NOSQL Databases

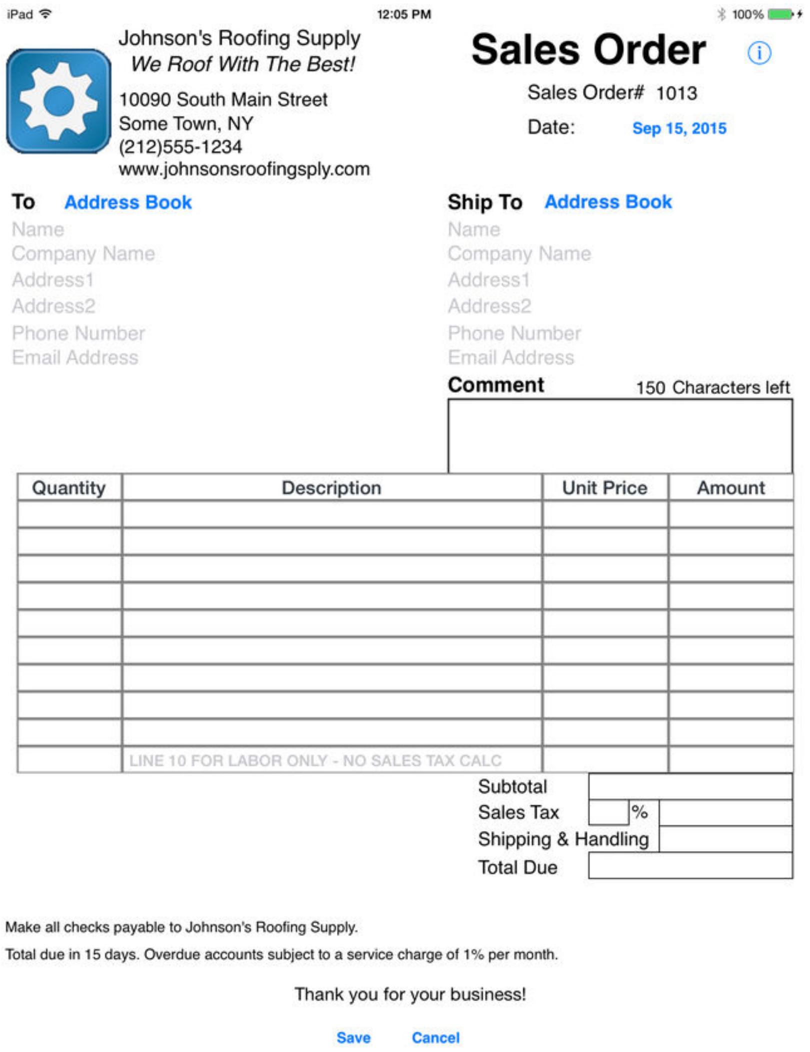
PART 2 – Key Concepts (Aggregates, Replication, Sharding)

Aggregate and Atomicity

- An **aggregate** is simply a collection of items that should be handled as a **single atomic unit.**
- An aggregate is the **unit of consistency** in most NoSQL databases.
- You can consider operations that update multiple fields in a single aggregate as exhibiting ACID like characteristics (note data replication changes this), although operations that modify data spread across more than one aggregate are unlikely to provide the same guarantees of consistency
  - e.g. In most document databases the unit of aggregation is the document, so a write operation that modifies data in several documents is not **atomic.**
  - (Note most NoSQL databases **do not support atomicity** for operations that span aggregates)

Note: The term "aggregation" comes from Domain-Driven Design

A Sales Order!



iPad 12:05 PM 100%

Johnson's Roofing Supply
We Roof With The Best!
10090 South Main Street
Some Town, NY
(212)555-1234
www.johnsonsroofingsply.com

Sales Order

Sales Order# 1013
Date: Sep 15, 2015

To Address Book
Name
Company Name
Address1
Address2
Phone Number
Email Address

Ship To Address Book
Name
Company Name
Address1
Address2
Phone Number
Email Address

Comment 150 Characters left

Quantity	Description	Unit Price	Amount
	LINE 10 FOR LABOR ONLY - NO SALES TAX CALC		

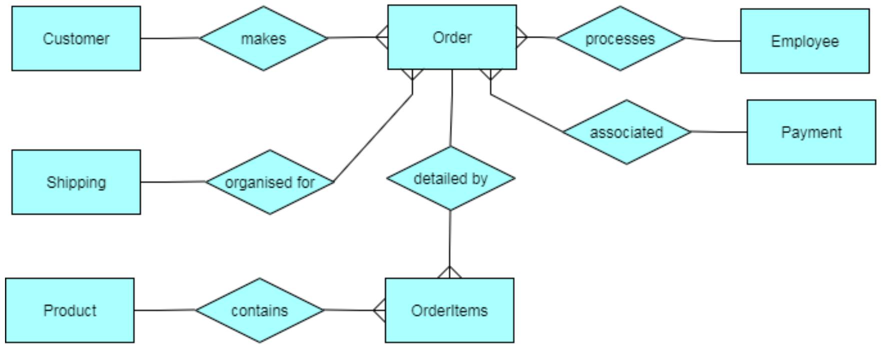
Subtotal		
Sales Tax	%	
Shipping & Handling		
Total Due		

Make all checks payable to Johnson's Roofing Supply.
Total due in 15 days. Overdue accounts subject to a service charge of 1% per month.

Thank you for your business!

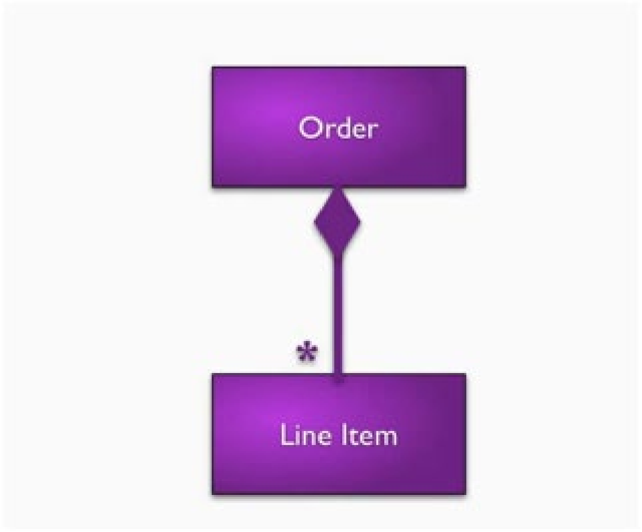
Save Cancel

Possible ERD – lots of joins in an Rdb!

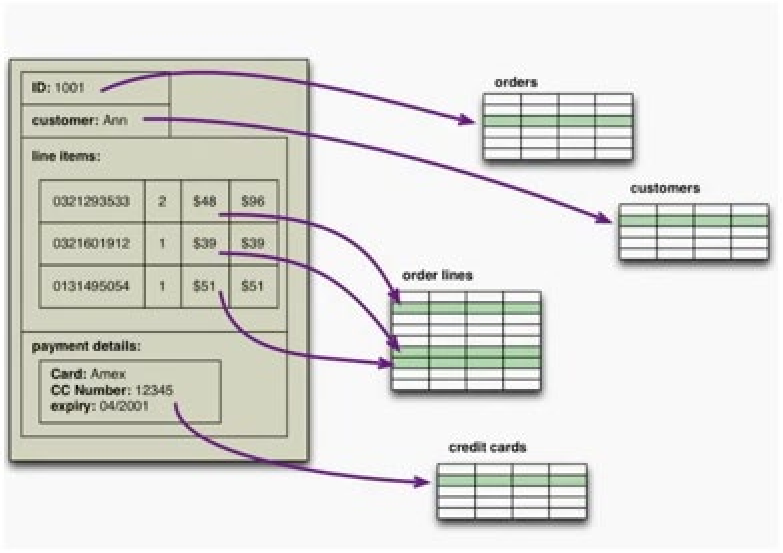


Design and Storage Example

- Modelling Orders as classes
- (a simple design)



