USN

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

Internal Assessment Test 2 – NOV 2024

Sub:	**NOSQL DATABASE**		Sub Code:	**21CS745**	Branch:	**ISE**
Date:	**19/11/2024**	Duration:	**90 min**	Max Marks: **50**	Sem/Sec: **VII/ A, B & C**	**OBE**

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1.	With a neat diagram and example, Explain the importance of partitioning and combining in the Map-reduce process.	[10]	2	L2

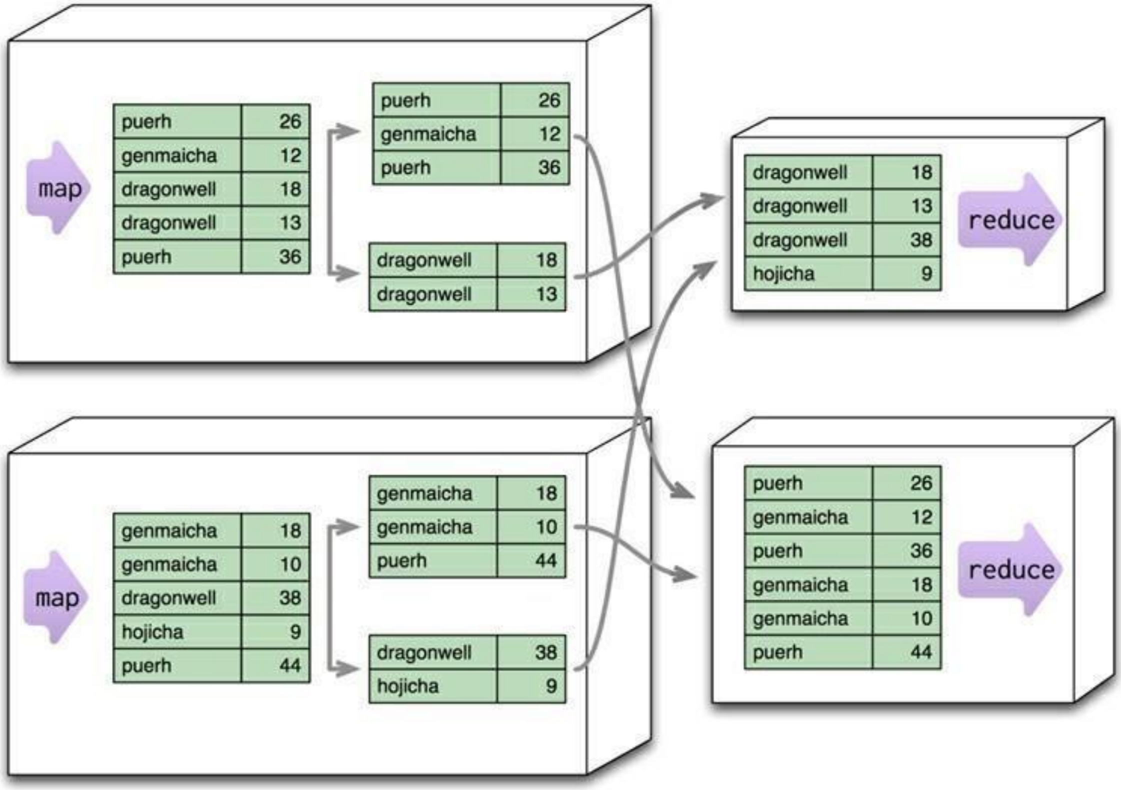
Scheme: Definition + explanation +example + Diagram – 2+3+2+3 Marks

Solution:

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 1.3)

