NOSQL Databases

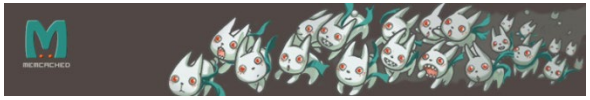
Part 3 – NoSQL Databases

NoSQL Implementation Categories

- Key-Value Databases
- Document Databases
- Column-Family stores
- Graph Databases

Key Value Stores

- Simplest of the NoSQL storage mechanisms
  - Focus on ability to store and retrieve data rather than data structure
  - Get a value for the key; put a value for a key; delete a key from the data store
- Essentially a large hash table
- Key/Value Pair
  - **Key** is unique to identify the data and value;
    - like an Oracle ROWID
    - All data access via the Key (mostly!)
  - **Value** is often opaque; Just stores data without caring what was stored. The value can be a blob, text, JSON,XML etc.
    - To store the value does not have structure
    - Responsibility of the application to understand what was stored; DBMS does not care!
- Can work in a distributed partitioned replicated environment



ORACLE
BERKELEY DB

Key Values and Associative Array

- An Associative Array
  - Like an Ordered array but fewer constraints on keys and values

1	True
2	True
3	False
4	True
5	False

'Pi'	3.14
'CapitalFrance'	'Paris'
17234	34468
'Foo'	'Bar'
'Start_Value'	1

- Key-Values build on this idea
  - Many key-value data stores keep persistent copies of data on long-term storage, such as hard drives or flash devices (e.g. Riak)
  - Some key-value data stores only keep data in memory. (e.g,MemcacheD)

Key And Nothing But The Key!

- Key Design critical to ensure they are easily managed
- RDBMS Keys a sequence of values e.g. 12234, 12235, 12236
  - Keys are not necessarily related to the purpose of information stored
- A prefix could be added: cust12234, cust12235, cust12236
  - Indicates that the keys refer to values about customers
- A more complex naming pattern e.g.
  - cust: 12234 :firstName
  - cust: 12234 :lastName
  - cust: 12234 :shippingAddress
- Developers must create a key naming convention
  - Keys too long – RAM considerations- kv databases are memory intensive
  - Keys too short – could lead to conflicts in key names

The Value

- A Value is an object associated with a key
  - Can be simple integers, floating-point numbers, strings of characters, BLOBs, semi-structured constructs such as JSON objects, images etc.
  - Most key-value databases will have a limit on the size of a value (multiple megabytes to a couple of kilobytes of data)
- Key-value databases typically treat values as atomic opaque units
  - Exceptions to this are key-value databases that provide text-search capabilities; this feature is not a standard part of key-value databases
    - E.g. Riak allows searching using Apache Solr
    - DynamoDB recommends queries using the key. However, allow search scans of the items (their value!)

Key Values and Namespaces

- **Namespace:**
  - A collection of key-value pairs.
  - A Set; duplicate keys are not allowed.
    - Note It is possible to have the same value assigned to multiple keys, but keys can be used only once in a namespace.
  - A namespace could be an ENTIRE key-value database OR
  - A namespace can be at Bucket Level

Database

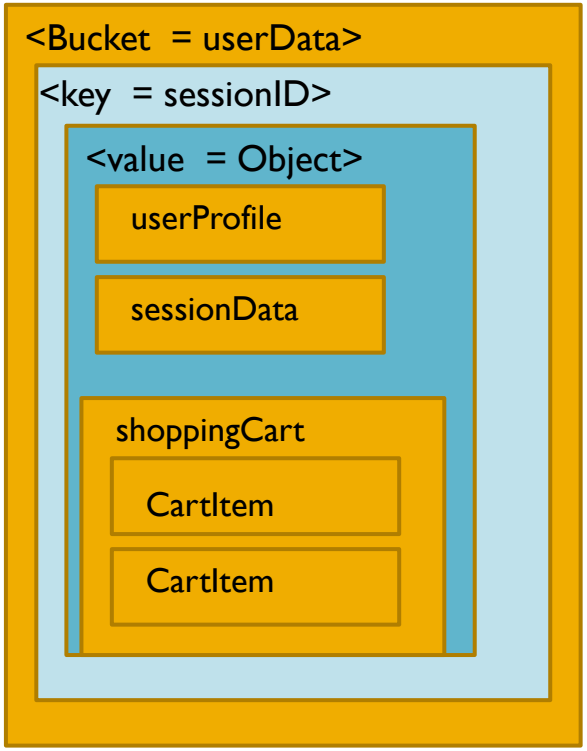
Bucket 1	
'Foo1'	'Bar'
'Foo2'	'Bar2'
'Foo3'	'Bar7'

Bucket 2	
'Foo1'	'Baz'
'Foo4'	'Baz3'
'Foo6'	'Baz2'

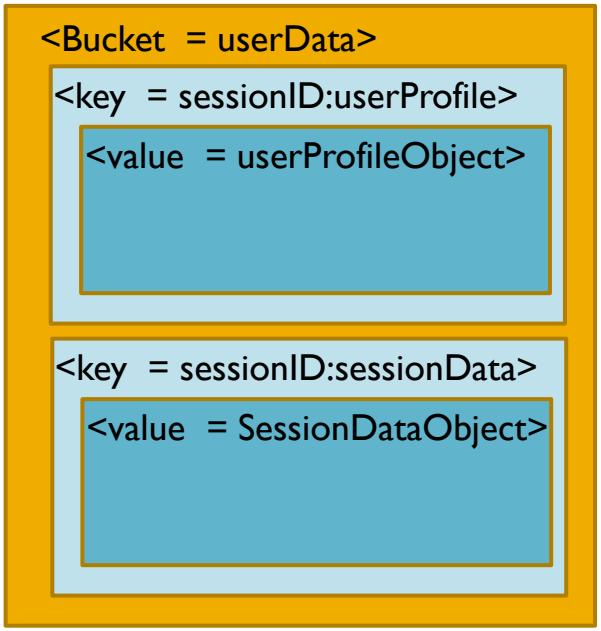
Bucket 3	
'Foo1'	'Bar7'
'Foo4'	'Baz3'
'Foo7'	'Baz9'

  - The essential characteristic of **a namespace** is that it is a collection of key-value pairs that **has no duplicate keys.**