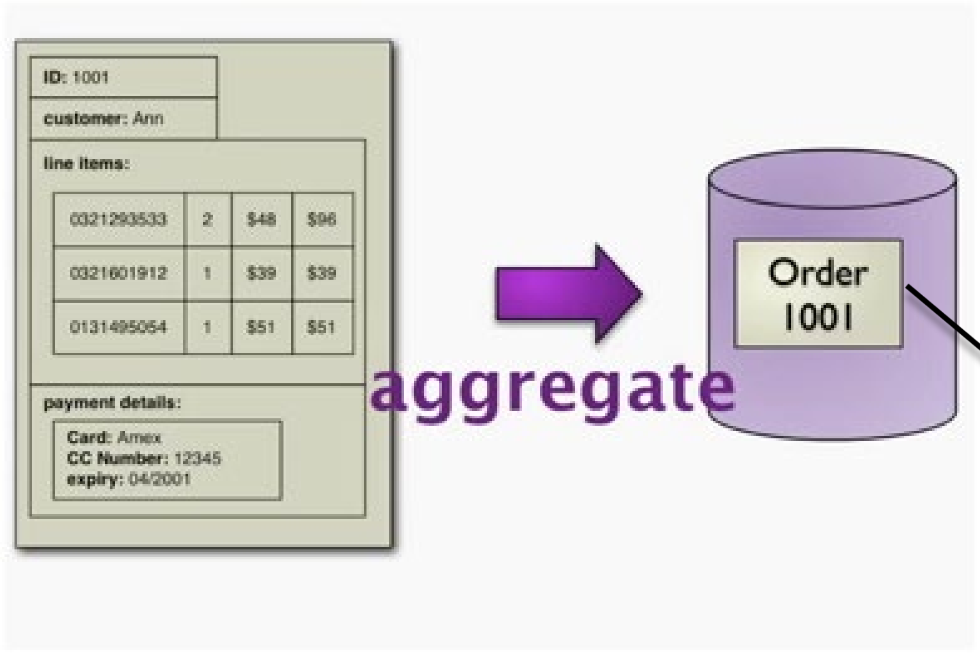
- Storing Orders in a relational database



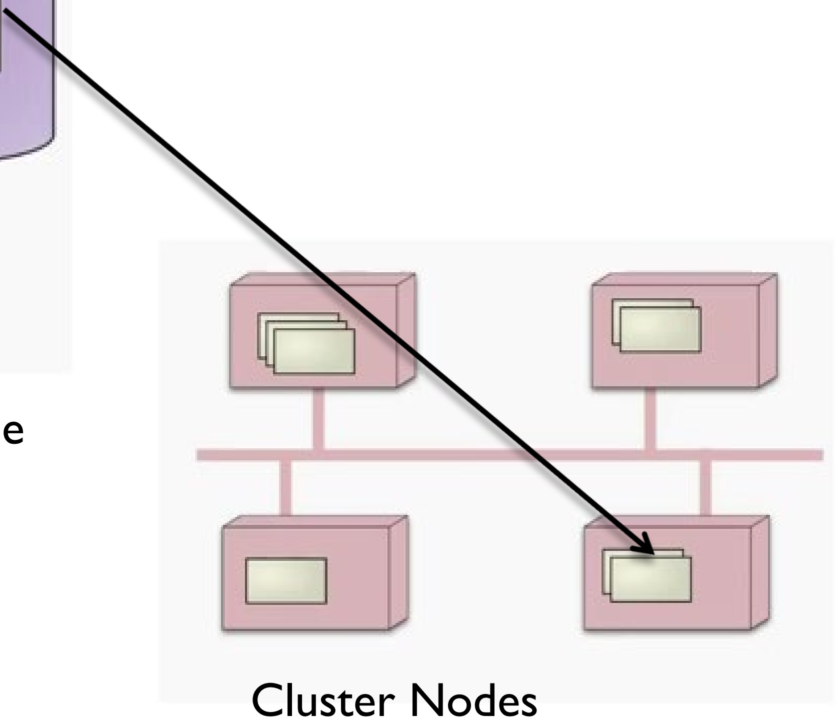
NoSql Common Data Database Types

Key/Value	Graph	Column	Document
Aggregate = Value	NOT an Aggregate Oriented Database	Aggregate = Column Family	Aggregate = Document

Aggregate Oriented Database



Store the aggregate as one unit in the database

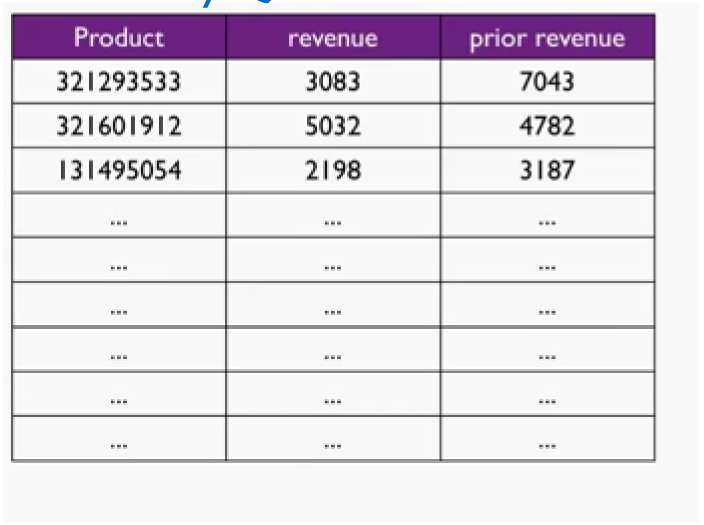


Cluster Nodes

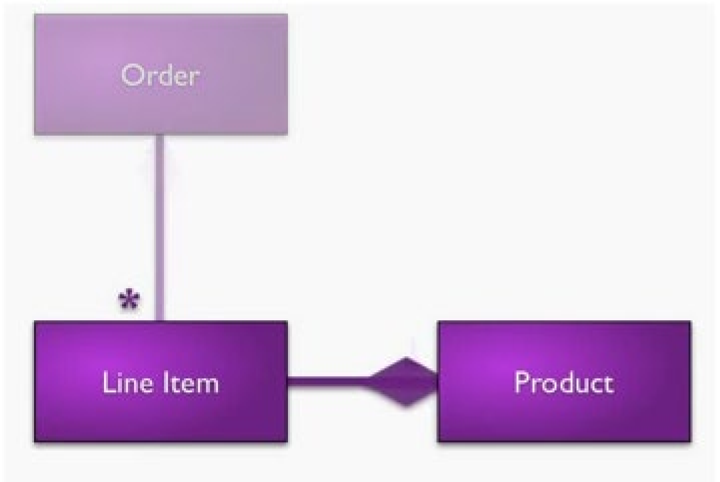
BUT Aggregate Orientation Can Cause Problems!

Want to look at Quarter One Sales for certain products?

Product by Quarter 1 Sales



Product	revenue	prior revenue
321293533	3083	7043
321601912	5032	4782
131495054	2198	3187
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...



Product NOW the root of the aggregate

- However, this data is organised as an **Order Aggregate**!
- NOT as **Product Aggregate**

What would we do in an Rdb?

MongoDB Document Store under the Microscope!

