CLASS- 8

Atomicity, Consistency, Replication, and Sharding

ACID Properties: Atomicity and Consistency

These two terms are fundamental to database transaction management and ensure data integrity.

Atomicity

- **Definition:** A transaction is an atomic operation. This means it either completes entirely, or it's rolled back to its original state. There's no partial completion.
- **Example:** Transferring money from one account to another. If the debit from one account is successful but the credit to the other fails, the transaction is rolled back. When you checkout on an online shopping website, the entire process of deducting the total amount from your account and updating inventory levels must happen as a single atomic operation. If your payment is processed but the inventory is not updated, it would result in an inconsistent state (you've paid but the item is still available for purchase by others).

Consistency

- **Definition:** A database must always maintain data integrity. Any transaction must move the database from one consistent state to another.
- **Example:** Maintaining account balances. A transaction that transfers money must ensure that the total amount of money in the system remains constant.
  In the online shopping scenario, the total amount in your account plus the total value of items in your cart should always be equal to a constant value. After checkout, the total amount in your account should decrease by the total value of items purchased, and the inventory levels for those items should decrease accordingly.

Replication and Sharding: Data Management Strategies

These techniques are used to improve system performance, availability, and scalability.

Replication

- **Definition:** Creating multiple copies of data across different servers or data centers.
- **Types:**
  - Synchronous Replication: Writes are committed only after all replicas are updated.
  - Asynchronous Replication: Writes are committed immediately, and replicas are updated later.
- **Benefits:** Improved read performance, increased availability, and disaster recovery.
- **Challenges:** Data consistency and synchronization overhead.

- **Example:** To improve performance and availability, an online shopping website might replicate product catalogs, user data, and order information across multiple servers in different geographical locations. This ensures that customers can access the website even if one server is down.

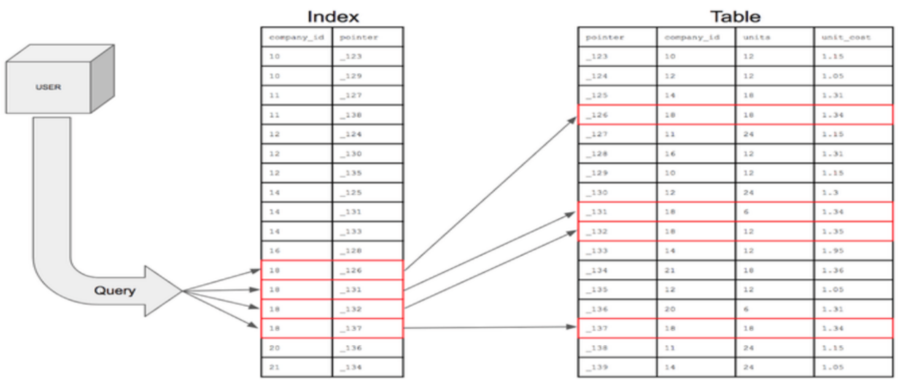
Sharding

- **Definition:** Dividing a large dataset into smaller subsets (shards) and storing each shard on a different server.
- **Benefits:** Improved write performance, scalability, and handling large datasets.
- **Challenges:** Data distribution, query optimization, and managing data consistency across shards.
- **Example:** To improve performance and availability, an online shopping website might replicate product catalogs, user data, and order information across multiple servers in different geographical locations. This ensures that customers can access the website even if one server is down.

  How They Work Together

- **Atomicity and Consistency** ensure that the shopping cart checkout process is reliable and accurate.
- **Replication** improves the availability of the website and the speed at which customers can access product information.
- **Sharding** helps the website handle a large number of products efficiently and without performance degradation.

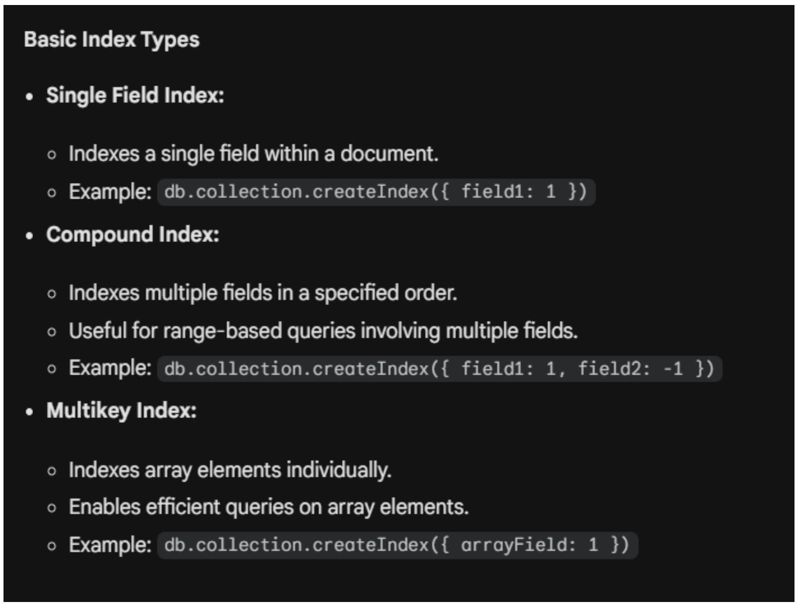
Indexs:



