In a multi-tenant MongoDB sharded cluster that supports both live application traffic and BI reporting workloads, it's essential to follow best practices to ensure data consistency, optimal performance, and scalability. This section focuses on detailed best practices for **mapping relational schemas to MongoDB**, **schema sampling**, **schema management changes**, and **performance improvements** for **multi-tenant sharded clusters**.

**1. Mapping Relational Schemas to MongoDB**

Mapping relational databases (SQL) to MongoDB requires careful planning to maintain data integrity while taking advantage of MongoDB’s flexible schema. Here are the best practices to follow:

* **Denormalize Where Appropriate**: In MongoDB, unlike relational databases, data is typically stored in a denormalized fashion. When transitioning from a relational schema to MongoDB, aim to store related data in the same document where possible (e.g., embedding).
  + **Use Embedding for One-to-Few Relationships**: For relationships such as a user and their address, embedding the address data inside the user document helps avoid costly joins.
  + **Use Referencing for One-to-Many Relationships**: For large datasets such as orders linked to customers, store a reference (customer ID) in the order documents rather than embedding the entire customer record.
* **Design for Query Patterns**: While designing the schema, prioritize frequent query access patterns and optimize for read/write operations.
  + Analyze the most common SQL queries and ensure that MongoDB’s schema is built around the necessary indexes.
  + Avoid deep nesting in documents as it can negatively impact read/write performance.
* **Handle Joins with Aggregation**: MongoDB doesn’t support traditional SQL joins. However, for certain cases, use the $lookup stage in MongoDB’s Aggregation Pipeline to achieve join-like behavior, especially for BI reporting.

**2. Schema Sampling Best Practices**

BI workloads often involve querying vast datasets. However, querying the full dataset every time can slow down performance and affect live applications. Schema sampling can help improve performance in such scenarios.

* **Leverage MongoDB BI Connector’s Sampling Options**:
  + MongoDB BI Connector supports querying a representative sample of data instead of the full dataset. Use LIMIT or sampling functions within your BI tools to fetch a subset of data.
  + Ensure that sampled data is relevant to the query requirements by predefining sampling criteria. This is especially useful for exploratory analysis or testing BI reports.
* **Create Materialized Views for Common Queries**: If a BI tool frequently queries certain datasets, consider creating a materialized view. This view can be regularly updated with a subset of the most commonly accessed data, significantly improving response times for large BI queries.
* **Run Complex Queries on Secondary Nodes**: Configure your BI connectors to prefer reading from secondary nodes. This will reduce the load on the primary nodes responsible for real-time application traffic. Be sure that replication lag is minimal to maintain data consistency.
* **Maintain a Backup Reporting Cluster**: For larger multi-tenant environments, it might be beneficial to deploy a secondary reporting cluster with delayed replication. This cluster can handle intensive BI queries without impacting the live system.

**3. Schema Management Changes Best Practices**

Schema changes in MongoDB can be more flexible compared to relational databases, but certain best practices must be followed to maintain data integrity and performance, especially in multi-tenant, sharded environments.

* **Use Schema Versioning**: When evolving your schema, it's critical to track changes and maintain backward compatibility, especially in multi-tenant environments. Add a version field in your documents so that applications can process documents according to their schema version.
* **Perform Rolling Schema Updates**: In production systems, avoid making immediate global schema changes. Instead, roll out schema changes incrementally across tenants to ensure that each tenant’s application remains operational without downtime.
* **Implement Validation Rules**: Use MongoDB's built-in schema validation to enforce certain structures and rules on collections. This prevents improper data from being inserted and helps ensure consistency during schema updates.
* **Index Management for Performance**: Reindex collections when schema changes involve key fields used in frequent queries. MongoDB’s sharded architecture performs optimally when queries are routed efficiently using the correct indexes. Use the explain() method to understand query performance and adjust indexes accordingly.

**4. SQL Functionality Support**

MongoDB's BI Connector translates SQL queries into MongoDB queries, but some SQL functions may not behave identically in MongoDB. The following guidelines can help optimize SQL function usage for BI reporting:

* **Support for SQL Aggregates**: The BI Connector supports SQL aggregate functions such as COUNT(), SUM(), AVG(), etc. Ensure that MongoDB indexes are created on the relevant fields to optimize these aggregate queries.
* **Date and Time Functions**: Functions such as DATE(), NOW(), YEAR(), and MONTH() are supported by the BI Connector. However, use MongoDB's date manipulation capabilities (e.g., $dateToString) in the aggregation pipeline for more complex date calculations.
* **Joins and Unions**: MongoDB supports SQL-like joins using $lookup in the aggregation pipeline. However, this can be resource-intensive. For multi-tenant systems, avoid frequent use of joins across different shards, as it can result in significant performance overhead.

