UNIT -III MongoDB Storage Engine

MongoDB Storage Engine: Data Storage Engine, Data File (Relevant for MMAPv1), Namespace (.ns File), Data File (Relevant for WiredTiger), Reads and Writes, How Data Is Written Using Journaling, GridFS – The MongoDB File System, The Rationale of GridFS, GridFSunder the Hood, Using GridFS, Indexing, Types of Indexes, Behaviors and Limitations

PYQ : (Previous Year Mumbai University Question)

Nov – 18

1. What is Data Storage engine? Which is the default storage engine in MongoDB? Also compare MMAP and Wired Tiger storage engines.
2. What is Journaling? Explain the importance of Journaling with the help of a neat diagram

Apr -19

1. What is Wired Tiger Storage Engine?
2. Write a short note on GridFS.
3. How read and write operations performed in MongoDB.
3. Discuss how data is written using Journaling.

Nov – 19

1. What is Data Storage Engine. Differentiate between MMAP and Wired Storage Engine

Nov – 22

1. Write a short note on Wired Storage Engine
2. Explain the concept of GridFS – The MongoDB File System

Dec – 23

1. Delineate the write operations performed using journaling .

b. Illustrate the working of following methods of GridFS: i) new_file() ii) get_version() iii) get_last_version() iv) delete() v) exists() and put()

Nov – 24

1. Describe the process of write operations performed using journaling .
2. Explain the Sparse and Geospatial Index in detail
3. Describe the "GridFS – MongoDB File System" in detail.

"MongoDB explained covers deep-dive concepts of what happens under the hood in MongoDB."

Data Storage Engine :

#### **Question 1 :**What is Data Storage engine? Which is the default storage engine in MongoDB? Also compare MMAP and Wired Tiger storage engines.

A **data storage engine** in MongoDB is like a special software that decides how data is saved and organized on your computer's hard drive. Think of it as a smart filing system that keeps track of all the information in your database.

When you put data into MongoDB, the storage engine makes sure it's saved properly so you can find it later. It also helps in quickly searching through the data and updating it when needed. The storage engine is responsible for:

1. **Storing Data:** It decides where and how to save data on the disk.
2. **Retrieving Data:** It helps quickly find the data when you ask for it.
3. **Updating Data:** It manages changes to the data, ensuring they are stored correctly.
4. **Ensuring Data Safety:** It makes sure your data is safe, even if there's a computer crash or power outage.

In simple terms, the storage engine is like a librarian in a library. The librarian knows where every book (data) is located, keeps them organized, and ensures they are safe and easily accessible whenever needed.

####**Types of Storage Engines in MongoDB**

1. **WiredTiger** (Default in MongoDB 3.2+)
2. **MMAPv1** (Deprecated)

1. WiredTiger Storage Engine

WiredTiger is the default storage engine in modern versions of MongoDB. It offers several key features:

- **Document-Level Concurrency Control:** Supports multiple concurrent read and write operations, making it highly efficient for multi-threaded applications.
- **Compression:** Data is compressed to save disk space, using algorithms like Snappy or zlib.
- **Journaling:** Ensures data integrity and durability by recording changes before applying them, making recovery possible in case of crashes.
- **Caching:** Utilizes an in-memory cache to speed up read and write operations.

Example Explanation

Imagine a library with many books. WiredTiger is like a super-efficient librarian who organizes books (data) in a way that they can be found and borrowed quickly. The librarian also keeps a small "cache" of popular books on a special shelf for quick access, compresses less popular books to save space, and writes down all borrowings and returns in a journal for record-keeping.

2. MMAPv1 Storage Engine (Deprecated)

MMAPv1 was the default storage engine in earlier MongoDB versions. It uses memory-mapped files for data storage and provides collection-level locking, which means it locks an entire collection during write operations.

Example Explanation

Think of MMAPv1 as an older librarian who locks the entire library whenever someone wants to borrow or return a book. This method can be slower, especially when many people need to borrow or return books at the same time.

Choosing a Storage Engine

- **WiredTiger:** Preferred for most modern applications due to its better performance, concurrency, and data compression features.
- **MMAPv1:** Suitable for legacy applications that still require it, though it's deprecated and not recommended for new projects.

Data File (Relevant for MMAPv1):

The **Data File** in MongoDB, specifically relevant for the MMAPv1 storage engine, represents the way data is stored on disk. MMAPv1 was MongoDB's original storage engine, and understanding how data files work in this context is essential for managing data persistence, efficiency, and performance.

### Key Concepts of Data Files in MMAPv1

1. **MMAPv1 Storage Engine**:
   - **Memory-Mapped Files**: MMAPv1 uses memory-mapped files, which map data files directly into the memory space of the MongoDB process. This allows MongoDB to access and manipulate data as if it were in memory, which can improve performance.
   - **Fixed-size Extents**: Data is stored in fixed-size extents, which are contiguous blocks of data. These extents grow as more data is added, but once an extent is allocated, it does not shrink.

2. **Data File Structure**:
   - **Namespace File (`.ns` file)**: Contains metadata about the collections and indexes.
   - **Data Files (`.0`, `.1`, etc.)**: These files store the actual data and indexes. The naming convention indicates the order of file creation, with `.0` being the first file, `.1` the second, and so on.

3. **Pre-allocation of Files**:
   - MongoDB pre-allocates data files to ensure that it has space to write data. This means that even if the database doesn't immediately need the space, MongoDB allocates it in advance to avoid delays when writing data.

4. **Journaling**:
   - MMAPv1 includes a journaling mechanism to provide durability. It records write operations in a separate journal file before applying them to the data files. In case of a crash, the journal can be replayed to bring the data files to a consistent state.

### Detailed Explanation

- ****Namespace File (`.ns` File)**:** This file contains information about all the collections and indexes in the database. It essentially maps collection names to the actual data files.

- ****Data Files (`.0`, `.1`, etc.)**:**
  - Each data file is divided into **extents**. Extents are contiguous blocks of space allocated for collections and indexes.
  - The first extent is usually smaller, and subsequent extents grow in size to accommodate more data.
  - As data grows, MongoDB allocates new data files. For instance, after filling `mydb.0`, MongoDB will create `mydb.1`, then `mydb.2`, and so on.

- ****Pre-allocation**:**
  - MongoDB pre-allocates files to reduce fragmentation and ensure that there is enough contiguous space for data. This can help avoid performance degradation that might occur if the database had to frequently extend data files.

- ****Journaling**:**
  - The journal file records changes in a write-ahead log. This ensures that even if a write operation fails, the database can recover to a consistent state by replaying the journal.

### Example Explanation

Let's say you have a MongoDB database named `mydb` using the MMAPv1 storage engine. Initially, MongoDB creates:

- A namespace file: `. mydb ns`
- The first data file: `mydb.0`

Suppose you start adding documents to a collection `users`. As you add more data:

1. The `mydb.0` file gets filled with data in fixed-size extents.
2. Once `mydb.0` is filled, MongoDB creates `mydb.1` to store additional data.
3. If there is a power failure or crash, the journal file ensures that any in-progress write operations can be safely recovered upon restart.

### Summary

The Data File concept in MMAPv1 involves the use of memory-mapped files and pre-allocated data files to store collections and indexes. It provides a mechanism for efficient data access and durability through journaling. While MMAPv1 has been replaced by more modern storage engines like WiredTiger, understanding its workings is useful for historical context and for those maintaining legacy systems.

