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

Internal Assessment Test 2 – June 2022

Sub:	NoSQL database					Sub Code:	18CS823	Branch :	CSE	
Date:	4/6/2022	Duration:	90 mins	Max Marks:	50	Sem/Sec :	VIII/A,B,C		OBE	

	Scheme	MARKS	CO	RBT
1	Explain i) Incremental Map-Reduce process ii) Scaling in key-value store Incremental Map-Reduce process -5 marks scaling in key-value store-5 marks	[10]	CO2	L2
2	Describe Map-reduce process to compare the sales of products for each month in 2011 to the prior year. Use suitable diagrams. Map-reduce process with description -8 marks Diagrams-2 marks	[10]	CO2	L3
3	Explain the importance of partitioning and combining in the Map-reduce process. Diagram-2 marks partitioning and combining -8 marks	[10]	CO2	L2
4	Explain key-value store with respect to consistency and transactions. Consistency in key-value store- 5 marks Transaction processing in key-value store- 5 marks	[10]	CO1	L2
5	Identify the situations where key-value store is i) applicable ii) not advisable. Justify your answer. key-value store is applicable with justification - 5 marks Not advisable with justification -5 marks	[10]	CO2	L3
6	Explain single server, master slave, and peer to peer distribution models. Single server- 3 marks master slave- 3 marks peer to peer- 4 marks	[10]	CO3	L2

solution

1. Explain A)incremental Map-Reduce process.

- Many map-reduce computations take a while to perform, even with clustered hardware, and new data keeps coming in which means we need to rerun the computation to keep the output up to date.

- Starting from scratch each time can take too long, so often it's useful to structure a map-reduce computation to allow incremental updates, so that only the minimum computation needs to be done.

- The map stages of a map-reduce are easy to handle incrementally—only if the input data changes does the mapper need to be rerun.

- Since maps are isolated from each other, incremental updates are straightforward.

- The more complex case is the reduce step, since it pulls together the outputs from many maps and any change in the map outputs could trigger a new reduction.

- This re-computation can be lessened depending on how parallel the reduce step is. If we are partitioning the data for reduction, then any partition that's unchanged does not need to be re-reduced.

- Similarly, if there's a combiner step, it doesn't need to be rerun if its source data hasn't changed.

- If our reducer is combinable, there's some more opportunities for computation avoidance.

- If the changes are additive—that is, if we are only adding new records but are not changing or deleting any old records—then we can just run the reduce with the existing result and the new additions.

- If there are destructive changes, that is updates and deletes, then we can avoid some recomputation by breaking up the reduce operation into steps and only recalculating those steps whose inputs have changed—essentially, using a Dependency Network [Fowler DSL] to organize the computation.

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 :

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.

Choose a W value to match your consistency needs; these values can be set as defaults during bucket creation.

2. Describe Map-reduce process to compare the sales of products for each month in 2011 to the prior year. Use suitable diagrams. [7 marks]

Consider an example where **we want to compare the sales of products for each month in 2011 to the prior year**. To do this, we'll break the calculations down into two stages.

- The first stage will produce records showing the aggregate figures for a single product in a single month of the year.
- The second stage then uses these as inputs and produces the result for a single product by comparing one month's results with the same month in the prior year (see Figure 7.8).

