Leveraging a NoSQL Database for Enhanced User Experience and Data Accessibility

Joanne Senoren

Master of Science, Data Analytics

Table of Contents

Part 1: Design Document

A1. Business Problem

HealthFit Innovations' development of the "HealthTrack" platform allows different users to manage and interact with a complex system of electronic health records, tracking data, and product inventory data. A platform like "HealthTrack" must focus on efficiency and security since it must adapt to different user groups and their data needs. This objective brings us to the business problem: How can HealthFit Innovations leverage its data to provide up-to-date and relevant health insights to their patient, clinician, and business-owner end-users? This paper aims to explore how a NoSQL database can support a platform like "HealthTrack" to organize various data types and structures so that the application remains adaptable to different user segments.

A healthcare database can quickly progress to big data depending on the data variety, structures, and frequency. Gamal et al. (2020) state that issues with different schemas and data formats, as data volume grows, can quickly face storage and structural challenges. HealthFit Innovations anticipates that they will receive data in unknown formats and structures. A relational database cannot resourcefully address these unknowns because it stores data in a way where relationships are predefined and explicit. A NoSQL or non-relational database system, such as a document-style database, can handle unknown data structures more efficiently because they are schema-less. Section A3 discusses why a NoSQL database can help solve HealthFit Innovations' business problem.

One entity cannot own electronic health records. Patients, clinicians, and health organization members access them for different purposes (Gamal et al., 2020). Since each group

accesses electronic health records for various reasons, we can derive several business questions based on the user. Section D3 explores these questions and their related queries.

A2. Justification for NoSQL Database

As mentioned in section A1, HealthFit Innovations collects data from various sources and expects to gain diverse data quickly. Currently, the organization has patient medical records and information on wearable devices. However, additional structured and unstructured data might culminate in the future. For instance, health data can come from mobile apps or wearable devices such as smartwatches that monitor steps, oxygen levels, and calories burned, processing volumes of data at high speed. HealthFit Innovations can expect a massive jump in data volume if they want to gather real-time health information from patients using their devices (MongoDB, n.d.-i).

Patients will also want to recurrently review or update their profiles as part of their health management through the platform. The flexible and scalable NoSQL databases can accommodate this level of complexity, data volume, and velocity (Coronel & Morris, 2022).

NoSQL databases, unlike SQL databases, can store unstructured, structured, and semi-structured data in one database. The data can also be stored in the same format as it appears on an application, making it an efficient choice for HealthFit Innovations' platform (MongoDB, n.d.-h). NoSQL databases scale out horizontally, which is more affordable than a SQL database's scale-up approach that requires paying for more CPU and RAM power (MongoDB, n.d.-h).

Furthermore, NoSQL databases can support large amounts of scant data that relational databases could find organizationally challenging due to their restrictive schemas (Coronel & Morris, 2022). For example, not all patients in the database could own a wearable device. This can become a relationship or constraint issue in a relational database where there would be

empty rows for the attribute. However, in a NoSQL document format, we can simply leave out the field for the wearable device.

Tables 1 and 2 below demonstrate some functional and non-functional requirements to support the development of a NoSQL database for the “HealthTrack” platform.

Table 1

Functional Requirements

Aspect	Functional Requirement
Focus	Users should be able to view and modify data without being limited by a predefined schema in a secure environment.
Nature	The database should allow users to update, retrieve, and review data on the platform independently.
Fast CRUD Operations	Users should be able to perform CRUD operations (create, read, update, delete) on the “HealthTrack” platform with low latency.

Table 2

Non-Functional Requirements

Aspect	Non-Functional Requirements
Security	Data should be stored and encrypted due to the highly sensitive nature of medical records.
Reliability	The system upholds high data availability through features such as utilizing replication, sharding, and failover to reduce downtime.
Flexibility	The database schema should be flexible with limited validation to adapt to evolving data.

A3. Type of NoSQL Database

There are several types of NoSQL databases. MongoDB stores records in a document-oriented structure with key-value pairs. The values can be an assortment of data types, such as integers, strings, arrays, and objects (MongoDB, n.d.-c).

The documents do not conform to a predefined schema, and each document can have its own structure (Coronel & Morris, 2022). A MongoDB database groups these documents into logical data sets called "collections." Each document database can have multiple collections that house documents with similar contents to be processed by a single application.

MongoDB can ingest HealthFit Innovations' JSON files as BSON (binary JSON), which supports more data types than JSON. BSON is also lightweight and efficient; a large amount of data can be stored in a BSON file, and since BSON uses prefixes, it is faster to scan and query than JSON (MongoDB, n.d.-e).

A4. How Business Data Will Be Used

Before loading the data to the D597 Task 2 database, a temporary staging environment will extract the data from the JSON files. Staging databases give data admins a temporary environment for cleaning, data type checks, and transformations. Once the data has been transformed, it will be loaded to the destination database, and we can delete the temporary database. This process follows the pattern of the ETL process – extract, transform, load. The ETL procedure ensures data quality across consistency, usability, and reliability for accurate analysis (Daniel, 2023). We can now perform a manual and high-level ETL procedure since only two JSON documents and limited data types exist. However, as data for HealthFit Innovations grows exponentially, automated ETL processing must take priority over manual methods to minimize

errors. Organizations that expect to manage big data should consider software solutions that can seamlessly perform ETL procedures and integrate a range of diverse data (Daniel, 2023).

The user's role will determine how to use the business data on "HealthTrack." The platform should be able to provide both clinicians and patients access to the data based on their data goals. Clinicians can gather general demographic trends and patterns about patients within the medical records dataset. Suppose clinicians want to see if there is a pattern between gender and the status of a patient's medical condition. The clinician can aggregate data from the medical record by gender and medical condition to observe whether there is a notable difference. Patients can access their health records and update their wearable devices. For example, a patient who owns a MiFit band and recently switched to using a FitBit device can update their profile to reflect this change. The patient user (as a consumer) and business owner can benefit from a platform page showing a current inventory of health trackers and promotional sales items.

B. Database Design and Scalability

Due to the dynamic nature of our target data, performance must take priority when querying and retrieving accurate data. NoSQL databases utilize sharding, meaning data is horizontally scaled in clusters and distributed across multiple servers. This design handles large amounts of spikes in data more efficiently than SQL databases, which scale out vertically and require an expensive increase in storage and CPU capabilities (MongoDB, n.d.-i). Sharding also supports a large amount of data reads and writes efficiently. This performant design accommodates instances such as when multiple clinicians update the database with information on hundreds of patients on an hourly basis.

As mentioned throughout section A, MongoDB's document-style database exhibits high availability. Since patients and clinicians will use the application, data must be retrieved

efficiently with little latency. MongoDB can store groups of documents together across a cluster of servers, each consisting of multiple nodes containing records. As the platform grows into millions of records, performance can improve because all nodes can read their data in parallel to each other for a single query (MongoDB, n.d.-b).

Fault tolerance is also important to a platform that must provide data to users on demand. We anticipate that queries requested from "HealthTrack" will run throughout the day due to its many users. That means there needs to be a programmed method to maintain data durability if data becomes unavailable. HealthFit Innovations can enable replications of each data node so that if a server goes down, it can access a secondary server with copies of the downed node (MongoDB, n.d.-b). Fault tolerance is related to scalability because the possibility that a server can fail increases as more data must be processed.

Suppose "HealthTrack" consumes highly dynamic, time-stamped data such as steps, heart rate, and sleep patterns. MongoDB can manage the incoming data through time series collections. These collections can live within the same database as other collections, consisting of elements such as when the data point was taken, metadata, and metrics observed. Time series collections have improved query efficiency with reduced disk usage (Team, n.d.-b). This special collection type is efficient because it allows users to access historical records by querying the same database.

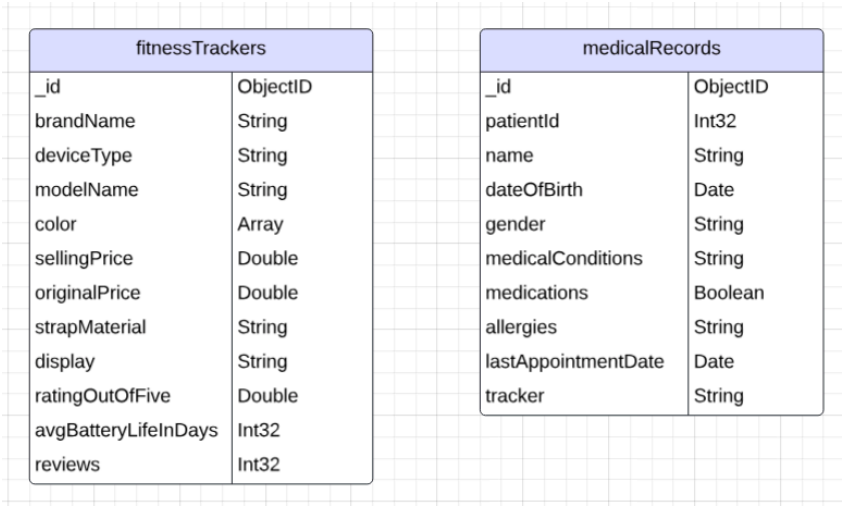
MongoDB can also scale the documents as embedded documents. This feature benefits users such as clinicians, who will upload a myriad of data from medical records to lab scans to treatment plans. The uploaded data will encompass a range of data types, such as files, images, and text. Suppose a clinician wants to upload a patient's latest treatment plan. This treatment plan can be stored in an embedded array with a dedicated field for that specific patient document.