Namespace :

- Within the data files you have data space divided into namespaces , where the namespace can correspond to either a collection or an index.
- The metadata of these namespaces are stored in the .ns file. If you check your data directory, you will find a file named [dbname].ns .
- The size of the .ns file that is used for storing the metadata is 16MB. This file can be thought of as a big hash table that is partitioned into small buckets, which are approximately 1KB in size.
- Each bucket stores metadata specific to a namespace

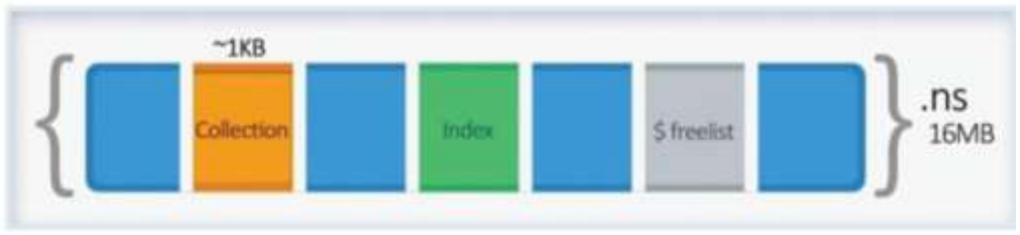


Figure 8-3. Namespace data structure

Collection Namespace :

the collection namespace bucket contains metadata such as

- Name of the collection
- A few statistics on the collection such as count, size, etc. (This is why whenever a count is issued against the collection it returns quick response.)
- Index details, so it can maintain links to each index created
- A deleted list

• A doubly linked list storing the extent details (it stores pointer to the first and the last extent)

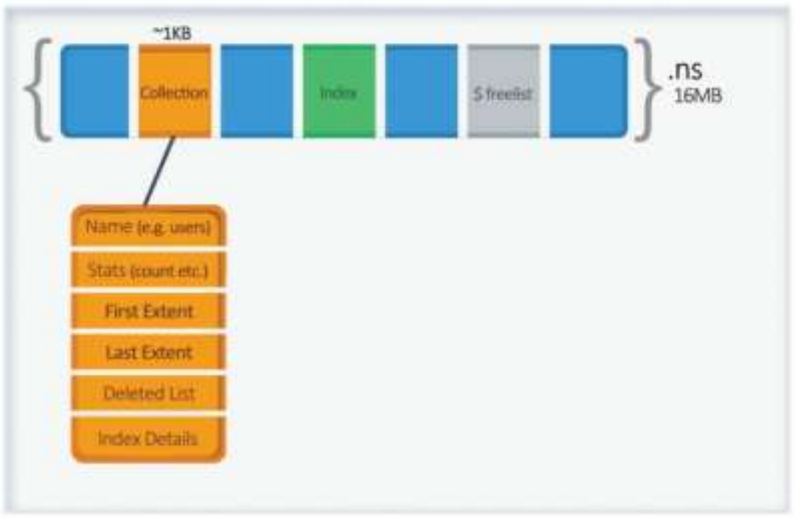


Figure 8-4. *Collection namespace details*

#### Extent :

- Extent refers to a group of data records within a data file, so a group of extents forms the complete data for a namespace.
- An extent uses disk locations to refer to the location on the disk where the data is actually residing.
- It consists of two parts: file number and offset. The file number specifies the data file it's pointing to (0, 1, etc.).
- Offset is the position within the file (how deep within the file you need to look for the data).
- The offset size is 4KB. Hence the offset's maximum value can be up to 2 31 -1, which is the maximum file size the data files can grow to (2048MB or 2 GB). As shown in Figure 8-5 ,
- an extent data structure consists of the following things:
    - Location on the disk, which is the file number it is pointing to.
    - Since an extent is stored as a doubly linked list element, it has a pointer to the next and the previous extent.
    - Once it has the file number it's referring to, the group of the data records within the file it's pointing to are further stored as doubly linked list. Hence it maintains a pointer to the first data record and the last data record of the data block it's pointing to, which are nothing but the offsets within the file (how deep within the file the data is stored).

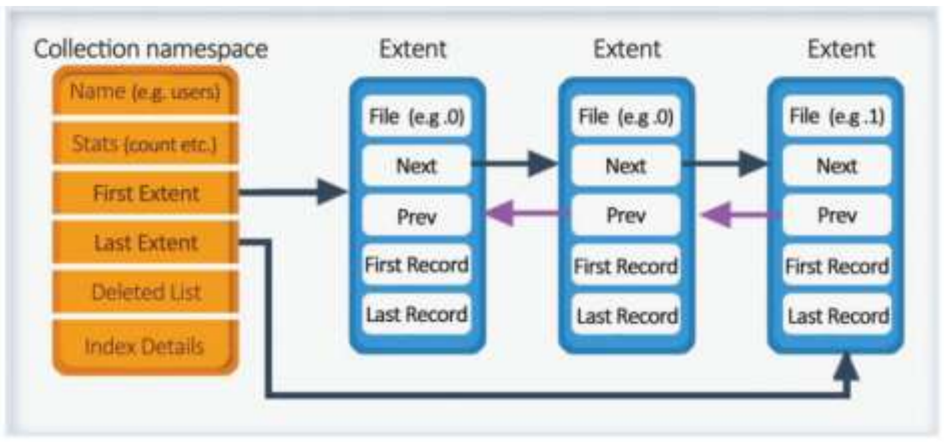


***Figure 8-5.** Extent*

#### Data Record :

The data structure consists of the following details:

• Since the data record structure is an element of the extent's doubly linked list, it stores information of the previous and the next record.

• It has length with headers.

• The data block.

- The data block can have either a BTree Bucket (in case of an index namespace) or a BSON object.
- The BSON object corresponds to the actual data for the collection.
- The size of the BSON object need not be same as the data block.
- Power of 2-sized allocation is used by default so that every document is stored in a space that contains the document plus extra space or padding. This design decision is useful to avoid movement of an object from one block to another whenever an update leads to a change in the object size.
- MongoDB supports multiple allocation strategies, which determine how to add padding to a document (Figure 8-6).
- As in-place updates are more efficient than relocations, all padding strategies trade extra space for increased efficiency and decreased fragmentation.
- Different workloads are supported by different strategies. For instance, exact fit allocation is ideal for collections with insert-only workloads where the size is fixed and never varies,

whereas power of 2 allocations are efficient for insert/update/delete workloads.

