USN										

Internal Assessment Test 1 – March 2023 Scheme and Solution

Sub:	NOSQL database				Sub Code:	18CS823	Branch:	CSE
Date:	11/03/23	Duration:	90 mins	Max Marks: 50	11/03/23	Sem/Sec:8 A/B/C		90 mins

Answer any FIVE FULL Questions

		MARKS	CO	RBT
1	Discuss the impact of combining sharding and replication? (add suitable diagrams) Sharding impact on different models - 5 marks Replication on different models - 5 marks	[10]	CO 1	L2
2	Describe i) Impedance mismatch with suitable examples ii) Graph database iii) CAP theorem Impedance mismatch with suitable examples- 3 marks Graph database and diagram-4 marks CAP theorem-3 marks	[10]	C O1	L2
3	Assume an organization needs to create an online platform for its employees to create posts and add various images, videos, audio, and comments. Any employee can comment on these posts and rate them. Employees can join different groups as per their interests. The landing page will have a feed of posts that employees can share and interact with. For the given scenario, suggest which type of database implementation (SQL / NoSQL) would be most suitable and specify appropriate reasons for your choice of the database implementation. (tip: write sample table names to show connections) Type of database implementation - 2 marks Specify appropriate reason- 7 marks	[10]	C O2	L4
4	What do you mean by write-write conflict/consistency and read-write consistency? Explain with suitable examples. write-write consistency with example - 3+2 marks read-write consistency with example- 3+2 marks	[10]	C O1	L2
5	Discuss advantages and disadvantages of Schemalessness? Explain Materialised views. advantages and disadvantages of Schemalessness- 5 marks Materialised view- 5 marks	[10]	C O1	L2
6	Explain Key-Value and Document Data Models. Take the scenario of online shopping portal. Identify the applicability of the stated data model for particular section. Justify your answer. Key-Value and Document Data Model explanation- 5 marks Identification and justification of the data model for shopping portal- 5 marks	[10]	C O1	L4

Q1.

1. 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.

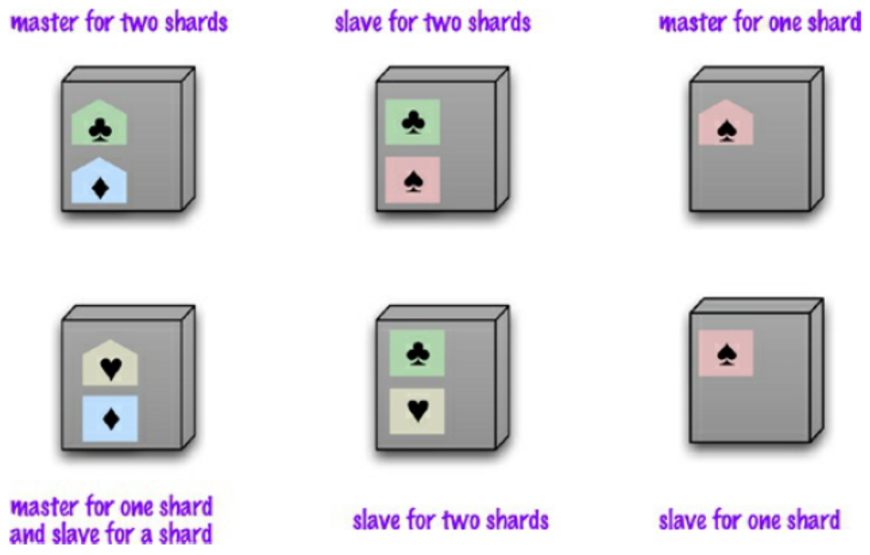


Figure 4.4. Using master-slave replication together with sharding

2. 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.
3. 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).

