CELEBRATING 25 YEARS
CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

No.132, AECS Layout
I.T. Park Road, Bangalore 560 037
T: +91 80 28524466/77, F : +91 80 28524630
E: info@cmr.ac.in | **www.cmr.ac.in**

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Date: 12th March 2023

Name of the course	:	**NOSQL**	Sub Code	:	18CS823
Name of the Faculty/s	:	Dr. Sudhakar K.N	Sem & Sec	:	VIII - A

IAT - 01, Scheme Solution

Q-01: Explain or differentiate Relational Model and Aggregate model with an example.

Solution
Data Model: Model through which we identify and manipulate our data. It describes how we interact with the data in the database. **Storage model:** Model which describes how the database stores and manipulates the data internally. The dominant data model is a relational data model which uses a set of tables:

- Each table has rows.
- Each row representing an entity.
- Column describe entity.
- Column may refer to relationship

In NoSQL, “data model” refer to the model by which the database organizes data, more formally called a metamodel. NoSQL move away from the relational model. Each NoSQL solution has a different model that it uses:

1) Key-value
2) Document
3) Column-family
4) Graph

Out of these first three share a common characteristic of their data models which is called as aggregate orientation.
Aggregates The **relational model** takes the information to store and divides it into tuples. A tuple is a limited data structure:

- You cannot nest one tuple within another to get nested records.
- You cannot put a list of values or tuples with in another.

Aggregate model recognizes that often we need to operate on data that have a more complex structure than a set of tuples.

- It has a complex record that allows lists and other record structures to be nested inside it.
- key-value, document, and column-family databases all make use of this more complex record.
- A common term used for this complex record is **"aggregate."**

Definition: In Domain-Driven Design, an aggregate is a collection of related objects that we wish to treat as a unit. It is a unit for data manipulation and management of consistency. Typically, we like to update aggregates with atomic operations and communicate with our data storage in terms of aggregates.

Q-02: Explain common characteristics of NoSQL databases.
Solution
For NoSQL there is no generally accepted definition, nor an authority to provide one, so all we can do is discuss some common characteristics of the databases that tend to be called "NoSQL."
Characteristics of NoSQL Databases

- The name NoSQL comes from the fact that the NoSQL databases 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.
- They are generally open-source projects.
- Most NoSQL databases are driven by the need to run on clusters. Relational databases use ACID transactions to handle consistency across the whole database. This inherently clashes with a cluster environment, so NoSQL databases offer a range of options for consistency and distribution.
- Not all NoSQL databases are strongly oriented towards running on clusters. Graph databases are one style of NoSQL databases that uses a distribution model similar to relational databases but offers a different data model that makes it better at handling data with complex relationships.
- NoSQL databases operate without a schema, allowing you to freely add fields to database records without having to define any changes in structure first. This is particularly useful when dealing with non-uniform data and custom fields, which forced relational databases to use names like customField6 or custom field tables that are awkward to process and understand.
- When you first hear "NoSQL," an immediate question is what does it stand for—a "no" to SQL? Most people who talk about NoSQL say that it really means "Not Only SQL," but this interpretation has a couple of problems. Most people write "NoSQL" whereas "Not Only SQL" would be written "NOSQL."
- To resolve these problems, don't worry about what the term stands for, but rather about what it means. Thus, when "NoSQL" is applied to a database, it refers to an ill-defined set of mostly open-source databases, mostly developed in the early 21st century, and mostly not using SQL.
- It's better to think of NoSQL as a movement rather than a technology. We don't think that relational databases are going away—they are still going to be the most common form of database in use. Their familiarity, stability, feature set, and available support are compelling arguments for most projects.
- The change is that now we see relational databases as one option for data storage. This point of view is often referred to as polyglot persistence—using different data stores in different circumstances.

- We need to understand the nature of the data we're storing and how we want to manipulate it. The result is that most organizations will have a mix of data storage technologies for different circumstances. In order to make this polyglot world work, our view is that organizations also need to shift from integration databases to application databases.
- In our account of the history of NoSQL development, we've concentrated on big data running on clusters. The big data concerns have created an opportunity for people to think freshly about their data storage needs, and some development teams see that using a NoSQL database can help their productivity by simplifying their database access even if they have no need to scale beyond a single machine.

Two primary reasons for considering NoSQL:

1) To handle data access with sizes and performance that demand a cluster.
2) To improve the productivity of application development by using a more convenient data interaction style.