When the patient looks up their new treatment plan on the platform, it takes only one query to look up nested values from the document. Patient documents can evolve into embedded documents with large amounts of data, but querying can remain performant since it can access all the necessary data in one read (MongoDB, n.d.-d).

After checking for nulls, duplicates, and field counts demonstrated in section D1, we can assume that all documents consist of the same fields and same number of fields. Since "HealthTrack" is one platform that uses data from both files, we can store them in two separate collections within the D597 Task 2 database. As data grows for each patient, we can implement embedded documents in the medical records to accommodate the additional data. Figure 1 below shows two separate collections, 'medicalRecords' and 'fitnessTrackers', as a preliminary database model for "HealthTrack." The schema analysis tool determined the appropriate BSON data type for each field. The data types and schema shape will change as data grows.

Figure 1

D597 Task 2 Physical Model



The generated model in Figure 1 shows the collections as pseudo tables, with the JSON field names and BSON data types. MongoDB does not possess the same modeling tools as standard SQL databases. So, they recommend a schema expressed in "a form that can be incorporated in an application, such as a JSON document" (MongoDB, n.d.-f).

Figure 1 also does not show explicit relationships between the two collections. NoSQL does not require inherent relationships between collections mainly because it stores data accessed together. As MongoDB states, "data that is accessed together should be stored together" (Jenkins, 2022). A database admin can create relationships through embedded documents and referencing.

C. Privacy and Security Measures

Data security is a crucial measure that any company of any size must take. As HealthFit Innovations begins implementing an exponentially growing database, data security must become a priority. The methods discussed in this paper cover both data privacy and data security. Data privacy determines how personal information is collected, shared, and used. In contrast, data security protects data from "unauthorized access, theft, or damage" (Babitz, 2024). HealthFit Innovations must prioritize data privacy compliance and security because the company uses sensitive patient data that is highly vulnerable to data breaches.

Fortunately, MongoDB enlists seven best practices for data security. See Table 3 below for a description and functionality of each best practice (MongoDB, n.d.-a).

Table 3

MongoDB's 7 Best Practices for Data Security (MongoDB, n.d.-a)

Practice	Description

Separate Security Credentials	Database owners should avoid sharing access authorizations to eliminate compromised accounts. Each user must have their own credentials and be assigned specific access privileges.
User-Based Access Control	Credential roles should be established across the organization, such as developer, database admin, database owner, and client user. These established roles will limit privileges so that only necessary data is accessed per role.
Limited Database Connections	Select IP addresses of database connections should be safelisted to avoid malicious remote access to databases.
Data Encryption	Data should be encrypted in transit as it passes from the server to the platform so that if malicious actors capture the data, they will be unable to read it. Data encryption can also add security when stored and not in use.
Additional Encryption	Personally identifiable information (PII) is particularly vulnerable to data breaches. Client-side field-level encryption ensures that PII, such as medical information, is only viewed by the owners, such as the clinician and patient. Database administrators could not view them when querying the document in the database.
Audits and Logs	Audit and activity logs through identity and access management services allow database owners to verify actions taken on the database, such as when they were made and who made them.
Community Server vs. Enterprise Server	When HealthFit Innovations reaches the level of big data, it should consider using a robust enterprise-level database with additional security measures that will benefit a much more extensive database at scale.

Babitz (2024) states that organizations that handle sensitive information must adhere to data security regulations. Such regulatory bodies include the General Data Protection Regulation (GDPR) in Europe, the Health Insurance Portability and Accountability Act (HIPAA) in healthcare, and the Payment Card Industry Data Security Standard (PCI DSS). HealthFit Innovations must practice data access auditing, encryption inspections, and develop reports to show their organization complies with these regulatory bodies (Babitz, 2024).

An established data security policy outlines how an organization handles, protects, and responds to data security breaches regarding sensitive data. A sound policy should include defined access controls, protocols for data handling, a plan for responding to a breach, and routine employee data security training.

Part 2: Implementation

D1. Database Instance

Performing a high-level ETL procedure to create the NoSQL database for "HealthTrack" will involve the following steps: create a staging database, clean the data, create the primary database, and then load the cleaned data to the permanent database. We can extract, clean, and transform the data specifically in the temporary staging database prior to loading it to its permanent database location (Medlock, 2021).

MongoDB documentation states that a user can only create a database at the initial store of a collection or a document (*Create a MongoDB Database*, n.d.). There are a few ways to create a MongoDB database and import data. MongoDB has database tools, which are command-line utilities downloaded alongside MongoDB. The database tool command mongoimport can create a database, generate a collection, and simultaneously import the JSON files as documents to the newly created database. Figure 2A demonstrates the command line script using mongoimport.

Figure 2A

Database and Collection Creation and Import for Medical Records Using mongoimport

```
(base) joanne@Joannes-MacBook-Air Task 2 Scenario 1 % mongoimport -d D597Task2Staging -c medicalRecords --jsonArray --file medical.json
2025-01-05T21:24:36.538-0800    connected to: mongodb://localhost/
2025-01-05T21:24:39.538-0800    [##############.........] D597Task2Staging.medicalRecords      15.8MB/25.8MB (61.0%)
2025-01-05T21:24:41.391-0800    [########################] D597Task2Staging.medicalRecords      25.8MB/25.8MB (100.0%)
2025-01-05T21:24:41.391-0800    100000 document(s) imported successfully. 0 document(s) failed to import.
```

Figure 2B

Database and Collection Creation and Import for Fitness Trackers Using mongoimport

```
(base) joanne@Joannes-MacBook-Air Task 2 Scenario 1 % mongoimport -d D597Task2Staging -c fitnessTrackers --jsonArray --file Task2Scenario1\ Dataset\ 1_fitness_trackers.json
2025-01-05T21:39:33.492-0800    connected to: mongodb://localhost/
2025-01-05T21:39:33.652-0800    565 document(s) imported successfully. 0 document(s) failed to import.
```

Once a database instance has been created with a collection, mongoimport can recognize this database and access it to create a new collection for additional imports. The script was rewritten to add the fitnessTrackers collection for the exercise trackers JSON file (see Figure 2B). Figure 3 shows the MongoDB shell commands that confirm the database instance was created along with two collections. It also confirms that all documents were imported into their respective collections, as demonstrated by the count documents and find queries.

Figure 3

MongoDB Shell Staging Creation and Import Confirmation

```
test> use D597Task2Staging
switched to db D597Task2Staging
D597Task2Staging> show collections
fitnessTrackers
medicalRecords
D597Task2Staging> show collections
fitnessTrackers
medicalRecords
D597Task2Staging> db.medicalRecords.countDocuments()
100000
D597Task2Staging> db.fitnessTrackers.countDocuments()
565
D597Task2Staging> db.medicalRecords.find({}).limit(1)
[
  {
    _id: ObjectId('677b6914f7998261cef81a4f'),
    patient_id: 1,
    name: 'Scott Webb',
    date_of_birth: '4/28/1967',
    gender: 'F',
    medical_conditions: 'None',
    medications: 'No',
    allergies: 'None',
    last_appointment_date: '7/26/2022',
    Tracker: 'Band 4'
  }
]
D597Task2Staging> db.fitnessTrackers.find({}).limit(1)
[
  {
    _id: ObjectId('677b6c9524770f14654d5715'),
    'Brand Name': 'Xiaomi',
    'Device Type': 'FitnessBand',
    'Model Name': 'Smart Band 4',
    Color: 'Black',
    'Selling Price': '2,099',
    'Original Price': '2,499',
    Display: 'AMOLED Display',
    'Rating (Out of 5)': 4.2,
    'Strap Material': 'Thermoplastic polyurethane',
    'Average Battery Life (in days)': 14,
    Reviews: ''
  }
]
```

At this point, the data needs to be observed for cleaning and data type checks. Regarding NoSQL documents, we cannot anticipate that a certain number of fields must exist for each document. This feature is a flexible and scalable characteristic of a NoSQL database. Therefore, we cannot assume and mandate that all documents must have the same number of fields. However, we can check if all current documents have the same fields to ensure an efficient migration to the permanent database. Aggregating in MongoDB is a method that goes through ‘stages’ to perform complex procedures on documents using built-in operators. Figure 4A shows how the aggregation pipeline and each stage takes each document in medicalRecords and converts it to key-value pairs, then groups the documents based on their keys with the forEach()

method printing out each unique key (Aqsa, 2024). Consequently, Figure 4B shows the fields found in the fitnessTrackers collection.

