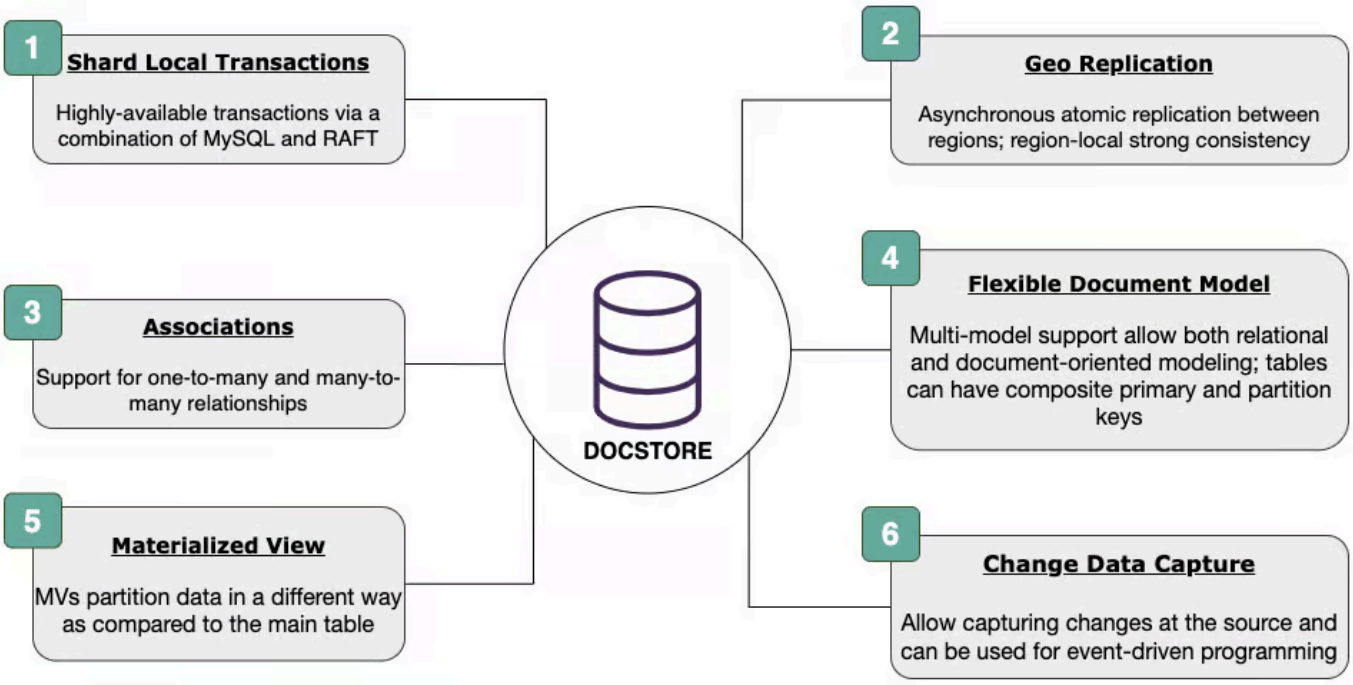
Evolving schemaless into a Distributed SQL DB



- evolution of schemaless into a general-purpose transactional DB called Docstore.
- Docstore is general-purpose multi-model database that provides a strict serializability consistency model on a partition level.

```
- **In Docstore**, **serializability** is guaranteed within a
**partition**
(a subset of the database), meaning all updates and queries in a partition
will appear
to happen in a strict sequence. This is especially useful when dealing
with data
that is updated frequently and needs to stay consistent,
like bank transactions or order statuses.
```

Comparison: Schemaless vs Docstore

- **Schemaless** was mainly a **key-value store** that allowed Uber to store large amounts of semi-structured data without a fixed schema. It was designed for high throughput and flexibility but lacked some of the advanced features required for transactional consistency and relational modeling.