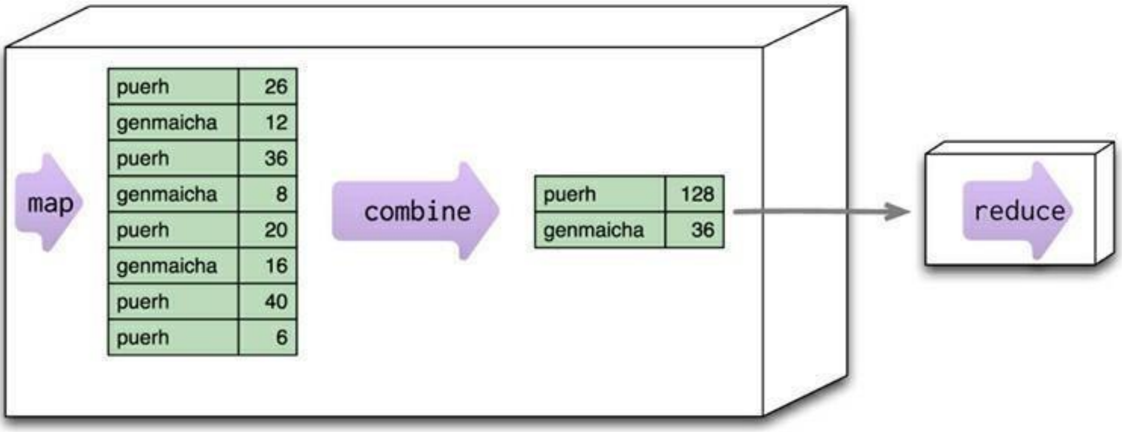
The first thing we can do is increase parallelism by partitioning the output of the mappers. Each reduce function operates on the results of a single key. This is a limitation it means you can't do anything in the reduce that operates across keys but it's also a benefit in that it allows you to run multiple reducers in parallel. 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. 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. Multiple reducers can then operate on the partitions in parallel, with the final results merged together.



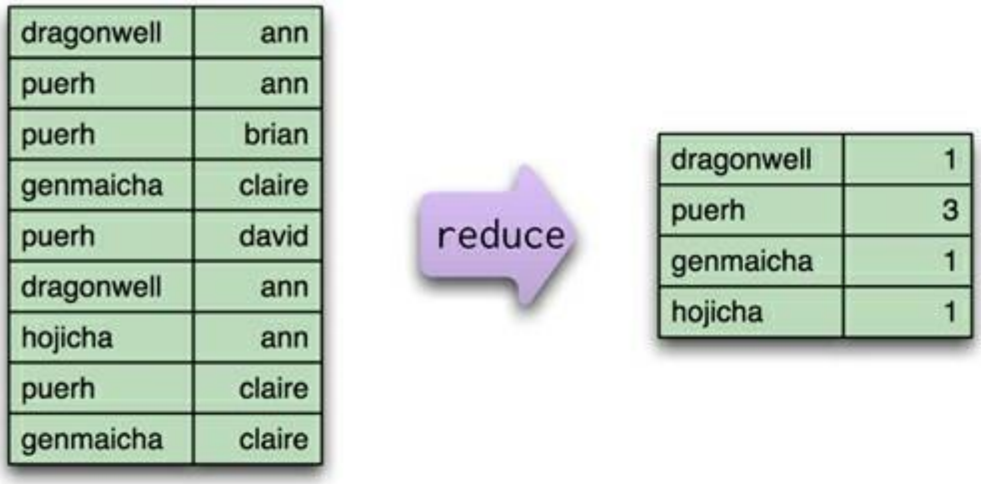
The next problem we can deal with is the amount of data being moved from node to node between the map and reduce stages. Much of this data is repetitive, consisting of multiple key-value pairs for the same key. A combiner function cuts this data down by combining all the data for the same key into a single value (see Figure 1.4). 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. 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.

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 1.5). 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.



puerh	26
genmaicha	12
puerh	36
genmaicha	8
puerh	20
genmaicha	16
puerh	40
puerh	6