Figure 4A

Aggregation Pipeline to Retrieve Unique Fields for medicalRecords

```
> db.medicalRecords.aggregate([
      { $project: { keys: { $objectToArray: "$$ROOT" } } },
      { $group: { _id: "$keys.k" } }
  ]).forEach(function(doc) { print(doc._id); });
< [
    '_id',
    'patient_id',
    'name',
    'date_of_birth',
    'gender',
    'medical_conditions',
    'medications',
    'allergies',
    'last_appointment_date',
    'Tracker'
  ]
D597Task2Staging>
```

Figure 4B

Aggregation Pipeline to Retrieve Unique Fields for fitnessTrackers

```
> db.fitnessTrackers.aggregate([
    { $project: { keys: { $objectToArray: "$$ROOT" } } },
    { $group: { _id: "$keys.k" } }
]).forEach(function(doc) { print(doc._id); });
< [
    '_id',
    'Brand Name',
    'Device Type',
    'Model Name',
    'Color',
    'Selling Price',
    'Original Price',
    'Display',
    'Rating (Out of 5)',
    'Strap Material',
    'Average Battery Life (in days)',
    'Reviews'
  ]
```

The figures above show that medicalRecords have ten unique fields while fitnessTrackers have twelve. The aggregation pipeline can also check whether each document's average number of fields is the same as the unique number of fields. Figure 5A and Figure 5B show the queries that generate the average number of fields for the collections.

Figure 5

Average Field Count in Each Document in medicalRecords and fitnessTrackers Collection

```
> db.medicalRecords.aggregate([{
    $project: {
      key_count: {
        $cond: {
          if: {
            $isArray: {
              $objectToArray: "$$ROOT"
            }
          },
          then: {
            $size: {
              $objectToArray: "$$ROOT"
            }
          },
          else: 0
        }
      }
    }
  },
  {
    $group: {
      _id: null,
      avg_keys_per_document: {
        $avg: "$key_count"
      }
    }
  }
])
< {
  _id: null,
  avg_keys_per_document: 10
}
```

```
> db.fitnessTrackers.aggregate([{
    $project: {
      key_count: {
        $cond: {
          if: {
            $isArray: {
              $objectToArray: "$$ROOT"
            }
          },
          then: {
            $size: {
              $objectToArray: "$$ROOT"
            }
          },
          else: 0
        }
      }
    }
  },
  {
    $group: {
      _id: null,
      avg_keys_per_document: {
        $avg: "$key_count"
      }
    }
  }
])
< {
  _id: null,
  avg_keys_per_document: 12
}
```

The aggregate method returned the same average number of fields per document as the number of unique fields found in documents in each collection. We can assume that all the documents have the same keys and the same quantity of keys.

Figure 6 shows the find query used with $or operator to check for nulls across all the fields in the medicalRecords collection. The query did not return documents, meaning all document fields have non-null values.

Figure 6

Check for Nulls in medicalRecords

Figure 7A shows a similar find method script for the fitnessTrackers collection. However, it shows that several “Reviews” fields are null. No documents are returned when the “Reviews” field is removed from the $or operator array (Figure 7B). We can assume that “Reviews” are null for some documents for now, and the fields will be filled in with review counts once the “HealthTrack” platform is in use.

Figure 7A

Query Results Showing Null Reviews Fields

```
> db.fitnessTrackers.find({
    "$or": [
        { "Brand Name": null },
        { "Device Type": null },
        { "Model Name": null },
        { "Color": null },
        { "Selling Price": null },
        { "Original Price": null },
        { "Display": null },
        { "Rating (Out of 5)": null },
        { "Strap Material": null },
        { "Average Battery Life (in days)": null },
        { "Review": null }
    ]
})
< {
  _id: ObjectId('677c47e68a7119b458edfdd2'),
  'Brand Name': 'Xiaomi',
  'Device Type': 'FitnessBand',
  'Model Name': 'HMSH01GE',
  Color: 'Black',
  'Selling Price': '1,722',
  'Original Price': '2,099',
  Display: 'LCD Display',
  'Rating (Out of 5)': 3.5,
  'Strap Material': 'Leather',
  'Average Battery Life (in days)': 14,
  Reviews: ''
}
{
  _id: ObjectId('677c47e68a7119b458edfdd3'),
  'Brand Name': 'Xiaomi',
  'Device Type': 'FitnessBand',
  'Model Name': 'Smart Band 5',
  Color: 'Black',
  'Selling Price': '2,499',
  'Original Price': '2,999',
  Display: 'AMOLED Display',
  'Rating (Out of 5)': 4.1,
  'Strap Material': 'Thermoplastic polyurethane',
  'Average Battery Life (in days)': 14,
  Reviews: ''
}
```

Figure 7B

Query Without Review Field in Operator Returns No Null Documents

```
> db.fitnessTrackers.find({
    "$or": [
        { "Brand Name": null },
        { "Device Type": null },
        { "Model Name": null },
        { "Color": null },
        { "Selling Price": null },
        { "Original Price": null },
        { "Display": null },
        { "Rating (Out of 5)": null },
        { "Strap Material": null },
        { "Average Battery Life (in days)": null }
    ]
 })
<
D597Task2Staging>
```

When it comes to duplicates, we must check for duplicate documents, not duplicate fields. The focus is on multiple documents with identical copies of keys and values. Fields can have multiple instances (i.e., more than one person can have the name “John Smith” and have the gender male). The aggregate method can query the collection to search for duplicated field combinations across the documents. The $group stage groups the documents by a group key, a composite of multiple fields. If the query returns multiple documents, creating a composite group key by incrementing the $group stage one field at a time until no records are returned is logical.

Figure 8A demonstrates this method with the fields in the medicalRecords collection and the results. Figure 8B presents the script and results for the fitnessTrackers collection.

Figure 8A

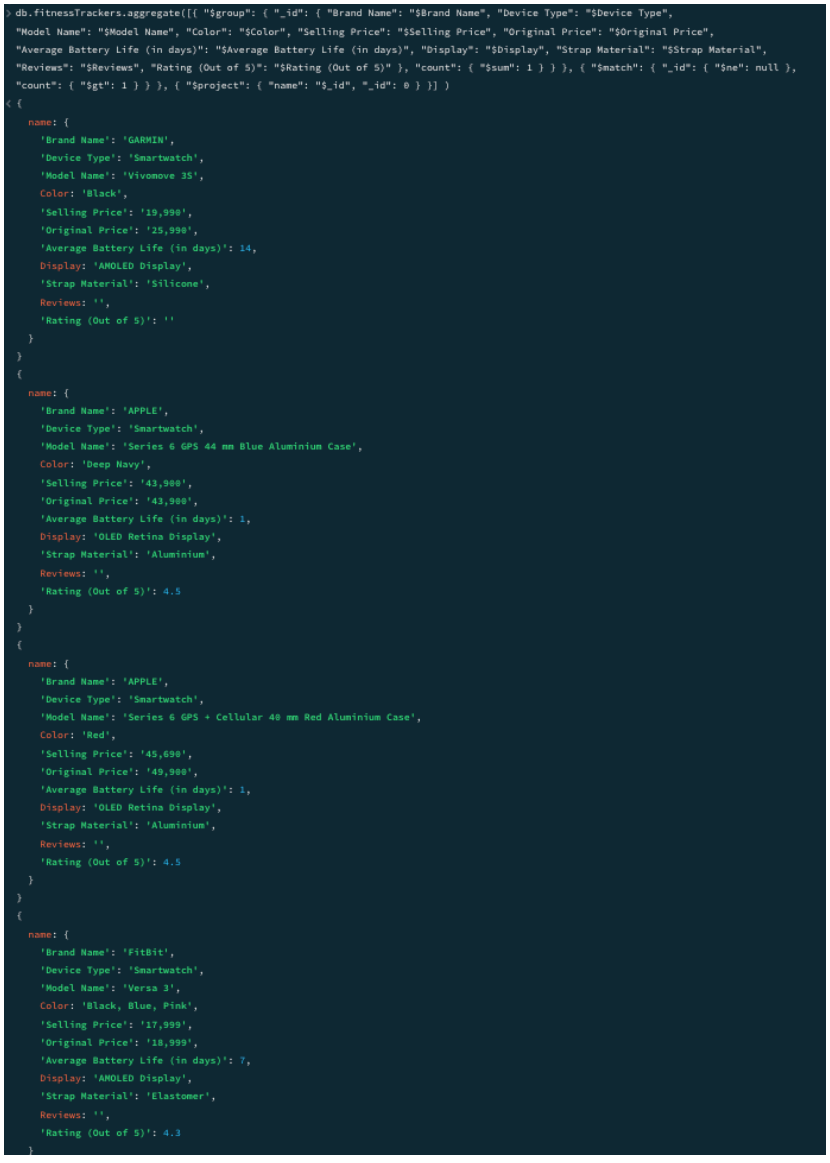
Aggregate Method for Checking Duplicated Documents in medicalRecords

```
> db.medicalRecords.aggregate([{ "$group": { "_id": { "name": "$name", "date_of_birth": "$date_of_birth", "gender": "$gender" },
  "count": { "$sum": 1 } } }, { "$match": { "_id": { "$ne": null }, "count": { "$gt": 1 } } }, { "$project": { "name": "$_id", "_id": 0 } }] )
<
D597Task2Staging>
```

The MongoDB shell did not return information on duplicate records because no duplicate documents have the same composite group key of name, date of birth, and gender. The composite fields confirm that duplicated documents are not in the medicalRecords collection.

