Scheme of Evaluation
Internal Assessment Test 2 – April 2024

Sub:	NoSQL Database							Code:	18CS823
Date:	13/4/2024	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	ISE

Note: Answer Any five full questions.

Question #		Description	Marks Distribution		Max Marks
1	a)	Explain the following- a. Relaxing consistency b. CAP theorem c. Relaxing durability Relaxing consistency CAP Theorem Relaxing Durability	 2M 2M 1M	5M	10M
1	b)	List and explain the approaches for constructing version stamps for multiple nodes data models. Any three approaches	5M	5M	
2	a)	Define Quorum. Explain Read and Write Quorums with examples Quorum Read Quorum Write quorum	2M 4M 4M	10M	10M
3	a)	Explain partitioning and combining stages with example. Diagram – explanation	2*2M=6M 4M	10M	10M

4	a)	Explain basic map reduce with neat diagram. Diagram Explanation	 3M 3M	6M	10M
	b)	Explain how data can be read and posted from and to the bucket using queries in Riak. Example Explanation	 2M 2M	4M	
5	a)	Explain the features of Key Value stores. Consistency Transactions Query Features Structure of dada Scaling	 2M*5	10M	10M
6	a)	Create a map reduce model for calculating average ordered quantity of each product. Compute and explain the calculations composed in map reduce with an example and neat diagram. Diagram Explanation	 5M 5M	10M	10M

Scheme Of Evaluation
Internal Assessment Test 2–April2024

Sub:	NoSQL Database							**Code:**	18CS823
Date**:**	13/4/2024	Duration:	90mins	Max Marks:	50	**Sem:**	VIII	**Branch:**	ISE

Note: Answer Any full five questions

Q 1. a)

Q-1 Explain the following-

a. Relaxing consistency
b. CAP theorem
c. Relaxing durability
d. Quorums

Ans 1-

Relaxing Consistency

It is always possible to design a system to avoid inconsistencies, but often impossible to do so without making unbearable sacrifices in other characteristics of the system. As a result, we often have to tradeoff consistency for something else. While some architects see this as a disaster, we see it as part of the inevitable tradeoffs involved in system design. Furthermore, different domains have different tolerances for inconsistency, and we need to take this tolerance into account as we make our decisions.

Trading off consistency is a familiar concept even in single-server relational database systems. Here, our principal tool to enforce consistency is the transaction, and transactions can provide strong consistency guarantees. However, transaction systems usually come with the ability to relax isolation levels, allowing queries to read data that hasn't been committed yet, and in practice we see most applications relax consistency down from the highest isolation level (serialized) in order to get effective performance. We most commonly see people using the read-committed transaction level, which eliminates some read-write conflicts but allows others.

Many systems forgo transactions entirely because the performance impact of transactions is too high. We've seen this in a couple different ways. On a small scale, we saw the popularity of MySQLduring the days when it didn't support transactions. Many websites liked the high speed of MySQL and were prepared to live without transactions. At the other end of the scale, some very large websites, such as eBay [Pritchett], have had to forgo transactions in order to perform acceptably— this is particularly true when you need to introduce sharding. Even without these

constraints, many application builders need to interact with remote systems that can't be properly included within a transaction boundary, so updating outside of transactions is a quite common occurrence for enterprise applications.

The 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 howyou define these three properties, and differing opinions have led to several debates on what the real consequences of the CAP theorem are.

Availability has a particular meaning in thecontext 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** meansthat the cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other (situation known as a **split brain**)

