CLOUD COMPUTING

STORAGE

Lorenzo Leonini
lorenzo.leonini@unine.ch

Objectives

- Present storage solutions for IaaS environments
- Relational databases
  - ACID
  - How to scale
- CAP Theorem
- NoSQL databases

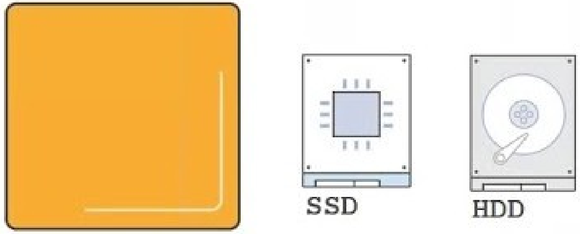
Block storage

- Virtual disks available for mounting in a VM instance
  - Visible as block devices, similarly to a local hard drive or SSD
  - Guest OS must mount storage using a file system
- Temporary and local block storage
  - Attached to the host, or available through local SAN (Storage Area Network)
  - Content lost when VM shut down
- Persistent block storage
  - Kept across VM shutdown/restart cycles
  - FS integrity requires clean shutdown

EC2 Instance Store *vs* EBS

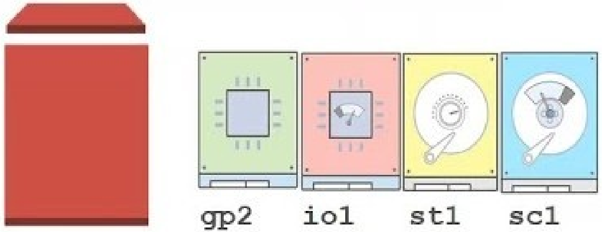
EC2 Instance Store

- Local to instance
- Non-persistent data store
- Data not replicated (by default)
- No snapshot support
- SSD or HDD



Elastic Block Store

- Persistent block storage volumes
- 99.999% availability
- Automatically replicated within its Availability Zone (AZ)
- Point-in-time snapshot support
- Modify volume type as needs change
- SSD or HDD
- Auto recovery



Storage for applications

- Block storage used for operating system, libraries, application binaries, container instances
- File system not adapted for storing application data
  - Local to one VM instance, not meant for sharing
  - May not survive the failure or shutdown of the instance
  - Difficult to handle partitioning of application data over multiple servers
- Use a database
  - As a (set of) container(s) deployed by the application
  - Using a managed service from the platform provider

Types of databases

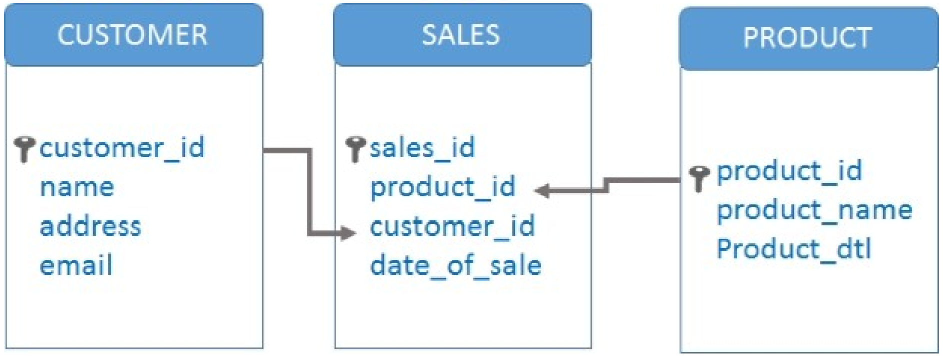
- Relational (SQL)
  - MySQL, PostgreSQL, Oracle, SQL Server, …
  - **RDBMS** = Relational DB Management System
- NoSQL = Not only SQL
  - Column-oriented
    - Cassandra, BigTable, HBase, ...
  - Document-oriented
    - MongoDB, Couchbase, DynamoDB, ...
  - Key/Value
    - Redis, Riak, DynamoDB, etcd, ...
  - Graph
    - Neo4J, …

Relational databases

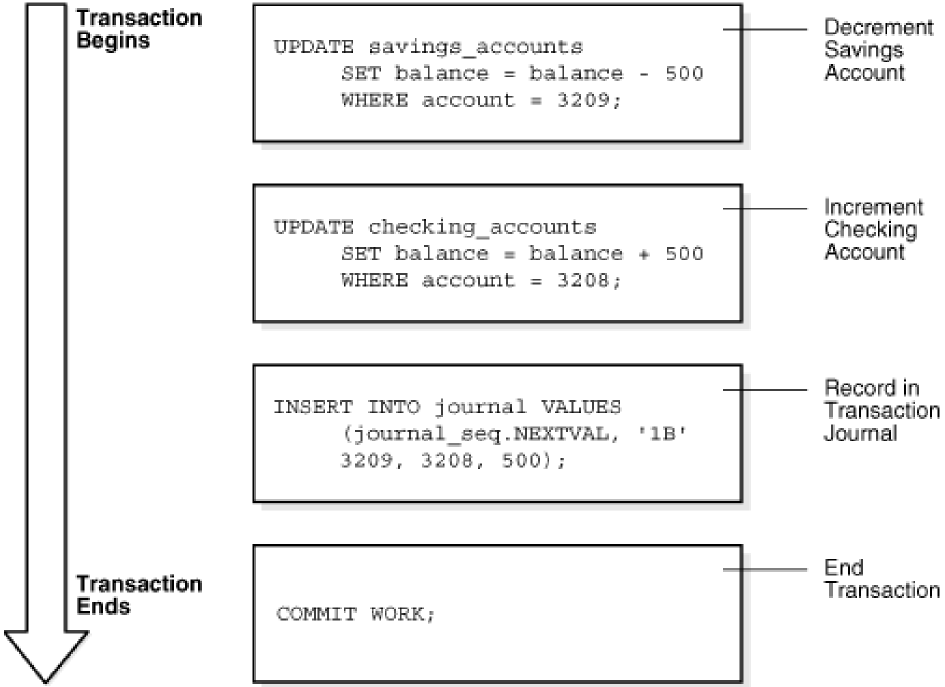
- Available in all cloud platforms
  - Amazon’s Aurora and RDS, …
- Relational databases are well understood by developers and offer a rich interface
  - Well-known and standard querying language (SQL)
  - Support transactions with ACID semantics
    - Atomicity, Consistency, Isolation, Durability
- Complex queries on structured data
  - Ability to use join queries between relations
  - Highly optimized query engines

SQL primer (1)

- Primary keys and their pointers
  - Relational
  - Used to JOIN between tables



- Transactions



SQL primer (2) - Indexes

- Datastructures to improve retrieval speed
  - N = number of records
  - V = number of **different** values for N
    - V <= N
  - Typical structures
    - Balanced tree
    - List of pointers to all records having a specific value
  - Without index
    - Read: O(N)
    - Write: O(1)
  - With index
    - Read: O(log N)
    - Write: O(log N)
    - Additional space: O(N)

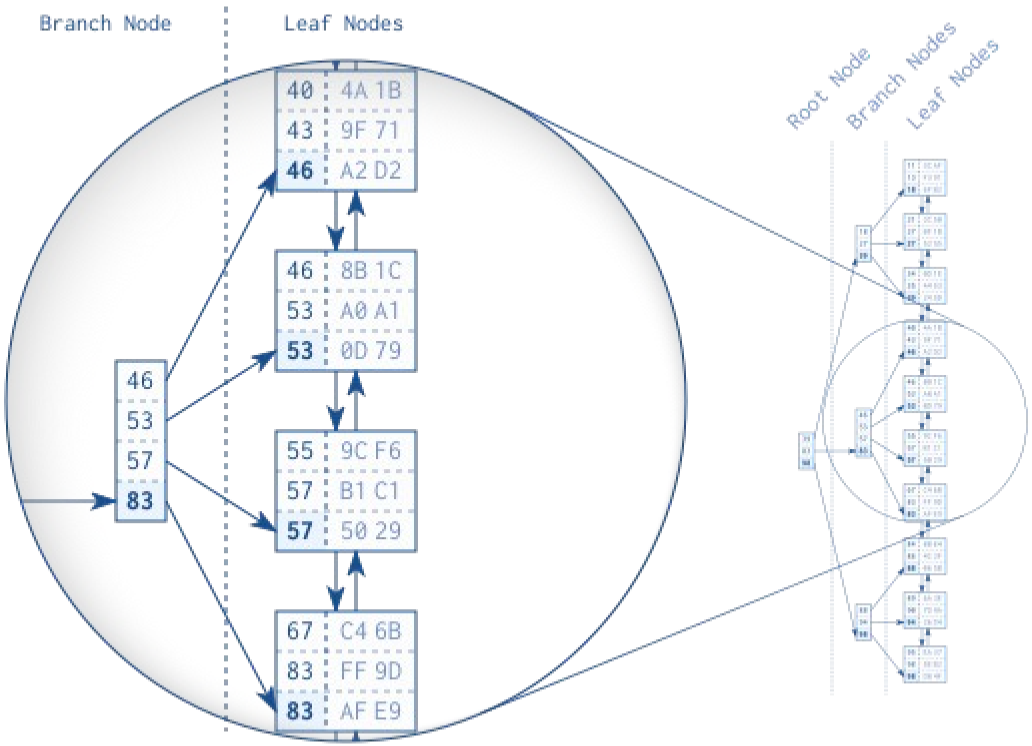


Image + explanations: https://use-the-index-luke.com/

ACID Transactions

- **Atomic**
  - A transaction is treated as a single unit: **succeeds completely or fails completely**
- **Consistent**
  - A transaction can only bring the database from one valid state to another
  - Maintaining database invariants: rules, constraints, triggers, …
- **Isolation**
  - Concurrent execution of transactions leaves the database in the same state that would have been obtained in transactions where executed **sequentially**
- **Durable**
  - Once a transaction has been committed, it will remain commited even in case of system failure

ACID - Isolation

- Transaction property often **relaxed** to improve performances, 4 isolation levels provided in the SQL standard
  - **Serializable**
    - Highest isolation level, the one described in **ACID**
    - If write collision, only one transaction authorized to commit
  - **Repeatable reads**
    - A transaction should always read the same value for the same data
    - Default in MySQL
  - **Read commited**
    - Can only read values that have been commited
    - But can be non repeatable
  - **Read uncommited**
    - Dirty reads

ACID - Isolation demo

- Initial value of A = 200
  - We have basically 2 "parallel" transactions on the same key A
    - 1) A = A * 2
    - 2) A = A - 50
  - Transactions can start in different orders but execution will often overlap
  - In the middle of each transaction we also write new "intermediatea" values for A:
    - 333 or 444, **but these values are never commited** !
  - What can be the final value(s) of A running this code ?
    - A = (200 * 2) - 50 = 350
    - A = (200- 50) * 2 = 300
    - Something else ?