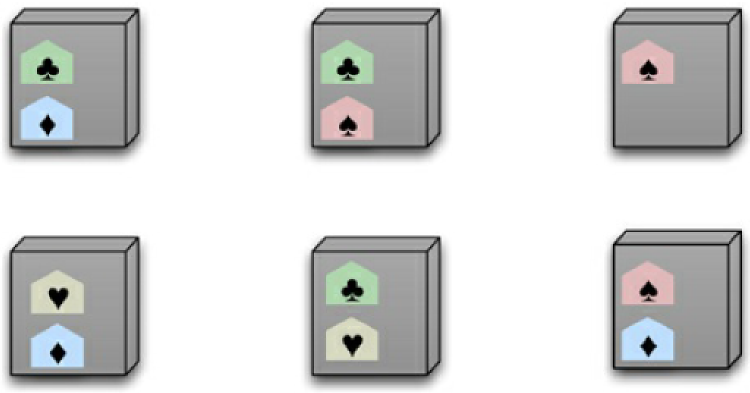


Figure 4.5. Using peer-to-peer replication together with sharding

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.

Sharding does little to improve resilience when used alone. **Although the data is on different nodes, a node failure makes that shard's data unavailable** just as surely as it does for a single-server solution. **The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing**. With a single server it's easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you're more likely to get a node failure. **So in practice, sharding alone is likely to decrease resilience.**

Despite the fact that sharding is made much easier with aggregates, **some databases are intended from the beginning to use sharding,** in which case it's wise to run them on a cluster from the very beginning of development, and certainly in production.
Other databases use sharding as a deliberate step up from a single-server configuration, in which case it's best to start single-server and only use sharding once your load projections clearly indicate that you are running out of headroom.

In any case the step from a single node to sharding is going to be tricky. Keeping sharding too late, so when they were used in production their database became essentially unavailable because **the sharding support consumed all the database resources for moving the data onto new shards**. The lesson here is to use sharding well before you need to—**when you have enough headroom to carry out the sharding.**

Q2. Describe i) Impedance mismatch with suitable examples ii) Graph database iii) CAP theorem

Impedance Mismatch

● 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.

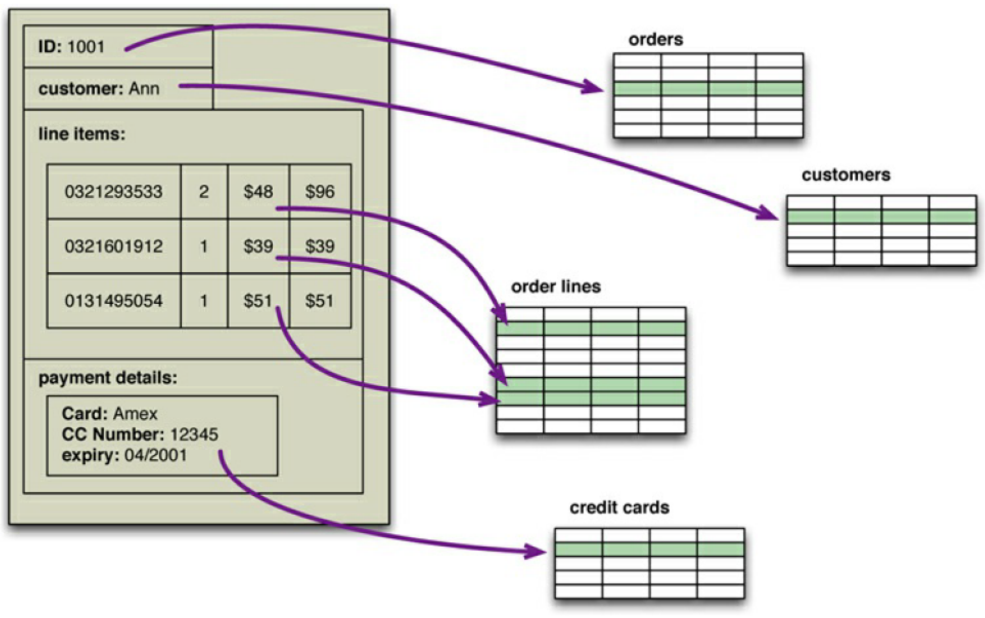
In particular, the values in a relational tuple have to be simple—they cannot contain any structure, such as a nested record or a list. This limitation isn't true for in-memory data structures, which can take on much richer structures than relations.