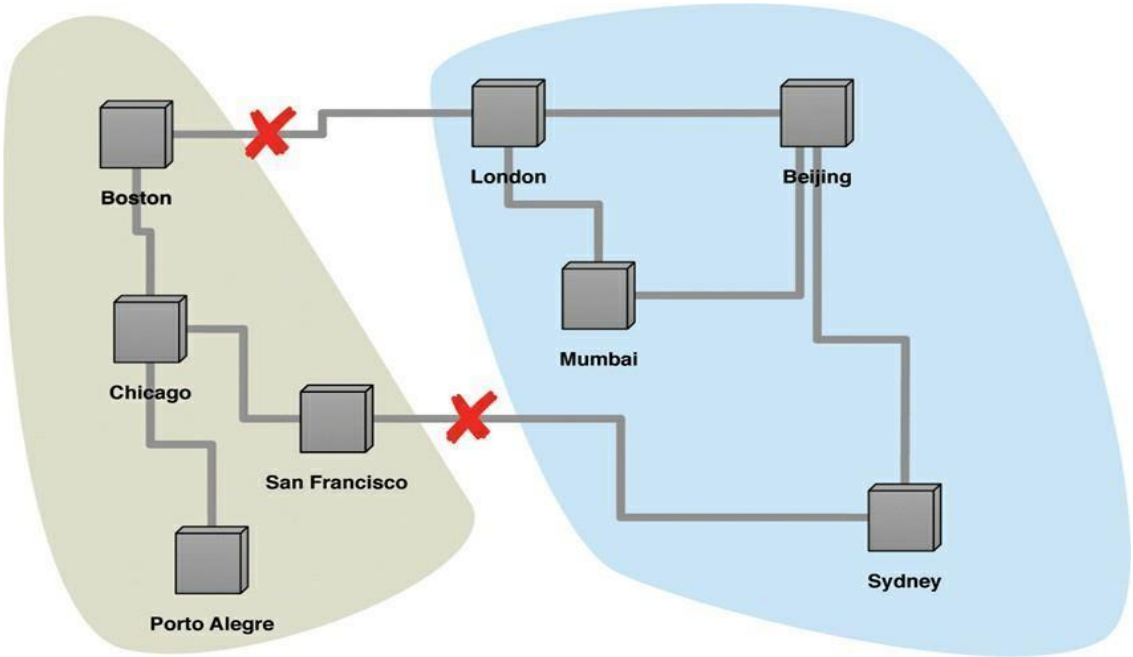


Figure 1.3. With two breaks in the communication lines, the network partitions into two groups.

A single-server system is the obvious example of a CA system a system that has Consistency andAvailability but not Partition tolerance. A single machine can't partition, so it does not have to worryabout 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.

It is theoretically possible to have a CA cluster. However, this would mean that if a partition ever

occurs in the cluster, all the nodes in the cluster would go down so that no client can talk to a node. By the usual definition of "available," this would mean a lack of availability, but this is where CAP'sspecial usage of "availability" gets confusing. CAP defines "availability" to mean "every request received by a nonfailing node in the system must result in a response" [Lynch and Gilbert]. So a failed, unresponsive node doesn't infer a lack of CAP availability.

This does imply that you can build a CA cluster, but you have to ensure it will only partition rarely and completely. This can be done, at least within a data center, but it's usually prohibitively expensive. Remember that in order to bring down all the nodes in a cluster on a partition, you also have to detect the partition in a timely manner—which itself is no small feat.

So clusters have to be tolerant of network partitions. And here is the real point of the CAP theorem. Although the CAP theorem is often stated as "you can only get two out of three," in practice what it's saying is that in a system that may suffer partitions, as distributed system do, you have to trade off consistency versus availability. This isn't a binary decision; often, you can trade off a little consistency to get some availability. The resulting system would be neither perfectly consistent nor perfectly available—but would have a combination that is reasonable for your particular needs.

An example should help illustrate this. Martin and Pramod are both trying to book the last hotel room on a system that uses peer-to-peer distribution with two nodes (London for Martin and Mumbai for Pramod). If we want to ensure consistency, then when Martin tries to book his room on the London node, that node must communicate with the Mumbai node before confirming the booking. Essentially, both nodes must agree on the serialization of their requests. This gives us consistency—but should the network link break, then neither system can book any hotel room, sacrificing availability.

One way to improve availability is to designate one node as the master for a particular hotel and ensure all bookings are processed by that master. Should that master be Mumbai, then Mumbai can still process hotel bookings for that hotel and Pramod will get the last room. If we use master- slave replication, London users can see the inconsistent room information but cannot make a booking and thus cause an update inconsistency. However, users expect that it could happen in this situation—so,again, the compromise works for this particular use case.