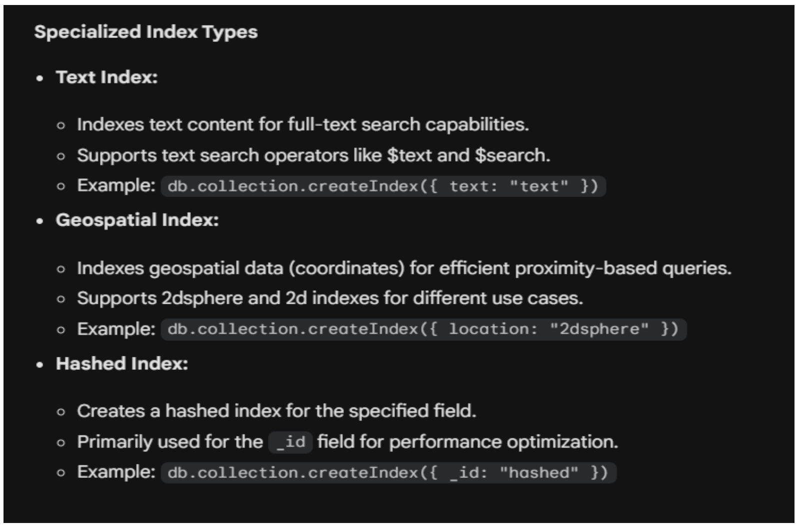
company_id	pointer
10	_123
10	_129
11	_127
11	_138
12	_124
12	_130
12	_135
14	_125
14	_131
14	_133
16	_128
18	_126
18	_131
18	_132
18	_137
20	_136
21	_134

pointer	company_id	units	unit_cost
_123	10	12	1.15
_124	12	12	1.05
_125	14	18	1.31
_126	18	18	1.34
_127	11	24	1.15
_128	16	12	1.31
_129	10	12	1.15
_130	12	24	1.3
_131	18	6	1.34
_132	18	12	1.35
_133	14	12	1.95
_134	21	18	1.36
_135	12	12	1.05
_136	20	6	1.31
_137	18	18	1.34
_138	11	24	1.15
_139	14	24	1.05

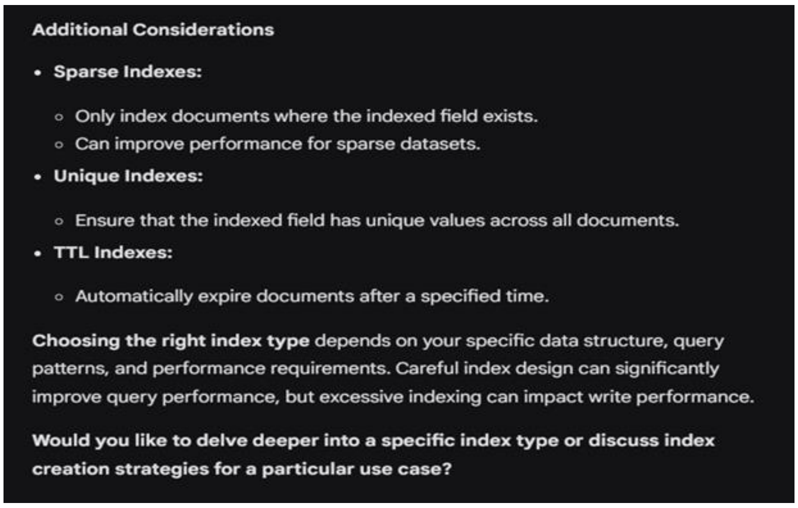
<u>Types of Indexes:</u>


Basic Index Types

- **Single Field Index:**
  - Indexes a single field within a document.
  - Example: `db.collection.createIndex({ field1: 1 })`
- **Compound Index:**
  - Indexes multiple fields in a specified order.
  - Useful for range-based queries involving multiple fields.
  - Example: `db.collection.createIndex({ field1: 1, field2: -1 })`
- **Multikey Index:**
  - Indexes array elements individually.
  - Enables efficient queries on array elements.
  - Example: `db.collection.createIndex({ arrayField: 1 })`