Transaction/Thread 1

```
BEGIN
$v = SELECT v FROM table WHERE k = A

// some perturbation...
UPDATE table WHERE k = A SET v = 333
(sleep)

UPDATE table WHERE k = A SET v = $v - 50
COMMIT
```

Transaction/Thread 2

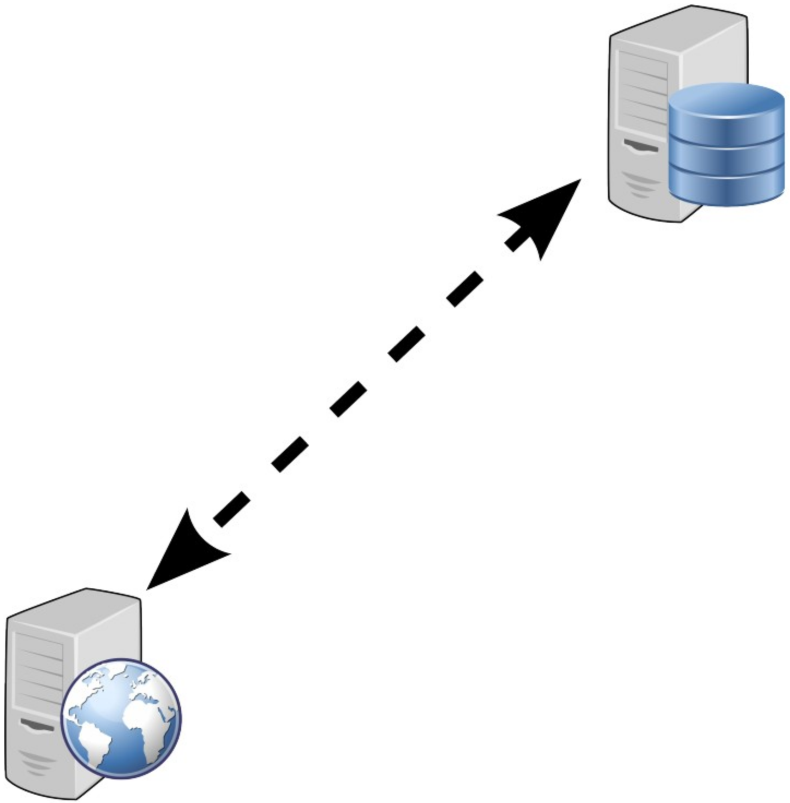
```
BEGIN
$v = SELECT v FROM table WHERE k = A

// some perturbation...
UPDATE table WHERE k = A SET v = 444
(sleep)

UPDATE table WHERE k = A SET v = $v * 2
COMMIT
```

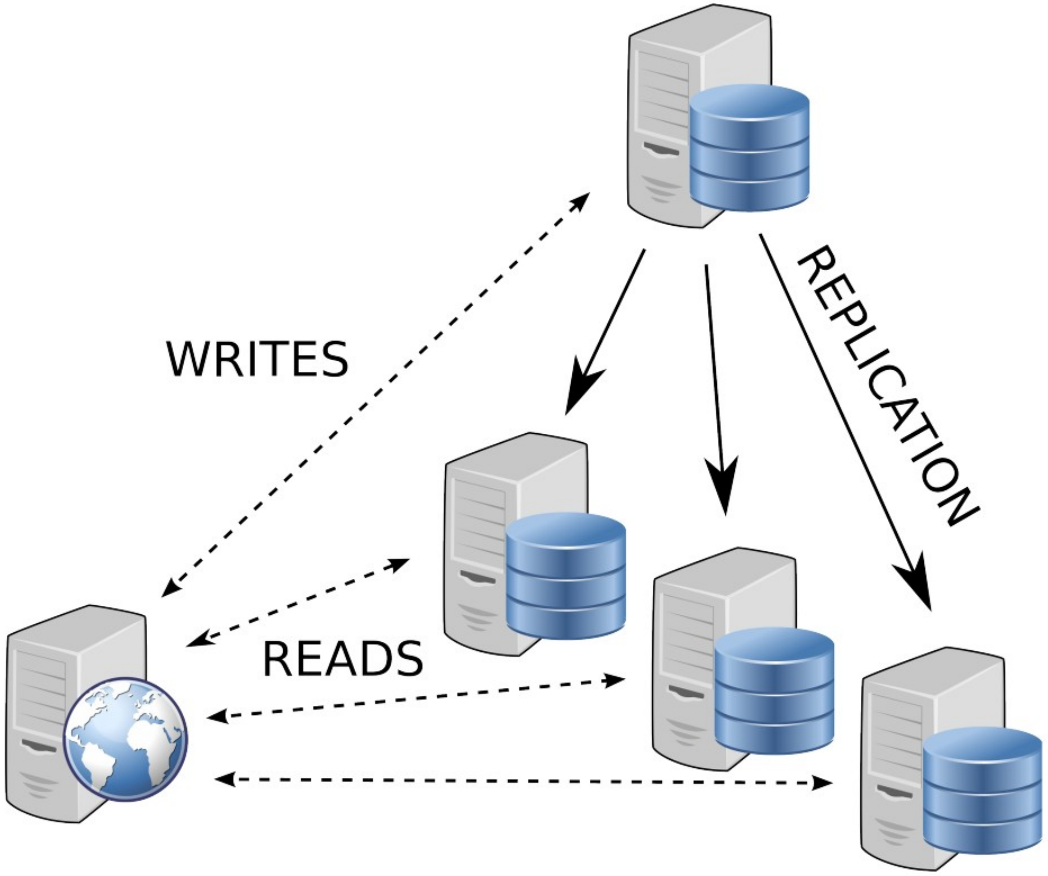
Scaling RDBMS (1)

- Starting with a single server
- Read-write requests starting to be too intensive
- What to do ?



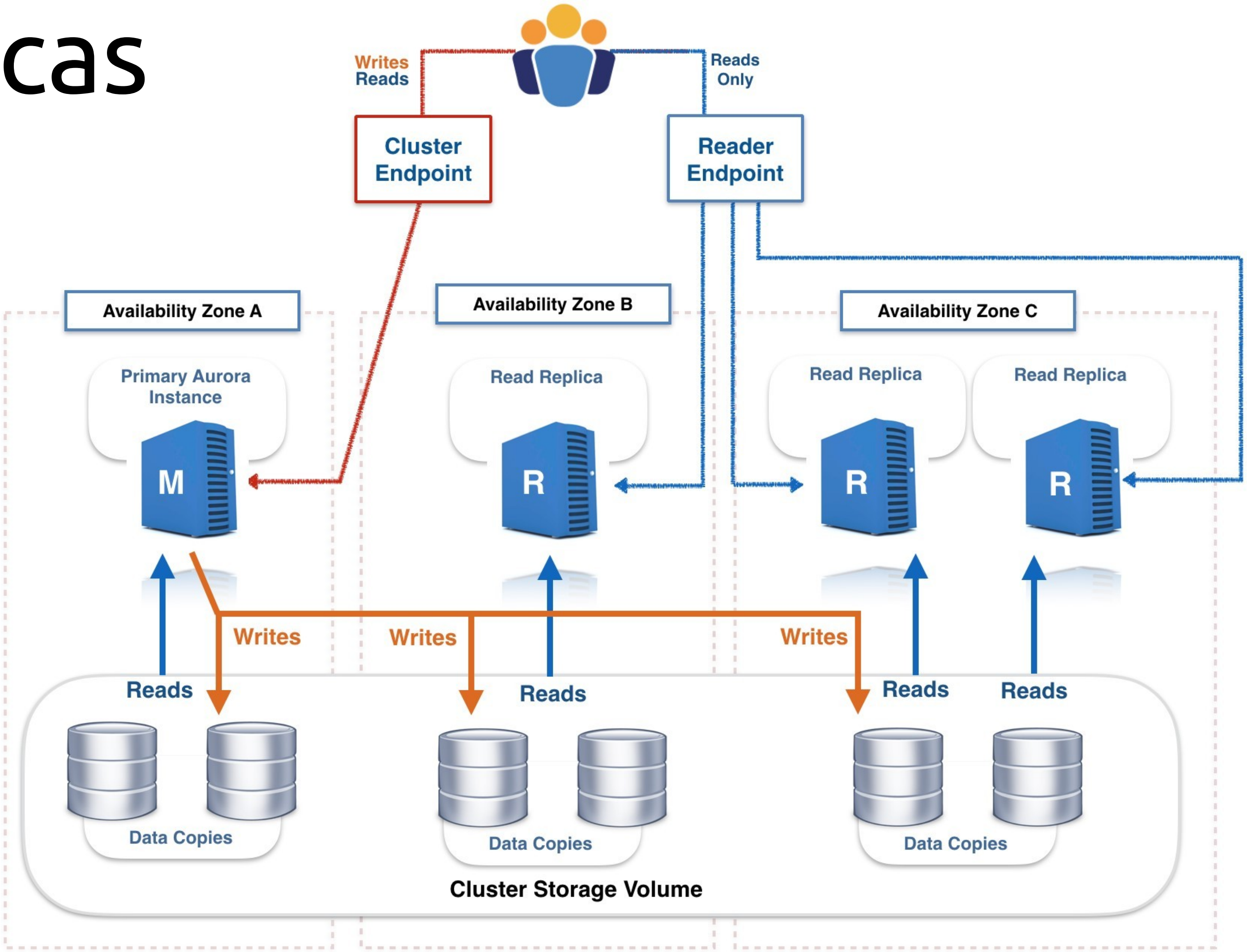
Scaling RDBMS (2)

- Horizontally scalling reads
  - **Read replicas**
- Master only handle writes
  - Reads are split