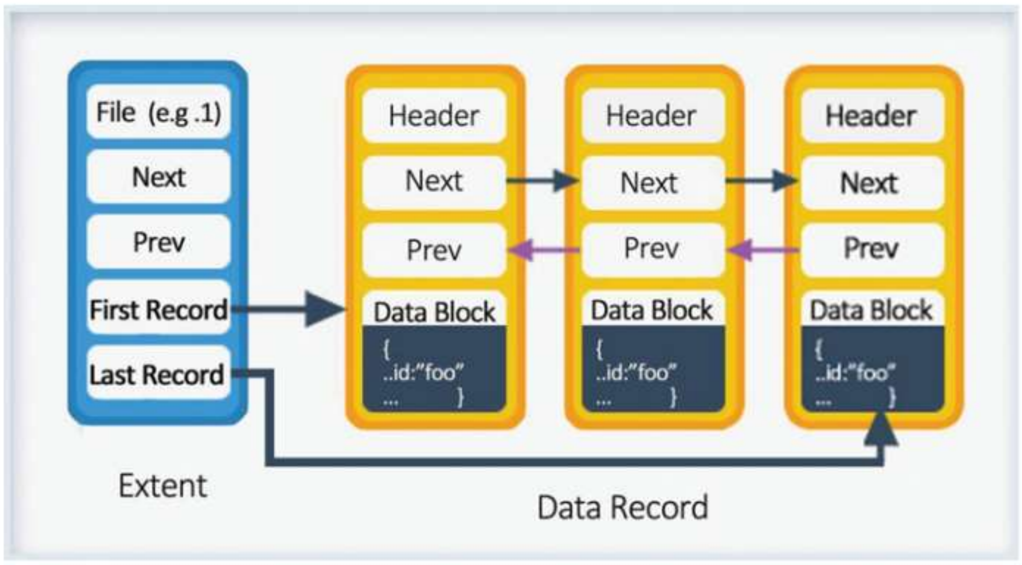


Figure 8-6. *Record data structure*

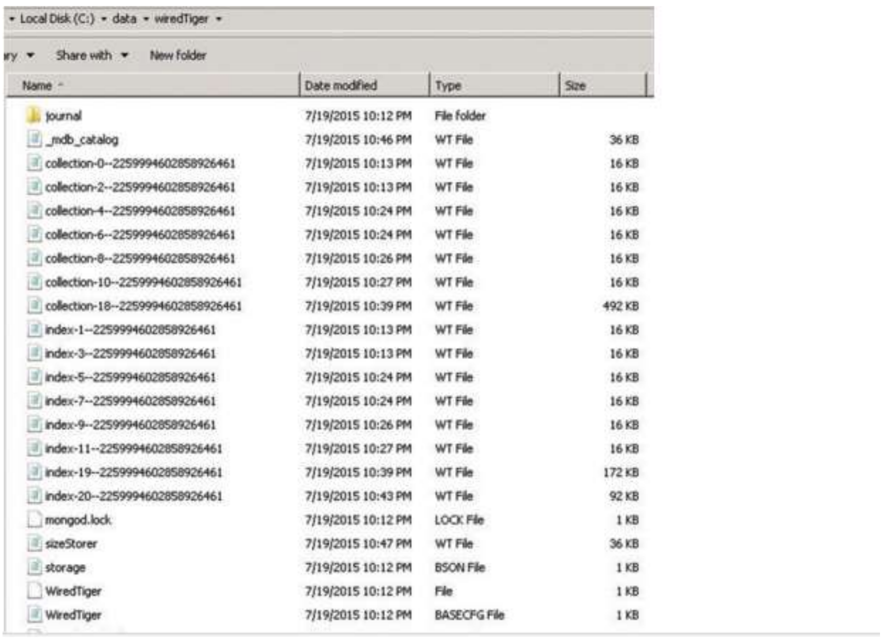
Deleted List :

- Deleted List The deleted list stores details of the extent whose data has been deleted or moved (movement whenever an update has caused change in size, leading to non-fitment of data in the allocated space).
- The size of the record determines the bucket in which the free extent needs to be placed.
- Basically these are bucketed single linked lists. When a new extent is required for fitting the data for the namespace, it will first search the free list to check whether any appropriate size extent is available.

Data File (Relevant for WiredTiger) :

#### **Question 1 :** What is Wired Tiger Storage Engine?

- When the storage option selected is WiredTiger, data, journals, and indexes are compressed on disk.
- The compression is done based on the compression algorithm specified when starting the mongod.
- Snappy is the default compression option.
- Under the data directory, there are separate compressed wt files corresponding to each collection and indexes.
- Journals have their own folder under the data directory.
- The compressed files are actually created when data is inserted in the collection (the files are allocated at write time, no preallocation).
- For example, if you create collection called users , it will be stored in collection-0—2259994602858926461 files and the associated index will be stored in index-1—2259994602858926461 , index-2—2259994602858926461 , and so on.
- In addition to the collection and index compressed files, there is **a _mdb_catalog** file that stores metadata mapping collection and indexes to the files in the data directory.
- In the above example it will store mapping of collection users to the wt file collection-0—2259994602858926461 .


Local Disk (C:) ▸ data ▸ wiredTiger ▸
ry ▾ Share with ▾ New folder

Name	Date modified	Type	Size
journal	7/19/2015 10:12 PM	File folder	
_mdb_catalog	7/19/2015 10:46 PM	WT File	36 KB
collection-0--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
collection-2--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
collection-4--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
collection-6--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
collection-8--2259994602858926461	7/19/2015 10:26 PM	WT File	16 KB
collection-10--2259994602858926461	7/19/2015 10:27 PM	WT File	16 KB
collection-18--2259994602858926461	7/19/2015 10:39 PM	WT File	492 KB
index-1--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
index-3--2259994602858926461	7/19/2015 10:13 PM	WT File	16 KB
index-5--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
index-7--2259994602858926461	7/19/2015 10:24 PM	WT File	16 KB
index-9--2259994602858926461	7/19/2015 10:26 PM	WT File	16 KB
index-11--2259994602858926461	7/19/2015 10:27 PM	WT File	16 KB
index-19--2259994602858926461	7/19/2015 10:39 PM	WT File	172 KB
index-20--2259994602858926461	7/19/2015 10:43 PM	WT File	92 KB
mongod.lock	7/19/2015 10:12 PM	LOCK File	1 KB
sizeStorer	7/19/2015 10:47 PM	WT File	36 KB
storage	7/19/2015 10:12 PM	BSON File	1 KB
WiredTiger	7/19/2015 10:12 PM	File	1 KB
WiredTiger	7/19/2015 10:12 PM	BASECFG File	1 KB

- WiredTiger cache is used for any read/write operations on the data.

Reads and Writes :

#### **Question 1 :** How read and write operations performed in MongoDB

##### Read and Write Operations in MongoDB Using the WiredTiger Storage Engine

1. Write Operations:

- **Data Storage Structure:** MongoDB stores data as documents in collections. These documents are stored in a binary format called BSON (Binary JSON), which is efficient for both storage and retrieval. WiredTiger, the default storage engine, uses a B-tree data structure to organize and index the data on disk.
- **In-Memory Changes:** When a write operation, such as an insert, update, or delete, is issued, MongoDB first modifies the data in an in-memory structure known as the WiredTiger cache. This is a temporary storage area that allows quick access to data.
- **Journal Files (Write-Ahead Logging):** To ensure durability (i.e., the persistence of data even in the event of a failure), MongoDB uses a technique called write-ahead logging. Before applying the changes to the actual data files, the changes are written to a separate journal file. The journal acts as a secure log that can be used to recover the data in case of a crash.
- **Checkpointing:** Periodically, MongoDB consolidates the changes in the WiredTiger cache and the journal into the data files on disk. This process is known as checkpointing. During checkpointing, all the in-memory changes are flushed to the data files, ensuring that the data on disk is up-to-date and consistent.
- **Example:** Suppose we insert a document { "name": "Alice", "age": 30 } into a collection. This operation is first recorded in the journal and the WiredTiger cache. During the next checkpoint, the data is written to the BSON file on disk, ensuring it is safely stored.

2. Read Operations:

- **Data Retrieval Process:** When a read operation, such as a query (find or findOne), is performed, MongoDB first checks if the requested data is available in the WiredTiger cache. This check is crucial as accessing data from memory is much faster than reading from disk.
- **Cache Hit and Cache Miss:**
  - **Cache Hit:** If the data is found in the cache, MongoDB quickly returns the result, minimizing the time taken to serve the request.

  - **Cache Miss:** If the data is not in the cache, MongoDB retrieves the required data from the disk. This data is then loaded into the cache for faster future access and returned to the user.
- **Concurrency Control:** WiredTiger employs a concurrency control mechanism called Multi-Version Concurrency Control (MVCC). MVCC allows multiple clients to read from the database without being blocked by write operations. It achieves this by maintaining multiple versions of the data, ensuring readers always see a consistent view, even as updates are being made.
- **Example:** For a query like { "name": "Alice" }, MongoDB first checks if this document is present in the WiredTiger cache. If it is, the document is returned immediately (cache hit). If not, MongoDB reads the data from the disk, caches it, and then returns it to the client (cache miss).

