# Aligning Data Science Workloads with Document Databases

# (Student Handout)

# Introduction

The number of organizations adopting big data technologies is on the rise (Watson, 2019). However, organizations face different challenges regarding these technologies. One of these challenges is that despite vendors’ promises, these technologies fail to improve the performance of existing tasks (Wang & He, 2016; Watson, 2019). While there could be many reasons for this, an important one is a misalignment between existing tasks and big data technologies (Prybylski, 2019; Ross, Beath, & Quaadgras, 2013). This phenomenon can be explained using the theory of task-technology fit which posits that one can realize performance gains only if the capabilities of a technology match the requirements of a task performed using this technology (Goodhue & Thompson, 1995). Naturally, a task can consist of many subtasks. Further, some (or all) of these subtasks could be analytic such that they leverage machine learning, artificial intelligence, and data science tools and techniques. If there is no alignment between a task, or its subtask(s), and a big data technology, one might not be able to achieve sought after performance improvements. This is important because big data technologies usually involve expensive investments that require not only specific hardware and software, but also talent that can manage, operate, and effectively use these technologies. If organizations cannot align big data technologies with tasks, they might lose faith in them, and therefore, start focusing on short-term goals rather than building data-driven analytic cultures (Bean & Davenport, 2019). As a result, they might lose their competitiveness in this critical area.

The goal of this tutorial is to investigate such an alignment between a data science task and a big data storage technology. The task is making movie recommendations using a collaborative filtering-based recommender system, and the storage technology is a document database. You will examine different database designs one can implement for a document database and discuss the level of alignment between these designs and an analytic subtask of the recommender system. Through the lens of task-technology fit, you will show the performance gains one can attain after ensuring a high degree of alignment between the analytic subtask and the database design.

At the end of this tutorial, you will be able to:

1. contrast a relational database with a document database
2. explain the collaborative filtering-based recommender system, which is one of the most popular recommender systems many online services use
3. contrast three types of designs one can implement in a document database
4. examine the performance implications of different levels of task-technology fit.

To achieve these objectives, you should have the following pre-requisite knowledge: relational database concepts (intermediate level); document database concepts (introductory level); and Jupyter Notebook and Python technologies (introductory level).

In the rest of this paper, you will learn about collaborative filtering-based recommender systems. Next, you will go over the document database concepts and see different types of designs one can implement in a document database. Then, you will learn about the theory of task-technology fit. Next, you will see the data used for this tutorial and be presented with the two questions faced by a hypothetical data scientist.

# Recommender Systems

The goal of recommender systems is to predict user responses to options (Leskovec, Rajaraman, & Ullman, 2020). Options also can be referred to as items. One of the industries that commonly uses recommender systems is the streaming industry where service providers predict who will watch what, and thus, provide suggestions for what a user should watch next. This, in turn, helps service providers keep their users on their platforms. As a result, providers reduce customer attrition and increase subscription-based revenues.

There are two broad methods for implementing recommender systems, namely content-based and collaborative filtering-based approaches. Content-based systems recommend items based on item properties. For example, if a user likes to watch science-fiction shows, these types of systems search for other science-fiction shows on the platform and start recommending them to the user. On the other hand, collaborative filtering systems recommend items based on similarities with other users (or content). These are the types of recommendations commonly known as "users who like this item also like that item" in many online services. Because of their popularity and effectiveness, we focus on collaborative filtering-based recommender systems in this tutorial.

To make recommendations using a collaborative filtering-based system, one needs to complete several subtasks, one of which is the construction of a *utility matrix[[1]](#footnote-1)* that captures user-item preferences. Users make up the rows of this matrix while items make up the columns. For each user-item pair, one can capture a chosen metric such as a binary value that shows whether the user engaged with that item. However, this metric does not capture whether a user liked the item with which they engaged. Instead, it could be more insightful if one captures a user’s rating of an item, say on a scale of one to five. We present such a *utility matrix* based on four users' ratings of five movies in Table 1 as an example. Note that the blank cells in this table indicate that the user has not rated that movie yet.

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 |

These kinds of utility matrices typically are sparse with many blanks cells because there might not be preferences for all user-item pairs. The goal of collaborative filtering is to predict these values based on what is already available in the matrix. To do this, one must calculate the similarity between each pair of users and identify those that are most similar. For example, if user A is most similar to user B, then one can make recommendations to A by observing B’s preferences (or vice versa). Consider the *utility matrix* provided in Table 1. It shows that User 1 and User 4 are very similar based on their ratings of movies. If there is no other user that is more similar to User 1, then one can recommend Movie 3 to User 1 because this movie is highly rated by User 4.