● 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.

● 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 could not solve problem of impedance mismatch

● Relational databases continued to dominate the enterprise computing world



ii) Graph database

• A characteristic of graph is high flexibility.

• Any number of nodes and any number of edges can be added to expand a graph.

• The complexity is high and the performance is variable with scalability.

• Data store as series of interconnected nodes.

• Graph with data nodes interconnected provides one of the best database system when

relationships and relationship types have critical values**.**

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

● Ideal 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.

iii) CAP Theorem

• Any two properties must be satisfied

• Consistency means all copies have the same value like in traditional DBs.

• Availability means at least one copy is available in case a partition becomes inactive or fails. For example, in web applications, the other copy in the other partition is available.

• Partition means parts which are active but may not cooperate (share) as in distributed DBs.

1. Consistency in distributed databases

• All nodes observe the same data at the same time. Therefore, the operations in one partition of the database should reflect in other related partitions in case of distributed database.

• Operations, which change the sales data from a specific showroom in a table should also reflect in changes in related tables which are using that sales data.

2.Availability

• Availability means that during the transactions, the field values must be available in other partitions of the database so that each request receives a response on success as well as failure. (Failure causes the response to request from the replicate of data).

• Distributed databases require transparency between one another.

• Network failure may lead to data unavailability in a certain partition in case of no replication.

• Replication ensures availability.

3.Partition

• Partition means division of a large database into different databases without affecting the operations on them by adopting specified procedures

----

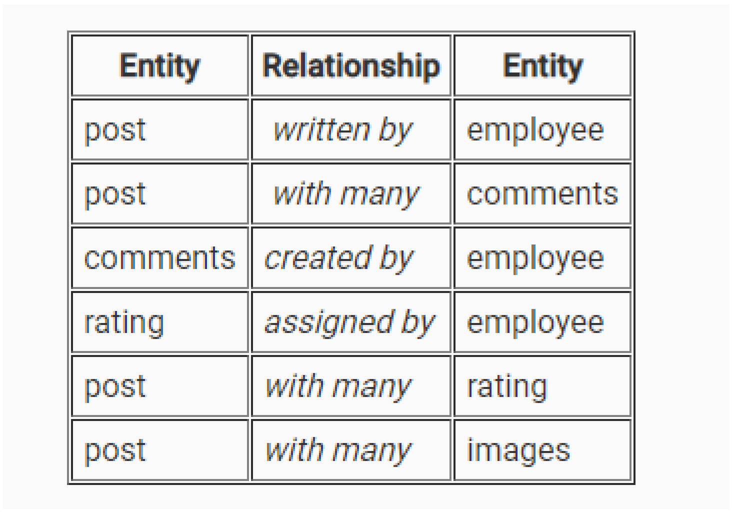
Q3.

Best suitable database implementation would be NoSQL databases.

Following are the reasons:

For the given scenario, there are several entities such as employee, post, comments, images, audios, videos, rating and so on.

You have several relationships that link these entities as shown in the table below:



Entity	Relationship	Entity
post	*written by*	employee
post	*with many*	comments
comments	*created by*	employee
rating	*assigned by*	employee
post	*with many*	rating
post	*with many*	images

Use NoSQL database implementation because,

NoSQL databases are very flexible in structure and can store all types of related data in one place

User can retrieve the whole post with a single query avoiding joins thus increasing the performance

Data on NoSQL databases scale out naturally and hence able to deal with the continuous streaming of posts

Using SQL databases is not suggested as,

Several joins are used to display the post containing various forms of data which is very time consuming

Data is existing in heterogeneous forms that SQL does not support

Continuous streaming of posts that are dynamically loaded onto the screen require thousands of queries to be performed

----

Q4.