#### Read and Write Operations in MongoDB Using the MMAPv1 Storage Engine

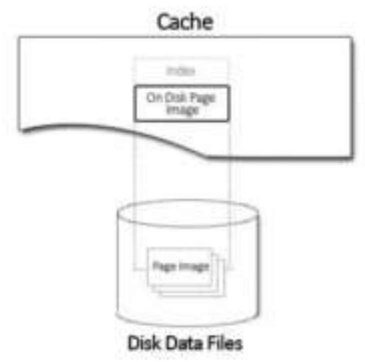
1. Write Operations:

- **Data Storage Structure:** The MMAPv1 storage engine stores data in collections, with each document stored as a BSON (Binary JSON) object. It uses a memory-mapped file approach, where data files on disk are mapped directly into the memory address space of MongoDB.
- **In-Memory Modifications:** When a write operation (insert, update, delete) occurs, MongoDB modifies the in-memory representation of the data. The changes are made directly in the mapped memory, which is also reflected in the corresponding disk location due to the memory-mapped nature of MMAPv1.
- **Memory-Mapped Files:** The use of memory-mapped files allows MMAPv1 to directly map the data files to virtual memory. This means that changes made to the in-memory data are also made to the files on disk, ensuring that the data remains consistent between memory and disk.
- **Journal Files (Write-Ahead Logging):** MMAPv1 uses a journal to log changes before they are applied to the data files. This journal acts as a write-ahead log that helps in recovering data in case of a system failure. The journal ensures durability by recording the operations before they are reflected in the data files.
- **Checkpointing and File Synchronization:** Periodically, MongoDB synchronizes the in-memory changes with the data files on disk. This process, known as checkpointing, ensures that all changes are safely written to disk. The synchronization can occur at different times, but it is essential to ensure data consistency.
- **Example:** Consider an insertion of a document { "name": "Bob", "age": 25 }. In MMAPv1, this document is added to the in-memory structure,

which is also reflected in the mapped file on disk. The operation is also recorded in the journal. During the next checkpoint, the data is permanently written to the data files.

2. Read Operations:

- **Data Retrieval Process:** For read operations, such as querying data with find or findOne, MongoDB uses the memory-mapped files to quickly access the data. Since the data files are mapped directly into memory, MongoDB can read data efficiently without the need to explicitly load it into a separate cache.
- **File Mapping and Data Access:** When a query is executed, MMAPv1 accesses the required data directly from the mapped memory. This process avoids the overhead of additional caching layers and provides quick access to the data.
- **Concurrency Control:** MMAPv1 uses a locking mechanism at the collection level to handle concurrent operations. This means that a write operation can block other operations on the same collection, potentially affecting concurrency and throughput.
- **Example:** If a query searches for a document with name: "Bob", MMAPv1 directly accesses the data from the mapped memory. Since the memory-mapped files reflect the actual data on disk, the retrieval is efficient. However, if a write operation is in progress, the read may be delayed until the write lock is released.



Difference Between MMAPv1 and Wired Tiger Storage Engine :

1. Data Storage Structure:

- **MMAPv1:** Uses memory-mapped files to directly map the data files into the memory address space, which means changes to data are immediately reflected in the files.

- **WiredTiger:** Utilizes a B-tree data structure for efficient storage and retrieval, providing better indexing and data management on disk.

2. Concurrency Control:

- **MMAPv1:** Employs collection-level locking, which means that a write operation can block other operations on the same collection, leading to potential bottlenecks.

- **WiredTiger:** Uses document-level locking with MVCC, allowing multiple operations to occur concurrently without blocking, resulting in better performance under load.

3. Write Performance:

- **MMAPv1:** Write operations directly modify the memory-mapped files, which can be slower and less efficient due to the locking mechanism.

- **WiredTiger:** Write operations are first written to an in-memory cache and journal, then periodically flushed to disk, making writes faster and more efficient.

4. Read Performance:

- **MMAPv1:** Reads data directly from the memory-mapped files, providing quick access but potentially slower under high concurrency due to the collection-level locking.

- **WiredTiger:** Reads data from an in-memory cache or disk using efficient indexing, with better performance under high concurrency due to document-level locking and MVCC.

5. Durability and Recovery:

- **MMAPv1:** Uses a journal for write-ahead logging to ensure durability. In case of a crash, the journal is used to replay operations and recover data.

- **WiredTiger:** Also uses a journal for write-ahead logging. Recovery is faster and more efficient due to better handling of in-memory and on-disk data consistency.

6. Checkpointing:

- **MMAPv1:** Periodically writes changes in memory to the data files on disk during checkpointing to ensure data consistency.

- **WiredTiger:** Periodically flushes the in-memory cache and journal to disk during checkpointing, ensuring consistent and up-to-date data files.

7. Storage Efficiency:

- **MMAPv1:** Less storage-efficient due to the overhead of memory-mapped files and potential fragmentation.

- **WiredTiger:** More storage-efficient with better compression and compaction features, reducing the overall storage footprint.

How Data Is Written Using Journaling :

#### **Question 1 :** What is Journaling? Explain the importance of Journaling with the help of a neat diagram
#### **Question 2 :** Discuss how data is written using Journaling.
#### **Question 3 :** Delineate the write operations performed using journaling .
#### **Question 4 :** Describe the process of write operations performed using journaling .

- MongoDB disk writes are lazy, which means if there are 1,000 increments in one second, it will only be written once.
- The physical writes occurs a few seconds after the operation.
- You will now see how an update actually happens in mongod.
- As you know, in the MongoDB system, mongod is the primary daemon process. So the disk has the data files and the journal files . see figure 8.15

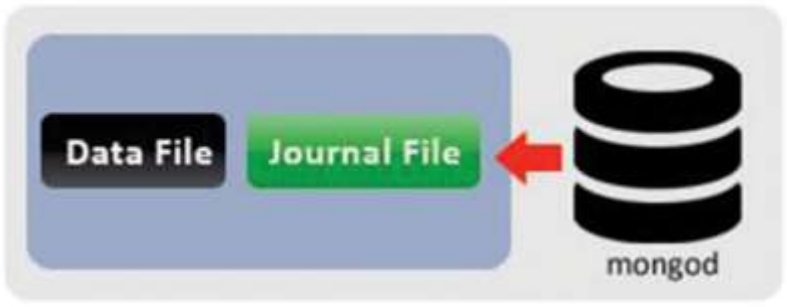


Figure 8-15. mongod

- When the mongod is started, the data files are mapped to a shared view. In other words, the data file is mapped to a virtual address space. See Figure 8-16 .

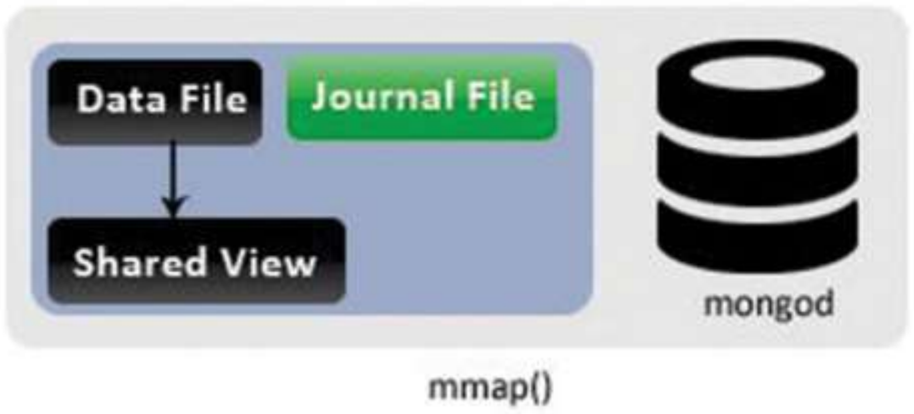


Figure 8-16. *maps to shared view*

- Basically, the OS recognizes that your data file is 2000 bytes on disk, so it maps this to memory address 1,000,000 – 1,002,000.
- Note that the data will not be actually loaded until accessed;
- The OS just maps it and keeps it. Until now you still had files backing up the memory. Thus any change in memory will be flushed to the underlying files by the OS.
- This is how the mongod works when journaling is not enabled. Every 60 seconds the in-memory changes are flushed by the OS.
- In this scenario, let's look at writes with journaling enabled.
- When journaling is enabled, a second mapping is made to a private view by the mongod. That's why the virtual memory amount used by mongod doubles when the journaling is enabled. See Figure 8-17 .