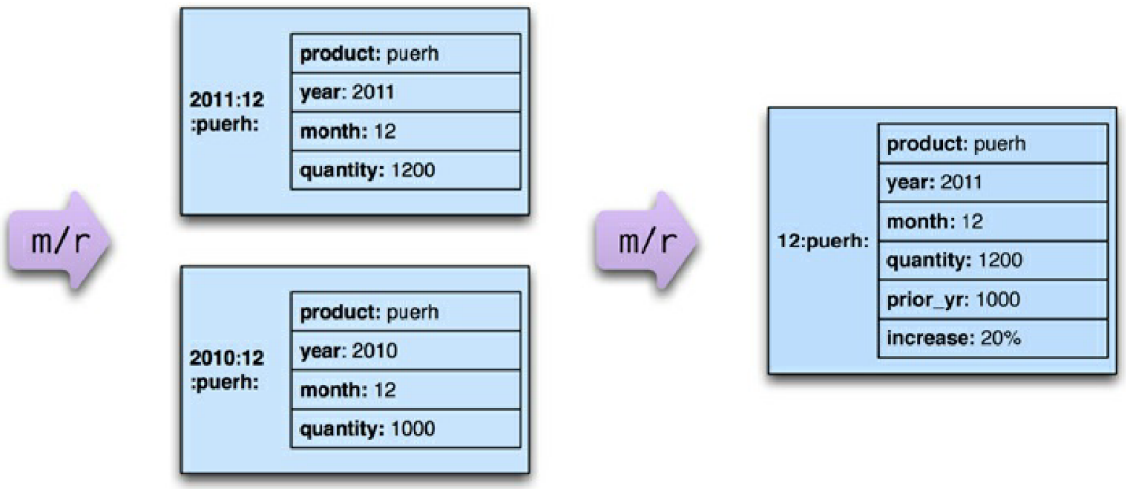

Figure 7.8. A calculation broken down into two map-reduce steps, which will be expanded in the next three figures

- A first stage (Figure 7.9) would read the original order records and output a series of key-value pairs for the sales of each product per month.

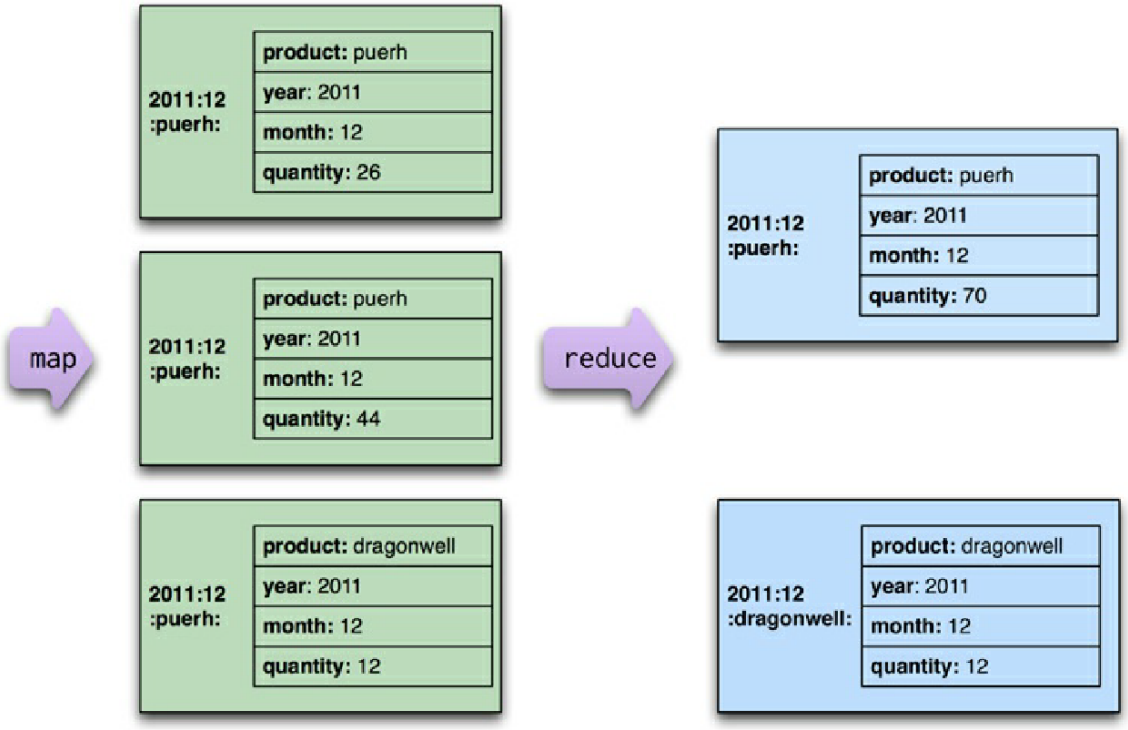

Figure 7.9. Creating records for monthly sales of a product

- The only new feature is using a composite key so that we can reduce records based on the values of multiple fields.
- The second-stage mappers (Figure 7.10) **process this output depending on the year.** A 2011 record populates the current year quantity while a 2010 record populates a prior year quantity. Records for earlier years (such as 2009) don't result in any mapping output being emitted.

