PROJECT 2

Big data

02.12.2020

Mathias W. Nilsen & Maria E. Pedersen

Contents

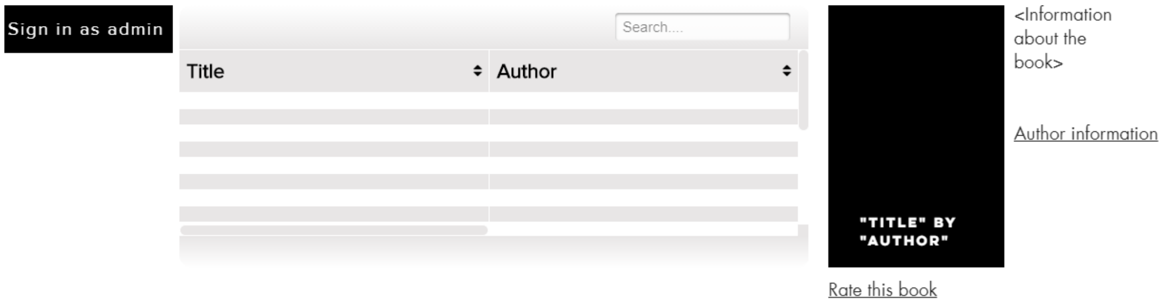
MILESTONE 1

Book ratings site

This is a website that allows users to search through the database to see information about specific books and user rating for that book. It is also possible for the user to rate a book themselves. In addition to allowing the user to see detailed information about an author.

Public

Home page:

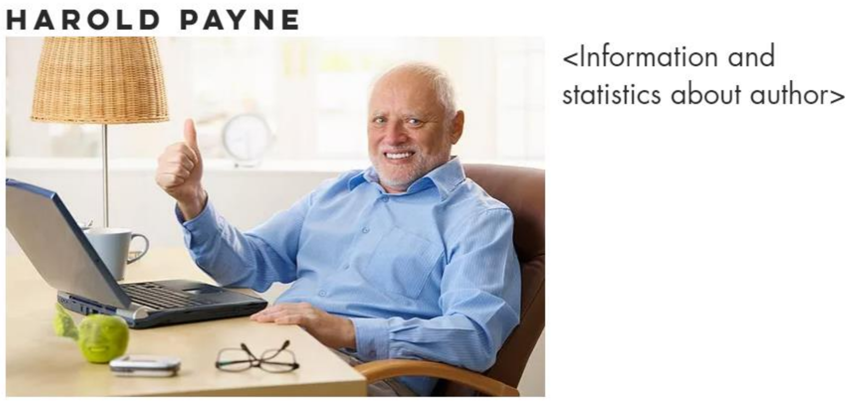


All data is retrieved during the initial loading phase of the site. The view does not allow for any modification of data, only read access to book and author information. In addition to write and read access to ratings related to a book. Thereby making users able to rate books.

The following functionalities are available here:

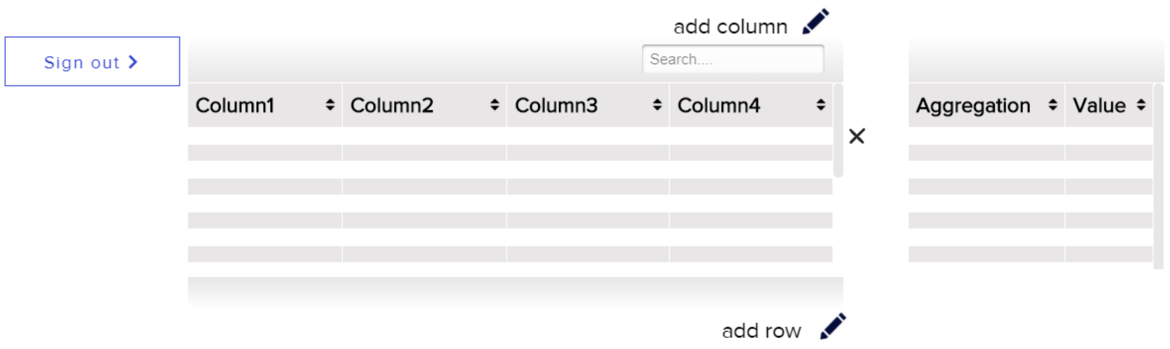
1. Search through the data based on information about the book. Such as for example title, author, ISBN etc.
2. Detailed information about a book is viewed by first choosing a book and then double clicking on this. Which will showcase the information about the book.
3. Rate a book between 1-5, inclusive both ends.
4. View more information about author via given link. More information about this view is defined under.