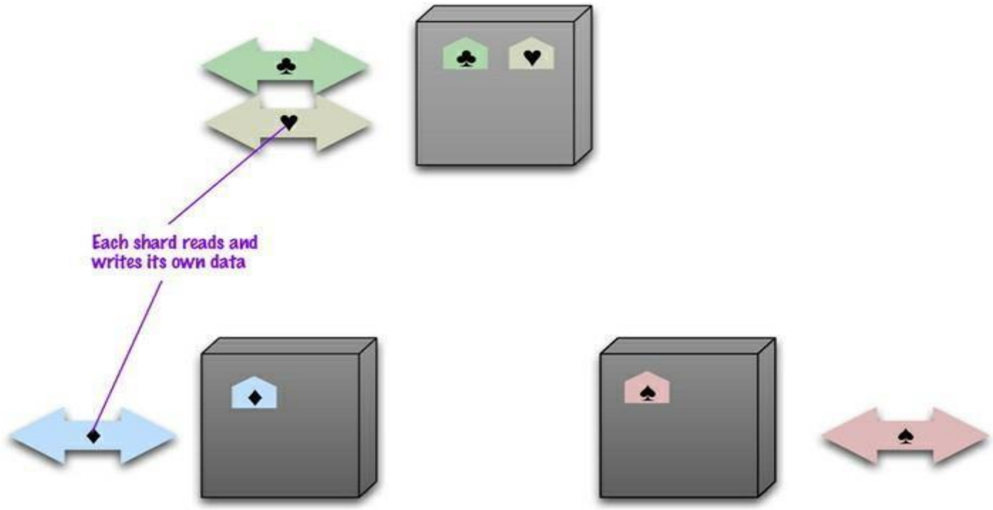
puerh	128
genmaicha	36

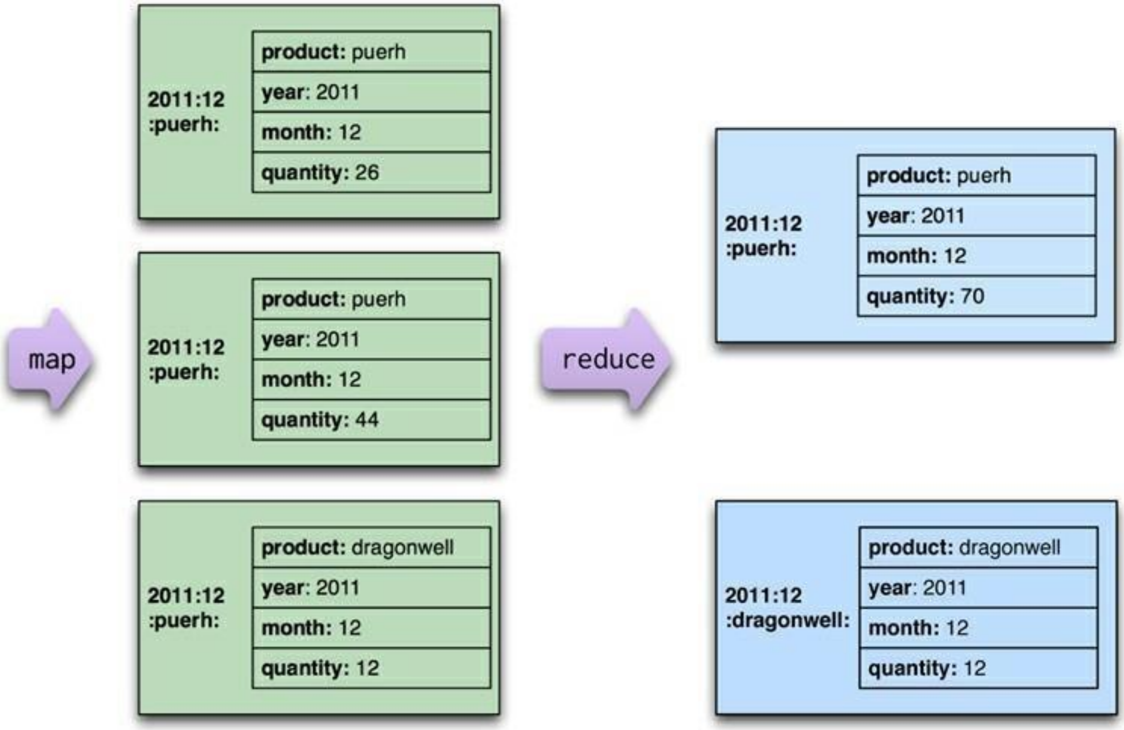


dragonwell	ann
puerh	ann
puerh	brian
genmaicha	claire
puerh	david
dragonwell	ann
hojicha	ann
puerh	claire
genmaicha	claire

dragonwell	1
puerh	3
genmaicha	1
hojicha	1

2.	a. Explain the role of quorum in maintaining consistency **Scheme: Definition + explanation – 2+3 Marks** **Solution :** In NoSQL databases, quorum refers to the minimum number of nodes that must respond to a read or write request to consider it successful. It’s used to ensure data consistency and fault tolerance in distributed systems. For example, if a NoSQL database replicates data across multiple nodes, a quorum-based approach might require that a majority of those nodes respond to any read or write operation. This helps the system avoid inconsistencies that could arise from network partitions or node failures. • Write-write conflicts occur when two clients try to write the same data at the same time. Read-write conflicts occur when one client reads inconsistent data in the middle of anotherclient’s write. • Pessimistic approaches lock data records to prevent conflicts. Optimistic approaches detectconflicts and fix them. • Distributed systems see read-write conflicts due to some nodes having received updates whileother nodes have not. Eventual consistency means that at some point the system will become consistent once all the writes have propagated to all the nodes. • Clients usually want read-your-writes consistency, which means a client can write and thenimmediately read the new value. This can be difficult if the read and the write happen ondifferent nodes. • To get good consistency, you need to involve many nodes in data operations, but this increaseslatency. So you often have to trade off consistency versus latency. • The CAP theorem states that if you get a network partition, you have to trade off availability ofdata versus consistency. • Durability can also be traded off against latency, particularly if you want to survive failureswith replicated data. • You do not need to contact all replicants to preserve strong consistency with replication; youjust need a large enough quorum.	[5]	2	L2