Riak Bucket



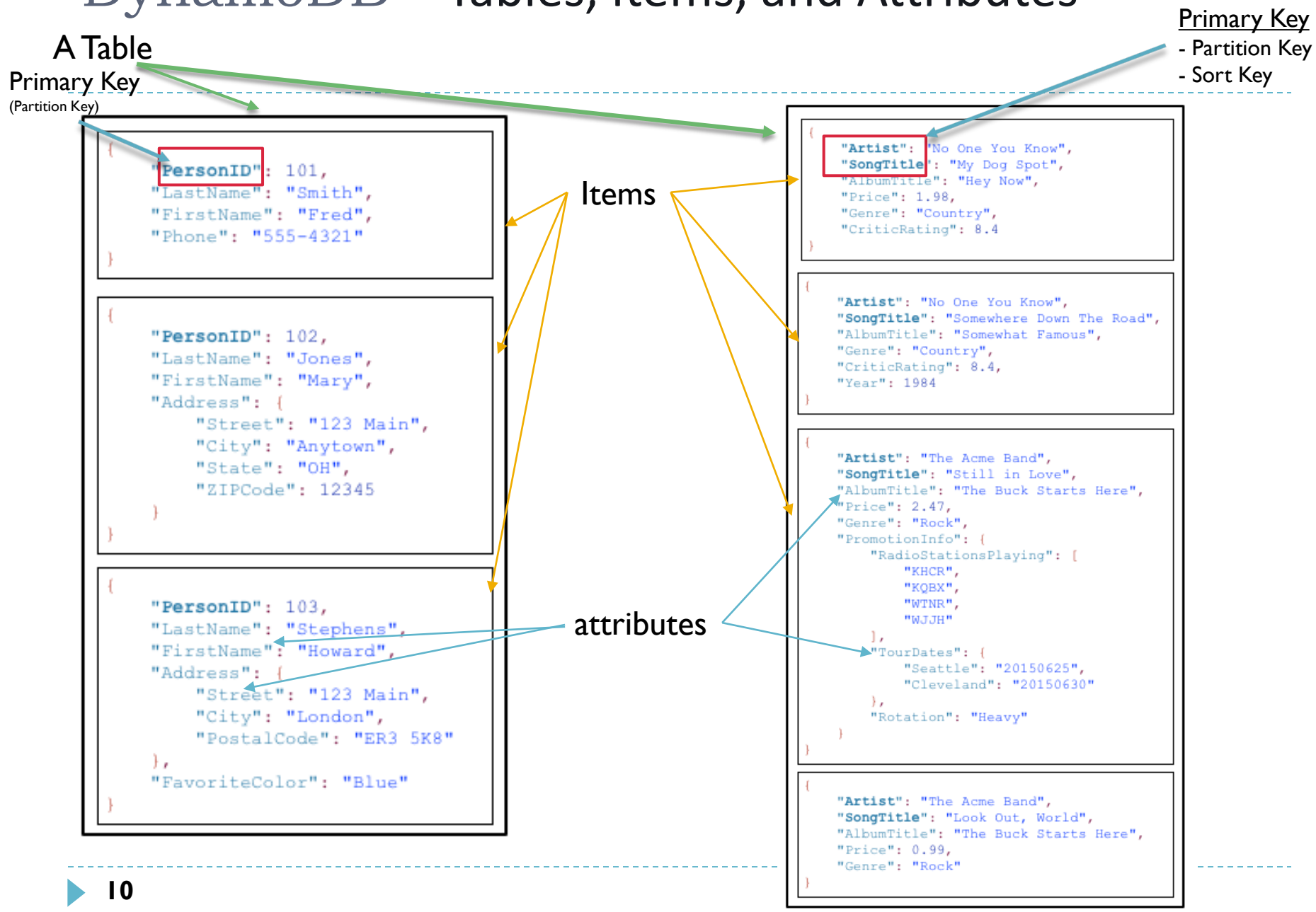
Storing all data in a single bucket



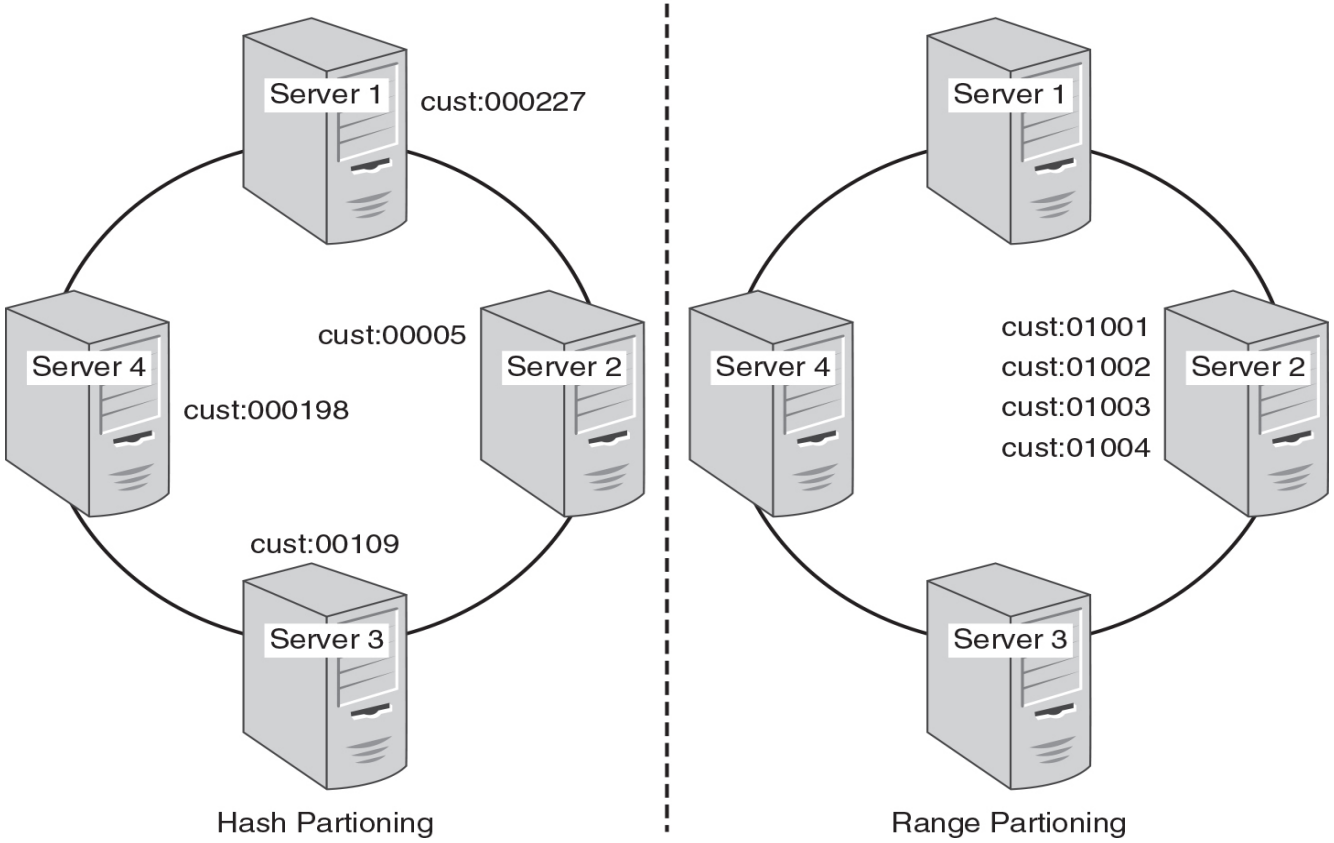
Change the key design to segment the data in a single bucket

Key Design is important in KV stores

DynamoDB - Tables, Items, and Attributes



Common Partitioning Strategies Supported



- Note: Riak and DynamoDB offer Hash Sharding (Partitioning).
- DynamoDB - also offer a combination where a composite key is used
- i.e. Partition Key: Sort Key

Mapping the Hash Key Value to a node

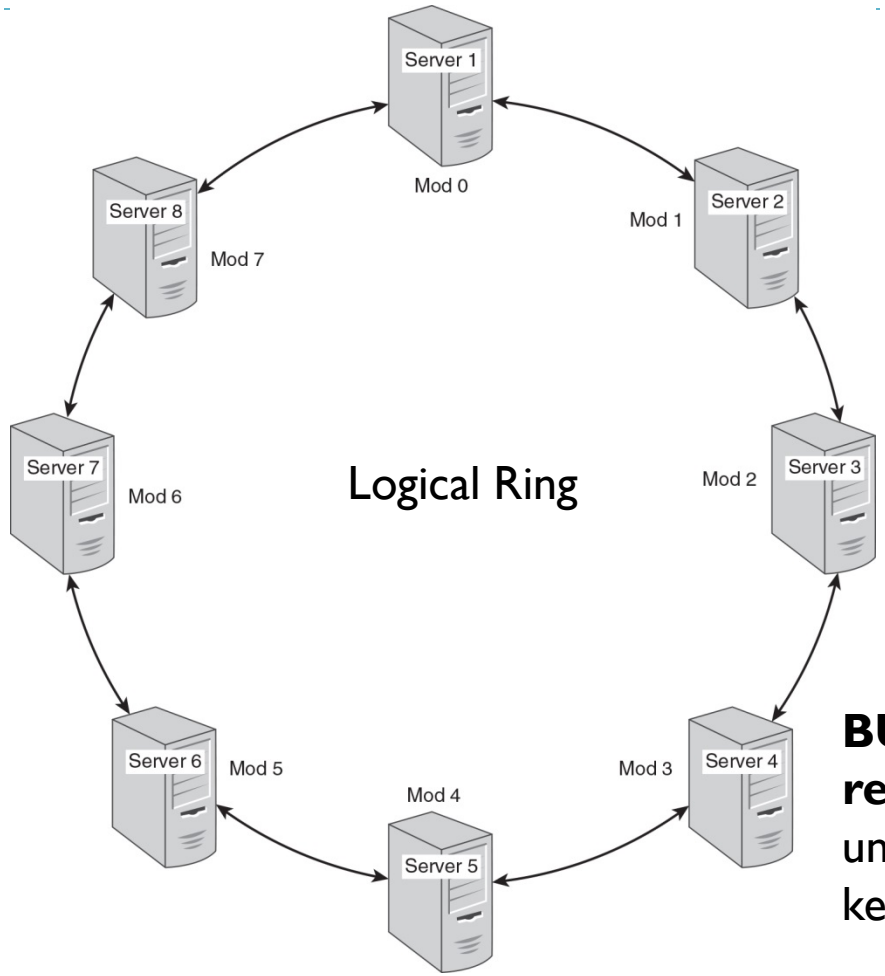


Figure: An eight-server cluster in a ring configuration with modulo number assigned.

Could use **Modulo Arithmetic** to Determine the node the key-value will reside

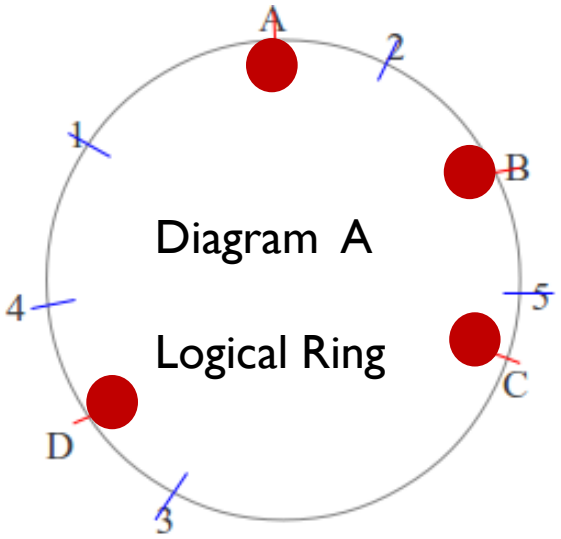
Hash key is passed through a Modulo Function

If we have a hash key of 27 where will the Key Value be placed on the ring?

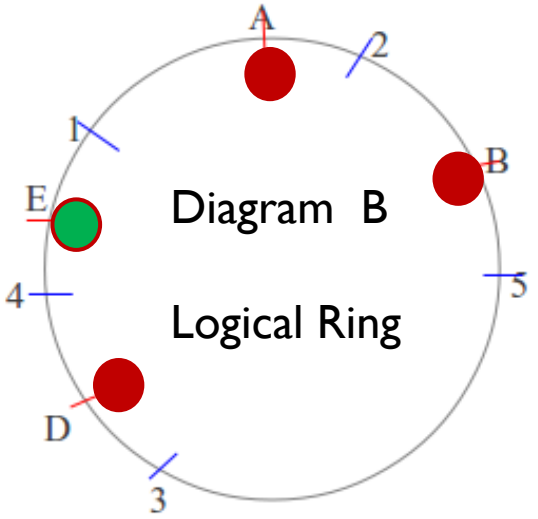
BUT - this works well UNTIL you **add** or **remove** nodes from the ring and become unworkable or serious amount of remapping of key-values to nodes is required.

Consistent Hashing is the answer!

Consistent Hashing: The Concept



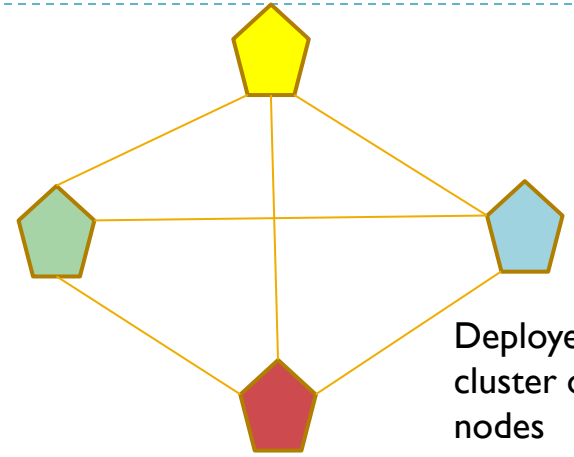
- Node A, B, C and D in ring each with a token
- 5 Key Value objects say 1,2,3,4,5
- Each position in the ring represents hashCode value.



- **2 Scenarios –Removing and Adding nodes**
- Node C is removed
  - Object 5 now belongs on node D
- Node E is added
  - Object 4 now belongs on the node E
- The key value objects 4 and 5 are moved to the nodes

Riak Key Value Store (Amazon DynamoDB inspired)

- Master-less – all nodes are equal
- Nodes can fail -> Hinted Hand-off
- Tunable Consistency offered
- Consistent hashing
  - Evenly distributes data across the cluster
  - No Manual sharding required
  - **160-bit integer keyspace**
  - Divided into fixed number of evenly sized partitions called **vnodes** as there is a vnode process that manages a partition
  - Partitions are claimed by nodes in the cluster
  - Replicas go to the N (=3 default) partitions following the key



