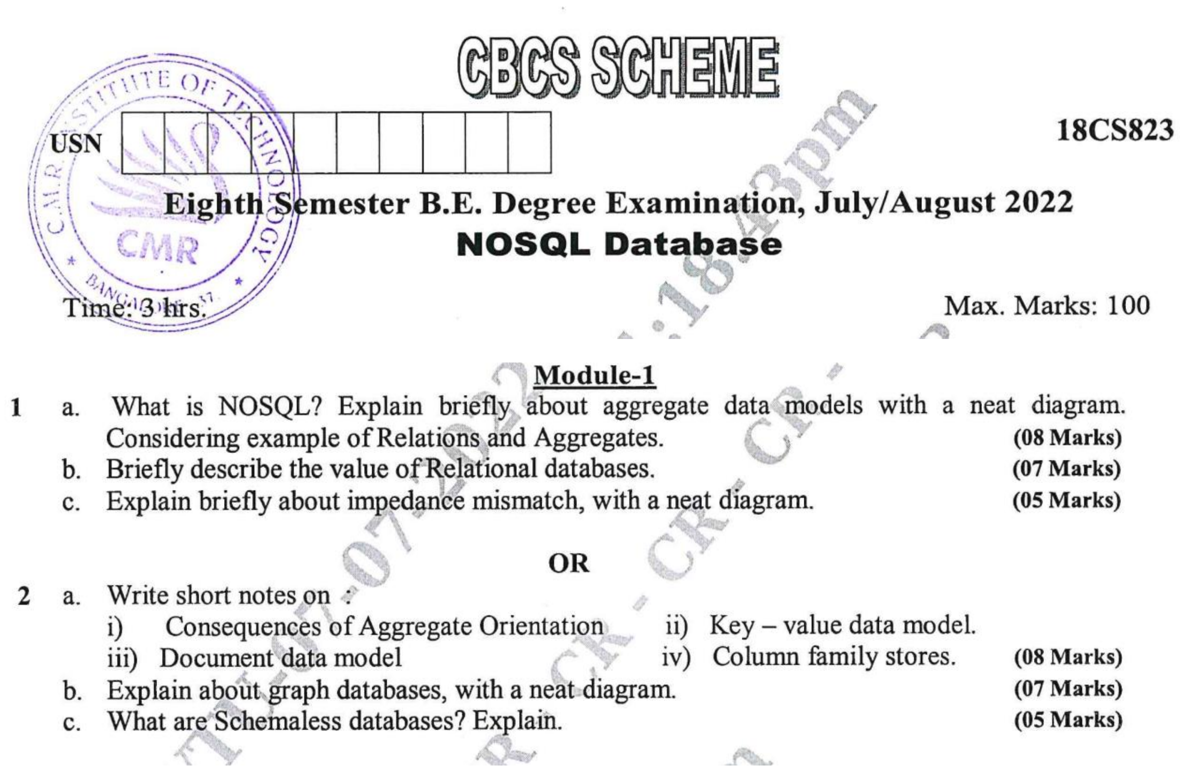
Nosql databses vtu question paper solution by Poonam Tijare, CSE



CBCS SCHEME

USN

18CS823

Eighth Semester B.E. Degree Examination, July/August 2022

NOSQL Database

Time: 3 hrs. Max. Marks: 100

Module-1

1 a. What is NOSQL? Explain briefly about aggregate data models with a neat diagram. Considering example of Relations and Aggregates. **(08 Marks)**

b. Briefly describe the value of Relational databases. **(07 Marks)**

c. Explain briefly about impedance mismatch, with a neat diagram. **(05 Marks)**

OR

2 a. Write short notes on :

i) Consequences of Aggregate Orientation ii) Key – value data model.

iii) Document data model iv) Column family stores. **(08 Marks)**

b. Explain about graph databases, with a neat diagram. **(07 Marks)**

c. What are Schemaless databases? Explain. **(05 Marks)**

1a. Important characteristic of these databases is that they are generally **open-source** projects.

- Most NoSQL databases are driven by the need to **run on clusters**
- Relational databases use **ACID transactions** to handle consistency across the whole database. This inherently clashes with a cluster environment, so NoSQL databases offer a range of options for consistency and distribution
- **Not all NoSQL databases are strongly oriented towards running on clusters. Eg Graph databases are one style of NoSQL databases used to handle complex relationships.**
- **NoSQL databases operate without a schema**, allowing you to freely add fields to database records without having to define any changes in structure first.
- Commonly interpreted as **“Not Only SQL,”** but Oracle or Postgres would fit that definition.
- The change is that now we see relational databases as one option for data storage. This point of view is often referred to as **polyglot persistence—using different data stores in different circumstances.**
- Instead of just picking a relational database because everyone does, we need to understand the nature of the data we’re storing and how we want to manipulate it.

The result is that most organizations will have a mix of data storage technologies for different circumstances

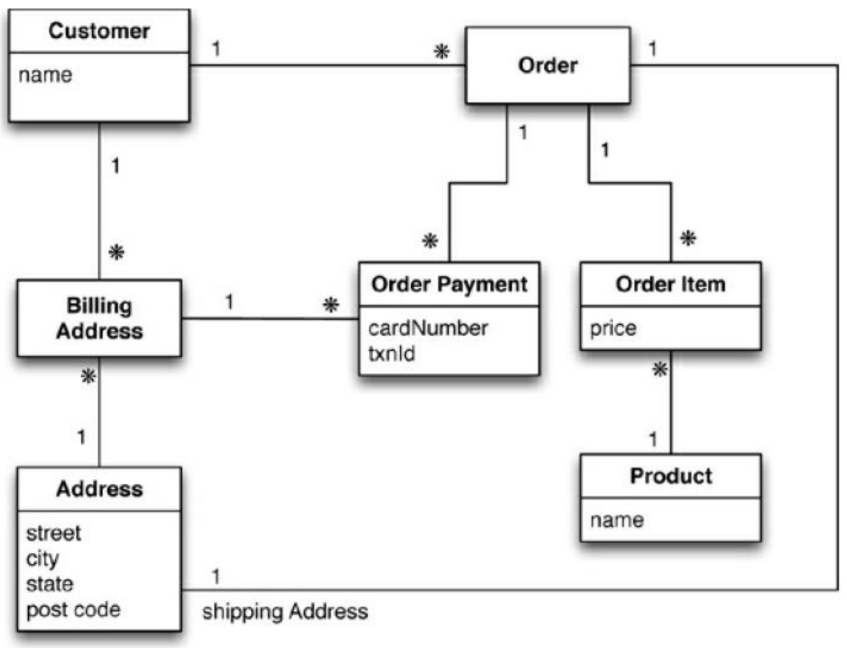


Figure 2.1. Data model oriented around a relational database (using UML notation [Fowler UML])

In this model, we have two main aggregates: **customer and order**.

The black-diamond composition marker used in UML to show how data fits into the aggregation structure.

The customer contains a list of billing addresses; the order contains a list of order items, a shipping address, and payments. The payment itself contains a billing address for that payment.

Aggregation boundaries(lines) can be drawn with a thought about accessing data—and make that part of your thinking when developing the application data model.

Indeed we could draw our aggregate boundaries differently, putting all the orders for a customer into the customer aggregate.

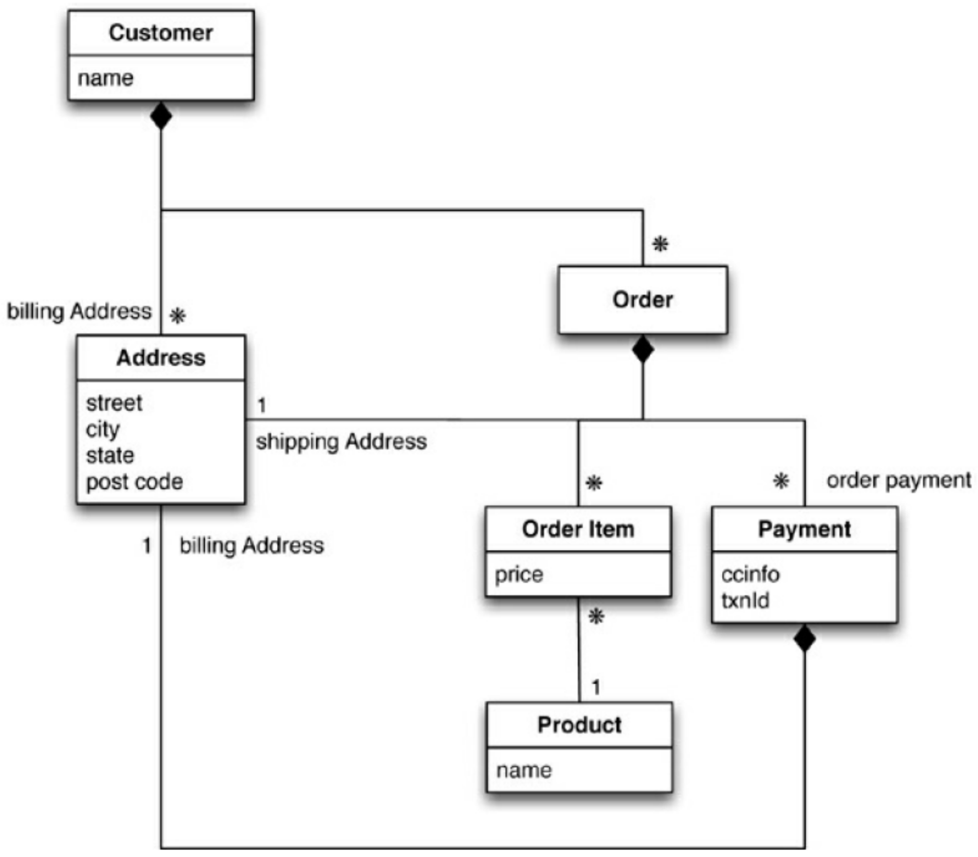


Figure 2.4. Embed all the objects for customer and the customer's orders

---

1b. Relational databases are built to work with large amounts of data and are commonly used by companies in the following scenarios:

- Large corporate institutions:
- Companies with multiple departments and divisions. They have multiple data stores for information used by different teams within the department.

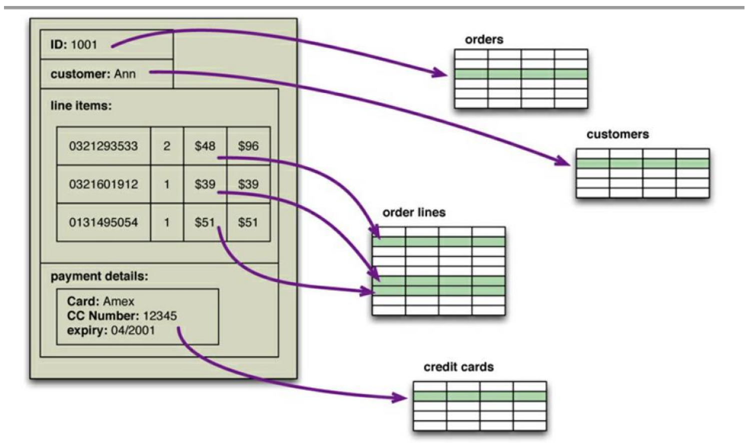
Relational databases are also called Relational Database Management Systems (RDBMS) or SQL databases. They are the most common type of database used in business. The most popular of these have been Microsoft SQL Server, Oracle Database, MySQL, and IBM DB2. These relational databases are used mostly in large enterprise scenarios. These are vital for businesses because they enable companies to store, access, update and manage information and have a clear path to the various departments that need this data.

-

1c. For application developers, the biggest frustration has been what's commonly called the impedance mismatch: the difference between the relational model and the in-memory data structures.

- The relational data model organizes data into a structure of tables and rows, or more properly, relations and tuples.

- In the relational model, a tuple is a set of name-value pairs and a relation is a set of tuples. (The relational definition of a tuple is slightly different from that in mathematics and many programming languages with a tuple data type, where a tuple is a sequence of values.)
- All operations in SQL consume and return relations, which leads to the mathematically elegant relational algebra.
- This foundation on relations provides a certain elegance and simplicity, but it also introduces limitations.



- In particular, the values in a relational tuple have to be simple—**they cannot contain any structure, such as a nested record or a list.** This limitation isn't true for in-memory data structures, which can take on much richer structures than relations.
- As a result, if you want to use a richer in memory data structure, **you have to translate it to a relational representation to store it on disk. Hence the impedance mismatch—two different representations that require translation.**
- Many OOP languages developed, while object-oriented languages succeeded in becoming the major force in programming, object-oriented databases faded into obscurity.
- **Growing professional divide between application developers and database administrator**
- **Impedance mismatch has been made much easier to deal with by the wide availability of object relational mapping frameworks,** such as Hibernate and iBATIS that implement well-known mapping patterns [Fowler PoEAA], but the mapping problem is still an issue.
- **Object-relational mapping frameworks remove a lot of grunt work**, but can become a problem of their own when people try too hard to ignore the database and **query performance suffers.**
- **Relational databases continued to dominate the enterprise computing world in the 2000s, but during that decade cracks began to open in their dominance.**

2a. 1) When working with aggregate-oriented databases, **we have a clearer semantics to consider by focusing on the unit of interaction with the data storage.** It is, however, not a logical data property: It's all about how the data is being used by applications—**a concern that is often outside the bounds of data modeling.**

Relational databases have no concept of aggregate within their data model, so we call them aggregate-ignorant. In the NoSQL world, **graph databases are also aggregate-ignorant.**

It's often difficult to draw aggregate boundaries well, particularly if the same data is used in many different contexts.

Eg. product monthly sales data evaluation by retailer

An aggregate-ignorant model allows you to easily look at the data in different ways, so it is a better choice when you don't have a primary structure for manipulating your data.

Aggregates have an important consequence for transactions.

Relational databases allow you to manipulate any combination of rows from any tables in a single transaction. Such transactions are called ACID transactions: Atomic, Consistent, Isolated, and Durable. ACID is a rather contrived acronym; the real point is the atomicity.

In general, **it's true that aggregate-oriented databases don't have ACID transactions that span multiple aggregates**.