	b. With a neat diagram, explain the mechanism of sharding in distribution models. **Scheme : Definition + explanation + Diagram – 1+2+2 Marks Solution :** Sharding is a technique used in NoSQL databases to improve scalability by distributing data across multiple servers, or "shards." Ideally, each user's data requests are handled by a single server, allowing for balanced load distribution. However, achieving this ideal requires careful data organization to ensure frequently accessed data is on the same shard. Aggregate-oriented design helps by grouping related data together, making sharding more effective. Effective sharding considers factors like physical location for latency reduction and balancing data evenly across nodes to avoid overload. While traditional sharding required application logic, many NoSQL databases now offer auto-sharding, simplifying data distribution and rebalancing. Sharding enhances both read and write performance but doesn't inherently improve resilience. Node failure still makes shard-specific data temporarily inaccessible, affecting users who need that data. Thus, sharding should be implemented early if scalability needs are anticipated, as transitioning from a single server to a sharded setup late in production can disrupt database performance due to resource demands. 	[5]		
3.	Consider the example of product sales data in each year. Apply MapReduce process to compare the sales of products for each month in 2011 to the prior year. **Scheme : 5+5** **Solution :** To compare the monthly product sales data for each month in 2011 to the prior year (2010), we can use the MapReduce process to analyze the large dataset effectively. MapReduce is particularly useful for this task as it allows us to break down the dataset into smaller, parallelizable tasks, apply computations, and then aggregate the results. Here's a detailed explanation of how the MapReduce process would work for this scenario:	[10]	3	L3



Step 1: Define the Input Data
Let’s assume we have a dataset with records for each sale, which includes:

Product ID: A unique identifier for each product.
Date: The date of the sale (including the month and year).
Sales Amount: The amount or units of sale.
For simplicity, we’ll assume the data is organized in a table format, where each row contains details of an individual sale.

Step 2: Map Phase
The Map function processes each record in the dataset. Our goal in this phase is to:

Extract the month and year from each sale’s date.
Organize the sales data into key-value pairs where each key is a combination of ProductID and Month, and the value is the Sales Amount.
Each mapper takes one sale record as input and outputs key-value pairs.

Example Input Record:
yaml
ProductID: 101, Date: 2011-02-15, SalesAmount: 150
ProductID: 101, Date: 2010-02-10, SalesAmount: 120
ProductID: 102, Date: 2011-03-25, SalesAmount: 200
ProductID: 102, Date: 2010-03-18, SalesAmount: 180

Mapper Output:
yaml
(101, 2011-02): 150
(101, 2010-02): 120
(102, 2011-03): 200
(102, 2010-03): 180

The key in each output pair is the product ID and month (e.g., (101, 2011-02)), while the value is the sales amount.

Step 3: Shuffle and Sort Phase
In this phase, all key-value pairs are grouped by key. This step consolidates all records with the same product and month across all mappers, organizing them for the next phase. After shuffling and sorting, our data might look like this:

(101, 2011-02): [150]
(101, 2010-02): [120]

(102, 2011-03): [200]
(102, 2010-03): [180]

Step 4: Reduce Phase

The Reduce function aggregates the sales amounts for each product and month across all records and compares the sales between 2011 and 2010. For each product and month in 2011, the Reducer will:

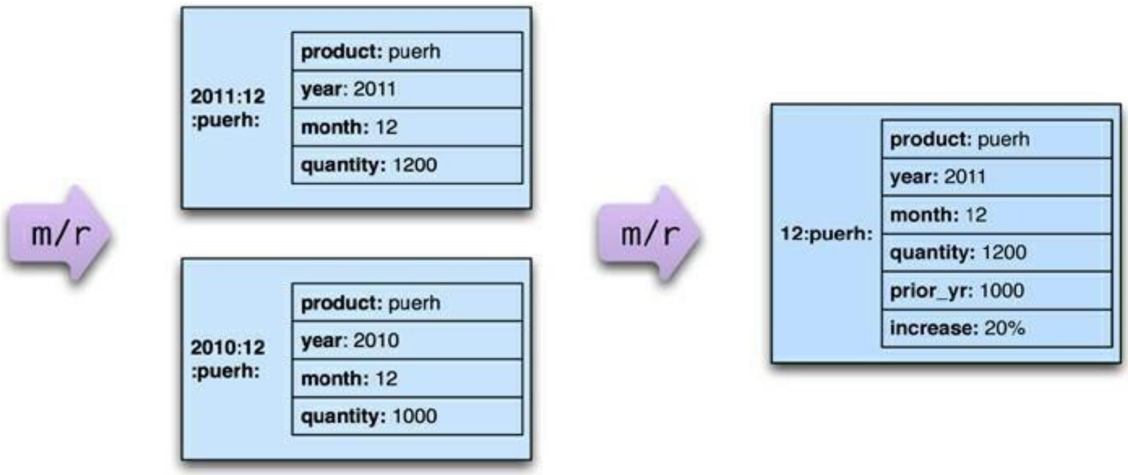
Look up the sales amount from the corresponding month in 2010.
Compute the difference or percentage change between 2011 and 2010 sales amounts.
Output the result in a readable format.
Reducer Calculation Example
For (101, 2011-02), where sales in 2011 are 150 and sales in 2010 are 120, the reducer will:

Calculate the percentage change: ((150 - 120) / 120) * 100 = 25%

Output the result:
Reducer Output:
yaml

```
ProductID: 101, Month: 02, 2010 Sales: 120, 2011 Sales: 150, Change: +25%
ProductID: 102, Month: 03, 2010 Sales: 180, 2011 Sales: 200, Change: +11.1%
```

Step 5: Final Output
The final output contains the product IDs, months, sales in 2010, sales in 2011, and the calculated change in sales. This output provides a clear, month-by-month comparison of sales for each product between 2010 and 2011, showing either the increase or decrease in sales.



4.	Explain the features of key-value stores.	[10]	3	L2

Scheme : Definition + explanation for each – 3+7 Marks

Solution :

Definition

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easilyscaled.

Features

1. Consistency

Consistency is applicable only for operations on a single key, since these operations are either a get, put, or delete on a single key. Optimistic writes can be performed, but are very expensive to implement, because a change in value cannot be determined by the data store.
In distributed key-value store implementations like Riak, the eventually consistent (p. 50) model of consistency is implemented. Since the value may have already been replicated to other nodes, Riak has two ways of resolving update conflicts: either the newest write wins and older writes loose, or both (all) values are returned allowing the client to resolve the conflict.
In Riak, these options can be set up during the bucket creation. Buckets are just a way to namespace keys so that key collisions can be reduced—for example, all customer keys may reside in the customer bucket. When creating a bucket, default values for consistency can be provided, for example that a write is considered good only when the data is consistent across all the nodes where the data is stored.

```
Bucket bucket = connection
.createBucket(bucketName)
.withRetrier(attempts(3))
.allowSiblings(siblingsAllowed)
.nVal(numberOfReplicasOfTheData)
.w(numberOfNodesToRespondToWrite)
.r(numberOfNodesToRespondToRead)
.execute();
```

If we need data in every node to be consistent, we can increase the numberOfNodesToRespondToWrite set by w to be the same as nVal. Of course doing that will decrease the write performance of the cluster. To improve on write or read conflicts, we can change the allowSiblings flag during bucket creation: If it is set to false, we let the last write to win and not create siblings.

2. Transactions

Different products of the key-value store kind have different specifications of transactions. Generally speaking, there are no guarantees on the writes. Many data stores do implement transactions in different ways. Riak uses the concept of quorum ("Quorums," p. 57) implemented by using the Wvalue
—replication factor—during the write API call.
Assume we have a Riak cluster with a replication factor of 5 and we supply the Wvalue of 3. When writing, the write is reported as successful only when it is written and reported as a success on at least three of the nodes. This allows Riak to have write tolerance; in our example, with Nequal to 5 and with a Wvalue of 3, the cluster can tolerate N - W = 2nodes being down for write operations, though we would still have lost some data on those nodes for read.