Figure 8B

Aggregate Method for Checking Duplicated Documents in fitnessTrackers



The aggregate method returned a few duplicated documents, with all fields in the collection returned as group composite keys. A review of the duplicated documents confirms that they are exact copies of each other. We should delete the replicas so that only one record exists. Figure 9A shows the script for checking one of the duplicate documents, and Figure 9B shows the script that removes one of the records. Although the duplicated records have different ObjectIDs, they are exact copies of each other. Deleting them ensures that the database storage remains efficient.

Figure 9A

Review of Duplicated Documents Example: Garmin

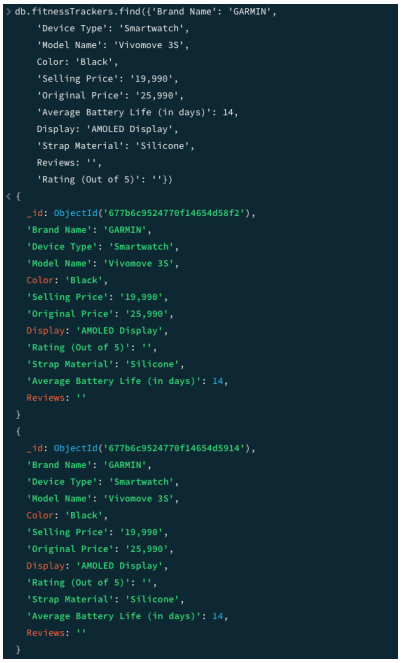


Figure 9B

Deleting Duplicated Documents Example: Garmin

```
> db.fitnessTrackers.deleteOne({"_id": ObjectId('677b6c9524770f14654d5914')})
< {
    acknowledged: true,
    deletedCount: 1
  }
> db.fitnessTrackers.find({'Brand Name': 'GARMIN',
      'Device Type': 'Smartwatch',
      'Model Name': 'Vivomove 3S',
      Color: 'Black',
      'Selling Price': '19,990',
      'Original Price': '25,990',
      'Average Battery Life (in days)': 14,
      Display: 'AMOLED Display',
      'Strap Material': 'Silicone',
      Reviews: '',
      'Rating (Out of 5)': ''})
< {
    _id: ObjectId('677b6c9524770f14654d58f2'),
    'Brand Name': 'GARMIN',
    'Device Type': 'Smartwatch',
    'Model Name': 'Vivomove 3S',
    Color: 'Black',
    'Selling Price': '19,990',
    'Original Price': '25,990',
    Display: 'AMOLED Display',
    'Rating (Out of 5)': '',
    'Strap Material': 'Silicone',
    'Average Battery Life (in days)': 14,
    Reviews: ''
  }
```

After removing all the duplicate documents, the aggregate method is executed again to ensure the database does not return duplicate records (Figure 10).

Figure 10

Confirm No Duplicated Records Exist for fitnessTrackers

```
> db.fitnessTrackers.aggregate([{ "$group": { "_id": { "Brand Name": "$Brand Name", "Device Type": "$Device Type",
  "Model Name": "$Model Name", "Color": "$Color", "Selling Price": "$Selling Price", "Original Price": "$Original Price",
  "Average Battery Life (in days)": "$Average Battery Life (in days)", "Display": "$Display", "Strap Material": "$Strap Material",
  "Reviews": "$Reviews", "Rating (Out of 5)": "$Rating (Out of 5)" }, "count": { "$sum": 1 } } }, { "$match": { "_id": { "$ne": null },
  "count": { "$gt": 1 } } }, { "$project": { "name": "$_id", "_id": 0 } }] )
<
D597Task2Staging>
```

We can perform the transformation part of ETL on the data by converting the data types of some fields within each collection. Some of the field values in the import process were uploaded with incorrect data types. For example, ‘date_of_birth’ should be a BSON date type,

not a string. After reviewing the fields' values using db.collection.distinct("fieldname"), we can determine the most logical data type to implement.

Additionally, neither collection matches their fields' naming convention. For consistency's sake, we can establish a camelCase convention for the fields in the collection. The camelCase convention aligns with JSON and JavaScript conventions, which makes data easier to parse for applications (*Best Practices for MongoDB Naming Conventions*, 2024). The fields in Table 4 demonstrate the field observations and changes that will take place.

Table 4

Fields to Change

Collection	Field	Changes
fitnessTrackers	"Brand Name"	Reformat the name to "brandName."
fitnessTrackers	"Device Type"	Reformat the name to "deviceType."
fitnessTrackers	"Model Name"	Reformat the name to "modelName."
fitnessTrackers	Color	Reformat the name to "color" and update the data type to an array.
fitnessTrackers	"Selling Price"	Reformat the name to "sellingPrice" and update the data type to double.
fitnessTrackers	"Original Price"	Reformat the name to "originalPrice" and update the data type to double.
fitnessTrackers	"Display"	Reformat the name to "display."
fitnessTrackers	"Ratings (out of 5)"	Reformat the name to "ratingOutOfFive" and update the data type to double.
fitnessTrackers	"Strap Material"	Reformat the name to "strapMaterial."
fitnessTrackers	"Average Battery Life (in days)"	Reformat the name to "avgBatteryLifeInDays" and update the data type to integer.

fitnessTrackers	“Reviews”	Reformat to “reviews” and update type to integer
medicalRecords	“patient_id”	Reformat to “patientId”
medicalRecords	“date_of_birth”	Reformat to “dateOfBirth” and update type to date
medicalRecords	“medical_conditions”	Reformat to “medicalConditions.”
medicalRecords	“medications”	Update type to boolean
medicalRecords	“last_appointment_date”	Reformat to “lastAppointmentDate” and update type to date
medicalRecords	“Tracker”	Reformat to “tracker”

Figures 11A and 11B show the query and results that update the field names in their respective collections. Figure 12A shows the script that updates the selected fields to their necessary data types in the medicalRecords collection. Figure 12B shows the script that updates selected fields to their necessary data types in the fitnessTrackers collection. See the physical model in Figure 1 to review the data types and reformatted names to be implemented in the permanent database.

Figure 11A

medicalRecords Fields Name Update

```
> db.medicalRecords.updateMany({},
    {
        $rename: {
            "patient_id": "patientId",
            "date_of_birth":"dateOfBirth",
            "medical_conditions":"medicalConditions",
            "last_appointment_date": "lastAppointmentDate",
            "Tracker": "tracker"
        }
    },
    false,
    true
);
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 100000,
    modifiedCount: 100000,
    upsertedCount: 0
}
```

```
> db.medicalRecords.aggregate([
    { $project: { keys: { $objectToArray: "$$ROOT" } } },
    { $group: { _id: "$keys.k" } }
]).forEach( function (doc) { print(doc._id) });
< [
    '_id',
    'name',
    'gender',
    'medications',
    'allergies',
    'dateOfBirth',
    'lastAppointmentDate',
    'medicalConditions',
    'patientId',
    'tracker'
]
```

Figure 11B

fitnessTrackers Fields Name Update and Confirmation

```
> db.fitnessTrackers.updateMany({},
    {
        $rename: {
            "Brand Name":"brandName",
            "Device Type":"deviceType",
            "Model Name": "modelName",
            "Color": "color",
            "Selling Price":"sellingPrice",
            "Original Price":"originalPrice",
            "Display": "display",
            "Rating (Out of 5)": "ratingOutOfFive",
            "Strap Material":"strapMaterial",
            "Average Battery Life (in days)":"avgBatteryLifeInDays",
            "Reviews": "reviews",
        }
    },
    false,
    true
);
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 561,
    modifiedCount: 561,
    upsertedCount: 0
}
```

```
> db.fitnessTrackers.aggregate([
    { $project: { keys: { $objectToArray: "$$ROOT" } } },
    { $group: { _id: "$keys.k" } }
]).forEach( function (doc) { print(doc._id) });
< [
    '_id',
    'avgBatteryLifeInDays',
    'brandName',
    'color',
    'deviceType',
    'display',
    'modelName',
    'originalPrice',
    'ratingOutOfFive',
    'reviews',
    'sellingPrice',
    'strapMaterial'
]
```

