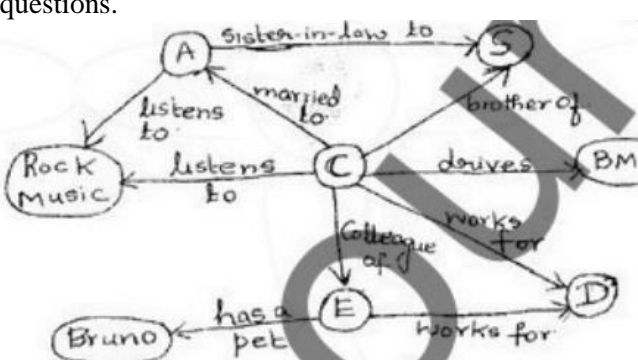


Scheme of Evaluation
Internal Assessment Test 1 – March 2024

Sub:	NoSQL Database						Code:	18CS823	
Date:	16/3/2024	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	ISE

Note: Answer Any five full questions.

Question #		Description	Marks Distribution		Max Marks
1	a)	What is NoSQL? Explain about aggregate data models with neat diagram. Considering example of Relational data models. Definition NoSQL Diagram Explanation	2M 3M 5M	10M	10M
2	a)	Define materialized view. How are they different from views? Briefly explain the two main strategies to build materialized view Define materialized view Two approaches Eager approach Batch approach	2M 4M 4M	10M	10M
3	a)	Explain the data management and access in column family data stores with example Diagram Explanation	3M 3M	6M	10M
3	b)	Briefly describe the value of relational databases. Getting persistent data Concurrency Integration	2M 1M 1M	4M	

4	a)	<p>What are distribution models? Briefly explain two paths of data distribution.</p> <p>Definition distribution models</p> <p>Replication</p> <p>Sharding</p> <p>Explantation</p> <p>Diagram</p>	<p>1M</p> <p>2M</p> <p>2M</p> <p>3M</p> <p>2M</p>	10M	10M
5	a)	<p>Identify the type of conflict in the following scenario, How can it be solved? Alice and Bob share a common Google sheet online. Both read a file. Alice updates the document and forgets to save the file. On the other hand Bob updates the sheet and saves the file. The content updated by Alice overwritten by Bob. The data updated by Alice is lost.</p> <p>Write-write conflict</p> <p>Locks used explanation with example</p>	<p>2M</p> <p>4M</p> <p>4M</p>		
6	a)	<p>Explain briefly impedance mismatch, with a neat diagram 5M</p> <p>Diagram</p> <p>explanation</p>	<p>2M</p> <p>3M</p>	5M	10M
	b)	<p>Use the above diagram and answer the following questions.</p>  <p>a) Who listens to rock music and works for D?</p> <p>b) Who works for D and has married to colleagues?</p> <p>c) Who listen to rock music?</p> <p>d) How are A and S related to each other and also to C?</p> <p>What are the genders of C and A?</p>	1M*1	5M	

Scheme Of Evaluation
Internal Assessment Test 1 –Mar 2024

Sub:	NoSQL Database						Code:	18CS823	
Date:	16/3/2024	Duration:	90mins	Max Marks:	50	Sem:	VIII	Branch:	ISE

Note: Answer Any full five questions

Q 1. What is NoSQL? Explain about aggregate data models with neat diagram. Considering example of Relational data models.

NoSQL, also referred to as “not only SQL” or “non-SQL”, is an approach to database design that enables the storage and querying of data outside the traditional structures found in relational databases.

