MongoDB :

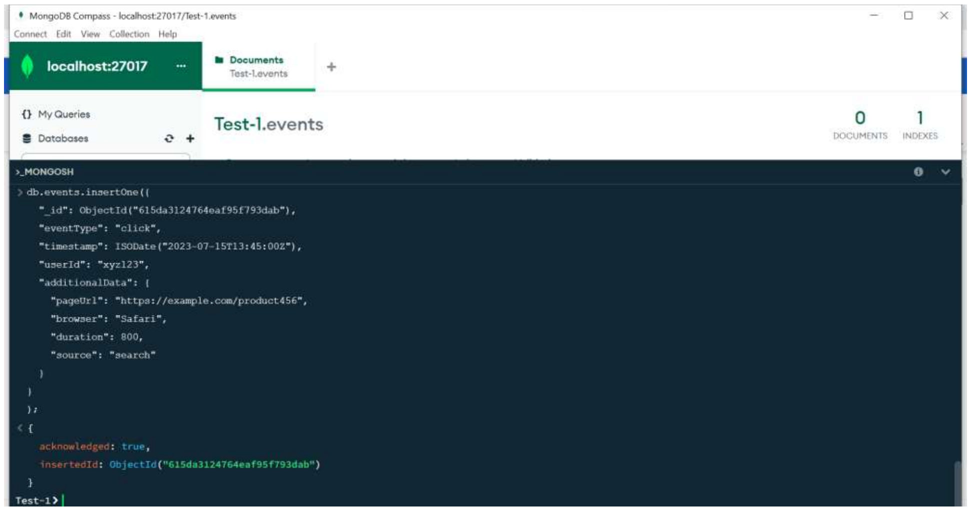
Database Schema Design:

- Create a collection to store event data, such as "events".
- Each document in the "events" collection represents an individual event and contains the necessary fields to describe the event, such as event type, timestamp, user ID, and any additional relevant data.
- Consider indexing the timestamp field for faster retrieval of events within specific time ranges.
- Example:

```
{
 "_id": ObjectId("615da3124764eaf95f793da8"),

  "eventType": "click",

  "timestamp": ISODate("2023-07-15T10:30:00Z"),

  "userId": "abc123",

  "additionalData": {

                          "pageUrl": "https://example.com",

                                 "browser": "Firefox",

                                 "duration": 1200,

                                 "source": "web-surfing"

  }

}
```

1) **Implement MongoDB queries for the following operations:**
    a) Insert a new event into the database.

```
db.events.insertOne({
 "_id": ObjectId("615da3124764eaf95f793dab"),
 "eventType": "click",
 "timestamp": ISODate("2023-07-15T13:45:00Z"),
"userId": "xyz123",
"additionalData": {
          "pageUrl": "https://example.com/product456",
           "browser": "Safari",
          "duration": 800,
                "source": "search"
    }
    });
```



MongoDB Compass - localhost:27017/Test-1.events

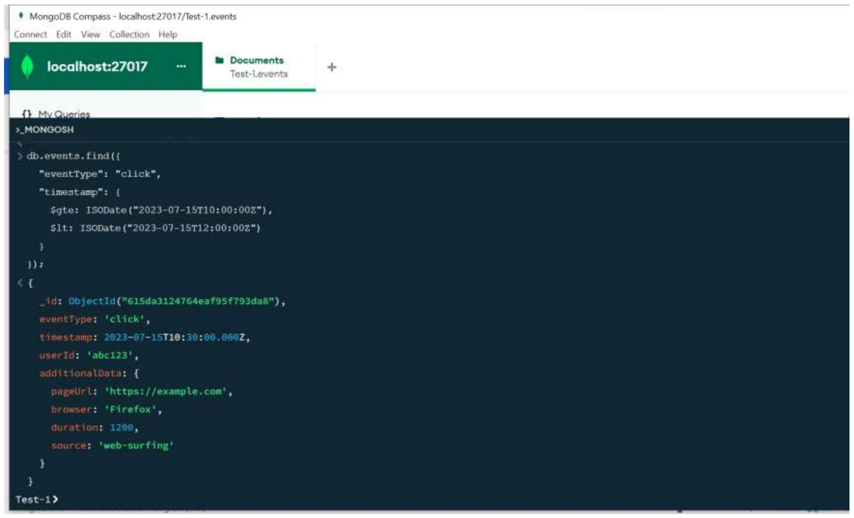
Test-1.events

0 DOCUMENTS 1 INDEXES

```
>_MONGOSH
> db.events.insertOne({
    "_id": ObjectId("615da3124764eaf95f793dab"),
    "eventType": "click",
    "timestamp": ISODate("2023-07-15T13:45:00Z"),
    "userId": "xyz123",
    "additionalData": {
      "pageUrl": "https://example.com/product456",
      "browser": "Safari",
      "duration": 800,
      "source": "search"
    }
  }
  );
< {
    acknowledged: true,
    insertedId: ObjectId("615da3124764eaf95f793dab")
  }
Test-1>
```

