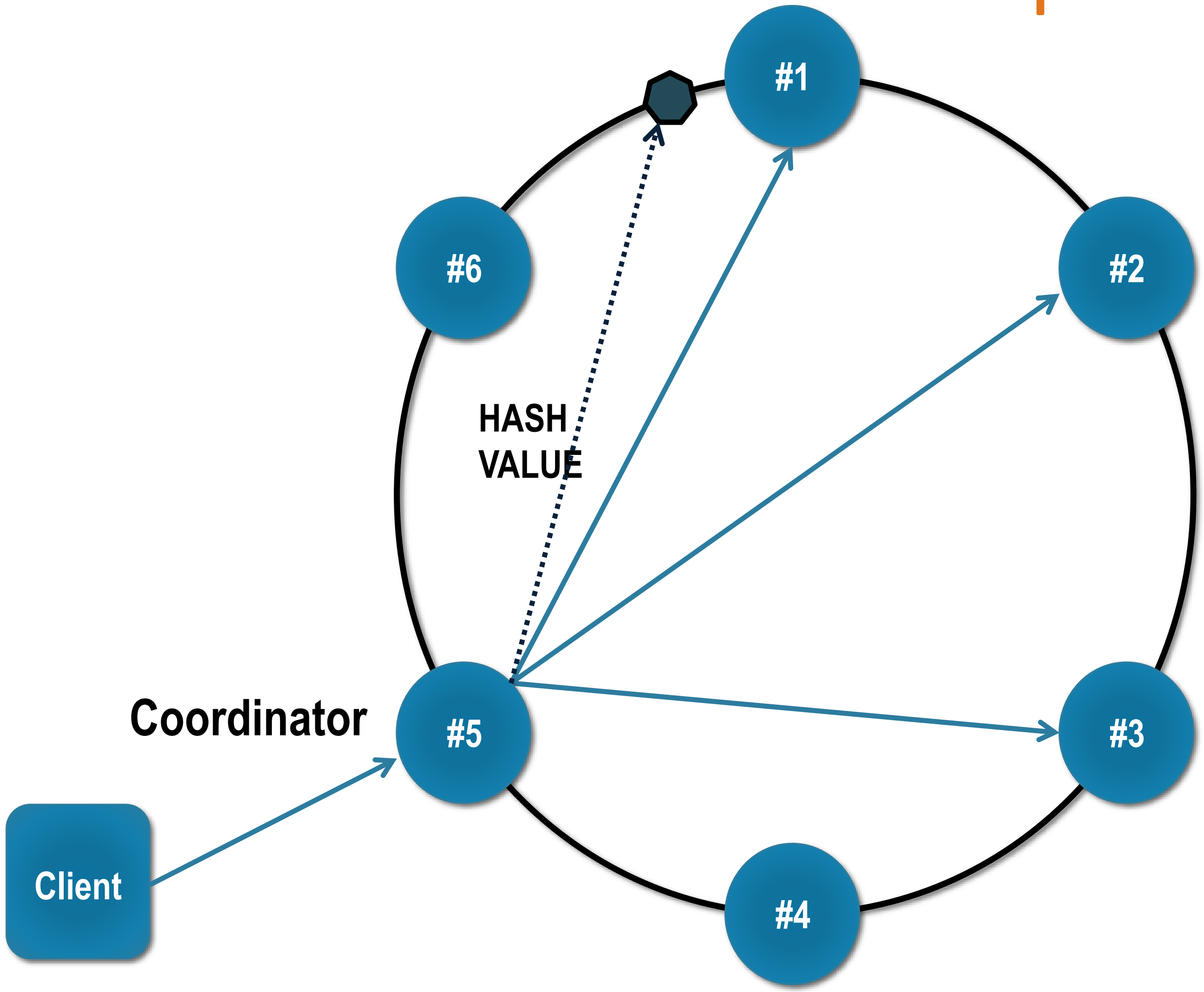
Cassandra Data Replication

- Used to achieve high availability and durability.
- How?
  - **Replication factor:** determines how many copies of your data exists.
  - **Each data item:** is replicated at N hosts (N=replication factor).
  - **Coordinator node:** in charge of the replication of the data items that fall within its range.
  - **Consistency level:** refers to how much up-to-date and synchronized a row of Cassandra is in all its replicas, e.g., quorum→ replication_factor/2 + 1.
  - **Various replication policies:** Rack Unaware, Rack Aware, and Datacentre Aware.
- Each row is replicated across multiple datacentres which are connected through high speed network links.