Specialized Index Types

- **Text Index:**
  - Indexes text content for full-text search capabilities.
  - Supports text search operators like $text and $search.
  - Example: `db.collection.createIndex({ text: "text" })`
- **Geospatial Index:**
  - Indexes geospatial data (coordinates) for efficient proximity-based queries.
  - Supports 2dsphere and 2d indexes for different use cases.
  - Example: `db.collection.createIndex({ location: "2dsphere" })`
- **Hashed Index:**
  - Creates a hashed index for the specified field.
  - Primarily used for the `_id` field for performance optimization.
  - Example: `db.collection.createIndex({ _id: "hashed" })`


Additional Considerations

- **Sparse Indexes:**
  - Only index documents where the indexed field exists.
  - Can improve performance for sparse datasets.
- **Unique Indexes:**
  - Ensure that the indexed field has unique values across all documents.
- **TTL Indexes:**
  - Automatically expire documents after a specified time.

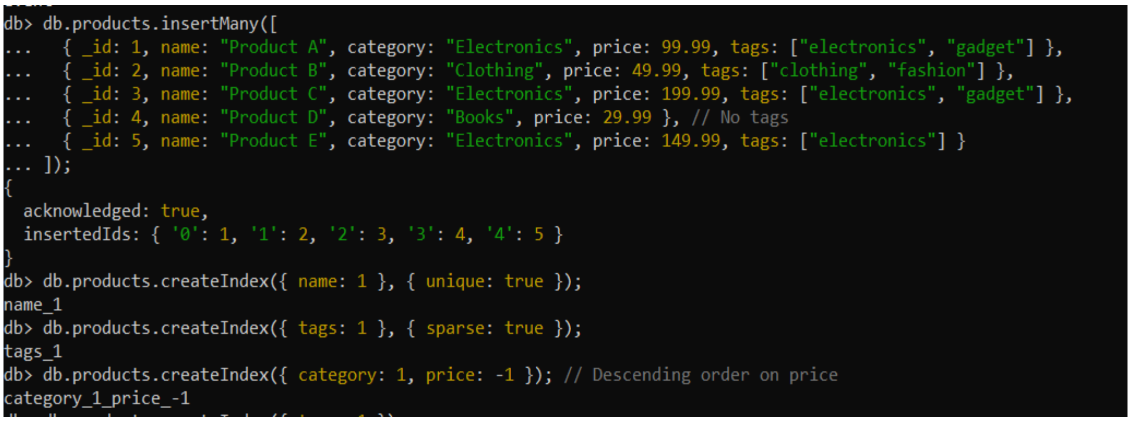
Choosing the right index type depends on your specific data structure, query patterns, and performance requirements. Careful index design can significantly improve query performance, but excessive indexing can impact write performance.

Would you like to delve deeper into a specific index type or discuss index creation strategies for a particular use case?

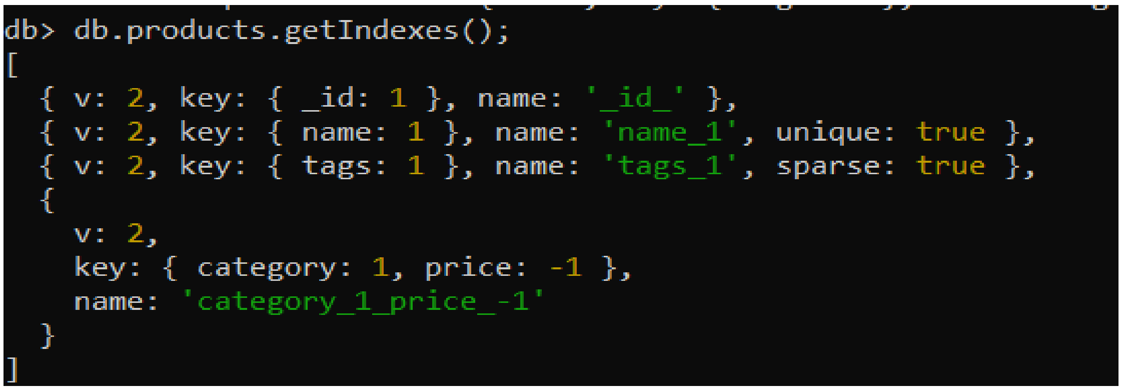
Demonstrating Index Creation and Output in MongoDB