Instead, they support atomic manipulation of a single aggregate at a time. This means that if we need to manipulate multiple aggregates in an atomic way, we have to manage that ourselves in the application code.

2aii) Perhaps the best way to think of the column-family model is as a two-level aggregate structure.

As with key-value stores, the first key is often described as a row identifier, picking up the aggregate of interest.

The difference with column-family structures is that this row aggregate is itself formed of a map of more detailed values.

These second-level values are referred to as columns.

As well as accessing the row as a whole, operations also allow picking out a particular column, so to get a particular customer's name from Figure 2.5 you could do something like get('1234', 'name').

2a iii) **key-value and document databases were strongly aggregate-oriented.**

- **The two models differ in that in a key-value database, the aggregate is opaque (non transparent) to the database**—just some big blob of mostly meaningless bits.

- In contrast, **a document database is able to see a structure in the aggregate**.
- **The advantage of opacity is that we can store whatever we like in the aggregate**. The database may impose some general size limit, but other than that we have complete freedom.

A document database imposes limits on what we can place in it, defining allowable structures and types. In return, however, we get more flexibility in access.

With a key-value store, we can only access an aggregate by lookup based on its key. With a document database, we can submit queries to the database based on the fields in the aggregate, we can retrieve part of the aggregate rather than the whole thing, and **database can create indexes based on the contents of the aggregate**.

The line between key-value and document gets a bit blurry. People often put an ID field in a document database to do a key-value style lookup.

Databases classified as key-value databases may allow you structures for data beyond just an opaque aggregate.

For example, Riak allows you to add metadata to aggregates for indexing and inter aggregate links, Redis allows you to break down the aggregate into lists or sets. You can support querying by integrating search tools such as Solr. As an example, **Riak includes a search facility that uses Solr-like searching on any aggregates that are stored as JSON or XML structures.**

With key-value databases, we expect to mostly look up aggregates using a key. With document databases, we mostly expect to submit some form of query based on the internal structure of the document; this might be a key, but it's more likely to be something else.

2a iv) One of the early and influential **NoSQL databases was Google's BigTable** [Chang etc.]. Its name conjured up a **tabular structure which it realized with sparse columns and no schema.**

Inspired HBASE and Cassandra

These databases with a bigtable-style data model are often referred to as column stores.

Pre-NoSQL column stores, such as C-Store [C-Store], were happy with SQL and the relational model. **The thing that made them different was the way in which they physically stored data. Most databases have a row as a unit of storage which, in particular, helps write performance.** However, there are **many scenarios where writes are rare**, but you often need to read a few columns of many rows at once. In this situation, **it's better to store groups of columns for all rows as the basic storage unit—which is why these databases are called column stores.**

2b.

Graph databases are an odd fish in the NoSQL pond. Most NoSQL databases were inspired by the need to run on clusters, which led to aggregate-oriented data models of large records with simple connections.

Graph databases are motivated by a different frustration with relational databases and thus have an opposite model—small records with complex interconnections, something like Figure 3.1.

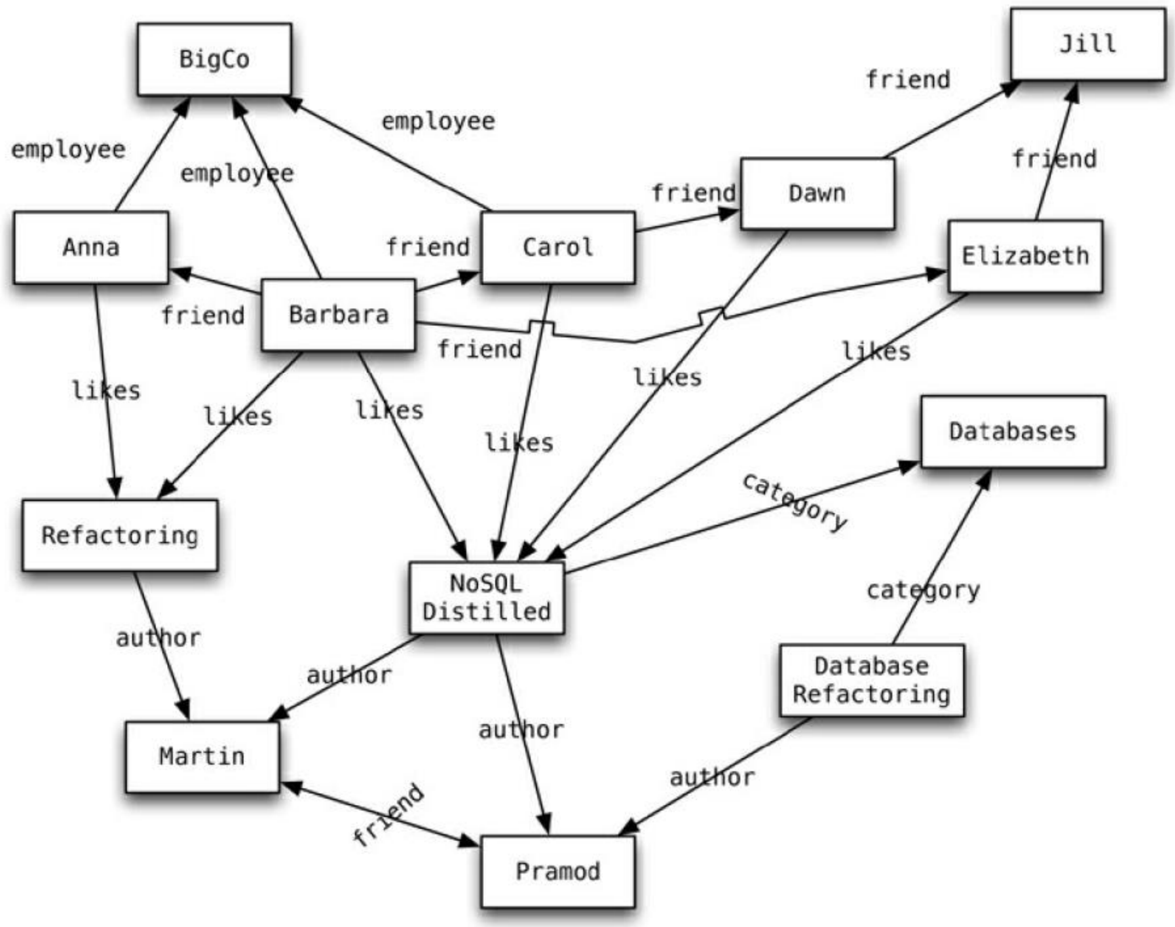


Figure 3.1. An example graph structure

n Figure 3.1 we have a web of information whose nodes are very small (nothing more than a name) but there is a rich structure of interconnections between them. With this structure, we can ask questions such as “find the books in the Databases category that are written by someone whom a friend of mine likes.”

This is ideal for capturing any data consisting of complex relationships such as social networks, product preferences, or eligibility rules

Once you have built up a graph of nodes and edges, a graph database allows you to query that network with query operations designed with this kind of graph in mind.

Doing same with lot of joins in RDBMS is very expensive process

Graph databases make traversal along the relationships very cheap

- This is majorly because **graph databases shift most of the work of navigating relationships from query time to insert time.**
- I**deal for situations where querying performance is more important than insert speed.**
- **The emphasis on relationships makes graph databases very different from aggregate-oriented databases.**
- **Databases are more likely to run on a single server rather than distributed across clusters.**
- **ACID transactions need to cover multiple nodes and edges to maintain consistency.**
- The only thing they have in common with aggregate-oriented databases is their rejection of the relational model.

- 2 c) A common theme across all the forms of NoSQL databases is that they are schemaless.
- In a relational database, first have to define a schema—a defined structure for the database which says what tables,which columns exist, and what data types each column can hold.
- Before you store some data, you have to have the schema defined for it.
- With NoSQL databases, storing data is much more casual. A key-value store allows you to store any data you like under a key.
- A document database effectively does the same thing, since it makes no restrictions on the structure of the documents you store.
- Column-family databases allow you to store any data under any column you like. Graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish.

Advocates of schemaless is freedom and flexibility.

With a schema, you have to figure out in advance what you need to store, but that can be hard to do.

Without a schema binding you, you can easily store whatever you need. This allows you to easily change your data storage as you learn more about your project. You can easily add new things as you discover them.

Furthermore, if you find you don't need some things anymore, you can just stop storing them, without worrying about losing old data as you would if you delete columns in a relational schema.

Schemaless store also makes it easier to deal with nonuniform data: data where each record has a different set of fields.

Problems of Schemalessness

Schemalessness is appealing, and but it brings some problems of its own.

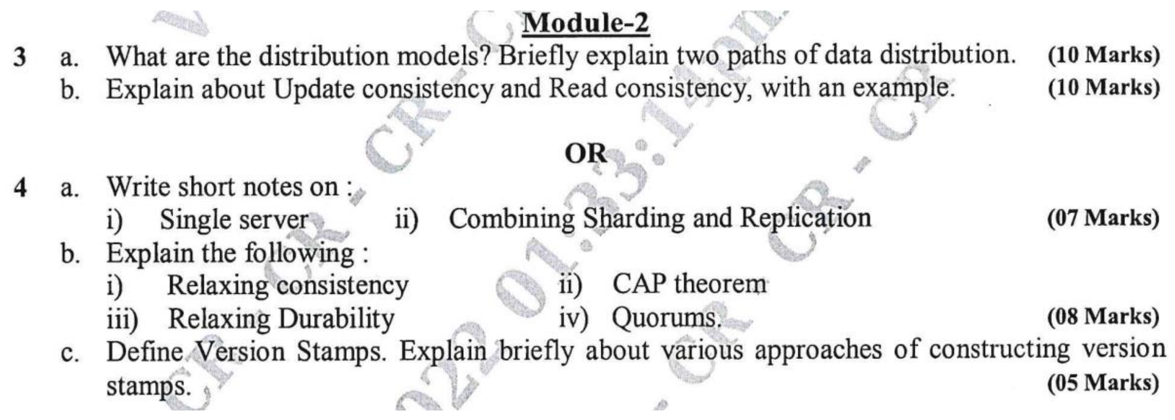
If all you are doing is storing some data and displaying it in a report as a simple list of fieldName: value lines then a schema is only going to get in the way.

Fact is that whenever we write a program that accesses data, that program almost always relies on some form of implicit schema.

This implicit schema is a set of assumptions about the data's structure in the code that manipulates the data. Having the implicit schema in the application code results in some problems. It means that in order to understand what data is present you have to dig into the application code.

If that code is well structured you should be able to find a clear place from which to deduce the schema. But there are no guarantees; it all depends on how clear the application code is.

Furthermore, the database remains can't use the schema to help it decide how to store and retrieve data efficiently. It can't apply its own validations upon that data to ensure that different applications don't manipulate data in an inconsistent way.



Module-2

3 a. What are the distribution models? Briefly explain two paths of data distribution. **(10 Marks)**
b. Explain about Update consistency and Read consistency, with an example. **(10 Marks)**

OR

4 a. Write short notes on :
i) Single server ii) Combining Sharding and Replication **(07 Marks)**
b. Explain the following :
i) Relaxing consistency ii) CAP theorem
iii) Relaxing Durability iv) Quorums. **(08 Marks)**
c. Define Version Stamps. Explain briefly about various approaches of constructing version stamps. **(05 Marks)**

- Q3 The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on.
- A more appealing option is to scale out—run the database on a cluster of servers.
- Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.

Depending on your distribution model, you can get a data store that will give you the

- **ability to handle larger quantities of data,**
- **the ability to process a greater read or write traffic, or**
- **more availability in the face of network slowdowns or breakages.**

These are often important benefits, but they come at a cost.

Running over a cluster introduces complexity—so it's not something to do unless the benefits are compelling.

Broadly, there are two paths to data distribution: replication and sharding.

- Replication takes the same data and copies it over multiple nodes.
- Sharding puts different data on different nodes.
- Replication and sharding are orthogonal techniques: You can use either or both of them.

Replication comes into two forms: master-slave and peer-to-peer.
Works on single-server, master-slave replication, with sharding, and finally peer-to-peer replication.

4.1. Single Server

The first and the simplest distribution option is the one we would most often recommend—**no distribution at al**l.
Run the database on a single machine that handles all the reads and writes to the data store. We prefer this option because **it eliminates all the complexities** that the other options introduce; it's **easy for operations people to manage and easy for application developers** to reason about.
NoSQL can be used with a single-server distribution model if the data model of the NoSQL store is more suited to the application. **Graph databases are the obvious category here**—these work best in a single-server configuration. If your data usage is mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it's easier on application developers.

4.2. Sharding

Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called sharding (see Figure 4.1).

