Communications of the Association for Information Systems

Volume 53 Paper in press

2023

Navigating Workload Compatibility Between a Recommender System and a NoSQL Database: An Interactive Tutorial

Varol O. Kayhan
School of Information Systems and Management Muma College of Business University of South Florida

Donald J. Berndt
School of Information Systems and Management Muma College of Business University of South Florida

Communications of the
Association for Information Systems

Accepted Manuscript

Navigating Workload Compatibility Between a Recommender System and a NoSQL Database: An Interactive Tutorial

Varol O. Kayhan
School of Information Systems and Management
Muma College of Business
University of South Florida

Donald J. Berndt
School of Information Systems and Management
Muma College of Business
University of South Florida

Please cite this article as: Kayhan, V. O., & Berndt, D. J. (in press). Navigating workload compatibility between a recommender system and a NoSQL database: An interactive tutorial. *Communications of the Association for Information Systems.*

Communications of the Association for Information Systems

Education Paper ISSN: 1529-3181

Navigating Workload Compatibility Between a Recommender System and a NoSQL Database: An Interactive Tutorial

Varol O. Kayhan
School of Information Systems and Management
Muma College of Business
University of South Florida

Donald J. Berndt
School of Information Systems and Management
Muma College of Business
University of South Florida

Abstract:

In this tutorial, the issue of compatibility between a big data storage technology and an analytic workload is explored using a fictitious streaming company as an example. The tutorial offers an interactive approach to help students understand the importance of considering workload compatibility when adopting new technologies. We provide instructors with two Jupyter Notebooks that analyze the compatibility, a detailed instructor guide on how to execute these notebooks, lessons learned, and appendices containing solutions and explanations. This tutorial provides a valuable resource for instructors teaching courses in database systems, big data, and analytic concepts, helping students develop practical skills to navigate the complexities of big data technologies effectively.

Keywords: NoSQL, MongoDB, Recommender System, Collaborative Filtering, Cosine Similarity.

[Department statements, if appropriate, will be added by the editors. Teaching cases and panel reports will have a statement, which is also added by the editors.]

[Note: this page has no footnotes.]

This manuscript underwent [editorial/peer] review. It was received xx/xx/20xx and was with the authors for XX months for XX revisions. [firstname lastname] served as Associate Editor.] **or** The Associate Editor chose to remain anonymous.]

1 Introduction

The adoption of big data technologies is rapidly increasing among organizations (Watson, 2019). According to a recent report by Statistica (2022), the global big data market is expected to nearly double from $56 billion in 2020 to $103 billion in 2027. These technologies enable organizations to handle large **volumes** of data, generate **value** by capturing patterns, manage a **variety** of data formats, process and serve data at high **velocity**, and ensure data **veracity** (Anuradha, 2015). These are commonly known as the five V's of big data technologies. Despite the anticipated benefits, the implementation of big data technologies can be fraught with challenges (Wang & He, 2016; Watson, 2019). One of the most common obstacles is that organizations may not be able to change their mindsets overnight, leading them to replace legacy systems with big data technologies without considering the compatibility between their current workloads and the big data technologies that will support them (Bean, 2020; Ramasamy, 2019). Therefore, organizations may not see any tangible results such as improved efficiency, increased revenue, or higher productivity despite making significant investments in big data technologies (Sharma, 2022). This lack of return on investment can be frustrating, especially considering the substantial costs associated with implementing these technologies. Therefore, it is crucial for organizations to take the time to assess the compatibility of their current workloads with big data technologies to ensure smooth and successful digital transformations.

In this tutorial, we explore this issue and examine the compatibility between an analytic workload and a big data technology using the example of a fictitious streaming company that provides online content to its subscribers. The company invests in a NoSQL (or non-relational) database, which is a technology often associated with big data, to manage its growing customer base and store more data. However, the data scientist of this company intends to use the same technology for an analytic workload that generates content recommendations for subscribers. As we demonstrate later, an incompatibility between the recommender system and the NoSQL database can lead to significant inefficiencies, jeopardizing the company's competitiveness in the industry. This scenario underscores the importance of considering workload compatibility when adopting big data technologies in an organization.

This tutorial offers a valuable resource for instructors teaching courses in database systems, big data, and analytic concepts. The tutorial's interactive approach enables instructors to engage students in discussions of compatibility between workloads and technologies using hands-on materials. The hands-on materials comprise three document databases with varying designs, all of which are freely accessible on the cloud. Additionally, there are two *Jupyter Notebooks*[1] available on GitHub. The interactive nature of the materials provides instructors with an opportunity to equip students with the knowledge and skills necessary to navigate the complexities of using big data technologies for analytic workloads.

This tutorial is designed to help students achieve the following learning objectives: 1) contrast a relational database with a document database; 2) explain collaborative filtering-based recommender systems, which are among the most widely used recommender systems in online services; 3) contrast three types of designs that can be implemented in a document database; and 4) analyze the query speed of different document database designs when making collaborative filtering-based recommendations. To attain these objectives, students should possess the following prerequisite knowledge: intermediate-level knowledge of relational databases, introductory-level knowledge of document databases, and introductory-level knowledge of *Jupyter Notebook* and Python technologies.