A NoSQL is a database that provides a mechanism for storage and retrieval of data, they are used in real-time web applications and big data and their use are increasing over time. Many NoSQL stores compromise consistency in favour of availability, speed and partition tolerance.
Advantages of NoSQL:

1) **High Scalability:** NoSQL databases use sharding for horizontal scaling. It can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.
2) **High Availability:** Auto replication feature in NoSQL databases makes it highly available.

Disadvantages of NoSQL:

1) **Narrow Focus:** It is mainly designed for storage, but it provides very little functionality.
2) **Open Source:** NoSQL is an open-source database that is two database systems are likely to be unequal.
3) **Management Challenge:** Big data management in NoSQL is much more complex than a relational database.
4) **GUI is not available:** GUI mode tools to access the database is not flexibly available in the market.
5) **Backup:** it is a great weak point for some NoSQL databases like MongoDB.
6) **Large Document size:** Data in JSON format increases the document size.

Q-03: What is Polyglot persistence?

Solution

- Different databases are designed to solve different problems. Using a single database engine for all of the requirements usually leads to non-performant solutions; storing transactional data, caching session information, traversing graph of customers and the products their friends bought are essentially different problems.

- Even in the RDBMS space, the requirements of an OLAP and OLTP system are very different, they are often forced into the same schema.
- Database engines are designed to perform certain operations on certain data structures and data amounts very well—such as operating on sets of data or a store and retrieving keys and their values really fast, or storing rich documents or complex graphs of information.

Disparate Data Storage Needs

- Many enterprises tend to use the same database engine to store business transactions, session management data, and for other storage needs such as reporting, BI, data warehousing, or logging information as shown in Figure 1
- The session, shopping cart, or order data do not need the same properties of availability, consistency, or backup requirements. Does session management storage need the same rigorous backup/recovery strategy as the e-commerce orders data? is questionable.
- In 2006, Neal Ford coined the term polyglot programming, to express the idea that applications should be written in a mix of languages to take advantage of the fact that different languages are suitable for tackling different problems. Complex applications combine different types of problems, so picking the right language for each job may be more productive than trying to fit all aspects into a single language.
- Similarly, when working on an e-commerce business problem, using a data store for the shopping cart which is highly available and can scale is important, but the same data store cannot help you find products bought by the customers' friends—which is a totally different question. We use the term polyglot persistence to define this hybrid approach to persistence.

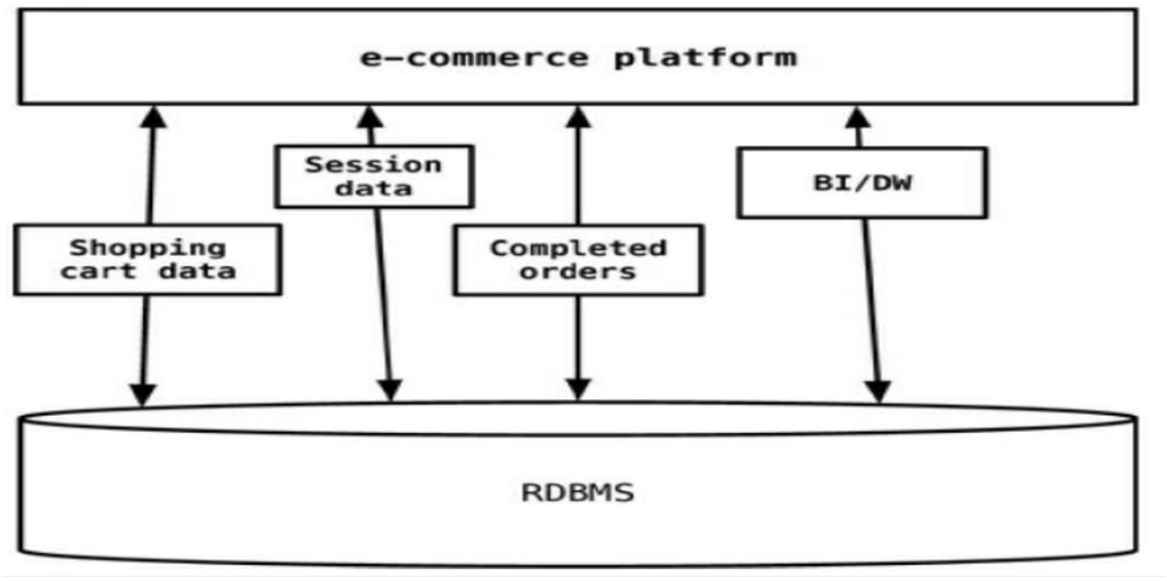

Figure 1: Use of RDBMS for every aspect of storage for the application

Polyglot Data Store Usage

- Let's take our e-commerce example and use the polyglot persistence approach to see how some of these data stores can be applied, as show in Figure 2.
- A key-value data store could be used to store the shopping cart data before the order is confirmed by the customer and also store the session data so that the RDBMS is not used for this transient data.