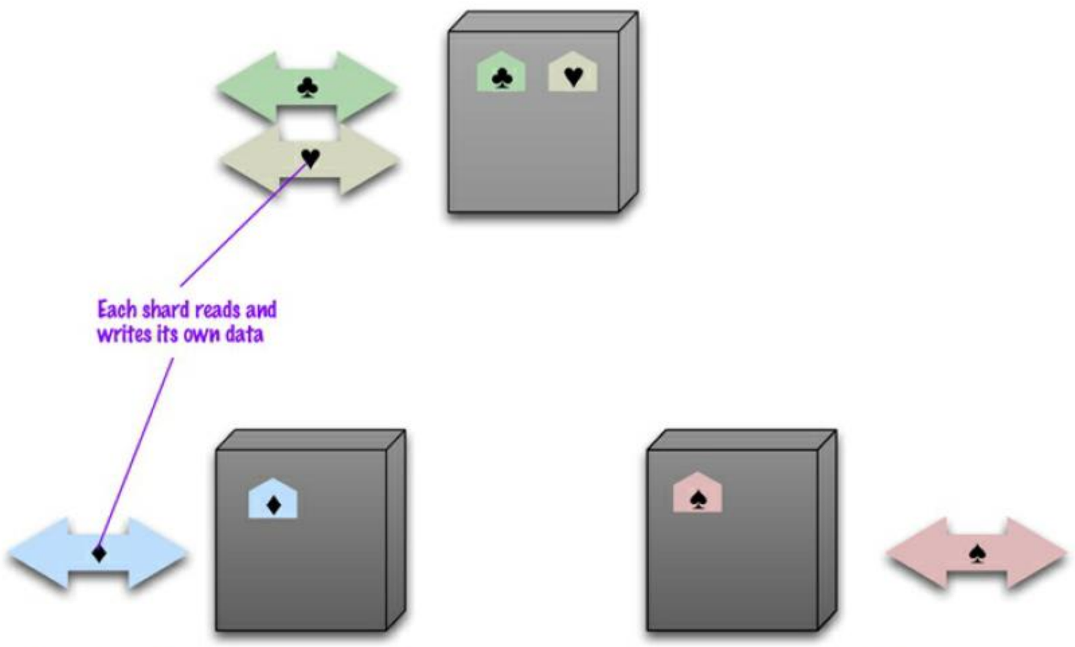


Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

> In order to get close to it we have to ensure that data that's accessed together is clustered together on the same node and that these clumps are arranged on the nodes to provide the best data access.
How to clump the data up so that one user mostly gets her data from a single server?.
This is where aggregate orientation comes in really handy. **The whole point of aggregates is that we design them to combine data that's commonly accessed together—so aggregates leap out as an obvious unit of distribution.**

> **When it comes to arranging the data on the nodes, there are several factors that can help improve performance.**

- If you know that most accesses of certain aggregates are **based on a physical location**, you can place the data close to where it's being accessed. Ex. If you have orders for someone who lives in Boston, you can place that data in your eastern US data center.
- **Another factor is trying to keep the load even.** This means that you should try to **arrange aggregates so they are evenly distributed across the nodes which all get equal amounts of the load.**
  This may vary over time, for example if some data tends to be accessed on certain days of the week—**so there may be domain-specific rules depending on application load.**
  In some cases, **it's useful to put aggregates together if you think they may be read in sequence**. The Bigtable paper [Chang etc.] described keeping its rows in lexicographic order and sorting web addresses based on reversed domain names (e.g., com.martinfowler).
  This way data for multiple pages could be accessed together to improve processing efficiency.

- **Historically most people have done sharding as part of application logic.** EX. You might put all customers with surnames starting from A to D on one shard and E to G on another.
    - This complicates the programming model, as application code needs to ensure that queries are distributed across the various shards.
    - Furthermore, rebalancing the sharding means changing the application code and migrating the data.
    - Many NoSQL databases offer auto-sharding, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard.

This can make it much easier to use sharding in an application.
(sharding vs Replication)
Sharding is particularly valuable for performance because it can improve both read and write performance.
Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.

3b. **Update Consistency**

Case study:
We'll begin by considering updating a telephone number. Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. **They both have update access, so they both go in at the same time to update the number**. We'll assume **they update it using a slightly different format. This issue is called a write-write conflict: two people updating the same data item at the same time**.
When the writes reach the server, the server will serialize them—decide to apply one, then the other. Let's assume it uses alphabetical order and picks Martin's update first, then Pramod's. **Without any concurrency control,** Martin's update would be applied and immediately overwritten by Pramod's. In this case **Martin's is a lost update**. We see this as a failure of consistency because Pramod's **update was based on the state** before Martin's update, yet was applied after it.

Approaches for maintaining consistency in the face of concurrency: pessimistic or optimistic

- **A pessimistic approach works by preventing conflicts from occurring;**
- **An optimistic approach lets conflicts occur, but detects them and takes action to sort them out.**
- For update conflicts, the most common pessimistic approach is to have **write locks**, so that in order to change a value you need to acquire a lock, and the system ensures that only one client can get a lock at a time.
  Eg. Martin and Pramod would both attempt to acquire the write lock, but only Martin (the first one) would succeed. Pramod would then see the result of Martin's writing before deciding whether to make his own update.
- A common optimistic approach is a **conditional update where any client that does an update tests the value just before updating it to see if it's changed since his last read**. In this case, Martin's update would succeed but Pramod's would fail. The error

would let Pramod know that he should look at the value again and decide whether to attempt a further update.

Both of the approaches work in a single server environment. In multiple server environments like peer to peer then two nodes might apply the updates in a different order, resulting in a different value for the telephone number on each peer.

Often, another approach for concurrency in distributed systems, sequential consistency—ensuring that all nodes apply operations in the same order.

- This approach is familiar to many programmers from **version control systems**, particularly distributed version control systems that by their nature will often have conflicting commits.
- The next step again follows from version control: **You have to merge the two updates somehow.**
- **Users may update the merged information or the computer may be able to perform the merge itself**; if it was a phone formatting issue, it may be able to realize that and apply the new number with the standard format.
- **Any automated merge of write-write conflicts is highly domain-specific and needs to be programmed for each particular case**.

Often, when people first encounter these issues, their reaction is to prefer pessimistic concurrency because they are determined to avoid conflicts. While in some cases this is the right answer, there is always a tradeoff.
Concurrent programming involves a fundamental tradeoff between safety (avoiding errors such as update conflicts) and liveness (responding quickly to clients).

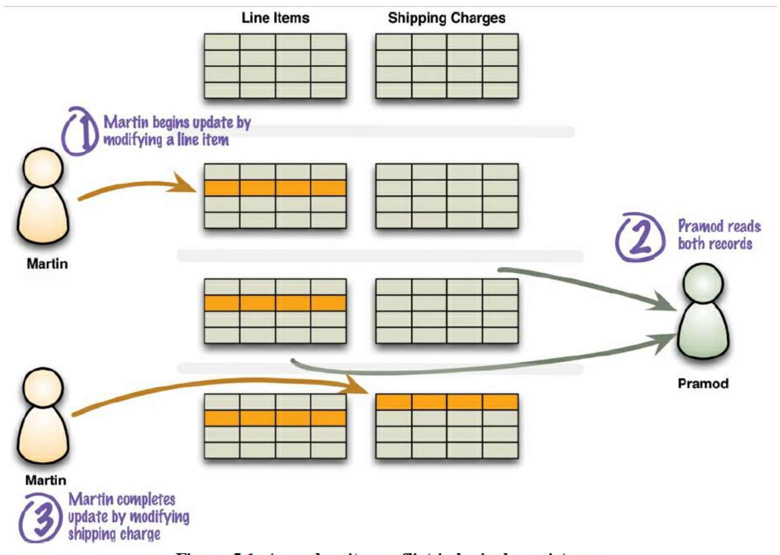
Disadvantage of Pessimistic approach

Pessimistic approaches often severely degrade the responsiveness of a system to the degree that it becomes unfit for its purpose. This problem is made worse by the danger of errors—pessimistic concurrency often leads to deadlocks, which are hard to prevent and debug.

Replication makes it much more likely to run into write-write conflicts. If different nodes have different copies of some data which can be independently updated, then you'll get conflicts unless you take specific measures to avoid them. Using a single node as the target for all writes for some data makes it much easier to maintain update consistency. Of the distribution models we discussed earlier, all but peer-to-peer replication do this.

Read Consistency

- Having a data store that maintains update consistency is one thing, but it doesn't guarantee that readers of that data store will always get consistent responses to their requests.
- Let's imagine we have an order with line items and a shipping charge. The shipping charge is calculated based on the line items in the order. If we add a line item, we thus also need to recalculate and update the shipping charge. In a relational database, the shipping charge and line items will be in separate tables.
- The danger of inconsistency is that Martin adds a line item to his order, Pramod then reads the line items and shipping charge, and then Martin updates the shipping charge. This is an inconsistent read or read-write conflict: In Figure 5.1 Pramod has done a read in the middle of Martin's write.


Line Items
Shipping Charges
Martin begins update by modifying a line item
Martin
Pramod reads both records
Pramod
Martin
Martin completes update by modifying shipping charge
Figure 5.1. A read-write conflict in logical consistency

- **We refer to this type of consistency as logical consistency: ensuring that different data items make sense together.**
- **To avoid a logically inconsistent read-write conflict, relational databases support the notion of transactions.**

Providing Martin wraps his two writes in a transaction, the system guarantees that Pramod will either read both data items before the update or both after the update.
A common claim we hear is that NoSQL databases don't support transactions and thus **can't be consistent**. **Such a claim is mostly wrong because**

- Our first clarification is that any statement about lack of transactions usually only applies to some NoSQL databases, in particular **the aggregate-oriented ones**. In contrast, graph databases tend to support ACID transactions just the same as relational databases.
- Secondly, a**ggregate-oriented databases do support atomic updates, but only within a single aggregate.** This means that you will have **logical consistency within an aggregate but not between aggregates.**

Q4 a. **Single server system**

- **A single-server system is the obvious example of a CA system—a system that has Consistency and Availability but not Partition tolerance. A single machine can't partition, so it does not have to worry about partition tolerance.**
- However, this would mean that if a partition ever occurs in the cluster, all the nodes in the cluster would go down so that no client can talk to a node.
- **By the usual definition of "available," this would mean a lack of availability, but this is where CAP's special usage of "availability" gets confusing.**

- **CAP defines "availability" to mean "every request received by a non failing node in the system must result in a response". So a failed, unresponsive node doesn't infer a lack of CAP availability.**
- **This does imply that you can build a CA cluster, but you have to ensure it will only partition rarely and completely. So clusters have to be tolerant of network partitions. But CAP say any two properties to satisfy.**
- **This isn't a binary decision; often, you can trade off a little consistency to get some availability. The resulting system would be neither perfectly consistent nor perfectly available—but would have a combination that is reasonable for your particular needs.**

Combining Sharding and Replication

- Replication and sharding are strategies that can be combined. If we use both master-slave replication and sharding (see Figure 4.4), **this means that we have multiple masters, but each data item only has a single master**. Depending on your configuration, you may choose a node to be a master for some data and slaves for others, or you may dedicate nodes for master or slave duties.

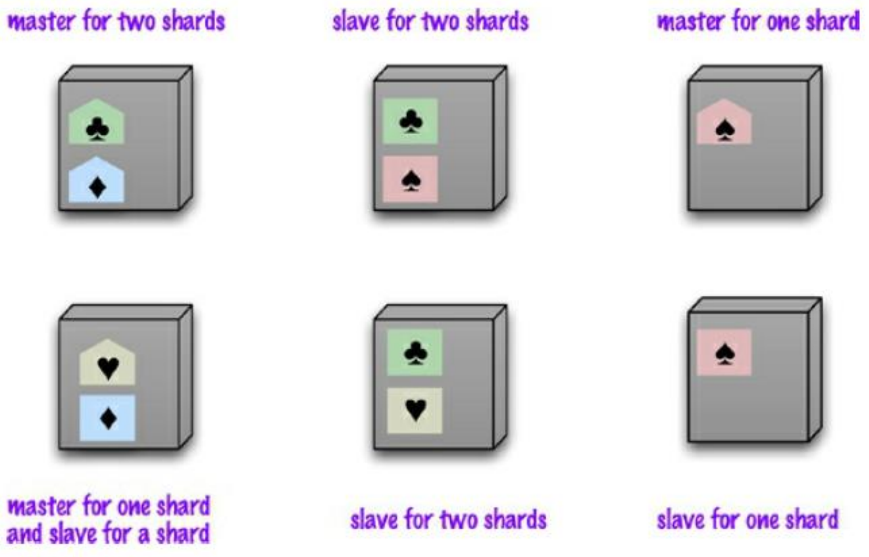


Figure 4.4. Using master-slave replication together with sharding

- Using peer-to-peer replication and sharding is a common strategy for column-family databases. In a scenario like this you might have tens or hundreds of nodes in a cluster with data sharded over them.
- A good starting point for peer-to-peer replication is to have a replication factor of 3, so each shard is present on three nodes. Should a node fail, then the shards on that node will be built on the other nodes (see Figure 4.5).

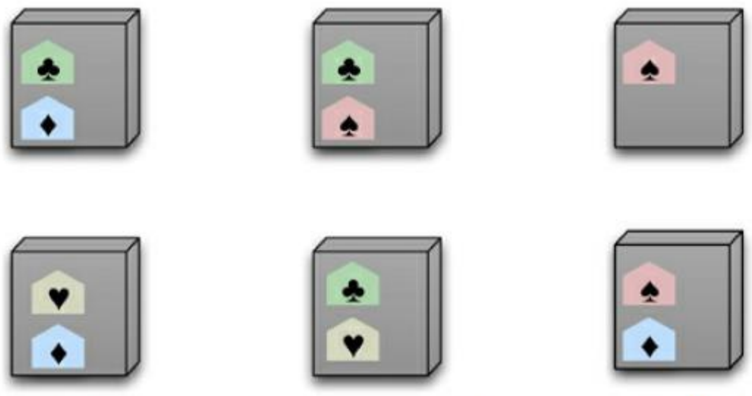


Figure 4.5. Using peer-to-peer replication together with sharding

4b) Relaxing Consistency

- Consistency is a Good Thing—but,it comes with sacrifices.
- **It is always possible to design a system to avoid inconsistencies, but often impossible to do so without making unbearable sacrifices in other characteristics of the system.**
- **As a result, we often have to tradeoff consistency for something else.**
- While some architects see this as a disaster, we see it as part of the inevitable tradeoffs involved in system design.
- **Furthermore, different domains have different tolerances for inconsistency, and we need to take this tolerance into account as we make our decisions.**
- **Trading off consistency is a familiar concept even in single-server relational database systems.** Here, our principal tool **to enforce consistency is the transaction, and transactions can provide strong consistency guarantees.**
- However, **transaction systems usually come with the ability to relax isolation levels, allowing queries to read data that hasn't been committed yet.**
- In practice we see **most applications relax consistency down from the highest isolation level (serialized) in order to get effective performance**.
- We most commonly see people u**sing the read-committed transaction level, which eliminates some read-write conflicts but allows others**.
- **Many systems forgo transactions entirely because the performance impact of transactions is too high.**
- On a small scale, **we saw the popularity of MySQL during the days when it didn't support transactions. Many websites liked the high speed of MySQL and were prepared to live without transactions.**

- At the other end of the scale, **some very large websites, such as eBay, have had to forgo transactions in order to perform acceptably**— this is particularly true when you need to introduce sharding.
- Even without these constraints, many application builders **need to interact** with **remote systems that are outside a transaction boundary,** so updating outside of transactions is a quite common occurrence for enterprise applications.

