USN | | | | | | | | | | |

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A++ GRADE BY NAAC

Internal Assessment Test 1 – March-2024

Sub:	NoSql				Sub Code:	18CS823	Branch:	CSE
Date:	16-03-2024	Duration:	90 mins	Max Marks: 50	Sem / Sec:	VIII (A, B & C)		OBE

Answer any FIVE FULL Questions — MARKS | CO | RBT

		MARKS	CO	RBT
1(a)	Explain the necessity for the emergence of NoSql databases in the tech landscape. Detail how NoSql databases specifically address the challenges encountered by social networking companies. Additionally, enumerate the obstacles or limitations associated with NoSql databases and propose strategies to mitigate these barriers.	5	CO1	L2

ANS:

Emergence 2M
CHALLENGES 3M

The term "NoSQL" first made its appearance in the late 90s as the name of an open source relational database. Led by Carlo Strozzi , this database stores its tables as ASCII files, each tuple represented by a line with fields separated by tabs. The name comes from the fact that the database doesn't use SQL as a query language. Instead, the database is manipulated through shell scripts that can be combined into the usual UNIX pipelines. Other than the terminological coincidence, Strozzi's NoSQL had no influence on the databases The usage of "NoSQL" that we recognize today traces back to a meetup on June 11, 2009 in San Francisco organized by Johan, a software developer based in London. (2)

The example of Big Table and Dynamo had inspired a bunch of projects experimenting with alternative data storage, and discussions of these had become a feature of the better software conferences around that time.

Johan was interested in finding out more about some of these new databases while he was in San Francisco for a Hadoop summit. Since he had little time there, he felt that it wouldn't be feasible to visit them all, so he decided to host a meetup where they could all come together and present their work to whoever was interested.

Challenges:

- **Back-ups in Application Consistent**: Determining the sequence of changes across replicas, which is critical in selecting which values should compose a snapshot, is a typical difficulty in quorum-based replication systems. For example, determining a rigorous ordering between two write requests to the same database object that arrived at two distinct nodes at the same time is challenging. As a result of the absence of ordering, determining the most recent value of a database item at any given time is difficult. (3)
- **Database and node failures during backups and restores** : Node failures are common in NoSQL databases since they are designed to grow to hundreds

<table>
<tr>
<td></td>
<td>of nodes. As a result, any backup method must be able to account for data collection failures from down nodes and their influence on quorum consistency. On the other hand, restorations must take into consideration failed cluster nodes and modify data re population correspondingly.
• Analytics and business intelligence: NoSQL was created to fulfill the needs of Web 2.0 applications, and as a result, all of its characteristics are geared toward that goal. Other commercial systems, on the other hand, necessitate moving beyond the insert-read-update-delete cycle. Even the most basic queries need extensive programming knowledge, and integrated BI tools are insufficient.
• Data Integrity: Verifying data integrity at the block level is another issue with distributed NoSQL databases. Checksum work for scale-up databases because the restored data is physically identical to the backup data. The restored data in scale-out databases is semantically comparable to the backup data, but it is not physically identical. In this situation, we'll need to come up with a unique method for identifying semantic equivalence between recovered and backup data, which will allow us to spot data corruption issues that may arise throughout the backup and restoration process.
• Human Errors: Due to the dynamic nature of NoSQL databases, it is quite common for human errors to happen, for example, data deletion or alteration. These errors can lead to data loss, data inconsistency and in some cases, data breaches. Organizations need to have strict protocols in place to minimize the chance of human errors happening and have proper disaster recovery plans in place.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1(b)

ANS:</td>
<td>What are four categories of NoSql Databases?List some database products for each category.

Key-Value and Document Data Models 2M

Column Data Models 1.5M

Graph Data Models 1.5M

Key-Value and Document Data Models
We said earlier on that key-value and document databases were strongly aggregate-oriented. What we meant by this was that we think of these databases as primarily constructed through aggregates. Both of these types of databases consist of lots of aggregates with each aggregate having a key or ID that's used to get at the data. The two models differ in that in a key-value database, the aggregate is opaque 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 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.