This improves the situation, but we still can't book a room on the London node for the hotel whosemaster is in Mumbai if the connection goes down. In CAP terminology, this is a failure of availabilityin that Martin can talk to the London node but the London node cannot update the data. To gain more availability, we might allow both systems to keep accepting hotel reservations even if the network link breaks down. The danger here is that Martin and Pramod book the last hotel room. However, depending on how this hotel operates, that may be fine. Often, travel companies tolerate a

certain amount of overbooking in order to cope with no-shows. Conversely, some hotels always keep a few rooms clear even when they are fully booked, in order to be able to swap a guest out of a room with problems or to accommodate a high-status late booking. Some might even cancel the booking with an apology once they detected the conflict—reasoning that the cost of that is less than the cost of losing bookings on network failures.

The classic example of allowing inconsistent writes is the shopping cart, as discussed in Dynamo [Amazon's Dynamo]. In this case you are always allowed to write to your shopping cart, even if network failures mean you end up with multiple shopping carts. The checkout process can merge the two shopping carts by putting the union of the items from the carts into a single cart and returning that.Almost always that's the correct answer—but if not, the user gets the opportunity to look at the cart before completing the order.

The lesson here is that although most software developers treat update consistency as The Way Things Must Be, there are cases where you can deal gracefully with inconsistent answers to requests. These situations are closely tied to the domain and require domain knowledge to know how to resolve. Thus you can't usually look to solve them purely within the development team— you have to talk to domain experts. If you can find a way to handle inconsistent updates, this gives you more options to increase availability and performance. For a shopping cart, it means that shoppers can always shop, and do so quickly. And as Patriotic Americans, we know how vital it is to support Our Retail Destiny.

A similar logic applies to read consistency. If you are trading financial instruments over a computerized exchange, you may not be able to tolerate any data that isn't right up to date. However, if you are posting a news item to a media website, you may be able to tolerate old pages for minutes.

In these cases you need to know how tolerant you are of stale reads, and how long the inconsistency window can be—often in terms of the average length, worst case, and some measure of the distribution for the lengths. Different data items may have different tolerances for staleness, and thus may need different settings in your replication configuration.

Relaxing Durability

So far we've talked about consistency, which is most of what people mean when they talk about the ACID properties of database transactions. The key to Consistency is serializing requests by forming Atomic, Isolated work units. But most people would scoff at relaxing durability—after all, what is thepoint of a data store if it can lose updates?

As it turns out, there are cases where you may want to trade off some durability for higher performance. If a database can run mostly in memory, apply updates to its in-memory representation,and periodically flush changes to disk, then it may be able to provide substantially higher responsiveness to requests. The cost is that, should the server crash, any updates since the last flush will be lost.

One example of where this tradeoff may be worthwhile is storing user-session state. A big website may have many users and keep temporary information about what each user is doing in some kind of session state. There's a lot of activity on this state, creating lots of demand, which affects the responsiveness of the website. The vital point is that losing the session data isn't too much of a tragedy—it will create some annoyance, but maybe less than a slower website would cause. This makes it a good candidate for nondurable writes. Often, you can specify the durability needs on a call-by-call basis, so that more important updates can force a flush to disk.

Another example of relaxing durability is capturing telemetric data from physical devices. It maybe that you'd rather capture data at a faster rate, at the cost of missing the last updates should the server go down.

Another class of durability tradeoffs comes up with replicated data. A failure of **replication durability** occurs when a node processes an update but fails before that update is replicated to the other nodes. A simple case of this may happen if you have a master-slave distribution model wherethe slaves appoint a new master automatically should the existing master fail. If a master does fail, any writes not passed onto the replicas will effectively become lost. Should the master come back online, those updates will conflict with updates that have happened since. We think of this as a durability problem because you think your update has succeeded since the master acknowledged it,but a master node failure caused it to be lost.

If you're sufficiently confident in bringing the master back online rapidly, this is a reason not to auto-failover to a slave. Otherwise, you can improve replication durability by ensuring that the masterwaits for some replicas to acknowledge the update before the master acknowledges it to the client.