Cassandra Data Replication

- Keys have associated a replication factor (number of replicas over the network)
- The nodes in charge of storing the replicas are the ones immediately to the right of the node that has to store the key copy based on the consistent hash function

Cassandra: Access to Data Replicas



Replication Factor = 3

Consistency Models: Write Operations

- How many replicas need to reply for the operation to become a success?

Level	Description
ANY	One node, including hinted handoff
ONE	One node
QUORUM	N/2 + 1 replicas
LOCAL_QUORUM	N/2 + 1 replicas in local data centre
EACH_QUORUM	N/2 + 1 replicas in each data centre
ALL	All replicas

http://wiki.apache.org/cassandra/API#Write

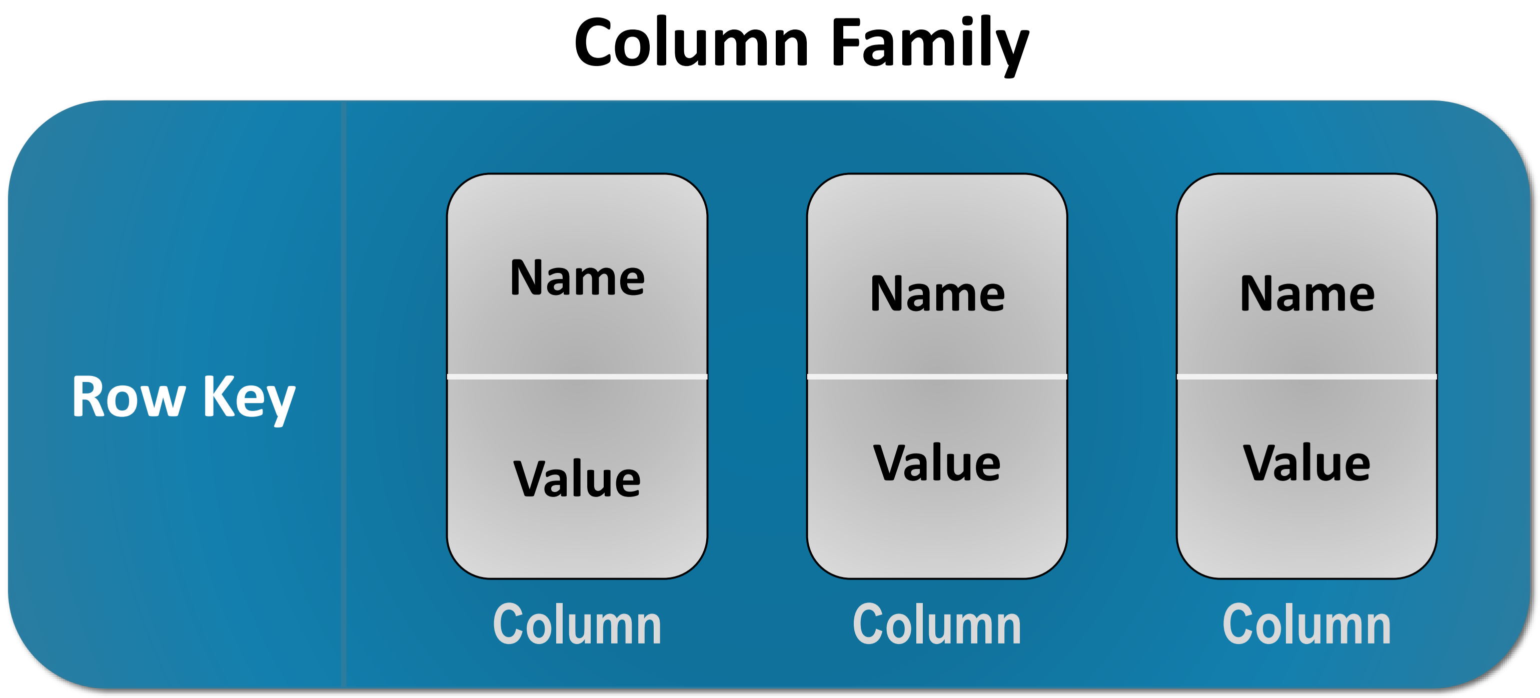
Consistency Models: Read Operations

- How many replicas need to reply for the operation to become a success?