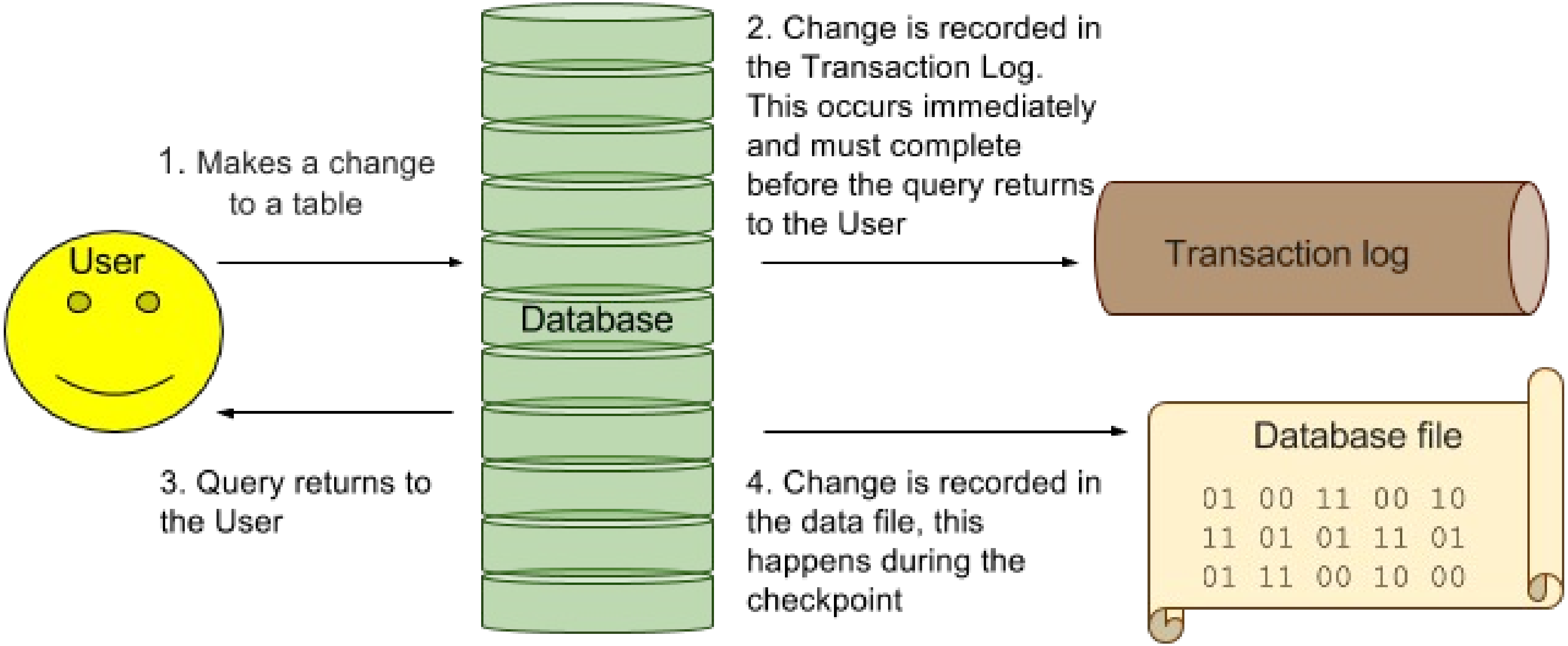


Read replicas

- AWS Aurora
  - Managed MySQL
  - 2 endpoints
    - Read/Write
    - Read only

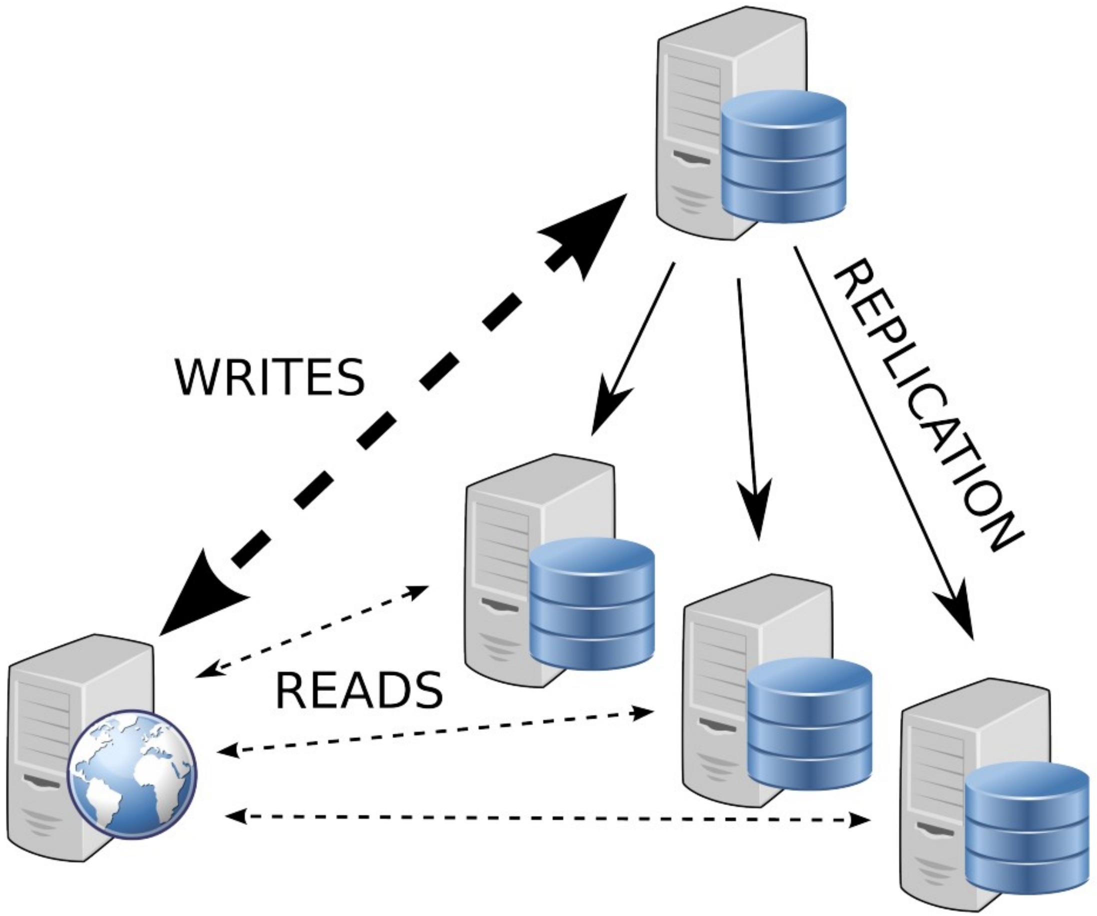


RDBMS Transaction log



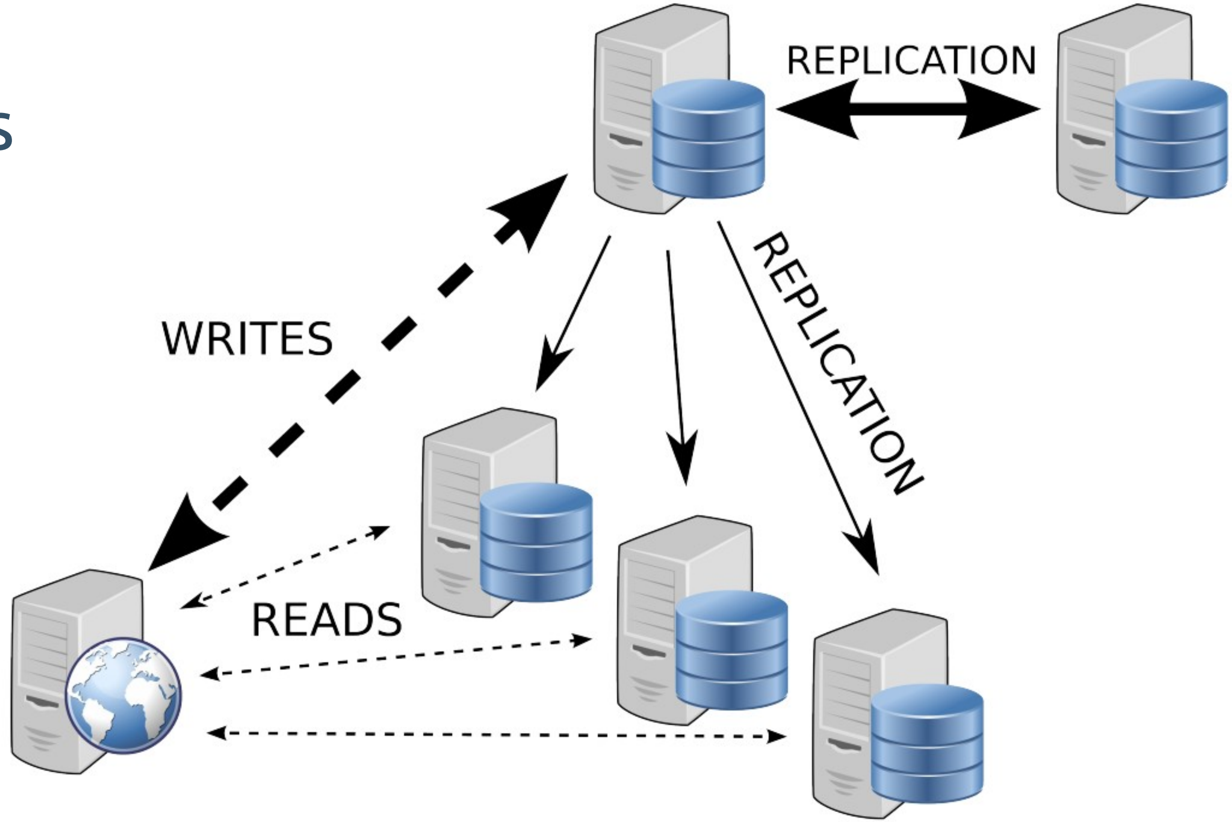
Scaling RDBMS (3)

- Writes become too intensive for the master
- What to do ?



Scaling RDBMS (4)

- Replicate writes
  - **Don't help**, load is also replicated
- Multi-master write
  - **Don't help**, synchronisation protocol too slow