- Supports Rich Documents and are Application-Driven
- No Mongo JOINS
  - Could pre-join documents by nesting (embedding)
  - No Support for Constraints (exception of _id)
- Atomics Operations
  - No Support for Transactions (pre V4.0)
  - Atomic at the document level is the norm
- No Declared Schema
  - Implied Schema
  - Documents in collection will have similar structures

A Document Example in JSON format

```
{
    '_id' : 1,                                                              KEY
    'name' : { 'first' : 'John', 'last' : 'Backus' },
    'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
    'awards' : [
        {
            'award' : 'W.W. McDowell Award',
            'year' : 1967,
            'by' : 'IEEE Computer Society'                                  VALUE
        }, {                                                                (Made up of name values
            'award' : 'Draper Prize',                                       pairs)
            'year' : 1993,
            'by' : 'National Academy of Engineering'
        }
    ]
}
```

MongoDB: Managing Relationships

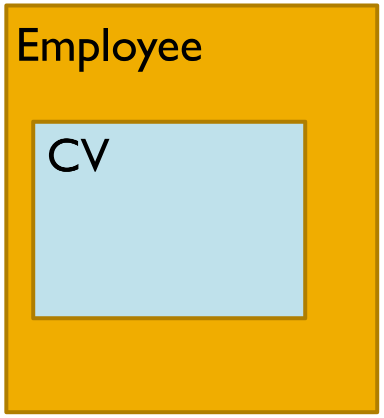
- 1:1 Relationships e.g.
  - Employee : CV
  - Building : Floor Plan
  - Patient : Medical History

- Options:
  - Provide a link in either collection e.g. place link in Employee or CV
  - Embed\Nest one document in another e.g. Embed CV document in the Employee Document
- Considerations
  - Frequency of Access of Documents and Access Pattern
  - How documents grow
  - Atomicity of the data

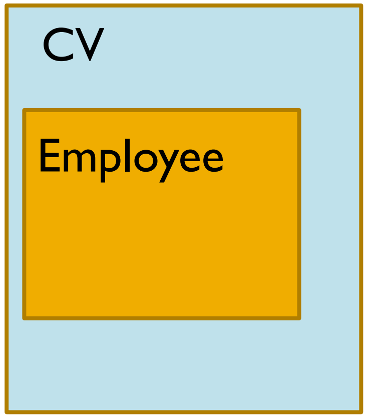
Employee: CV Example

Embedding(nesting) Options:

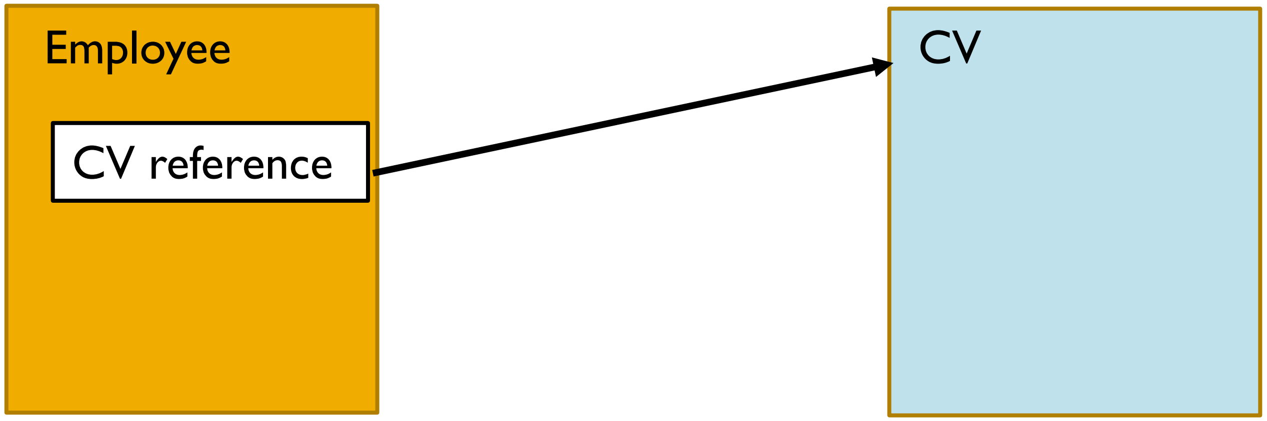
a)



b)

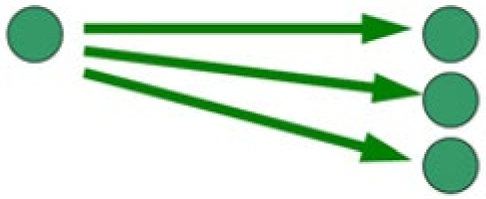


Linking (One option shown):



MongoDb: Managing Relationships

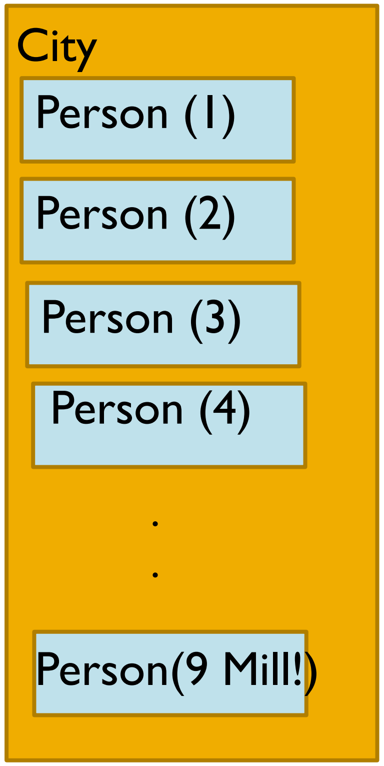
- 1:M Relationships e.g.
  - City : Person ------- One To Many Relationship
  - Blog Post : Comments ------- One To Few Relationship!
- The Options:
  - Embed\Nest a document array on the one side
  - Embed\Nest a document on the many side
  - Through Linking: Insert a link on the Many Side or an array of links on the One side
  - Let us explore
    - One To Many Relationship
    - One To Few Relationship!



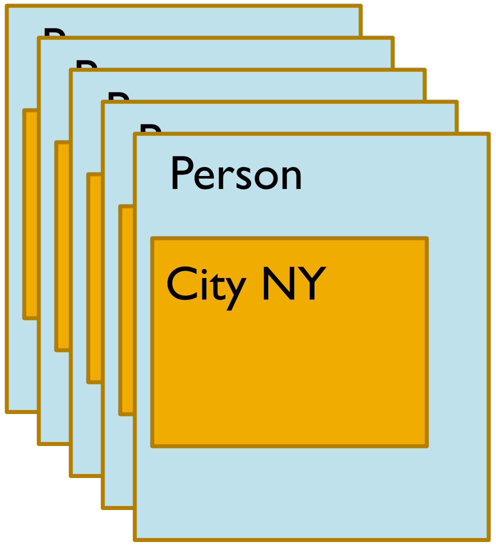
Example:City: Person (One to Many)

Embedding(nesting) Options:

a)



b)

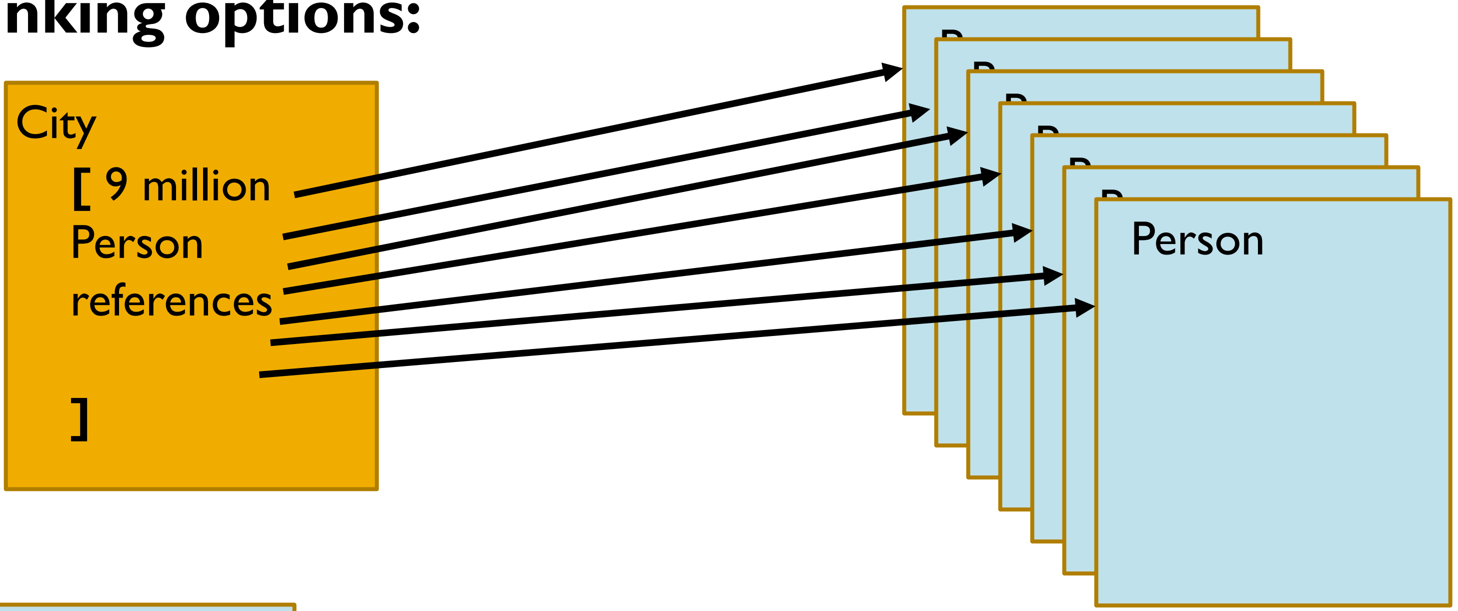


Note: 9 million person docs (e.g. New York [NY])