In the rest of this paper, we first delve into collaborative filtering-based recommender systems. Then, we explore document database concepts and discuss various designs that can be implemented in a document database, in contrast with a relational database. We then proceed to discuss the data used in this tutorial and present the issue faced by the fictitious company. After providing a detailed instructor guide on how to run the tutorial, we share the lessons learned from using this tutorial in multiple graduate-level courses over a two-year period. Appendix A contains the solution to the compatibility issue raised in the tutorial.

[1] Italicized terms are defined in Appendix B Table B1.

2 Recommender Systems

In our daily lives, we often rely on the recommendations of others because we may not have enough information about the alternatives from which we have to choose. In the digital age, this social process is performed seamlessly using recommender systems that match recommenders with those seeking recommendations (Resnick & Varian, 1997). Some widely used approaches, including the so-called collaborative filtering-based systems, rely on measuring similarity between individuals (Goldberg, Nichols, Oki, & Terry, 1992). It is important to note that these systems allow individuals to switch roles dynamically. One person can be a recommender in one instance and a seeker in another by virtue of algorithms that identify individuals with similar preferences. Therefore, a collaborative filtering-based system predicts the alternative (also known as an item) that an individual may prefer based on how similar this individual is to another. Once two individuals are matched based on their similarity, the system can recommend items preferred by one to the other.

It is worth noting that recommender systems can match individuals with not only other individuals but also items. These types of recommenders are known as item-based systems because they make recommendations based on the attributes of the items. For example, if a user enjoys watching science-fiction movies, an item-based system starts recommending other science-fiction movies because it establishes a match based on the genre attribute. While both item-based and collaborative filtering-based systems have their advantages and disadvantages, collaborative filtering-based systems have gained popularity in recent years due to their effectiveness. Streaming platforms, for instance, heavily rely on collaborative filtering-based systems because of their ability to enhance user engagement, reduce customer churn, and increase subscription-based revenues (Hinkle, 2021). Therefore, in this tutorial, we focus specifically on collaborative filtering-based recommender systems.

A collaborative filtering-based system relies on a *utility matrix* that captures user-item preferences. In this matrix, users are represented by the rows while items are represented by the columns. Each user-item pair is assigned a metric such as a binary value indicating whether the user interacted with the item. However, this binary metric does not convey whether the user enjoyed the item. To provide more accurate recommendations, it is preferable to capture the user's rating of the item on a numerical scale, such as one to five stars. Table 1 presents an example *utility matrix* based on the movie ratings of four users for five movies using this numerical scale. Note that empty cells indicate that the user has not rated that movie.

Table 1. Example Utility Matrix to Capture user Ratings for Movies

	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5
User 1	5	1			4
User 2		3	2	2	
User 3	1		5	1	
User 4	5	1	5		5

A typical *utility matrix* has many blank cells indicating missing user-item preference data. The aim of collaborative filtering is to predict these missing values based on the information that is already available in the matrix. To achieve this, the similarity between each pair of users must be calculated, and the most similar users must be identified. For instance, if User A is very similar to User B, then recommendations for User A can be derived by observing User B's preferences (and vice versa). Table 1 presents an example *utility matrix*, which reveals that User 1 and User 4 share similar movie ratings. If no other user is more similar to User 1, then Movie 3 could be recommended to User 1 because it is highly rated by User 4.

To systematically identify the most similar users, pairwise user similarities must be computed. One widely used similarity metric is *cosine similarity*, which calculates the cosine of the angle between two user preference vectors using Equation 1, where A and B denote the preference vectors of two users. By computing all pairwise *cosine similarities* between users in a *utility matrix*, users who are most similar can be identified. Positive and higher *cosine similarity* values indicate that the angle between two users is smaller, which implies a higher degree of similarity between them. In other words, the larger the *cosine similarity* value, the greater the similarity between the users.

$$\frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}=\frac{\sum_{i=1}^{n}A_iB_i}{\sqrt{\sum_{i=1}^{n}A_i^2}\sqrt{\sum_{i=1}^{n}B_i^2}} \quad (1)$$

We can use Equation 1 to compute the *cosine similarity* between User 1 and User 4 as 0.814 (as shown below). It is important to note that when computing *cosine similarity*, missing values in the *utility matrix* are treated as zeros. However, this does not necessarily mean that a user's preference for a particular item is zero, but rather that the preference for that user-item pair is unknown.

$$\text{Cosine similarity between User 1—User 4} = \frac{(5x5)+(1x1)+(0x5)+(0x0)+(4x5)}{\left(\sqrt{5^2+1^2+0^2+0^2+4^2}\right)x\left(\sqrt{5^2+1^2+5^2+0^2+5^2}\right)}=0.814$$

Similarly, the *cosine similarity* between User 1 and User 2 can be calculated as 0.112 as follows:

$$\text{Cosine similarity between User 1—User 2} = \frac{(5x0)+(1x3)+(0x2)+(0x2)+(4x0)}{\left(\sqrt{5^2+1^2+0^2+0^2+4^2}\right)x\left(\sqrt{0^2+3^2+2^2+2^2+0^2}\right)}=0.112$$

The above computations illustrate that User 1 is more similar to User 4 than to User 2, as evident from their *cosine similarity* values. By computing pairwise similarities between all users, we can identify the users who are most similar to each other. In this example, User 1 is most similar to User 4. Based on this, we can recommend items to User 1 by looking for the items that are highly rated by User 4 but not yet rated by User 1 (or vice versa). For instance, Movie 3 is rated 5 out of 5 by User 4, but it has no rating from User 1. Therefore, Movie 3 can be recommended to User 1.