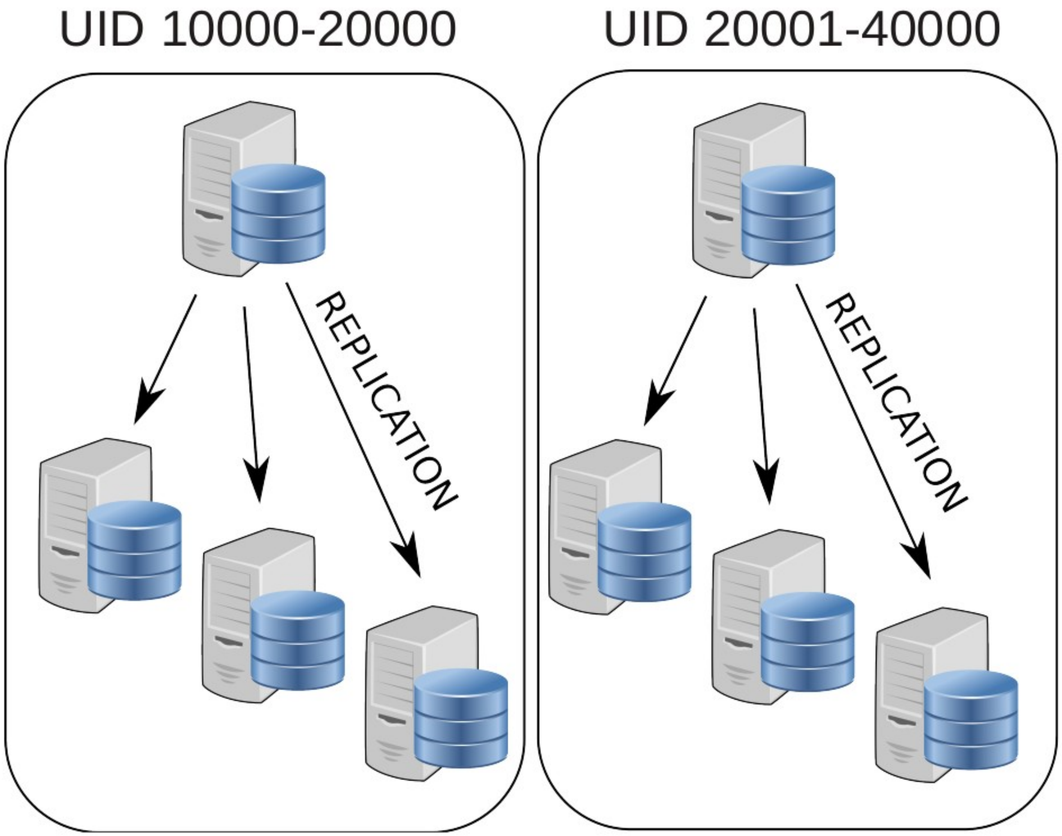


Vertical scaling

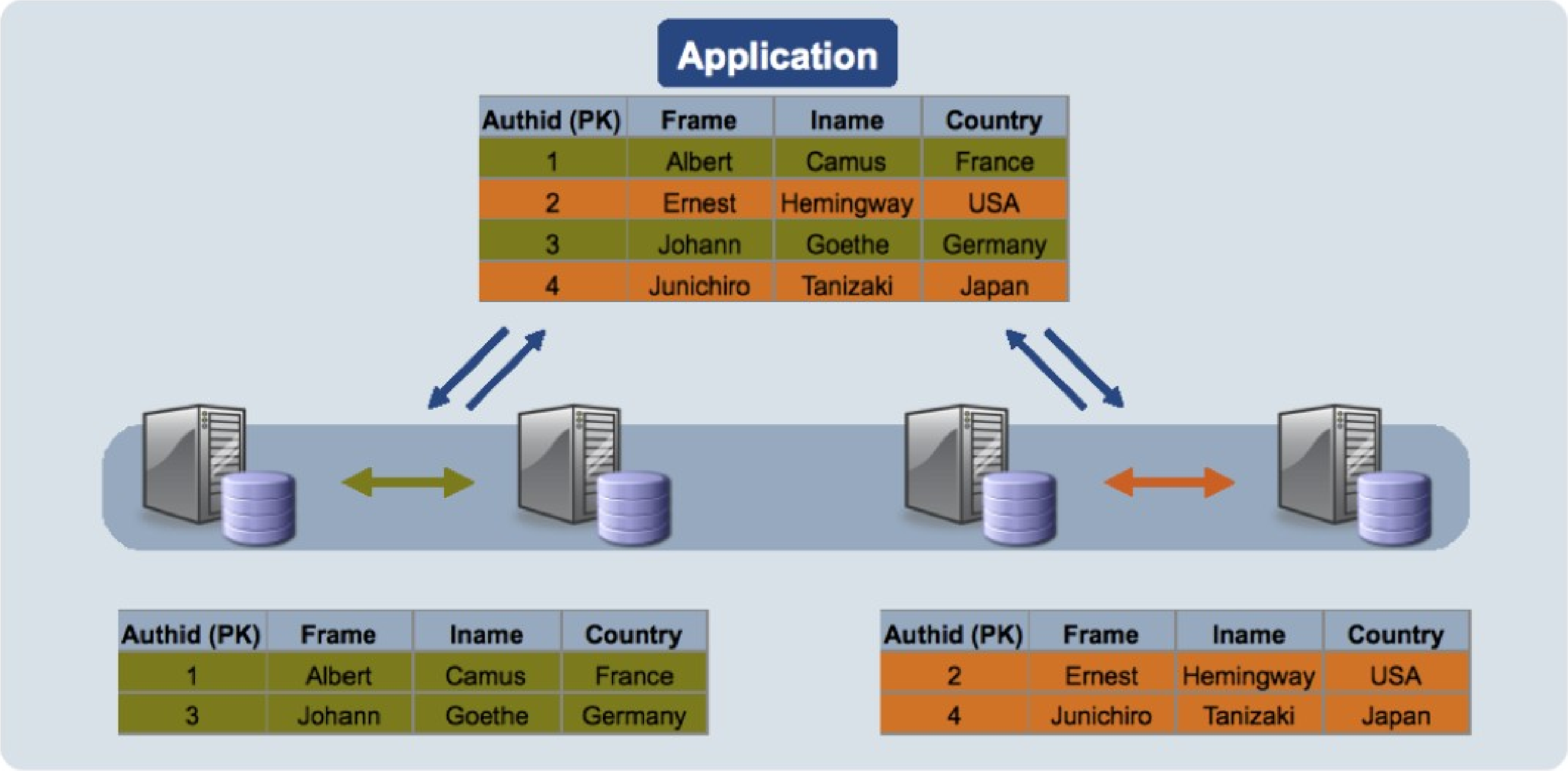
- Add ressources CPU/RAM to the node
  - More memory limits disk I/O
- Expensive
  - A server twice as fast is more than twice as expensive
- Vertical scaling is limited by the hardware available
  - Also limited by locking when number of parallel requests (threads) increase
    - Adding CPUs will not help at some point

Scaling RDBMS (5)

- Partition database
  - Distribute writes
  - Called **sharding**
  - A proxy split incoming queries and aggregate the results



Sharding



Authid (PK)	Frame	lname	Country
1	Albert	Camus	France
2	Ernest	Hemingway	USA
3	Johann	Goethe	Germany
4	Junichiro	Tanizaki	Japan

Authid (PK)	Frame	lname	Country
1	Albert	Camus	France
3	Johann	Goethe	Germany

Authid (PK)	Frame	lname	Country
2	Ernest	Hemingway	USA
4	Junichiro	Tanizaki	Japan

Limits of sharding

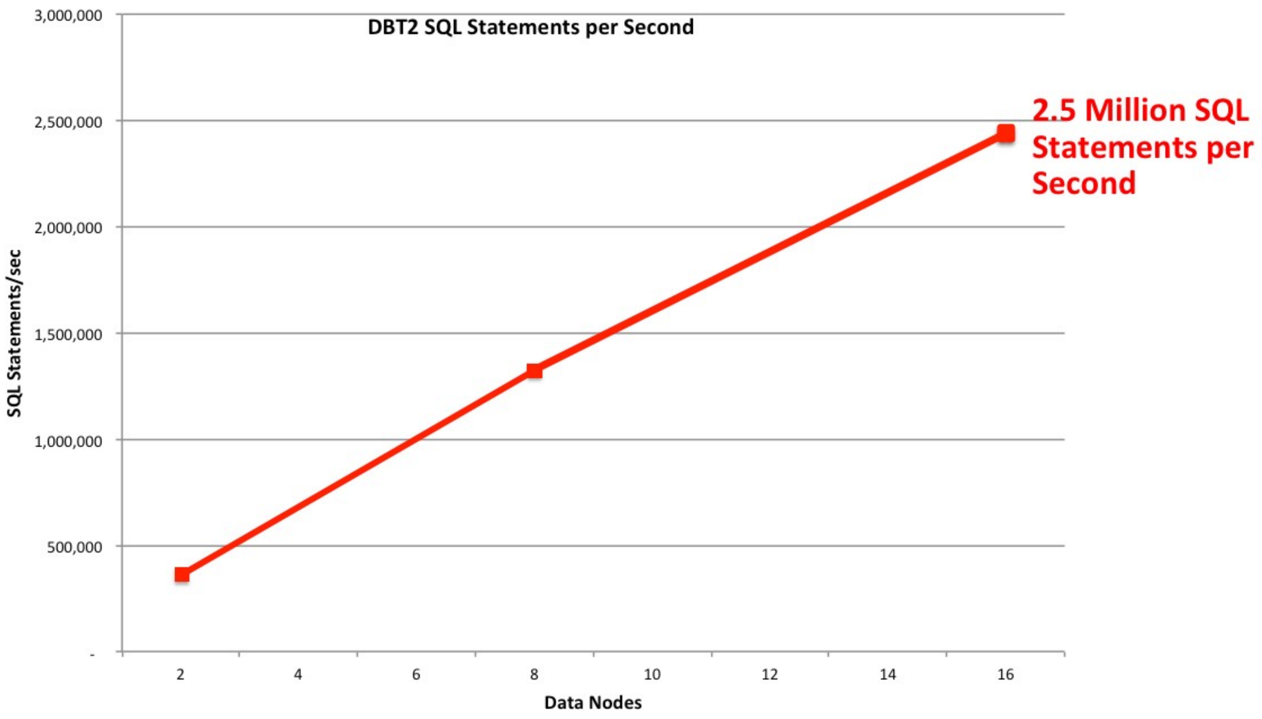
- Tables split over their primary key
  - Often using a hash function over this key
  - Balances entries over all instances
- SQL queries are generally querying entire tables
  - Join queries require comparing and merging data from multiple tables
  - Most queries will involve all instances!
- Devising a good application-specific sharding is possible, but complex and workload-dependent

Sharding and horizontal elasticity

- Careful sharding can allow scaling to a few (typically a dozen) database instances
- But the sharding plan is fixed!
- What if we want to add or remove a node?
  - With hash-partitioning, need to move virtually all data around to accommodate the new node!
  - Transaction throughput would be greatly impaired
  - Elastic sharding generally not supported

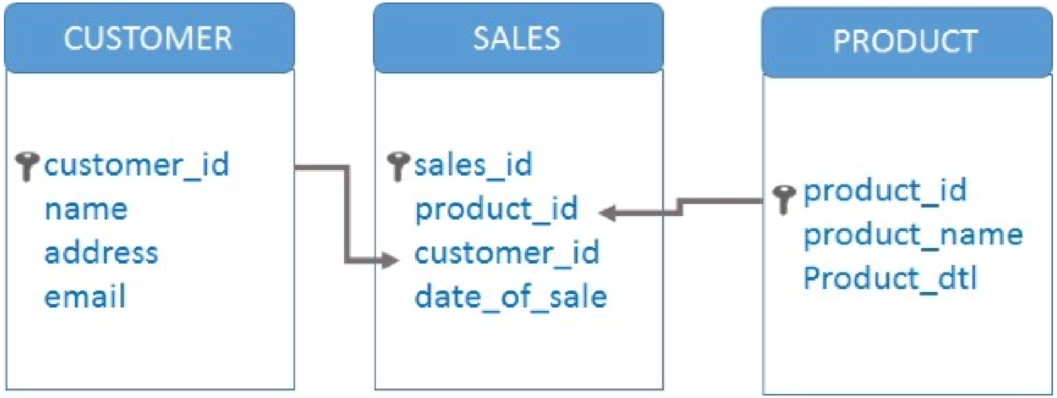
MySQL Cluster - Sharding

- In **memory** distributed DB
- Data replication
- Transparent sharding
- Transaction isolation
  - Only read commited is supported
- High availability
- Max 48 nodes
  - 3TB



SQL columns "classification"

- Primary keys and their pointers
  - Relational
  - Used to JOIN between tables
  - **Indexed**
- WHERE columns
  - Used in WHERE statements
    - Should be **indexed**
- Transactional columns
  - Written by transactions
    - Important data
- Other columns
  - Complementary information retrieved with SELECT
    - comments, ~~images~~...



RDBMS and memory

- Relational DBs are very efficient as long as working sets are in the main memory
  - Including data, indexes and temporary data structures
  - SSDs don't solve the problem
    - Read 1Mo from main memory ~ 1 us
    - Read 1Mo from SSD ~ 1ms
    - Still a 10^3 difference, network I/O has same order of magnitude than SSD
- One can improve RDBMS efficiency by
  - Reducing data size to fit better in memory
    - **Should that column be in a SQL database ?**
  - Add additonal memory (vertical scalling)

How to scale RDBMS - Summary (1)

- When possible: separate unrelated group of tables in different databases - affinity with microservices
- For all read usage that can tollerate (slightly) outdated data, use read replicas
  - Replica lags generally < 100ms
  - Very easy to setup
    - But require application to **distinguish** 2 endpoints
- Use vertical scalability
  - Straightforward to change when cloud managed

How to scale RDBMS - Summary (2)

- Reduce the amount of data stored in your relational database
  - Remove all columns that do not require specific features of your relational database
    - **Horizontal partitionning**… extra data can go in **NoSQL DBs** !
    - Your objects are split in X parts: SQL and NoSQL
  - Keep only efficiently searchable columns (indexed search)
    - For longs TEXT with full text search it is preferable to use specialized full text search DB
- Sharding should be used in last ressort

RDBMS - Takeaway

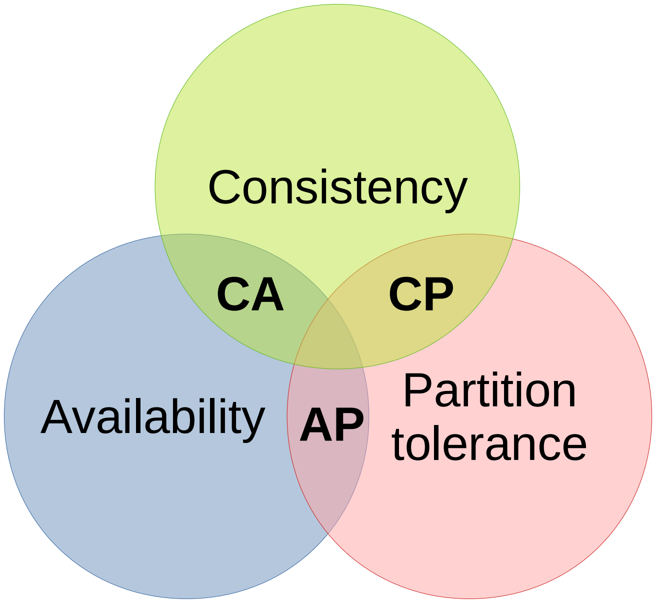
- Still have their place and use
  - Many application do not have so much data
  - A RDBMS using a large EC2 instance can handle a high transaction throughput
- But sometimes we need
  - Better scalability (hundreds or thousands of servers)
  - Ability to elastically scale in and out

NoSQL - Definition

- Schema agnostic
  - A schema describe all possible data and data structures in a relational database
  - In NoSQL, no schema is required
    - But indexes can be specified afterwards
- Non-relational
  - Relations establish connections between data
  - In (document) NoSQL, related informations are stored as aggregates
    - Easy to fetch but duplicated information
- Highly distributable
  - Horizontal scalability is achieved addeing new commodity hardware