b) Retrieve events within a specific time range, filtered by event type.

```
db.events.find({
                 "eventType":  "click",
                "timestamp": {
                         $gte: ISODate("2023-07-15T10:00:00Z"),
                         $lt: ISODate("2023-07-15T12:00:00Z")
                 }
        });
```

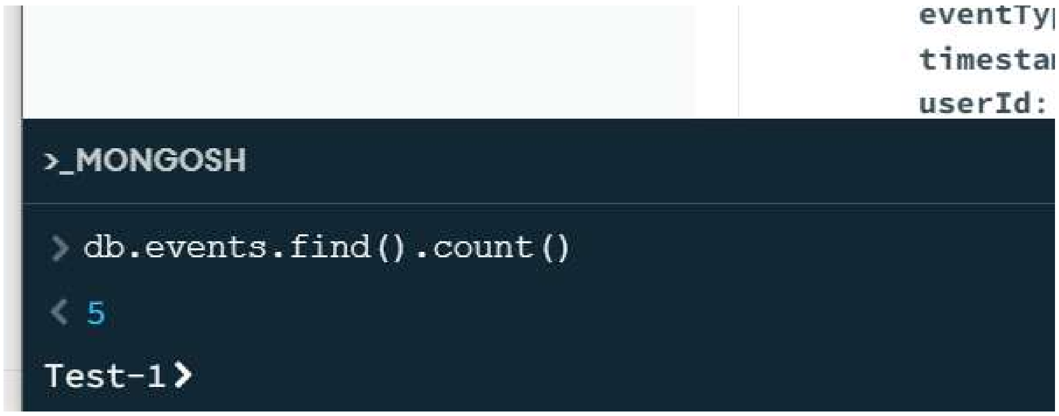


MongoDB Compass - localhost:27017/Test-1.events

```
>_MONGOSH
> db.events.find({
    "eventType": "click",
    "timestamp": {
      $gte: ISODate("2023-07-15T10:00:00Z"),
      $lt: ISODate("2023-07-15T12:00:00Z")
    }
  });
< {
    _id: ObjectId("615da3124764eaf95f793da8"),
    eventType: 'click',
    timestamp: 2023-07-15T10:30:00.000Z,
    userId: 'abc123',
    additionalData: {
      pageUrl: 'https://example.com',
      browser: 'Firefox',
      duration: 1200,
      source: 'web-surfing'
    }
  }
Test-1>
```

c) Perform aggregation queries to calculate metrics like total events, events per hour, and user activity.

1. Total events - db.events.find().count();

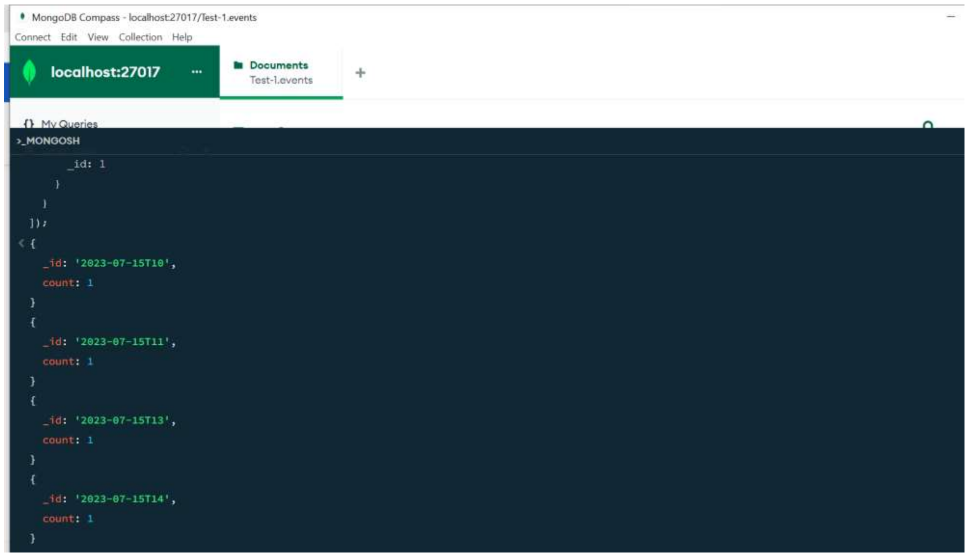


```
> db.events.find().count()
< 5
Test-1>
```