In summary, collaborative filtering involves constructing a *utility matrix*, computing pairwise *cosine similarities*, and analyzing the preferences of the top-*n* most similar users to provide recommendations. These are analytical tasks that do not require a particular storage technology. The *utility matrix* can be constructed using the application programming interface (API) of data analysis and manipulation libraries such as `Numpy` or `Pandas`. However, the data needs to be retrieved from a database using a query. In the case of a streaming service, the data may be stored in a transactional database that manages users, movies, and ratings to support the core processes of the service. As databases can get very large, organizations might opt for big data technologies like a NoSQL database, which can store vast amounts of data in a parallel and highly scalable architecture. Therefore, in the next section, we delve into the document database concepts, a particular type of NoSQL database.

3 Document Databases

NoSQL databases are considered big data technologies that offer several benefits over relational databases with respect to data storage, management, and availability. In particular, document databases such as MongoDB provide the flexibility of schema-free design with the scalability of a distributed computing infrastructure. From a data storage perspective, they use horizontal scaling to handle large volumes of data without sacrificing performance. From a data management perspective, some NoSQL databases are schema-less such that they can not only store structured, semi-structured, and unstructured data, but also allow changing the database schema as needed by adding new fields or entities without sophisticated migration procedures. Further, they support popular programming languages through APIs, making them developer-friendly for managing the data. Finally, NoSQL databases are an ideal choice for ensuring data availability, particularly in critical applications, as they are predominantly cloud-based and experience little to no downtime.

There are four main types of NoSQL databases: document databases, key-value stores, graph databases, and column-oriented databases. In this tutorial, we focus specifically on document databases because of their popularity. In a document database, data is stored in *collections*, which consist of multiple *documents*. Compared to a relational database, a *collection* corresponds to a table and a *document* corresponds to a row. Each *document* is stored using key-value pairs (sections of the *document*) following the JavaScript Object Notation (JSON) standard. For example, a MOVIES table with *m* rows and *n* columns in a relational database, as shown in Figure 1 (a), might correspond to a MOVIES *collection* with

m documents in a document database as shown in Figure 1 (b). Each *document* may consist of *n* key-value pairs, although some *documents* may have more or fewer key-value pairs depending on the data being captured. This flexibility allows for a *collection* to have a variable schema, unlike a table in a relational database that has a rigid schema. *Documents* in a *collection* can have as many key-value pairs as needed, making it easy to accommodate changes in data requirements without the need for complex schema modifications or migrations.

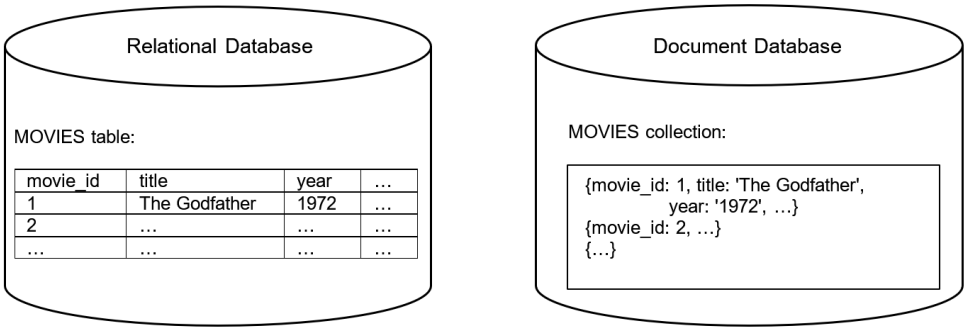

Figure 1. Comparison Between Relational Databases and Document Databases

Document databases can have multiple *collections*, like a relational database with multiple tables. These *collections* can be independent of each other or have relationships through foreign keys. For instance, consider the example of adding a RATINGS *collection* to the document database shown in Figure 1 (b) to keep track of each movie's ratings. By including a foreign key, such as movie_id in the RATINGS *collection*, it becomes possible to identify every rating of a particular movie in the database. Figure 2 provides an illustration of this scenario. In this way, document databases can entertain complex data models.