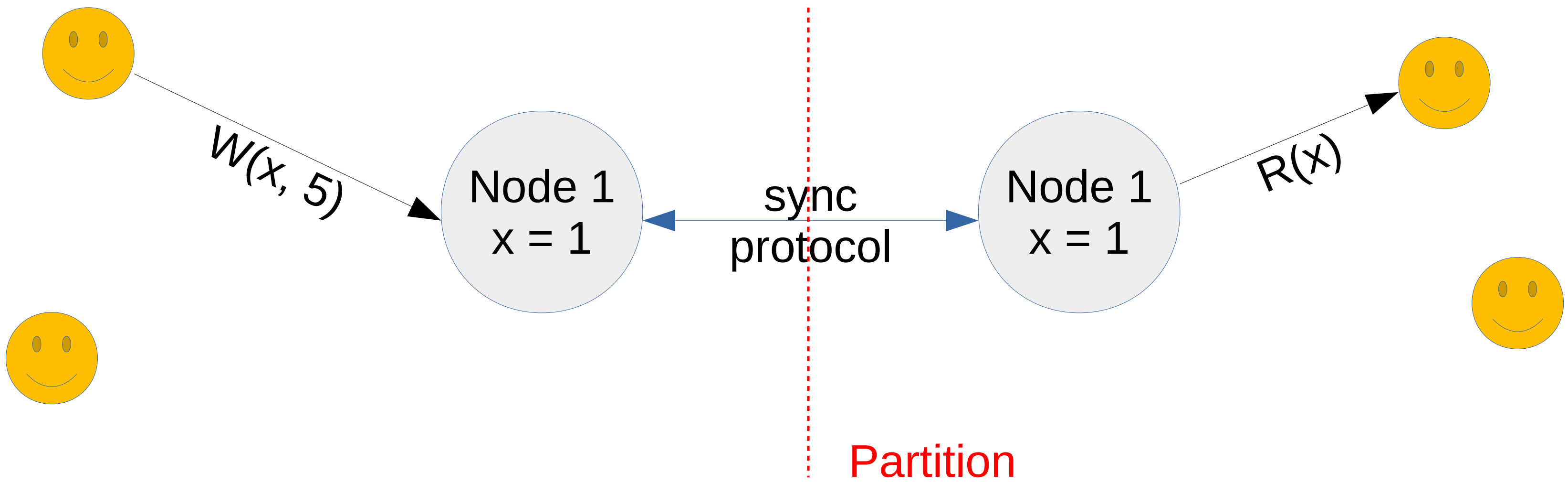
CAP Theorem

- **Properties**
  - **Consistent**
    - Every read receives the most recent write or an error
  - **Available**
    - Every request receives a response that is not an error
  - **Partition** tolerance



- **You can have at most two of these properties for any shared-data system**

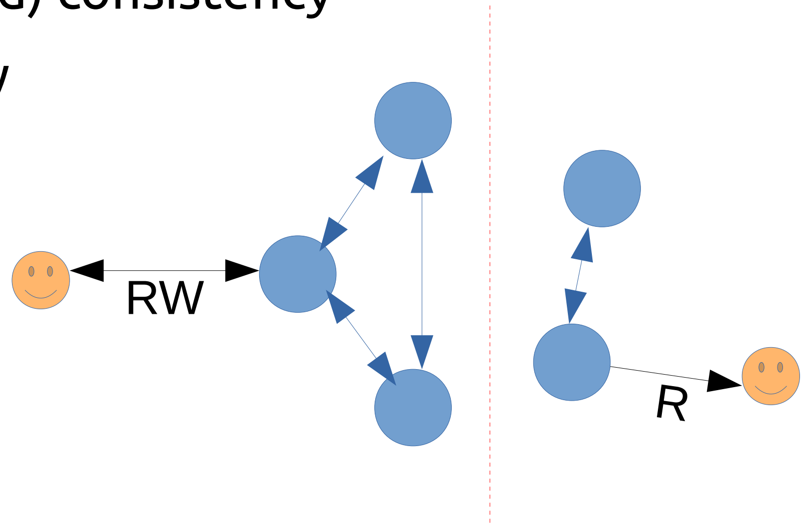
CAP Theorem



- The system is **available**
  - You can write => potentially inconsistent
- The system is **consistent**
  - You cannot write => not available

CAP Theorem - Critisism

- Partition tolerance is not a choice
  - Network are always unreliable even local networks
- Consistency means only "strong" (linearized) consistency
- Does not describe when network gets slow
- Does not describe nodes failures
- Many intermediate solutions
  - Quorum
    - One side is fully available
  - Partially available for read
    - During a certain delay
- Classifying systems as CP or AP is too simplistic



BASE

- (In contrast to ACID)
- **B**asically **A**vailable
- **S**oft state
- **E**ventual consistency
  - Accept to send approximative answers
  - Conflict resolution
    - **Reconciliation** of differences between multiple copies
    - No universal approach
      - Time based (last change wins)
      - Ask user (Dropbox)

ACID - CAP - BASE

- Consistency has not the same meaning in ACID and in CAP
  - In ACID: about respecting DB rules
  - In CAP: are the nodes fully synchronized ?
- It's easier to reason (and develop application) with strong concistency models
  - Will see next week the different consistency models
- General tradeoffs
  - Traditional database systems designed with ACID guarantees choose consistency over availability
  - Systems designed around the BASE philosophy, common in the NoSQL movement, generally choose availability over consistency

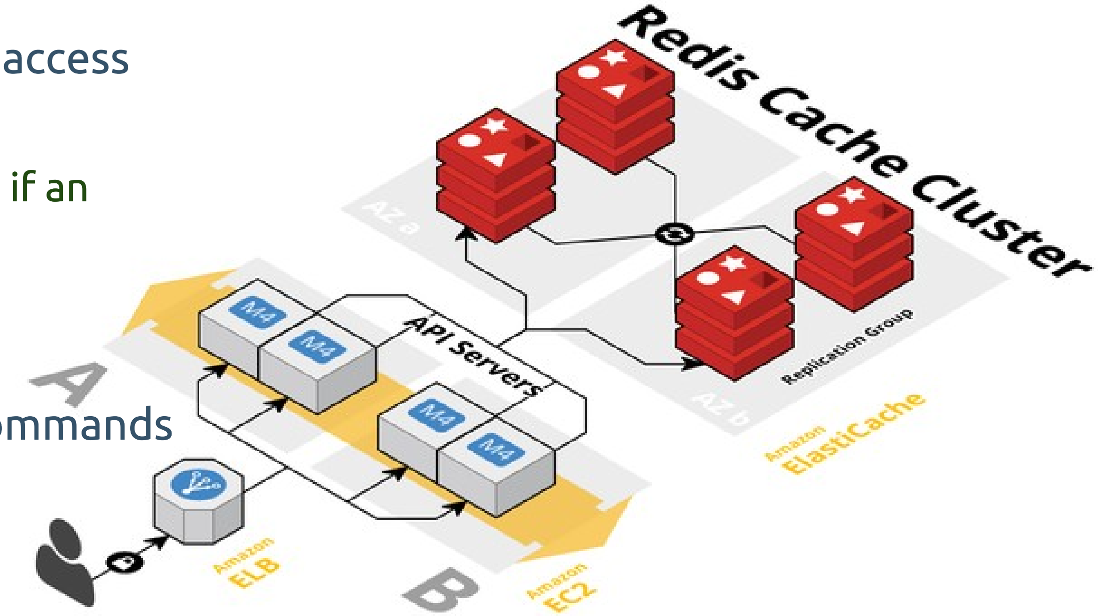
NoSQL - Key/Value store

- Most common NoSQL interface
  - Like a global Hash Map
    - put(key, value)
    - get(key) => value
- Focus on linear scalability
- Common usages
  - Sessions, cache, storage, configuration, …

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

NoSQL - Redis

- Key/Value store
  - Extensible to schema-less Document store
- In memory storage
  - Focus on high performance, low-latency access
  - By default no durability
    - Frequently used for data that can be lost if an instance disapear
      - **Caching**
- Atomic execution
  - All or nothing execution of a group of commands
- Binary protocol
- Server-side scripts (Lua language)



NoSQL - Document store

- Subclass of Key/Value store
- Document as central notion
  - Generally structured in JSON, XML or YAML
  - Document are accessed via a unique key
  - Contrary to relational DB, where relations are expressed, a document in an **aggregat**
- Grouped by
  - Collections
    - Correspond to tables in relational databases
- Query language aware of the document's content
  - CRUD
  - Specific properties indexing

```
{
    "FirstName": "Bob",
    "Address": "5 Oak St.",
    "Hobby": "sailing"
}
```

```
<contact>
  <firstname>Bob</firstname>
  <lastname>Smith</lastname>
  <phone type="Cell">(123) 555-0178</phone>
  <phone type="Work">(890) 555-0133</phone>
  <address>
    <type>Home</type>
    <street1>123 Back St.</street1>
    <city>Boys</city>
    <state>AR</state>
    <zip>32225</zip>
    <country>US</country>
  </address>
</contact>
```

NoSQL - AWS DynamoDB (1)

- Key/Value store + Document store available as PaaS
  - Table / Items / Attributes
    - One primary key
    - Schema-less
    - Secondary indexes
  - JSON/HTTP interface
    - Query with JSON based interface with variables replacement
  - Also selectable consistency level
    - Strong consistency flag in the query
- Scalable seamlessly, low latency, events stream

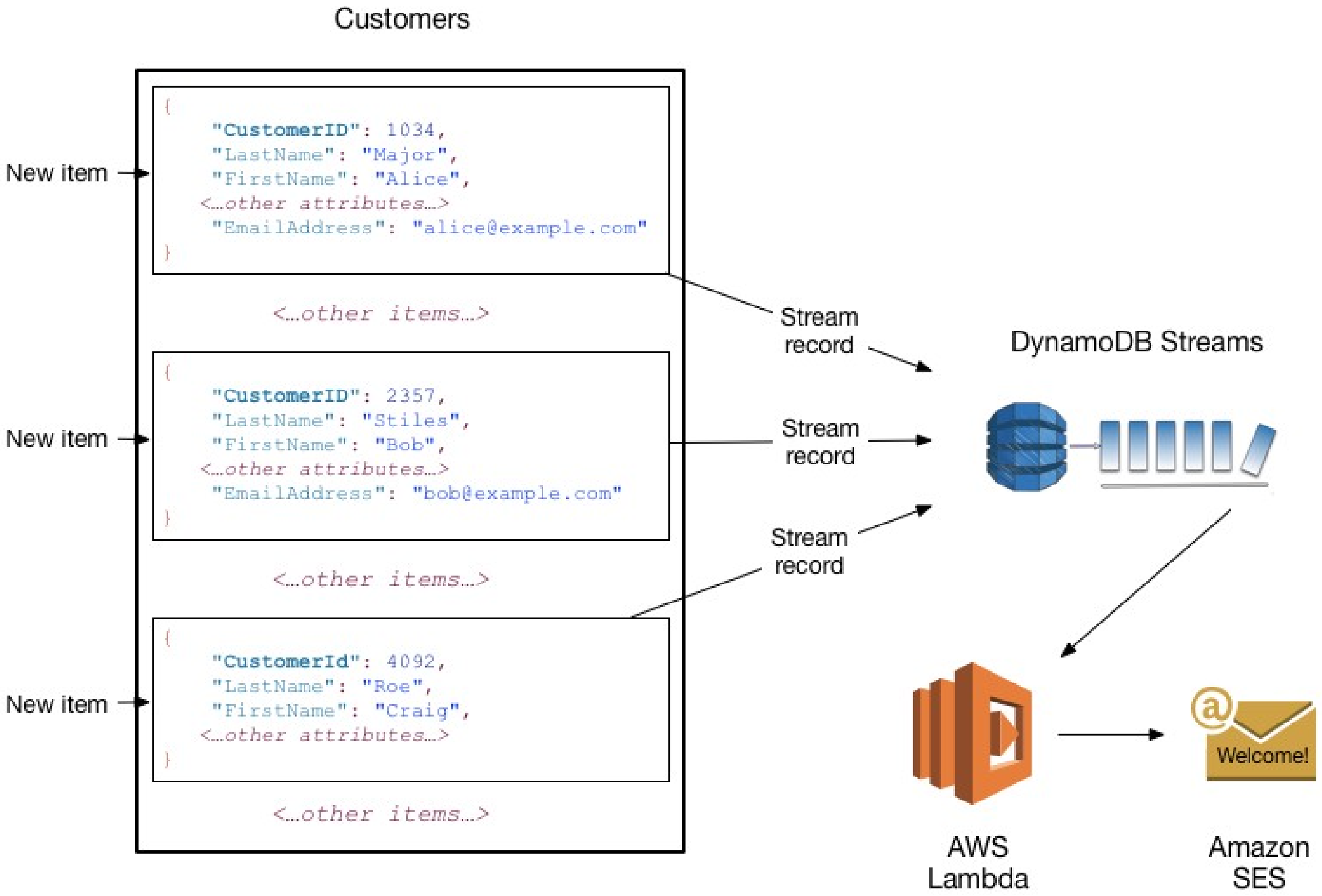
People


```
{
    "PersonID": 101,
    "LastName": "Smith",
    "FirstName": "Fred",
    "Phone": "555-4321"
}
```

```
{
    "PersonID": 102,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
        "Street": "123 Main",
        "City": "Anytown",
        "State": "OH",
        "ZIPCode": 12345
    }
}
```

```
{
    "PersonID": 103,
    "LastName": "Stephens",
    "FirstName": "Howard",
    "Address": {
        "Street": "123 Main",
        "City": "London",
        "PostalCode": "ER3 5K8"
    },
    "FavoriteColor": "Blue"
}
```