Figure 12A

medicalRecords Fields Data Type Update and Compass Schema Confirmation

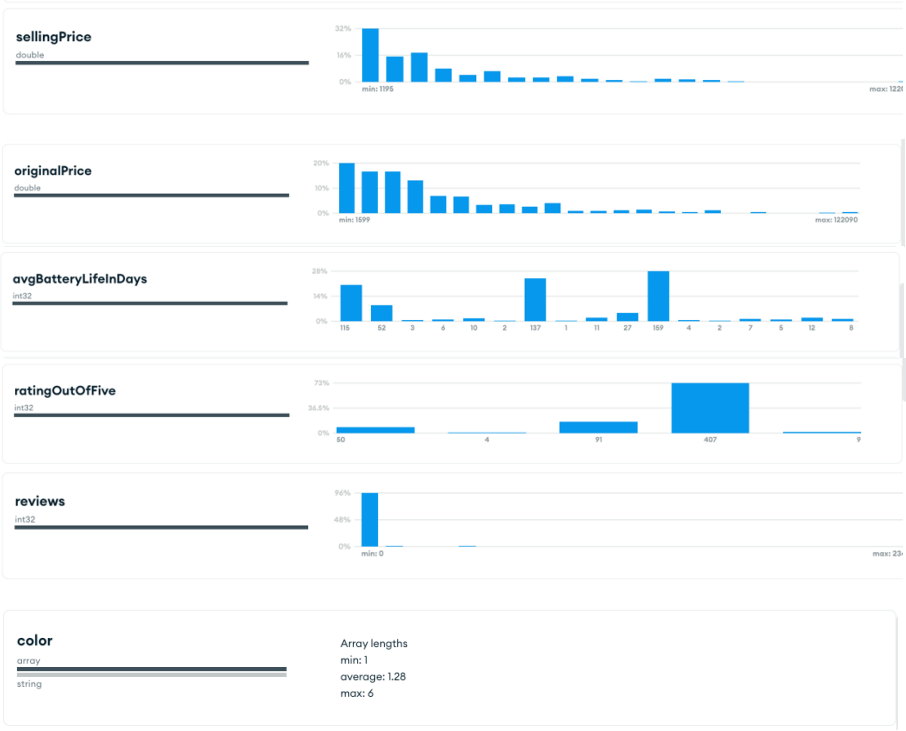
```
> db.medicalRecords.updateMany({},
    [
        { "$set": { "dateOfBirth": {"$toDate": "$dateOfBirth"}}},
        { "$set": { "lastAppointmentDate": {"$toDate": "$lastAppointmentDate"}}},
        { "$set": {
            medications: {
              $cond: {
                if: {
                  $eq: ["$medications", "Yes"]
                },
                then: true,
                else: false
              }
            }
          }
        },
    ]
);
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 100000,
  modifiedCount: 100000,
  upsertedCount: 0
}
```

Figure 12B

fitnessTrackers Fields Data Type Update and Compass Schema Confirmation

```
> db.fitnessTrackers.updateMany({},
    [
        { "$set": { "sellingPrice": {"$toDouble": "$sellingPrice"}}},
        { "$set": { "originalPrice": {"$toDouble": "$originalPrice"}}},
        { "$set": { "avgBatteryLifeInDays": {"$toInt": "$avgBatteryLifeInDays"}}},
        { "$set": {
            "ratingOutOfFive": {
              $cond: {
                if: {
                  $eq: ["$ratingOutOfFive", '']
                },
                then: 0,
                else: { "$toInt": "$ratingOutOfFive" }
              }
            }
          }
        },
    ]
);
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 561,
  modifiedCount: 561,
  upsertedCount: 0
}
```

```
> db.fitnessTrackers.updateMany({},
    [{ "$set":
        {
            "color": {
              $cond: {
                if: {
                  $isArray: "$color"
                },
                then: "$color",
                else: {
                  $split: ["$color", ", "]
                }
              }
            }
          }
    }]
);
< {
  acknowledged: true,
  insertedId: null,
  matchedCount: 561,
  modifiedCount: 561,
  upsertedCount: 0
}
```



Additional executed scripts for data type updates, such as further transformations on the reviews field, can be found in the file "ETL_Queries.js."

Figures 13A and 13B compare the updated field names and BSON data types to the model of each collection.

Figure 13A

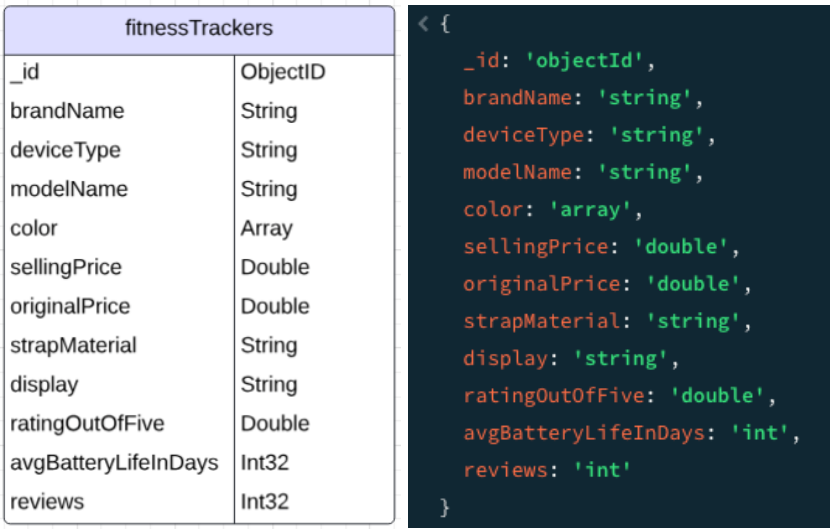
medicalRecords Model Comparison to Database Field Name Object

medicalRecords	
_id	ObjectID
patientId	Int32
name	String
dateOfBirth	Date
gender	String
medicalConditions	String
medications	Boolean
allergies	String
lastAppointmentDate	Date
tracker	String

```
{
  _id: 'objectId',
  patientId: 'int',
  name: 'string',
  dateOfBirth: 'date',
  gender: 'string',
  medicalConditions: 'string',
  medications: 'bool',
  allergies: 'string',
  lastAppointmentDate: 'date',
  tracker: 'string'
}
```

Figure 13B

fitnessTrackers Model Comparison to Database Field Name Object



fitnessTrackers	
_id	ObjectID
brandName	String
deviceType	String
modelName	String
color	Array
sellingPrice	Double
originalPrice	Double
strapMaterial	String
display	String
ratingOutOfFive	Double
avgBatteryLifeInDays	Int32
reviews	Int32

```
{
  _id: 'objectId',
  brandName: 'string',
  deviceType: 'string',
  modelName: 'string',
  color: 'array',
  sellingPrice: 'double',
  originalPrice: 'double',
  strapMaterial: 'string',
  display: 'string',
  ratingOutOfFive: 'double',
  avgBatteryLifeInDays: 'int',
  reviews: 'int'
}
```

D2. Database Mapping

There are several ways to map the cleaned data from the staging environment to the permanent database. The following figures utilize the graphical user interface tool called MongoDB Compass.

The screenshots in Figure 14A show the steps to create the D597 Task 2 database and the collection for medicalRecords. Figure 13B shows the query that imports the updated documents from the staging medicalRecords collection to the newly created collection. The operator $merge copies all the medicalRecords documents in the staging database into the permanent medicalRecords collection.

Figure 14A

Creating D597 Task 2 Database in Compass

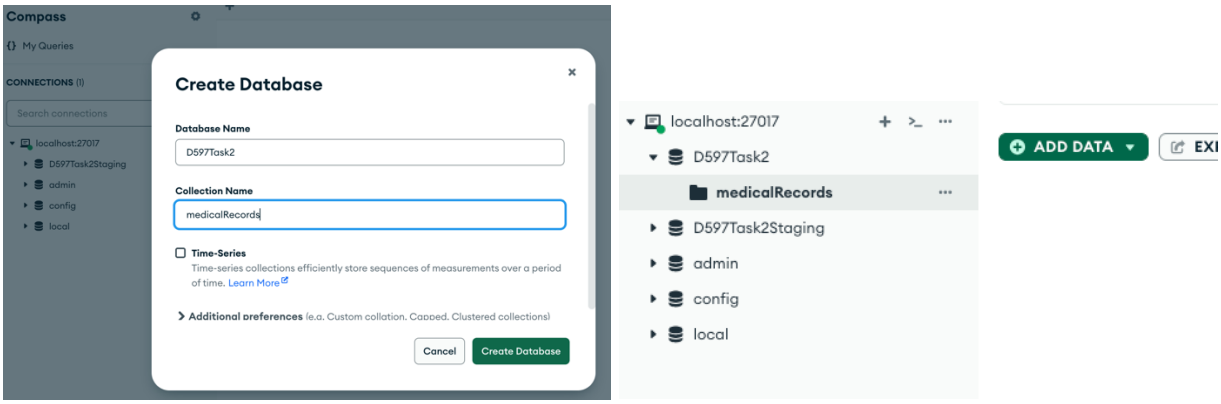


Figure 14B

Copying Staging Data to Permanent medicalRecords Collection

```
> use D597Task2Staging
< switched to db D597Task2Staging
> db.medicalRecords.aggregate([
        {
          $merge: {
            into: {
              db: "D597Task2",
              coll: "medicalRecords"
            }
          }
        }
]);
<
> use D597Task2
< switched to db D597Task2
> db.medicalRecords.countDocuments()
< 100000
```

Figures 15A and 15B demonstrate the creation of the permanent fitnessTrackers collection and import of staging data using the MongoDB shell in Compass. Figure 16 confirms that the staging database has been deleted.