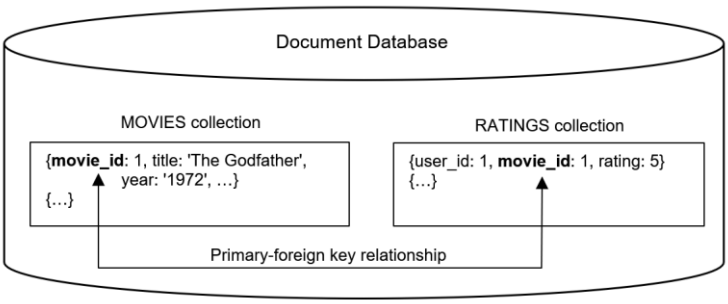

Figure 2. Relationships in a Document Database

One of the key benefits of a document database is its flexibility in handling relationships. In addition to storing related data in separate *collections*, a document database also allows for combining multiple collections into a single collection by nesting *documents* within other *documents*, which are referred to as *subdocuments*. This allows for hierarchical arrangements and embedding one *collection* in another. For example, in a scenario where a movie can have many ratings, rather than creating a separate RATINGS *collection* and creating a foreign key relationship, the ratings can be converted into an array of *subdocuments* and embedded in the MOVIES collection. This helps store the one-to-many relationship between movies and ratings in the same *collection,* simplifying data retrieval, and eliminating the need for costly join operations between *collections*. Please see Figure 3 for an example.

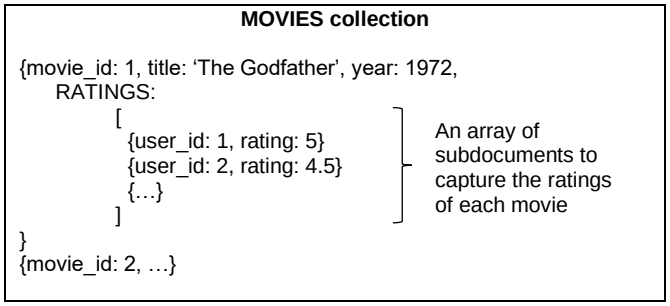

Figure 3. RATINGS Collection is Embedded in MOVIES Collection as Subdocuments

However, this approach also comes with a tradeoff. Any analysis of these *subdocuments* (such as the RATINGS array in Figure 3) requires an *unwind* operation to flatten them so that the data they contain can be processed. The *unwind* operation converts an array of *subdocuments* into individual *documents*, allowing them to be processed independently. This can be a time-consuming operation, especially when dealing with large *collections*, as it involves creating a new *document* for each *subdocument* in the array. Despite the need for an *unwind* operation, the ability to nest *documents* to avoid join operations is a significant advantage of document databases, providing a more efficient and flexible way to store and access data.

In summary, a document database allows storing large volumes of data with considerable flexibility. For instance, it allows for creating a normalized document database, where foreign keys establish relationships between *collections*. Alternatively, it enables embedding one *collection* in another using an array of *subdocuments*. In the following section, we look at an example dataset and provide three different document database designs to store this dataset. Then, we examine the compatibility of these designs with a collaborative filtering-based recommender system at a fictitious streaming company.

4 Dataset

The dataset used in this tutorial is obtained from the MovieLens database (Harper & Konstan, 2015). For the purposes of this tutorial, we only focus on the three entities of this database, namely MOVIES, USERS, and RATINGS from the 100K observation benchmark dataset.

The MOVIES entity captures data about each movie in the database, such as its unique identifier (movie_id), title, release year, and its Internet Movie Database (IMDb) URL. It contains a total of 1,682 movies. The USERS entity, on the other hand, captures data about each user who rated the movies in the database. There are a total of 943 unique users. User attributes include a unique identifier (user_id), age, gender, occupation, and zip code. Because there is a many-to-many relationship between MOVIES and USERS (such that a user can rate many movies, and a movie can be rated by many users), the RATINGS entity acts as the associative entity, capturing each user's rating for a movie and the timestamp of the rating (i.e., rating_tstamp). Put simply, when a user rates a movie, the database records the rating and the timestamp in the RATINGS entity. Consequently, the RATINGS entity has a composite primary key comprising of movie_id and user_id. In total, there are 100,000 ratings captured in the database. If this database were designed using a relational model, the design would resemble the entity relationship diagram illustrated in Figure 4.

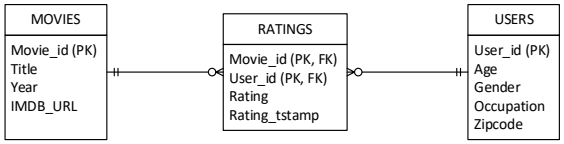

Figure 4. The Entity Relationship Diagram of the Example Database Used in this Tutorial

5 Issue at Hand

A fictious streaming company is adopting a document database to manage its growing customer base and store the data discussed above. As document databases provide greater flexibility in schema design, the company's data scientist is exploring three database designs. One key consideration is that the document database will also be used for making recommendations using collaborative filtering. Hence, the challenge is to identify the database design that is most compatible with the query required to generate the *utility matrix* for collaborative filtering. In the context of this tutorial, the level of compatibility is determined by the time it takes to execute the query. Therefore, a higher level of compatibility indicates that the query to generate the *utility matrix* executes in less time. Below, we discuss these three database designs in detail.

5.1 Option 1: Keep Each Entity Separate and Independent

This design option entails the creation of a distinct *collection* for each entity in the document database. Thus, the relational model depicted in Figure 4 will be replicated in the document database, resulting in three *collections*: MOVIES, RATINGS, and USERS. These *collections* contain foreign keys, enabling them to be joined when necessary.