Level	Description
ONE	1st Response
QUORUM	N/2 + 1 replicas
LOCAL_QUORUM	N/2 + 1 replicas in local data centre
EACH_QUORUM	N/2 + 1 replicas in each data centre
ALL	All replicas

http://wiki.apache.org/cassandra/API#Read

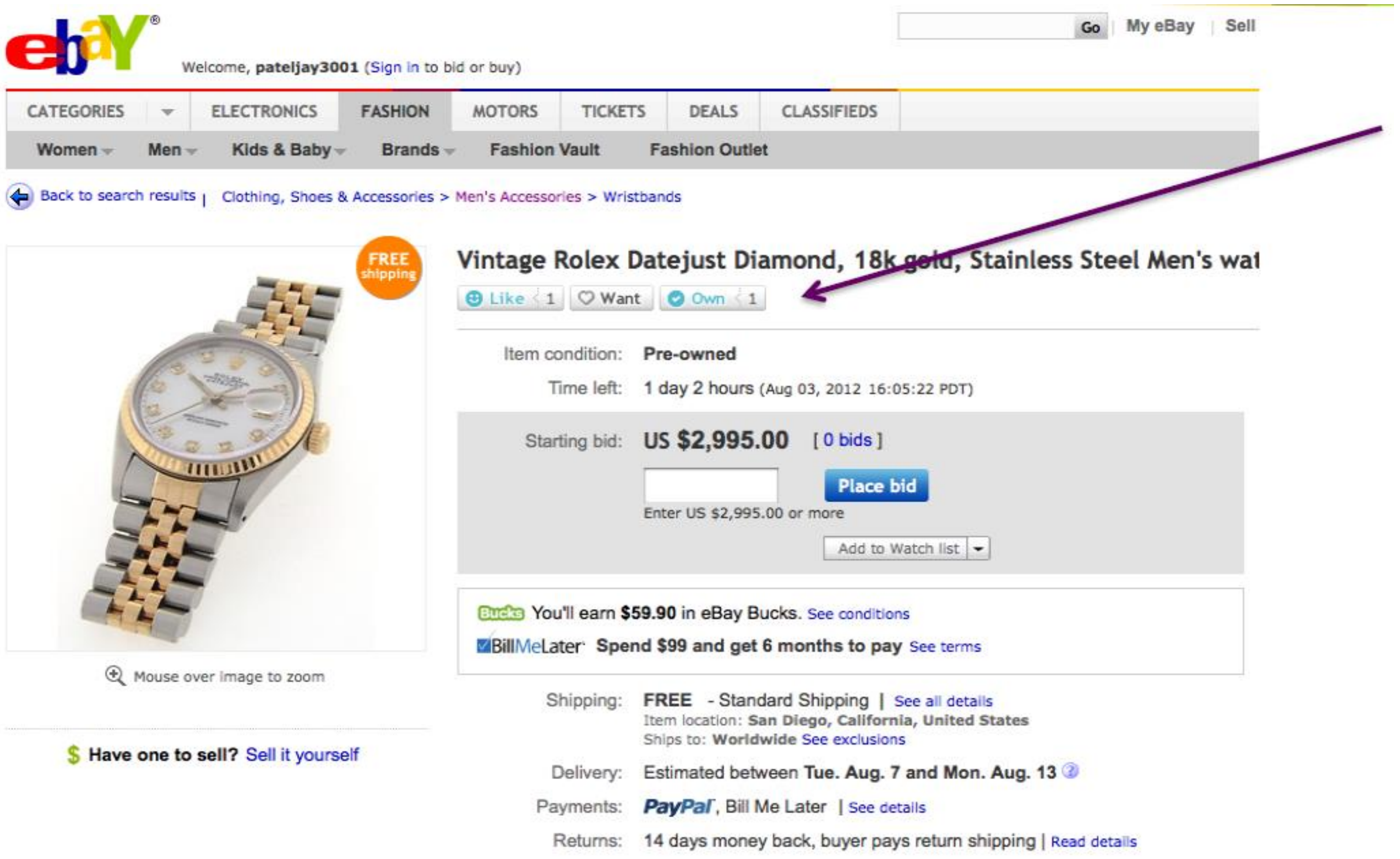
Cassandra Data Model (from BigTable)