Deployed as a cluster of physical nodes

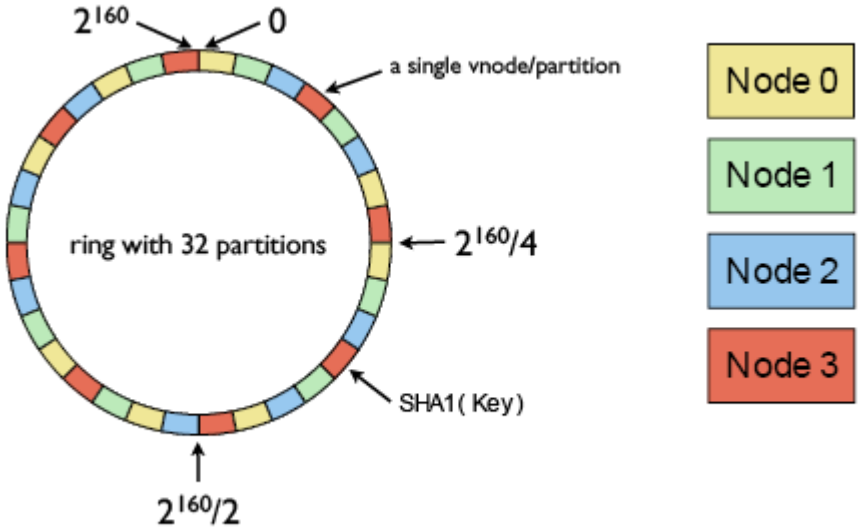
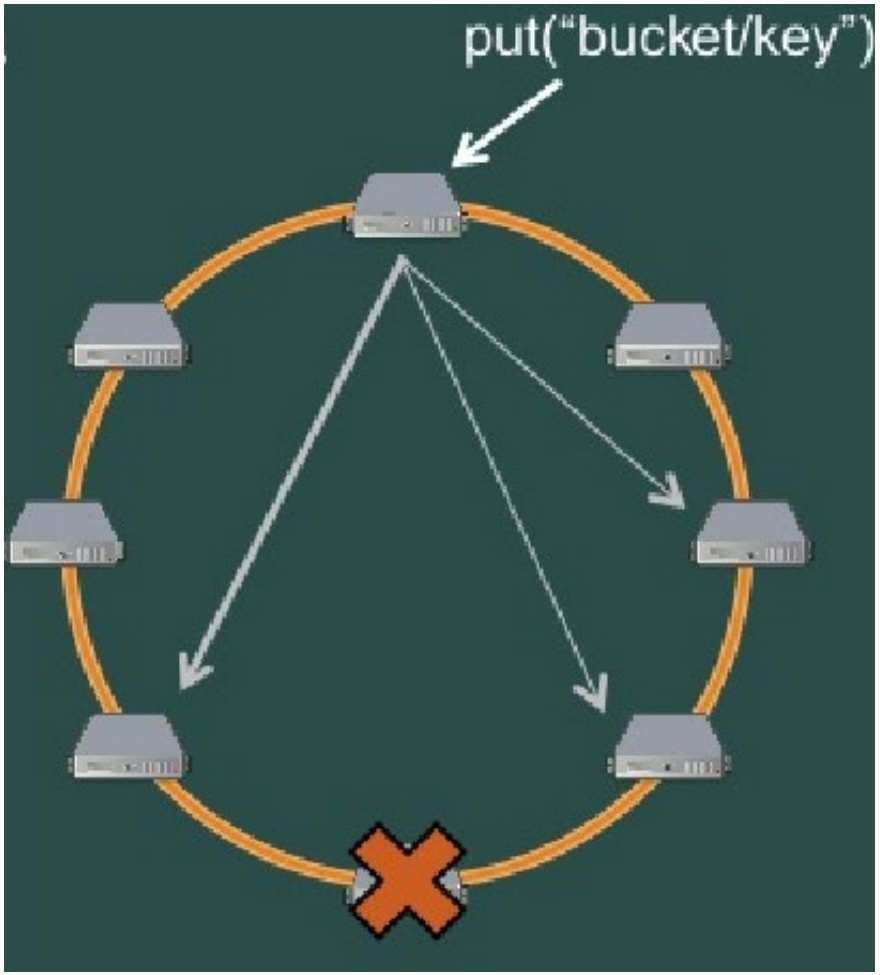


Figure 1: The Riak "Ring"

Hinted Handoff

- Allows Riak nodes to temporarily take over storage operations for a failed node and updates that node with changes when it comes back online.
- Used in other NoSQL databases like Cassandra
- Ensures High Availability



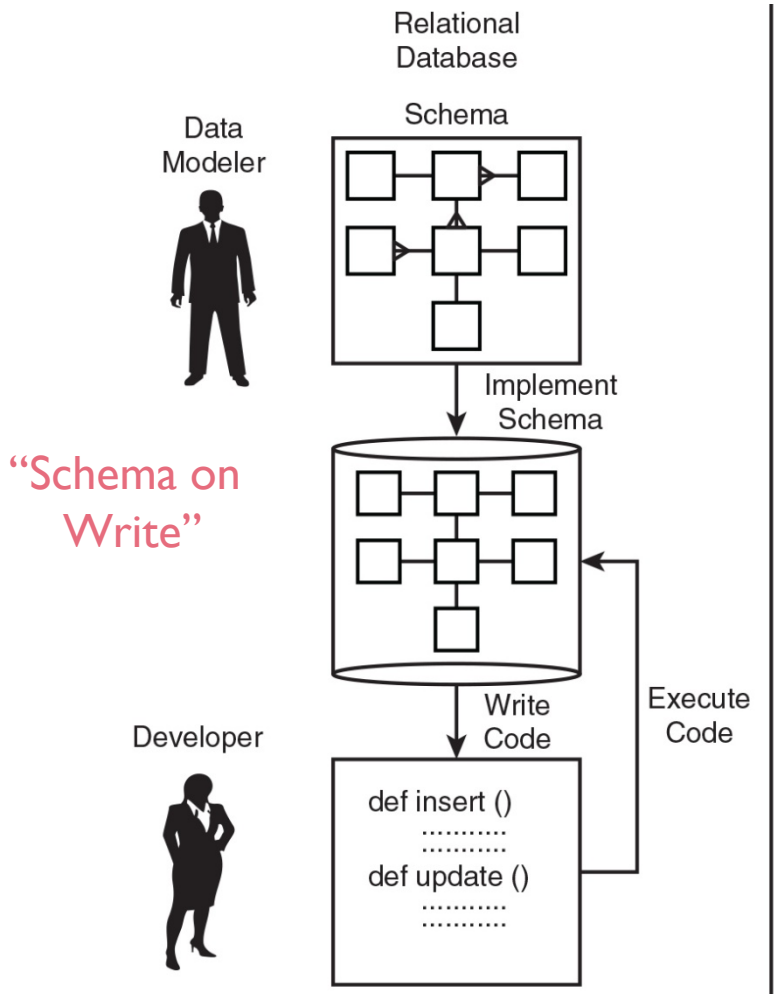
Key-Value Stores - Key Features (1)

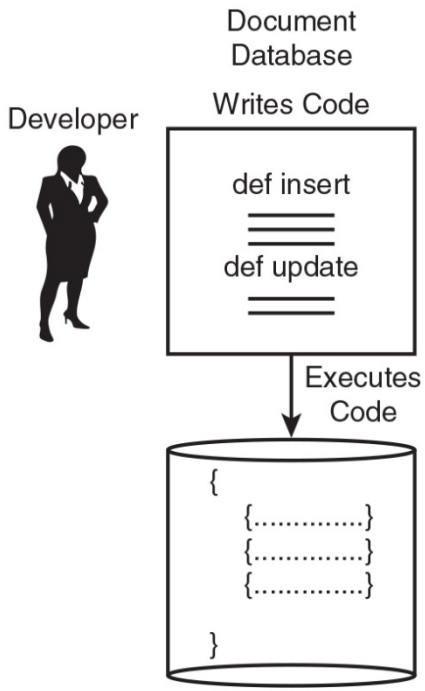
- Consistency
  - In distributed implementations using replication, **eventual consistency** supported
  - How could Write Conflicts (Entropy) be resolved?
    - Last Write wins OR
    - All values returned to client for conflict resolution (manual resolution)
    - Riak uses Read Repair and later versions (1.3 onwards) use Active Anti Entropy (via Merkle trees)
- No ACID Transaction support
  - Generally Speaking - No Guarantees on the writes particularly in distributed replicated environment – RIAK uses quorums for strong consistency
  - Partial Updates are not possible. An update is either a Delete or an Insert
  - Deletes – Using hashing –
    - Deleted values are marked as being deleted
    - Avoids losing existing values that previously collided with the deleted item's key
- Scaling
  - Horizontal Scaling using Replication and Sharding generally supported

Document Database

- A document database is similar in concept to a key/value store except that the values stored are documents.
- A document is a collection of named fields and values, each of which could be simple scalar items or compound elements such as lists and child documents.
  - Fields in a document can be encoded in a variety of ways, including XML, YAML, JSON, BSON, or even stored as plain text.
  - The fields in the documents are exposed to the database management system,
  - An application can query documents by using the document key (but a more common approach is to retrieve documents based on the value of one or more fields in a document
  - Some document databases support indexing to facilitate fast lookup of documents based on an indexed field.
  - Many document databases support in-place updates
  - Usually Atomic at the Document Level
    - Read and write operations over multiple fields in a single document are usually atomic.

NoSQL - Schemaless

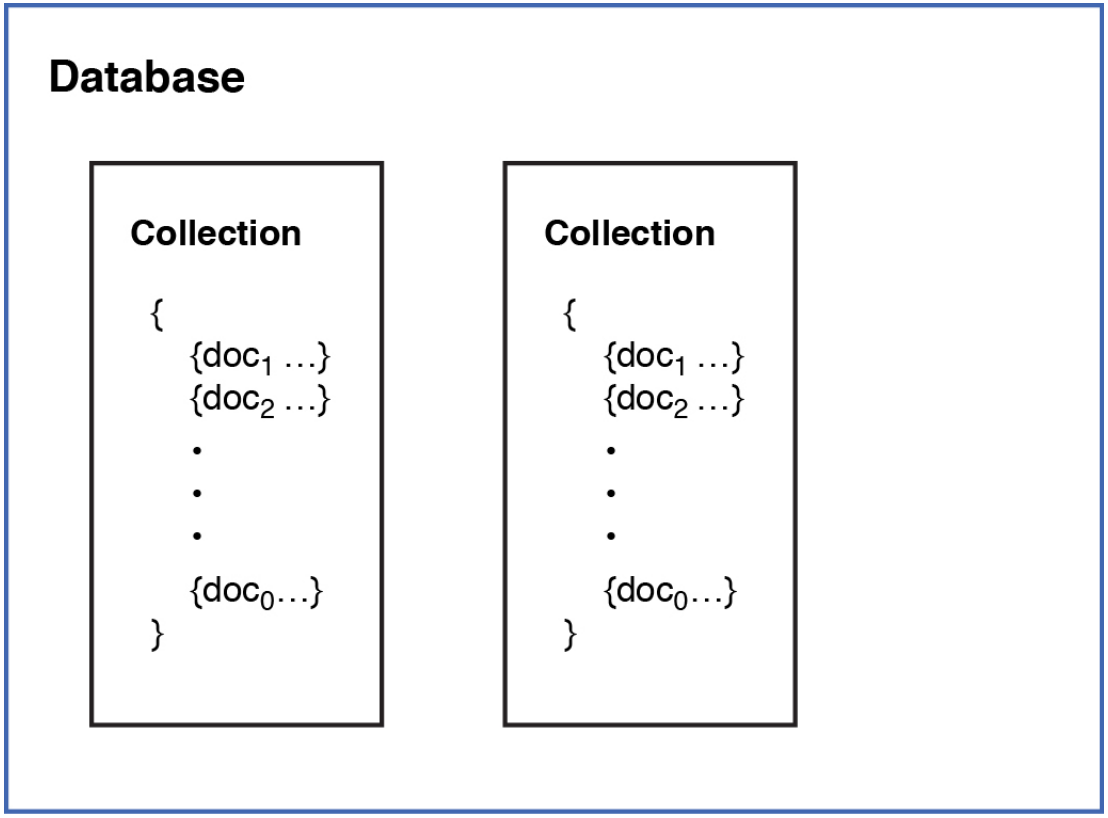




"Schema on Read"