This option offers the advantage of capturing data in a normalized manner, facilitating easier maintenance. Modifying, adding, or removing data can be done without encountering insertion, deletion, or update anomalies. Additionally, this design offers flexibility by allowing new *collections* to be added (with appropriate foreign key relationships) when necessary. However, the drawback of this design is that it may necessitate expensive join operations if users query data from multiple *collections* simultaneously. Such a scenario could result in an increased query execution time.

5.2 Option 2. RATINGS are subdocuments of MOVIES

For this option, each rating of a movie will be made a *subdocument* of that movie. Therefore, the MOVIES *collection* will contain a RATINGS array that stores ratings as *subdocuments*, as demonstrated in Figure 5. Additionally, a separate *collection* will be created for the USERS entity, allowing it to be joined with the MOVIES *collection* as needed. This approach results in two *collections* in the document database: MOVIES and USERS.

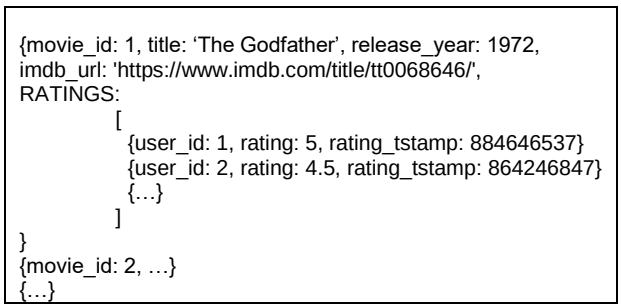

Figure 5. RATINGS are Subdocuments of MOVIES (for Option 2)

One advantage of using this option is that it reduces the number of *collections* in the database (compared to Option 1), which can help avoid costly join operations. As a result, queries related to movies and their ratings may produce faster results than with a normalized design. However, this approach has some disadvantages, such as the possibility of update and deletion anomalies. If a movie is deleted from the MOVIES *collection*, all its *subdocuments* will be removed as well, potentially causing data loss. Another disadvantage is that analyzing the RATINGS *subdocuments* will require an *unwind* operation, which may increase query execution times.

5.3 Option 3. RATINGS are subdocuments of USERS

Option 3 involves storing each rating of a user as a *subdocument* of that user in the USER *collection*. The RATINGS array in each *document* will hold these *subdocuments*. Please refer to Figure 6 for an illustration. Additionally, a separate MOVIES *collection* will be created to enable joins with the USERS

collection as needed. Consequently, the document database will consist of two *collections*: MOVIES and USERS.

```
{user_id: 1, age: 21, gender: 'F', occupation: 'student', zipcode: 33620,
RATINGS:
        [
          {movie_id: 1, rating: 5, rating_tstamp: 884646537}
          {movie_id: 2, rating: 3, rating_tstamp: 864246847}
          …
        ]
}
{user_id: 2, …)
{…}
```

Figure 6. RATINGS are subdocuments of USERS (for Option 3)

This design has a similar advantage, when compared with Option 1, in that it reduces the number of *collections* in the database, which can help avoid costly join operations. Therefore, queries related to users and their ratings may produce much faster results than with a normalized design. However, this option, like the previous one, is vulnerable to potential update and deletion anomalies. For instance, deleting a user from the USERS *collection* will also remove all user's *subdocuments*, potentially resulting in data loss. Further, like the previous design, this option requires an *unwind* operation to analyze the RATINGS *subdocuments*, which may increase query execution times.

It is important to note that there could be many other designs to store the same data. For this tutorial, we focus on these three specific designs. To illustrate these designs, we created three separate document databases in *MongoDB Atlas*, the cloud-based version of the popular document database provider, MongoDB. These databases correspond to the three options described earlier and are named ML_Option_1 (for Option 1), ML_Option_2 (for Option 2), and ML_Option_3 (for Option 3). We encourage instructors and students to connect to these databases using the connection details provided in Table 2 and explore them. The program for accessing these databases, *MongoDB Compass*, can be downloaded and installed for free from https://www.mongodb.com/try/download/compass.

Table 2. Connection details of MongoDB Atlas

Server:	cluster0.dadyq.mongodb.net
Username:	movielens
Password:	movielens123

5.4 Question 1

To efficiently manage the growing customer base, the data scientist will implement one of the database designs discussed earlier. This database will also be used to make recommendations using collaborative filtering. This will involve writing a query against the database to retrieve the necessary fields for constructing the *utility matrix*. Which database design provides the shortest query execution time? To answer this question, it is necessary to write a query to construct the *utility matrix* from each database and compare the query execution times in seconds.

5.5 Question 2

If the data scientist wants to perform gender-specific recommendations, which database design would provide the shortest query execution time? It is worth emphasizing that providing gender-specific recommendations entails generating separate utility matrices for each gender represented in the database. In this context, gender has a binary representation, with users identified as either male or female. As a result, two separate queries are required, one for each gender. To answer this question, you should compare the query execution times of both queries across all three databases.