Column-Family Stores

One of the early and influential NoSQL databases was Google's BigTable [Chang</td>
<td>5

2

1.5</td>
<td>CO1</td>
<td>L2</td>
</tr>
</table>

etc.]. Its name conjured up a tabular structure which it realized with sparse columns and no schema. As you'll soon see, it doesn't help to think of this structure as a table; rather, it is a two-level map. But, however you think about the structure, 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, but that name has been around for a while to describe a different animal. Pre-NoSQL column stores, such as C-Store [C-Store], were happy with SQL and the relational model.

Graph Databases

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.

1.5

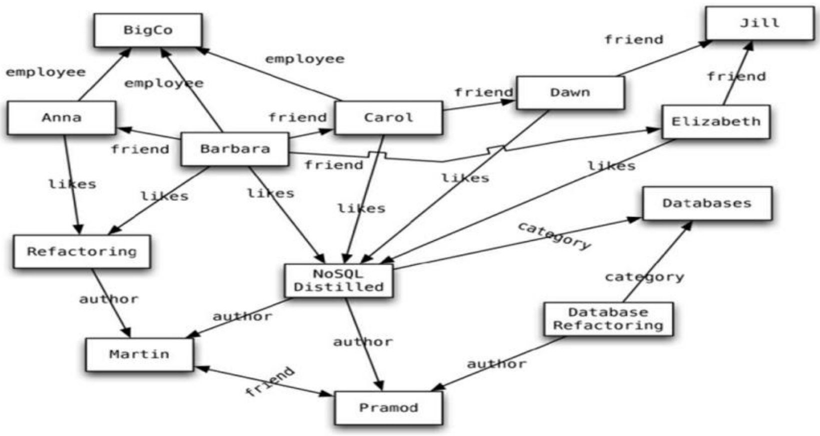


Figure 3.1. An example graph structure

2(a) You are tasked with designing a database system for an e-commerce platform that needs to efficiently handle large volumes of product data. Explain the concept of aggregates in the context of this scenario and how NoSql databases support their management. Additionally, outline two potential consequences or impacts of utilizing aggregates in the database system design for this e-commerce platform. — 5 — CO1 — L2

Ans:

Aggregate 1.5 M

2 Consequences 3.5M

1.5

Aggregate:

The relational model takes the information that we want to store and divides it into tuples (rows). A tuple is a limited data structure: It captures a set of values, so you cannot nest one tuple within another to get nested records, nor can you put a list of values or tuples within another. This simplicity underpins the relational model—it allows us to think of all operations as operating on and returning tuples.

Consequences of Aggregate Orientation:

While the relational mapping captures the various data elements and their relationships reasonably well, it does so without any notion of an aggregate entity.

	In our domain language, we might say that an order consists of order items, a shipping address, and a payment. This can be expressed in the relational model in terms of foreign key relationships—but there is nothing to distinguish relationships that represent aggregations from those that don't. As a result, the database can't use a knowledge of aggregate structure to help it store and distribute the data. Various data modeling techniques have provided ways of marking aggregate or composite structures. The problem, however, is that modelers rarely provide any semantics for what makes an aggregate relationship different from any other; where there are semantics, they vary. 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. Being aggregate-ignorant is not a bad thing. It's often difficult to draw aggregate boundaries well, particularly if the same data is used in many different contexts. An order makes a good aggregate when a customer is making and reviewing orders, and when the retailer is processing orders. 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: Many rows spanning many tables are updated as a single o isolated from each other so they cannot see a partial update.	3.5		
2(b) ANS:	Write a note on I) Impedance Mismatch II) Schemaless databases IMPEDANCE MISMATCH 2.5M SCHEMALESS DB 2.5M **Impedance Mismatch:** Relational databases provide many advantages, but they are by no means perfect. Even from their early days, there have been lots of frustrations with them. 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. The impedance mismatch is a major source of frustration to application developers, and in the 1990s many people believed that it would 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.	5 2.5	CO1	L2