Relational databases require an explicitly defined schema, but document databases do not. (also applies to KV stores and Column Family stores)

Document Database - Conceptual

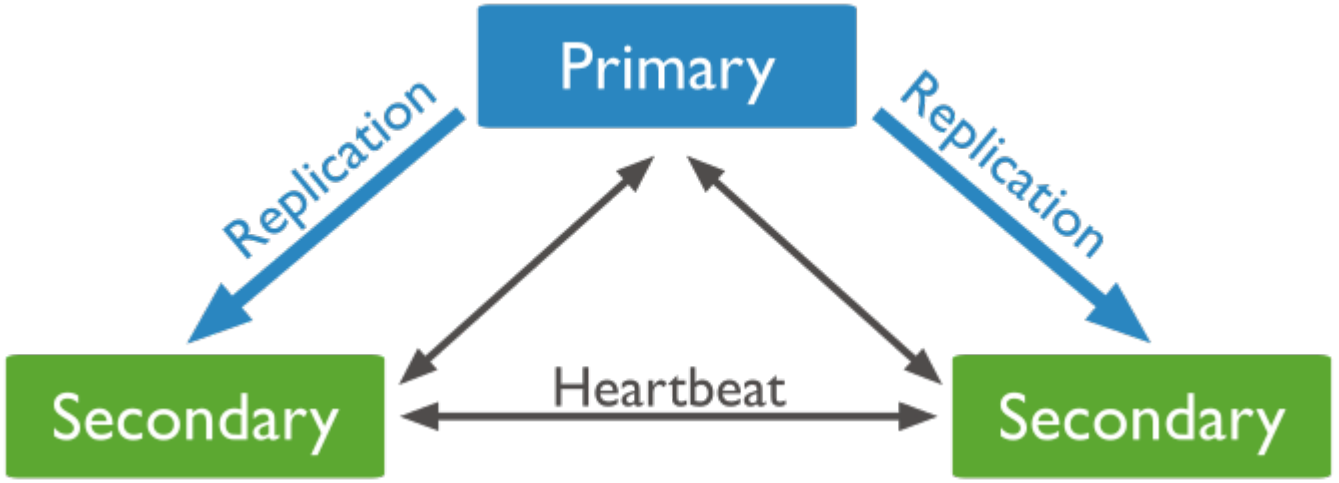


The database is the highest-level logical structure in a document database and contains collections of documents.

MongoDB Terminology

Oracle	MongoDB
Database Instance	MongoDB Instance
Schema	Database
Table	Collection
Row	Document
Rowid	_id
Join (or more correctly Foreign key)	No Joins- 2nd Query; $Lookup

MongoDB Replica Sets



- A replica set is a group of mongod instances that maintain the same data set. A replica set contains several data bearing nodes and optionally **one arbiter node**. Of the data bearing nodes, one and only one member is deemed the **primary node**, while the other nodes are deemed **secondary nodes**.
- Source: MongoDB Reference Manual

Key Features MongoDB(1)

- Consistency
  - Applicable for operations on a single document default
  - Uses Master/Slave replication
  - Eventual Consistency to Strong Consistency.
- ACID Transactions not supported generally
  - Transactions at the single-document level are known as atomic transactions (default)
  - Atomic Transactions across multiple documents now possible with version 4.04
  - Durability and Consistent Writes
    - Main levels of **WriteConcern** are offered
      - **0** => write operation not acknowledged
      - **1** => acknowledgement from the primary only required - DEFAULT
      - **Majority** => acknowledgement from majority in replica set required
      - **n** => acknowledgement from n members of replica set required
    - Main levels of ReadConcern offered
      - **"local"** – return the instances most recent data available even if not persisted to a majority of replica set members and may be aborted
      - "**majority**" – read data that has been written to a majority of replica set members and thus cannot be aborted(only with WiredTiger storage engine)

Column-Family Database

- A column-family database organizes its data into rows and columns;
  - Aka *Wide Column databases or Extensible Record databases*
  - In its simplest form it can appear very similar to a relational database, at least conceptually.
  - The real power is its denormalized approach to structuring sparse data.
  - You can think of a column-family database as holding tabular data comprising of rows and columns, but you can divide the columns into groups known as *column-families*.
  - Each column-family holds a set of columns that are logically related together.
  - Can be schema-less **but can also define a schema!**

Column Family Store: Cassandra

Oracle	Cassandra (peer-peer replication)
Database Instance	Cluster
Database Schema	Keyspace
Table	Column Family (aka Table)
Row	Row
Rowid	Row Key
Column (same for all rows)	Column (can be different per row)

Conceptual Structure of Data in a Column Family

CustomerID	Identity Column Family	
1	Title	Mr
	FirstName	Mark
	MiddleName1	William
	LastName	Hanson
2	Title	Ms
	FirstName	Lisa
	MiddleName1	Sarah
	MiddleName2	Louise
	LastName	Andrews
3	FirstName	Walter
	LastName	Harp

A simple Column Family with a set of column names

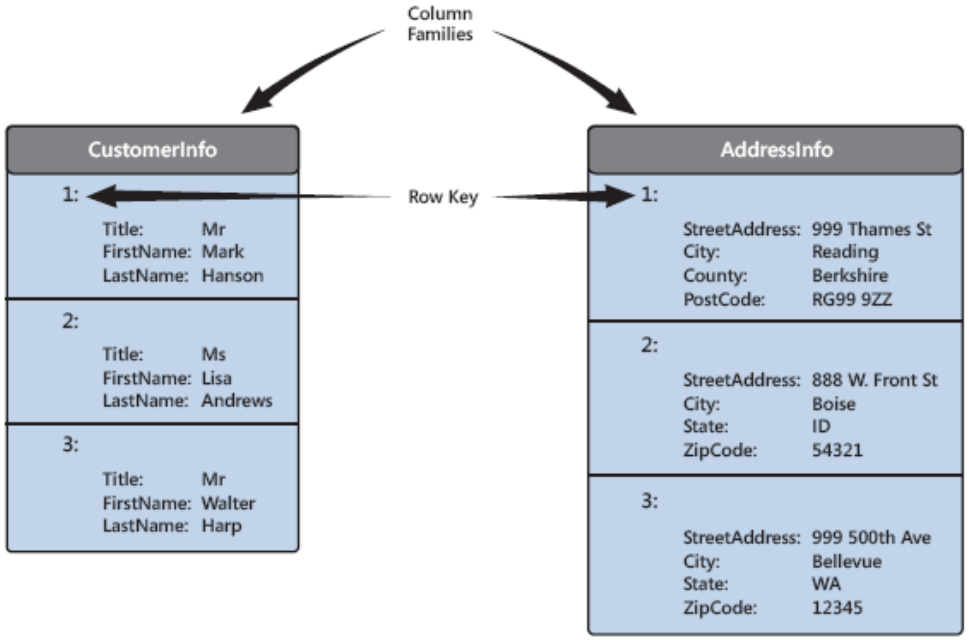
Row Key	Column Families			
CustomerID	CustomerInfo		AddressInfo	
1	CustomerInfo:Title	Mr	AddressInfo:StreetAddress	999 Thames St
	CustomerInfo:FirstName	Mark	AddressInfo:City	Reading
	CustomerInfo:LastName	Hanson	AddressInfo:County	Berkshire
			AddressInfo:PostCode	RG99 922
2	CustomerInfo:Title	Ms	AddressInfo:StreetAddress	888 W. Front St
	CustomerInfo:FirstName	Lisa	AddressInfo:City	Boise
	CustomerInfo:LastName	Andrews	AddressInfo:State	ID
			AddressInfo:ZipCode	54321
3	CustomerInfo:Title	Mr	AddressInfo:StreetAddress	999 500th Ave
	CustomerInfo:FirstName	Walter	AddressInfo:City	Bellevue
	CustomerInfo:LastName	Harp	AddressInfo:State	WA
			AddressInfo:ZipCode	12345

Different rows can have different fields and the data does not conform to a rigid layout.

From a physical perspective, Column Families are generally stored separately

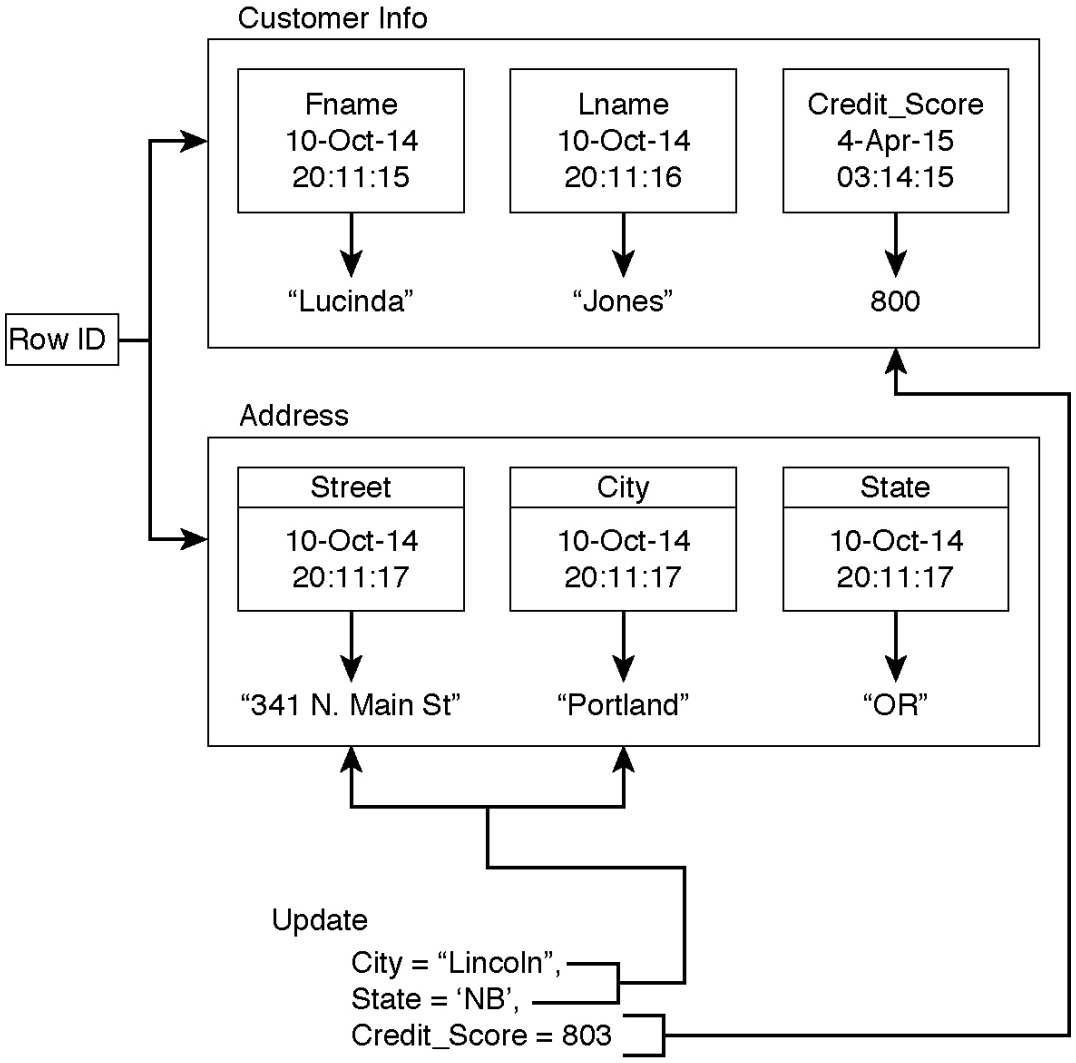
Structure of Data in a Column Family contd

- The data for a single entity that spans multiple column-families will have the same row key in each column-family.