6 Instructor Guide

6.1 Initial Discussion

Instructors running this tutorial can use *MongoDB Compass* to connect to the three document databases using the connection details provided in Table 2. Once connected, instructors can display each database design using *MongoDB Compass*. We suggest showing all *collections* in each database and displaying any *subdocuments* (if present) in each *collection*. Instructors can lead a discussion on the design differences between the three databases. During this discussion, instructors can prompt students to identify and locate the fields necessary to construct the *utility matrix* in each database and think about the query required to retrieve these fields from each database.

6.2 Hands-on Demonstration

After the class discussion, instructors can proceed to the hands-on portion of the tutorial by using the two *Jupyter Notebooks* available at https://github.com/varolkayhan/movielens_tutorial. These files can be opened using the *Jupyter Notebook* application, which requires a Python engine greater than version 3. We recommend using the Anaconda distribution to install both *Jupyter Notebook* and Python on a local device.

Both notebooks establish a connection to *MongoDB Atlas*. The notebooks retrieve the fields required for the *utility matrix* by sending a query to each of the three databases discussed earlier. The notebooks keep track of the query execution time for each query and compare them. The remaining parts of the notebooks demonstrate one approach to collaborative filtering and how to leverage the *utility matrix* to make movie recommendations. Please note that the collaborative filtering approach is optional and does not affect the query execution times. The explanation of the code is provided in each notebook.

6.3 Debriefing

After running the notebooks, instructors can debrief students using the information provided in Appendix A. It is also important to discuss the reasons for long query execution times, which are also provided in the same appendix.

After debriefing, instructors can continue the conversation by addressing scaling issues. For example, it is important to think about scaling for a million users. Retrieving the required fields to create the *utility matrix* from the database for a million users is a challenging task, and it becomes even more challenging to calculate the pairwise *cosine similarity* values from a *utility matrix* that has a million users, as it involves dealing with an $O(n^2)$ complexity.

One possible approach to reduce the computational burden of this task is to sample users. For example, 5% of users can be sampled to compute the cosine similarities using a 5,000 x 5,000 matrix. Then, a fixed number of similar users, such as 100, can be cached and stored in the database. With subsequent re-sampling, this list can be updated with the discovery of new similar users. Finally, a subset of users can be sampled for generating recommendations (e.g., 30 out of 100) so that recommendations are fresh for each request. These parameters can be adjusted to control computational costs.

7 Lessons Learned

We used this tutorial in several graduate-level database courses over the course of two years at a university in the southeast United States. We learned the following lessons.

7.1 Preparing the Students

To manage class time effectively, instructors may consider assigning the front end of this tutorial as a reading assignment before the class meeting intended for the hands-on demonstration. Students should read the sections on recommender systems, collaborative filtering, *utility matrix*, and document databases as discussed in this tutorial. This way, students can come prepared to the class and the instructor can focus on the hands-on demonstration of the tutorial during class time.

7.2 Initial Discussion

Before the hands-on demonstration, we recommended that instructors engage in a class discussion about each database design to help students understand the required fields for the *utility matrix* and where to locate them in each *collection*. Additionally, the concept of *utility matrix* should be revisited to ensure that students can identify the appropriate fields to retrieve from the *collections*. During the discussion, instructors should highlight some of the expensive operations, such as joins and *unwinding*, in document databases. For example, joining *collections* in document databases can be as costly as joining tables in relational databases. Therefore, instructors can guide students in identifying the design options with minimal to no joins when retrieving the required fields for the *utility matrix*. Similarly, retrieving fields from *subdocuments* using the *unwind* operation can be more costly than retrieving them directly from *documents*. Instructors can offer guidance on identifying the design options that provide the required fields without the *unwind* operation.

It is important to keep in mind that the discussions should revolve around query execution times. The objective is to help students identify the database design that would result in the shortest query execution time while retrieving the fields required for the *utility matrix*.

7.3 Hands-on Demonstration

We suggest that instructors demonstrate and run the *Jupyter Notebook* titled “Question 1” in class cell-by-cell. This will give students practical experience on how to connect to the *MongoDB Atlas* server and execute queries. As the queries in this notebook execute fast, instructors can run the entire notebook relatively quickly. However, we recommend that instructors run the second notebook (titled “Question 2”) before coming to class and open the notebook to show only the results. Otherwise, queries in this notebook require more time to run, and thus, might consume valuable class time that instructors could use for debriefing.

7.4 Debriefing

During the debrief, we suggest that instructors explicitly distinguish between the various steps of collaborative filtering, which include constructing the *utility matrix*, calculating cosine similarities, and comparing the preferences of similar users. It is important to emphasize that the database design solely influences the construction of the *utility matrix* but not the speed at which other steps are completed. The computational power of the device that executes the recommender system is the only factor affecting the speed of the other steps.

To encourage further engagement with the material, instructors can ask students to brainstorm additional analytic queries that could leverage the same data set. For instance, analyzing the number of ratings received by each movie per month can reveal which movies are losing popularity over time. By doing so, a streaming platform can selectively include or exclude these movies from its recommendations based on its user engagement policy. Additionally, examining the number of ratings given by each user per month and segmenting the results by user demographics, such as gender or profession, can assist a streaming platform in developing targeted strategies to boost user engagement. These types of discussions can help students gain a deeper understanding of how a database can serve different types of queries.