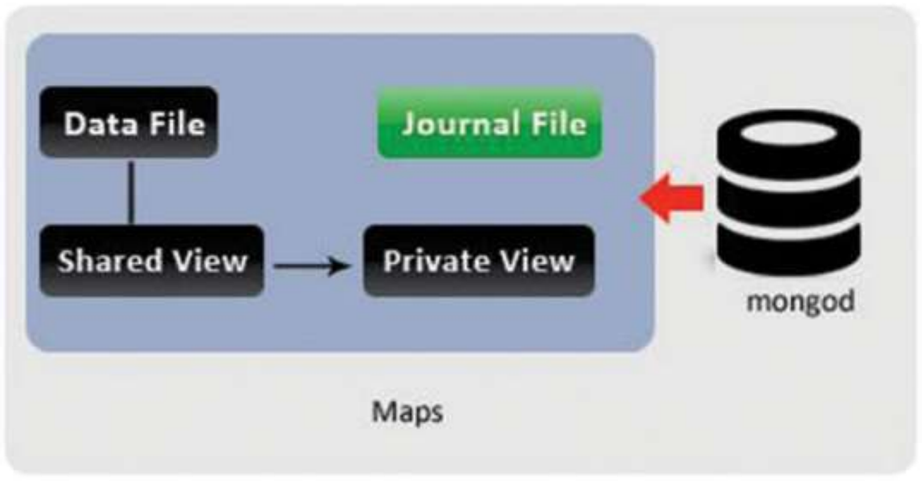


Figure 8-17. *maps to private view*

- You can see in Figure 8-17 how the data file is not directly connected to the private view, so the changes will not be flushed from the private view to the disk by the OS.

- Let's see what sequence of events happens when a write operation is initiated. When a write operation is initiated it, first it writes to the private view (Figure 8-18).

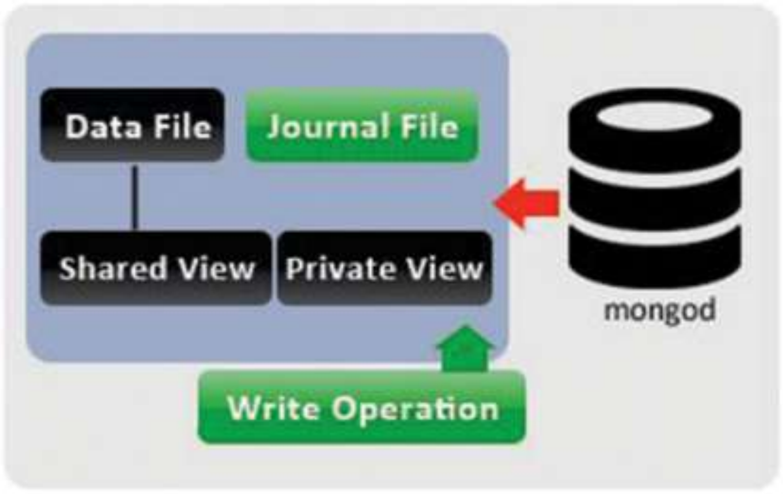


Figure 8-18. *Initiated write operation*

- Next, the changes are written to the journal file, appending a brief description of what's changed in the files (Figure 8-19).

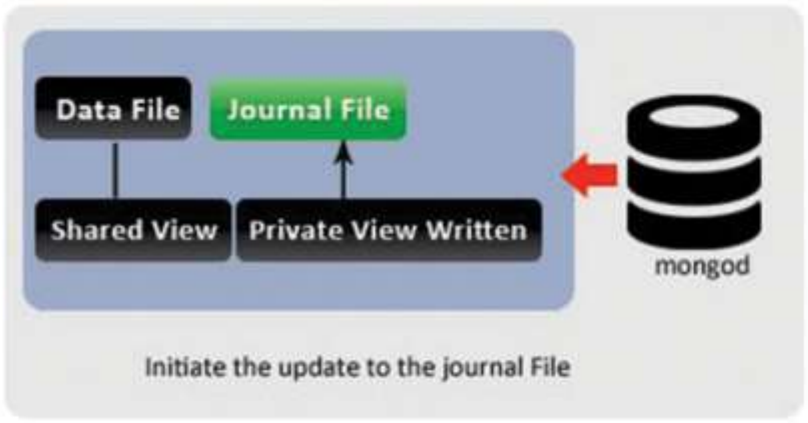


Figure 8-19. *Updating the journal file*

The journal keeps appending the change description as and when it gets the change. If the mongod fails at this point, the journal can replay all the changes even if the data file is not yet modified, making the write safe at this point.
The journal will now replay the logged changes on the shared view (Figure 8-20).

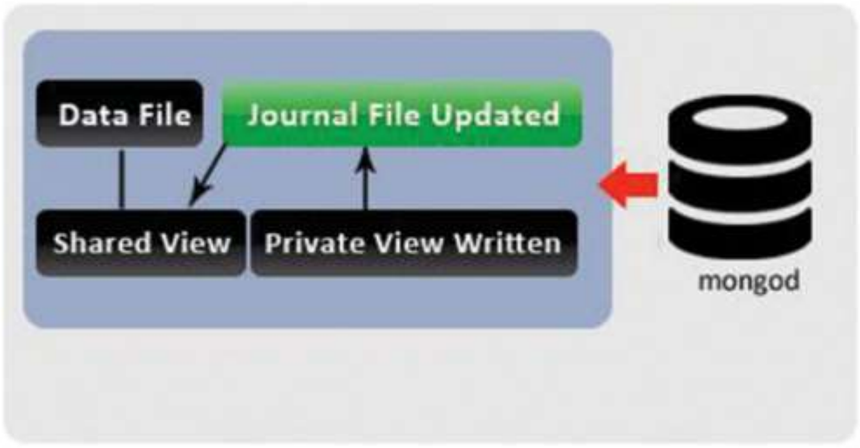


Figure 8-20. *Updating the shared view*

Finally, with a very fast speed the changes are written to the disk. By default, the OS is requested to do this every 60 seconds by the mongod (Figure 8-21).

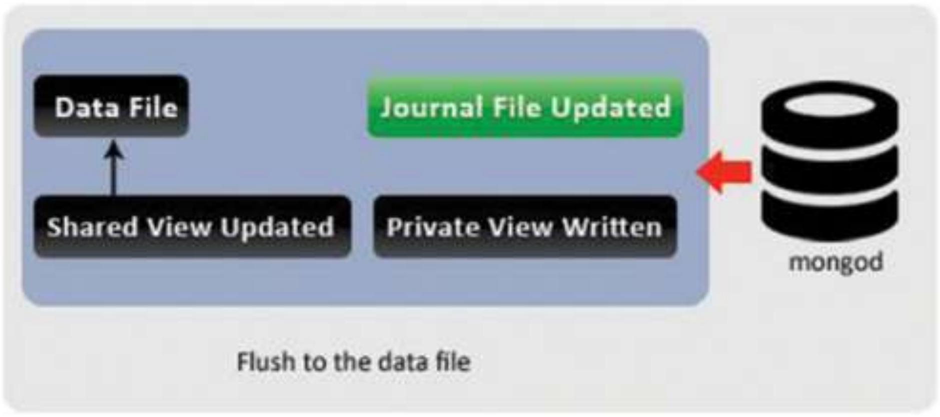


Figure 8-21. *Updating the data file*

In the last step, the shared view is remapped to the private view by the mongod. This is done to prevent the private view from getting too dirty (Figure 8-22).