- In a :ions for the part of a **row in a single column-family** are **usually atomic**, although a **number of implementations provide atomicity across the entire row** (spanning multiple column-families e.g. Hbase, BigTable) as well.

Atomicity: Google BigTable An Example



Read and write operations **are atomic at the Row Level.** All columns are read or written or none are.

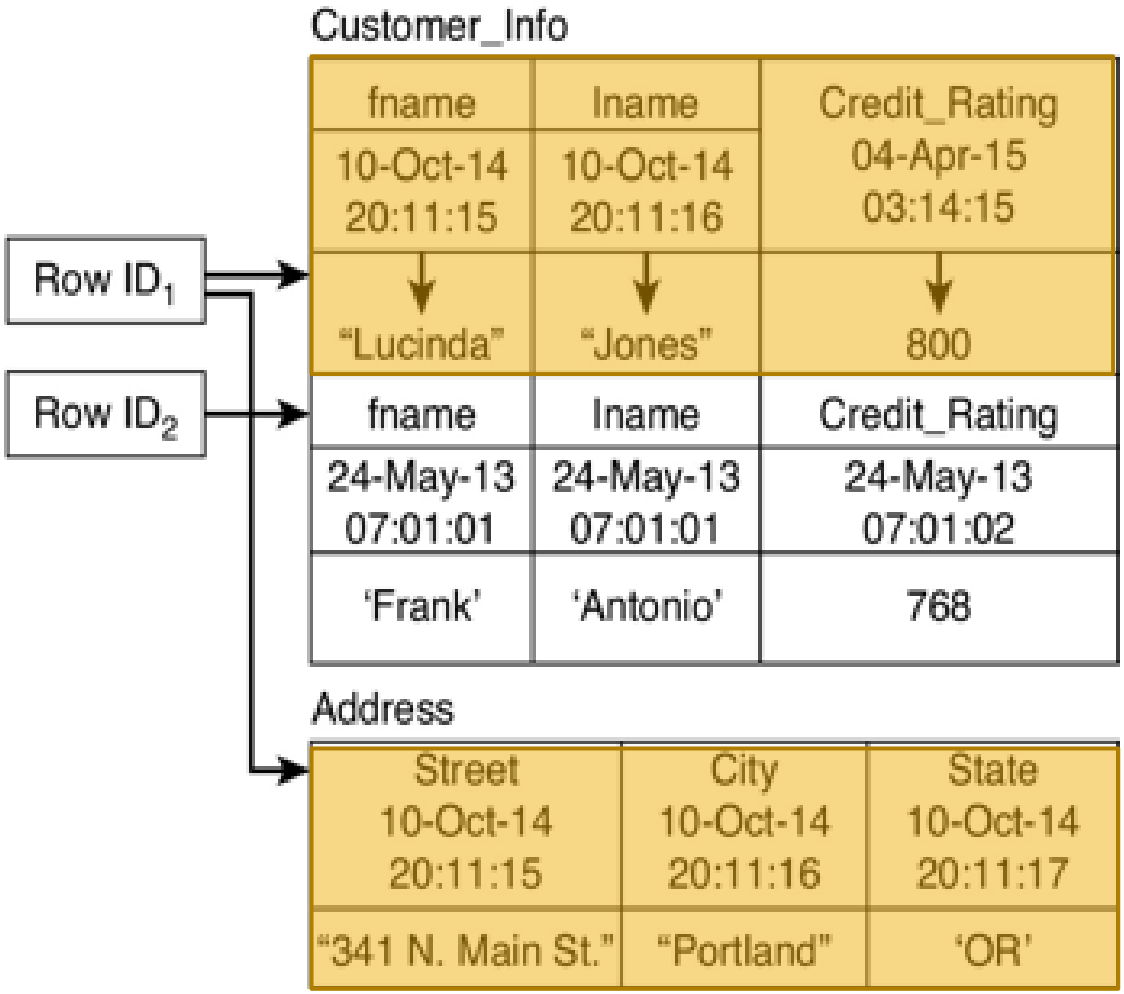
Note: values are not Overwritten but appended

Document DB VS Column Family DB

Document Database:

```
{fname: 'Lucinda',
lname: 'Jones',
Credit_Rating: 800,
Address:
        {Street: '341 N. Main St.',
        City: 'Portland',
        State: 'OR'
        }
{fname: 'Frank',
lname: 'Antonio',
Credit_Rating: 768
}
```

Column Family Database:



Wide Column Family Example

StockTicker	StockPrices Column Family	
ABBT	01/01/2013 11:54:16	130
	01/01/2013 11:58:22	131.5
	01/01/2013 12:02:58	132
	01/01/2013 12:03:18	135
	01/01/2013 12:08:57	133.5
	...	
BAXD	01/01/2013 11:58:32	11.5
	01/01/2013 11:59:42	10.5
	01/01/2013 12:08:30	9
	01/01/2013 12:09:24	9.5
	...	
EFCD	01/01/2013 12:01:17	55
	01/01/2013 12:07:12	54.5
	01/01/2013 12:10:32	55
	01/01/2013 12:19:14	55.5
	01/01/2013 12:20:40	56
	...	
FAMI	01/01/2013 11:57:19	228
	01/01/2013 12:09:45	225
	01/01/2013 12:15:48	227
	01/01/2013 12:19:55	27.5
	01/01/2013 12:22:09	228.5
	...	

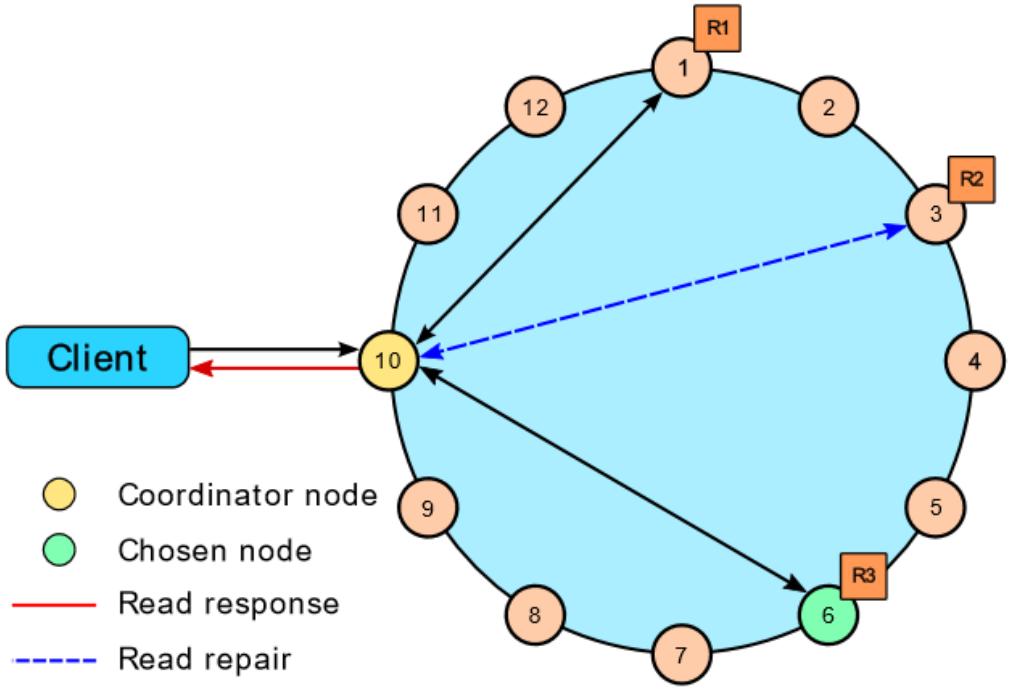
What are the column names?

What is the Row Key?

Example of a Dynamic Column Family

Column Families can contain thousands of columns.

Cassandra
Logical Ring – **Read Request** – an example



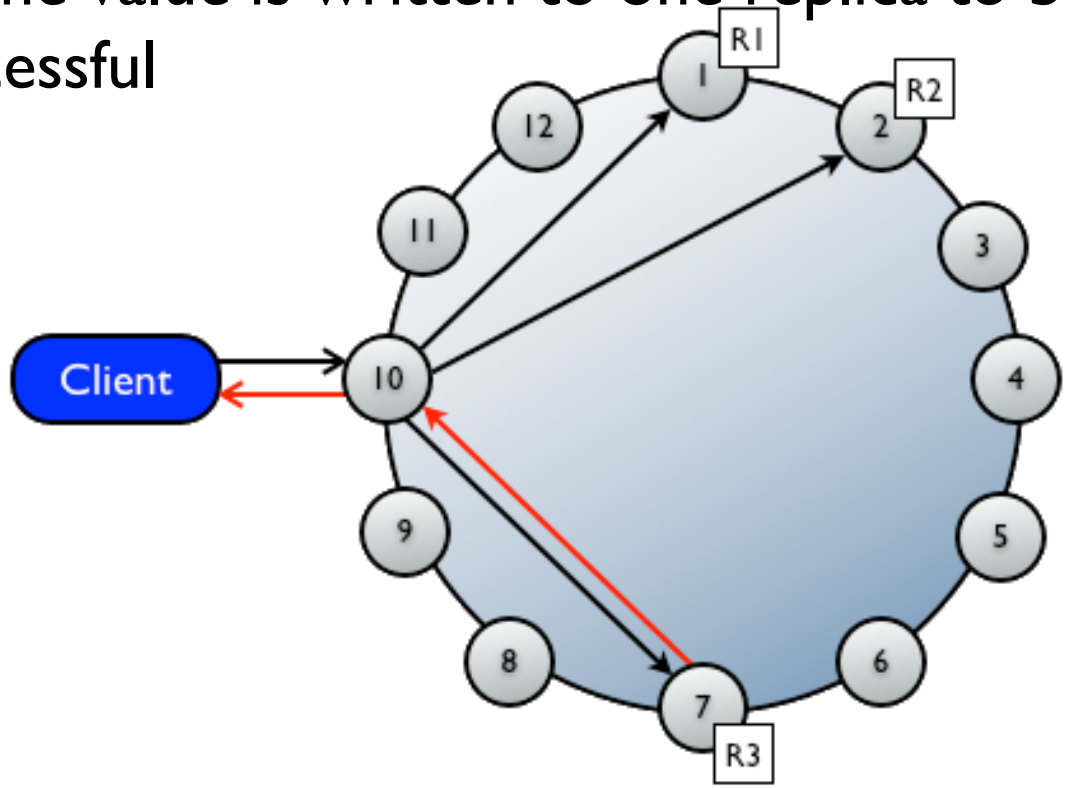
3 Replica Nodes with a consistency level set to QUORUM (majority = 2 here)