	Schemaless databases: A common theme across all the forms of NoSQL databases is that they are Schemaless. When you want to store data in a relational database, you first have to define a schema—a defined structure for the database which says what tables exist, 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 rejoice in this 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.	2.5		
3(a) ANS:	Which data model does not support data aggregate orientation?Differentiate between key value and document oriented data models. Not supporting Data model 1M Differences 4M The aggregate-Oriented database is the NoSQL database which does not support ACID transactions and they sacrifice one of the ACID properties. Aggregate orientation operations are different compared to relational database operations. **Key-Value and Document Data Models:** We said earlier on that key-value and document databases were strongly aggregate-oriented. What we meant by this was that we think of these databases as primarily constructed through aggregates. Both of these types of databases consist of lots of aggregates with each aggregate having a key or ID that's used to get at the data. The two models differ in that in a key-value database, the aggregate is opaque 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. In practice, 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	5 1 2	CO1	L2

	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. Despite this blurriness, the general distinction still holds. 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.	2		
3(b) ANS:	Assume you're a data engineer for a financial analytics company that needs to optimize its data processing pipeline for generating daily reports on stock market trends. Describe the concept of materialized views within the context of this scenario and elucidate two approaches to implementing them, providing relevant examples for each approach. When we talked about aggregate-oriented data models, we stressed their advantages. 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. Views provide a mechanism to hide from the client whether data is derived data or base 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, since most applications will have to deal with some queries that don't fit well with the aggregate structure. 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. 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.	5 2 3	CO1	L3
4(a)	Why data distribution is important. List the different data distribution models of NOSQL.	5	CO2	L2

ANS: Data distribution 1M
Sharding 2M
Any One Replication 2M

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. 1

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.

There are two paths to data distribution: replication and sharding.

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). 2

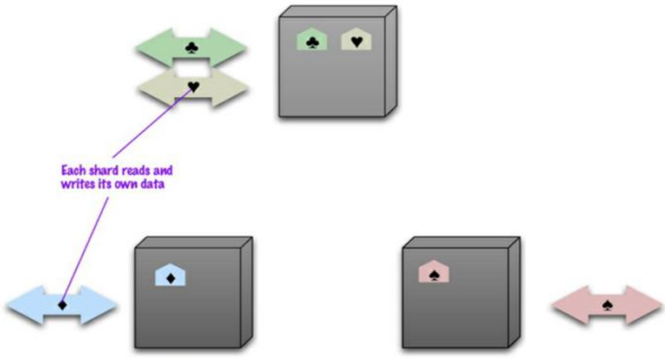


Figure 4.1. Sharding puts different data on separate nodes, each of which does its own reads and writes.

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.

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. 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 you'd like to use.

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).

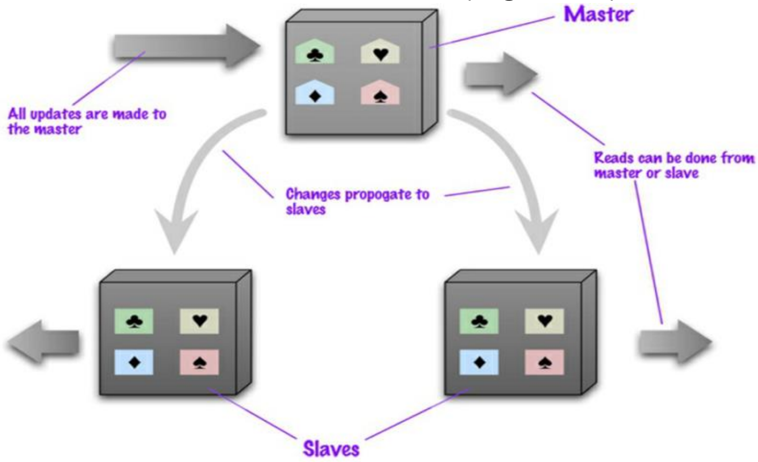


Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

2

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 the doesn't prevent access to the data store. With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance.