The CAP Theorem

In the NoSQL world it's common to refer to the CAP theorem as the reason why you may need to relax consistency. It was originally proposed by Eric Brewer in 2000 [Brewer] and given a formal proof by Seth Gilbert and Nancy Lynch [Lynch and Gilbert] a couple of years later.

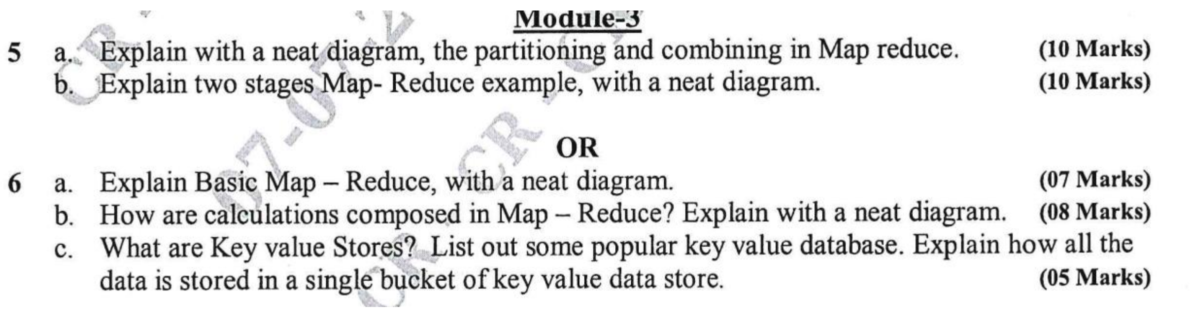
- **The basic statement of the CAP theorem is that, given the three properties of Consistency, Availability, and Partition tolerance, you can only get two.**
- **Obviously this depends very much on how you define these three properties, and differing opinions have led to several debates on what the real consequences of the CAP theorem are.**

Consistency

- **Availability has a particular meaning in the context of CAP—it means that if you can talk to a node in the cluster, it can read and write data.**
- **Partition tolerance means that the cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other (situation known as a split brain, see Figure 5.3).**

----

1. 4c. **Another approach for session consistency is to use version stamps and ensure every interaction with the data store includes the latest version stamp seen by a session.** The **server node must then ensure that it has the updates that include that version stamp before responding to a request.**
2. Maintaining session consistency with sticky sessions and master-slave replication can be awkward if you want to read from the slaves to improve read performance but still need to write to the master.
   3.1 One way of handling this is for **writes to be sent the slave, who then takes responsibility for forwarding them to the master while maintaining session consistency for its client.**
   3.2 **Another approach is to switch the session to the master temporarily when doing a write, j**ust long enough that reads are done from the master until the slaves have caught up with the update.
   3.3 About replication consistency in the context of a data store, but it's also an important factor in overall application design. It's usually a bad idea to keep a transaction open during user interaction because there's a real danger of conflicts when the user tries to make her update.



Module-3

5	a.	Explain with a neat diagram, the partitioning and combining in Map reduce.	(10 Marks)
	b.	Explain two stages Map- Reduce example, with a neat diagram.	(10 Marks)

OR

6	a.	Explain Basic Map – Reduce, with a neat diagram.	(07 Marks)
	b.	How are calculations composed in Map – Reduce? Explain with a neat diagram.	(08 Marks)
	c.	What are Key value Stores? List out some popular key value database. Explain how all the data is stored in a single bucket of key value data store.	(05 Marks)

5a. In the simplest form, we think of a map-reduce job as having a single reduce function. **The outputs from all the map tasks running on the various nodes are concatenated together and sent into the reduce.**

While this will work, there are things we can do to **increase the parallelism and to reduce the data transfer (see Figure 7.3)**.

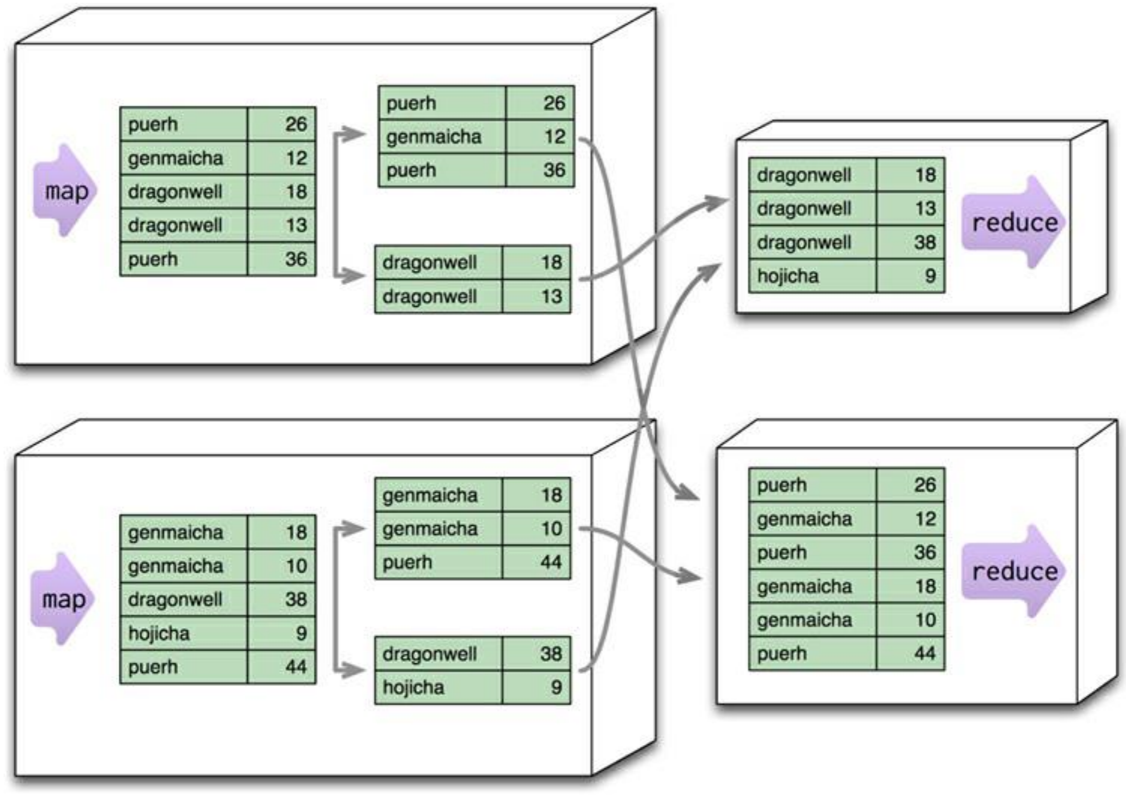


Figure 7.3. Partitioning allows reduce functions to run in parallel on different keys.

1. The first thing we can do is **increase parallelism by partitioning the output of the mappers**.
2. **Each reduce function operates on the results of a single key**. This is a limitation—it means you can't do anything in the reduce that operates across keys—but it's also a **benefit in that it allows you to run multiple reducers in parallel.**
3. **To take advantage of this, the results of the mapper are divided up based the key on each processing node. Typically, multiple keys are grouped together into partitions.**
4. The framework then takes the data from all the nodes for one partition, **combines it into a single group for that partition, and sends it off to a reducer.**

5. **Multiple reducers can then operate on the partitions in parallel, with the final results merged together.** (This step is also called "shuffling," and the partitions are sometimes referred to as "buckets" or "regions.")
6. The next problem we can deal with is the amount of data being moved from node to node between the map and reduce stages. Much of this data is repetitive, consisting of multiple key-value pairs for the same key.
7. **A combiner function cuts this data down by combining all the data for the same key into a single value** (see Figure 7.4).
8. **A combiner function is, in essence, a reducer function—indeed, in many cases the same function can be used for combining as the final reduction.**
9. **The reduce function needs a special shape for this to work: Its output must match its input. We call such a function a combinable reducer.**

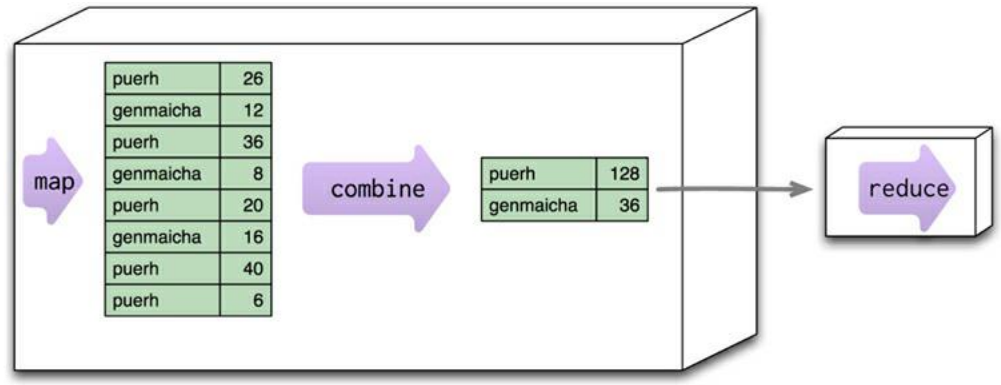
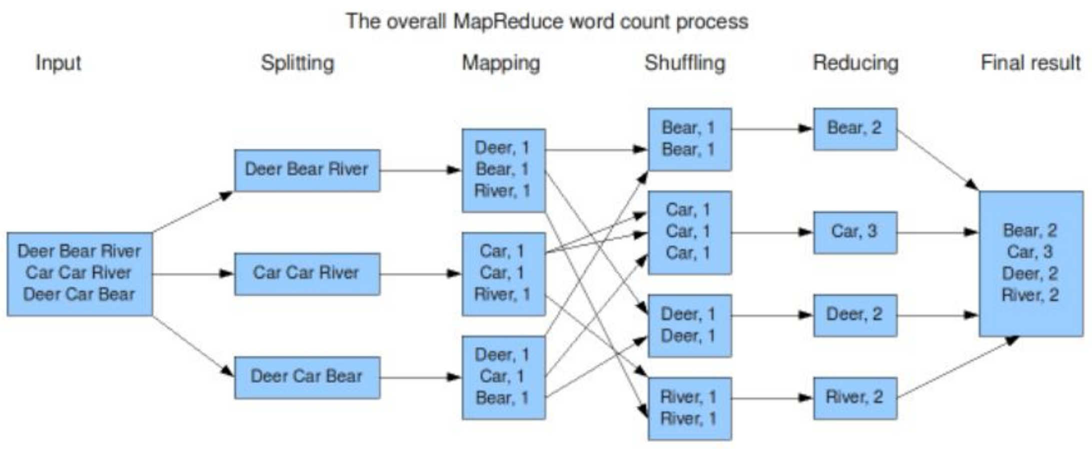


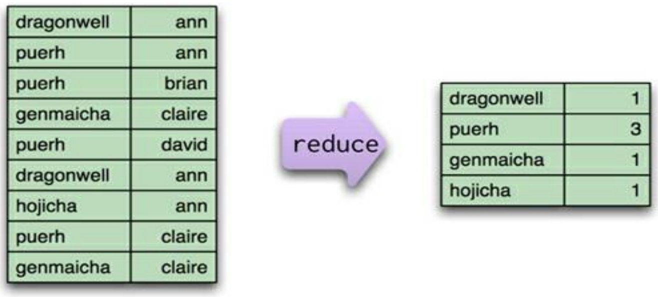
Figure 7.4. Combining reduces data before sending it across the network.



Not required to add above diagram in the answer.

Reduce functions not combinable

10. **Not all reduce functions are combinable**. Consider a function that counts the number of **unique customers for a particular product**. The map function for such an operation would need to emit the product and the customer**. The reducer can then combine them and count how many times each customer appears for a particular product, emitting the product and the count (see Figure 7.5).** But this reducer's output is different from its input, so it can't be used as a combiner. You can still run a combining function here: one that just eliminates duplicate product-customer pairs, but it will be different from the final reducer.



dragonwell	ann
puerh	ann
puerh	brian
genmaicha	claire
puerh	david
dragonwell	ann
hojicha	ann
puerh	claire
genmaicha	claire

dragonwell	1
puerh	3
genmaicha	1
hojicha	1

Figure 7.5. This reduce function, which counts how many unique customers order a particular tea, is not combinable.

11. **When you have combining reducers, the map-reduce framework can safely run not only in parallel (to reduce different partitions), but also in series to reduce the same partition at different times and places.**
12. In addition to allowing **combining to occur on a node before data transmission, combining can be started before mappers have finished.**
13. This provides a bit of extra flexibility to the map-reduce processing. **Some map-reduce frameworks require all reducers to be combining reducers, which maximizes this flexibility**.
14. If you need to do a non-combining reducer with one of these frameworks, then separate the processing into pipelined map-reduce steps.

5b. Consider an example where **we want to compare the sales of products for each month in 2011 to the prior year**. To do this, we'll break the calculations down into two stages.

- The first stage will produce records showing the aggregate figures for a single product in a single month of the year.
- The second stage then uses these as inputs and produces the result for a single product by comparing one month's results with the same month in the prior year (see Figure 7.8).

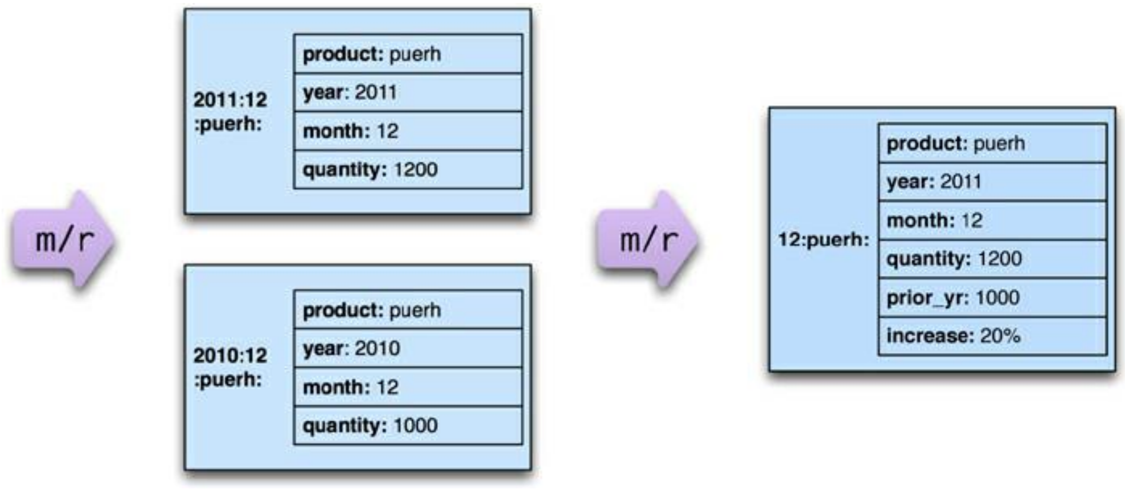


Figure 7.8. A calculation broken down into two map-reduce steps, which will be expanded in the next three figures

- A first stage (Figure 7.9) would read the original order records and output a series of key-value pairs for the sales of each product per month.