To do this in a more systematic way, one needs to calculate pairwise user similarities. Thus, one can identify those users that are most similar to each other. One popular metric to calculate these similarities is *cosine similarity*. This metric finds the cosine of the angle between two users using Equation 1, where A and B represent the preference vectors of two users. Therefore, one can calculate all pairwise cosine similarities between users in a *utility matrix*. Higher and positive values of this metric indicate that the cosine of the angle is smaller, and thus the distance between the users is smaller. Therefore, the larger the cosine value, the higher the similarity between users.

|  |  |
| --- | --- |
| Text, whiteboard  Description automatically generated | (1) |

While calculating cosine similarity, one can consider missing values in the *utility matrix* as zeros. This does not necessarily mean that a user's preference is zero, but rather the preference for this user-item pair is not available. As an example, one can calculate the cosine similarity between User 1 and User 4 (as provided in Table 1) as 0.814 as follows:

|  |  |
| --- | --- |
| Cosine similarity between User 1—User 4 = |  |

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

|  |  |
| --- | --- |
| Cosine similarity between User 1—User 2 = |  |

These two calculations show that User 1 is more similar to User 4 than User 2. In fact, one can calculate all other pairwise similarities only to find that User 1 is most similar to User 4 in this example data set. Then, one can provide recommendations to User 1 by searching for the items that are rated highly by User 4 but not rated by User 1 (or vice versa). In this case, one such example is Movie 3, which received a rating of 5 (out of 5) from User 4 but no rating from User 1. Therefore, one can recommend Movie 3 to User 1.

As described above, collaborative filtering requires one to complete several subtasks. At a high level, these include constructing a *utility matrix*, calculating pairwise cosine similarities from this matrix, and comparing the preferences of similar users. Note that these subtasks are analytic and do not require a storage technology. For instance, one might construct a *utility matrix* using an application programming interface (API) call of a data analysis and manipulation library (such as Numpy or Pandas). However, one must execute a query on a database to retrieve the data first. For example, in the case of a streaming service, this data might reside in a database that keeps track of users, movies, and ratings to support the core processes of the service. Because these types of databases also can get very large, organizations might gravitate toward big data technologies, such as a document database, due to its ability to store large amounts of data in a parallel and scalable architecture. Therefore, the document database concepts is discussed in the next section.

# Document Databases

Broadly, a document database is one of the four types of NoSQL (i.e., non-relational) database management systems. As a big data technology, it can store large volumes of data using a scalable architecture based on distributed computing without sacrificing performance. It is common that one uses a document database to store the data needed to run many analytic workloads such as recommender systems (see Fine, 2018). This is because a document database can store millions of records using a cluster-based architecture that has multiple compute nodes. These nodes split data into multiple partitions (or shards) and spread it across independent disks. The number of nodes can be scaled horizontally by adding more nodes to the cluster when needed. If one executes a query on the database, the query engine reads and processes the data in these nodes in parallel, which can significantly improve query performance. Therefore, a document database is one of the most popular data storage technologies to store large volumes of data.

In a typical document database, one stores data in a *collection.* Each *collection* consists of multiple *documents*. When compared with a relational database, a *collection* corresponds to a table and *documents* correspond to rows. Each *document* is a stored using field-value pairs according to the JavaScript Object Notation (JSON) standard. As an example, a MOVIES table that has *m* rows and *n* columns in a relational database, as seen in Figure 1, might correspond to a MOVIES *collection* with *m* *documents* in a document database. Each *document* might consist of *n* field-value pairs. Though, depending on what one captures, a *document* can have more (or fewer) field-value pairs than another. Therefore, unlike a table of a relational database, a *collection* of a document database does not have a rigid structure. *Documents* of a *collection* can have as many field-value pairs as needed.

Diagram

Description automatically generated

Figure 1. Comparison between relational databases and document databases

A document database can have more than one *collection*, just like having more than one table in a relational database. These *collections* can be completely independent of each other. Though, it is also possible that *collections* have relationships with one another through foreign keys. For example, consider adding a RATINGS *collection* to the document database presented in Figure 1 to keep track of each movie’s ratings. A foreign key (such as movie\_id) in the RATINGS *collection* can enable one to identify every single rating of a movie in this database. Please see Figure 2 for an example.