- **Docstore**, on the other hand, builds on the ideas from Schemaless but introduces powerful **transactional** and **multi-model** capabilities, such as:
  - Strict serializability.
  - Materialized views for fast querying.
  - Support for relationships between data (associations).
  - Real-time change tracking (CDC).
- The eventual consistency offered by Cassandra ended up impeding developer productivity as they had to design around the lack of strong consistency and this complicated application architecture.
- We came to the conclusion that evolving schemaless into a general-purpose transactional DB would be the best choice.
- Schemaless had traditionally been a highly reliable system but now we needed to focus on usability while achieving similar or better reliability.

Design Consideration:

- We didn't want to build a NoSQL system.
- wanted to provide best of both worlds: schema flexibility of document models and schema enforcement seen in traditional relational models.
- designed tables in docstore to enforce a schema on the data
- applications that work with data usually assume some kind of structure.
- means either they utilize schema-on-read in which case the application interprets data when it is read or
- they utilize schema-on-write which ensures that schema is explicit and DB ensures schema conformity of data.
- with schema-enforcement as above, doscstore also provides schema flexibility.
- shcema can evolve.
- records with different schemas are allowed to co-exist and schema updates don't require full table rebuilds.

---

1. Schema-on-Write vs. Schema-on-Read

- **Schema-on-Write**:
  - When data is **written** to the DB, it must **conform to a specific schema**. The schema is enforced **before** the data is saved in the DB.
  - **Example**: Imagine a table that stores user information. Before a new user's data is stored (like name, email, and age), the database checks if these fields follow the

expected format (e.g., email must be a valid email format). If any required field is missing or incorrect, the data won't be stored.
    - **Docstore's Approach**: Docstore uses **schema-on-write** by default. This means that when you store data in Docstore, it will be checked against the defined schema to ensure it meets the structure.
- **Schema-on-Read**: The data is stored without enforcing a strict schema. The schema is applied **only when the data is read**, meaning the DB doesn’t care what the data looks like until it is actually used.
    - **Example**: In a schema-on-read system like MongoDB, if you store a record with an arbitrary structure (like different fields for each document), the database will not check the structure when saving the data. The schema is only interpreted when you read the data.
    - **Docstore's Approach**: Docstore doesn't use schema-on-read by default, but it could still allow it in certain cases, depending on how data is consumed.

2. Schema Flexibility in Docstore

Even though **Docstore enforces schema-on-write**, it still provides flexibility:

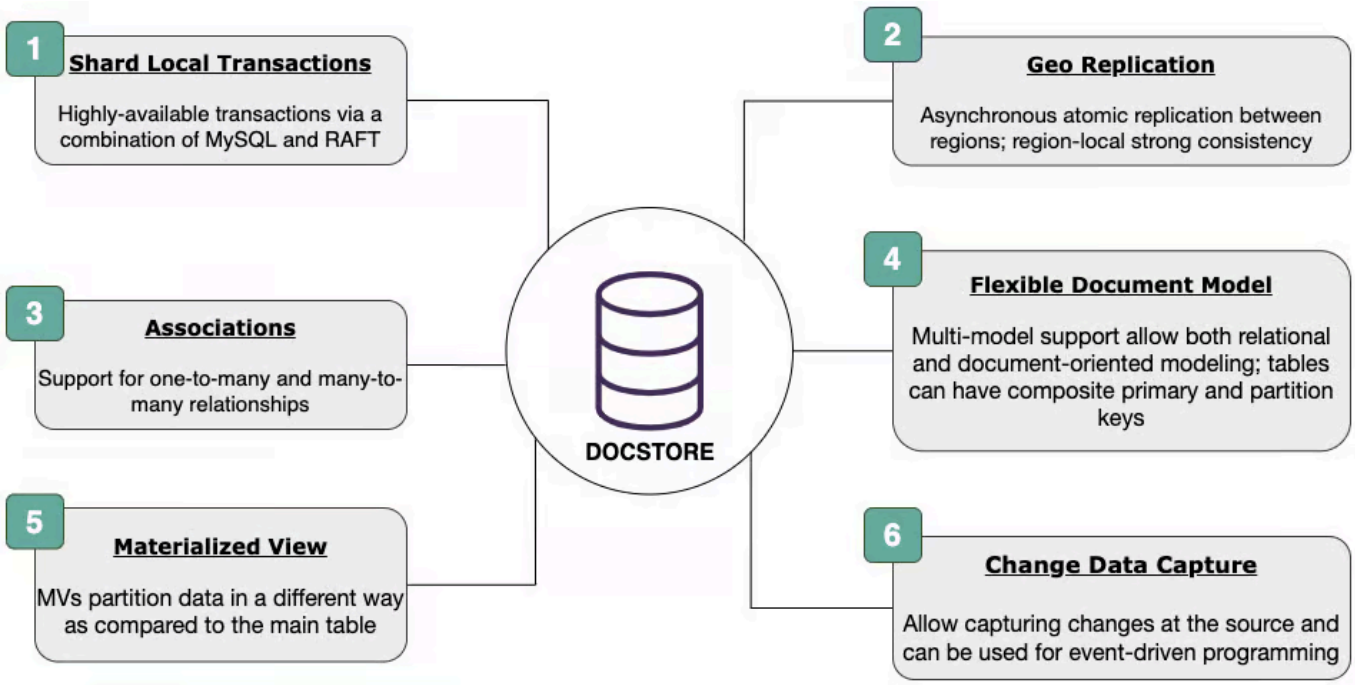
- **Schemas Can Evolve**: Over time, the schema can change without needing to rebuild the entire table or disrupt existing records. For instance, you can add new fields to a record without breaking older records that don’t have those fields.
    - **Example**: Suppose you have a table for storing orders, and initially, it only stores `order_id`, `customer_name`, and `total_amount`. Later, you might want to add a field `delivery_status`. In Docstore, you can add the `delivery_status` field without requiring changes to all the existing records, so old records can still work even without this new field.
- **Different Schemas Co-Exist**: You can have records with different schemas (structure) in the same table. This is useful in situations where data evolves over time or different sources of data need to be combined.
    - **Example**: In an e-commerce system, some product records may contain a `price` field, while others may have a `discounted_price` field. Docstore allows these records with different fields to coexist in the same table, which makes it flexible enough for real-time systems with changing data.
- **Support for Sparse Data**: Docstore supports storing records where some fields might not be populated (i.e., missing or null). This is helpful when dealing with data that may not always have the same structure for every record.
    - **Example**: For some products, you might have a field for `color`, but for others, you may not. Docstore allows those fields to be **sparse**, meaning some records might

have the `color` field, and others won't. This reduces the need for having empty or NULL values everywhere.

- **Support for Nested Data Types**: Docstore allows complex data structures, like **nested objects** or **arrays**, as part of the schema.
  - **Example**: You can store customer data where the `address` is a nested object (containing street, city, and zip code), or where the `order_items` is an array of product details. Docstore can easily handle these nested structures without breaking the schema rules.