Q-1 b List and explain the approaches through which version stamps for multiple nodes data models

Ans -

The basic version stamp works well when you have a single authoritative source for data, such as a single server or master-slave replication. In that case the version stamp is controlled by the master. Any slaves follow the master's stamps. But this system has to be enhanced in a peer-to-peer distribution model because there's no longer a single place to set the version stamps.

If you're asking two nodes for some data, you run into the chance that they may give you different answers. If this happens, your reaction may vary depending on the cause of that difference. It may be that an update has only reached one node but not the other, in which case you can accept the latest (assuming you can tell which one that is). Alternatively, you may have run into an inconsistent update, in which case you need to decide how to deal with that. In this situation, a simple GUID or etag won't suffice, since these don't tell you enough about the relationships.

The simplest form of version stamp is a counter. Each time a node updates the data, it increments the counter and puts the value of the counter into the version stamp. If you have blue and green slave replicas of a single master, and the blue node answers with a version stamp of 4 and the green node with 6, you know that the green's answer is more recent.

In multiple-master cases, we need something fancier. One approach, used by distributed version control systems, is to ensure that all nodes contain a history of version stamps. That way you can see if the blue node's answer is an ancestor of the green's answer. This would either require the clients to hold onto version stamp histories, or the server nodes to keep version stamp histories and include them when asked for data. This also detects an inconsistency, which we would see if we get two version stamps and neither of them has the other in their histories. Although version control systems keep these kinds of histories, they aren't found in NoSQL databases.

A simple but problematic approach is to use timestamps. The main problem here is that it's usually difficult to ensure that all the nodes have a consistent notion of time, particularly if updates can happen rapidly. Should a node's clock get out of sync, it can cause all sorts of trouble. In addition, you can't detect write-write conflicts with timestamps, so it would only work well for the single- master case—and then a counter is usually better.

The most common approach used by peer-to-peer NoSQL systems is a special form of version stamp which we call a vector stamp. In essence, a vector stamp is a set of counters, one for each node. A vector stamp for three nodes (blue, green, black) would look something like [blue: 43, green: 54, black: 12]. Each time a node has an internal update, it updates its own counter, so an update in the green node would change the vector to [blue: 43, green: 55, black: 12].

Whenever two nodes communicate, they synchronize their vector stamps. There are several variations of exactly how this synchronization is done. We're coining the term "vector stamp" as a general term in

this book; you'll also come across vector clocks and version vectors—these are specific forms of vector stamps that differ in how they synchronize.

By using this scheme you can tell if one version stamp is newer than another because the newer stamp will have all its counters greater than or equal to those in the older stamp. So [blue: 1, green: 2, black: 5] is newer than [blue:1, green: 1, black 5] since one of its counters is greater. If both stamps have a counter greater than the other, e.g. [blue: 1, green: 2, black: 5] and [blue: 2, green: 1, black: 5], then you have a write-write conflict.

There may be missing values in the vector, in which case we use treat the missing value as 0. So [blue: 6, black: 2] would be treated as [blue: 6, green: 0, black: 2]. This allows you to easily add new nodes without invalidating the existing vector stamps.

Vector stamps are a valuable tool that spots inconsistencies, but doesn't resolve them. Any conflict resolution will depend on the domain you are working in. This is part of the consistency/latency tradeoff. You either have to live with the fact that network partitions may make your system unavailable, or you have to detect and deal with inconsistencies.

Q-2- Define Quorum. Explain Read and Write Quorums with examples.

Ans 4-

The more nodes are involve in a request, the higher is the chance of avoiding an inconsistency. This naturally leads to the question: How many nodes need to be involved to get strong consistency?

Imagine some data replicated over three nodes. You don't need all nodes to acknowledge a write to ensure strong consistency; all you need is two of them—a majority. If you have conflicting writes, only one can get a majority. This is referred to as a **write quorum** and expressed in a slightly pretentious inequality of $W > N/2$, meaning the number of nodes participating in the write (W) must be more than the half the number of nodes involved in replication (N). The number of replicas is often called the **replication factor**.

Similarly to the write quorum, there is the notion of **read quorum**: How many nodes you need to contact to be sure you have the most up-to-date change. The read quorum is a bit more complicated because it depends on how many nodes need to confirm a write.