Diagram

Description automatically generated

Figure 2. Relationships in a document database

However, a document database is flexible and also allows one to combine multiple collections into a single collection. For example, rather than creating a new collection, one can nest a *document* in another to implement hierarchical arrangements. The nested *document* is referred to as a *subdocument.* Naturally, a *document* can have many *subdocuments*. This enables one to embed one *collection* in another using *subdocuments*. As a result, one can easily store one-to-one and one-to-many relationships between two *collections* in the same *collection*. For example, a movie can have many ratings. One can capture this one-to-many relationship between the MOVIES and RATINGS *collections* by converting RATINGS to an array of *subdocuments* in the MOVIES *collection*. Thus, a movie can have many subdocuments. Please see Figure 3 for an example.

|  |
| --- |
| **MOVIES collection**  {movie\_id: 1, title: ‘The Godfather’, year: 1972,   RATINGS:  An array of subdocuments to capture the ratings of each movie  [  {user\_id: 1, rating: 5}  {user\_id: 2, rating: 4.5}  {…}  ]  }  {movie\_id: 2, …} |

Figure 3. RATINGS collection is embedded in MOVIES collection as subdocuments

The ability to create *subdocuments*, and thus, embed one *collection* in another, eliminates the need to perform costly join operations between *collections*. However, this also comes at a cost because any analysis involving these *subdocuments* (such as the RATINGS array in Figure 3) requires one to flatten these *subdocuments* using an *unwind* operation. Only then can one process the information stored in these *subdocuments*.

As seen in the above discussion, a document database can be more flexible than a relational database with respect to database design. One can create a normalized document database where foreign keys establish relationships between *collections*. Alternatively, one can create another document database where one *collection* is embedded in another using an array of *subdocuments*. In the section that follows, we discuss the theory of task-technology fit to understand how the design of a document database might impact a task that this database supports.

# Task-Technology Fit

The theory of task-technology fit asserts that the alignment between a technology and the task this technology supports determines performance gains obtained from this technology (Goodhue & Thompson, 1995). Note that the theory defines technology as any information system (consisting of hardware, software, and data), and task as any action one performs to generate outputs from a set of inputs. Performance is usually conceptualized using efficiency, effectiveness, and quality. While performance can be operationalized using many different metrics, one of the most commonly used metrics is time savings (Delone & McLean, 2003).

According to the theory, as the gap between a technology and the task it supports widens, it becomes more difficult to achieve performance gains from the use of this technology. As an example, consider an accountant who wants to calculate an organization’s quarterly revenue from customer orders (which is the task) using a word processor (which is the technology). Because word processors are not geared toward keeping records or performing calculations, it might take more time to perform the task using this technology. Therefore, one can conclude that there is a misalignment between the task and technology.

In the context of this tutorial, your goal is to make recommendations using a collaborative filtering-based recommender system. You will use a document database as the back-end storage technology. As discussed in Section 2, making recommendations has at least three subtasks, one of which is constructing a *utility matrix*. Even though this is an analytic workload independent of the document database, one still needs to retrieve the data used in the *utility matrix* from the document database. Further, as discussed in Section 3, document databases can have flexible designs ranging from normalized collections to collections with documents and subdocuments. Therefore, your focus will be on the alignment between the query that retrieves the data required for the *utility matrix* and the document database’s design. The outcome of this alignment can be measured using the query execution time. Therefore, if one can retrieve the data in a shorter amount of time, there is a greater degree of alignment between the recommender system and the document database.

In the next section, you will examine an example dataset and go over three different document database designs that can store this dataset. Then, you will see the problem faced by a hypothetical data scientist for choosing the right database design.

# Dataset

The dataset used in this tutorial is retrieved from the MovieLens database (Harper & Konstan, 2015). For the purposes of this tutorial, you will focus on the three entities of this database, namely MOVIES, USERS, and RATINGS.

The MOVIES entity captures data about movies, such as each movie’s unique identifier (movie\_id), title, release year, and Internet Movie Database (IMDb) URL. It contains a total of 1,682 movies. The USERS entity captures data about each user who rated the movies in the database. It contains a total of 943 users. The attributes include each user’s 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 between MOVIES and USERS to capture each user’s rating for a movie and the timestamp of the rating (i.e., rating\_tstamp). To put it differently, when a user rates a movie, the database captures the rating and the rating’s timestamp in the RATINGS entity. Therefore, the RATINGS entity has a composite primary key consisting of movie\_id and user\_id. There is a total of 100,000 ratings captured in the database. If this were a relational database, the database design would look like the entity relationship diagram presented in Figure 4.



Figure 4. The entity relationship diagram of the example database used in this tutorial.

# The Problem

A data scientist wants to achieve the greatest degree of task-technology fit between a collaborative filtering-based system and a document database that stores the above data. The data scientist wants to use a document database because of its ability to store large volumes of data as well as the parallelism and flexibility it provides. The data scientist is considering three different designs for the document database.

## Option 1: Keep Each Entity Separate and Independent

In this option, the data scientist will create a separate *collection* in the document database for each entity. Therefore, the relational database shown in Figure 4 will be recreated in the document database. As a result, there will be three *collections* in the database: MOVIES, RATINGS, and USERS. There will be foreign keys between the *collections* so that one can join these *collections* when needed.