If a single database design may not be able to provide short execution times for some of these queries, instructors can guide students to consider potential trade-offs and strategies for prioritizing the queries based on criteria such as execution frequency and business value. For instance, they might consider creating separate databases with differing designs for different queries, or they might choose to optimize the database design to support multiple queries. These discussions can help students identify the trade-offs involved in using a single technology to support multiple workloads.

7.5 Query Optimization

To extract the necessary fields for the *utility matrix*, multi-step queries must be written for document databases like MongoDB. For instance, a query for Question 2 might involve *unwinding* subdocuments, joining them with another *collection*, and filtering by gender. Unlike relational databases, MongoDB's query optimizer doesn't reorder these steps, so queries are executed as written. This offers instructors an opportunity to challenge students to think creatively about query construction and evaluate the query execution times. For instance, students could experiment with different approaches to gender-based

filtering (as in Question 2, Option 3) to see how query execution times are affected, gaining valuable insights into effective query-writing practices for document databases.

7.6 Student Feedback

We incorporated this tutorial into multiple graduate-level courses that focus on advanced database design and administration. The feedback we received from students has been consistently positive. In one particular course, we administered a short survey comprising five questions, and 75% of the students strongly agreed that the tutorial was informative. The students appreciated the discussions on database scalability, relational versus document database design, and collaborative filtering. The students were especially pleased with the live Jupyter notebook-based demonstrations.

It is important to highlight an observation that emerged during the use of the tutorial in these courses. Students who lacked familiarity with NoSQL concepts did not actively participate in the initial discussion. However, this lack of participation does not imply that these students did not benefit from the tutorial. On the contrary, students found the tutorial to be highly informative. Their limited contributions to the discussion were primarily due to their lack of background knowledge. To address this issue, we revisited several important topics during the tutorial's discussion, including the NoSQL landscape, scaling database systems (from shared memory relational databases to distributed computing NoSQL databases), document databases (including nested subdocuments), relational versus document database designs, and design options for collaborative filtering and cosine similarity computations. By reviewing these topics, we aimed to establish a foundation for the tutorial. Therefore, we recommend that the tutorial should be accompanied by at least one additional module focusing on NoSQL database concepts. This approach ensures that students have the necessary background before engaging in discussions about document database designs.

8 Conclusion

This paper presents a tutorial that explores the compatibility between a big data technology and a collaborative-filtering based recommender system. Specifically, we investigate which document database design implemented in the big data technology generates the shortest execution time for a query that extracts the necessary data to construct a *utility matrix* for a collaborative filtering-based recommender system. After introducing essential concepts related to recommender systems and document databases, we present a fictitious data scientist's challenge and provide a detailed guide for instructors on how to conduct the tutorial, including discussion, hands-on demonstration, and debriefing. Finally, we share some of the insights gained from using the tutorial in several graduate-level database courses.

References

Anuradha, J. (2015). A brief introduction on Big Data 5Vs characteristics and Hadoop technology. *Procedia computer science, 48*, 319-324.

Bean, R. (2020). The 'Failure' Of Big Data. *Forbes.com*. Retrieved from https://www.forbes.com/sites/randybean/2020/10/20/the-failure-of-big-data/?sh=3f7395b9a218

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM, 35*(12), 61-70.

Harper, F. M., & Konstan, J. A. (2015). The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst., 5*(4), Article 19.

Hinkle, D. (2021). How Streaming Services Use Algorithms. *amt-lab.org*. Retrieved from https://amt-lab.org/blog/2021/8/algorithms-in-streaming-services

MongoDB. Advantages of NoSQL Databases. Retrieved from https://www.mongodb.com/nosql-explained/advantages

Ramasamy, K. (2019). How Big Data Can Be A Big Problem. *Forbes.com*. Retrieved from https://www.forbes.com/sites/forbestechcouncil/2019/07/26/how-big-data-can-be-a-big-problem/?sh=18e377b435c9

Resnick, P., & Varian, H. R. (1997). Recommender systems. *Communications of the ACM, 40*(3), 56-58.

Sharma, R. (2022). Overcoming Key Challenges To Achieve Success With Digital Transformation. *Forbes.com*. Retrieved from https://www.forbes.com/sites/forbestechcouncil/2022/07/07/overcoming-key-challenges-to-achieve-success-with-digital-transformation/?sh=7be398962f66

Statistica. (2022). Big data market size revenue forecast worldwide from 2011 to 2027. Retrieved from https://www.statista.com/statistics/254266/global-big-data-market-forecast/

Wang, X., & He, Y. (2016). Learning from uncertainty for big data: future analytical challenges and strategies. *IEEE Systems, Man, and Cybernetics Magazine, 2*(2), 26-31.

Watson, H. J. (2019). Update tutorial: Big Data analytics: Concepts, technology, and applications. *Communications of the Association for Information Systems, 44*(1), 21.