2. Events per hour

db.events.aggregate([

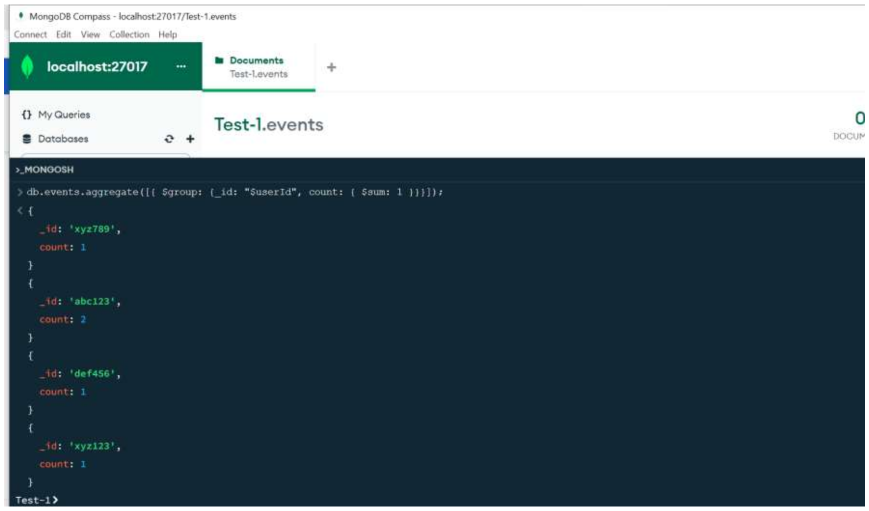
```
db.events.aggregate([
        { $group: { _id: { $substr: ["$timestamp", 0, 13]},
        count: { $sum: 1 }
        }},
        {$sort: { _id: 1}}]);
```


MongoDB Compass - localhost:27017/Test-1.events
Connect Edit View Collection Help
localhost:27017
Documents
Test-1.events
My Queries
>_MONGOSH

```
        _id: 1
      }
    }
  ]);
< {
    _id: '2023-07-15T10',
    count: 1
  }
  {
    _id: '2023-07-15T11',
    count: 1
  }
  {
    _id: '2023-07-15T13',
    count: 1
  }
  {
    _id: '2023-07-15T14',
    count: 1
  }
```

3. User activity (events count per user)

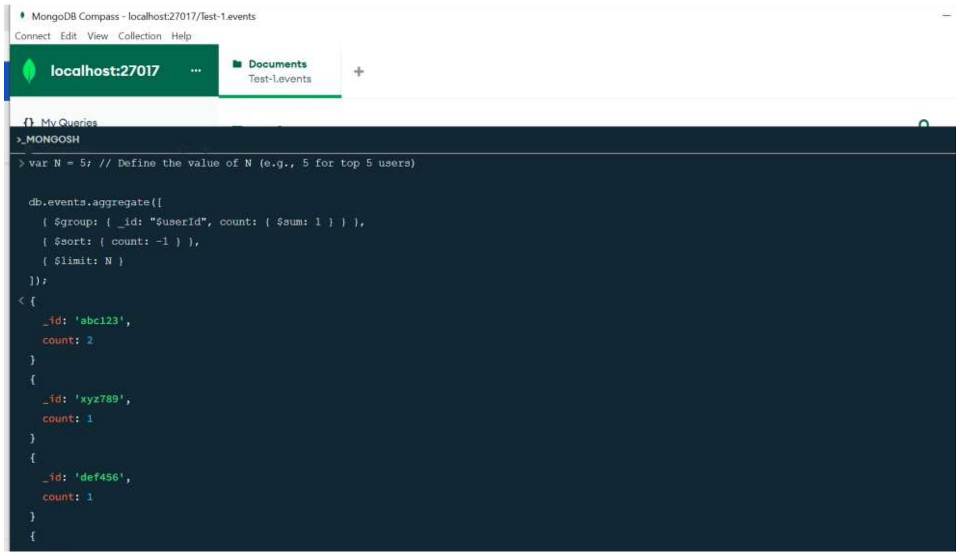
```
db.events.aggregate([
        { $group: {_id: "$userId", count: { $sum: 1 }}}
]);
```



```
> db.events.aggregate([{ $group: {_id: "$userId", count: { $sum: 1 }}}]);
< {
    _id: 'xyz789',
    count: 1
  }
  {
    _id: 'abc123',
    count: 2
  }
  {
    _id: 'def456',
    count: 1
  }
  {
    _id: 'xyz123',
    count: 1
  }
Test-1>
```

d) Retrieve the top N users based on the number of events generated.

```
var N = 5; // Define the value of N (e.g., 5 for top 5 users)
db.events.aggregate([
        { $group: { _id: "$userId", count: { $sum: 1 } } },
        { $sort: { count: -1 } },
        { $limit: N }]);
```