NoSQL - AWS DynamoDB (2)

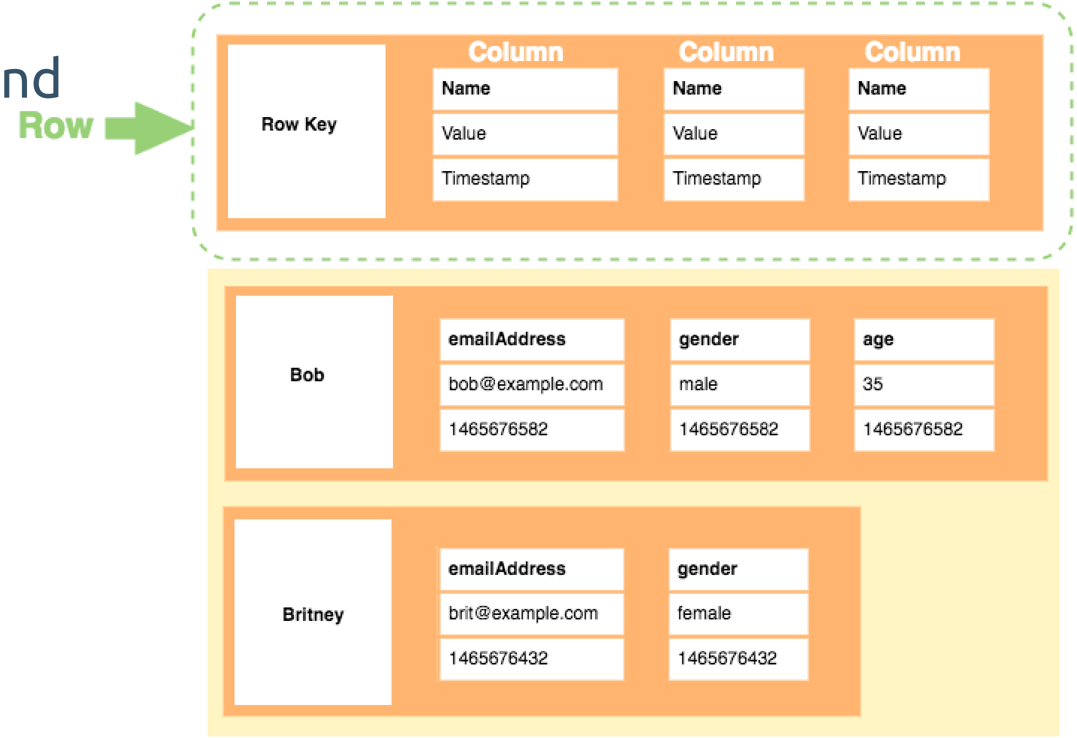


NoSLQ - Apache Cassandra (1)

- Design influenced by 2 papers
  - Google BigTable
    - Memory and disk storage
  - AWS Dynamo
    - Distributed architecture
      - No SPOF
      - Linear scalability
- Created by Facebook
  - Now open source at Apache Foundation

NoSLQ - Apache Cassandra (2)

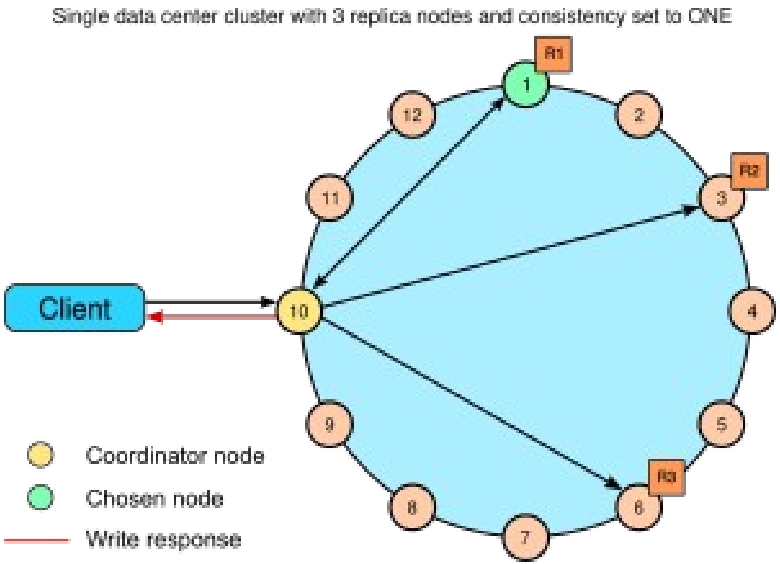
- Wide Column store
  - Hybrid between Key/Value and tabular
    - Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
  - Columns
    - Efficiently compressed
    - Easy to partition
    - Efficient for aggregation queries: SUM, COUNT, ...



Images: https://database.guide/what-is-a-column-store-database/

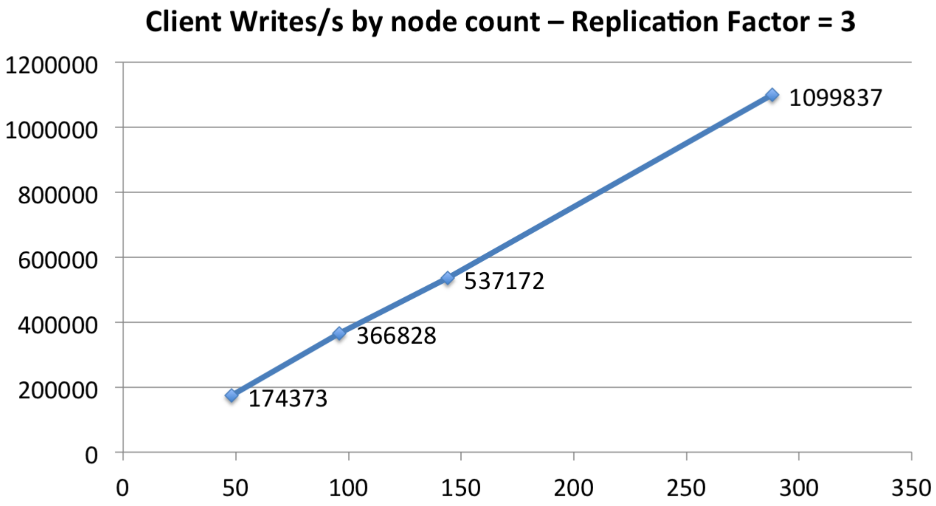
NoSLQ - Apache Cassandra (3)

- Dynamo
  - P2P
    - All nodes have the same role
    - Every node can accept get() and put()
      - The node become the coordinator of the write operation
  - DHT (Distributed Hash Table)
    - Nodes are placed in a virtual ring
    - Key hashed to a position in that ring
    - The closer node is will insert key
      - The data is replicated to N nodes, replication level is configurable



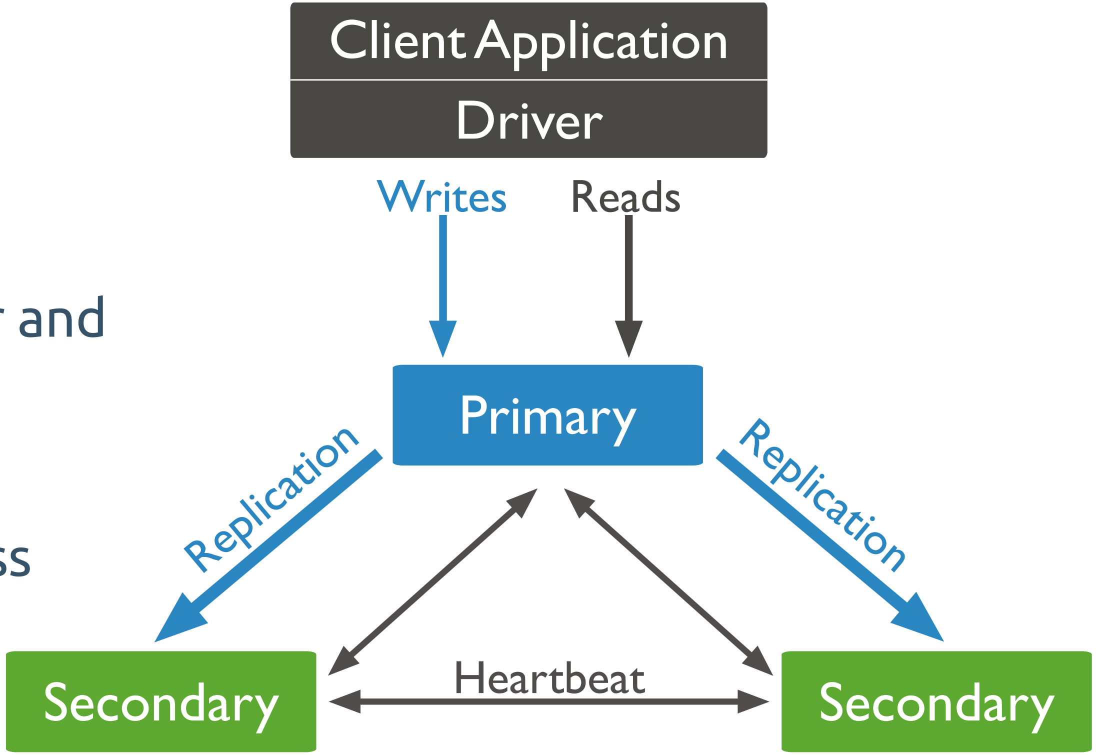
NoSLQ - Apache Cassandra (4)

- Selectable consistency level
  - Eventual consistency
    - Last Write Wins (LLW)
  - Different modes: ANY, QUORUM…
- Cassandra query language
  - Schema definition
  - Many similarities with SQL
- Batch requests (pseudo transaction)
- Known deployments with more than 100.000+ servers (at Apple)