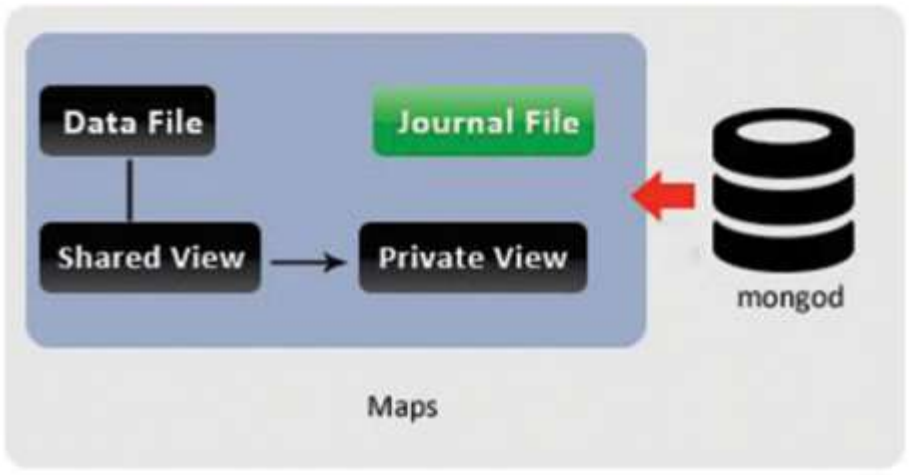


***Figure 8-22.** Remapping*

GridFS – The MongoDB File System :

#### **Question 1 :** Write a short note on GridFS
#### **Question 2 :** Explain the concept of GridFS – The MongoDB File System
#### **Question 3 :** Describe the "GridFS – MongoDB File System" in detail.

GridFS is a specification for storing and retrieving large files, such as images, audio files, video files, or any other binary data, in MongoDB.

- It overcomes the BSON document size limit of 16 MB by dividing large files into smaller chunks and storing each chunk as a separate document.

### Why Use GridFS?

1. **File Size Limitation**: Overcomes MongoDB's BSON document size limit of 16 MB by dividing files into smaller chunks.

2. **Efficient Storage**: Facilitates efficient storage and retrieval of large files.

3. **Partial Retrieval**: Allows partial retrieval of files without loading the entire file into memory.

4. **Metadata Storage**: Stores file metadata, useful for indexing and querying files.

### How GridFS Works

1. **Chunks Collection**: The large file is divided into smaller chunks, and each chunk is stored as a separate document in the `fs.chunks` collection.

2. **Files Collection**: Metadata about the file, such as filename, file size, upload date, and more, is stored in the `fs.files` collection.

3. **Chunk Size**: By default, each chunk is 255 KB in size, though this can be adjusted as needed.

### GridFS Structure

- **Chunks Collection**: Contains documents for each file chunk. Fields include:

  - `files_id`: Reference to the file's metadata document in the `fs.files` collection.

  - `n`: The sequence number of the chunk.

  - `data`: The actual binary data of the chunk.

- **Files Collection**: Contains metadata about the file. Fields include:

  - `_id`: Unique identifier for the file.

  - `length`: Total size of the file.

- `chunkSize`: Size of each chunk.
- `uploadDate`: Date and time when the file was uploaded.
- `filename`: Name of the file.
- `md5`: MD5 hash of the file for integrity verification.

### Benefits of GridFS

- **Handles Large Files**: Efficiently manages and stores files larger than 16 MB.

- **Streamlined File Access**: Enables partial reads and writes to large files.

- **Metadata Management**: Provides robust metadata storage for better file management and querying capabilities.

### Conclusion

GridFS is an effective solution in MongoDB for handling and storing large files by splitting them into smaller, manageable chunks and storing the metadata separately. This approach not only circumvents MongoDB's BSON document size limit but also enhances the efficiency of file storage and retrieval operations.

The Rationale of GridFS

### Why Use GridFS?

1. **Large File Handling:** MongoDB documents have a size limit of 16 MB. GridFS allows you to store files that are larger than this limit by breaking them into smaller chunks.

2. **Efficient Storage:** By dividing files into chunks, GridFS enables efficient storage and retrieval. Only the necessary chunks are loaded when accessing part of a file.

3. **Metadata Support:** GridFS supports metadata for each file, which can be useful for additional information such as file type, upload date, and user-specific data.

4. **Streamlining:** GridFS allows for the streaming of files, making it easier to handle large media files (such as videos and music) that need to be streamed to users.

GridFS under the Hood :

GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16MB. Instead of storing a large file in a single document, GridFS divides the file into smaller chunks and stores each chunk as a separate document. This allows MongoDB to handle and efficiently retrieve large files. Here’s a detailed explanation of how GridFS works, along with example explanations in simple words:

### How GridFS Works

1. **Splitting Files into Chunks:**

- GridFS splits large files into smaller chunks, typically 255 KB in size. Each chunk is stored as a separate document in a `chunks` collection.

- The `files` collection stores metadata for each file, such as filename, upload date, and length.

2. **Collections**:

- **`files` Collection**: Stores metadata about the files. Each document in this collection represents a single file.

- **`chunks` Collection**: Stores the actual data chunks of the files. Each chunk is a separate document, containing a reference to the file it belongs to and the chunk data.

3. **Storing Files**:

- When you upload a file, GridFS splits the file into chunks and stores each chunk in the `chunks` collection.

- A corresponding document is created in the `files` collection to store metadata about the file.

4. **Retrieving Files**:

- To retrieve a file, GridFS reads the file's metadata from the `files` collection and then fetches all the chunks from the `chunks` collection.

- The chunks are then reassembled to reconstruct the original file.

### Example Explanation

Suppose you have a large image file named `large_image.jpg` that is 1 MB in size. Here's how GridFS would handle this file:

1. **Splitting the File**:

- GridFS splits `large_image.jpg` into four chunks (since the default chunk size is 255 KB).

- Each chunk will be around 255 KB, except the last chunk, which will contain the remaining bytes of the file.

2. **Storing Metadata**:

- A document is created in the `files` collection with metadata for `large_image.jpg`. This document includes fields like:

  - `filename`: "large_image.jpg"

  - `length`: 1 MB

  - `chunkSize`: 255 KB

  - `uploadDate`: Timestamp when the file was uploaded

3. **Storing Chunks**:

- Four documents are created in the `chunks` collection, each representing a chunk of the file. These documents include fields like:

  - `files_id`: Reference to the file in the `files` collection

  - `n`: The chunk number (0, 1, 2, 3)

  - `data`: Binary data of the chunk

4. **Retrieving the File**:

- When you want to download `large_image.jpg`, GridFS looks up the metadata in the `files` collection to find the file's ID.

- GridFS then retrieves all chunks with the matching `files_id` from the `chunks` collection and sorts them by the `n` field.

- The chunks are reassembled in order to reconstruct the original file.