Returns the value with most recent timestamp

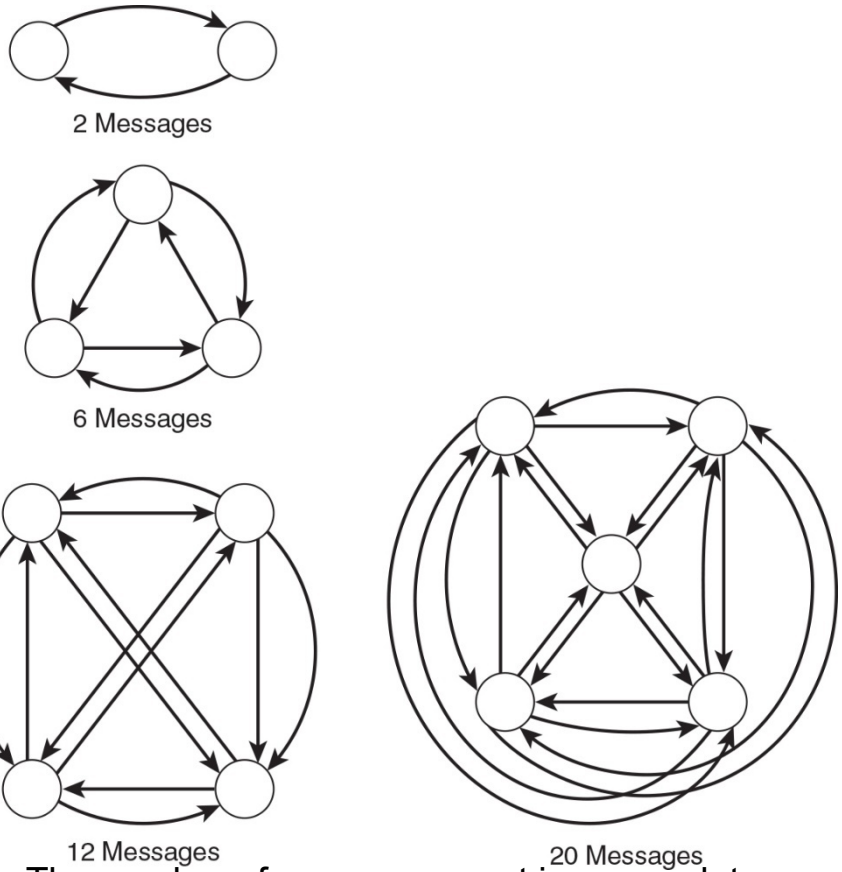
- Logical Ring of vNodes
- Gossip protocol use for inter node communications
- Tunable Consistency supported
- Supports **Consistent Hashing** as well as **Read Repair**
- Supports **Hash Partitioning** (default) and Range Partitioning
- CQL, a SQL like language has become the primary API

Cassandra
Logical Ring – **Write Request An Example**

- Write Consistency Level of ONE
  - Replicas will eventually be updated
  - There are other Consistency Levels
- Ensure that the value is written to one replica to be deemed successful



Gossip Protocol

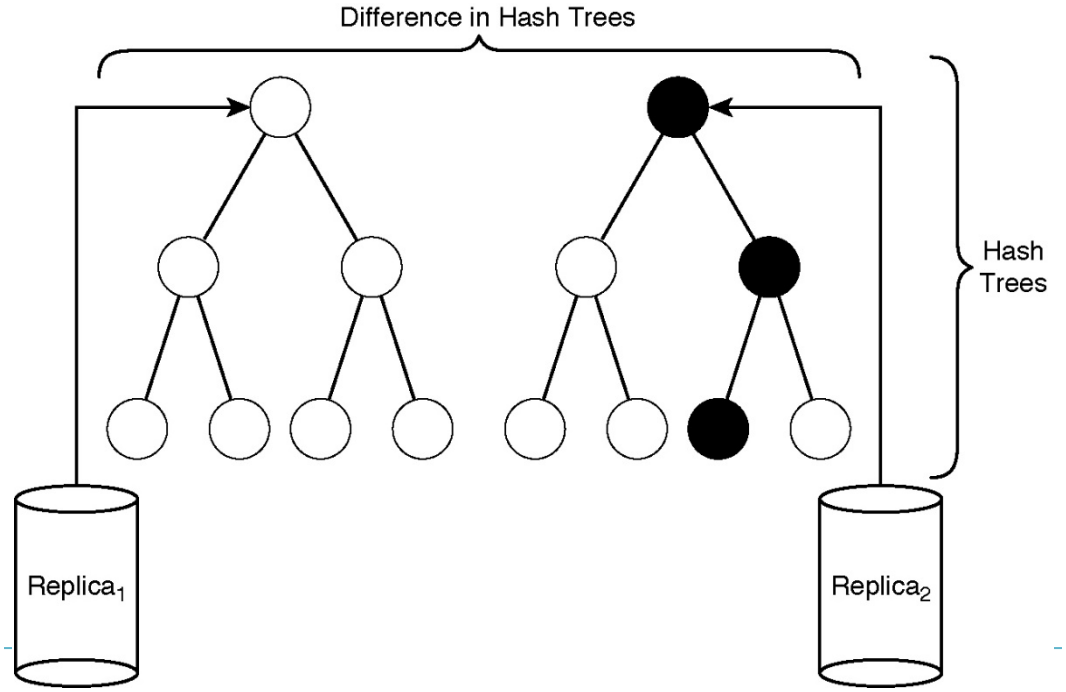


The number of messages sent in a complete server-to-server communication protocol grows more rapidly each time a server is added to the cluster.

- If N is the number of servers the N X (N-1) is the number of messages needed to update all servers with information about all other servers! Very Expensive!
- A more efficient method of sharing information is the Gossip Protocol

Anti-Entropy

- The process of detecting differences in replicas
- 2 common Methods
  - Read Repair
  - Hash trees, or Merkle trees,
    - Allow for rapid checks on consistency between two data sets.



Key Features

- Querying
  - Can query particular column family rather than all column families that make up a row
    - Remember – Key/Value Stores often returns the whole entity
- Transactions
  - Piece-wise updates supported
  - Not the best solution for systems that require full ACID transactions
  - Atomicity at the Row level
    - Cassandra – atomic at the row level i.e. a single read/write will either succeed or fail
- Cassandra Peer-Peer Logical Ring for read and writes
- Sharding\Partitioning and Replication supported
    - Eventual Consistency as well as Stronger Consistency supported where all replicas must respond for transactions to be successful.
      - Read and Write Quorums also supported

Key Features

- Appropriate when
  - Real-time random\read access capability is required and data being stored has some defined structure
  - Applications that that require the ability to **always write**
  - Applications with dynamic fields and can tolerate short-term inconsistencies in replicas
  - Applications with **truly large volumes** of data (**100's of TB**)
  - Applications that are geographically distributed over multiple data centres
- Not Appropriate when
  - Joins are required
  - Systems that require ACID transactions
  - Binary data needs to be supported

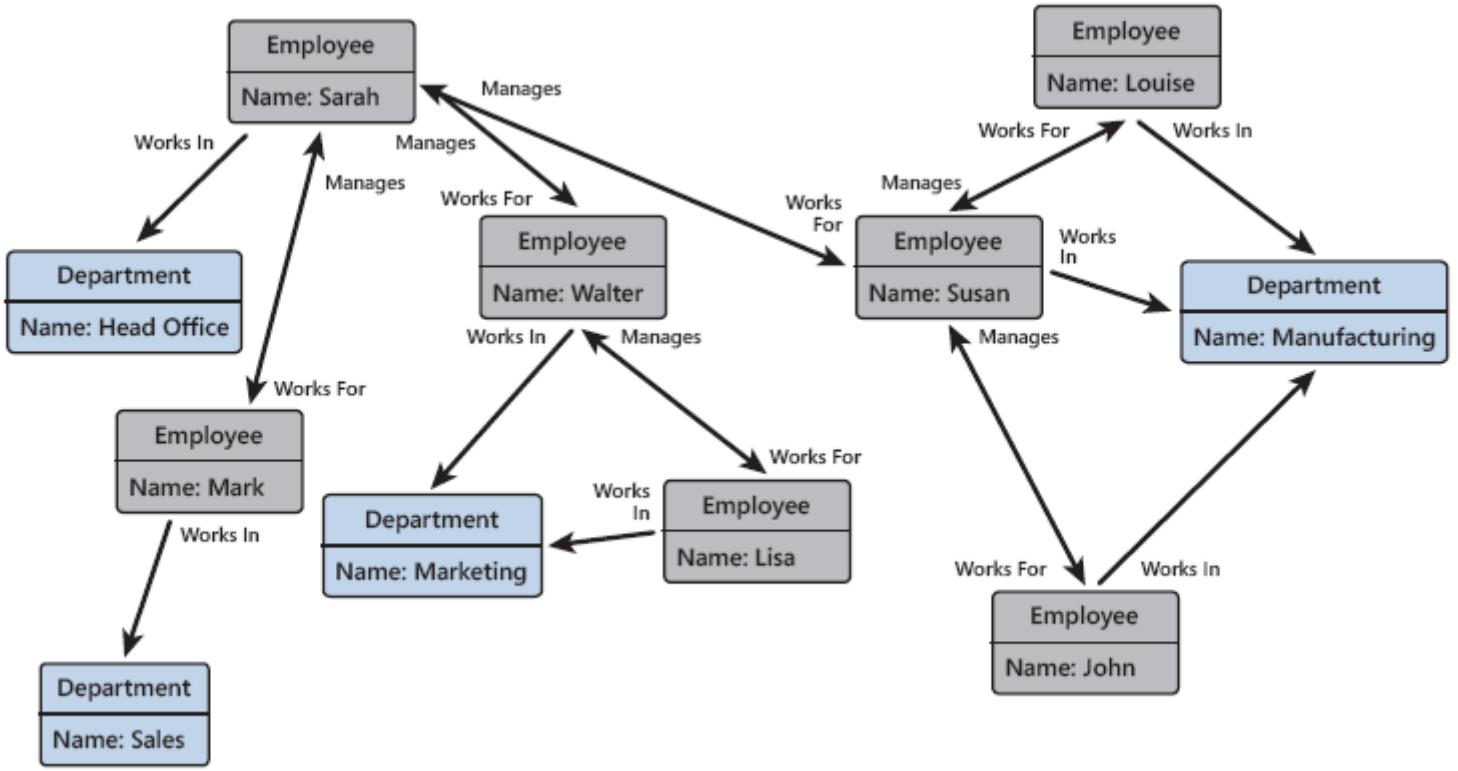
Graph Databases

Graphs have a history dating back to 1736, when Leonhard Euler solved the "Seven Bridges of Königsberg" problem

- Stores two types of information:
  - **Nodes** (aka vertices) that you can think of as **instances of entities**, and
  - **Relationships** (aka links or edges) which specify the relationships between nodes.
  - Nodes and edges can both have properties that provide information about that node or edge (like columns in a table). Additionally, edges can have a direction indicating the nature of the relationship.
- The purpose of a graph database is to enable an application to efficiently perform queries that traverse the network of nodes and edges, and to analyze the relationships between entities.