Let's consider a replication factor of 3. If all writes need two nodes to confirm ($W = 2$) then we need to contact at least two nodes to be sure we'll get the latest data. If, however, writes are only confirmed by

a single node (W = 1) we need to talk to all three nodes to be sure we have the latest updates. In this case, since we don't have a write quorum, we may have an update conflict, but by contacting enough readers we can be sure to detect it. Thus we can get strongly consistent reads even if we don't have strong consistency on our writes.

This relationship between the number of nodes you need to contact for a read (R), those confirming a write (W), and the replication factor (N) can be captured in an inequality: You can have a strongly consistent read if R + W > N.

These inequalities are written with a peer-to-peer distribution model in mind. If you have a master-slave distribution, you only have to write to the master to avoid write-write conflicts, and similarly only read from the master to avoid read-write conflicts. With this notation, it is common to confuse the number of nodes in the cluster with the replication factor, but these are often different. I may have 100 nodes in my cluster, but only have a replication factor of 3, with most of the distribution occurring due to sharding.

Indeed most authorities suggest that a replication factor of 3 is enough to have good resilience. This allows a single node to fail while still maintaining quora for reads and writes. If you have automatic rebalancing, it won't take too long for the cluster to create a third replica, so the chances of losing a second replica before a replacement comes up are slight.

The number of nodes participating in an operation can vary with the operation. When writing, we might require quorum for some types of updates but not others, depending on how much we value consistency and availability. Similarly, a read that needs speed but can tolerate staleness should contact less nodes.

Often you may need to take both into account. If you need fast, strongly consistent reads, you could require writes to be acknowledged by all the nodes, thus allowing reads to contact only one (N = 3, W= 3, R = 1). That would mean that your writes are slow, since they have to contact all three nodes, and you would not be able to tolerate losing a node. But in some circumstances that may be the tradeoff to make.