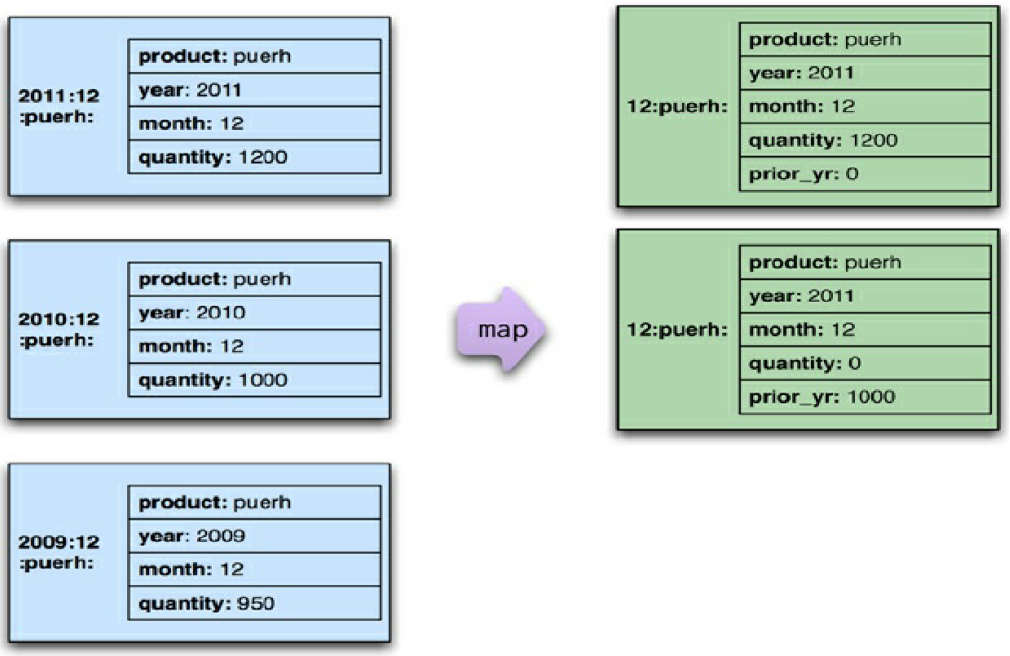

Figure 7.10. The second stage mapper creates base records for year-on-year comparisons.

- The reduce in this case (Figure 7.11) is a merge of records, where combining the values by summing allows two different year outputs to be reduced to a single value (with a calculation based on the reduced values thrown in for good measure).

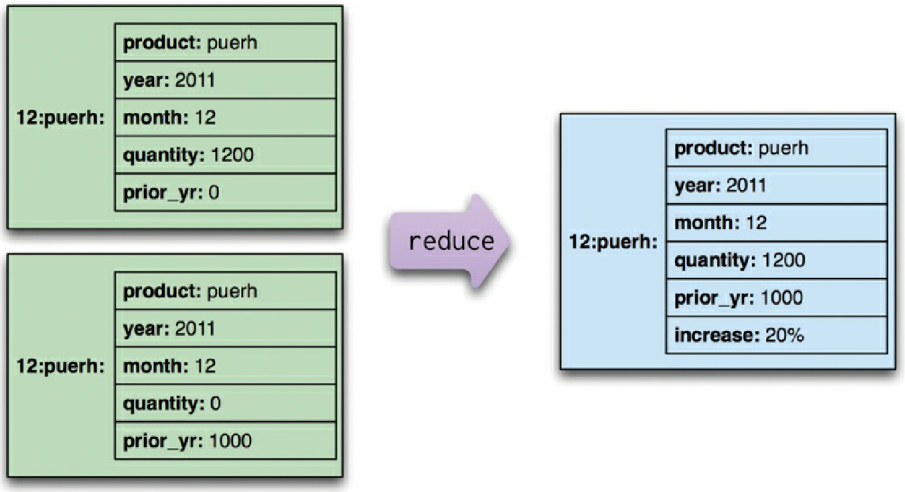

Figure 7.11. The reduction step is a merge of incomplete records.