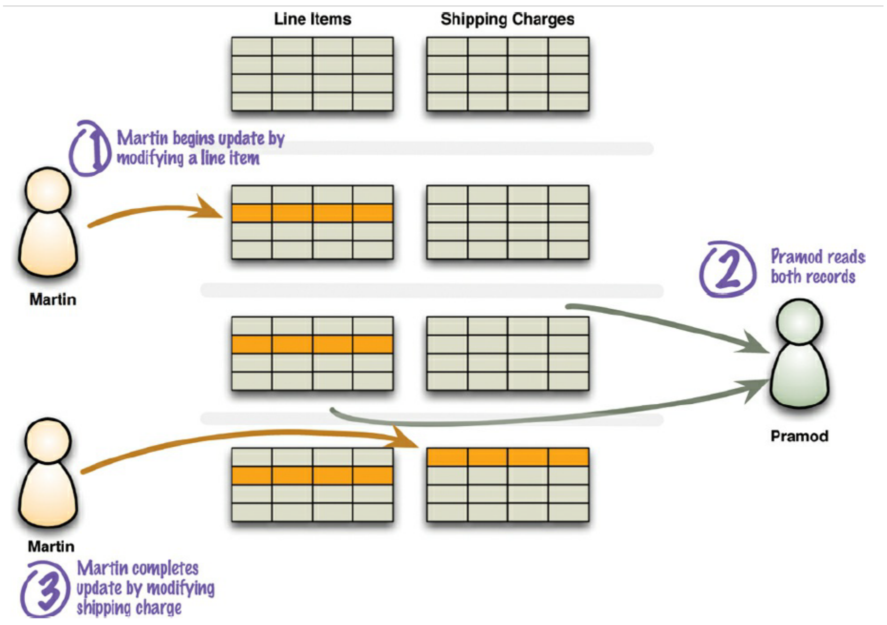
Take an example of considering updating a telephone number. Coincidentally, Martin and Pramod are looking at the company website and notice that the phone number is out of date. **They both have update access, so they both go in at the same time to update the number**. We'll assume **they update it using a slightly different format**. **This issue is called a write-write conflict: two people updating the same data item at the sametime**.

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** beforeMartin's update, yet was applied after it.

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, Pramodthen reads the line items and shipping charge, and then Martin updates the shipping charge. This is an inconsistent read or read-write conflict: Pramod has done a read in the middle of Martin’s write.


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

Q5.

Schemaless Databases

● 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, 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.

Schemaless DB: shifting of schema to code

Essentially, a schemaless database shifts the schema into the application code that accesses it. This becomes problematic if multiple applications, developed by different people, access the same database.

Materialized Views

● In aggregate-oriented data models, if you want to access orders, it's useful to have all the data for an order contained in a single aggregate that can be stored and accessed as a unit.

● In scenarios where if a product manager wants to know how much a particular item has sold over the last couple of weeks? Aggregate oriented data model force to check all the orders in that aggregate, indexing on product works but works against aggregate structure.

● In relational db can access data in different ways using views views. A view is like a relational table (it is a relation) but it's defined by computation over the base tables.

● Views provide a mechanism to hide from the client whether data is derived data orbase data—but can't avoid the fact that some views are expensive to compute.

● 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.

● It's also much more of a central aspect for aggregate-oriented databases than it is for relational systems. Eg MapReduce

There are two rough 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.

Second approach: 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.

● You'll need to understand your business requirements to assess how stable your materialized views can be.

● You can build materialized views outside of 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 incrementalmap-reduce.

● Materialized views can be used within the same aggregate. An order document might include an order summary element that provides summary information about the order so that a query for an order summary does not have to transfer the entire order document.

● Using different column families for materialized views is a common feature of column-family databases.

● An advantage of doing this is that it allows you to update the materialized view within the same atomic operation.

Q6.

Explain Key-Value and Document Data Models. Take the scenario of online shopping portal. Identify the applicability of the stated data model for particular section. Justify your answer.

- **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.**