FlockDB

HR Data Structured as a Graph

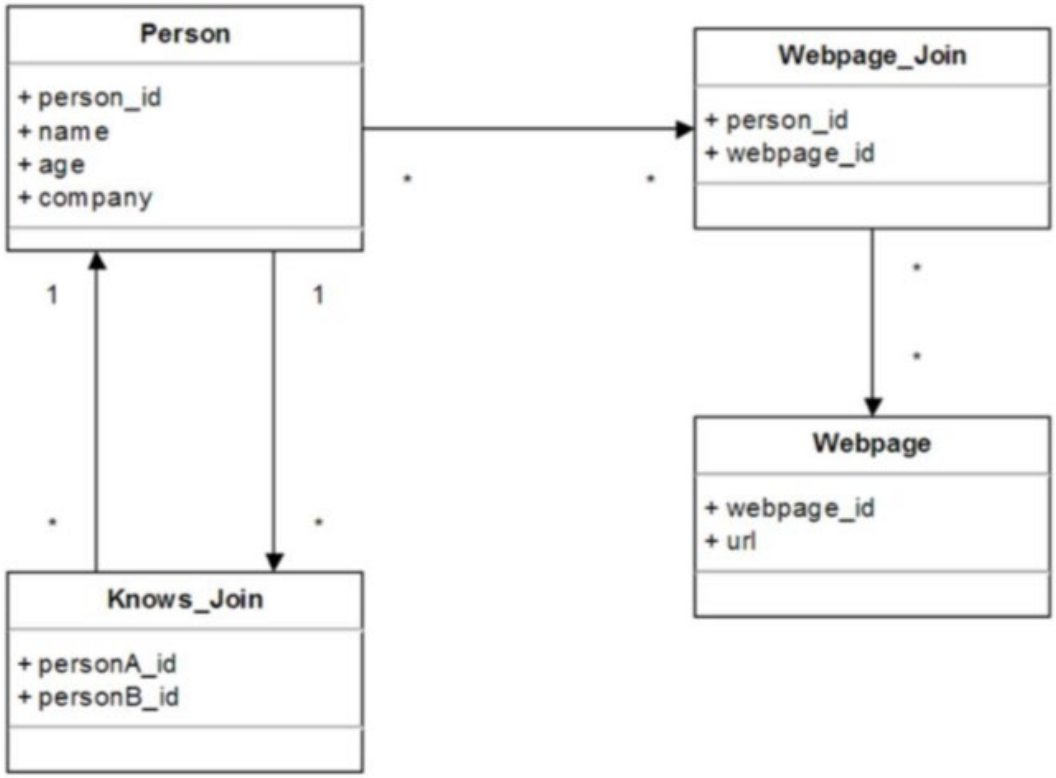


Comparing with Relational Database

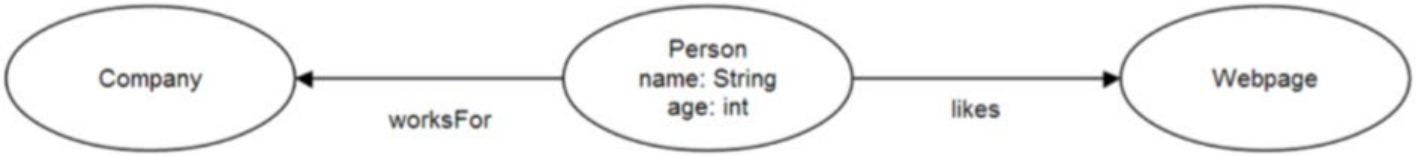
- Pros
  - Schema Flexibility
  - More intuitive querying
  - Avoids Joins
  - Local Hops are not a function of total nodes
  - ACID Compliant
- Cons
  - Query Languages are not unified

Relational Database Example

- In this example, we have a schema of a relational database that stores people, their name, age, the company they work at, who they know, and webpages that they like.
- The **knows** relation and Webpage(**likes)** relation is represented by join tables.
- Perhaps due to poor oversight, this schema only stores one company per person.
- What if we wanted to allow multiple companies? We could either allow duplicates rows (!) or create another join table.

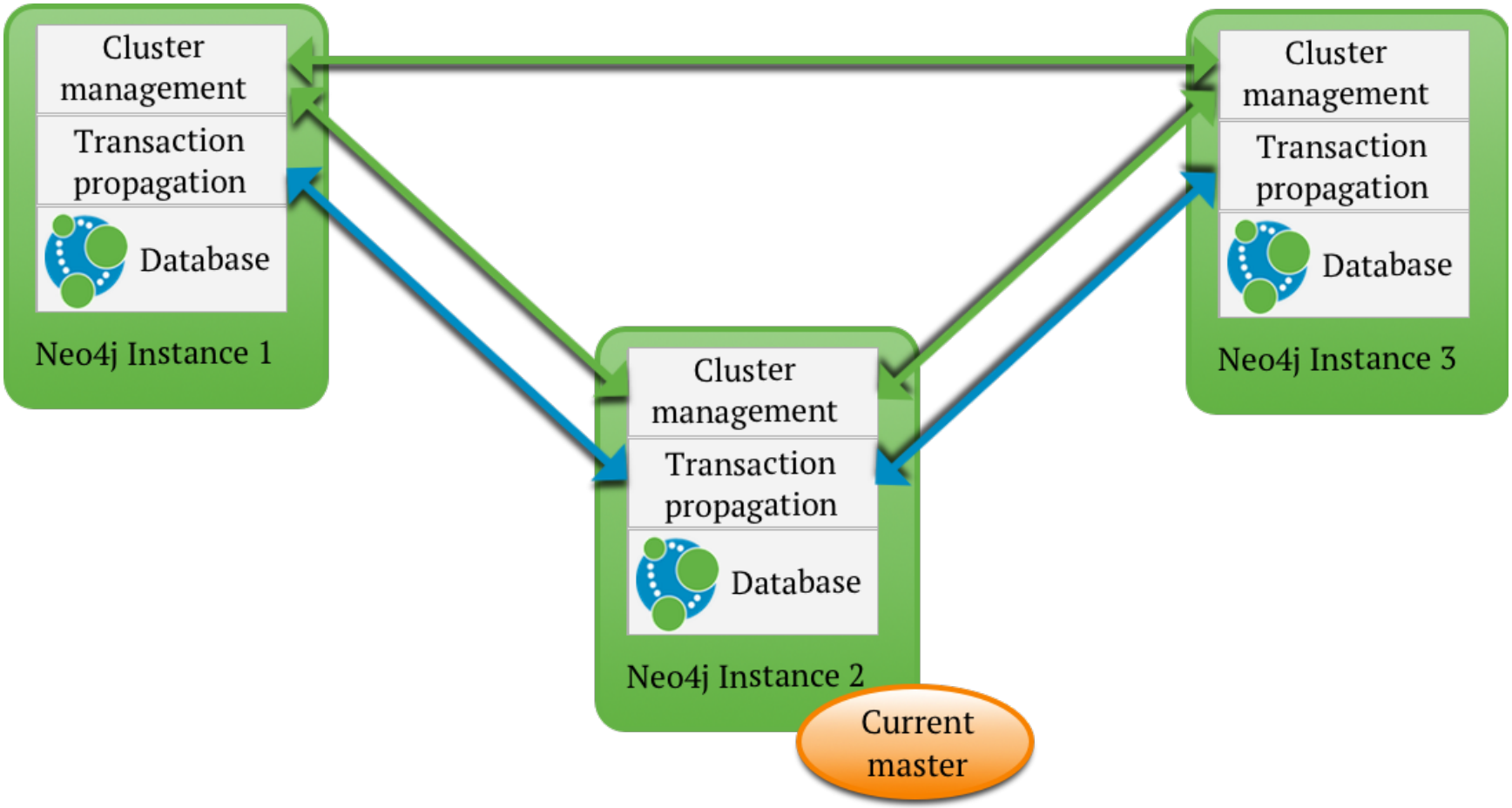


Graph Database Example

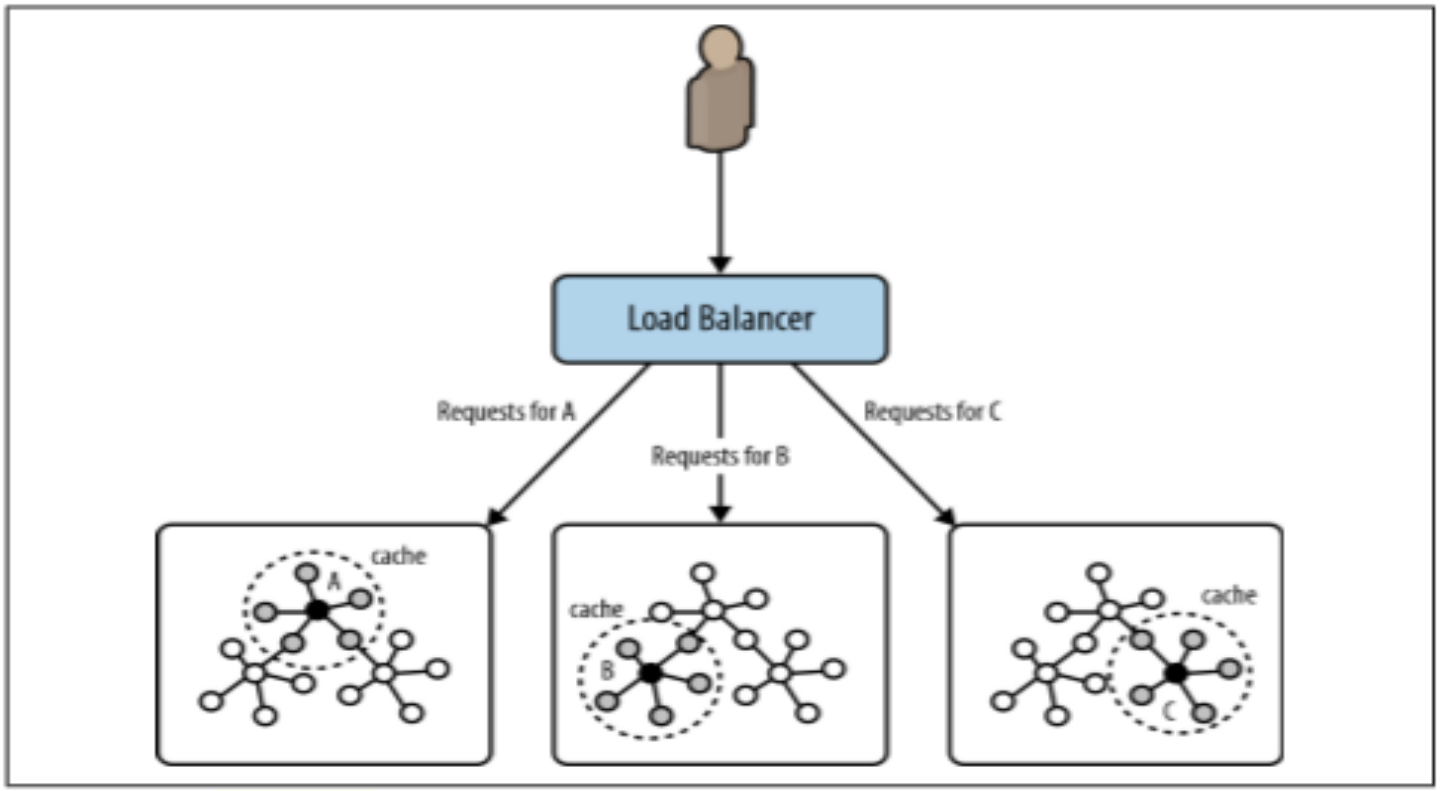


- This diagram is the graph database schema that represents the same people data that was presented before. The circles represent nodes (vertices), and the solid lines represent relationships(edges).
- To solve the same problem in a graph database, we need only create a new edge from the Person node to the Company node. This is a much simpler solution.
- An important thing to note is that graph databases do not have to execute joins for each edge traversal, and avoid join bombs!