## Option 2. RATINGS are subdocuments of MOVIES

In this option, the data scientist will make each rating of a movie a *subdocument* of that movie. Therefore, the MOVIES *collection* will have a RATINGS array that stores ratings as *subdocuments*. Please see Figure 5 for an example *document*. In this option, the data scientist will create a separate *collection* for the USERS entity so that one can join it with the MOVIES *collection* when needed. As a result, there will be two *collections* in the document database: MOVIES and USERS.

|  |
| --- |
| {movie\_id: 1, title: ‘The Godfather’, release\_year: 1972, imdb\_url: 'https://www.imdb.com/title/tt0068646/',  RATINGS:  [  {user\_id: 1, rating: 5, rating\_tstamp: 884646537}  {user\_id: 2, rating: 4.5, rating\_tstamp: 864246847}  {…}  ]  }  {movie\_id: 2, …}  {…} |
|  |

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

## Option 3. RATINGS are subdocuments of USERS

In this option, the data scientist will make each rating of a user a *subdocument* of that user. Therefore, the USER *collection* will have a RATINGS array that stores ratings as *subdocuments*. Please see Figure 6 for an example *document*. In this option, the data scientist will create a separate *collection* for the MOVIES entity so that one can join it with the USERS *collection* when needed. As a result, there will be two *collections* in the document database: 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)

These three options are created as three separate document databases in *MongoDB Atlas*, which is the cloud version of the MongoDB document database. MongoDB is one of the leading document database vendors in the big data industry. You should connect to these databases using the connection details provided in Table 2 and examine the database designs. The program needed to access these databases, namely *MongoDB Compass*, can be downloaded and installed freely from <https://www.mongodb.com/try/download/compass>. The databases named as ML\_Option\_1, ML\_Option\_2, and ML\_Option\_3 correspond to the designs described in Option 1, Option 2, and Option 3, respectively.

Table 2. Connection details of MongoDB Atlas

|  |  |
| --- | --- |
| **Server:** | cluster0.dadyq.mongodb.net |
| **Username:** | movielens |
| **Password:** | movielens123 |

# Question 1

The data scientist must use one of these document databases to make recommendations using collaborative filtering. To do this, the data scientist must write a query to retrieve the fields required to construct the *utility matrix*. Which database design has the greatest alignment with the query, and why? Better alignment means that one can execute the query and retrieve the fields in the shortest amount of time. Therefore, justify your answer by examining the time it takes to execute the query in each document database. You can measure this by capturing the query execution time (in seconds) in each database.

# Question 2

If the data scientist wants to perform gender-specific recommendations, which database design would have the greatest alignment with the query? Note that gender-specific recommendations mean that one must construct a separate *utility matrix* for each gender (i.e., one for males and another for females). Therefore, one must execute two separate queries, one for males and another for females, in the document databases to retrieve the fields required for two utility matrices. As before, justify your answer by examining the time it takes to execute two queries in each document database. You can measure these by capturing the query execution times (in seconds) in each database.

# References

Bean, R., & Davenport, T. H. (2019). Companies Are Failing in Their Efforts to Become Data-Driven. *Harverd Business Review*. Retrieved from <https://hbr.org/2019/02/companies-are-failing-in-their-efforts-to-become-data-driven>

Delone, W. H., & McLean, E. R. (2003). The DeLone and McLean model of information systems success: A ten-year update. *Journal of Management Information Systems, 19*(4), 9-30.

Fine, J. (2018). Building Intelligent Apps with MongoDB and Google Cloud - Part 1. Retrieved from <https://www.mongodb.com/blog/post/building-intelligent-apps-with-mongodb-and-google-cloud--part-1>

Goodhue, D. L., & Thompson, R. L. (1995). Task-technology fit and individual performance. *MIS Quarterly, 19*(2), 213-236.

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

Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). *Mining of massive data sets* (3rd ed.): Cambridge university press.

Prybylski, H. (2019). Why Most Companies Aren't Ready To Adopt Advanced Technology At Scale. *forbes.com*. Retrieved from <https://www.forbes.com/sites/hankprybylski/2019/02/08/why-most-companies-arent-ready-to-adopt-advanced-technology-at-scale/?sh=53ac3e0f316b>

Ross, J. W., Beath, C. M., & Quaadgras, A. (2013). You May Not Need Big Data After All. *Harvard Business Review (Magazine)*. Retrieved from <https://hbr.org/2013/12/you-may-not-need-big-data-after-all>

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: 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 data required to create the utility matrices in MongoDB. It is used to prevent users from interacting with a “cold” database which might take more time to process queries in the absence of cached data. | |

Table A1. Definition of key terms

1. Italicized terms are defined in Appendix A Table A1. [↑](#footnote-ref-1)