Example of Relations and Aggregates At this point, an example may help explain what we’re talking about. Let’s assume we have to build an e-commerce website; we are going to be selling items directly to customers over the web, and we will have to store information about users, our product catalog, orders, shipping addresses, billing addresses, and payment data. We can use this scenario to model the data using a relation data store as well as NoSQL data stores and talk about their pros and cons. For a relational database, we might start with a data model shown in Figure 2.1. Figure 2.1. Data model oriented around a relational database (using UML notation

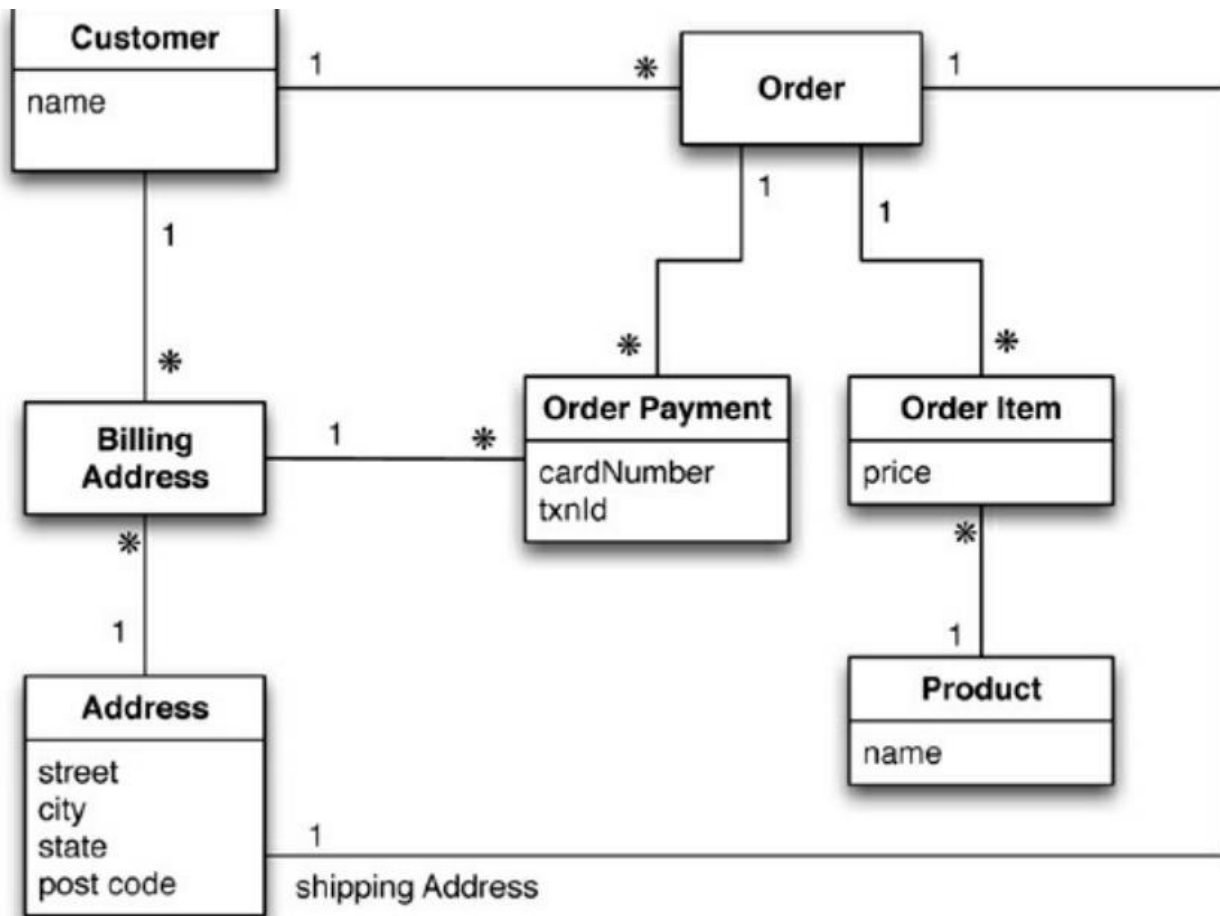


Figure 2.1. Data model oriented around a relational database

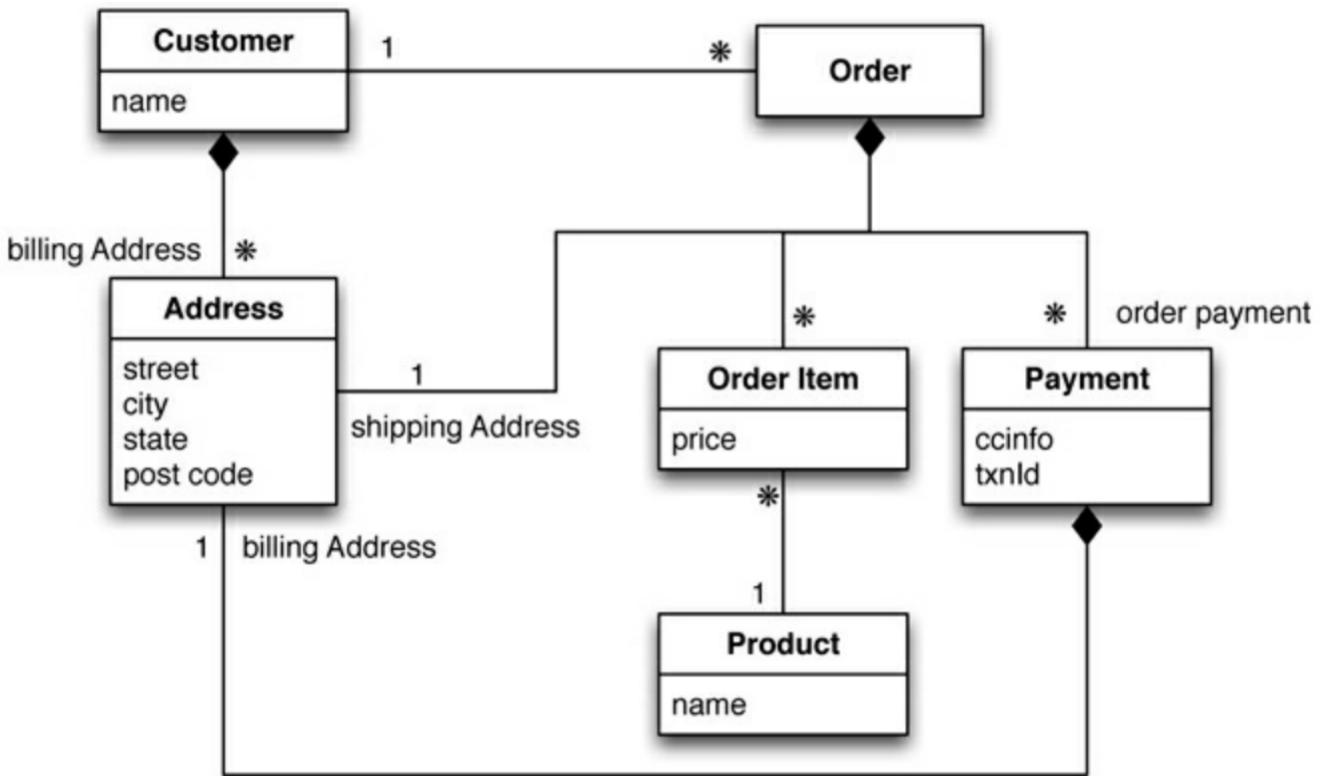


Figure 2.3. An aggregate data model

In this model, we have two main aggregates: customer and order. We've used the black-diamond composition marker 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. A single logical address record appears three times in the example data, but instead of using IDs it's treated as a value and copied each time. This fits the domain where we would not want the shipping address, nor the payment's billing address, to change. In a relational database, we would ensure that the address rows aren't updated for this case, making a new row instead. With aggregates, we can copy the whole address structure into the aggregate as we need to. The link between the customer and the order isn't within either aggregate—it's a relationship between aggregates. Similarly, the link from an order item would cross into a separate aggregate structure for products, which we haven't gone into. We've shown the product name as part of the order item here—this kind of denormalization is similar to the tradeoffs with relational databases, but is more common with aggregates because we want to minimize the number of aggregates we access during a data interaction. The important thing to notice here isn't the particular way we've drawn the aggregate boundary so much as the fact that you have to think about accessing that 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

Q. 2 a) Define materialized view. How are they different from views? Briefly explain the two main strategies to build materialized view

Materialized Views

- To cope with this, materialized views were invented, which are views that are computed in advance and cached on disk. Materialized views are effective for data that is read heavily but can stand being somewhat stale.
- Although NoSQL databases don't have views, they may have precomputed and cached queries, and they reuse the term "materialized view" to describe them. Often, NoSQL databases create materialized views using a map-reduce computation.