Figure 15A

Creating Permanent fitnessTrackers Collection

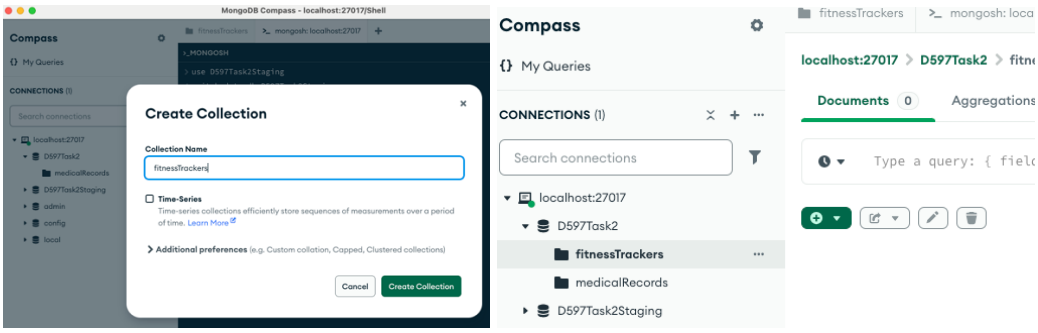


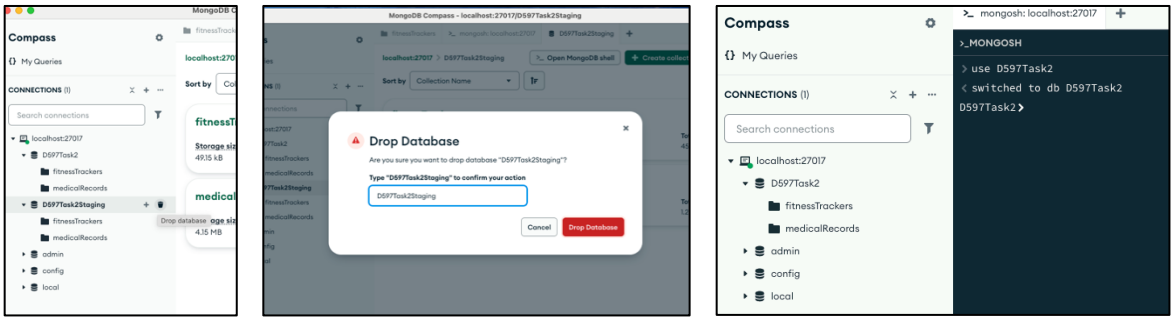
Figure 15B

Copying Staging Data to Permanent fitnessTrackers Collection

```
> use D597Task2Staging
< switched to db D597Task2Staging
> db.fitnessTrackers.aggregate([
        {
          $merge: {
            into: {
              db: "D597Task2",
              coll: "fitnessTrackers"
            }
          }
        }
]);
<
> use D597Task2
< switched to db D597Task2
> db.fitnessTrackers.countDocuments()
< 561
```

Figure 16

Drop Staging Database Using Compass



D3. Business Problem Queries

Let us revisit the business problem verbatim: How can HealthFit Innovations leverage its data to provide up-to-date and relevant health insights to their patient, clinicians, and business-owner end-users? HealthFit Innovations' objective is to empower patient users to foster more influence over their health data. Additionally, the platform should also give healthcare provider users access to electronic health records to provide improved personalized care.

The approach for each query in this section addresses and explores scenarios of differing platform users: a patient, a clinician, and an organization member such as a product manager.

The situations below guide the following queries based on the user's role:

1. A clinician wants to establish a pattern between the wearable device type and the medical condition's severity to recommend specific wearables to future patients with similar profiles.
    - *Query Details: What are the most popular trackers for women with a medical condition status of "Watch"?*

2. A patient wants to add additional health details, such as allergies, to their profile to provide clinicians with additional health information for optimized care.
    - *Query Details: Update a specific medicalRecords document by looking up a patient ID and inserting additional values to the allergies field.*

3. A product manager wants to find a pattern between device attributes within each brand and average rating to make better-informed inventory selections.
    - *Query Details: What are the most highly rated brands (above average rating of 4) based on device type?*

The scripts for these situational queries are in the "Business_Queries.js."

The query for Question 1 should provide a list of the top ten most popular trackers owned by female patients with a medical condition status of "Watch." Figure 17 shows the executed code and results.

Figure 17

Question 1 Query and Results

```
> db.medicalRecords.aggregate([
    {
      "$match": {
        "gender": "F",
        "medicalConditions": "Watch"
      }
    },
    {
      "$group": {
        "_id": "$tracker",
        "count": {
          "$sum": 1
        }
      }
    },
    {
      "$sort": {
        "count": -1
      }
    },
    {
      "$limit": 10
    }
  ])
```

```
< {
    _id: 'Band 5',
    count: 1378
  }
  {
    _id: 'Amazfit Bip',
    count: 1078
  }
  {
    _id: 'Amazfit Verge',
    count: 927
  }
  {
    _id: 'Amazfit Bip S',
    count: 903
  }
  {
    _id: 'Band 3',
    count: 881
  }
  {
    _id: 'Band 5i',
    count: 878
  }
  {
    _id: 'Amazfit GTS',
    count: 866
  }
  {
    _id: 'Band 4',
    count: 844
  }
  {
    _id: '41mm',
    count: 489
  }
  {
    _id: 'S Pro',
    count: 475
  }
```

A clinician looking to support a patient with a similar profile can use this list as a suggestion of devices to use for enhanced self-monitoring.

One primary goal for “HealthTrack” is to increase patient interaction with their medical information. The query for Question 2 works as the database-side command aligned with a patient-user's actions on the platform. Suppose a user, Peter Garcia, finds out they are allergic to peanuts and milk - in addition to a current allergy to eggs - after seeing an allergist in another organization. Peter wants to update their profile on “HealthTrack” to inform other clinicians of the new peanuts and milk allergy. Clinicians must know updated allergy information to avoid detrimental medication reactions.

Although the user will only interact with the platform interface, this type of update requires a MongoDB query to make changes in the database. Figure 18 shows the query and

results that add to the allergies field. The update transforms the allergies field into an array to capture the added information, leveraging NoSQL flexible capabilities.

Figure 18

Question 2 Query and Results

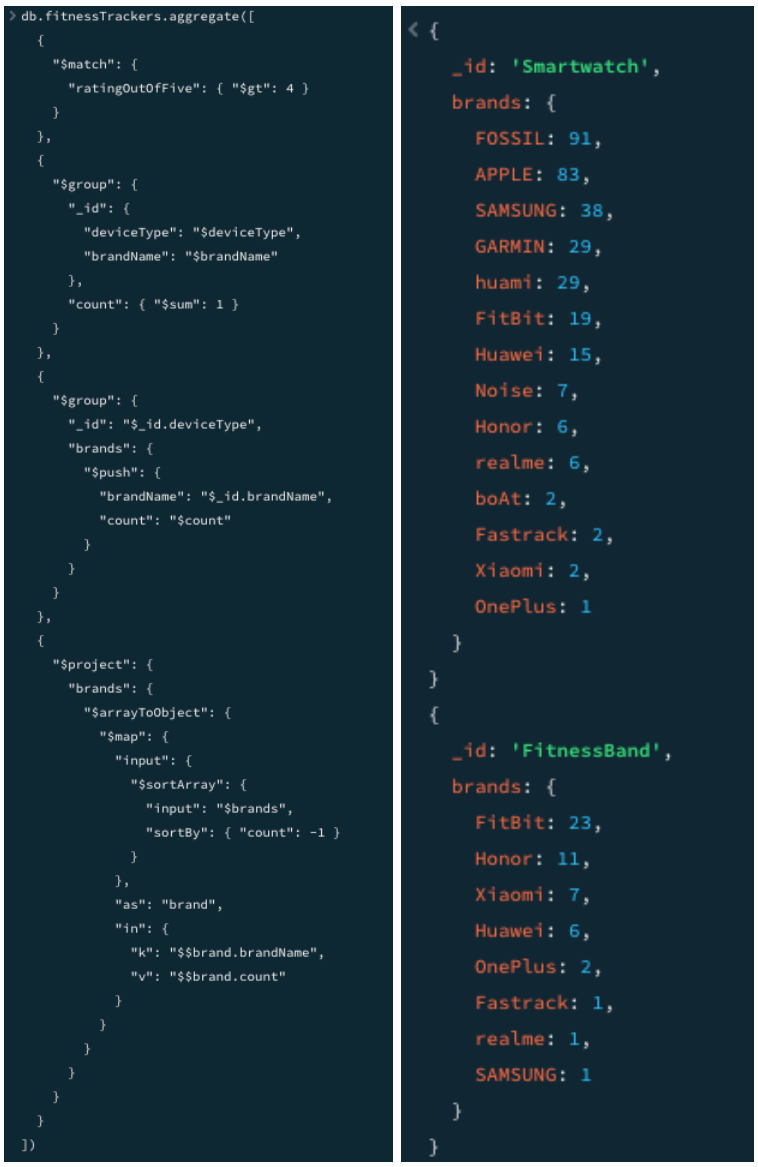
```
> db.medicalRecords.aggregate([
    {
      $match: {
        patientId: 965,
      },
    },
    {
      $set: {
        allergies: {
          $concatArrays: [
            {
              $split: ["$allergies", ", "],
            },
            ["milk", "peanuts"],
          ],
        },
      },
    },
  ]);
< {
    _id: ObjectId('677df3720fb353a1dfe353dd'),
    allergies: [
      'egg',
      'milk',
      'peanuts'
    ],
    dateOfBirth: 1981-05-21T00:00:00.000Z,
    gender: 'F',
    lastAppointmentDate: 2022-06-14T00:00:00.000Z,
    medicalConditions: 'None',
    medications: false,
    name: 'Peter Garcia',
    patientId: 965,
    tracker: 'Amazfit Bip S Lite'
  }
```