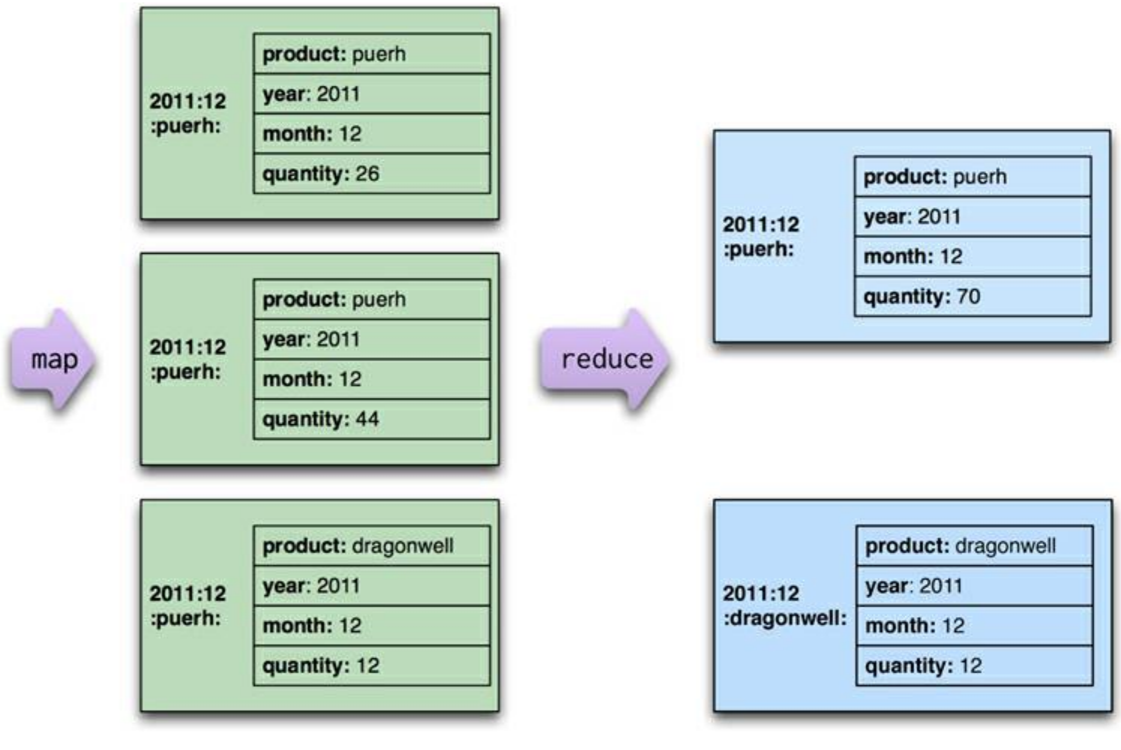


Figure 7.9. Creating records for monthly sales of a product

- The only new feature is using a composite key so that we can reduce records based on the values of multiple fields.
- The second-stage mappers (Figure 7.10) **process this output depending on the year.** A 2011 record populates the current year quantity while a 2010 record populates a prior year quantity. Records for earlier years (such as 2009) don't result in any mapping output being emitted.

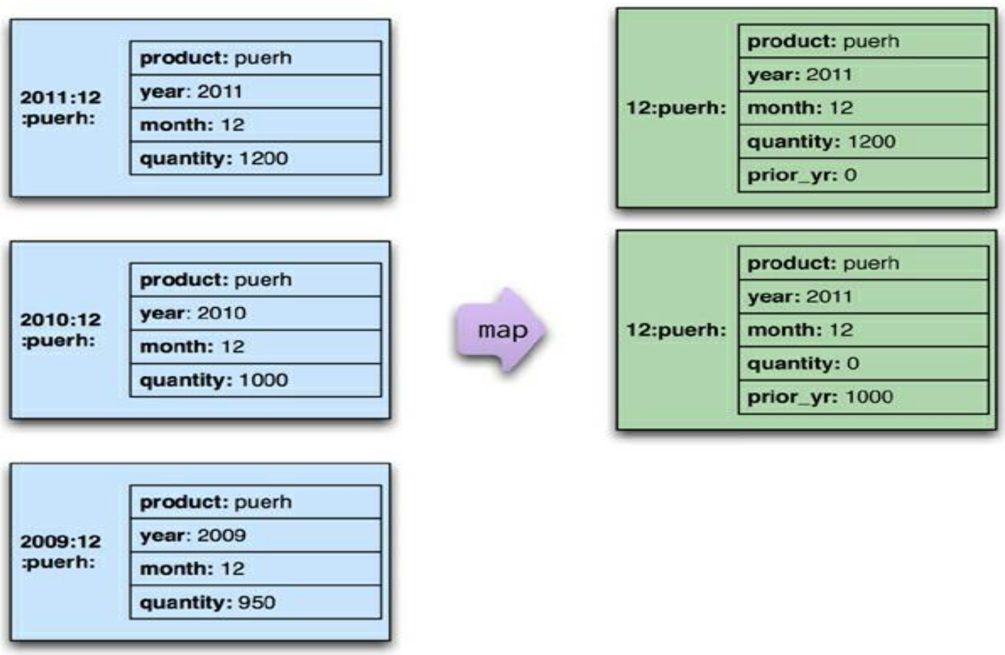


Figure 7.10. The second stage mapper creates base records for year-on-year comparisons.

- The reduce in this case (Figure 7.11) is a merge of records, where combining the values by summing allows two different year outputs to be reduced to a single value (with a calculation based on the reduced values thrown in for good measure).

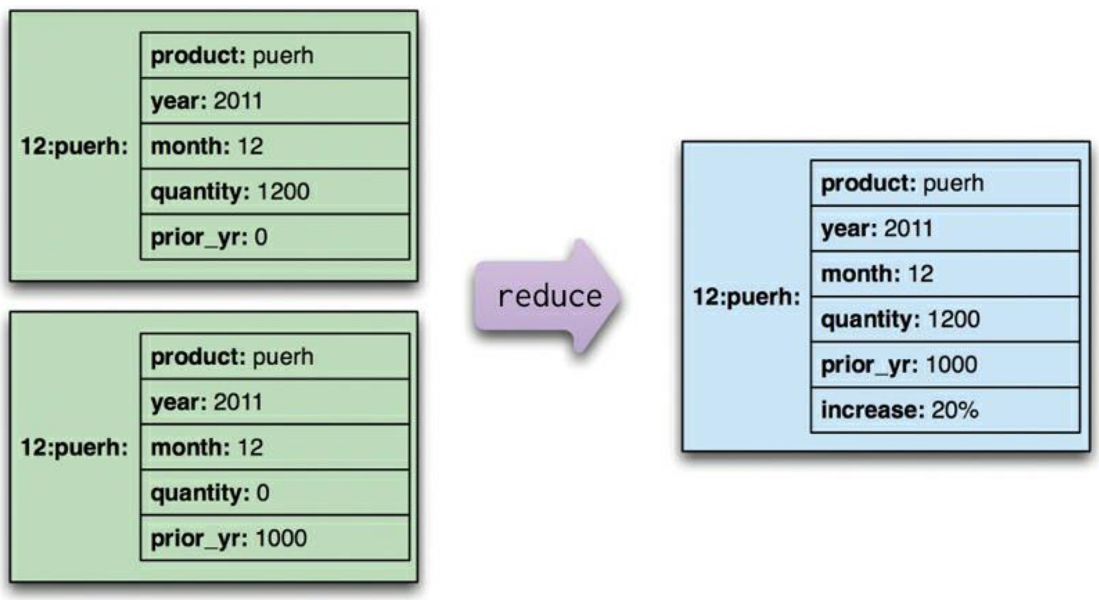


Figure 7.11. The reduction step is a merge of incomplete records.

Advantages of breaking down the process into steps

- **Decomposing this report into multiple map-reduce steps makes it easier to write**.

- Like many transformation examples, once you've identify a transformation framework that makes it easy to compose steps**, it's easier to compose many small steps together than try to cram heaps of logic into a single step.**
- Another advantage is that **the intermediate output may be useful for different outputs too, so you can get some reuse**. This reuse is important as it saves time both in programming and in execution.
  The intermediate records can be saved in the data store, forming a materialized view.
  Early stages of map-reduce operations are particularly valuable to save since they often represent the heaviest amount of data access, so building them once as a basis for many downstream uses saves a lot of work.
  As with any reuse activity, it's important to build them out of experience with real queries, as speculative reuse rarely fulfills its promise.
  So it's important to look at the forms of various queries as they are built and factor out the common parts of the calculations into materialized views.
- Map-reduce is a pattern that can be implemented in any programming language. However, the constraints of the style make it a good fit for languages specifically designed for map-reduce computations.
- **Apache Pig [Pig**], an offshoot of the Hadoop [Hadoop] project, is a language specifically built **to make it easy to write map-reduce programs**. It certainly makes it much easier to
  work with Hadoop than the underlying Java libraries.
- In a similar vein, if you want to **specify mapreduce programs using an SQL-like syntax**, there is **hive [Hive],** another Hadoop offshoot.
- The map-reduce pattern is important to know about even outside of the context of NoSQL
  databases.
  Google's original map-reduce system operated on files stored on a distributed file system
  — an approach that's used by the **open-source Hadoop project**.
- The result of Map-Reduce calculation is inherently well-suited to running on a cluster.
- When dealing with high volumes of data, you need to take a cluster-oriented approach. Aggregate-oriented databases fit well with this style of calculation.
- In the next few years many more organizations will be processing the volumes of data that demand a cluster-oriented solution—and the map-reduce pattern will see more and more use.

6a. Consider customer and order lines. Order is an aggregate unit. Sine there are many orders the data needed to be sharded on multiple systems. However, **sales analysis people want to see a product and its total revenue for the last seven days**. This report doesn't fit the aggregate structure which is the downside of using aggregates. In order to get the product revenue report system have to visit every machine in the cluster and examine many records on each machine.

This is exactly the kind of situation that calls for map-reduce.

1. The **first stage** in a map-reduce job is the map. **A map is a function whose input is a single aggregate and whose output is a bunch of keyvalue pairs**. In this case, the input would be an order. The output would be key-value pairs corresponding to the line items. Each one would have the **product ID as the key and an embedded map with the quantity and price as the values (**see Figure 7.1).

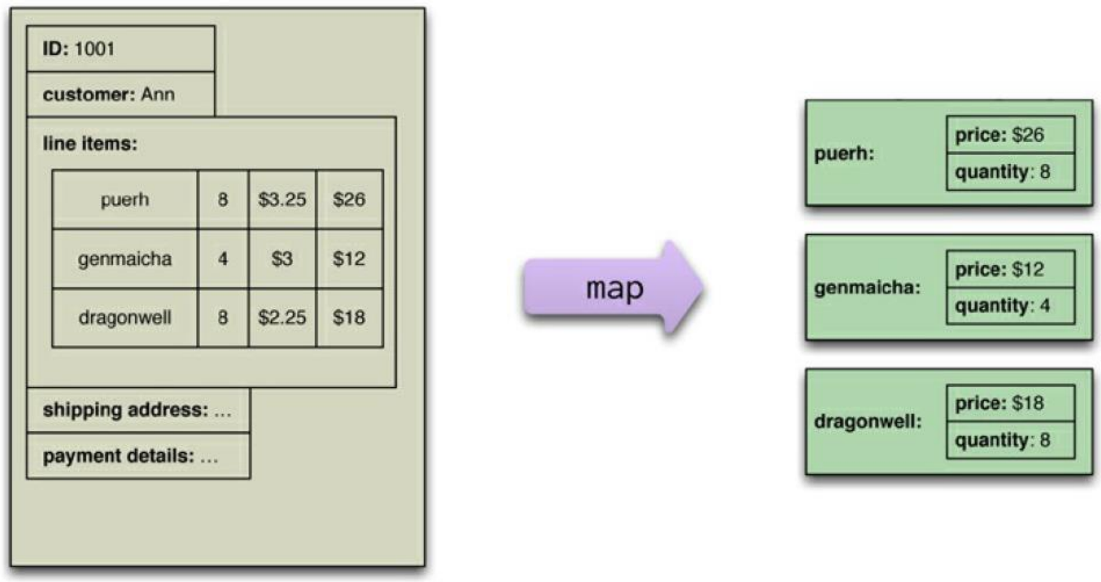


Figure 7.1. A map function reads records from the database and emits key-value pairs.

2. Each application of the map function is **independent** of all the others.
3. This allows them to be **safely parallelizable**, so that a map-reduce framework can create efficient **map tasks** on each node and freely allocate each order to a map task.
4. **A map operation only operates on a single record; the reduce function takes multiple map outputs with the same key and combines their values**. So, a map function might yield 1000 line items from orders for **"Database Refactoring**"; the reduce function would reduce down to one, with the totals for the quantity and revenue.

5. While the map function **is limited to working only on data from a single aggregate, the reduce function can use all values emitted for a single key** (see Figure 7.2).
6. The map-reduce framework arranges for **map tasks to be run on the correct nodes** to process all the documents and for **data to be moved to the reduce function**.
7. To make it easier to write the reduce function**, the framework collects all the values for a single pair and calls the reduce function once with the key and the collection of all the values for that key.** So to run a map-reduce job, you just need to write these two functions.