There are two strategies to building a materialized view

- The first is the eager approach, where you update the materialized view at the same time you update the base data for it. In this case, adding an order would also update the purchase history aggregates for each product.
- This approach is good when you have more frequent reads of the materialized view than you have writes, and you want the materialized views to be as fresh as possible. The application database approach is valuable here, as it makes it easier to ensure that any updates to base data also update materialized views.
- If you don't want to pay that overhead on each update, you can run batch jobs to update the materialized views at regular intervals as per requirements.
- This approach is good when you have more frequent reads of the materialized view than you have writes, and you want the materialized views to be as fresh as possible. The application database approach is valuable here, as it makes it easier to ensure that any updates to base data also update materialized views.
- If you don't want to pay that overhead on each update, you can run batch jobs to update the materialized views at regular intervals as per requirements.
- You can build materialized views outside the database by reading the data, computing the view, and saving it back to the database.
- More often, databases will support building materialized views themselves.
- In this case, you provide the computation that needs to be done, and the database executes the computation when needed according to some parameters that you configure. This is particularly handy for eager updates of views with incremental map-reduce.

Q 3 a) Explain the data management and access in column family data stores with example

Column-Family Stores

- One of the early and powerful NoSQL databases was Google's BigTable, it is a two-level map. It has been a model that influenced later databases such as HBase and Cassandra.
- These databases with a BigTable-style data model are often referred to as column stores. 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.
- BigTable and its next generation follow this notion of storing groups of columns (column families) together, we refer to this as column-family databases.
- Column-family model is 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 you could do something like `get('1234', 'name')`.

-
•

Column-family databases organize their columns into column families. Each column has to be part of a single column family, and the column acts as a unit for access, with the assumption that data for a particular column family will usually be accessed together.

- This also gives you a couple of ways to think about how the data is structured.
- Row-oriented: Each row is an aggregate (for example, customer with the ID of 1234) with column families representing useful chunks of data (profile, order history) within that aggregate.
- Column-oriented: Each column family defines a record type (e.g., customer profiles) with rows for each of the records. You then think of a row as the join of records in all column families.