Business members such as product managers can gain insights to manage inventory with data-driven decision-making. Question 3 addresses how a product manager can look at the highest-rated products (above an average rating of 4) to optimize inventory and anticipate profitability efficiently. The query in Figure 19 counts all brands with an average rating above four and groups the counts by device type, sorted by the counts. FOSSIL is the highest-rated device among smartwatches, while FitBit is the most well-received among fitness bands. Based

on this information, a product manager can make well-informed inventory choices and marketing suggestions.

Figure 19

Question 3 Query and Results



```
> db.fitnessTrackers.aggregate([
  {
    "$match": {
      "ratingOutOfFive": { "$gt": 4 }
    }
  },
  {
    "$group": {
      "_id": {
        "deviceType": "$deviceType",
        "brandName": "$brandName"
      },
      "count": { "$sum": 1 }
    }
  },
  {
    "$group": {
      "_id": "$_id.deviceType",
      "brands": {
        "$push": {
          "brandName": "$_id.brandName",
          "count": "$count"
        }
      }
    }
  },
  {
    "$project": {
      "brands": {
        "$arrayToObject": {
          "$map": {
            "input": {
              "$sortArray": {
                "input": "$brands",
                "sortBy": { "count": -1 }
              }
            },
            "as": "brand",
            "in": {
              "k": "$$brand.brandName",
              "v": "$$brand.count"
            }
          }
        }
      }
    }
  }
])
```

```
< {
    _id: 'Smartwatch',
    brands: {
      FOSSIL: 91,
      APPLE: 83,
      SAMSUNG: 38,
      GARMIN: 29,
      huami: 29,
      FitBit: 19,
      Huawei: 15,
      Noise: 7,
      Honor: 6,
      realme: 6,
      boAt: 2,
      Fastrack: 2,
      Xiaomi: 2,
      OnePlus: 1
    }
  }
  {
    _id: 'FitnessBand',
    brands: {
      FitBit: 23,
      Honor: 11,
      Xiaomi: 7,
      Huawei: 6,
      OnePlus: 2,
      Fastrack: 1,
      realme: 1,
      SAMSUNG: 1
    }
  }
```

The results above return a list of brands grouped by device type. The count values are the number of times these brands appeared in documents with an average rating above 4.

D4. Optimization Techniques

Optimization is important to ensure the database returns its results most efficiently (MongoDB, n.d.-g). Indexes are a straightforward optimization solution that can make databases more performant. A database index provides references for the database based on fields for quick access. Suppose a database does not implement an index. In that case, it will read each document in the collection. Scanning each document can be systematically cumbersome with hundreds of thousands of documents. Applying indexes on relevant fields will improve execution times for the queries by allowing the database to quickly reference the index and retrieve only the relevant documents, minimizing execution time.

Several index types exist to address specific queries and data types in MongoDB. The queries in this section use single and compound indexes. A single index is a basic index with one field. A compound index collects and sorts data using two or more fields, ordering the sorted groups by the order of fields in the index (Team, n.d.-a).

Table 5 below shows the indexes created on applicable fields for each query.

Table 5

Query Indexes

Collection	Fields	Index Type	Query
medicalRecords	gender medicalConditions	Compound	1
medicalRecords	patientId	Single	2
fitnessTrackers	ratingOutOfFive	Single	3

The explain button in the Aggregations tab opens a window to the explain plan showing execution times. Figure 20A shows the screenshot of the execution time for Query 1. Figure 20B shows the execution time for Query 1 using a compound index.

Figure 20A

Query 1 Explain Plan Without Index Optimization

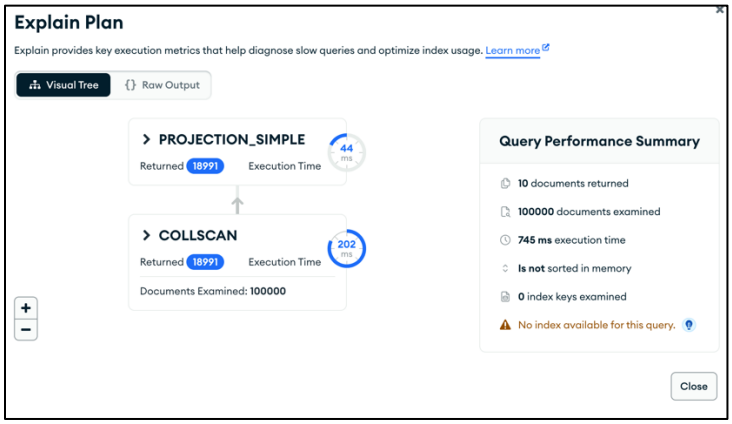
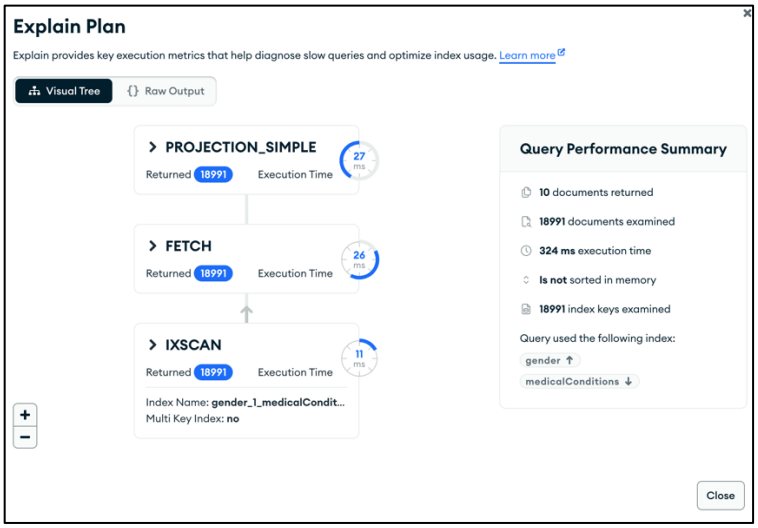


Figure 20B

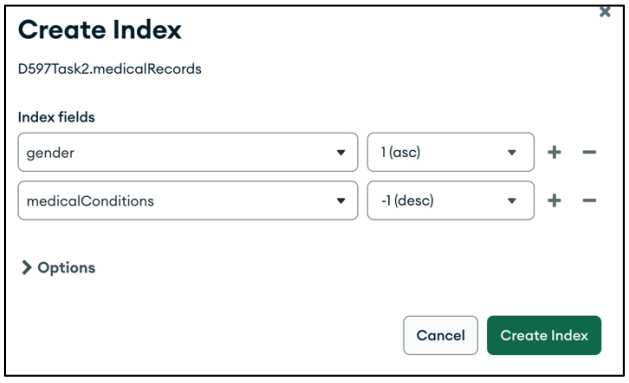
Query 1 Explain Plan With Compound Index Optimization



According to the query performance summary, the unindexed query took 421 milliseconds longer to execute than the compound indexed query. Note that all 100,000 documents had to be examined with the unindexed query. In contrast, only 18,991 documents were scanned in the indexed query. A good strategy in building the compound index is understanding the general order of operations in how the database should read the collection. Since the query is filtering by gender and medical conditions, we can manipulate the sort order of the field values so that the database can reference the index quickly. Figure 21 below demonstrates how to create a compound index in MongoDB Compass.

Figure 21

Creating A Compound Index



Figures 22A – 23B show the other two queries' unindexed and optimized explain plans.