Q. 3 Explain partitioning and combining stages with example.

1.2 Partitioning and Combining

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. (This step is also called

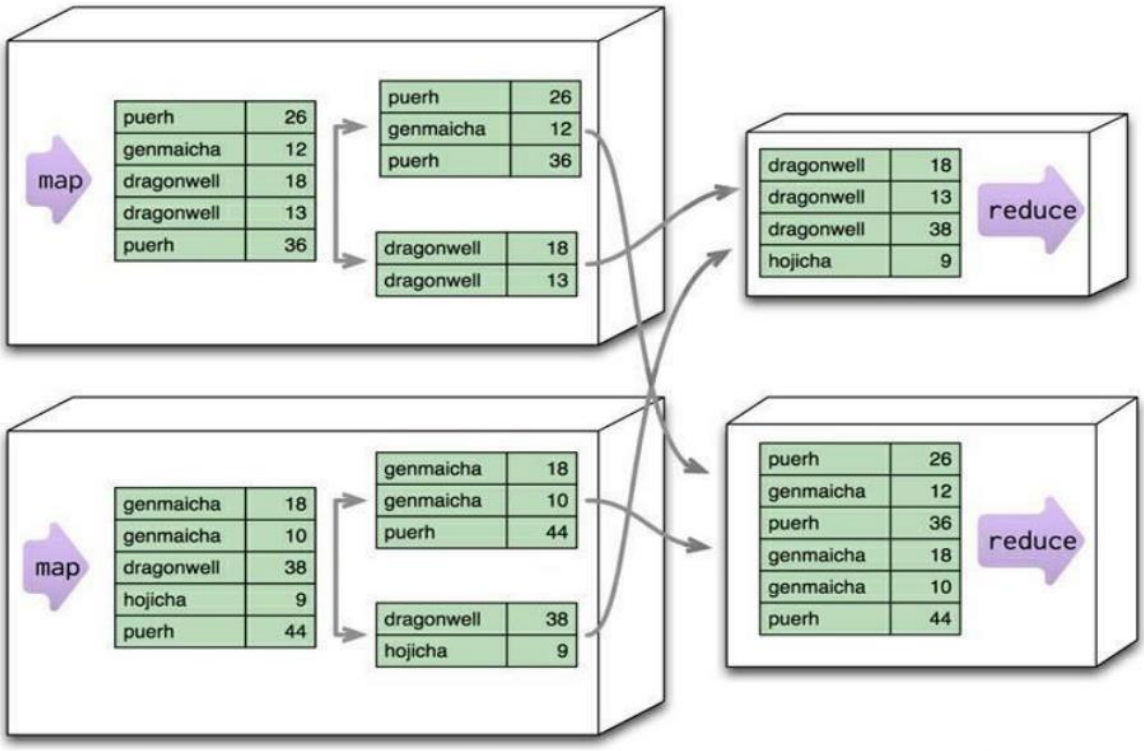


Figure 1.3. Partitioning allows reduce functions to run in parallel on different

keys.

"shuffling," and the partitions are sometimes referred to as "buckets" or "regions.")

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

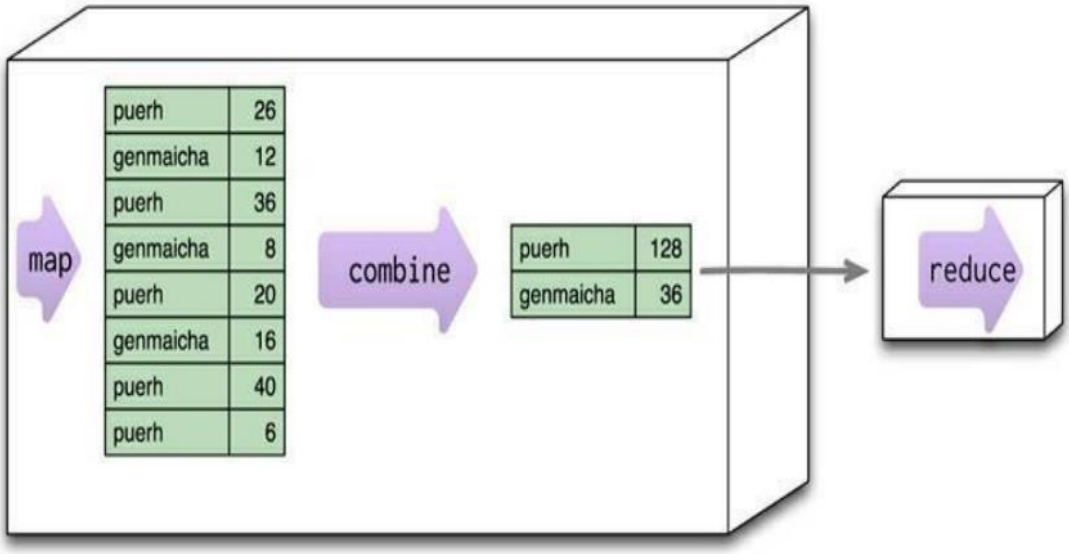
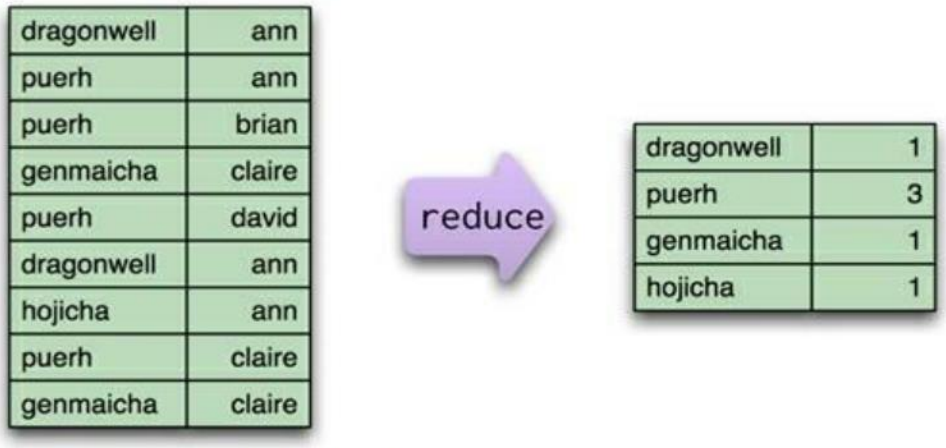


Figure 1.4. Combining reduces data before sending it across the network.