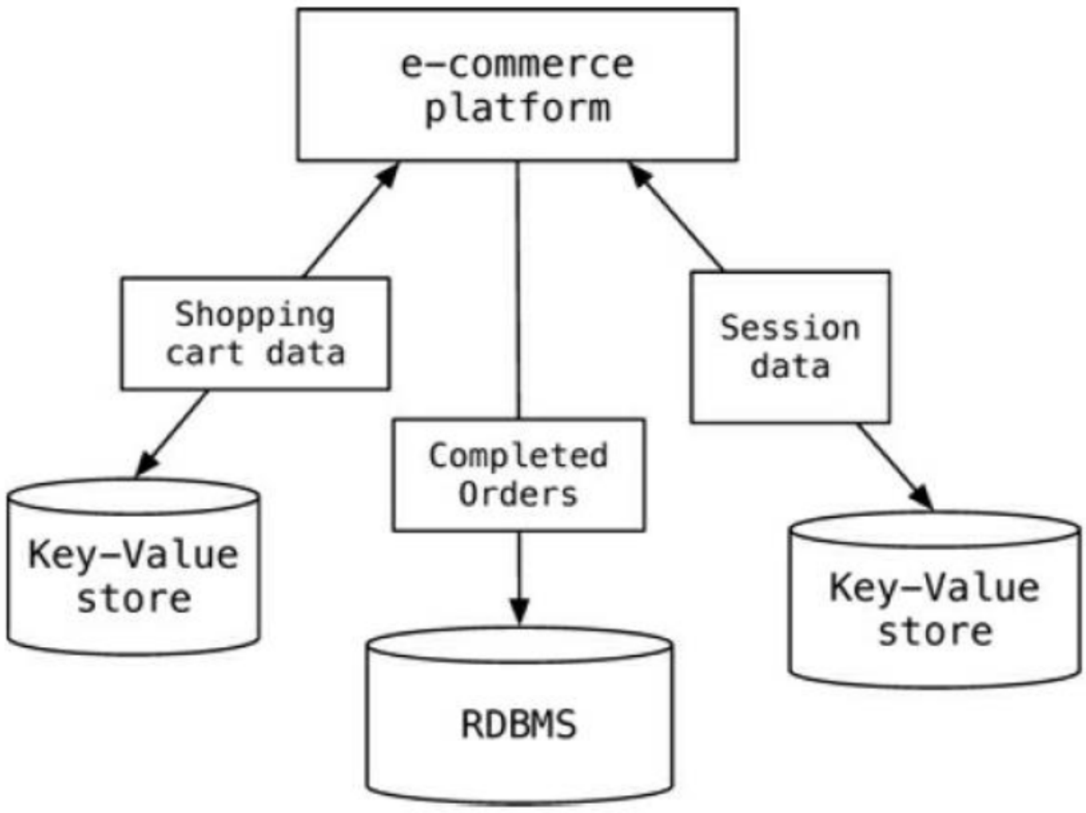

Figure 2: Use of key-value stores to offload session and shopping cart data storage

- Key-value stores make sense here since the shopping cart is usually accessed by user ID and, once confirmed and paid by the customer, can be saved in the RDBMS. Similarly, session data is keyed by the session ID.
- If we need to recommend products to customers when they place products into their shopping carts —for example, "your friends also bought these products" or "your friends bought these accessories for this product"—then introducing a graph data store in the mix becomes relevant as show in Figure 3
- It is not necessary for the application to use a single data store for all of its needs, since different databases are built for different purposes and not all problems can be elegantly solved by a singe database.
- Even using specialized relational databases for different purposes, such as data warehousing appliances or analytics appliances within the same application, can be viewed as polyglot persistence.