3. Explain the importance of partitioning and combining in the Map-reduce process

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 7.3)**.

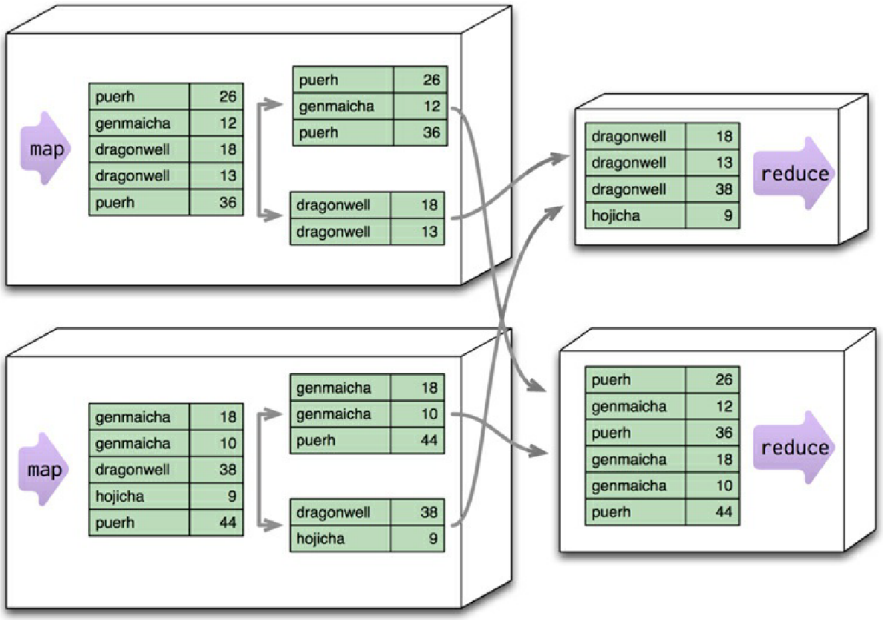

Figure 7.3. Partitioning allows reduce functions to run in parallel on different keys.

1. The first thing we can do is **increase parallelism by partitioning the output of the mappers**.
2. **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.**
3. **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.**
4. 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.**
5. **Multiple reducers can then operate on the partitions in parallel, with the final results merged together.** (This step is also called "shuffling," and the partitions are sometimes referred to as "buckets" or "regions.")
6. 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.
7. **A combiner function cuts this data down by combining all the data for the same key into a single value** (see Figure 7.4).
8. **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.**
9. **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.**

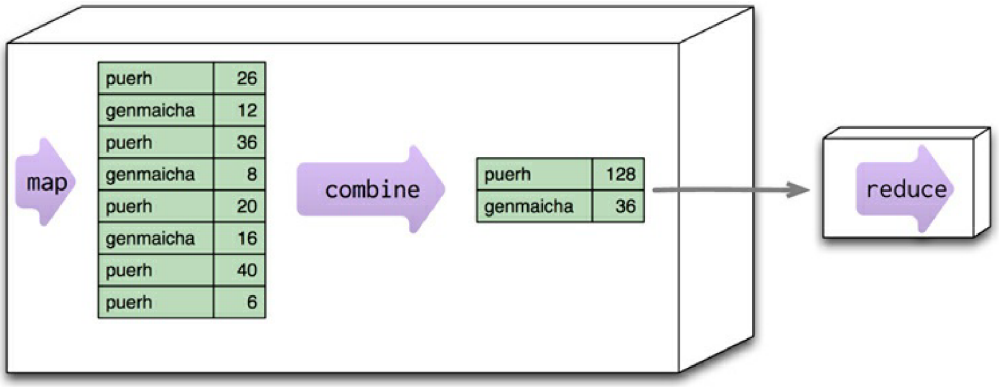

Figure 7.4. Combining reduces data before sending it across the network.