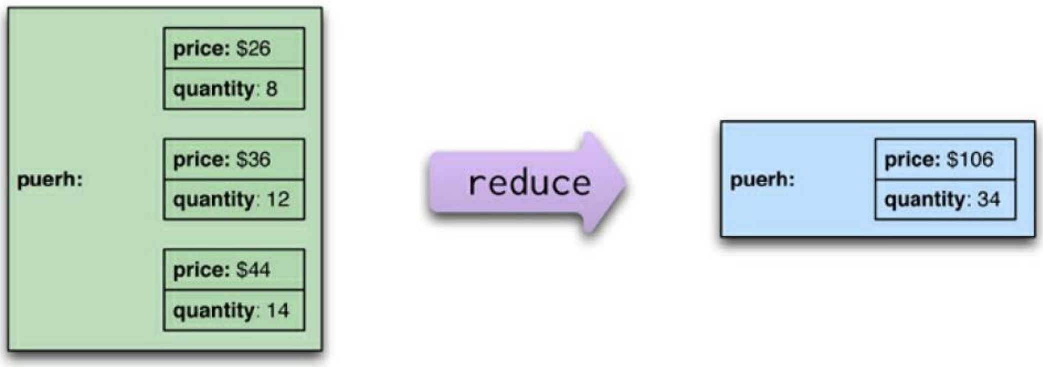


Figure 7.2. A reduce function takes several key-value pairs with the same key and aggregates them into one.

- **6b. Map-Reduce constraint:**
  Within a map task, **you can only operate on a single aggregate**. **Within a reduce task, you can only operate on a single key**. This means you have to think differently about structuring your programs so they work well within these constraints.
  One simple limitation is: **need to structure your calculations around operations that fit in well with the notion of a reduce operation**.

- Example.
  Suppose we want to know **the average ordered quantity of each product**. I need to take total amount and the count of orders from each group, combine those, and then calculate the average from the combined sum and count (see Figure 7.6).

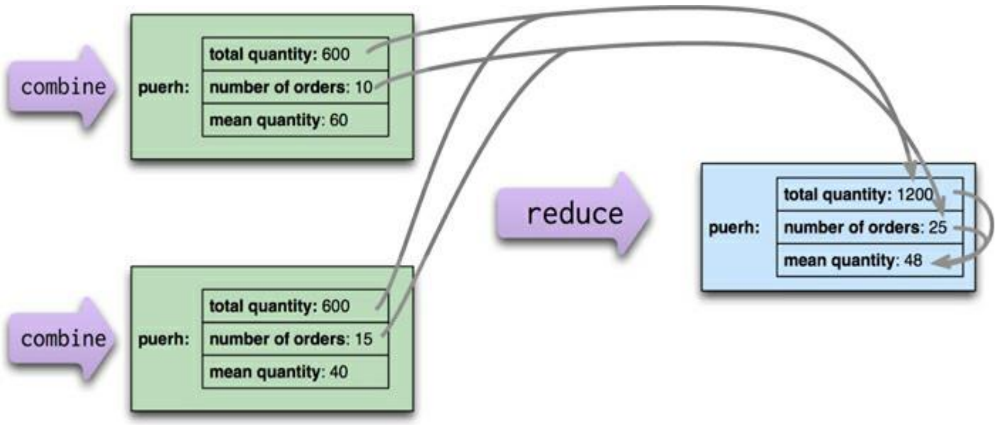


Figure 7.6. When calculating averages, the sum and count can be combined in the reduce calculation, but the average must be calculated from the combined sum and count

- To make a count, the mapping function will emit count fields with a value of 1, which can be summed to get a total count (see Figure 7.7).

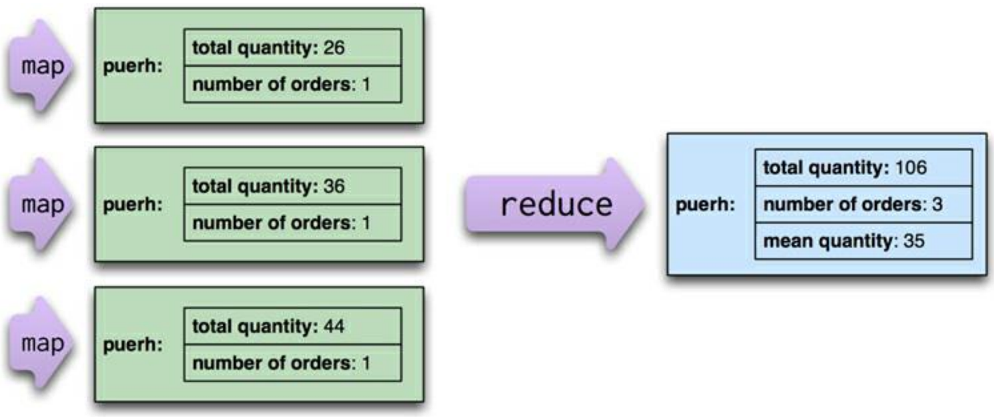


Figure 7.7. When making a count, each map emits 1, which can be summed to get a total.

==
=====

6c. **A key-value store is a simple hash table, primarily used when all access to the database is via primary key.** Think of a table in a traditional RDBMS with two columns, such as ID and NAME, the ID column being the key and NAME column storing the value. In an RDBMS, the NAME column is restricted to storing data of type String. The application can provide an ID and VALUE and persist the pair; if the ID already exists the current value is overwritten, otherwise a new entry is created. Let's look at how terminology compares in Oracle and Riak.

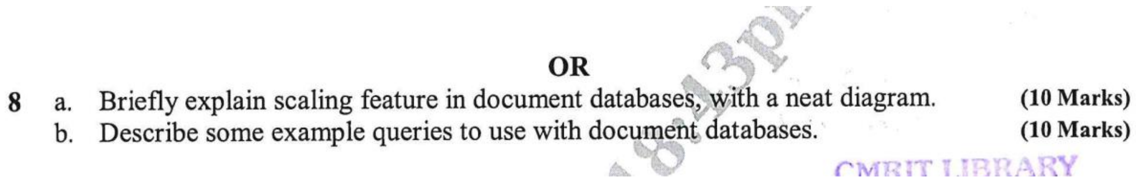
Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

==

Module-4

7 a. What are Document Databases? Explain with an example. List and explain any 2 features of document databases. **(10 Marks)**

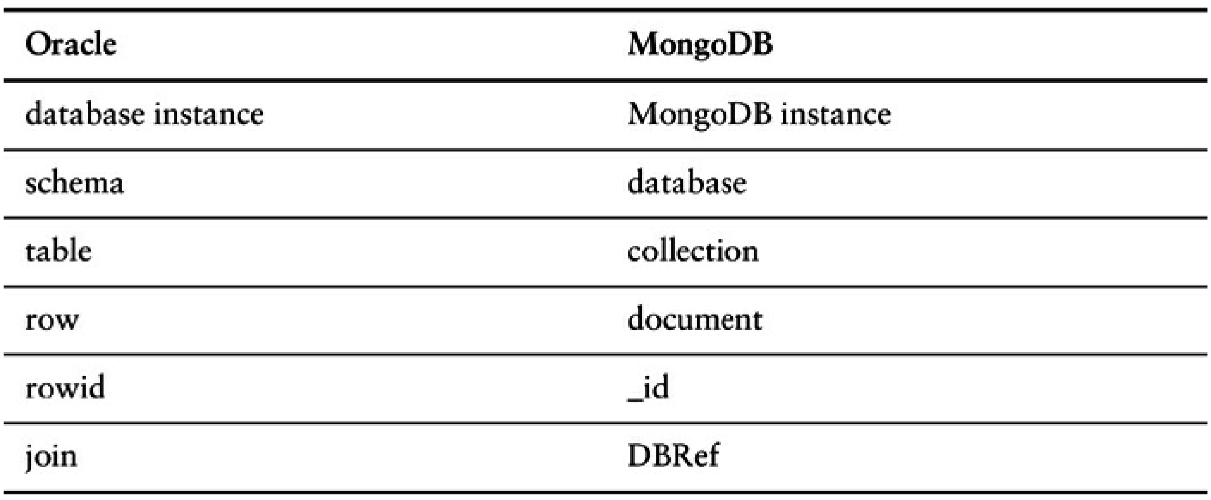
b. Elaborate the suitable use cases of document databases. When document databases are not suitable? Explain. **(10 Marks)**


OR

8 a. Briefly explain scaling feature in document databases, with a neat diagram. (10 Marks)

b. Describe some example queries to use with document databases. (10 Marks)

- 7a. Documents are the main concept in document databases. **The database stores and retrieves documents, which can be XML, JSON, BSON, and so on.**
- **These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values.**
- **The documents stored are similar to each other but do not have to be exactly the same**. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable.
- Let's look at how terminology compares in Oracle and MongoDB.



Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

- The _id is a special field that is found on all documents in Mongo, just like ROWID in Oracle. In MongoDB, _id can be assigned by the user, as long as it is unique.

9.1. What Is a Document Database?

```
{ "firstname": "Martin",
  "likes": [ "Biking",
             "Photography" ],
  "lastcity": "Boston",
  "lastVisited":
}
```

The above document can be considered a row in a traditional RDBMS. Let’s look at another document:

```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ],
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R" }
  ],
  "lastcity": "Chicago"
}
```

- Looking at the documents, we can see that they are similar, but **have differences in attribute names. This is allowed in document databases**
- **The schema of the data can differ across documents, but these documents can still belong to the same collection**—unlike an RDBMS where every row in a table has to follow the same schema.

  Eg. We represent a list of citiesvisited as an array, or a list of addresses as list of documents embedded inside the main document.
- **Embedding child documents as subobjects inside documents provides for easy access and better performance.**

  Eg. If you look at the documents, you will see that some of **the attributes are similar**, such as firstname or city. At the same time, there are attributes in the second document which **do not exist in the first document**, such as addresses, while likes is in the first document but not the second.
- **This kind of different representation of data is not the same as in RDBMS where every column has to be defined, and if it does not have data it is marked as empty or set to null**.

- In documents, **there are no empty attributes**; if a given attribute is **not found, assume that it was not set or not relevant to the document**.
- Documents allow for new **attributes to be created without the need to define them or to change the existing documents.**
- Some of the popular document databases we have seen are MongoDB [MongoDB], CouchDB [CouchDB], Terrastore [Terrastore], OrientDB [OrientDB], RavenDB [RavenDB], and of course the well-known and often reviled Lotus Notes [Notes Storage Facility] that uses document storage.

Consistency

- Consistency in MongoDB database is configured by using the **replica sets** and **choosing to wait for the writes to be replicated to all the slaves or a given number of slaves.**
- **Every write can specify the number of servers the write has to be propagated to before it returns as successful.**
- A command like db.runCommand({ getlasterror : 1 , w : "majority" }) tells the database how strong is the consistency you want.
- For example*, if you have one server and specify the w as majority, the write will return immediately since there is only one node. If you have three nodes in the replica set and specify w as majority, the write will have to complete at a minimum of two nodes before it is reported as a success. You can increase the w value for stronger consistency but you will suffer on write performance, since now the writes have to complete at more nodes.*
- **Replica sets also allow you to increase the read performance** by allowing reading from slaves by **setting slaveOk**; this parameter can be set on the connection, or database, or collection, or individually for each operation.

```
Mongo mongo = new Mongo("localhost:27017");
mongo.slaveOk();
```

Here we are setting slaveOk per operation, so that we can decide which operations can work with data from the slave node.

```
DBCollection collection = getOrderCollection();
BasicDBObject query = new BasicDBObject();
query.put("name", "Martin");
DBCursor cursor = collection.find(query).slaveOk();
```

- Similar to various options available for read, **you can change the settings to achieve strong write consistency, if desired.**
- *By default,* ***a write is reported successful once the database receives it; you can change this so as to wait for the writes to be synced to disk or to propagate to two or more slaves.***
- **This is known as WriteConcern:** ***You make sure that certain writes are written to the master and some slaves by setting WriteConcern to REPLICAS_SAFE****.* Shown below is code where we are setting the WriteConcern for all writes to a collection:

```
DBCollection shopping = database.getCollection("shopping");
shopping.setWriteConcern(REPLICAS_SAFE);
```

- **WriteConcern can also be set per operation by specifying it on the save command:**

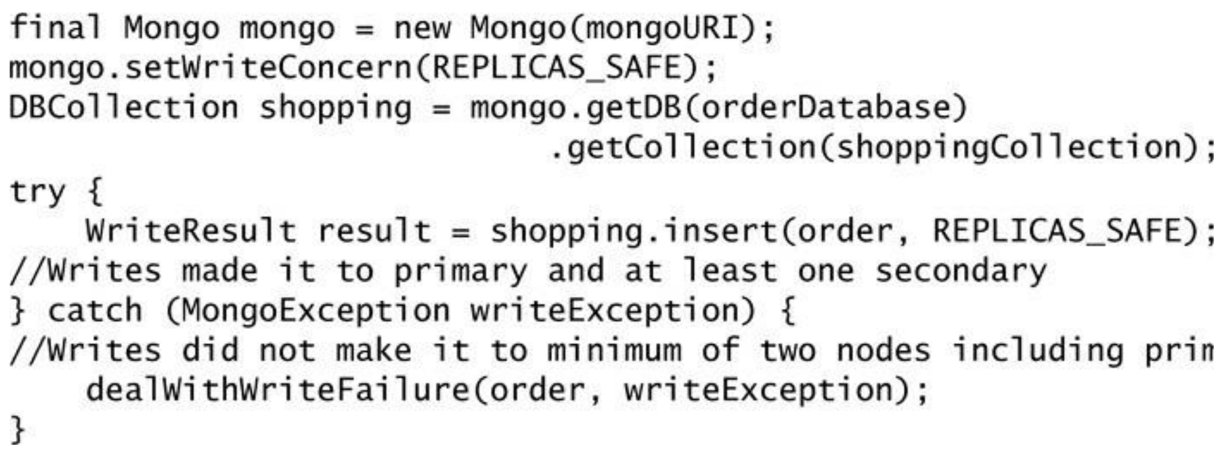
```
WriteResult result = shopping.insert(order, REPLICAS_SAFE);
```

- There is a tradeoff that you need to carefully think about, based on your application needs and business requirements, to decide what settings make sense for slaveOk during read or what safety level you desire during write with WriteConcern.