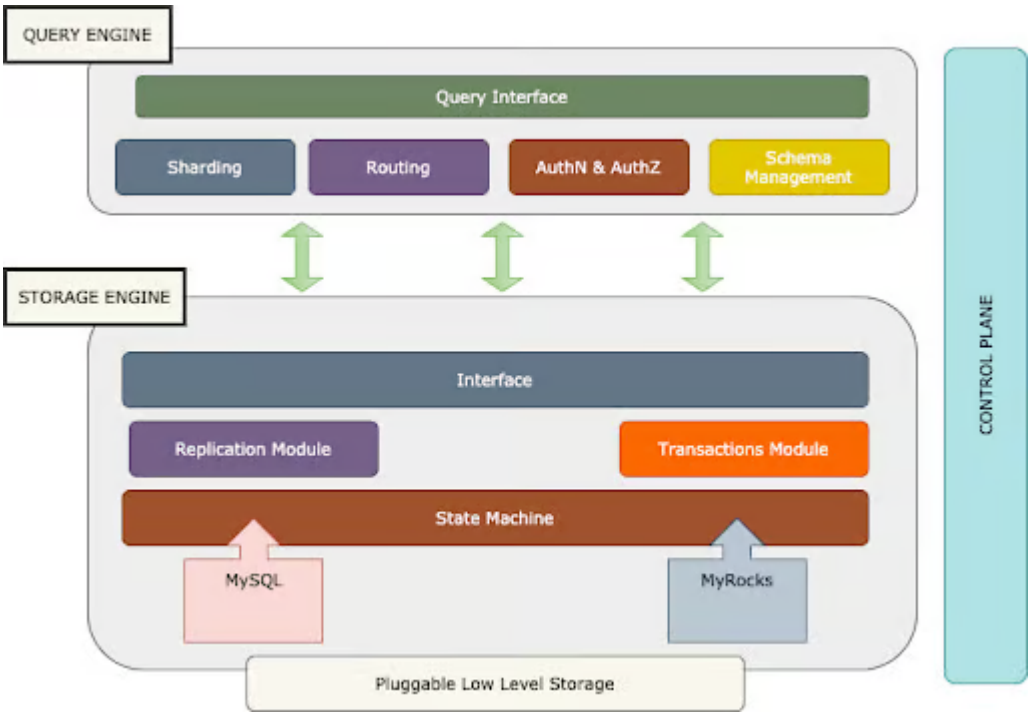
Feature set:

- Docstore has the following features built-in out of the box.



Architecture:

- Docstore has a layered architecture and Docstore deployment is called an instance.
- Each instance is divided into a query engine layer, a storage enginer layer, and a control plane.



- Query layer is stateless and responsible for routing requests to storage layer.

```
- query engine's job is only to route the request to correct storage
partition, which is where the actual data lives.
- it's stateless, means it doesn't store any data about previous requests.
```

- storage enginer, responsible for storing the data, is organized as a set of partitions, data is distributed across these partitions.

```
- **Example**: Think of the **storage engine** as a **library**.
Instead of having all books in one giant pile, the books are organized
into different **shelves** (partitions).
Each shelf holds a specific category of books (data).
If a shelf is full, you can add more shelves to store more books without
overcrowding one spot.
```

- The control plane is responsible for assigning shards to Docstore partitions and adaptively adjusts shard placement in response to failure events.

```
- it is like brain of the system.
- responsible for managing entire DB's operations.
```

```
    - keeps track of data's location, handles any issue related data placement
    or system failures.
```

```
- Shard:
        - is a horizontal division of data in DB.
        - subset of data that is stored on a single machine.
        - goal is to distribute data across multiple machines.
        - this allows DBs to scale more effectively and handle large volumes
of data and high traffic.

- Partition:
        - refers to logical division of data that groups one or more shards
together.
        - In a distributed DB, partitions are used to organize and manage
shards across multiple nodes.
        - A partition can handle one or more shards of data.

- Example:
        - Let’s say you have a database with 4 **shards**:
        -   Shard 1: Contains customers with IDs `1-1000`.
        -   Shard 2: Contains customers with IDs `1001-2000`.
        -   Shard 3: Contains customers with IDs `2001-3000`.
        -   Shard 4: Contains customers with IDs `3001-4000`.
        - These **4 shards** can be grouped into a **single partition**.
        So, the partition might contain data for a certain region or set of
related data, and inside that partition, there could be multiple shards.

- shards hold actual data (like customer info/orders) and are distributed
across multiple nodes.
- partitions are higher level structures that group multiple shards for easier
management and efficient data distribution.
```

Example: Where Are Partitions Stored?

Let's say you have a distributed system with **3 machines (nodes)**, and you are working with a **customer database** where the data is partitioned based on customer IDs.

- **Machine 1 (Node 1)** could store **Partition A** which contains **Shards 1 and 2**.
- **Machine 2 (Node 2)** could store **Partition B** which contains **Shards 3 and 4**.
- **Machine 3 (Node 3)** could store **Partition C** which contains **Shards 5 and 6**.