Q-04: Discuss master slave replication in detail.
Solution 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 secondary. A replication process synchronizes the slaves with the master

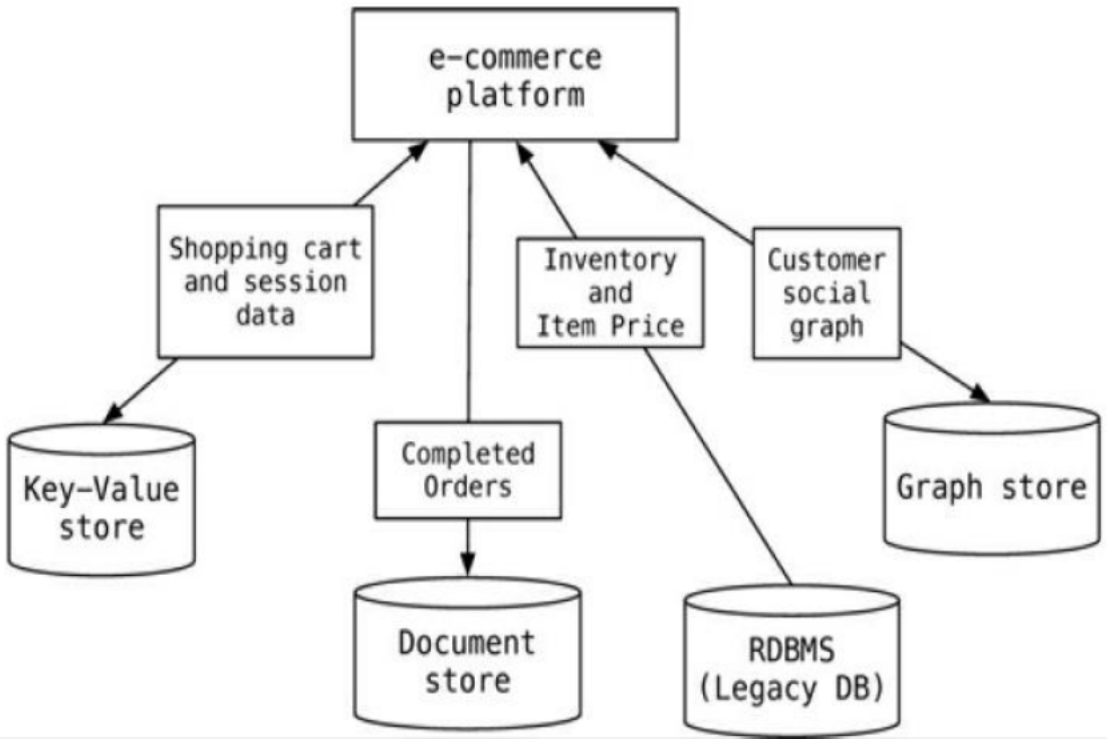

Figure 3: Example implementation of polyglot persistence

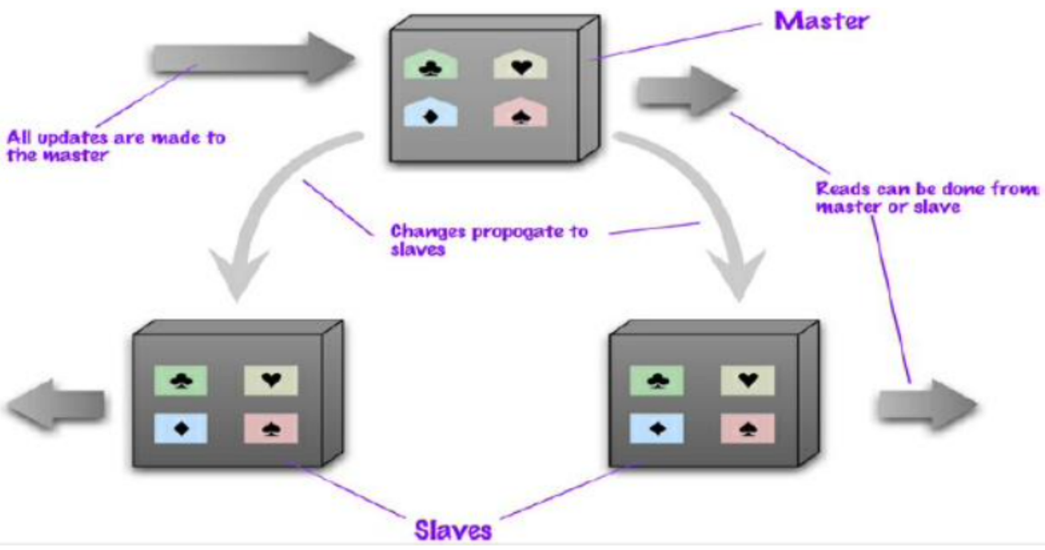
Master
All updates are made to the master
Reads can be done from master or slave
Changes propogate to slaves
Slaves

Figure 4: Data is replicated from master to slaves.