### Detailed Example

Let's go through a more detailed example with specific document structures:

#### Uploading `large_image.jpg`

1. **Metadata Document in `files` Collection**:

```
  {
    "_id": ObjectId("60c72b2f8f1b2c6d5f7a3e77"),
    "filename": "large_image.jpg",
    "length": 1048576,  // 1 MB
    "chunkSize": 261120,  // 255 KB
    "uploadDate": ISODate("2023-07-16T12:00:00Z"),
    "md5": "c1a3b2f9c6d5f7a3e772b2f8f1b2c6d5"  //for data
integrity
  }
```

2. **Chunk Documents in `chunks` Collection**:

```
  {
    "_id": ObjectId("60c72b2f8f1b2c6d5f7a3e78"),
    "files_id": ObjectId("60c72b2f8f1b2c6d5f7a3e77"),
    "n": 0,
    "data": <binary data of chunk 0>
  }
  {
    "_id": ObjectId("60c72b2f8f1b2c6d5f7a3e79"),
    "files_id": ObjectId("60c72b2f8f1b2c6d5f7a3e77"),
    "n": 1,
    "data": <binary data of chunk 1>
  }
  {
    "_id": ObjectId("60c72b2f8f1b2c6d5f7a3e7a"),
    "files_id": ObjectId("60c72b2f8f1b2c6d5f7a3e77"),
    "n": 2,
    "data": <binary data of chunk 2>
  }
```

```
{
  "_id": ObjectId("60c72b2f8f1b2c6d5f7a3e7b"),
  "files_id": ObjectId("60c72b2f8f1b2c6d5f7a3e77"),
  "n": 3,
  "data": <binary data of chunk 3>
}
```

#### Downloading `large_image.jpg`

1. **Retrieve Metadata:**

- Look up the document in the `files` collection using the filename or `_id`.

2. **Retrieve and Reassemble Chunks:**

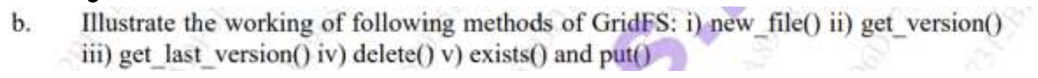
- Fetch all documents from the `chunks` collection with `files_id` matching the `_id` of the metadata document.
- Sort the chunks by the `n` field to get them in the correct order.
- Combine the `data` fields of the sorted chunks to reconstruct the original file.

### Summary

GridFS is an efficient way to store and retrieve large files in MongoDB by splitting them into manageable chunks. This allows MongoDB to handle files larger than the BSON-document size limit and ensures efficient storage and retrieval operations.

Using GridFS :

#### **Question 1 :**


b. Illustrate the working of following methods of GridFS: i) new_file() ii) get_version() iii) get_last_version() iv) delete() v) exists() and put()

GridFS is a specification for storing and retrieving files that exceed the BSON-document size limit of 16MB. Instead of storing a large file in a single document, GridFS divides the file into smaller chunks and stores each chunk as a separate document. Here's how to use GridFS with PyMongo, the Python driver for MongoDB:

### 1. Add Reference to the Filesystem

To start using GridFS, first, you need to import the required libraries and establish a connection to the MongoDB database.

```
from pymongo import MongoClient
import gridfs

# Establish a connection to the MongoDB server
client = MongoClient('mongodb://localhost:27017/')

# Select the database
db = client['mydatabase']

# Create a GridFS object
fs = gridfs.GridFS(db)
```

### 2. Writing a File to GridFS

```
To write a file to GridFS, you use the `put()` method.


# Example: Writing a file to GridFS

with open('example.txt', 'rb') as file:
    file_id = fs.put(file, filename='example.txt')
print(f'File ID: {file_id}')
```

### 3. Finding a File in GridFS

To find a file in GridFS, you use the `find()` method.

```
# Example: Finding a file in GridFS
file = fs.find_one({'filename': 'example.txt'})
if file:
    print('File found:', file.filename)
else:
    print('File not found')
```

### 4. Forcing a Split of the File

GridFS automatically splits the file into chunks, so there's no need to manually force a split. The default chunk size is 255KB, but it can be changed if needed.

```
# Example: Writing a file with a custom chunk size
with open('example.txt', 'rb') as file:
    file_id = fs.put(file, filename='example.txt', chunkSizeBytes=1024)
print(f'File ID: {file_id}')
```

### 5. Reading a File from GridFS

To read a file from GridFS, you use the `read()` method.

```
# Example: Reading a file from GridFS
file = fs.find_one({'filename': 'example.txt'})
if file:
    content = file.read()
    print(content.decode('utf-8'))
else:
    print('File not found')
```

### 6. Treating GridFS More Like a File System

#### a. Creating a New File in GridFS

To create a new file in GridFS, you can use the `new_file()` method.

```
# Example: Creating a new file in GridFS
new_file = fs.new_file(filename='newfile.txt')
new_file.write(b'Hello, GridFS!')
new_file.close()
```

#### b. Getting a Specific Version of a File

To get a specific version of a file, you can use `get_version()` or `get_last_version()`.

```
# Example: Getting the latest version of a file
```

```
file = fs.get_last_version('example.txt')

if file:

   print('Latest version content:', file.read().decode('utf-8'))
```

```
# Example: Getting a specific version of a file

file = fs.get_version('example.txt', version=1)

if file:

   print('Version 1 content:', file.read().decode('utf-8'))
```

#### c. Deleting a File from GridFS

To delete a file from GridFS, you use the `delete()` method.

```
# Example: Deleting a file from GridFS

file = fs.find_one({'filename': 'example.txt'})

if file:

   fs.delete(file._id)

   print('File deleted')
```

#### d. Checking if a File Exists

To check if a file exists in GridFS, you use the `exists()` method.

```
# Example: Checking if a file exists in GridFS
if fs.exists({'filename': 'example.txt'}):
   print('File exists')
else:
   print('File does not exist')
```

#### e. Putting a File into GridFS

The `put()` method is used to write a file to GridFS, as shown earlier.

```
# Example: Putting a file into GridFS
with open('example.txt', 'rb') as file:
    file_id = fs.put(file, filename='example.txt')
print(f'File ID: {file_id}')
```

### Complete Example

Here's a complete example that includes all the steps mentioned above:

```
from pymongo import MongoClient
import gridfs

# Connect to MongoDB
client = MongoClient('mongodb://localhost:27017/')
db = client['mydatabase']
fs = gridfs.GridFS(db)

# Writing a file to GridFS
with open('example.txt', 'rb') as file:
    file_id = fs.put(file, filename='example.txt')

# Finding and reading the file
file = fs.find_one({'filename': 'example.txt'})
if file:
    print('File found:', file.filename)
    content = file.read()
    print('Content:', content.decode('utf-8'))

# Creating a new file
new_file = fs.new_file(filename='newfile.txt')
new_file.write(b'Hello, GridFS!')
new_file.close()

# Getting the latest version
file = fs.get_last_version('example.txt')
if file:
    print('Latest version content:', file.read().decode('utf-8'))

# Deleting the file
file = fs.find_one({'filename': 'example.txt'})
if file:
    fs.delete(file._id)
    print('File deleted')

# Checking if a file exists
if fs.exists({'filename': 'example.txt'}):
    print('File exists')
else:
    print('File does not exist')
```

This comprehensive example demonstrates how to use various GridFS operations with PyMongo in a simple and clear manner.

Indexing :

Indexing is a way to make querying in MongoDB faster. When you search for something in a database, MongoDB uses indexes to quickly locate the data without scanning every document in the collection. Think of an index like the index of a book that helps you find the page number of a topic quickly.

### Types of Indexes in MongoDB

1. **_id Index**

2. **Secondary Indexes**

#### 1. _id Index

- **What is it?**

  Every document in a MongoDB collection automatically has a unique identifier called `_id`. MongoDB automatically creates an index on the `_id` field. This index is unique, meaning no two documents can have the same `_id` value.

- **Why is it important?**

  The `_id` index ensures that each document can be uniquely identified and quickly retrieved using its `_id`. Since MongoDB uses `_id` internally to fetch documents, this index is always created by default.

- **Example Explanation:**

  Imagine you have a collection of students, and each student has a unique ID (like a roll number). The `_id` index works like a fast way to find a student by their ID. If you want to find the student with `_id:

101`, MongoDB uses the `_id` index to quickly locate the student without checking every student's record.

#### 2. Secondary Indexes

- **What are they?**

Besides the `_id` index, you can create additional indexes on other fields in your documents. These are called **secondary indexes**. They help you to optimize queries that filter or sort by fields other than `_id`.