3. Query Features

Key-value stores typically support querying only by key, limiting options for retrieving data based on specific attributes within the value. If you need to filter data by a value's attribute, the application itself must load the value and evaluate it independently.

This query-by-key limitation has some implications. For example, if the key is unknown, particularly in cases like ad-hoc debugging, it can be challenging to retrieve records. Most key-value databases do not readily provide a list of all keys, and even if they did, querying each key individually for its associated value would be inefficient. Some systems, like Riak Search, address this by enabling searches within the value, similar to Lucene indexing.

Designing keys is crucial in key-value stores. You might generate keys algorithmically, use user identifiers (like user ID or email), or base them on timestamps or other external data. This approach to key design makes key-value stores ideal for storing session information (where the session ID serves as the key), shopping cart details, user profiles, etc. An expiration property (expiry_secs) allows keys to expire after a set time, which is especially useful for temporary data, like session and cart objects.

To store data in Riak, for example, you specify a key-value pair within a bucket. Using the store API, the value is saved for a particular key. Similarly, data can be retrieved with the fetch API:

```
Bucket bucket = getBucket(bucketName);
IRiakObject riakObject = bucket.store(key, value).execute();
...
IRiakObject riakObject = bucket.fetch(key).execute();
byte[] bytes = riakObject.getValue();
String value = new String(bytes);
```

Riak also supports an HTTP interface, enabling operations through a web browser or curl command. Here’s how to store a JSON object in Riak’s session bucket with a specified key (a7e618d9db25):

```
curl -v -X POST -d '{
  "lastVisit": 1324669989288,
  "user": {
    "customerId": "91cfdf5bcb7c",
    "name": "buyer",
    "countryCode": "US",
    "tzOffset": 0
  }
}' -H "Content-Type: application/json"
http://localhost:8098/buckets/session/keys/a7e618d9db25
```

To fetch this data by its key:

```
curl -i http://localhost:8098/buckets/session/keys/a7e618d9db25
```

4. Structure of Data

Key-value databases don’t care what is stored in the value part of the key-value pair. The value can be a blob, text, JSON, XML, and so on. In Riak, we can use the Content-Type in the POST request to specify the data type.

5. Scaling

Many key-value stores scale by using sharding. With sharding, the value of the key determines on which node the key is stored. Let’s assume we are sharding by the first character of the key; if the key is f4b19d79587d, which starts with an f, it will be sent to different node than the key ad9c7a396542. This kind of sharding setup can increase performance as more nodes are added to the cluster.

Sharding also introduces some problems. If the node used to store f goes down, the data stored on that node becomes unavailable, nor can new data be written with keys that start with f.

Data stores such as Riak allow you to control the aspects of the CAP Theorem (“The CAP Theorem,” p. 53): N (number of nodes to store the key-value replicas), R(number of nodes that have to have the data being fetched before the read is considered successful), and W (the number of nodes the write has to be written to before it is considered successful).

Let’s assume we have a 5-node Riak cluster. Setting N to 3 means that all data is replicated to at least three nodes, setting R to 2 means any two nodes must reply to a GET request for it to be considered successful, and setting W to 2 ensures that the PUT request is written to two nodes before the write is considered successful.

These settings allow us to fine-tune node failures for read or write operations. Based on our need, we can change these values for better read availability or write availability. Generally speaking choose a W value to match your consistency needs; these values can be set as defaults during bucket creation.