Cassandra data model

- Effectively a 3D hashtable
- A table contains a set of Column Family elements, each one has a key(rowName), and a value
- The value of a Column Family is in turn a collection of key/value elements, called column
- Different Column Family elements can (will) have different values
- No joins, limited query capabilities

Sample: eBay data modeling of product likes


Welcome, pateljay3001 (Sign in to bid or buy)

Go | My eBay | Sell

Back to search results | Clothing, Shoes & Accessories > Men's Accessories > Wristbands

Vintage Rolex Datejust Diamond, 18k gold, Stainless Steel Men's wat

Item condition: Pre-owned

Time left: 1 day 2 hours (Aug 03, 2012 16:05:22 PDT)

Starting bid: US $2,995.00 [0 bids]

Enter US $2,995.00 or more

Bucks You'll earn $59.90 in eBay Bucks. See conditions

BillMeLater Spend $99 and get 6 months to pay See terms

Mouse over image to zoom

$ Have one to sell? Sell it yourself

Shipping: FREE - Standard Shipping | See all details
Item location: San Diego, California, United States
Ships to: Worldwide See exclusions

Delivery: Estimated between Tue. Aug. 7 and Mon. Aug. 13

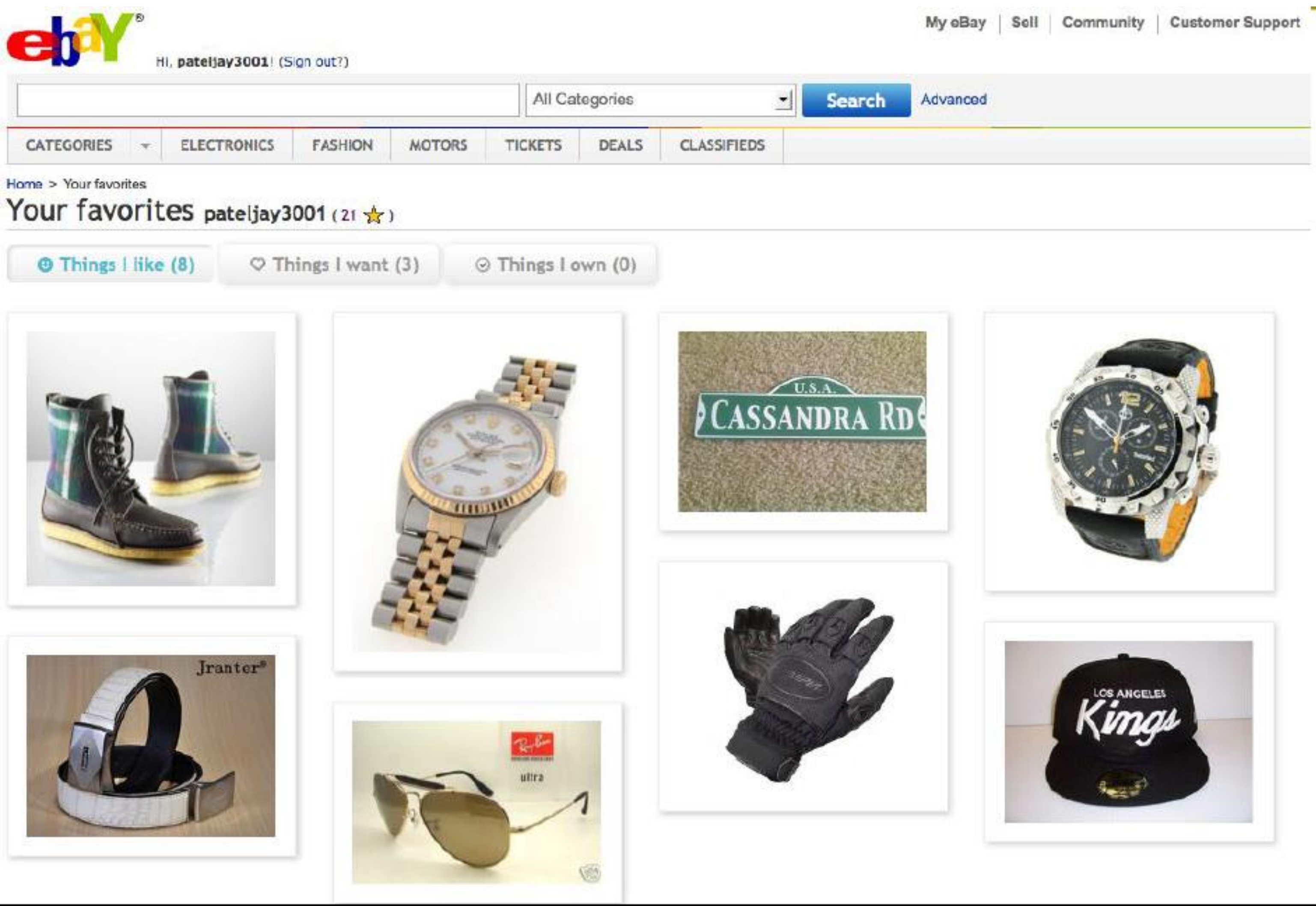
Payments: PayPal, Bill Me Later | See details