The **control plane** is responsible for deciding which **partition** goes to which **machine** (node). It uses a hashing algorithm to assign data to these partitions and distributes the partitions evenly across machines.

- For example, customer data with **ID 1001** might be assigned to **Partition A**, which is stored on **Machine 1**.
- A customer with **ID 2001** might be assigned to **Partition B**, stored on **Machine 2**.
- And so on for other data.

Handling Failures and Load Balancing

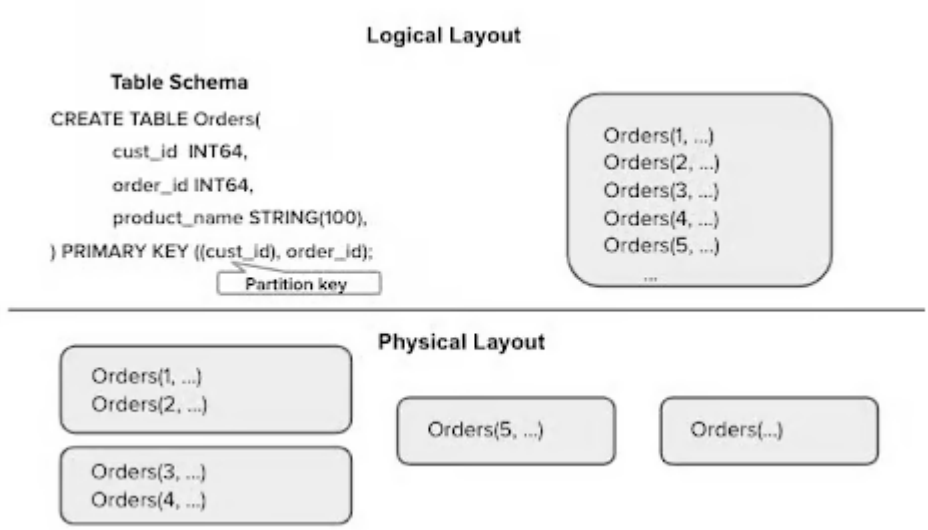
- If **Machine 1** goes down, the **control plane** can reassign **Partition A** (and its shards) to **another machine** (say **Machine 2** or **Machine 3**) so the system remains available.
- The system may also **replicate** partitions so that **Partition A** is also stored on **Machine 2** or **Machine 3**. This ensures that if a machine fails, there is still a backup copy of the data.