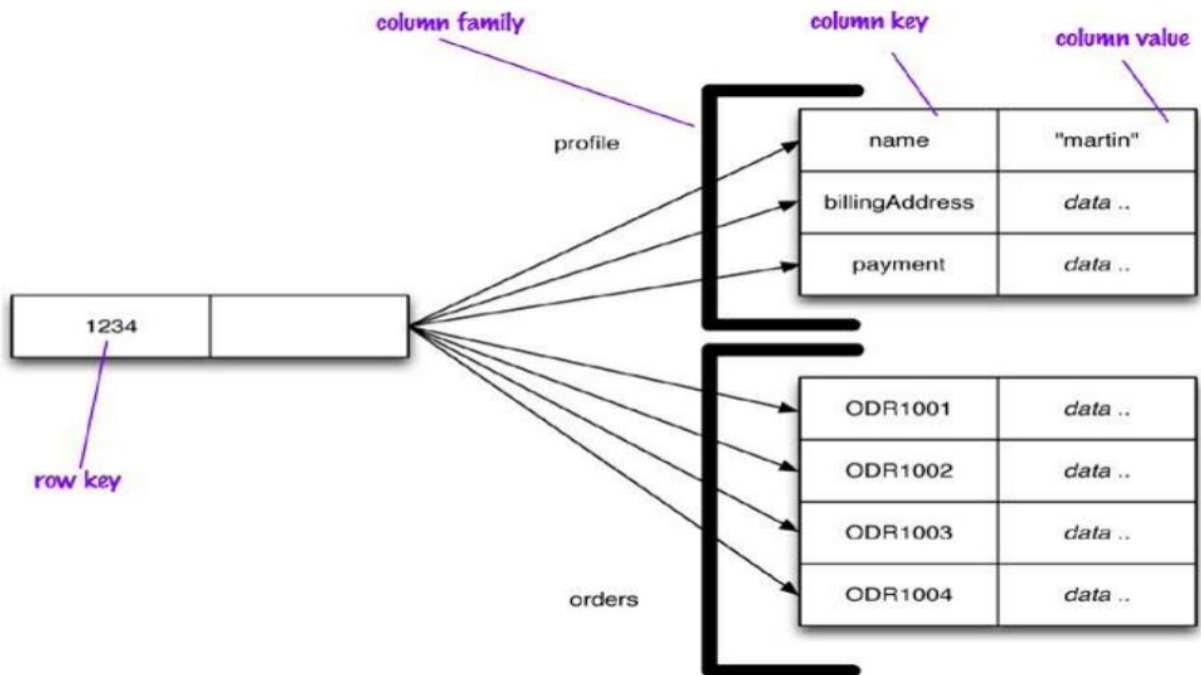


Figure 5: Representing customer info in a column-family structure

- This latter aspect reflects the columnar nature of column-family databases. Since the database knows about these common groupings of data, it can use this information for its storage and access behaviour.
- Cassandra uses the terms “wide” and “skinny.”
- Skinny rows have few columns, with the same columns used across the many different rows.
- In this case, the column family defines a record type, each row is a record, and each column is a field.
- A wide row has many columns (perhaps thousands), with rows having very different columns.
- A wide column family models a list, with each column being one element in that list.

Q. 3 b) Briefly describe the value of relational databases.

1.1 The Value of Relational Databases

Relational databases have become such an embedded part of our computing culture that it's easy to take them for granted. It's therefore useful to revisit the benefits they provide.

1.1.1. Getting at Persistent Data

Probably the most obvious value of a database is keeping large amounts of persistent data. Most computer architectures have the notion of two areas of memory: a fast volatile "main memory" and a larger but slower "backing store." Main memory is both limited in space and loses all data when you lose power or something bad happens to the operating system. Therefore, to keep data around, we write it to a backing store, commonly seen as a disk (although these days that disk can be persistent memory).

The backing store can be organized in all sorts of ways. For many productivity applications (such as word processors), it's a file in the file system of the operating system. For most enterprise applications, however, the backing store is a database. The database allows more flexibility than a file system in storing large amounts of data in a way that allows an application program to get at small bits of that information quickly and easily.

1.1.2. Concurrency

Enterprise applications tend to have many people looking at the same body of data at once, possibly modifying that data. Most of the time they are working on different areas of that data, but occasionally they operate on the same bit of data. As a result, we have to worry about coordinating these interactions to avoid such things as double booking of hotel rooms.

Concurrency is notoriously difficult to get right, with all sorts of errors that can trap even the most careful programmers. Since enterprise applications can have lots of users and other systems all working concurrently, there's a lot of room for bad things to happen. Relational databases help handle this by controlling all access to their data through transactions. While this isn't a cure-all (you still have to handle a transactional error when you try to book a room that's just gone), the transactional mechanism has worked well to contain the complexity of concurrency.

Transactions also play a role in error handling. With transactions, you can make a change, and if an error occurs during the processing of the change you can roll back the transaction to clean things up.

1.1.3. Integration

Enterprise applications live in a rich ecosystem that requires multiple applications, written by different teams, to collaborate in order to get things done. This kind of inter-application collaboration is awkward because it means pushing the human organizational boundaries. Applications often need to use the same data and updates made through one application have to be visible to others.

A common way to do this is shared database integration [Hohpe and Woolf] where multiple applications store their data in a single database. Using a single database allows all the applications to use each others' data easily, while the database's concurrency control handles multiple applications in the same way as it handles multiple users in a single application.

1.1.4. A (Mostly) Standard Model

Relational databases have succeeded because they provide the core benefits we outlined earlier in a (mostly) standard way. As a result, developers and database professionals can learn the basic relational model and apply it in many projects. Although there are differences between different relational databases, the core mechanisms remain the same: Different vendors' SQL dialects are similar, transactions operate in mostly the same way.