1

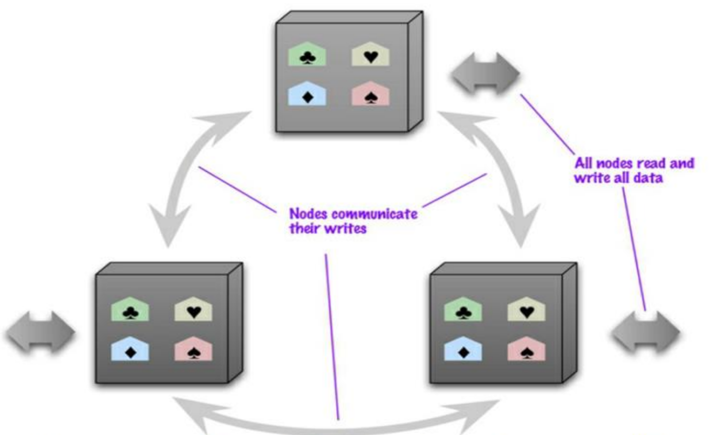


Figure 4.3. Peer-to-peer replication has all nodes applying reads and writes to all the data.

4(b)	As the leader of the backend development team for a high-traffic e-commerce website, you need to ensure data availability and scalability during peak hours. Consider the above scenario and Explain how master-slave replication can address these challenges, detailing its implementation, benefits, and a concrete example of its application in your e-commerce platform.	5	CO2	L3

ANS: Diagram 1.5M
Explanation 3.5M

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 (see Figure 4.2).

• 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.

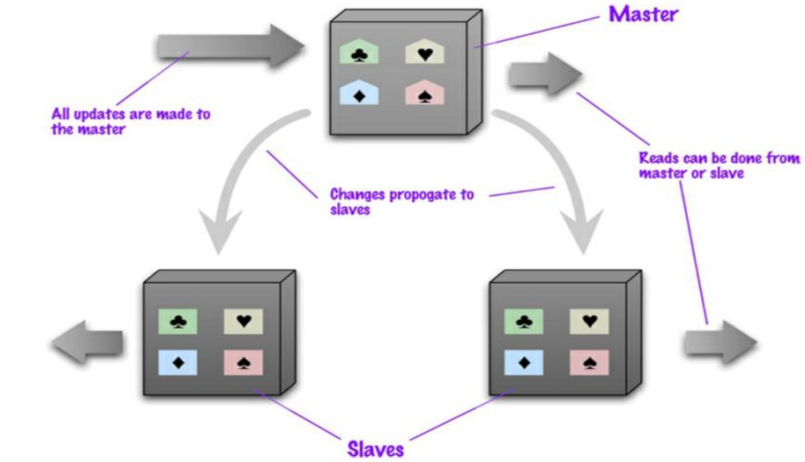


Figure 4.2. Data is replicated from master to slaves. The master services all writes; reads may come from either master or slaves.

● 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.

● A second advantage of master-slave replication is read resilience: Should the master fail, the slaves can still handle read requests.

•The failure of the master does eliminate the ability to handle writes until either the master is restored or a new master is appointed.

-> However, having slaves as replicates of the master does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.

->The ability to appoint a slave to replace a failed master means that master-slave replication is useful even if you don't need to scale out.

● All read and write traffic can go to the master while the slave acts as a hot backup.

	->Masters can be appointed manually or automatically. Manual appointing typically means that when you configure your cluster, you configure one node as the master. ->With automatic appointment, you create a cluster of nodes and they elect one of themselves to be the master. •Apart from simpler configuration, automatic appointment means that the cluster can automatically appoint a new master when a master fails, reducing downtime. In order to get read resilience, you need to ensure that the read and write paths into your application are different, so that you can handle a failure in the write path and still read. ->Replication comes with some alluring benefits, but it also comes with the problem of inconsistency. •You have the danger that different clients, reading different slaves, will see different values because the changes haven't all propagated to the slaves. •In the worst case, that can mean that a client cannot read a write it just made.			