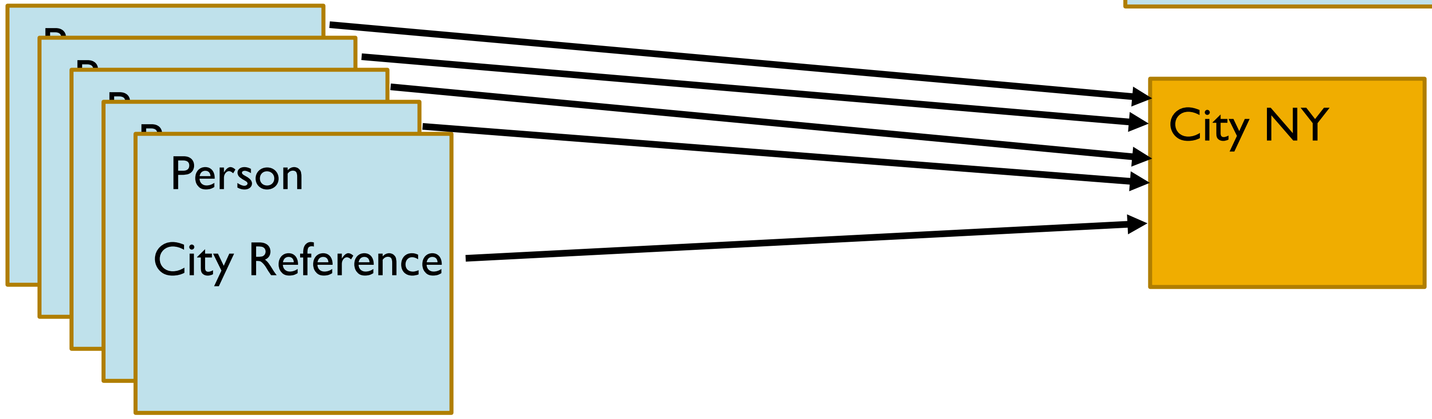
Example:City: Person (One to Many)

Linking options:

a)

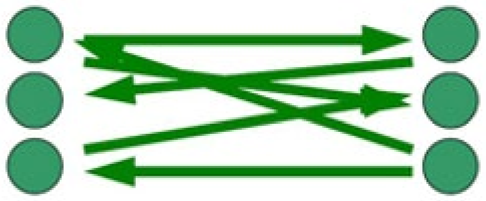


b)



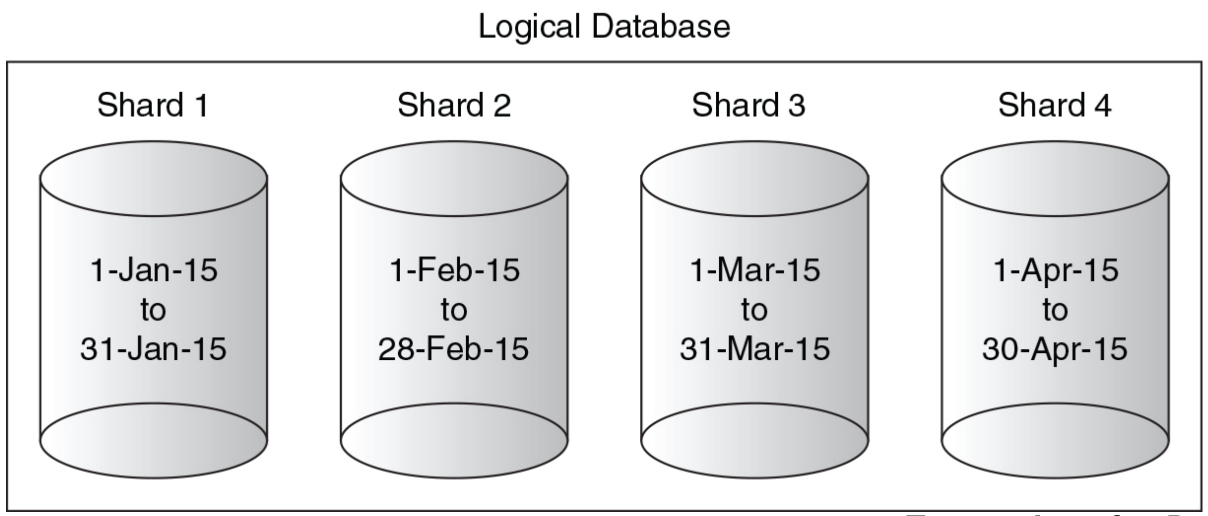
MongoDb: Managing Relationships

- M:M Relationships e.g.
  - Books : Authors ------- Few To Few Relationship
  - Students:Teachers ------- Many To Many Relationship!
- The Options:
  - Through Linking: Insert a link array in one direction or both directions!
  - Embed\Nest a document array on one of the sides of the relationship
  - Let us explore
    - Many To Many Relationship
    - Few To Few Relationship!



Sharding – Horizontal Partitioning

- Sharding in the process of dividing documents in a collection or key values in a Key Value Store in to parts.
  - The parts are know as shards (partitions)
  - A single shard may be stored on multiple servers(if the database is replicated)



Example of a Range Shard\Partition

Horizontal Partitioning Vs. Vertical Partitioning

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

An example of Range Partitioning

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

Shard 1

Also called Sharding

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

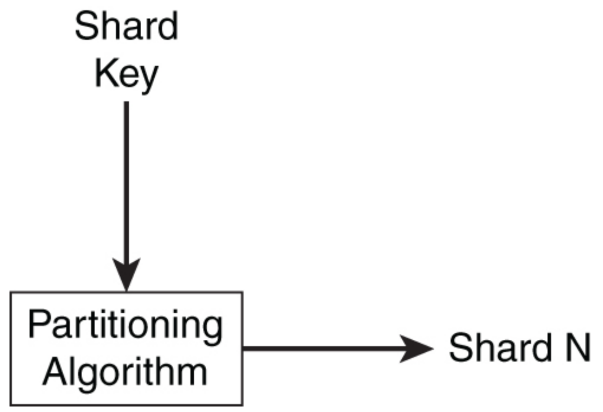
Shard 2

Separating Data With A Shard Key

- A shard key is one or more keys or fields that exist in all documents in a collection that is used to divide documents (in MongoDB) and can be an atomic field
- Common Shard Keys include
  - Unique Document ID
  - Name e.g. product name or customer name
  - Date e.g. Creation Date
  - Category type
  - Geographical region