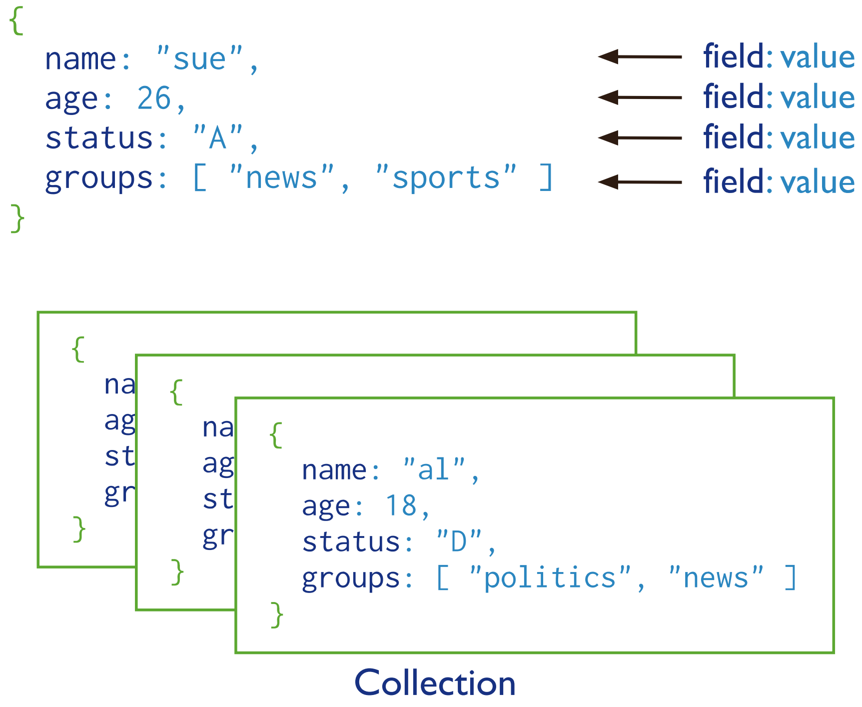
NoSQL - MongoDB (1)

- Most popular NoSQL Document store
- High performance
  - Document properties indexing
- High availability
  - **Replica set**: automatic failover and redundancy
- Horizontal scalability
  - Sharding distributes data across cluster
  - Zones: locality aware



NoSQL - MongoDB (2)

- JSON-like documents
  - Collections
- Rich Query Language
  - Server-side Javascript execution
  - Data aggregation
    - MapReduce
  - Text search
- Change Streams
  - Receive events when data is modified
- Transaction
  - Multi-document ACID Transactions with snapshot isolation



NoSQL - MongoDB JS queries

```
db.users.insertOne(
  {
    name: "sue",
    age: 26,
    status: "pending"
  }
)
```

collection
field: value
field: value
field: value
document

```
db.users.find(
  { age: { $gt: 18 } },
  { name: 1, address: 1 }
).limit(5)
```

collection
query criteria
projection
cursor modifier

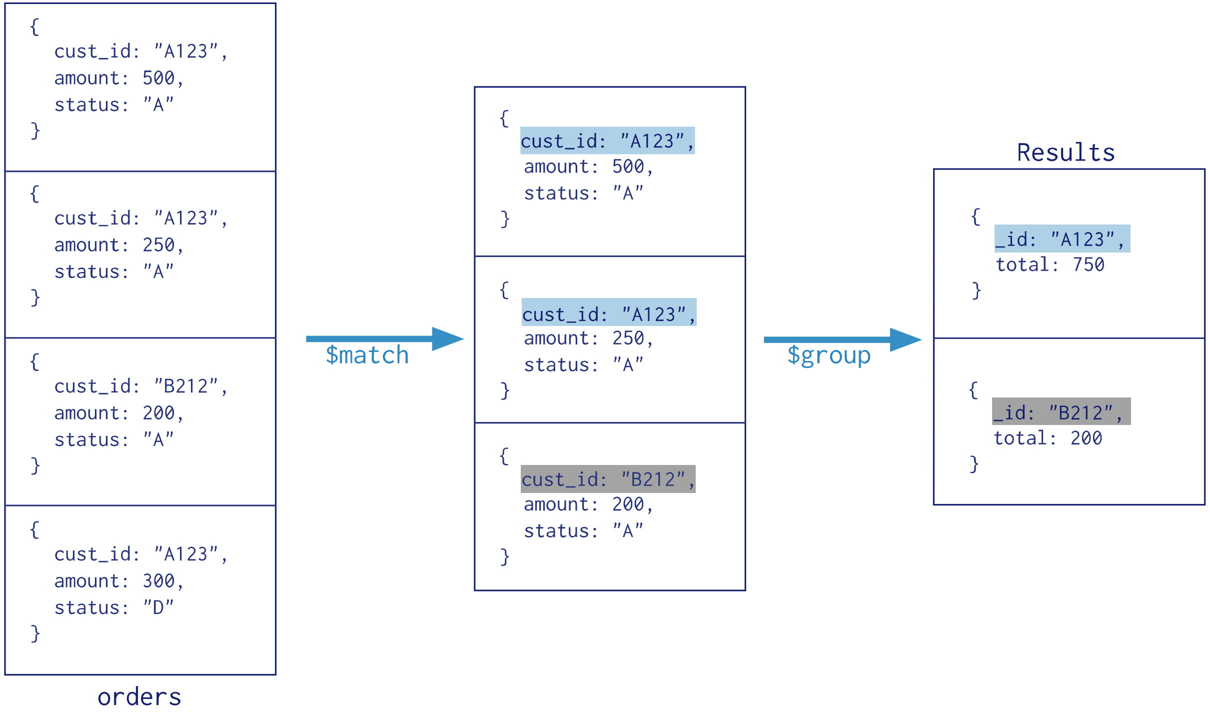
```
db.users.deleteMany(
  { status: "reject" }
)
```