Q. 4 a) What are distribution models? Briefly explain two paths of data distribution.

Distribution Models

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.

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. We will now discuss these techniques starting at the simplest and working up to the more complex: first single-server, then master-slave replication, then sharding, and finally peer-to-peer replication.

In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server. The load is balanced out nicely between servers—for example, if we have ten servers, each one only has to handle 10% of the load.

1.2 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** ([Figure 1.1](#)).

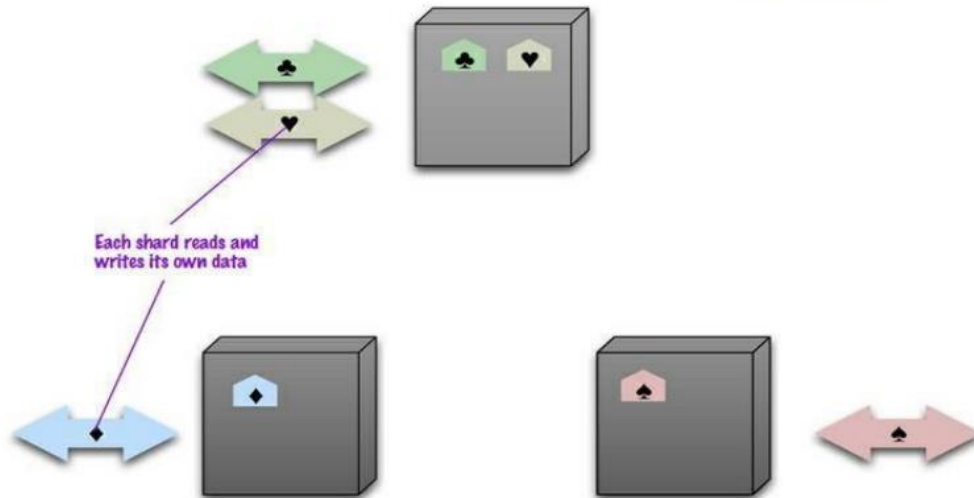


Figure 1.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

In the ideal case, we have different users all talking to different server nodes. Each user only has to talk to one server, so gets rapid responses from that server. The load is balanced out nicely between servers—for example, if we have ten servers, each one only has to handle 10% of the load.

1.3 Master-Slave Replication

With master-slave distribution, you replicate data across multiple nodes. One node is designated as the master, or primary. This master is the authoritative source for the data and is usually responsible for processing any updates to that data. The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master ([Figure 1.2](#)).

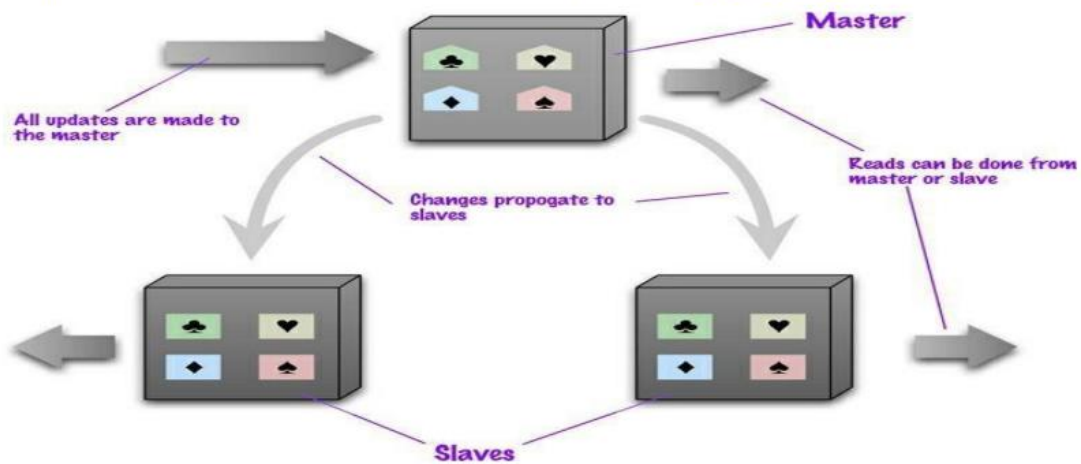


Figure 1.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

Master-slave replication is most helpful for scaling when you have a read-intensive dataset. You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves. You are still, however, limited by the ability of the master to process updates and its ability to pass those updates on. Consequently it isn't such a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a bit with handling the write load.