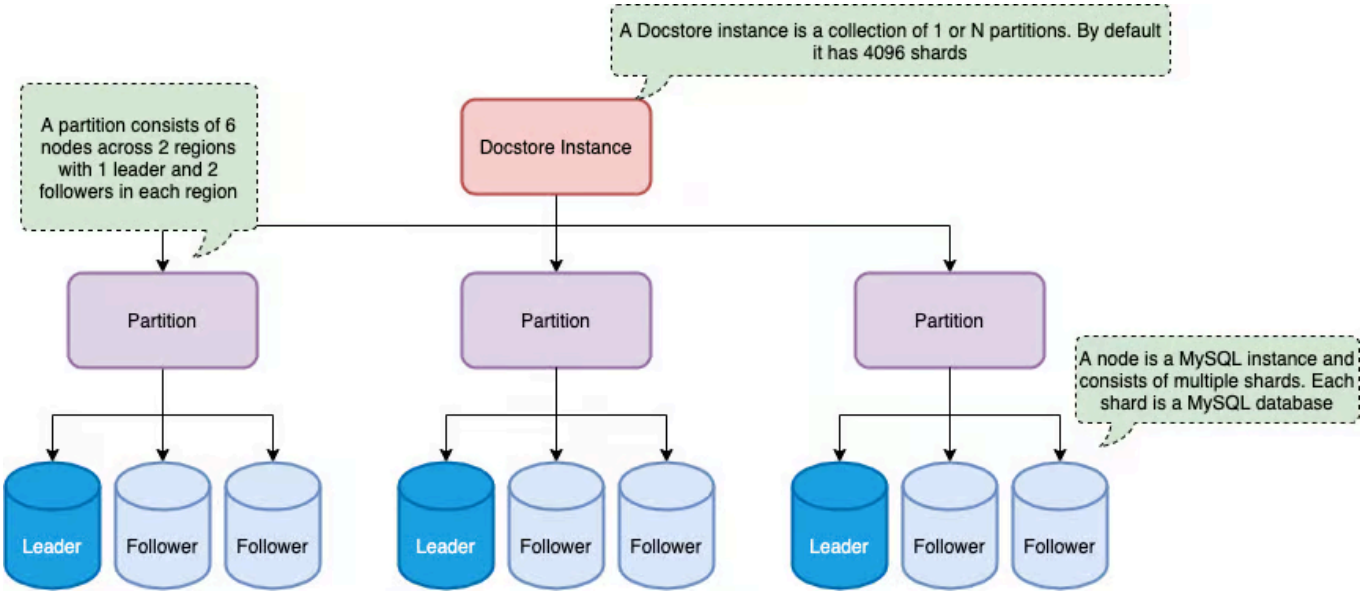
Dynamic Shard and Partition Movement

- The **control plane** can **reassign** partitions or **move shards** dynamically to **balance the load**. For example, if **Machine 1** is getting too much traffic, the system might decide to move **some of Partition A's shards** to **Machine 3** to balance the load across the machines.
- This allows the system to **scale out** (add more machines) or **scale in** (remove machines) as needed, without interrupting access to the data.

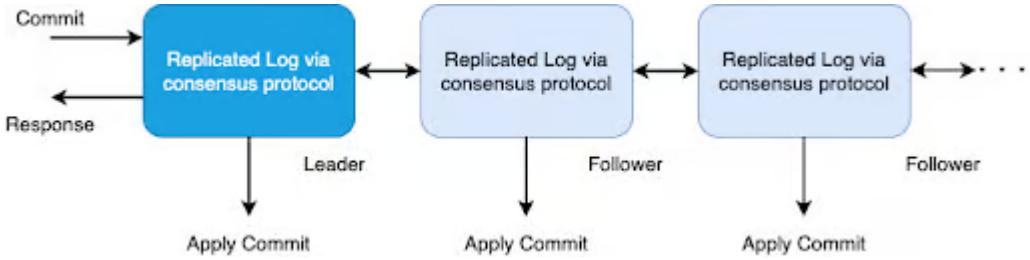
- Docstore has a concept of tables
- tables look like relational DB tables in that they are structured with rows, columns, and values.
- there are no restrictions on how tables are modeled in Docstore
- and with the use of user defined types, docstore can store nested records as rows.
- this is useful, for eg. when data has a document-like structure where entire hierarchy is loaded at once.
- Docstore also provides support for “Associations”, which allows for representation of one-to-many and many-to-many relationships.

- each table can have one or more materialized views.
- A materialized view is a view that allows the data to be partitioned differently as compared to the main table, by using different columns.
- Adding a materialized view partitioned by a non-primary key column makes it efficient to look up data by that column and allows different query access patterns.
- Every table must have a primary key and that primary key can be composed of one or more columns.
- primary key identifies the row in the table and enforces a unique constraint.
- Internally, the primary key and partition key columns are stored as a byte array and the value is derived by doing the order-preserving encoding of the key column values.
- Docstore stores rows in sorted order by primary key values.
- This combined with composite partition keys enables sophisticated query patterns, including grabbing all rows with a given partition key or using the remainder of the primary key to narrow the relevant rows for a particular query.





Replicated state machine

- To ensure consistency, each partition runs the raft consensus protocol. There is a single leader and multiple followers.
- 

- All writes are initiated by the leader.
- The consensus protocol implemented is used to keep the replicated log consistent across the nodes in the partition.
- this guarantees that all nodes in a partition contains same writes in same order ensuring serializability.
- state machine running on each node only proceeds in committing write if and only if consensus is reached.