Understanding the Data

Let's create a sample collection named products with some documents:



```
db> db.products.insertMany([
...   { _id: 1, name: "Product A", category: "Electronics", price: 99.99, tags: ["electronics", "gadget"] },
...   { _id: 2, name: "Product B", category: "Clothing", price: 49.99, tags: ["clothing", "fashion"] },
...   { _id: 3, name: "Product C", category: "Electronics", price: 199.99, tags: ["electronics", "gadget"] },
...   { _id: 4, name: "Product D", category: "Books", price: 29.99 }, // No tags
...   { _id: 5, name: "Product E", category: "Electronics", price: 149.99, tags: ["electronics"] }
... ]);
{
  acknowledged: true,
  insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }
}
db> db.products.createIndex({ name: 1 }, { unique: true });
name_1
db> db.products.createIndex({ tags: 1 }, { sparse: true });
tags_1
db> db.products.createIndex({ category: 1, price: -1 }); // Descending order on price
category_1_price_-1
```



```
db> db.products.getIndexes();
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { name: 1 }, name: 'name_1', unique: true },
  { v: 2, key: { tags: 1 }, name: 'tags_1', sparse: true },
  {
    v: 2,
    key: { category: 1, price: -1 },
    name: 'category_1_price_-1'
  }
]
```

Understanding the Requirements

Before we proceed, let's clarify the expected output for the reviews summary:

- **Product ID:** The unique identifier of the product.
- **Average Rating:** The average rating for the product.
- **Review Count:** The total number of reviews for the product.
- **Comments:** (Optional) A list of comments for the product (consider limitations for large datasets).

MongoDB Queries and Outputs

1. Finding Listings with Host Picture URL

```
db.listingsAndReviews.find(
  { "host.host_picture_url": { $exists: true, $ne: null } },
  { listing_url: 1, name: 1, address: 1, "host.host_picture_url": 1 }
)
```

Output :

```
[
 {
   "_id": ObjectId("647f8509c1274f937c1632f0"),
   "listing_url": "https://www.example.com/listings/123",
   "name": "Cozy Beachfront Apartment",
   "address": "123 Ocean View Blvd, Malibu, CA",
   "host": {
     "host_picture_url":
"https://www.example.com/profile_pics/john_smith.jpg"
   }
 },
 // ... other listings with host picture URLs
]
```

2. Displaying Reviews Summary

```
db.eCommerceCollection.aggregate([
 { $unwind: "$reviews" },
{  $group: {
        _id: "$product_id",
        average_rating: { $avg: "$reviews.rating" },
         review_count: { $sum: 1 },
 },{
 $project: {
      _id: 0,
product_id: "$_id",
 average_rating: 1,
 review_count: 1,
}}])
```

Output:

```
[

{

"product_id": 123,

"average_rating": 4.5,

"review_count": 2

},

{

"product_id": 456,

"average_rating": 3,

"review_count": 1

}, // ... other products

]
```

LISTTING:

We have two primary tasks:

1. **Find listings with host pictures:** Retrieve listings from the listingsAndReviews collection where the host has a picture URL.
2. **Display reviews summary:** Calculate summary statistics for reviews in the eCommerce collection.

MongoDB Queries

1. **Find Listings with Host Picture URL**

```
db.listingsAndReviews.find(

    { "host.host_picture_url": { $exists: true, $ne: null } },

    { listing_url: 1, name: 1, address: 1, "host.host_picture_url": 1 }

 )
```

Explanation:

- db.listingsAndReviews.find({}): Selects the listingsAndReviews collection.

- { "host.host_picture_url": { $exists: true, $ne: null } }: Filters documents where the host.host_picture_url field exists and is not null.
- { listing_url: 1, name: 1, address: 1, "host.host_picture_url": 1 }: Projects only the required fields.

2. Display Reviews Summary

```
db.eCommerce.aggregate([
 { $unwind: "$reviews" },
 {
  $group: {
   _id: "$product_id",
   averageRating: { $avg: "$reviews.rating" },
   reviewCount: { $sum: 1 },
   comments: { $push: "$reviews.comment" }
  } }, {
  $project: {
   _id: 0,
   productId: "$_id",
   averageRating: 1,
   reviewCount: 1,
   comments: 1
  } }])
```

Explanation:

- db.eCommerce.aggregate([]): Initiates an aggregation pipeline on the eCommerce collection.
- $unwind: "$reviews": Decomposes the reviews array into individual documents.
- $group: Groups documents by product_id and calculates average rating, review count, and a list of comments.
- $project: Reshapes the output, renaming _id to productId and excluding the original _id.