Neo4j – offers a form of Master Slave



Cache Sharding supported in Neo4j



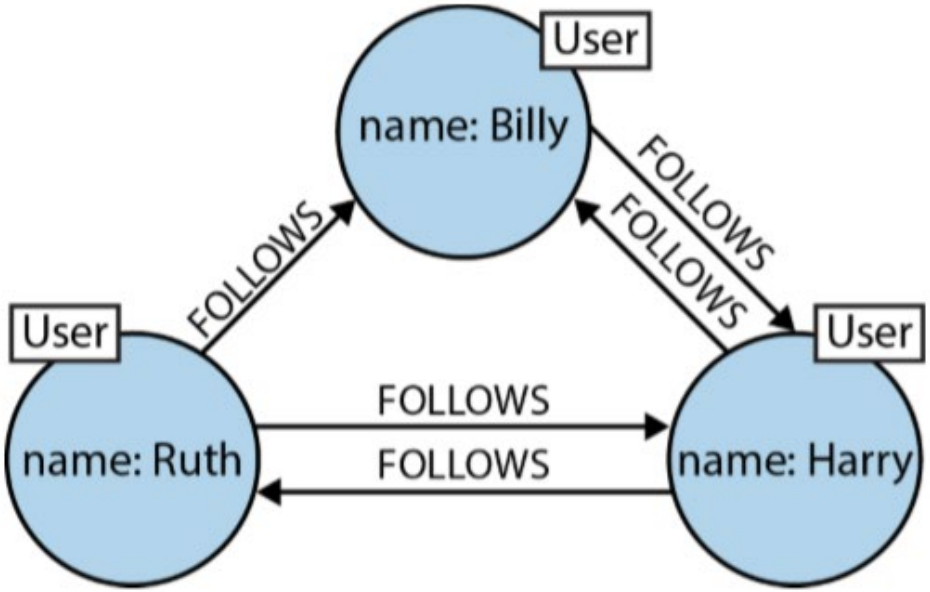
Features(1)

- Consistency
  - Graph Database usually do not support distributing nodes on different servers
    - Within single server data is always consistent
    - Neo4j (open source, java) – fully ACID Compliant on a single server environment
  - Transactions
    - Transactions are supported
      - Neo4j must initiate a transaction for a Write
      - fully ACID Compliant on a single server environment
  - Availability
    - Neo4j provides high availability through replicated slaves
    - Slaves can also handle writes but the write must be committed at the master and then at the slave
  - Some Scaling options available
  - Querying
    - Indexing supported
    - A number of query languages are supported
      - Gremlin; Domain specific language for traversing graphs; Neo4J - Cypher

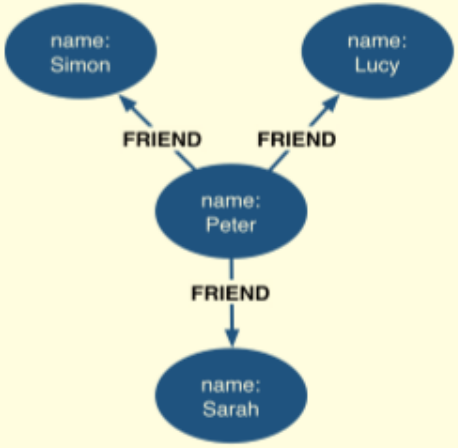
FOUR building Blocks in Neo4j Graph Database

- Nodes
- Relationships
- Properties
- Labels

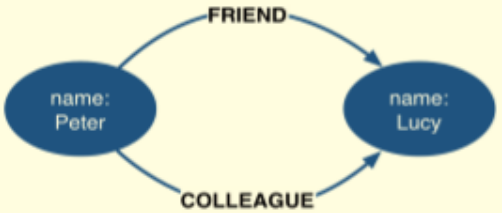
Nodes & Relationships



Relationships



Nodes can have more than one relationship



Nodes can be connected by more than one relationship

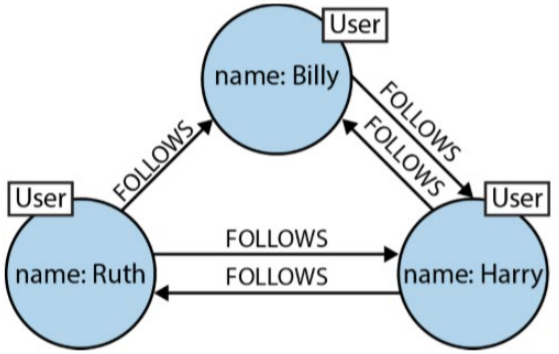
Self relationships are allowed

Relationships

- Every relationship has a name and a direction
- Can contain properties
- Used to represent quality or weight of relationship or metadata
- Every relationship must have a startnode and endnode
- Relationships are defined with regard to node instances ,not classes of nodes
- Relationship=>A join in a relational database
- Two nodes representing the same kind of "thing" can be connected in very different ways

Labels

- Every node can have zero or more labels
  - Node instance Ruth has a label User but could also have another label Author for example
- Used to represent roles (e.g.user, product, company)
- Group nodes
- Allow us to associate indexes and constraints with groups of nodes



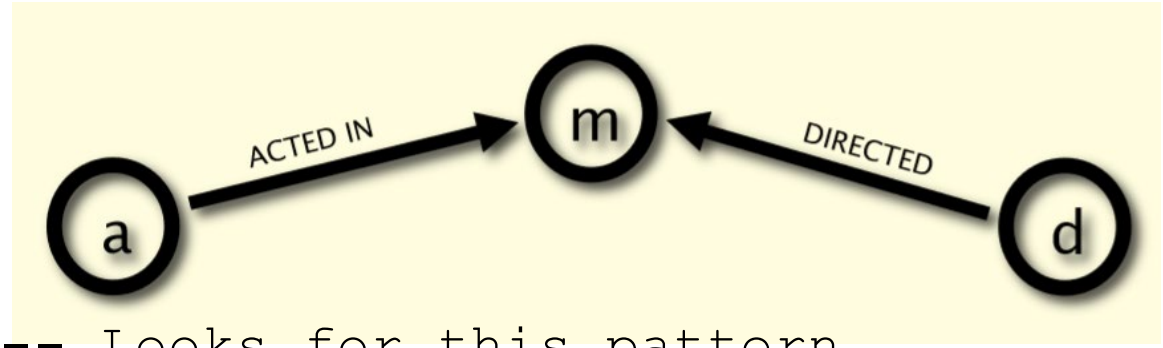
Cypher Query Language

- Declarative Pattern-Matching language
- SQL-like syntax
- Designed for graphs
- E.g.

MATCH (a)-->(b) RETURN a, b

```
--Returns all nodes and their relationships in the
graph database (a and b are just placeholders)
```

Paths -Examples




```
-- Looks for this pattern
MATCH (a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN a.name, m.title, d.name;
```

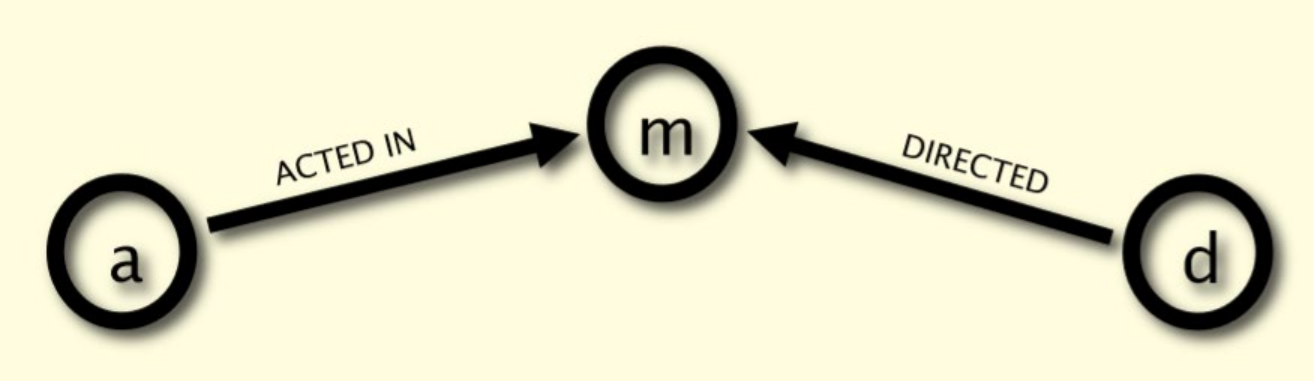
Provide actor's name, movie title and director' name to output returned

Paths - Another way


```
MATCH (a)-[:ACTED_IN]->(m), (d)-[:DIRECTED]-
>(m) RETURN a.name, m.title, d.name;
```

An Implied AND

Paths – returns the nodes that are related by a ACTED IN and DIRECTED relationships




```
MATCH p=(a)-[:ACTED_IN]->(m)<-[:DIRECTED]-(d)
RETURN nodes(p);
```

Node Syntax

- ()
  - #represents an anonymous, uncharacterized node
- (matrix)
  - #if we want to refer a node we need an identifier
- (:Movie)
  - #represents anonymous, characterized node
- (matrix:Movie)
  - #represents a referenced, characterized node i.e. matched Movie nodes
- (matrix:Movie {title: "The Matrix"})
  - #Movie node with a particular property value
- (matrix:Movie {title: "The Matrix", released: 1999})
  - #Movie node with a number of property values

Relationship Syntax

- Cypher uses a pair of dashes (--) to represent an undirected relationship.
- Directed relationships have an arrowhead at one end (eg, <--, -->).
- Bracketed expressions (eg: [...]) can be used to add details.
- This may include identifiers, properties, and/or type information.
- Similar Concepts to the Node Syntax

```
-->       #represents an anonymous, uncharacterized node
-[role]->  #if we want to refer a relationship we need an identifier\alias
-[:ACTED_IN]->  #represents  anonymous characterized relationship
-[role:ACTED_IN]-> #represents  a referenced, characterized relationship
-[role:ACTED_IN {roles: ["Neo"]}]-> #ACTED_IN relationship with a particular property value
```