MongoDB Compass - localhost:27017/Test-1.events
Connect Edit View Collection Help
localhost:27017
Documents
Test-1.events
My Queries
>_MONGOSH

```
> var N = 5; // Define the value of N (e.g., 5 for top 5 users)

  db.events.aggregate([
    { $group: { _id: "$userId", count: { $sum: 1 } } },
    { $sort: { count: -1 } },
    { $limit: N }
  ]);
< {
    _id: 'abc123',
    count: 2
  }
  {
    _id: 'xyz789',
    count: 1
  }
  {
    _id: 'def456',
    count: 1
  }
  {
```

2) **Discuss the benefits of using MongoDB's flexible schema for handling diverse event data.**

Benefits are as follows:

1) MongoDB's flexible schema allows you to store event data with varying structures and fields without predefined schemas. This adaptability enables you to handle diverse event data without the need for strict data models or schema migrations. You can easily accommodate new event types or add new fields to existing events without disrupting the application or database operations.
2) MongoDB's flexible schema supports horizontal scaling and sharding, allowing you to handle large volumes of event data efficiently. As your event data grows, you can distribute it across multiple servers or shards, ensuring high performance and scalability for real-time analytics.
3) With MongoDB's flexible schema, developers can quickly iterate and experiment with new event data structures during the development process. It eliminates the need for upfront schema design and enables rapid prototyping and iterative development. Developers can focus on analyzing and deriving insights from event data rather than spending time on schema management.
4) MongoDB's flexible schema simplifies the integration of event data from various sources. You can ingest and store events with different structures and fields, making it easier to work with data coming from multiple systems or APIs.
5) Query Flexibility: MongoDB's flexible schema allows you to query event data based on different criteria and fields, even if the structure varies between events. You can perform ad-hoc queries and aggregations without being restricted by a fixed schema. This flexibility is especially useful for real-time analytics, where data exploration and analysis requirements can evolve rapidly.
6) MongoDB's flexible schema reduces the maintenance overhead associated with schema migrations. You can modify or extend event data structures without requiring downtime or complex migration scripts.

3) Explain how MongoDB's replica sets and automatic failover can ensure high availability and data durability in the analytics system.

MongoDB's replica sets and automatic failover provide high availability and data durability in the analytics system by ensuring that data is replicated across multiple servers and automatically maintaining a functioning primary replica member.

1. Replica Sets: MongoDB uses a concept called replica sets to replicate and distribute data across multiple servers. A replica set consists of a primary replica member and one or more secondary replica members.
2. Primary Replica Member: The primary replica member is responsible for handling all write operations and becomes the main point of contact for client applications. It receives write requests and updates the data in its local copy of the database.
3. Secondary Replica Members: Secondary replica members replicate the data from the primary replica member and act as backups. They receive a copy of the primary's data and continuously apply the write operations to stay in sync. Secondary members can also serve read operations, providing additional read capacity to the system.
4. Automatic Failover: In the event of a failure or unavailability of the primary replica member, MongoDB's automatic failover mechanism comes into play. The replica set monitors the health of each member through heartbeats and elects a new primary if the current primary becomes unavailable.

Example Scenario: Let's say you have a MongoDB replica set for your analytics system with three members: Member A as the primary replica member and Members B and C as secondary replica members.

1. High Availability: If Member A, the primary, experiences a hardware failure or network issue, the replica set automatically detects the unavailability of the primary and triggers a failover process.
2. Automatic Failover: During the failover process, Members B and C communicate and elect a new primary among themselves. Once the new primary is elected, it takes over the write operations, ensuring uninterrupted service for client applications.
3. Data Durability: MongoDB ensures data durability by writing data to the primary replica member's disk and then replicating it to the secondary replica members. This replication ensures that even if the primary replica member fails, the data remains accessible from the secondary members.

As a result of MongoDB's replica sets and automatic failover, your analytics system achieves high availability because it can continue to function even if the primary replica member becomes unavailable. Additionally, data durability is ensured by replicating data across multiple servers, reducing the risk of data loss.