Advantages:

1) Scaling: 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.

2) You are still, 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.

3) Read resilience: if the master fail, the slaves can still handle read requests. Again, this is useful if most of your data access is reads. 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.

4) All read and write traffic can go to the master, while the slave acts as a hot backup. In this case, it's easiest to think of the system as a single-server store with a hot backup. You get the convenience of the single-server configuration but with greater resilience— which is particularly handy if you want to be able to handle server failures gracefully.

5) Masters can be appointed manually or automatically.

6) Manual appointing typically means that when you configure your cluster, you configure one node as the master.

7) With automatic appointment, you create a cluster of nodes, and they elect one of themselves to be the master.

8) Apart from simpler configuration, automatic appointment means that the cluster can automatically appoint a new master when a master fails, reducing downtime.

9) Replication comes with some attractive benefits, but it also comes with an unavoidable dark side— inconsistency.

10) You have the danger that different clients, reading different slaves, will see different values because the changes haven't all propagated to the slaves.

11) In the worst case, that can mean that a client cannot read a write it just made.

12) 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.

Q-05: What is CAP theorem? How it is applicable to NO SQL systems? Explain

Solution The CAP Theorem

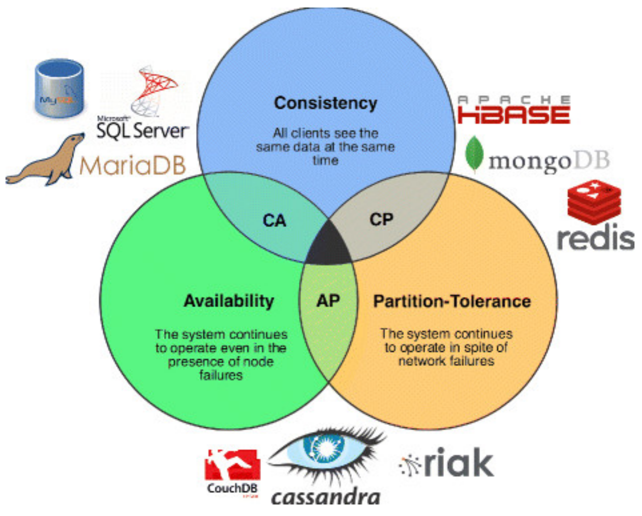
Consistency
All clients see the same data at the same time
CA
CP
AP
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

Figure 5: CAP Theorem

- In the NoSQL world refer CAP theorem as a reason why you may need to relax consistency.

- 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.
- **Consistency** means that data is the same across the cluster, so you can read or write from/to any node and get the same data.
- **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.
- **Partition tolerance** means that the cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other.

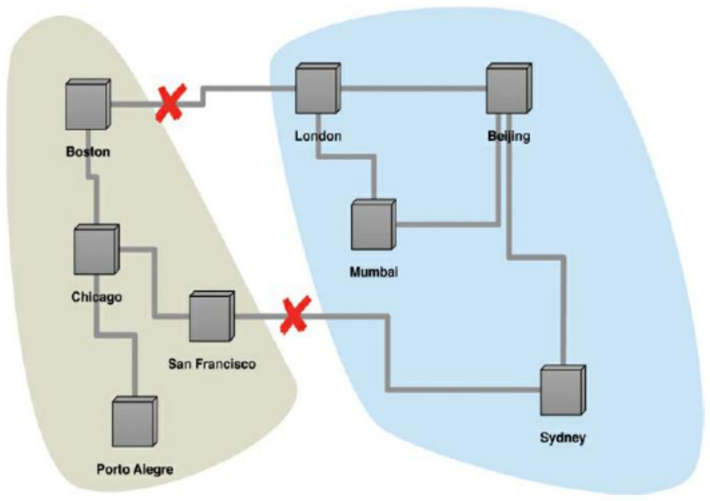

Figure 6: With two breaks in the communication lines, the network partitions into two groups.

- A single-server system is the obvious example of a CA system. *(A system that has Consistency and 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.
- 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's special usage of "availability" gets confusing. CAP defines "availability" to mean "every request received by a non failing node in the system must result in a response". So a failed, unresponsive node doesn't conclude 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.
- 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 compromise consistency versus availability.