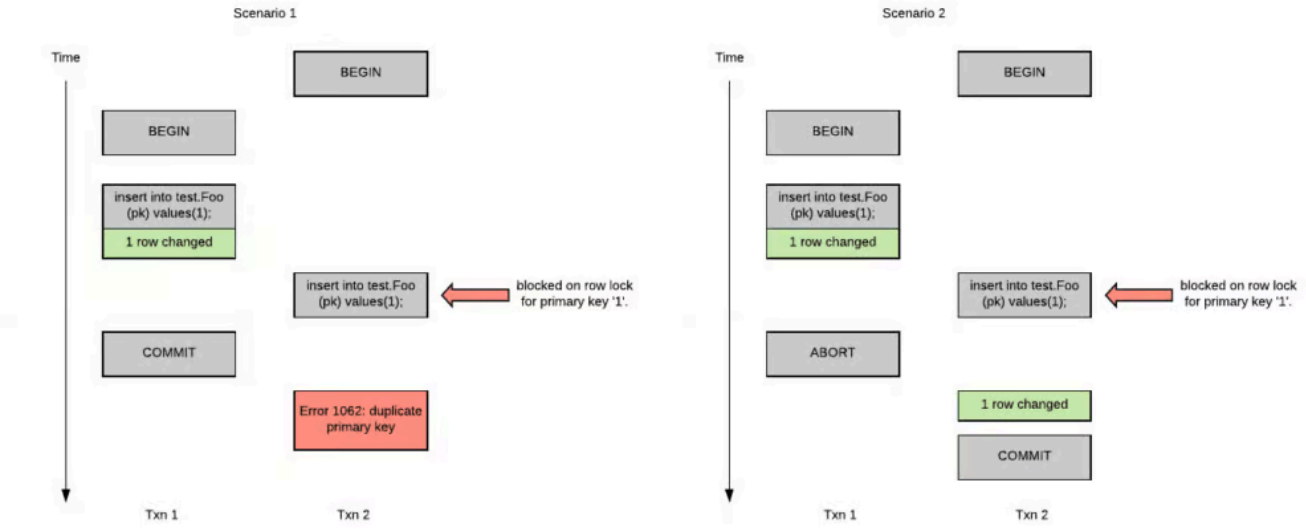
Consistency Model:

- Docstore provides a strict serializability consistency model on a partition level.
- This provides a nice property to the user where they can imagine that transactions are getting executed sequentially.
- This ensures that a read operation always returns the result from the most recent write.
- From a CAP theorem terminology, Docstore favors consistency over availability and therefore is a CP system.