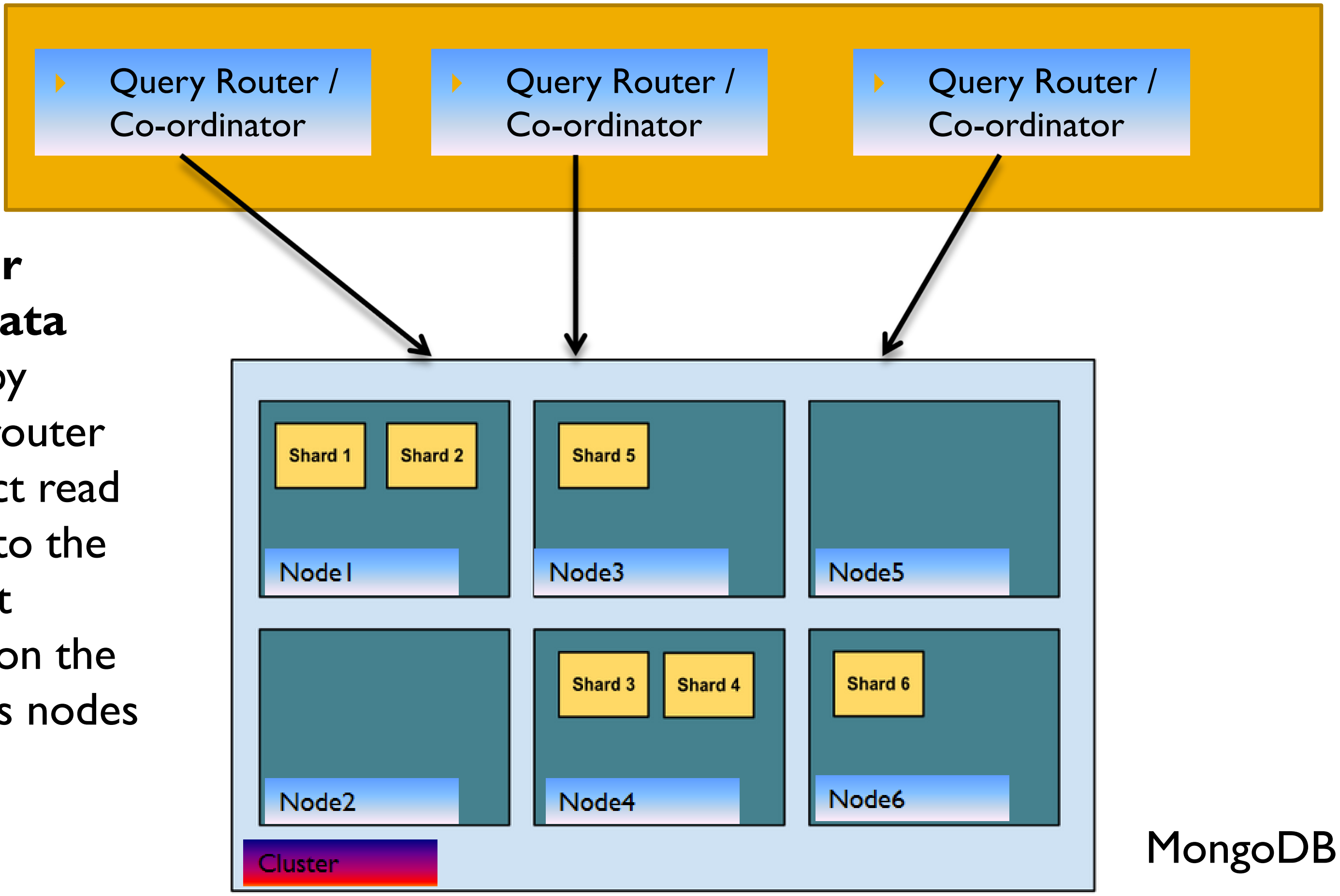
Distributing Data with a Partitioning Algorithm

- **Sharding\Partioning Algorithm**
  - **Shard Key** is input to the partitioning algorithm that outputs a shard



- **Sharding\Partioning Strategies**
  - Range - partition on an ordered set of values
  - List - partition on a list of discrete values e.g. product categories
  - Hash – partition using a hashing algorithm e.g SHA-1 in Riak

Sharding – Horizontal Partitioning


Query Router / Co-ordinator
Query Router / Co-ordinator
Query Router / Co-ordinator
Shard 1
Shard 2
Node1
Shard 5
Node3
Node5
Node2
Shard 3
Shard 4
Node4
Shard 6
Node6
Cluster

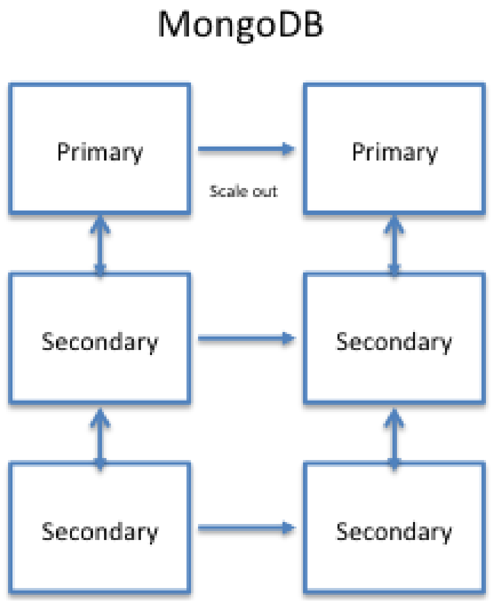
Cluster Metadata
Used by query router to direct read writes to the relevant shards on the cluster's nodes

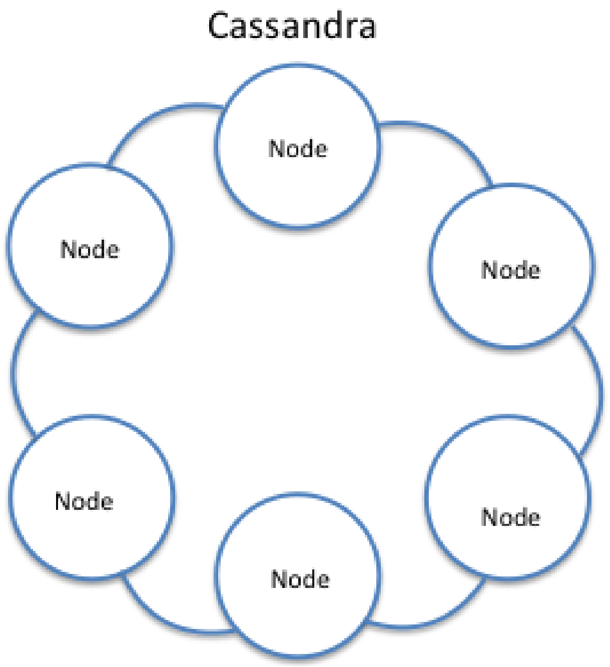
MongoDB

Sharding puts different data on separate nodes, each of which does its own Reads and Writes. How does this differ to Table Partitioning in RDBMS?

Replication Strategies

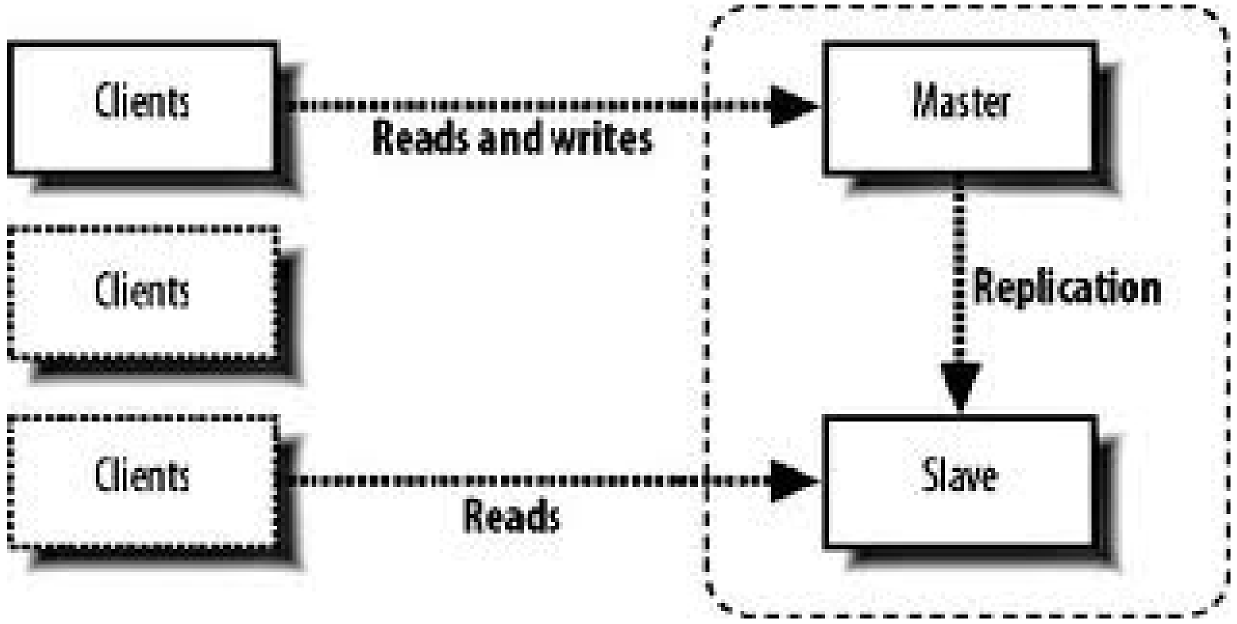
- Replication
  - Provides Redundancy and high availability
- 2 Key Types
  - Master Slave Replication
  - Peer – Peer Replication