9.2.2. Transactions

Transactions, in the traditional RDBMS sense, mean that you can start modifying the database with insert, update, or delete commands over different tables and then decide if you want to keep the changes or not by using commit or rollback. These constructs are generally not available in NoSQL solutions—a write either succeeds or fails.

- Transactions at the single-document level are known as **atomic transactions**.
- **Transactions involving more than one operation are not possible**, although there are products such as RavenDB that do support transactions across multiple operations.
- **By default, all writes are reported as successful**. A finer control over the write can be achieved by using **WriteConcern** parameter.
- **We ensure that order is written to more than one node before it's reported successful by using WriteConcern.REPLICAS_SAFE.**
- Different levels of WriteConcern let you choose the safety level during writes; for example, when writing log entries, you can use lowest level of safety, WriteConcern.NONE.



```
final Mongo mongo = new Mongo(mongoURI);
mongo.setWriteConcern(REPLICAS_SAFE);
DBCollection shopping = mongo.getDB(orderDatabase)
                             .getCollection(shoppingCollection);
try {
    WriteResult result = shopping.insert(order, REPLICAS_SAFE);
//Writes made it to primary and at least one secondary
} catch (MongoException writeException) {
//Writes did not make it to minimum of two nodes including prin
    dealWithWriteFailure(order, writeException);
}
```

==

7b. **Event Logging**

- Applications have different event logging needs; within the enterprise, there are many different applications that want to log events.
- Document databases can store all these different types of events and can act as a central data store for event storage.
- This is especially true when the type of data being captured by the events keeps changing.
- Events can be sharded by the name of the application where the event originated or by the type of event such as order_processed or customer_logged.

9.3.2. Content Management Systems, Blogging Platforms

Since document databases have **no predefined schemas and usually understand JSON** documents, **they work well in content management systems or applications for publishing websites, managing user comments, user registrations, profiles, web-facing documents.**

9.3.3. Web Analytics or Real-Time Analytics

Document databases can store data for real-time analytics; **since parts of the document can be updated, it's very easy to store page views or unique visitors, and new metrics can be easily added without schema changes.**

9.3.4. E-Commerce Applications

E-commerce applications often need to have flexible schema for products and orders, as well as the ability to evolve their data models without expensive database refactoring or data migration

9.4. When Not to Use

There are problem spaces where document databases are not the best solution.

9.4.1. Complex Transactions Spanning Different Operations

If you need to have **atomic cross-document operations, then document databases may not be for you**. However, there are some document databases that do support these kinds of operations, such as RavenDB.

9.4.2. Queries against Varying Aggregate Structure

- Flexible schema means that the database does not enforce any restrictions on the schema.
- Data is saved in the form of application entities. If you need to query these entities ad hoc, your queries will be changing (in RDBMS terms, this would mean that as you join criteria between tables, the tables to join keep changing).
- **Since the data is saved as an aggregate, if the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity—basically, you need to normalize the data.**
- In this scenario, document databases may not work.

==
===

8 a) The idea of scaling is **to add nodes or change data storage without simply migrating the database to a bigger box**.

- We are **not talking about making application changes to handle more load; instead, we are interested in what features are in the database so that it can handle more load**.

  Scaling for reads (explain horizontal scaling for reads.)
- **Scaling for heavy-read loads can be achieved by adding more read slaves, so that all the reads can be directed to the slaves**. Given a heavy-read application, with our 3-node replica-set cluster, we can add more read capacity to the cluster as the read load increases just by adding more slave nodes to the replica set to execute reads with the *slaveOk* flag (Figure 9.2). This is horizontal scaling for reads.

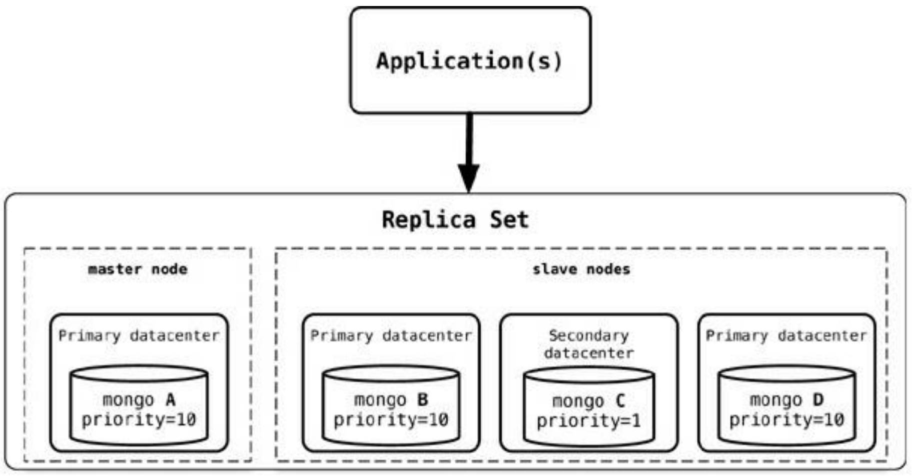


Figure 9.2. Adding a new node, mongo D, to an existing replica-set cluster

- Once the new node, mongo D, is started, it needs to be added to the replica set.

  rs.add("mongod:27017");

  When a new node is added, it will sync up with the existing nodes, join the replica set as secondary node, and start serving read requests.

- An advantage of this setup is that **we do not have to restart any other nodes, and there is no downtime** for the application either.

  Scaling for writes

- When **we want to scale for write**, we can start sharding the data.
- Sharding is similar to partitions in RDBMS where we split data by value in a certain column, such as state or year. With RDBMS, partitions are usually on the same node, so the client application does not have to query a specific partition but can keep querying the base table; the RDBMS takes care of finding the right partition for the query and returns the data.
- **In sharding, the data is also split by certain field, but then moved to different Mongo nodes.**
- **The data is dynamically moved between nodes to ensure that shards are always balanced.**
- We can add more nodes to the cluster and increase the number of writable nodes, enabling horizontal scaling for writes.

```
db.runCommand( { shardcollection : "ecommerce.customer", key : {firstname : 1} } )
```

- **Splitting the data** on the first name of the customer **ensures that the data is balanced across the shards for optimal write performance**; furthermore**, each shard can be a replica set ensuring better read performance within the shard** (Figure 9.3).
- When we add a new shard to this existing sharded cluster, the data will now be balanced across four shards instead of three.

- **As all this data movement and infrastructure refactoring is happening, the application will not experience any downtime,** although the cluster may not **perform optimally** when large amounts of data are being moved to rebalance the shards.

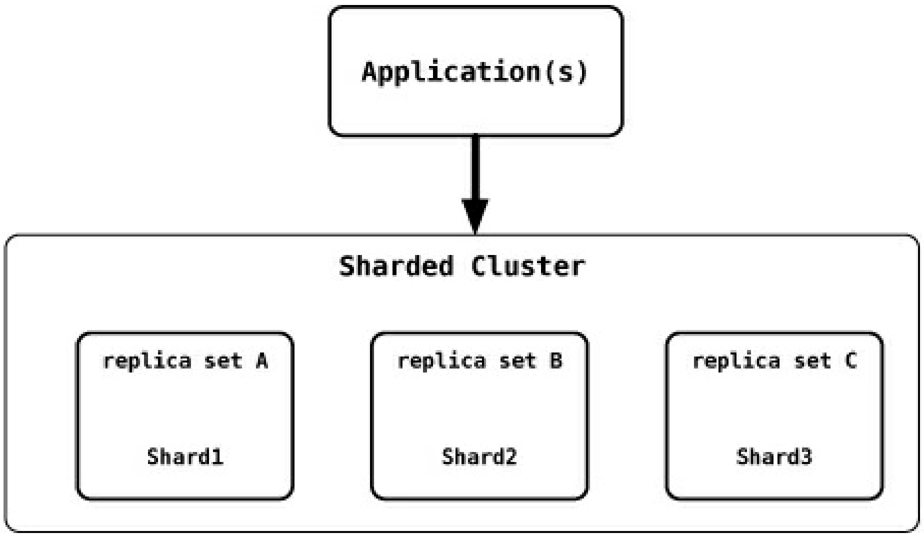


Figure 9.3. MongoDB sharded setup where each shard is a replica set

- **The shard key plays an important role**. You may want to place your MongoDB database shards closer to their users, so sharding based on user location may be a good idea.
- When sharding by customer location, all user data for the East Coast of the USA is in the shards that are served from the East Coast, and all user data for the West Coast is in the shards that are on the West Coast.

===
===

8b. Document databases provide different query features.

- **CouchDB allows you to query via views**—complex queries on documents which can be either **materialized or dynamic**.
- With CouchDB, if you need to aggregate the number of reviews for a product as well as the average rating, you could add a view implemented via map-reduce to return the count of reviews and the average of their ratings.
- **When there are many requests, you don't want to compute the count and average for every request; instead you can add a materialized view that precomputes the values and stores the results in the database.**
- These materialized views are updated when queried, if any data was changed since the last update.
- One of the good features of document databases, as compared to key-value stores, is that **we can query the data inside the document without having to retrieve the whole document by its key and then introspect the document. This feature brings these databases closer to the RDBMS query model.**

MongoDB (queries in mongodb**)**

- MongoDB has a **query language which is expressed via JSON and has constructs such as**
  $query for the where clause,
  $orderby for sorting the data, or
  $explain to show the execution plan of the query.
- There are many more constructs like these that can be combined to create a MongoDB query.
- Let's look at certain queries that we can do against MongoDB. **Suppose we want to return all the documents in an order collection (all rows in the order table).** The SQL for this would be:

SELECT * FROM order

The equivalent query in Mongo shell would be:

db.order.find()

Selecting the orders for a single customerId of 883c2c5b4e5b would be:

SELECT * FROM order WHERE customerId = "883c2c5b4e5b"

The equivalent query in Mongo to get all orders for a single customerId of 883c2c5b4e5b:

db.order.find({"customerId":"883c2c5b4e5b"})

Similarly, selecting orderId and orderDate for one customer in SQL would be:

SELECT orderId, orderDate FROM order WHERE customerId = "883c2c5b4e5b"

and the equivalent in Mongo would be:

db.order.find({customerId:"883c2c5b4e5b"},{orderId:1,orderDate:1})

- Similarly, queries to count, sum, and so on are all available.
- **Since the documents are aggregated objects, it is really easy to query for documents that have to be matched using the fields with child objects.**

Let's say we want to query for all the orders where one of the items ordered has a name like Refactoring. The SQL for this requirement would be:

SELECT * FROM customerOrder, orderItem, product

WHERE

customerOrder.orderId = orderItem.customerOrderId

AND orderItem.productId = product.productId

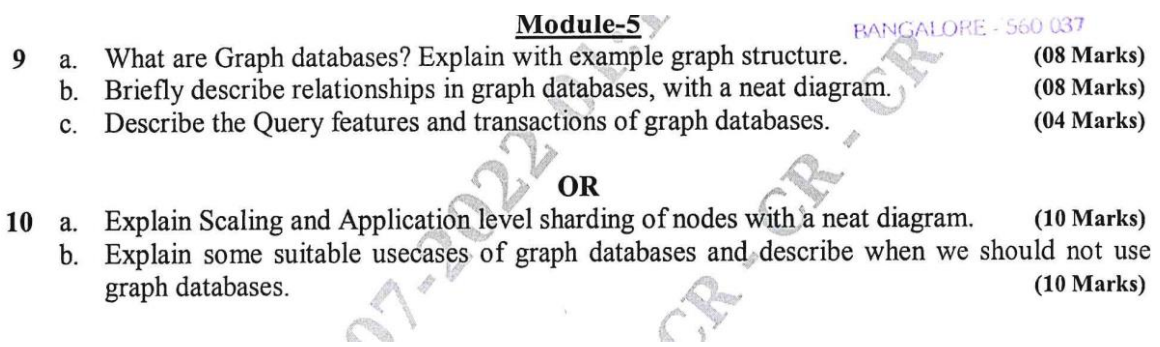
AND product.name LIKE '%Refactoring%'

and the equivalent Mongo query would be:

db.orders.find({"items.product.name":/Refactoring/})

The query for MongoDB is simpler because the objects are embedded inside a single document and you can query based on the embedded child documents.

===



Module-5

9 a. What are Graph databases? Explain with example graph structure. (08 Marks)
b. Briefly describe relationships in graph databases, with a neat diagram. (08 Marks)
c. Describe the Query features and transactions of graph databases. (04 Marks)

OR

10 a. Explain Scaling and Application level sharding of nodes with a neat diagram. (10 Marks)
b. Explain some suitable usecases of graph databases and describe when we should not use graph databases. (10 Marks)

- 9a. Graph databases allow you **to store entities and relationships between these entities. Entities are also known as nodes, which have properties**. Think of a **node as an instance of an object in the application.**
- **Relations are known as edges that can have properties. Edges have directional significance;** nodes are organized by relationships which allow you to find interesting patterns between the nodes.
- **The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.**
- In the example graph in Figure 11.1, we see a bunch of nodes related to each other. Nodes are entities that have properties, such as name. The node of Martin is actually a node that has property of name set to Martin.