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

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.

Figure 1.5. This reduce function, which counts how many unique customers order a particular tea, is not combinable.

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

Q. 4 b) Explain how data can be read and posted from and to the bucket using queries in Riak.

All key-value stores can query by the key—and that's about it. If you have requirements to query by using some attribute of the value column, it's not possible to use the database: Your application needs to read the value to figure out if the attribute meets the conditions. Query by key also has an interesting side effect. What if we don't know the key,

especially during ad-hoc querying during debugging? Most of the data stores will not give you a list of all the primary keys; even if they did, retrieving lists of keys and then querying for the value would be very cumbersome. Some key-value databases get around this by providing the ability to search inside the value, such as Riak Search that allows you to query the data just like you would query it using Lucene indexes.

While using key-value stores, lots of thought has to be given to the design of the key. Can the key be generated using some algorithm? Can the key be provided by the user (user ID, email, etc.)? Or derived from timestamps or other data that can be derived outside of the database?

These query characteristics make key-value stores likely candidates for storing session data (with the session ID as the key), shopping cart data, user profiles, and so on. The expiry_secs property can be used to expire keys after a certain time interval, especially for session/shopping cart objects.

Bucket bucket = getBucket(bucketName);

IRiakObject riakObject = bucket.store(key, value).execute();

When writing to the Riak bucket using the storeAPI, the object is stored for the key provided.

Similarly, we can get the value stored for the key using the fetchAPI.

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

Riak provides an HTTP-based interface, so that all operations can be performed from the web browser or on the command line using curl. Let’s save this data to Riak:

```
{
"lastVisit":132
4669989288,
"user":{
   "customerId
   ":"91cfdf5b
   cb7c",
   "name":"bu
   yer",
   "countryCo
   de":"US",
   "tzOffset":0
   }
}
```

Use the curlcommand to POSTthe data, storing the data in the sessionbucket with the key of a7e618d9db25 (we have to providethis key):

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

The data for the key a7e618d9db25can be fetched by using the curlcommand:

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

Q. 5 a) Explain the features of Key Value stores.

While using any NoSQL data stores, there is an inevitable need to understand how the features compare to the standard RDBMS data stores that we are so used to. The primary reason is to understand what features are missing and how does the application architecture need to change to better use the features of a key-value data store. Some of the features we will discuss for all the NoSQL data stores are consistency, transactions, query features, structure of the data, and scaling.

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 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 number Of Nodes To Respond To Write set by w to be the same as n Val. Of course, doing that will decrease the write performance of the cluster. To improve on write or read conflicts, we can change the allow Siblings flag during bucket creation: If it is set to false, we let the last write to win and not create siblings.

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 W value
—replication factor—during the write API call.
Assume we have a Riak cluster with a replication factor of 5 and we supply the W value 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 N equal to 5
and with a W value 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.

Query Features

All key-value stores can query by the key—and that's about it. If you have requirements to query by using some attribute of the value column, it's not possible to use the database: Your application needs to read the value to figure out if the attribute meets the conditions. Query by key also has an interesting side effect. What if we don't know the key, especially during ad-hoc querying during debugging? Most of the data stores will not give you a list of all the primary keys; even if they did, retrieving lists of keys and then querying for the value would be very cumbersome. Some key-value databases get around this by providing the ability to search inside the value, such as Riak Search that allows you to query the data just like you would query it using Lucene indexes.
While using key-value stores, lots of thought has to be given to the design of the key. Can the key be generated using some algorithm? Can the key be provided by the user (user ID, email, etc.)? Or derived from timestamps or other data that can be derived outside of the database?
These query characteristics make key-value stores likely candidates for storing session data (with the session ID as the key), shopping cart data, user profiles, and so on. The expiry_secs property can be used to expire keys after a certain time interval, especially for session/shopping cart objects.
Bucket bucket = getBucket(bucketName);
IRiakObject riakObject = bucket.store(key, value).execute();
When writing to the Riak bucket using the storeAPI, the object is stored for the key provided.
Similarly, we can get the value stored for the key using the fetchAPI.