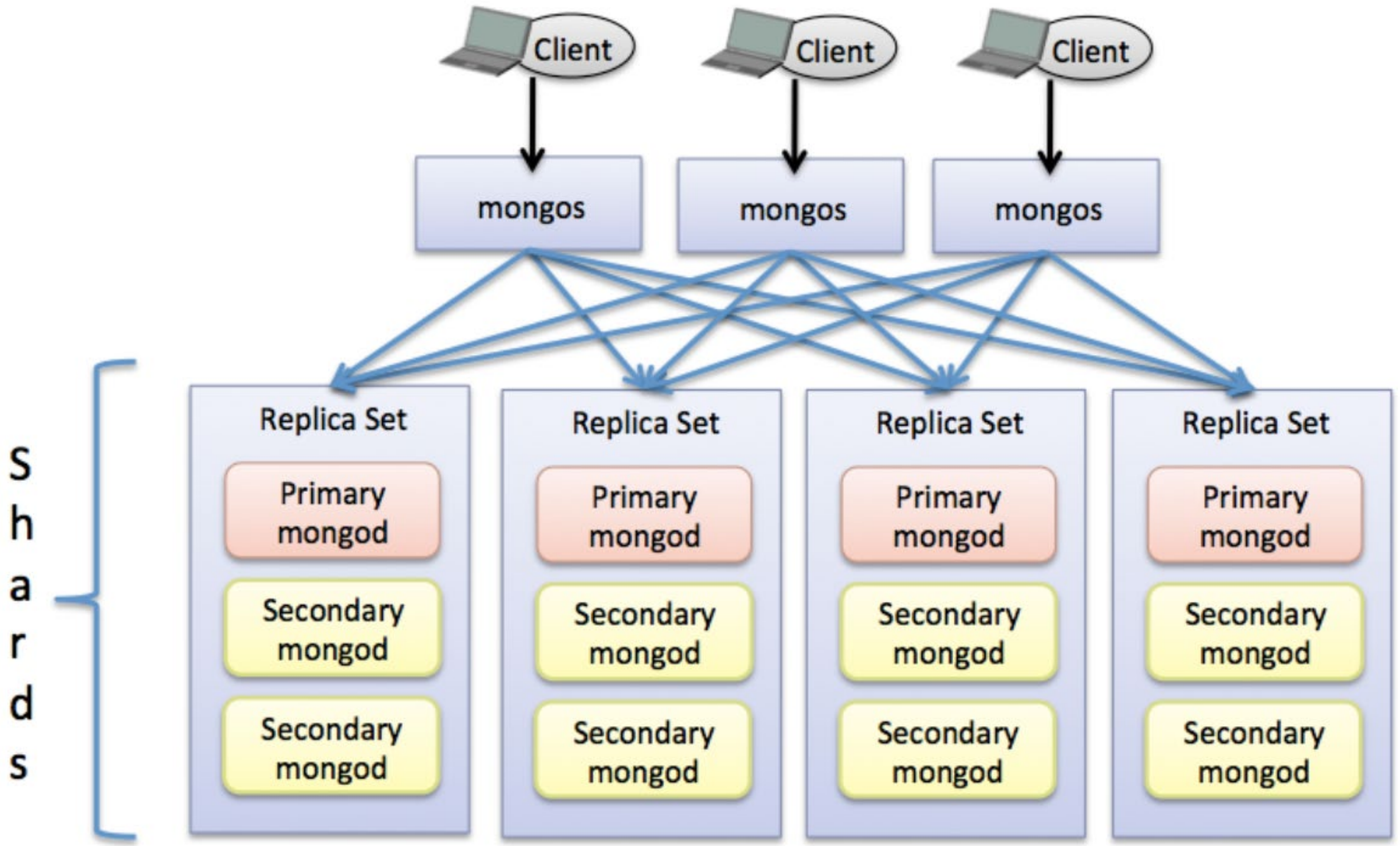


Replication – Master Slave

- You replicate data across multiple nodes through the Master
- One node is master(primary) other nodes are slaves (secondaries)
  - Writes go through the Master; reads may come from Master or Slaves

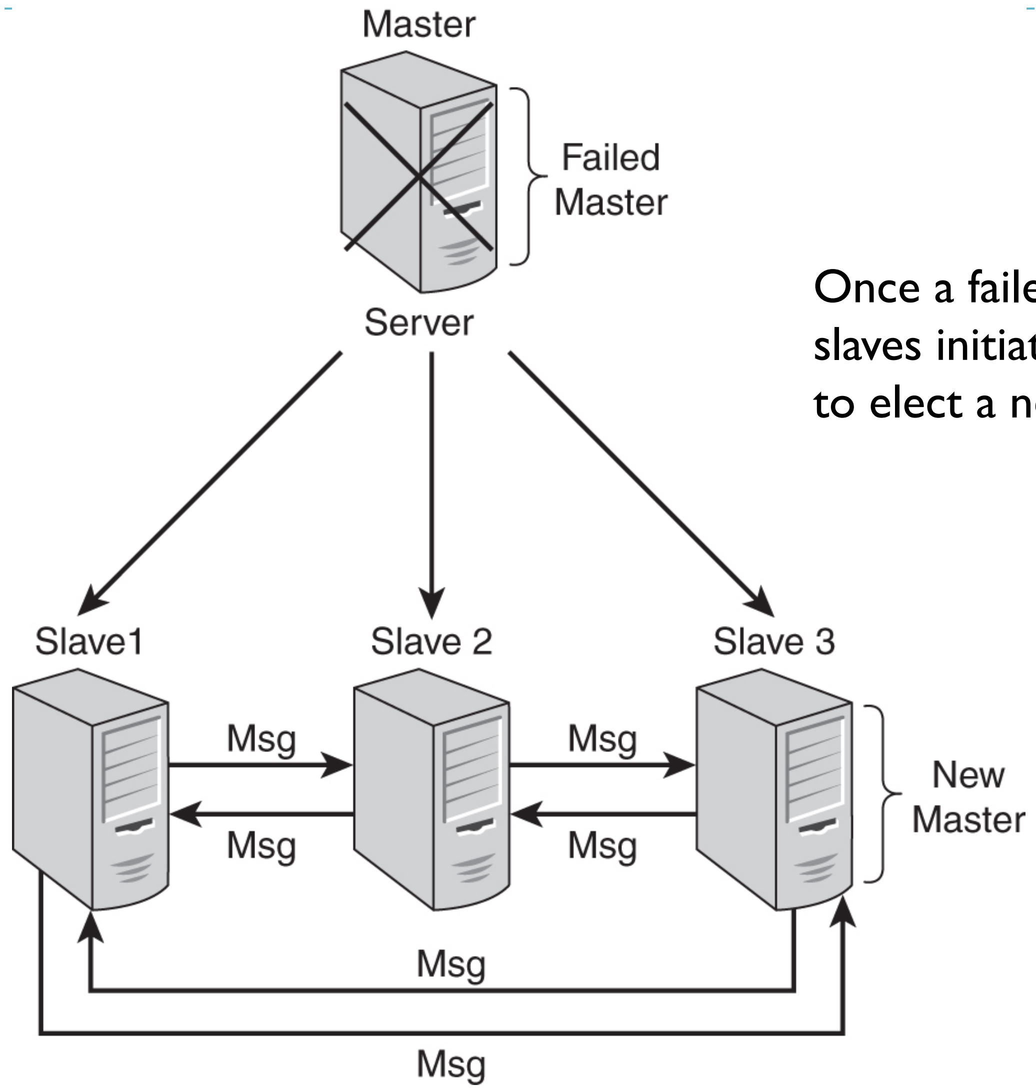


MongoDB Master Slave – An Example



- Replica sets can have up to 50 members.
- Primary and secondaries of a replication should be on separate physical nodes in the cluster