5.	a. Identify the situations where Key-value stores are ideal and not a best solution. **Scheme : 2.5+2.5** **Ideal Solution** Storing Session Information Generally, every web session is unique and is assigned a unique sessionid value. Applications that store the sessionidon disk or in an RDBMS will greatly benefit from moving to a key-value store, since everything about the session can be stored by a single PUTrequest or retrieved using GET. This single-request operation makes it very fast, as everything about the session is stored in a single object. Solutions such as Memcached are used by many web applications, and Riak can be used when availability is important. User Profiles, Preferences Almost every user has a unique userId, username, or some other attribute, as well as preferences such as language, color, timezone, which products the user has access to, and so on. This can all be put into an object, so getting preferences of a user takes a single GET operation. Similarly, product profiles can be stored. Shopping Cart Data E-commerce websites have shopping carts tied to the user. As we want the shopping carts to be available all the time, across browsers, machines, and sessions, all the shopping information can be put into the value where the key is the userid. A Riak cluster would be best suited for these kinds of applications. **When Not to Use** There are problem spaces where key-value stores are not the best solution. Relationships among Data If you need to have relationships between different sets of data, or correlate the data between different sets of keys, key-value stores are not the best solution to use, even though some key-value stores provide link-walking features. Multioperation Transactions If you’re saving multiple keys and there is a failure to save any one of them, and you want to revert or roll back the rest of the operations, key-value stores are not the best solution to be used. Query by Data If you need to search the keys based on something found in the value part of the key-value pairs, then key-value stores are not going to perform well for you. There is no way to inspect the value on the database side, with the exception of some products like Riak Search or indexing engines like Lucene [Lucene] or Solr [Solr]. Operations by Sets Since operations are limited to one key at a time, there is no way to operate upon multiple keys at the same time. If you need to operate upon multiple keys, you have to handle this from the client side.	[5]	2	L2
5.	b. Explain how data can be read & posted from and to the bucket using queries in Riak. **Scheme : 2.5+2.5** In Riak, a distributed NoSQL key-value database, data can be read from and posted to a bucket (a logical grouping of keys) using queries. Riak provides an HTTP-based RESTful interface, making it accessible through HTTP requests, such as with curl, as well as programmatically through various client libraries. Posting (Writing) Data to a Bucket To store data in Riak, you typically specify a bucket, a unique key within that bucket, and the value you want to store. The process involves an HTTP PUT or POST request where the key-value pair is saved in the specified bucket. For example, let’s say we want to store a user’s session data in a bucket called session with the key user1234. 1. Prepare the JSON Data: Organize the data in JSON format (or any format supported by Riak).	[5]		

```
{
  "lastVisit": 1324669989288,
```

```
  "user": {
    "customerId": "91cfdf5bcb7c",
    "name": "buyer",
    "countryCode": "US",
    "tzOffset": 0
  }
}
```

2. Execute the curl Command to POST Data: Use curl to make a POST request to Riak's HTTP interface:

```
curl -X POST -d '{
  "lastVisit": 1324669989288,
  "user": {
    "customerId": "91cfdf5bcb7c",
    "name": "buyer",
    "countryCode": "US",
    "tzOffset": 0
  }
}' -H "Content-Type: application/json" http://localhost:8098/buckets/session/keys/user1234
```

>>POST sends the JSON object to the specified bucket (session) with the key (user1234).
>>-H "Content-Type: application/json" specifies the data format.

Reading (Fetching) Data from a Bucket
To read or fetch data in Riak, you use an HTTP GET request to retrieve the value associated with a specific key from a bucket.

For instance, to retrieve the session data stored with key user1234 in the session bucket:

1. Execute the curl Command to GET Data:

```
curl -i http://localhost:8098/buckets/session/keys/user1234
```

>>GET requests the data at the URL specified.
>>The response will include the data stored for user1234 in JSON format, along with HTTP headers.

2. Example Response: Riak will return the data associated with the key:

```
{
  "lastVisit": 1324669989288,
  "user": {
    "customerId": "91cfdf5bcb7c",
    "name": "buyer",
    "countryCode": "US",
    "tzOffset": 0
  }
}
```

Using Riak Client Libraries
Alternatively, Riak offers client libraries in languages like Java, Python, and Ruby, allowing data posting and reading programmatically without directly writing HTTP requests.

For example, using Java:

```
Bucket bucket = getBucket("session");
bucket.store("user1234", jsonData).execute();  // To post data
IRiakObject riakObject = bucket.fetch("user1234").execute();  // To fetch data
```

6.	Explain: a. Various approaches of constructing version stamps b. CAP Theorem **Scheme : Definition & Explanation– 2+3+2+3 Marks**	[10]	3	L2

Solution:

1. **Various approaches of constructing version stamps**

Incremental Version Numbering

Each time data is modified, a version number is incremented. This version counter is unique to each item or record, often starting from zero.
Advantages: Easy to track the order of changes, and there is no reliance on external time sources.
Disadvantages: Not well-suited for distributed systems with multiple nodes updating the data simultaneously, as it can lead to version conflicts.

UUIDs (Universally Unique Identifiers)

Unique identifiers, like UUIDs, can be generated each time an item is updated, representing a new version. UUIDs use a combination of random elements and time-based factors to ensure uniqueness.
Advantages: Can be generated independently on different nodes without conflict.
Disadvantages: Lacks natural ordering of versions and may complicate conflict resolution.