**5. Performance Improvements for Multi-Tenant Sharded Clusters**

In a multi-tenant environment where data is sharded, managing the performance of both live applications and BI workloads requires the following considerations:

* **Use Zone-Aware Sharding**: Zone sharding allows you to place tenant data in specific shards based on geographical or usage-based criteria. For example, tenants with high BI reporting demands can have their data sharded in specific nodes optimized for heavy read operations.
* **Optimize Sharding Key**: The choice of sharding key is critical for multi-tenant performance. Choose a sharding key that ensures even data distribution across all shards and avoids performance bottlenecks. A tenant ID or a range-based key can help balance the load across shards.
* **Partition BI Workloads**: Design your architecture so that BI workloads are executed on a separate set of replica nodes. This prevents heavy BI queries from interfering with the live transactional workload.
* **Minimize Cross-Shard Queries**: BI queries that span multiple shards can be expensive, particularly in multi-tenant clusters. To avoid this, ensure that your data is sharded in such a way that most queries can be isolated to a single shard.
* **Monitor Shard Utilization**: Use MongoDB's built-in monitoring tools to track the utilization of each shard. If one shard is handling significantly more data or queries than others, consider rebalancing the data to avoid performance issues.
* **Leverage Read Preferences**: Set read preferences to direct BI queries to secondary replicas. This prevents BI workloads from overloading primary nodes and ensures that real-time application data remains available without latency.
* **Tenant Isolation**: Use MongoDB’s sharding techniques to isolate tenant data into separate shards. This ensures that queries on one tenant’s data do not affect another tenant’s operations. Additionally, BI queries that involve multiple tenants should be carefully optimized to prevent overloading shared resources.

**Conclusion**

This set of best practices will help you optimize your MongoDB multi-tenant, sharded cluster for both live application operations and BI workloads. By focusing on schema mapping, sampling, schema management, SQL support, and performance optimization, you can maintain high availability, improve data access performance, and ensure that BI queries do not disrupt live application traffic. Regular performance tuning, indexing strategies, and monitoring should be implemented to keep the system running at peak performance.

**Limitations of MongoDB BI Connector**

**1. SQL Functionality Support**

* **Partial SQL Support**: The BI Connector provides a subset of SQL features. Complex SQL queries, particularly those involving multiple joins or subqueries, might not be fully supported or may require rewriting to align with MongoDB’s aggregation capabilities.
* **Performance with Joins**: The BI Connector supports simple joins through the $lookup aggregation stage, but complex joins, especially across large datasets, can be slow and resource-intensive.

**2. Performance Overhead**

* **Latency**: The BI Connector translates SQL queries into MongoDB queries, which adds overhead. This translation layer can result in increased latency, especially for complex queries or aggregations.
* **Resource Utilization**: Since BI operations can be resource-heavy, running these on primary nodes might impact the performance of operational workloads. It's crucial to route BI traffic to secondary nodes wherever possible.

**3. Data Freshness**

* **Replica Set Delays**: When querying secondary nodes to reduce load on primary nodes, there can be a replication lag depending on the replica set configuration. This lag means that the BI reports might not always reflect the most up-to-date data.

**4. Schema Constraints**

* **Denormalized Data**: MongoDB's schema-less, denormalized data model can be advantageous for flexibility and performance but poses challenges for traditional SQL-based reporting tools which expect normalized data schemas.
* **Document Depth and Complexity**: MongoDB documents can be deeply nested, and certain BI tools might struggle to efficiently handle this complexity without customized queries or additional processing layers.

**5. Aggregation Pipeline Limitations**

* **Memory Usage**: Aggregation queries that require large amounts of data processing might exceed the memory limit of the aggregation pipeline if not properly indexed or optimized.
* **Complexity in Aggregations**: Constructing equivalent SQL queries using MongoDB's aggregation framework can be complex and may require substantial adjustments to traditional SQL queries.