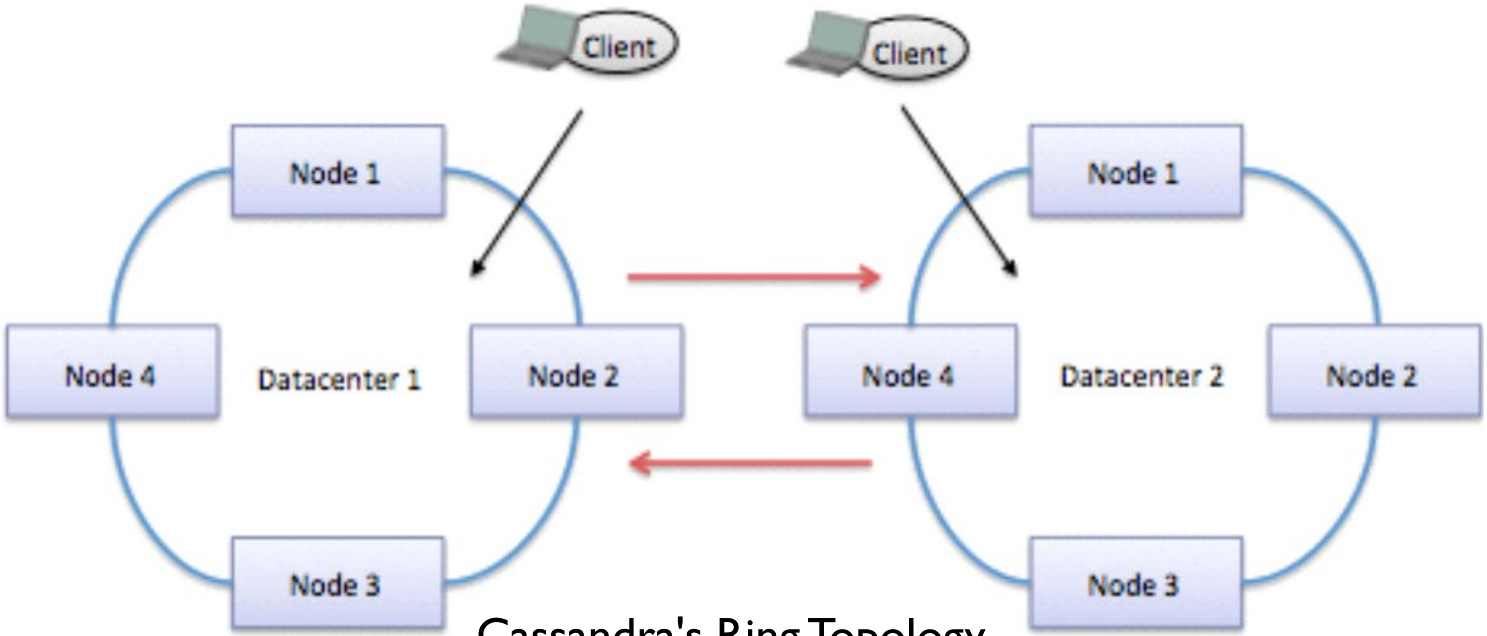
Replication – Master Slave - Failures



Once a failed master node is detected, the slaves initiate a protocol
to elect a new master.

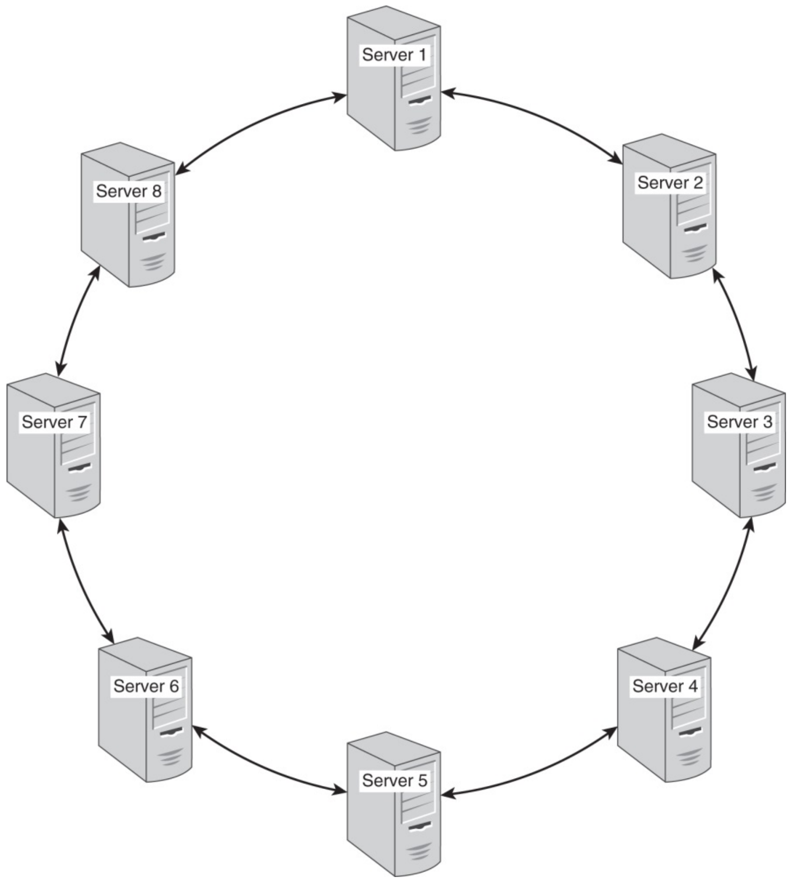
Replication – Peer-Peer (Masterless)

- All replicas have equal weight
  - All replicas can accept writes – no bottleneck
  - Loss of a replica does not prevent Read or Write access
  - Consistency may be an issue. Why?