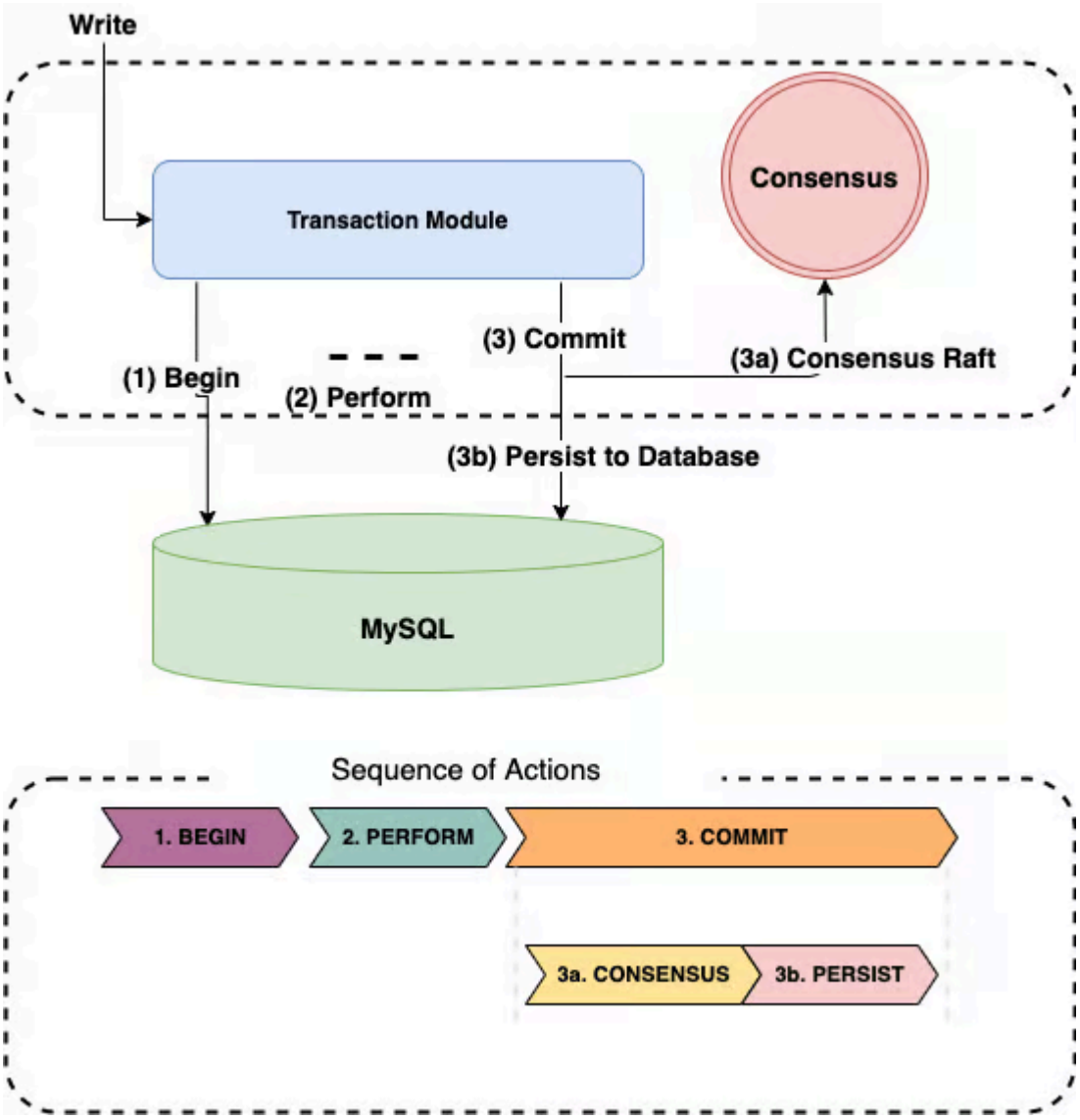
Transactions:

- Docstore uses MySQL as the underlying database engine.
- The unit of replication in the Replicated State Machine is a MySQL transaction.
- All the operations are executed within the context of a MySQL transaction to guarantee ACID semantics.
- These transactions are then replicated across the nodes using the raft consensus protocol.
- We depend on MySQL for concurrency control.
- It is imp to realize that MySQL relies on raw locks for concurrency control of write operations (insert, update, deletes)
- Thus, the concurrent updates to the same rows are effectively serialized by MySQL, and by the time the control flow reaches the client issuing the commit, all the locking has been

taken care of.



- A raft based replicated state machine implementation allows for exposing MySQL's transaction to clients in a highly-available way, i.e. all the replicas coordinate with each other to apply the transactions so that automatic failover among replicas is possible while the ACID property of transactions are still maintained even if failover happens.
- Note that because we rely on exposing MySQL's transactions to clients, all of the benefits and the constraints of MySQL transactions are inherited.


Write
Transaction Module
Consensus
(3) Commit
(3a) Consensus Raft
(1) Begin
(2) Perform
(3b) Persist to Database
MySQL

Sequence of Actions
1. BEGIN
2. PERFORM
3. COMMIT
3a. CONSENSUS
3b. PERSIST