5(a)	What is the CAP theorem? What are three properties which cannot be simultaneously guaranteed? How it is applicable to NoSql Systems.	5	CO2	L2

ANS: 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. (You may also hear this referred to as Brewer's Conjecture.) 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.

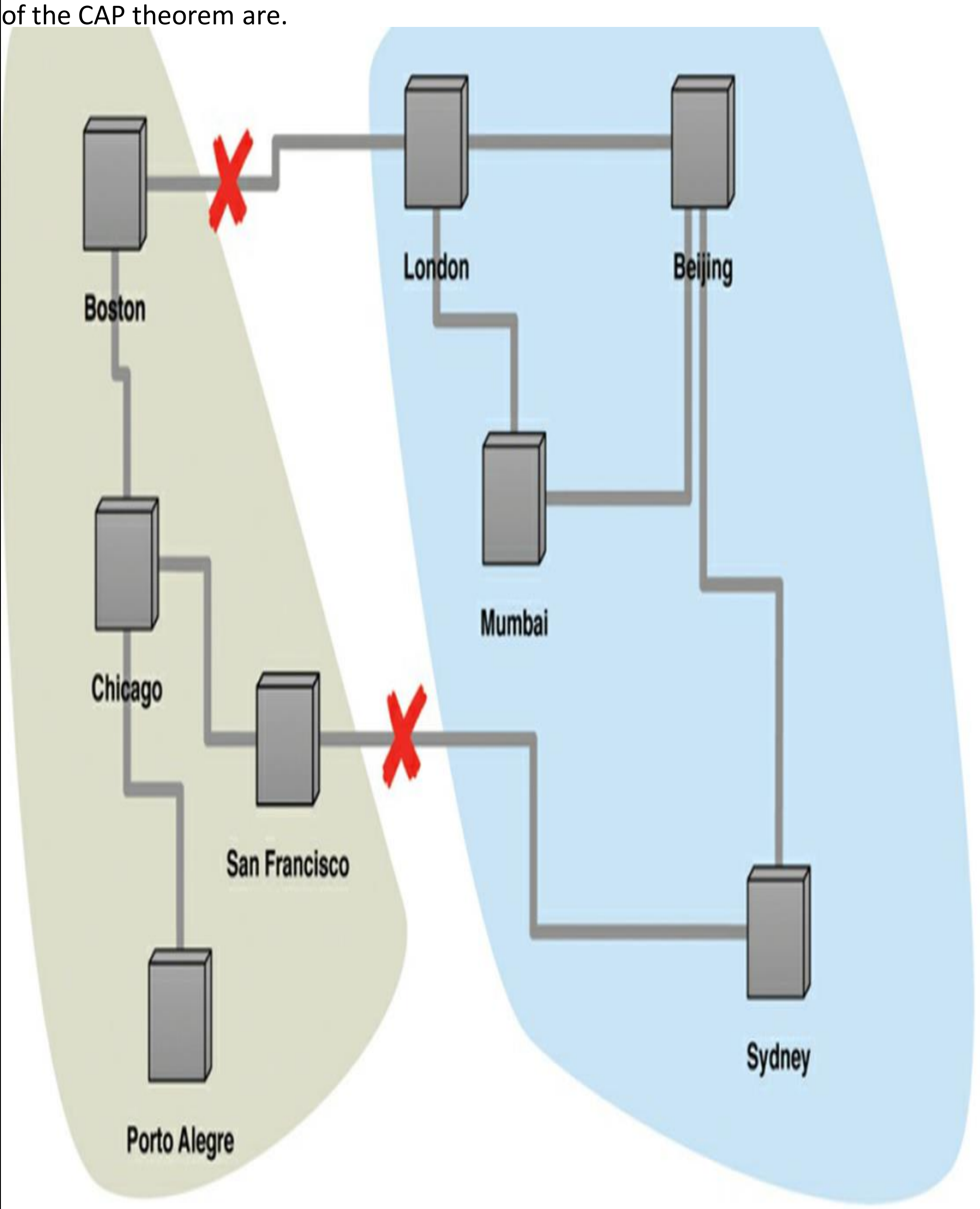


Figure 5.3. With two breaks in the communication lines, the network partitions into two groups.