collection
delete filter

NoSQL - MongoDB aggregation

Collection

```
db.orders.aggregate( [
   $match stage →  { $match: { status: "A" } },
   $group stage →  { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```

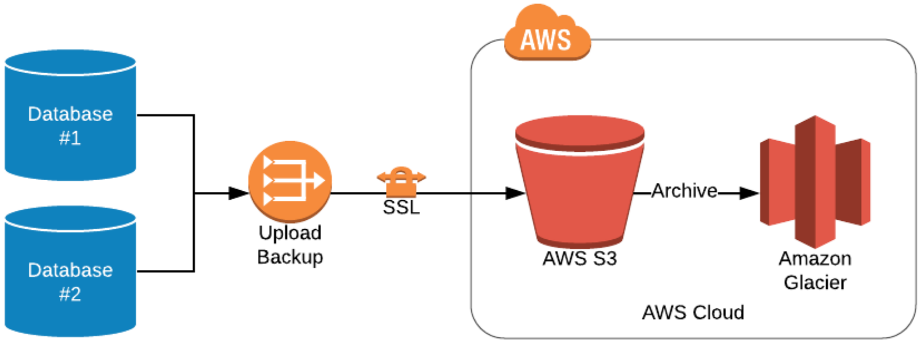


NoSQL - Objects stores

- A form of key/value stores for immutable data
  - Once published a value is not modified
  - Can use a version number in its identifier
- Advantages for application
  - URI to resource can be provided to client, who can send GET requests using HTTP (instead of going through a service interface)
  - Resource representation can be cached or distributed by CDNs
- Generally available as a managed platform service
  - Google Cloud storage
  - OpenStack Swift
  - Amazon Simple Storage Service (S3)

NoSQL - AWS S3 (1)

- Designed for binary object storage
  - Represented as a hierarchical file system
    - Key = <path>, Value = <content>
  - Objects are grouped by **buckets** (= a namespace)
- 99.999999999% of durability for objects
  - Distributed across 3 availability zones
  - Different access levels
- Supports versionning
- A bucket can be used as a website
  - Route53 => DNS
  - AWS CloudFront => CDN

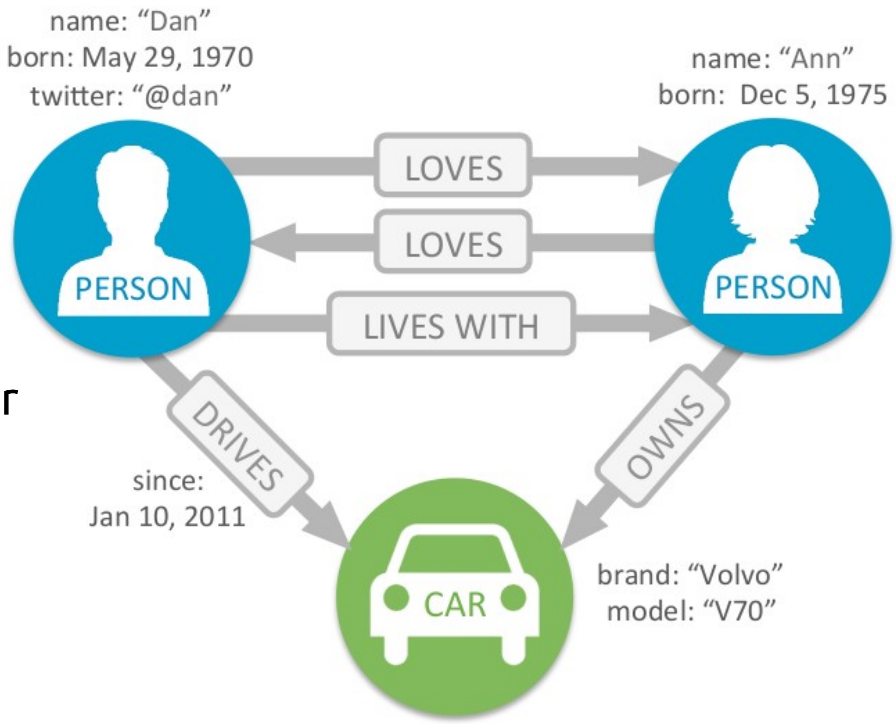


NoSQL - AWS S3 (2)

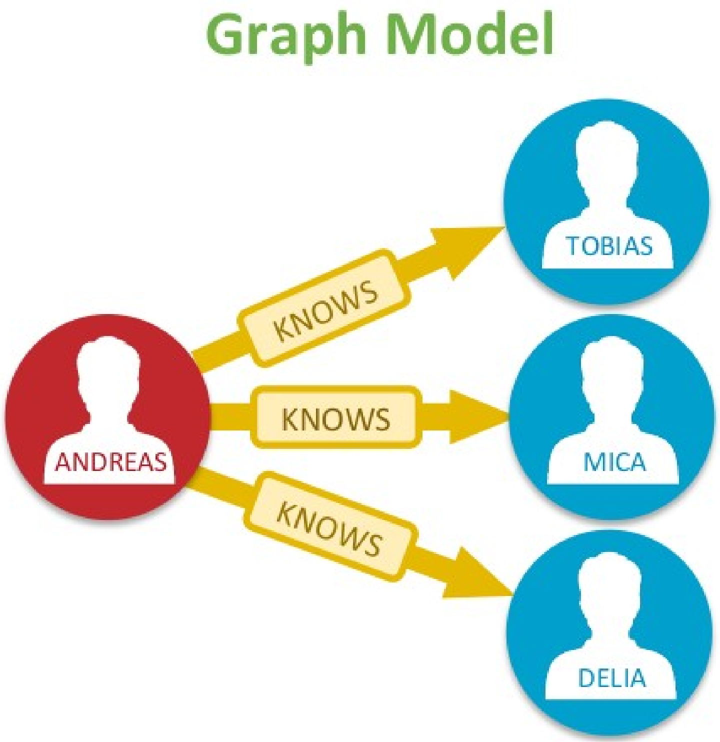
	S3 Standard	S3 Standard-IA	S3 One Zone-IA	Amazon Glacier
Designed for Durability	99.999999999%	99.999999999%	99.999999999%†	99.999999999%
Designed for Availability	99.99%	99.9%	99.5%	N/A
Availability SLA	99.9%	99%	99%	N/A
Availability Zones	≥3	≥3	1	≥3
Minimum Capacity Charge per Object	N/A	128KB*	128KB*	N/A
Minimum Storage Duration Charge	N/A	30 days	30 days	90 days
Retrieval Fee	N/A	per GB retrieved	per GB retrieved	per GB retrieved**
First Byte Latency	milliseconds	milliseconds	milliseconds	select minutes or hours***
Storage Type	Object	Object	Object	Object
Lifecycle Transitions	Yes	Yes	Yes	Yes

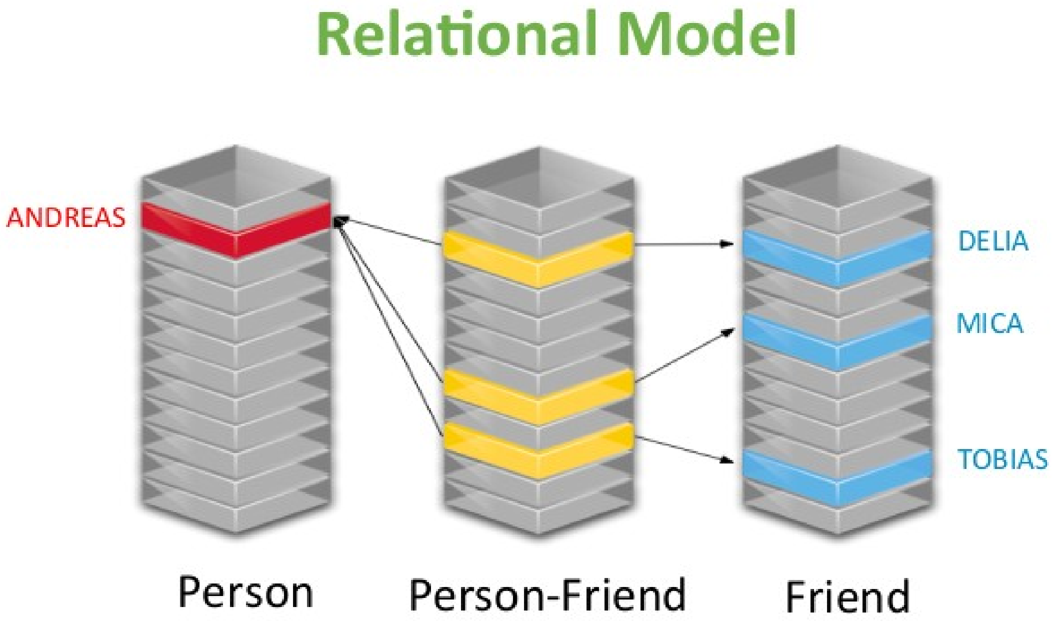
NoSQL - Graph DB

- Based on graph theory
  - Nodes used to store data
    - Schema-less, flexibility
  - Edges (relationships) to connect nodes
    - Also contains data to indicate the nature of the relation
- Permits to modelize data more intuitively, closer to the reality, full of relations
  - Model is naturally adaptive
  - Social relations, shortest path, decisions, ...
- Following edges
  - Avoid JOINing tables
  - Avoid the need for keys and indexes
- Specific query language

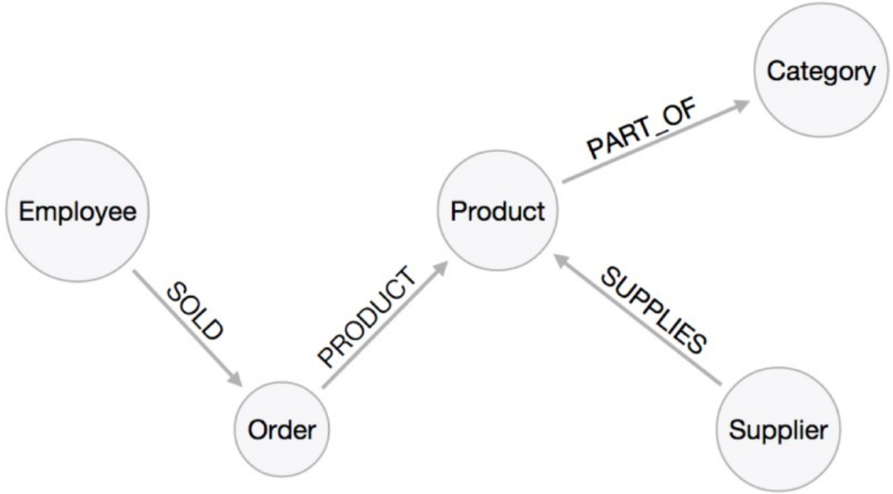


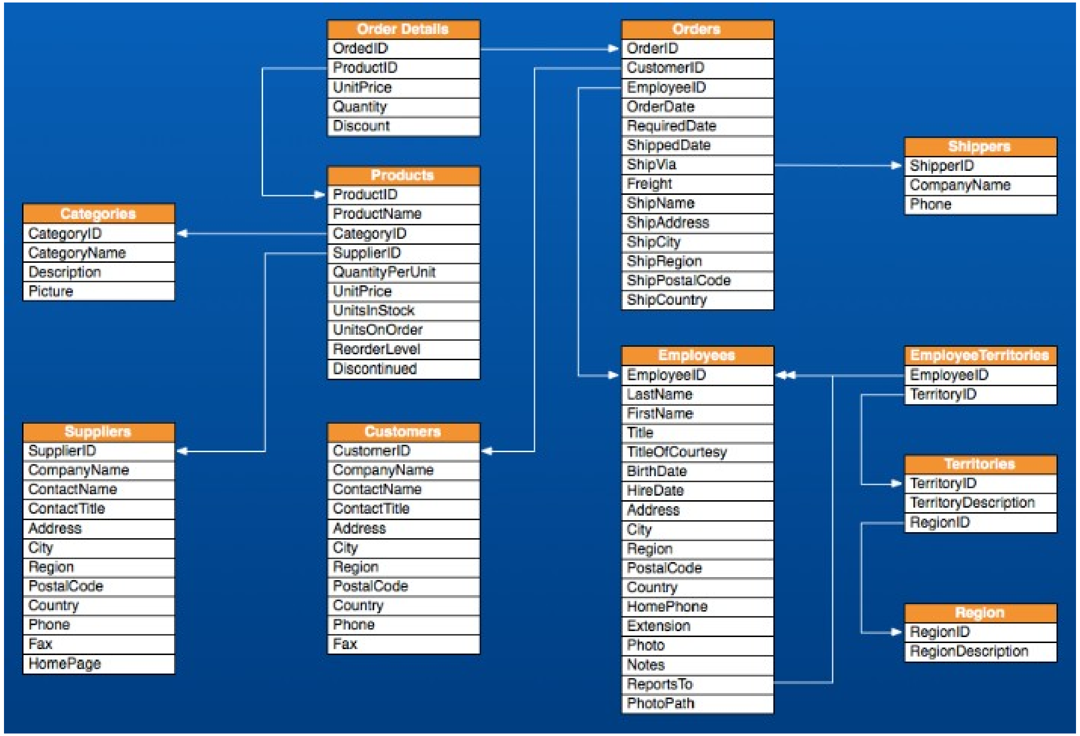
NoSQL - Graph DB vs SQL (1)





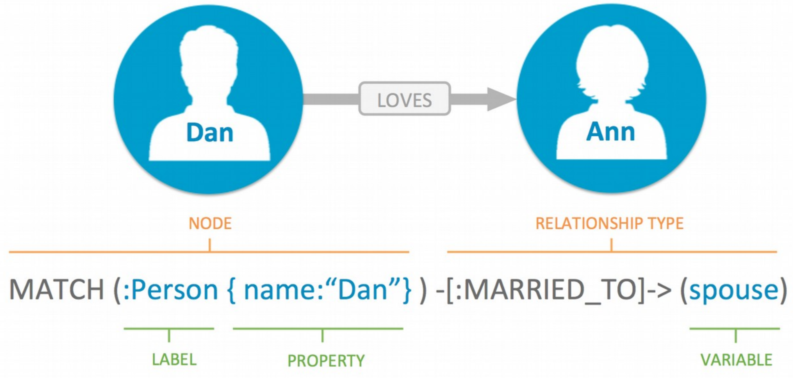
NoSQL - Graph DB vs SQL (2)





NoSQL - Neo4J

- Graph DB
  - Nodes properties as key/value
  - Directional relationships
  - Properties can be indexed
  - Labels group nodes into sets
- Cypher query language




```
MATCH (nicole:Actor {name: 'Nicole Kidman'})-[:ACTED_IN]->(movie:Movie)
WHERE movie.year < $yearParameter
RETURN movie
```

```
MATCH (start:Content)-[:RELATED_CONTENT]->(content:Content)
WHERE content.source = 'user'
OPTIONAL MATCH (content)-[r]-()
DELETE r, content
```

NoSQL - Cypher vs SQL

Find all direct reports and how many people they manage, up to 3 levels down

Cypher Query

```
MATCH (manager)-[:REPORTS_TO*0..3]->(boss),
      (report)-[:REPORTS_TO*1..3]->(manager)
WHERE boss.name = “John Doe”
RETURN manager.name AS Manager,
  count(report) AS TotalReports
```

SQL Query

```
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.pid AS directReportees, 0 AS count
  FROM person_reportee manager
  WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
  SELECT manager.pid AS directReportees, count(manager.directly_manages) AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT manager.pid AS directReportees, count(L2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
FROM (
SELECT manager.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
UNION
SELECT reportee.pid AS directReportees, count(reportee.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT depth1Reportees.pid AS directReportees,
count(depth2Reportees.directly_manages) AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT T.directReportees AS directReportees, sum(T.count) AS count
  FROM(
  SELECT reportee.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee reportee
ON manager.directly_manages = reportee.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
UNION
SELECT L2Reportees.pid AS directReportees, count(L2Reportees.directly_manages)
AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
GROUP BY directReportees
) AS T
GROUP BY directReportees)
UNION
(SELECT L2Reportees.directly_manages AS directReportees, 0 AS count
FROM person_reportee manager
JOIN person_reportee L1Reportees
ON manager.directly_manages = L1Reportees.pid
JOIN person_reportee L2Reportees
ON L1Reportees.directly_manages = L2Reportees.pid
WHERE manager.pid = (SELECT id FROM person WHERE name = "fName lName")
)
```

AWS databases

If You Need	Consider Using	Product Type
A fully managed MySQL and PostgreSQL-compatible relational database with the performance and availability of enterprise databases at 1/10th the cost.	Amazon Aurora	Relational Database
A managed relational database in the cloud that you can launch in minutes with just a few clicks.	Amazon RDS	Relational Database
A serverless, NoSQL database that delivers consistent single-digit millisecond latency at any scale.	Amazon DynamoDB	NoSQL Database
A fast, fully managed, petabyte-scale data warehouse at 1/10th the cost of traditional solutions.	Amazon Redshift	Data Warehouse
To deploy, operate, and scale an in-memory data store based on Memcached or Redis in the cloud.	Amazon ElastiCache	In-Memory Data Store
A fast, reliable, fully managed graph database to store and manage highly connected data sets.	Amazon Neptune	Graph Database

Conclusion

- A very large choice of different NoSQL databases is available on the market
  - NoSQL data models range from very simple to very expressive
  - Various (configurable) levels of concistency guarantees
  - Simple transactional models (generally less guarantees than ACID)
    - => It's more difficult to compare NoSQL systems !
- Today's architectures are generally composed of
  - One (or more) SQL database
  - **Many different NoSQL** databases
    - Documents, sessions, cache, logs, …
      - **Choose the right tool for the right job !**
- For better customer experience, many companies choose high availability over strong consistency

References

- Google's papers:
  - Google File System (Ghemawat et al, 2003)
  - Bigtable: A Distributed Storage System for Structured Data (Chang et al, 2006)
  - F1: A Distributed SQL Database That Scales (Shute et al, 2013)
  - Pregel: a system for large-scale graph processing (Malewicz et al, 2010)
- Dynamo: Amazon’s Highly Available Key-value Store (DeCandia et al, 2007)
- Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services
- https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e