Returns: 14 days money back, buyer pays return shipping | Read details

Managed by Cassandra

Queen Mary University of London

Sample: eBay, manage your favorites



Your favorites pateljay3001 (21 ★)

Things I like (8) | Things I want (3) | Things I own (0)

Whole content from the page delivered by Cassandra

eBay’s data model in Cassandra

ItemCount

itemid1	“likeCount”	“wantCount”	“ownCount”
	2000	5000	1000
⋮			

- Get signal count for a item

UserCount

userid1	“likeCount”	“wantCount”	“ownCount”
	20	100	50
⋮			

- Get signal count for a user

ItemLike

itemid1	userid1	userid2	userid3	…
	timeuuid1	timeuuid2	timeuuid3	
⋮				

- Check if user has already liked a item or not.

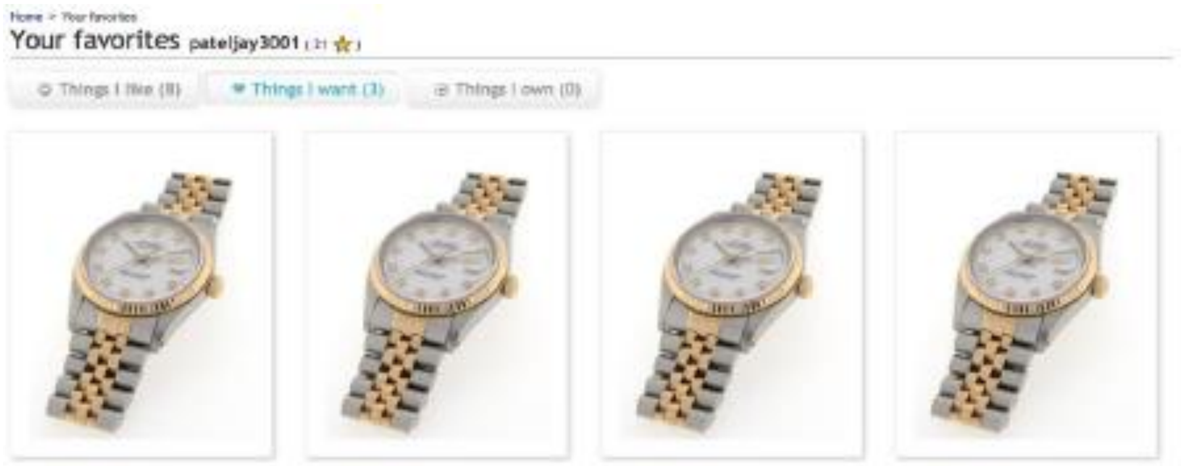
UserLike

Composite column name/key

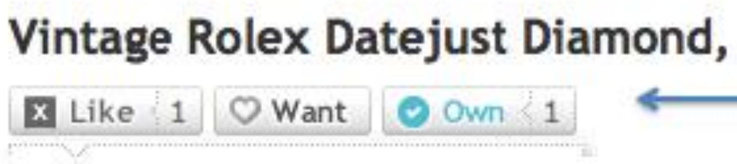
userid1	timeuuid1 | itemid1	timeuuid2 | itemid2	…
	-null-	-null-	
⋮			

- Get user’s liked items in chronological order.

Eventual Consistency problems!



Duplicates!



Oh, toggle button!

Signal --> De-signal --> Signal...