Figure 22A

Query 2 Unindexed Explain Plan

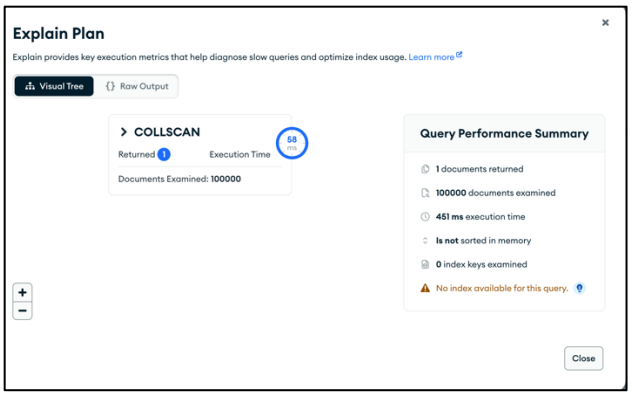
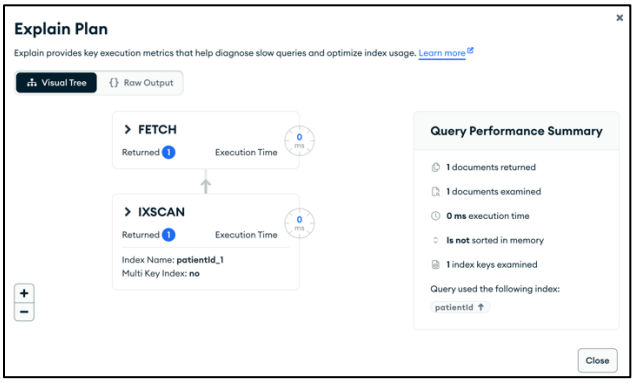


Figure 22B

Query 2 Indexed Explain Plan



The COLLSCAN in the unindexed query is less performant than the indexed query because it must scan every document in the collection to find the patient ID. In Figure 22B, the database retrieves all the documents without scanning. It references the index for the patient ID to return only one document. Thus, the indexed query is much faster.

Figure 23A

Query 3 Unindexed Explain Plan

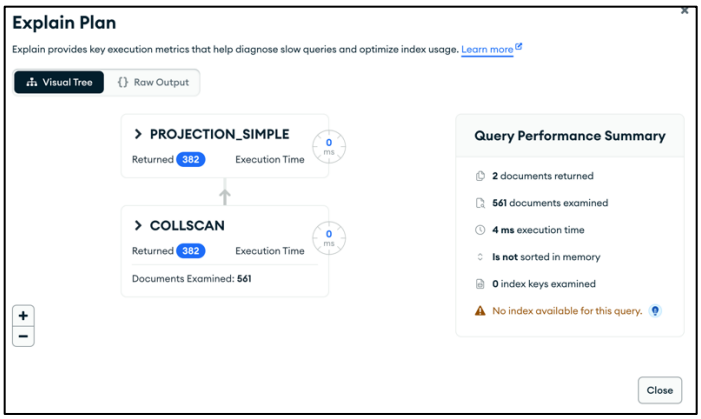
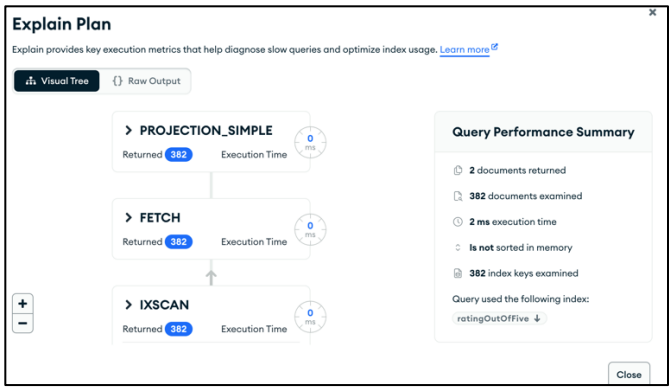


Figure 23B

Query 3 Indexed Explain Plan

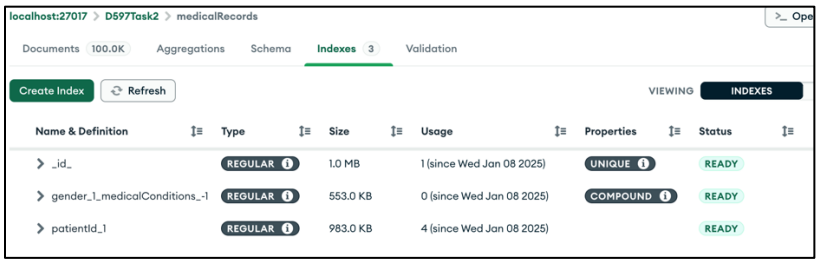


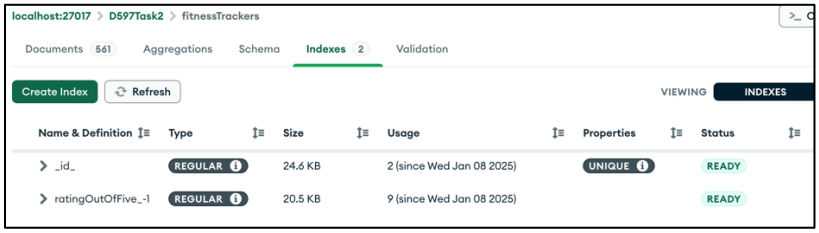
Although, the difference between the two explain plans for Query is minimal, the indexed query is still more efficient by two milliseconds. It also read fewer documents than the unindexed query.

Figure 24 shows the indexes created for both collections.

Figure 24

Query Indexes for medicalRecords and fitnessTrackers





The three queries in this section demonstrate how MongoDB's flexible, NoSQL capabilities can be invaluable to a platform like "HealthTrack" that must accommodate diverse and wide-ranging user groups. By addressing the challenges of different data structures, evolving data, and various user needs, HealthFit Innovations can create custom user experiences without being limited by strict SQL data schemas. NoSQL's adaptability allows the platform to scale efficiently and remain user-friendly regardless of the user segment. The platform's broad capabilities will lend itself to the organization's success in managing its data as it grows to big data.

Part 3: Resources

F. Resources

1. Aqsa. (2024, May 9). *Get all field names in MongoDB*. Spark by {Examples}. https://sparkbyexamples.com/mongodb/get-all-field-names-in-mongodb/

2. Babitz, K. (2024, December 10). *How to maintain data security*. www.datacamp.com. Retrieved December 12, 2024, from https://www.datacamp.com/blog/how-to-maintain-data-security
3. *Best Practices for MongoDB naming Conventions*. (2024, December 16). W3resource. https://www.w3resource.com/mongodb/snippets/best-practices-for-mongodb-naming-conventions.php#:~:text=in%20larger%20databases.-,Field%20Names%3A,and%20work%20with%20in%20applications.
4. Coronel, C., & Morris, S. (2022). *Database systems*.
5. *Create a MongoDB database*. (n.d.). MongoDB. https://www.mongodb.com/resources/products/fundamentals/create-database
6. Daniel. (2023, October 30). *ETL or Extract Transform Load: Definition and use*. Data Science Courses | DataScientest. https://datascientest.com/en/etl-or-extract-transform-load-definition-and-use
7. Gamal, A., Barakat, S., & Rezk, A. (2020). Standardized electronic health record data modeling and persistence: A comparative review. *Journal of Biomedical Informatics*, *114*, 103670. https://doi.org/10.1016/j.jbi.2020.103670
8. Jenkins, J. (2022, December 16). *How you need to think differently - Learning MongoDB* [Video]. LinkedIn. https://www.linkedin.com/learning/learning-mongodb-17360744/how-you-need-to-think-differently?seekTo=18&u=2045532
9. Medlock, J. (2021, December 7). How to ETL with MongoDB & Postgres (Part 3) - Chingu - Medium. *Medium*. https://medium.com/chingu/how-to-etl-with-mongodb-postgres-part-3-

4aeb95c2cd3a#:~:text=The%20goal%20of%20an%20extract,steps%20in%20the%20ETL%20process.

10. MongoDB. (n.d.-a). *7 Best Practices for MongoDB Security*. https://www.mongodb.com/resources/products/capabilities/best-practices
11. MongoDB. (n.d.-b). *Database scaling*. https://www.mongodb.com/resources/basics/scaling
12. MongoDB. (n.d.-c). *Document Database - NoSQL*. https://www.mongodb.com/resources/basics/databases/document-databases#:~:text=Documents%20store%20data%20in%20field,JSON%2C%20BSON%2C%20and%20XML.
13. MongoDB. (n.d.-d). *Embedding MongoDB documents for ease and performance*. https://www.mongodb.com/resources/products/fundamentals/embedded-mongodb
14. MongoDB. (n.d.-e). *Explaining BSON with examples*. https://www.mongodb.com/resources/languages/bson
15. MongoDB. (n.d.-f). *How to design schema for NoSQL data Models*. https://www.mongodb.com/resources/basics/databases/nosql-explained/data-modeling
16. MongoDB. (n.d.-g). *What is a database index? | Examples of indices | MongoDB*. https://www.mongodb.com/resources/basics/databases/database-index#:~:text=What%20is%20Indexing%20in%20a%20Database%3F&text=A%20good%20indexing%20strategy%20is,patterns%2C%20and%20database%20server%20configuration.

17. MongoDB. (n.d.-h). *When to use NoSQL Databases*. https://www.mongodb.com/resources/basics/databases/nosql-explained/when-to-use-nosql

18. MongoDB. (n.d.-i). *Why use MongoDB and when to use it?* https://www.mongodb.com/resources/products/fundamentals/why-use-mongodb

19. Team, M. D. (n.d.). *Time series*. MongoDB Manual v8.0. https://www.mongodb.com/docs/manual/core/timeseries-collections/