Databases

1. This behavior in Cassandra is likely due to eventual consistency, which is a fundamental characteristic of distributed databases like Cassandra. Here's an explanation of the issue and how to address it:

Likely reason:

1. Eventual consistency: Cassandra prioritizes availability and partition tolerance over strict consistency (as per the CAP theorem).

2. Replication lag: Data changes may not have propagated to all replicas yet.

3. Read consistency level: The consistency level what is chosen might not be strong enough to ensure all replicas are in sync.

How to avoid it:

1. Increase read consistency level:

Use a stronger consistency level for reads, such as QUORUM or ALL. This ensures that more replicas are consulted before returning results.

```cql

SELECT \* FROM your\_table WHERE condition = value CONSISTENCY QUORUM;

```

2. Use tunable consistency:

Implement tunable consistency in your application, allowing for stronger consistency when needed. For example, implement logic in your application to dynamically adjust consistency levels based on the operation's importance. As like, use QUORUM for critical reads and ONE for less important queries. This allows you to balance between performance and consistency as needed.

3. Implement read repairs:

Enable read repair to fix inconsistencies during read operations.

```cql

ALTER TABLE your\_table WITH read\_repair\_chance = 1.0;

```

4. Use lightweight transactions:

For critical operations, use lightweight transactions (LWTs) to ensure linearizable consistency.

```cql

INSERT INTO your\_table (id, value) VALUES (1, 'new\_value') IF NOT EXISTS;

```

5. Implement proper delete operations:

Use tombstones and ensure proper TTL (Time To Live) settings for deleted data.

6. Monitor and tune replication:

Ensure your replication factor and strategy are appropriate for your use case.

7. Consider using SERIAL or LOCAL\_SERIAL consistency:

These provide linearizable reads and writes within the local datacenter.

```cql

SELECT \* FROM your\_table WHERE condition = value CONSISTENCY LOCAL\_SERIAL;

```

8. Implement application-level conflict resolution:

Design your application to handle potential conflicts and merge divergent data when necessary.

By implementing these strategies, you can minimize the occurrence of inconsistent query results and better manage the trade-offs between consistency, availability, and partition tolerance in your Cassandra cluster.

1. To shard the collection sanfrancisco.company\_name based on \_id in MongoDB, given the situation you've described with replicaset\_1, you'll need to follow these steps:

1. Enable sharding on the MongoDB cluster:

```

sh.enableSharding("sanfrancisco")

```

This command enables sharding for the "sanfrancisco" database. It's a prerequisite for sharding any collection within this database. It prepares the cluster to distribute data across multiple shards for collections in this database.

2. Choose a shard key and shard the collection:

```

sh.shardCollection("sanfrancisco.company\_name", { "\_id": 1 })

```

This command shards the "company\_name" collection using "\_id" as the shard key. The shard key determines how MongoDB distributes the documents across shards. Using "\_id" as the shard key ensures an even distribution if "\_id" values are evenly distributed. However, be aware that this may lead to monotonically increasing writes on a single shard if "\_id" values are always increasing.

3. Add the new replicaset as a shard:

```

sh.addShard("replicaset\_2/hostname1:port,hostname2:port,hostname3:port")

```

This adds the new replicaset (replicaset\_2) to the sharded cluster. Replace "hostname1:port,hostname2:port,hostname3:port" with the actual hostnames and ports of the members in replicaset\_2. This step makes the new replicaset available to store sharded data.

4. Monitor the sharding process:

```

sh.status()

```

This command provides a detailed report on the sharding status of your cluster. It shows information about databases, collections, chunks, and their distribution across shards. Run this periodically to check the progress of sharding and ensure data is being distributed as expected.

5. Pre-split the data (optional):

```

for (var i = 1; i <= 10; i++) {

sh.splitAt("sanfrancisco.company\_name", { \_id: i \* 1000000 })

}

```

This script creates split points in the collection, which can help distribute data more evenly across shards from the start. Adjust the number of splits and the values based on your data distribution. This step is particularly useful for large existing collections to prevent all data from initially being on a single shard.

6. Enable balancing:

```

sh.setBalancerState(true)

```

The balancer is responsible for evenly distributing chunks of data across shards. It's usually enabled by default, but this command ensures it's running. The balancer will move chunks between shards to maintain an even distribution.

7. Verify the sharding:

```

db.company\_name.getShardDistribution()

```

This command shows how the data in the "company\_name" collection is distributed across the shards. It provides information on the number of chunks, their size, and their distribution, helping you verify that sharding is working as expected.