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

Riak provides an HTTP-based interface, so that all operations can be performed from the web browser or on the command line using curl. Let's save this data to Riak:

```
{
"lastVisit":132
4669989288,
"user":{
   "customerId
   ":"91cfdf5b
   cb7c",
   "name":"bu
   yer",
   "countryCo
   de":"US",
   "tzOffset":0
   }
}
```

Use the curlcommand to POSTthe data, storing the data in the sessionbucket with the key of a7e618d9db25 (we have to providethis key):

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

The data for the key a7e618d9db25can be fetched by using the curlcommand:

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

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.

Scaling

Many key-value stores scale by using sharding ("Sharding," p. 38). 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).

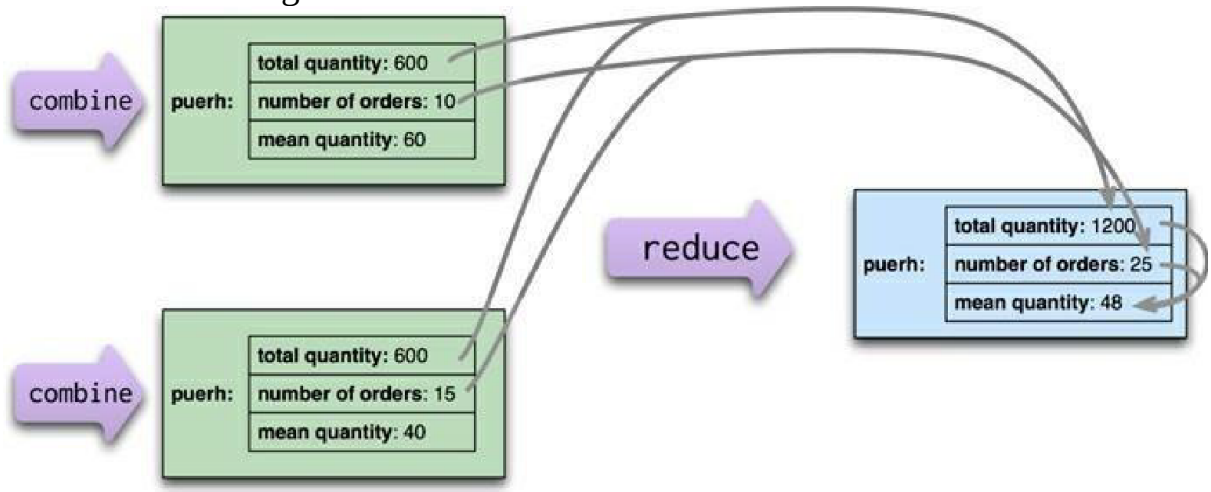
Let's assume we have a 5-node Riak cluster. Setting N to 3 means that all data is replicated to at least three nodes, setting R to 2 means any two nodes must reply to a GET request for it to be considered successful, and setting W to 2 ensures that the PUT request is written to two nodes before the write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

Q. 6 **Create a map reduce model for calculating average ordered quantity of each product. Explain how are calculations composed in map reduce. Explain with neat diagram and example.**

The map-reduce approach is a way of thinking about concurrent processing that trades off flexibility in how you structure your computation for a relatively straightforward model for parallelizing the computation over a cluster. Since it's a tradeoff, there are constraints

on what you can do in your calculations. Within a map task, you can only operate on a single aggregate. Within a reduce task, you can only operate on a single key. This means you have to think differently about structuring your programs so they work well within these constraints.
One simple limitation is that you have to structure your calculations around operations that fit in well with the notion of a reduce operation. A good example of this is calculating averages. Let's consider the kind of orders we've been looking at so far; suppose we want to know the average ordered quantity of each product. An important property of averages is that they are not comparable.
that is, if I take two groups of orders, I can't combine their averages alone. Instead, I need to take total amount and the count of orders from each group, combine those, and then calculate the average from the combined sum and count.



This notion of looking for calculations that reduce neatly also affects how we do counts. To make a count, the mapping function will emit count fields with a value of 1, which can be summed to get a total count.