5(b)	Write a short note on i) Peer to Peer Consistency ii) Replication	5	CO2	L2
ANS:	**Peer to Peer Consistency:** 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. The prospect here looks mighty fine. With a peer-to-peer replication cluster, you can ride over node failures without losing access to data. Furthermore, you can easily add nodes to improve your performance. There's much to like here but there are complications.The biggest complication is, again, consistency. When you can write to two different places, you run the risk that two people will attempt to update the same record at the same time a write-write conflict. Inconsistencies on read lead to problems but at least they are relatively transient. Inconsistent writes are forever. We'll talk more about how to deal with write inconsistencies later on, but for the moment we'll note a couple of broad options. At one end, we can ensure that whenever we write data, the Replicas coordinate to ensure we avoid a conflict. This can give us just as strong a guarantee as a master, albeit at the cost of network traffic to coordinate the writes. We don't need all the replicas to agree on the write, just a majority, so we can still survive losing a minority of the replica nodes. At the other extreme, we can decide to cope with an inconsistent write. There are contexts when we can come up with policy to merge inconsistent writes. In this case we can get the full performance benefit of writing to any replica. These points are at the ends of a spectrum where we trade off consistency for availability. Diagram: 1 M	4		
6(a)	Highlight the significance of consistency in database systems and enumerate the various types of consistency that are commonly recognized within this context.	5	CO2	L2
ANS:	**Update Consistency:** 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. Implausibly, they both have update access, so they both go in at the same time to update the number. To make the example interesting, we'll assume they update it slightly differently, because each uses a slightly different format. This issue is called a write-write conflict: two people updating the same data item at the same time. 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.	1.5		

	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. **Relaxing Consistency:** Consistency is a Good Thing but, sadly, sometimes we have to sacrifice it. 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 trade off consistency for something else. While some architects see this as a disaster, we see it as part of the inevitable trade-offs 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.	2 1.5		
6(b) ANS:	Explain the read-write conflict in logical consistency with the proper example. **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 2.1 Pramod has done a read in the middle of Martin's write. 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 claim is mostly wrong because it glosses over lots of important details. 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, aggregate-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. So in the example, you could avoid	5	CO2	L2

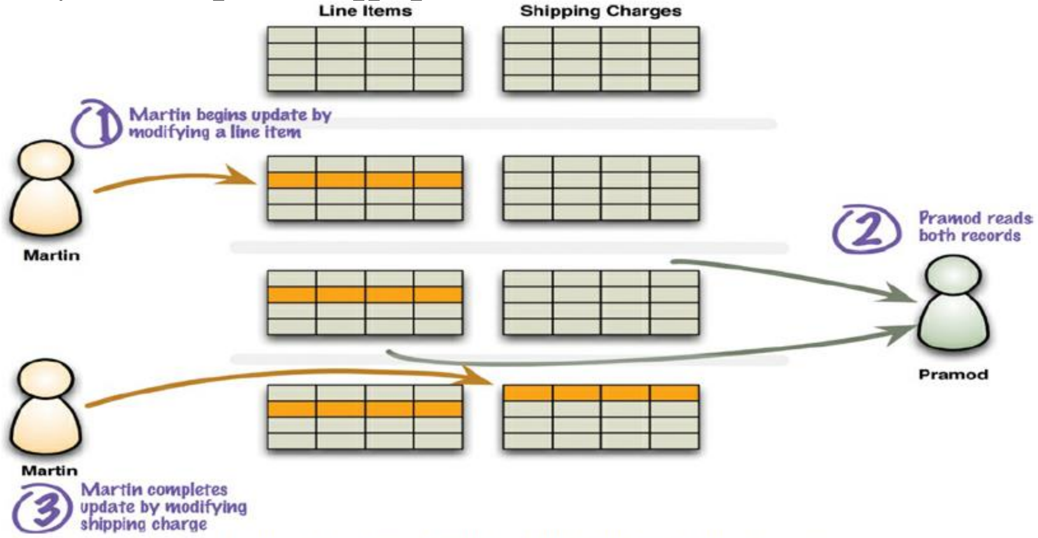
<table>
<tr><td></td><td>running into that inconsistency if the order, the delivery charge, and the line items are all part of a single order aggregate.