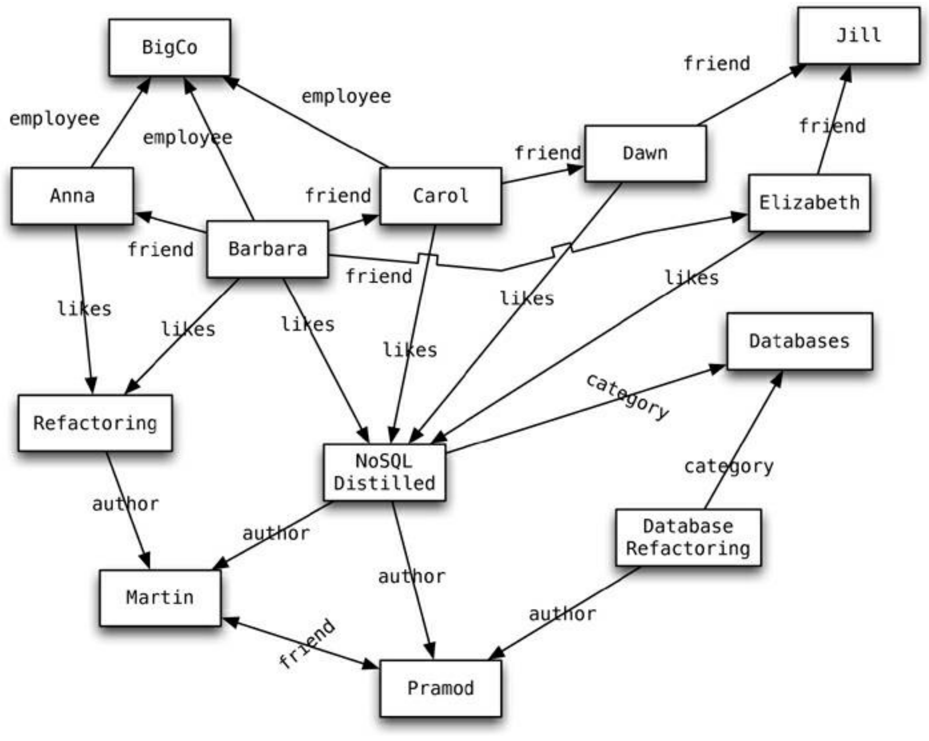


Figure 11.1. An example graph structure

We also see that edges have types, such as likes, author, and so on. These properties let us
organize the nodes; for example, the nodes Martin and Pramod have an edge connecting them with a relationship type of friend.

- **Edges can have multiple properties**. We can assign a property of since on the friend relationship type between Martin and Pramod.
- **Relationship types have directional significance; the friend relationship type is bidirectional but 'likes' is not**. When Dawn likes NoSQL Distilled, it does not automatically mean NoSQL Distilled likes Dawn.
- Once we have a graph of these nodes and edges created, we can query the graph in many ways, such as
 "get all nodes employed by Big Co that like NoSQL Distilled."
- **A query on the graph is also known as traversing the graph. An advantage of the graph databases is that we can change the traversing requirements without having to change the nodes or edges**.
- If we want to "get all nodes that like NoSQL Distilled," we can do so without having to change the existing data or the model of the database, because we can traverse the graph any way we like.
- **Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship** ("who is my manager" is a common example).
- **Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases.**
- Similarly, in relational databases we model the graph beforehand based on the Traversal we want; if the Traversal changes, the data will have to change.
- In graph databases**, traversing the joins or relationships is very fast.** The relationship between nodes is not calculated at query time but is actually persisted as a relationship.
- **Traversing persisted relationships is faster than calculating them for every query.**
- Nodes can have different types of relationships between them, allowing you to both represent **relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access.**
- **Since there is no limit to the number and kind of relationships a node can have, all they can be represented in the same graph database.**

==

- 9b. In Neo4J, creating a graph is as simple as creating two nodes and then creating a relationship. Let's create two nodes, Martin and Pramod:

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");

Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");
```

We have assigned the name property of the two nodes the values of Martin and Pramod. Once we have more than one node, we can create a relationship:

```
martin.createRelationshipTo(pramod, FRIEND);

pramod.createRelationshipTo(martin, FRIEND);
```

- **We have to create relationship between the nodes in both directions, for the direction of the relationship matters:**
  For example, a product node can be liked by user but the product cannot like the user. This directionality helps in designing a rich domain model (Figure 11.2). Nodes know about INCOMING and OUTGOING relationships that are traversable both ways.

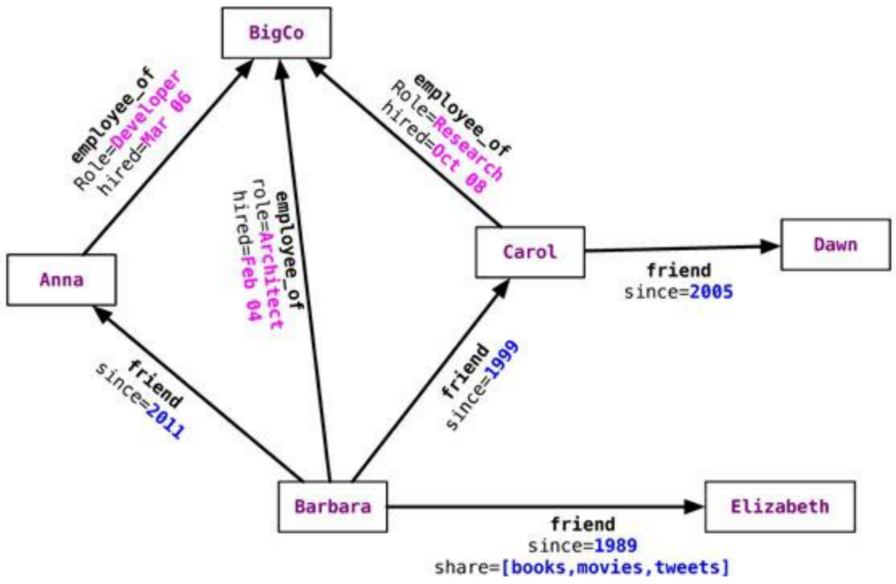


Figure 11.2. Relationships with properties

- **Relationships are first-class citizens in graph databases; most of the value of graph databases is derived from the relationships.**
- **Relationships don’t only have a type, a start node, and an end node, but can have properties of their own**. Using these properties on the relationships, we can **add intelligence to the relationship—for example, since when did they become friends, what is the distance between the nodes, or what aspects are shared between the nodes. These properties on the relationships can be used to query the graph.**
- **Since most of the power from the graph databases comes from the relationships and their properties, a lot of thought and design work is needed to model the relationships** in the domain that we are trying to work with.
- **Adding new relationship types is easy; changing existing nodes and their relationships is similar to data migration, because these changes will have to be done on each node and each relationship in the existing data.**

==

9c. Transactions

- Neo4J is **ACID-compliant**.
- **Before changing any nodes or adding any relationships to existing nodes, we have to start a transaction**.
- Without wrapping operations in transactions, we will get NotInTransactionException.
- Read operations can be done without initiating a transaction.

```
Transaction transaction = database.beginTx();
try {
    Node node = database.createNode();
    node.setProperty("name", "NoSQL Distilled");
    node.setProperty("published", "2012");
    transaction.success();
} finally {
    transaction.finish();
}
```

- In the above code, **we started a transaction on the database, then created a node and set properties on it. We marked the transaction as success and finally completed it by finish.**
- A transaction has to be marked as success, otherwise **Neo4J assumes that it was a failure and rolls it back when finish is issued**. **Setting success without issuing finish also does not commit the data to the database.**
- This way of managing transactions has to be remembered when developing, as it differs from the standard way of doing transactions in an RDBMS.

Query Features **(****most imp 20 marks : algo , queries and explanation)**

- Graph databases are supported by **query languages such as Gremlin** [Gremlin].
- Gremlin is a domain specific language **for traversing graphs**; it **can traverse all graph databases that implement the Blueprints [Blueprints] property graph.**
- Neo4J also has **the Cypher [Cypher] query language for querying the graph**.
- Outside these query languages**, Neo4J allows you to query the graph for properties of the nodes, traverse the graph, or navigate the nodes relationships using language bindings.**
- **Properties of a node can be indexed using the indexing service.** Similarly, **properties of relationships or edges can be indexed, so a node or edge can be found by the value**.
- Indexes should be queried to find the starting node to begin a traversal. Let's look at searching for the node using node indexing.
- If we have the graph shown in Figure 11.1, we can index the nodes as they are added to the database, or we can index all the nodes later by iterating over them. **We first need to create an index for the nodes using the IndexManager**.

```
Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
```

- We are indexing the nodes for the name property. Neo4J uses **Lucene** [Lucene] as its indexing service.
  More about Lucene indexing : https://youtu.be/vLEvmZ5eEz0

- **When new nodes are created, they can be added to the index.**

```
Transaction transaction = graphDb.beginTx();
try {
    Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
    nodeIndex.add(martin, "name", martin.getProperty("name"));
    nodeIndex.add(pramod, "name", pramod.getProperty("name"));
    transaction.success();
} finally {
    transaction.finish();
}
```

- **Adding nodes to the index is done inside the context of a transaction.**
- **Once the nodes are indexed, we can search them using the indexed property. If we search for the node with the name of Barbara, we would query the index for the property of name to have a value of Barbara**.

```
Node node = nodeIndex.get("name", "Barbara").getSingle();
```

- **We get the node whose name is Martin; given the node, we can get all its relationships.**

```
Node martin = nodeIndex.get("name", "Martin").getSingle();
 allRelationships = martin.getRelationships();
```

- **We can get both INCOMING or OUTGOING relationships.**

```
incomingRelations = martin.getRelationships(Direction.INCOMING);
```

- We can also apply directional filters on the queries when querying for a relationship. With the graph in Figure 11.1, **if we want to find all people who like NoSQL Distilled, we can find the NoSQL Distilled node and then get its relationships with Direction.INCOMING**. At this point we can also add the type of relationship to the query filter, since **we are looking only for nodes that LIKE NoSQL Distilled**.

```
Node nosqlDistilled = nodeIndex.get("name",
"NoSQL Distilled").getSingle();
relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
for (Relationship relationship : relationships) {
likesNoSQLDistilled.add(relationship.getStartNode());
}
```

==

10 a. Scaling

In NoSQL databases, one of the commonly used scaling techniques is sharding, where data is split
and distributed across different servers.

- **With graph databases, sharding is difficult, as graph databases are not aggregate-oriented but relationship-oriented.**
- **Since any given node can be related to any other node, storing related nodes on the same server is better for graph traversal.**

- **Traversing a graph when the nodes are on different machines is not good for performance.**

Knowing this limitation of the graph databases, **we can still scale them using some common techniques** described by Jim Webber [Webber Neo4J Scaling].
Generally speaking, **there are three ways to scale graph databases.**

- We can **add enough RAM to the server so that the working set of nodes and relationships is held entirely in memory.** This technique is only helpful if the dataset that we are working with will fit in a realistic amount of RAM.
- We can **improve the read scaling of the database by adding more slaves with read-only access to the data, with all the writes going to the master**. This pattern of writing once and reading from many servers is a proven technique in MySQL clusters and is really useful when the dataset is large enough to not fit in a single machine's RAM, but small enough to be replicated across multiple machines.
- **Slaves can also contribute to availability and read-scaling, as they can be configured to never become a master, remaining always read-only.**
- When the dataset size makes replication impractical, **we can shard the data from the application side using domain-specific knowledge**. For example, nodes that relate to the North America can be created on one server while the nodes that relate to Asia on

another. This application-level sharding needs to understand that nodes are stored on physically
different databases (Figure 11.3).

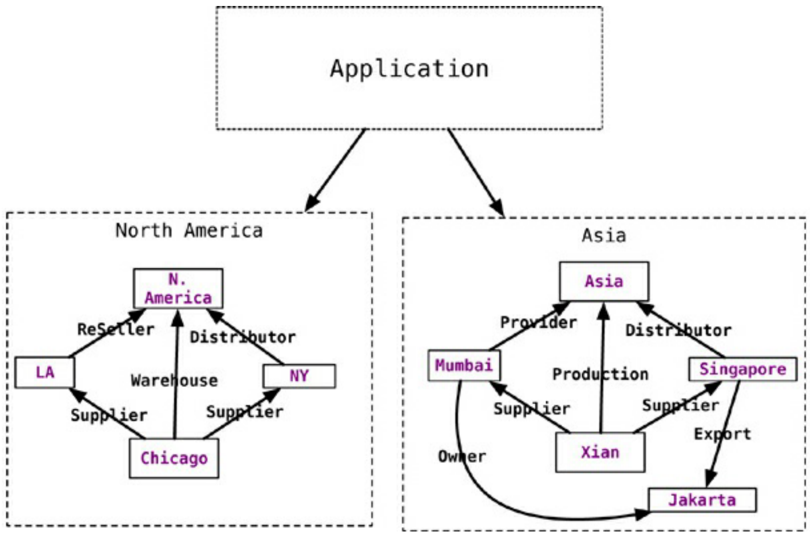


Figure 11.3. Application-level sharding of nodes

==

10b. Suitable Use Cases

Let's look at some suitable use cases for graph databases.

11.3.1. Connected Data

- Social networks are where graph databases can be deployed and used very effectively.

- These social graphs don’t have to be only of the friend kind; for example, they can represent employees, their knowledge, and where they worked with other employees on different projects.
- Any link-rich domain is well suited for graph databases.
- If you have relationships between domain entities from different domains (such as social, spatial, commerce) in a single database, you can make these relationships more valuable by providing the ability to traverse across domains.

11.3.2. Routing, Dispatch, and Location-Based Services

- Every location or address that has a delivery is a node, and all the nodes where the delivery has to be made by the delivery person can be modeled as a graph of nodes.
- Relationships between nodes can have the property of distance, thus allowing you to deliver the goods in an efficient manner.
- Distance and location properties can also be used in graphs of places of interest, so that your application can provide recommendations of good restaurants or entertainment options nearby.
- You can also create nodes for your points of sales, such as bookstores or restaurants, and notify the users when they are close to any of the nodes to provide location-based services.

11.3.3. Recommendation Engines

- As nodes and relationships are created in the system, they can be used to make recommendations like “your friends also bought this product” or “when invoicing this item, these other items are usually invoiced.” Or, it can be used to make recommendations to travelers mentioning that when other visitors come to Barcelona they usually visit Antonio Gaudi’s creations.
- An interesting side effect of using the graph databases for recommendations is that as the data size grows, the number of nodes and relationships available to make the recommendations quickly increases.
- The same data can also be used to mine information—for example, which products are always bought together, or which items are always invoiced together; alerts can be raised when these conditions are not met.
- Like other recommendation engines, graph databases can be used to search for patterns in relationships to detect fraud in transactions.

11.4. When Not to Use

In some situations, graph databases may not appropriate.

- When you want to update all or a subset of entities—for example, in an analytics solution where all entities may need to be updated with a changed property—graph databases may not be optimal since changing a property on all the nodes is not a straightforward operation.
- Even if the data model works for the problem domain, some databases may be unable to handle lots of data, especially in global graph operations (those involving the whole graph).

==