- **Why are they important?**

Secondary indexes are useful when you frequently search, sort, or filter data by fields other than `_id`. Without these indexes, MongoDB would have to scan every document to find matches, which can be slow for large collections.

- **Types of Secondary Indexes:**

- **Single Field Index:**

This is an index on a single field in a document. For example, if you often search for students by their `name`, you can create an index on the `name` field.

- **Compound Index:**

This is an index on multiple fields. For instance, if you frequently search for students by both `name` and `age`, you can create a compound index on `name` and `age`.

- **Unique Index:**

A unique index ensures that the indexed field does not have duplicate values. For example, if each student has a unique email, you can create a unique index on the `email` field to enforce this rule.

- **Example Explanation:**

Let’s say you have a collection of books, and you often search for books by their `title` and `author`. If you create a single field index on the `title`, MongoDB can quickly find books by title. If you create a compound index on both `title` and `author`, MongoDB can efficiently handle queries that search for books by both fields together.

### Summary

- **_id Index:** Automatically created, unique, used to find documents quickly by their `_id`.

- **Secondary Indexes:** Custom indexes you create on other fields to speed up specific queries, including single field, compound, and unique indexes.

### Indexing - Indexes with Keys Ordering in MongoDB

#### Explanation

In MongoDB, **indexes** are special data structures that store a small portion of the data set in a form that's easy to search. This makes querying data much faster.

Indexes with Keys Ordering refers to how MongoDB organizes the order of keys (fields) in an index. When you create an index on multiple fields, the order in which you list these fields matters. This order affects how MongoDB can efficiently use the index for queries.

#### Key Concepts:

1. **Ascending and Descending Order:**

- When creating an index, you can specify whether the index should sort the data in ascending (`1`) or descending (`-1`) order.

- Example: `{ "field1": 1, "field2": -1 }`

  - This creates an index where `field1` is sorted in ascending order, and if there are multiple records with the same `field1` value, `field2` is sorted in descending order.

2. **Compound Indexes:**

- An index that includes more than one field is called a **compound index**.

- The order of fields in a compound index is critical because MongoDB will first sort by the first field, then by the second field, and so on.

- Example: `{ "lastName": 1, "firstName": 1 }`

  - This index first orders by `lastName` in ascending order, and within each `lastName`, it orders by `firstName` in ascending order.

3. **Use in Queries:**

- The order of fields in an index influences how well MongoDB can optimize a query. Queries that match the order of fields in the index can use the index efficiently.

- If your query matches the exact order of the fields in the index, MongoDB can quickly locate the relevant data.

- Example: If you have an index on `{ "lastName": 1, "firstName": 1 }` and your query is `{ "lastName": "Smith", "firstName": "John" }`, MongoDB can use this index effectively.

#### Example Explanation:

Imagine you have a database of students, and each student has a `lastName` and `firstName`. If you frequently search for students by both their last name and first name, you might create a compound index like this:

```
db.students.createIndex({ "lastName": 1, "firstName": 1 })
```

- This index sorts the students by `lastName` in ascending order. For students with the same `lastName`, it sorts them by `firstName` in ascending order.

Now, if you run a query like this:

```
db.students.find({ "lastName": "Smith", "firstName": "John" })
```

MongoDB will quickly find all students with the `lastName` "Smith" and the `firstName` "John" because the index is ordered in a way that supports this query.

However, if you query only by `firstName`, like this:

```
db.students.find({ "firstName": "John" })
```

MongoDB might not use the index as effectively because the index is ordered by `lastName` first. This is why the order of fields in a compound index matters.

#### Summary

- **Indexes with Keys Ordering** allow MongoDB to optimize queries by sorting fields in a specified order.

- The order of fields in a compound index is crucial for efficient querying.

- Always consider how you will query the data when deciding the order of fields in an index.

### 1. **Unique Indexes**

Explanation:

A **unique index** ensures that the indexed field does not contain duplicate values. This is useful when you need to enforce uniqueness for a field, like ensuring that email addresses in a collection are unique.

Example Explanation:

Imagine you have a MongoDB collection called `users` with fields like `username`, `email`, and `age`. If you want to make sure that no two users can have the same email address, you would create a unique index on the `email` field.

```
db.users.createIndex({ email: 1 }, { unique: true })
```

Here, `{ email: 1 }` indicates the field and the sort order (1 for ascending), and `{ unique: true }` makes the index unique. Now, if you try to insert another document with an existing email, MongoDB will prevent the insertion.

### 2. **dropDups**

Explanation:

The `dropDups` option was used in earlier versions of MongoDB (before version 3.0) to remove duplicate documents from a collection when creating a unique index. If duplicates were found, only the first occurrence was kept, and others were dropped. However, this option has been deprecated because it could lead to unexpected data loss.

Example Explanation:

Suppose you had a collection with multiple users having the same email due to some earlier mistake. If you created a unique index on the `email` field with the `dropDups` option, MongoDB would keep the first user with a unique email and delete the rest. Since this option is deprecated, it's recommended to handle duplicates manually.

### 3. **Sparse Indexes**

#### **Question 1 :** Explain the Sparse and Geospatial Index in detail

Explanation:

A **sparse index** only indexes documents that have the indexed field present and not `null`. This is helpful when you have fields that are optional or not present in every document.

Example Explanation:

Consider a `products` collection where some products have an `expiryDate` field and others don't. If you create a sparse index on `expiryDate`, MongoDB will only index those documents where `expiryDate` is present.

```
db.products.createIndex({ expiryDate: 1 }, { sparse: true })
```

Now, searches involving `expiryDate` will be more efficient, but documents without this field won't be included in the index.

### 4. **TTL Indexes (Time To Live)**

Explanation:

A **TTL index** is a special type of index that automatically deletes documents from a collection after a specified amount of time. This is useful for data that should expire, like sessions, logs, or cache entries.

Example Explanation:

Suppose you have a `sessions` collection where each document has a `createdAt` field indicating when the session was created. If you want sessions to expire after 24 hours, you can create a TTL index on `createdAt`.

```
db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds:
86400 })
```

Here, `86400` seconds equals 24 hours. MongoDB will automatically delete documents older than this period.

### 5. **Geospatial Indexes**

#### **Question 1 :** Explain the Sparse and Geospatial Index in detail

Explanation:

A **geospatial index** allows MongoDB to efficiently query documents based on geographic location data, like finding the nearest points or searching within a geographical area. There are two types of geospatial indexes: `2d` (for flat, two-dimensional plane) and `2dsphere` (for spherical, Earth-like surfaces).

Example Explanation:

Imagine you have a `locations` collection where each document has a `location` field with coordinates. To perform efficient queries like finding places near a user's location, you would create a geospatial index.

```
db.locations.createIndex({ location: "2dsphere" })
```

With this index, you can easily find the nearest locations to a given point or search within a specified radius.

Each of these indexes serves a different purpose and can optimize the performance of your queries in MongoDB, depending on your use case.

Behaviors and Limitations :

Finally, the following are a few behaviors and limitations that you need to be aware of:

• More than 64 indexes may not be allowed in a collection.

• Index keys cannot be larger than 1024 bytes.

• A document cannot be indexed if its fields' values are greater than this size.

• The following command can be used to query documents that are too large to index: db.practicalCollection.find({: }).hint({$natural: 1})

• An index name (including the namespace) must be less than 128 characters.

• The insert/update speeds are impacted to some extent by an index.

• Do not maintain indexes that are not used or will not be used.

• Since each clause of an $or query executes in parallel, each can use a different index.

• The queries that use the sort() method and the $or operator will not be able to use the indexes on the $or fields.

• Queries that use the $or operator are not supported by the second geospatial query.