Difference between Riak and oracle.

Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

4. Explain key-value store with respect to consistency, transactions

Consistency

1. Consistency is applicable only for operations on a single key, since these operations are either a get, put, or delete on a single key. Across key writes are very expensive.

2. In distributed key-value store implementations like Riak, **the eventually consistent model** of consistency is implemented.

3. 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.

4. **I**n Riak, th**ese options can be set up during the bucket creation**.

5. **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**.

6. **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();
```

7. If we need data in every node to be consistent, increase the numberOfNodesToRespondToWrite set by w to be the same as nVal.
8. Of course doing that will decrease the write performance of the cluster.
9. To improve on write or read conflicts, change the allowSiblings 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. There are no guarantees on the writes.

Many data stores do implement transactions in different ways.

Riak uses the concept of quorum 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 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 = 2 nodes being down for write operations, though we would still have lost some data on those nodes for read.

5. Identify the situations where key-value store is applicable and not applicable.

Storing Session Information

- Generally, every web session is unique and is assigned a unique sessionid value.
- Applications that store the sessionid on 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 PUT request 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

To express 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

For saving multiple keys and if 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.

6. Explain single server, master slave, and peer to peer distribution models

   Single Server

   The first and the simplest distribution option is the one we would most often recommend—no distribution at all.

   Run the database on a single machine that handles all the reads and writes to the data store. We prefer this option because it eliminates all the complexities that the other options introduce; it's easy for operations people to manage and easy for application developers to reason about.

   NoSQL can be used with a single-server distribution model if the data model of the NoSQL store is more suited to the application. Graph databases are the obvious category here—these work best in a single-server configuration. If your data usage is

mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it's easier on application developers.

4.2. Sharding

Often, a busy data store is busy because different people are accessing different parts of the dataset. In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called sharding (see Figure 4.1).

> In order to get close to it we have to ensure that data that's accessed together is clustered together on the same node and that these clumps are arranged on the nodes to provide the best data access.

How to clump the data up so that one user mostly gets her data from a single server?. 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. Ex. 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 depending on application load.

In some cases, it's useful to put aggregates together if you think they may be read in sequence. The Bigtable paper [Chang etc.] described keeping its rows in lexicographic order and sorting web addresses based on reversed domain names (e.g., com.martinfowler).

This way data for multiple pages could be accessed together to improve processing efficiency.

Historically most people have done sharding as part of application logic. EX. You might put all customers with surnames starting from A to D on one shard and E to G on another.

This complicates the programming model, as application code needs to ensure that queries are distributed across the various shards.

Furthermore, rebalancing the sharding means changing the application code and migrating the data.

Many NoSQL databases offer auto-sharding, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard.

This can make it much easier to use sharding in an application.

(sharding vs Replication)
Sharding is particularly valuable for performance because it can improve both read and write performance.
Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.
(How does sharding decrease resilience? / Why is single node sharding going to be tricky?)
Sharding does little to improve resilience when used alone. Although the data is on different nodes, a node failure makes that shard's data unavailable just as surely as it does for a single-server solution. The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing. With a single server it's easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you're more likely to get a node failure. So in practice, sharding alone is likely to decrease resilience.

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

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

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

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. (This includes such things as putting the reads and writes through separate database connections—a facility that is not often supported by database interaction libraries.)
->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.
->Even if you use master-slave replication just for hot backup this can be a concern, because if the master fails, any updates not passed on to the backup are lost.
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.

4.4. Peer-to-Peer Replication

Peer-to-peer replication (see Figure 4.3) 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.

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 reading lead to problems but at least they are relatively transient.
Inconsistent writes are forever.
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.
The inconsistent write can be solved with a policy to merge inconsistent writes. In this case we can get the full performance benefit of writing to any replica.