1.4 Peer-to-Peer Replication

Master-slave replication helps with read scalability but doesn't help with scalability of writes. It provides resilience against failure of a slave, but not of a master. Essentially, the master is still a bottleneck and a single point of failure. Peer-to-peer replication ([Figure 1.3](#)) attacks these problems by not having a master. All the replicas have equal weight, they can all accept writes, and the loss of any of them doesn't prevent access to the data store.

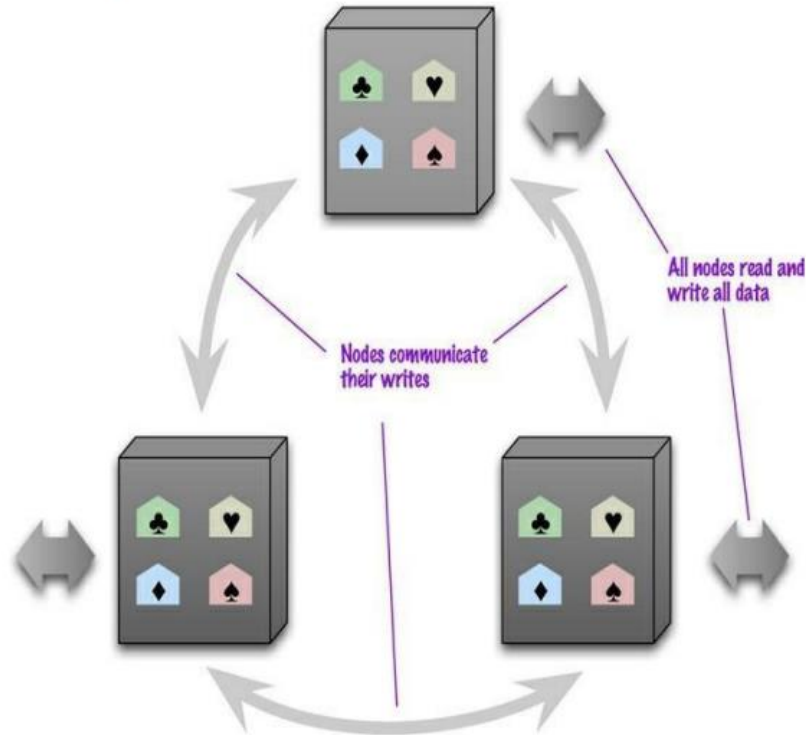


Figure 1.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

Q. 5 a) **Identify the type of conflict in the following scenario, How can it be solved? Alice and Bob share a common Google sheet online. Both read a file. Alice updates the document and forgets to save the file. On the other hand Bob updates the sheet and saves the file. The content updated by Alice overwritten by Bob. The data updated by Alice is lost.**

This issue is called as 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.

In this case Alice's is a lost update. Here the lost update is not a big problem, but often it is. We see this as a failure of consistency because Bob's update was based on the state before Alice's update, yet was applied after it. She forgot to save the file.

Approaches for maintaining consistency in the face of concurrency are often described as **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.

So Alice and Bob would both attempt to acquire the write lock, but only Alice(the first one) would succeed. Bob would then see the result of Bob’s write before deciding whether to make his own update.

There is another optimistic way to handle a **write-write conflict**—save both updates and record that they are in conflict. 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. Maybe you show both values to the user and ask them to sort it out— this is what happens if you update the same contact on your phone and your computer. Alternatively, 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). 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.