Appendix A: Answers to the Hands-on Demonstration

Question 1 of the tutorial aims to identify the document database design that achieves the shortest query execution time for constructing the *utility matrix* for the collaborative filtering-based recommender system. The *Jupyter Notebook* for Question 1 compares the three database options and shows that Option 1, which uses a normalized database, has the shortest query execution time. On average, constructing the *utility matrix* from this database is 21% faster than using Option 2 and 11% faster than using Option 3. It's important to note that these values are not based on the hardware of the client computer where the notebooks are run, but rather on the performance of the cloud server at the time of notebook execution. Option 1 has the shortest query execution time because of its normalized database structure. The necessary fields for the *utility matrix* are stored in the RATINGS *collection*, and a simple `find()` query retrieves them directly from there without the need for additional operations like join or *unwind*. This query is equivalent to the following select statement in a relational database: `SELECT movie_id, user_id, rating FROM RATINGS`. Since no additional operations are required, the normalized database design delivers the shortest query execution time. In contrast, Option 2 and Option 3 both require an *unwind* operation to flatten *subdocuments*, which adds to the query execution time. Although the *unwind* operation is simple, it still has a cost in terms of query performance.

Question 2 of the tutorial aims to identify the document database design that achieves the shortest query execution time for constructing the gender-based utility matrices for the collaborative filtering recommender system. The *Jupyter Notebook* for Question 2 compares the three database options and finds that Option 3 achieves the shortest query execution times. Queries that use this database execute, on average, 97-99% faster than those that use the other databases. It is important to note that these values may depend on the cloud server's performance at the time of notebook execution. The short query execution time observed in Option 3 is because this database design does not require a join operation unlike the other databases. In this database, ratings are stored as *subdocuments* of USERS. By performing the gender-based filter, we can *unwind* the ratings *subdocuments* and retrieve all the required fields without any additional operation. In contrast, queries that construct the utility matrices in Option 1 and Option 2 join the results of the gender filter in the USERS *collection* with other *collections* to retrieve the required fields. Join operations are expensive in document databases, just like in relational databases. As a result, Option 3 is the only option that doesn't require join operations, making it the database with the shortest query execution time.

Appendix B: Definition of Key Terms

Table B1. Definition of Key Terms

Term	Definition
$addFields	An operation performed in MongoDB to add new fields to a document. It is used to remove the object notation of subdocuments in this tutorial.
$lookup	An operation performed in MongoDB to join two collections.
$match	An operation performed in MongoDB to filter the documents of a collection.
$project	An operation performed in MongoDB to select requested fields only.
$unwind	An operation performed in MongoDB to deconstruct an array of a document. It can also be used to flatten the subdocuments of a document.
Collection	A group of records in a document database that is equivalent to a “table” in relational databases.
Cosine similarity	A metric that measures similarity between two vectors using the cosine of the angle between the vectors. Higher values indicate higher similarity.
cosine_similarity	A function of the Pandas library (in Python) that calculates pairwise cosine similarities between multiple vectors.
Dictionary	A data structure in Python that consists of key-value pairs. Each key is mapped to the associated value.
Document	An individual record in a collection of a document database that is equivalent to a single “row” of a table in relational databases.
find()	A command to execute a query and retrieve results in MondoDB.
Jupyter Notebook	A web-based interactive computing platform that can be installed locally to run Python code.
MongoDB Atlas	The cloud version of the MongoDB document database.
MongoDB Compass	The client program for accessing a MongoDB document database (either installed locally or on the cloud).
pivot_table	A function of the Pandas library (in Python) that can create a cross tabulation using a aggregate function. It is used to create the utility matrix in this tutorial.
Subdocument	A document that is nested in another document in document databases.
Unwind	See $unwind.
Utility matrix	A matrix used in collaborative filtering recommender systems that captures users’ preferences of items.
Warm up	In the context of this tutorial, it refers to caching the queried data. It lets the database to cache the data so faster results can be obtained.

About the Authors

Varol Kayhan is an Associate Professor of Information Systems in the Muma College of Business at the University of South Florida. His research focuses on decision-making and the use of machine learning in cybersecurity and has appeared in *Communications of the Association for Computing Machinery, Communications of the Association for Information Systems, Information & Management, Behavior Research Methods, Big Data, Journal of Computer Information Systems*, and others. He is the author of an eBook on data mining, which has been officially integrated into the curriculum of numerous colleges and universities. He teaches data analytics and machine learning courses at the undergraduate, graduate, and executive levels. He holds Ph.D. and MS degrees from the University of South Florida, and a BS degree from the Middle East Technical University in Turkey.

Don Berndt is a faculty member in the School of Information Systems and Management at the University of South Florida's Muma College of Business. He received his Ph.D. in Information Systems from the Stern School of Business at New York University. He also holds a M.S. in Computer Science from the State University of New York at Stony Brook. His research interests focus on the intersection of artificial intelligence (AI) and database systems, including data warehousing, machine learning, and text analytics. His work appears in leading journals, including the *Association for Computing Machinery Transactions on Management Information Systems, Communications of the Association for Computing Machinery, Decision Support Systems, Discrete Applied Mathematics, Institute of Electrical and Electronics Engineers Computer, Journal of Biomedical Informatics and Journal of the American Medical Informatics Association*. Along with his academic focus, he has participated in a series of entrepreneurial ventures, most recently co-founding Kliken.com, an AI-based online advertising company, after co-founding Medegy Inc., a healthcare analytics company.