- Often, you can compromise 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.
- Example : 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 if 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. If 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.
- We still can't book a room on the London node for the hotel whose master is in Mumbai if the connection goes down.
- In CAP terminology, this is a failure of availability in 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 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.
- 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.

- 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.
- Different data items may have different tolerances for staleness, and thus may need different settings in your replication configuration.
- Promoters of NoSQL often say that instead of following the ACID properties of relational transactions, NoSQL systems follow the BASE properties (Basically Available, Soft state, Eventual consistency).
- It's usually better to think not about the tradeoff between consistency and availability, but rather between consistency and latency (response time).
- We can improve consistency by getting more nodes involved in the interaction, but each node we add increases the response time of that interaction.
- We can then think of availability as the limit of latency that we're prepared to tolerate; once latency gets too high, we give up and treat the data as unavailable—which neatly fits its definition in the context of CAP.

Q-06: Explain the concept of sharding with example.

Solution Sharding Often, a busy data store is busy because different people are accessing different

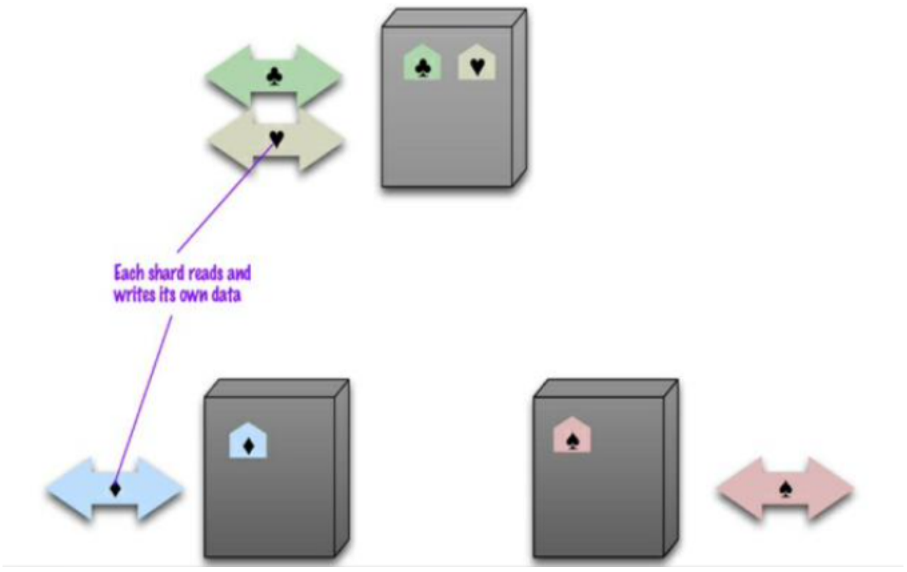

Figure 7: Sharding puts different data on separate nodes, each of which does its own reads and writes.

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.

- 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.

- In order to get close to the ideal case, we have to ensure that data that's accessed together is clumped together on the same node and that these clumps are arranged on the nodes to provide the best data access.
- Data should be clump up such that one user mostly gets her data from a single server. This is where aggregate orientation comes in really handy. Aggregates, designed to combine data that's commonly accessed together—so aggregates leap out as an obvious unit of distribution.
- While arranging the data on the nodes, there are several factors that can help to improve performance.
- If most accesses of certain aggregates are based on a physical location, place the data close to where it's being accessed.
- Example: 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. 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.
- 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.
- In some cases, it's useful to put aggregates together if you think they may be read in sequence.
- Historically, most people have done sharding as part of application logic. 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 is particularly valuable for performance because it can improve both read and write performance.
- Using replication, particularly with caching, can greatly improve read performance but does little for applications that have a lot of writes. Sharding provides a way to horizontally scale writes.
- Sharding does little to improve resilience when used alone. Although the data is on different nodes, a node failure makes that shard's data unavailable just as surely as it does for a single-server solution.
- The resilience benefit it does provide is that only the users of the data on that shard will suffer; however, it's not good to have a database with part of its data missing.
- With a single server, it's easier to pay the effort and cost to keep that server up and running; clusters usually try to use less reliable machines, and you're more likely to get a node failure. So in practice, sharding alone is likely to decrease resilience.
- Despite the fact that sharding is made much easier with aggregates, it's still not a step to be taken lightly.

- 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. The lesson here is to use sharding well before you need to—when you have enough headroom to carry out the sharding.