Q. 6 a) Explain briefly impedance mismatch, with a neat diagram.

Solution

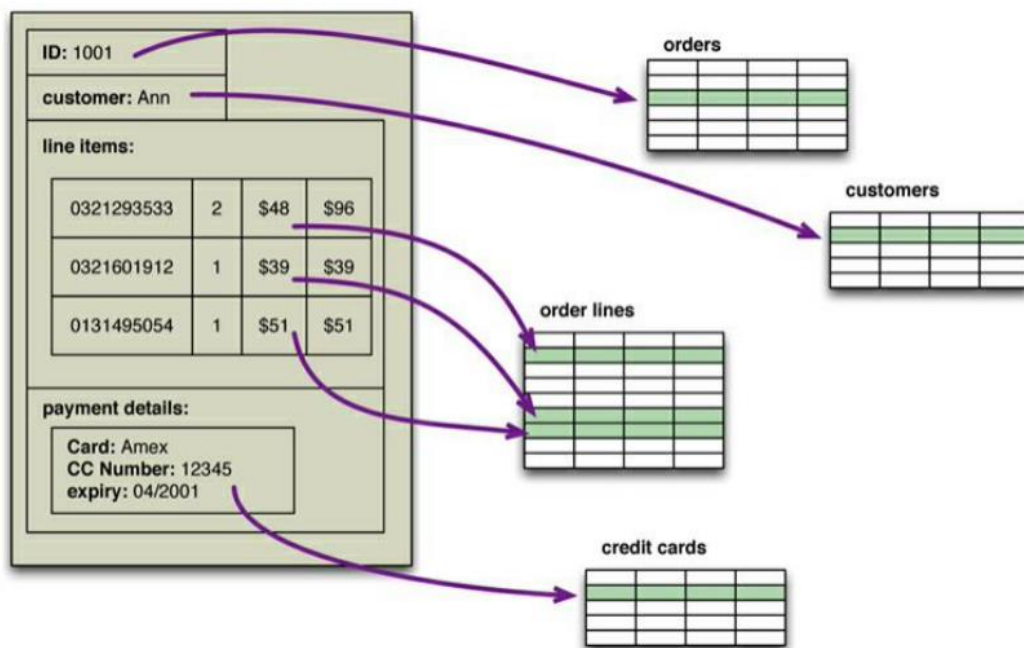
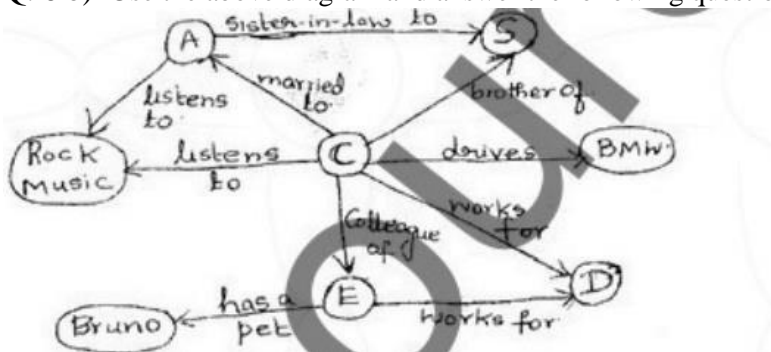


Figure 1: An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

- For Application developers using relational databases, the biggest frustration has been what's commonly called the impedance mismatch: **the difference between the relational model and the in-memory data structures**, as show in Figure: 1
- The relational data model organizes data into a structure of tables. Where a tuple is a set of name-value pairs and a relation is a set of tuples.
- 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.
- 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.
- The impedance mismatch lead to relational databases being replaced with databases that replicate the in- memory data structures to disk. That decade was marked with the growth of object-oriented programming languages, and with them came object-oriented databases—both looking to be the dominant environment for software development in the new millennium. However, while object-oriented languages succeeded in becoming the major force in programming, object-oriented databases faded into obscurity.
- 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, but the mapping problem is still an issue.
- Relational databases continued to dominate the enterprise computing world in the 2000s, but during that decade, cracks began to open in their dominance.

Q. 6 b) Use the above diagram and answer the following questions.



- Who listens to rock music and works for D?
- Who works for D and has married to colleagues?
- Who listen to rock music?
- How are A and S related to each other and also to C?
- What are the genders of C and A?

Ans:

- C
- E
- C, A

- d) A is married C and sister-in law to S
- e) A-Female
- f) C-Male