Hash-Based Versioning

In this approach, the data's content itself is hashed (e.g., using SHA-256) to create a unique version stamp for each modification. Changes to the data will produce a different hash, indicating a new version.
Advantages: Useful for ensuring data integrity, as any alteration to the data results in a new hash.
Disadvantages: Hashes do not inherently indicate version order, making it harder to track the sequence of changes.

Vector Clocks

Vector clocks are a sophisticated technique used in distributed systems where each node keeps a vector of counters (one per node) for a given data item. When a node modifies the item, it increments its own counter in the vector clock.
Advantages: Helps track causality and manage concurrent updates across distributed nodes.
Disadvantages: Vector clocks grow in size with the number of nodes, which may lead to overhead in highly distributed environments.

Lamport Timestamps

Lamport timestamps are logical clocks that use counters instead of actual time to represent the order of events. Each process in a system maintains a counter, incrementing it with each event. This timestamp is attached to messages and helps order events in a distributed setup.
Advantages: Simple to implement and suitable for distributed systems where only event order is important.
Disadvantages: Only provides partial ordering of events and may not handle concurrent updates well.

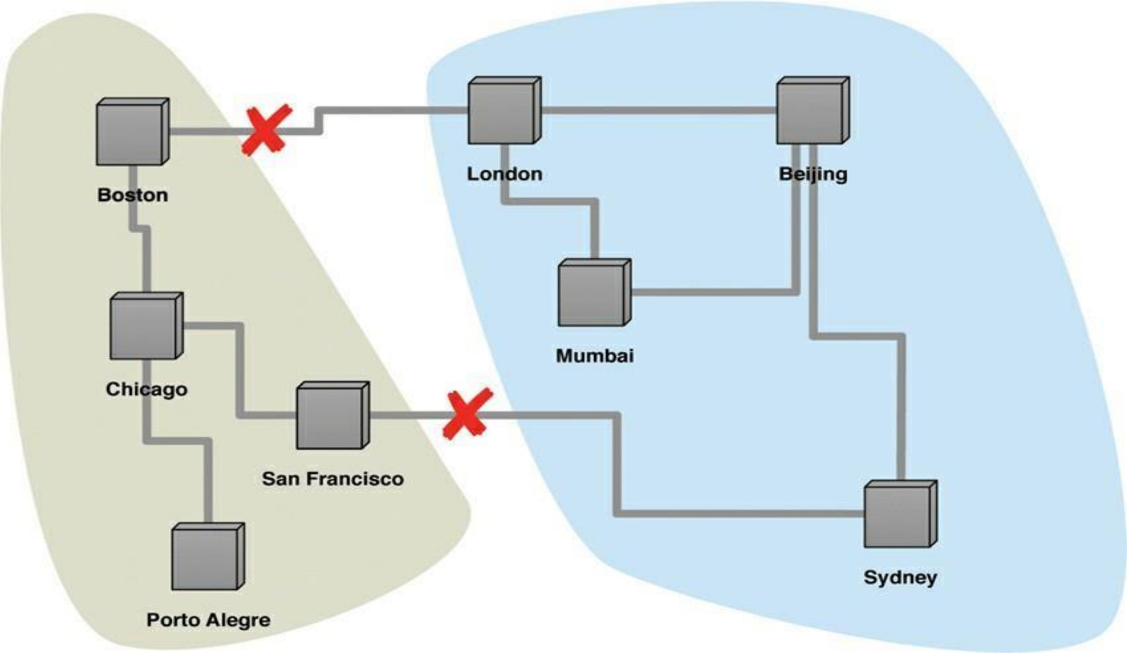
Commit Hashes in Version Control Systems (e.g., Git)

Systems like Git use commit hashes (a combination of the content and parent history) to track versions. Each commit is uniquely identified, allowing for branching, merging, and comparison.
Advantages: Efficient for systems requiring detailed version history with branching.
Disadvantages: Not suitable for real-time versioning due to its complexity and reliance on content history.

2. CAP Theorem

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 is pretty much as we've defined it so far. **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. That's subtly different from the usual meaning, which we'll explore later. **Partition tolerance** means that the cluster can survive communication breakages in the cluster that separate the clusterinto multiple partitions unable to communicate with each other.



A single-server system is the obvious example of a CA system a system that has Consistencyand Availability but not Partition tolerance. A single machine can't partition, so it does not have to worry about partition tolerance. There's only one node so if it's up, it's available. Being up and keeping consistency is reasonable. This is the world that most relational database systems live in.

Faculty Signature CCI Signature HOD Signature