Figure 5.1. A read-write conflict in logical consistency</td><td></td><td></td><td></td></tr>
</table>

CI **CCI** **HOD**

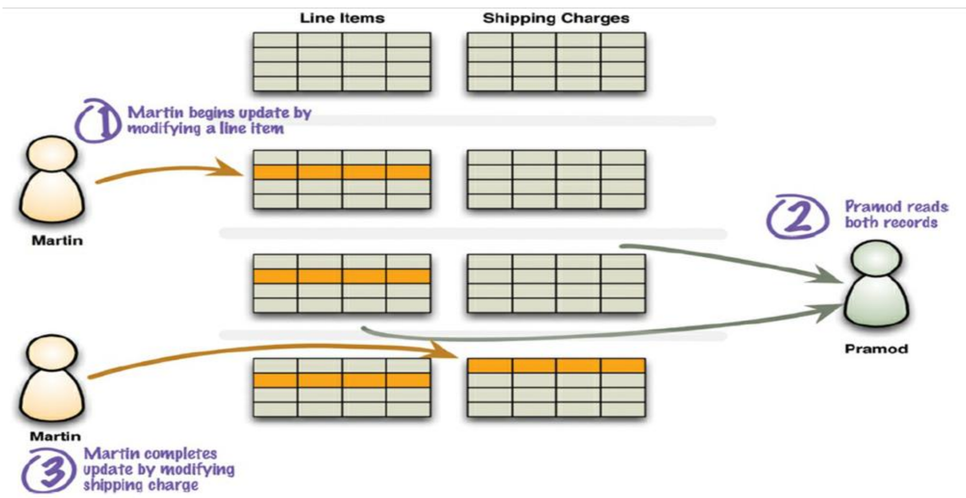


Figure 5.1. A read-write conflict in logical consistency

PO Mapping

Course Outcomes			Modules covered	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	Define, compare and use the four types of NoSQL Databases	**L2**	**1**	2	2	3	2	3	-	-	-	-	-	-	-	3	-	-	-
CO2	Demonstrate an understanding of the detailed architecture, define objects, load data, query data and performance tune Column-oriented NoSql databases.	**L2**	**2,3,4,5**	2	2	3	3	3	-	-	-	-	-	-	-	3	-	-	-
CO3	Explain the detailed architecture, define objects, load data, query data and performance tune Document-oriented NoSql databases.	**L2**	**2,3,4,5**	2	3	3	3	3	-	-	-	-	-	-	-	3	-	-	-

COGNITIVE LEVEL	REVISED BLOOMS TAXONOMY KEYWORDS
L1	List, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
L2	summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend
L3	Apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover.
L4	Analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer.
L5	Assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize.

PROGRAM OUTCOMES (PO), PROGRAM SPECIFIC OUTCOMES (PSO)				CORRELATION LEVELS	
PO1	Engineering knowledge	PO7	Environment and sustainability	0	No Correlation
PO2	Problem analysis	PO8	Ethics	1	Slight/Low
PO3	Design/development of solutions	PO9	Individual and team work	2	Moderate/ Medium
PO4	Conduct investigations of complex problems	PO10	Communication	3	Substantial/ High
PO5	Modern tool usage	PO11	Project management and finance		
PO6	The Engineer and society	PO12	Life-long learning		
PSO1	Develop applications using different stacks of web and programming technologies				
PSO2	Design and develop secure, parallel, distributed, networked, and digital systems				
PSO3	Apply software engineering methods to design, develop, test and manage software systems.				
PSO4	Develop intelligent applications for business and industry				