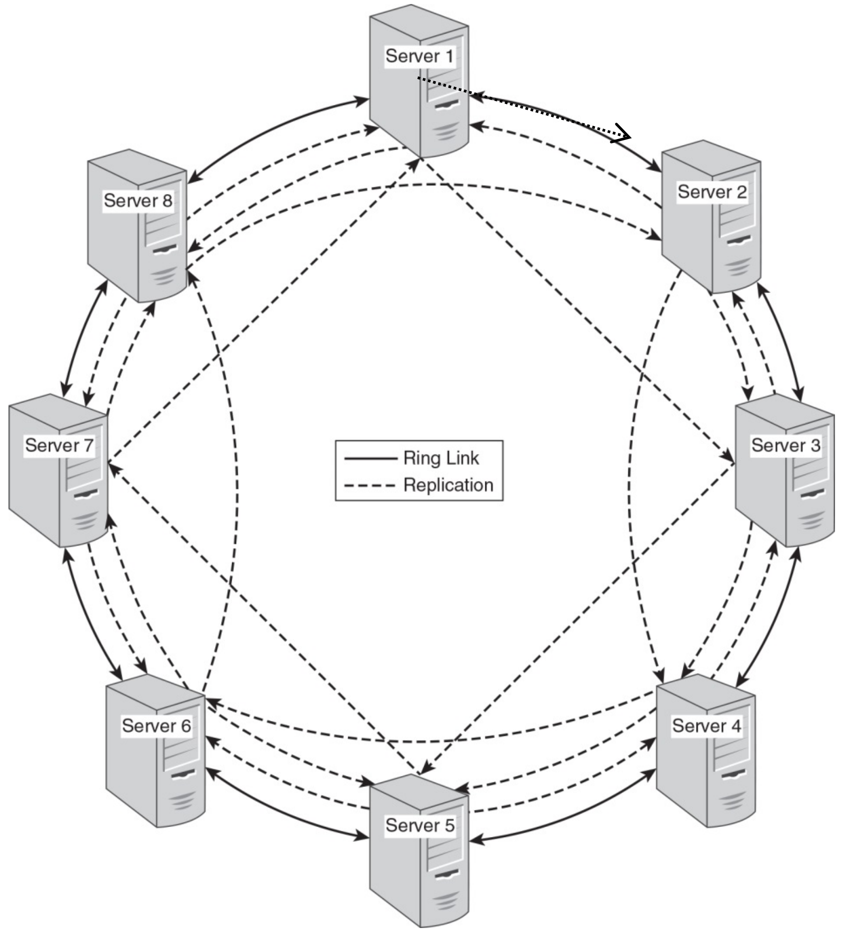


Cassandra's Ring Topology

Replication – Peer-Peer (Master-less)



An eight-server cluster in a ring configuration



An eight-server cluster in a ring configuration with a replication factor of 4.

Courtesy *NoSQL for Mere Mortals* by Dan Sullivan (ISBN: 0134023218) © 2015 Pearson Education, Inc

Brewers's CAP – Theorem

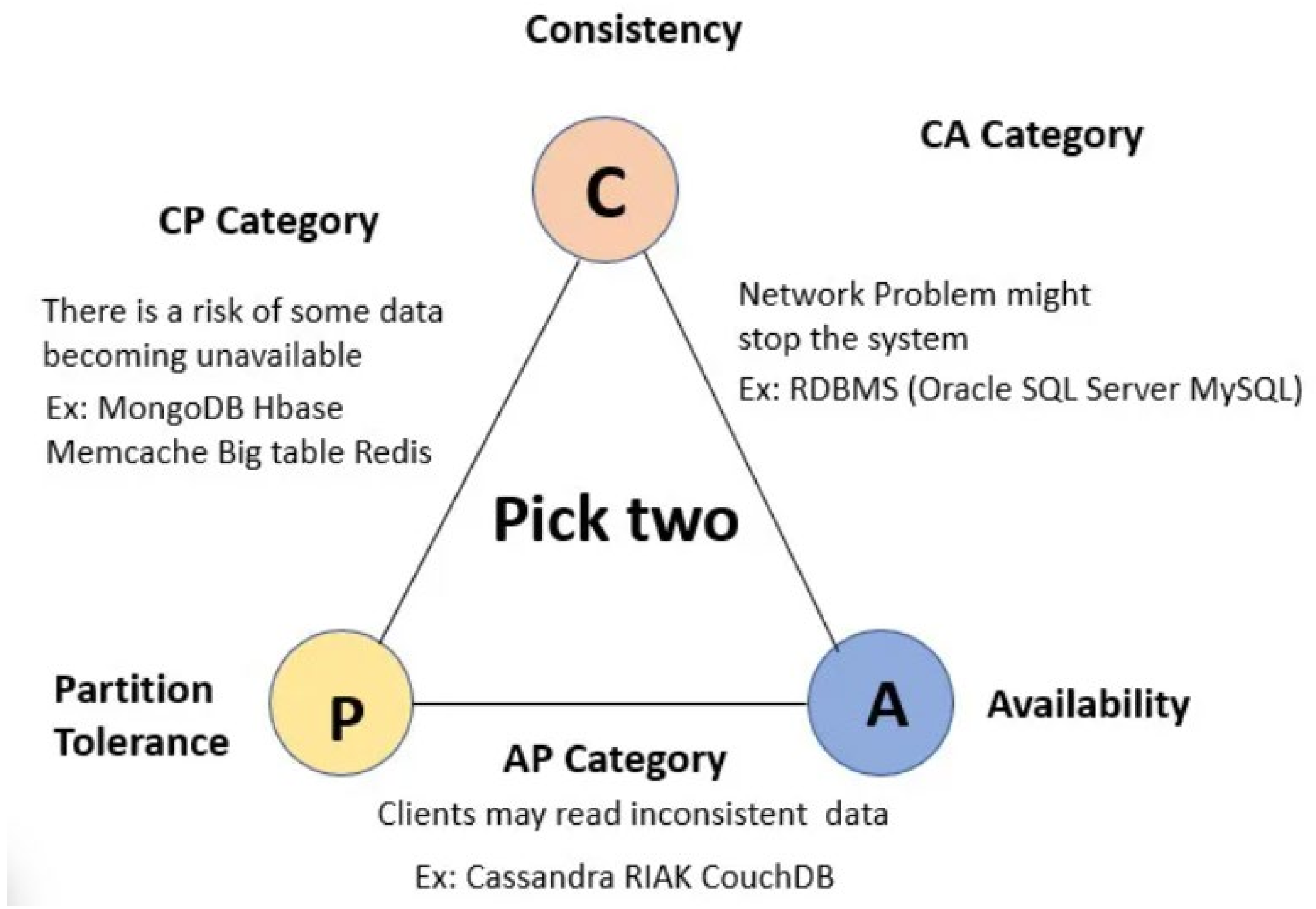
- **Consistency** – All nodes see exactly the same data at the same time. This is the same idea presented in ACID.
- **Availability** especially **high availability** meaning that a system is designed and implemented in that allows it continue operation **if nodes in the cluster fail** or not available for some reason. **All requests will receive a response.**
- **Partition Tolerance -** the ability of the system to **continue operation** in the **presence of network partitions**.
- Brewer alleges that one can at most choose two of the three characteristics
- In practice, CAP is saying that a system may suffer network partitions as distributed systems do, so it is a **trade off** between **Consistency VS. Availability**

CAP theorem NoSQL database types

- **CP database:** A CP database delivers consistency and partition tolerance at the expense of availability.
  - When a partition occurs between any two nodes, the system has to shut down the non-consistent node (i.e., make it unavailable) until the partition is resolved.
- **AP database:** An AP database delivers availability and partition tolerance at the expense of consistency.
  - When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. (When the partition is resolved, the AP databases typically resync the nodes to repair all inconsistencies in the system.)
- **CA database:** A CA database delivers consistency and availability across all nodes.
  - It cannot do this if there is a partition between any two nodes in the system

Source: IBM Cloud Education(2019)

Two CAP Characteictics



There is a risk of some data becoming unavailable

Ex: MongoDB Hbase Memcache Big table Redis

Network Problem might stop the system

Ex: RDBMS (Oracle SQL Server MySQL)

Clients may read inconsistent data

Ex: Cassandra RIAK CouchDB

Consistency Eventually!

- How is consistency achieved in the Relational World?
- Relaxing Consistency
  - Particularly in a distributed NoSQL databases consistency is usually relaxed for high availability and partition tolerance
  - **Eventual Consistency** – replication inconsistencies a fact of life – if no further updates, eventually all nodes will be updated to a single value
  - Relaxing Consistency is a design choice
    - In a distributed database, conflicts are a " normal state of your data"
    - Your application needs to handle them!
  - NoSQL Databases beginning to offer "**tunable**" consistency
- Replication increases likelihood of write-write conflict (peer-peer)
    - This needs to be managed

Distributed Transactions ACID vs BASE

- Distributed synchronous transactions that implement **ACID** semantics tend not to scale well.
  - Network Latency can lead to long running transactions
  - Provides **Strict Consistency** BUT Resources locked for extended periods (e.g. 2 Phase Commit - 2PC)
- **BASE** (Basically available, Soft State, Eventual Consistency) approach popular in NoSQL World
  - Eventual consistency will suffice (stale data okay)
  - Readers will see writes eventually as the system replicates to all nodes
  - Improves responsiveness and availability