Author's page:



On the author's page the user get access to information about the author. The statistics about the author will also show up here to give the user an indication about how popular/unpopular the author is. Which in this case is Harold Payne, a very popular figure.

Private



When signed into the website as admin, the following functionalities are allowed:

1. Data search based on column value.
2. Alteration of existing data by clicking on a value in a column and alter it there.
3. Deletion of data works by selecting the value or values to remove and then clicking on the "X" on the website or "Delete"-button on the keyboard.
4. Insertion of new data, like new rows and columns.
5. When creating a new column, the user is asked to either define a default value for every row, or not define a value at all. If none has been defined, then NULL is used. Though this value is not stored in the database. During storage this value is simply omitted.
6. When a row with a book has been selected, then an aggregation table will show up. Here all the aggregation for the book and the author will show up.

The following aggregations are supported:

Aggregation	Description
Books pr author	Contains information about each book pertaining to a specific author.
All books	Contains details about every book in the database. This might be stored in pages, where each page contains maybe 20 books. As this would alleviate network load as it reduces the overall size of the request. Though this would only be necessary if the total list of books impair the load performance.
All book keys	Contains all keys for every book. Which may be used for checking if a book exists. Therefore, this requires having strong persistence.
All session keys	Contains all keys for every session. Which may be used to check if a session exists. This would require a strong persistence in the database.
All averages and summations pr author	Contains averages and summations of interesting fields pertaining to each author. Such as average number of pages pr book by an author.

Averages and summations of a specific author	Contains averages and summations of interesting fields pertaining to a single author.
Detailed information about a single book	Contains detailed information pertaining to a single book.

Key-value database

Data models

Data models are extrapolated based on requirements from supported aggregations.

User:

```
{"user_id": {
   "name": {
     "f_name": String,
     "l_name": String
   }
  }
}
```

User session:

```
{"sess_id": {
   "sess_id": ObjectID,
   "session_info": String
  }
}
```

Book:

```
{"bookID": {
   "bookID": ObjectID,
   "title": String,
   "authors": String,
   "average_rating": Double,
   "isbn": String,
   "isbn13": String,
   "language_code": String,
   "num_pages": Int,
   "ratings_count": Int,
   "publication_date": String,
   "publisher": String
  }
}
```

Books:

Data is divided into buckets of size 20 each. Thereby decreasing overall network load.

```
{"all_books_bucket_X": [{
   "bookID": ObjectID,
   "title": String,
   "authors": String,
   "average_rating": Double,
   "isbn": String,
   "isbn13": String,
   "language_code": String,
   "num_pages": Int,
   "ratings_count": Int,
   "publication_date": String,
   "publisher": String
  }]
}
```

Books by Author:

```
{"books_by_author": [
   "bookID": {
     "bookID": ObjectID,
     "title": String,
     "authors": String,
     "average_rating": Double,
     "isbn": String,
     "isbn13": String,
     "language_code": String,
     "num_pages": Int,
     "ratings_count": Int,
     "publication_date": String,
     "publisher": String
   }
  ]
}
```

Statistics pr author regarding the books:

Data is divided into buckets of size 10 each. Thereby decreasing overall network load.

```
{"all_author_stats_bucket_x": [
   "author_stats": {
     "total_books": Int,
     "total_num_pages": Int,
     "total_ratings_count": Int,
     "avg_ratings_count": Double,
     "avg_num_pages": Double,
     "avg_rating_author": Double
   }
  ]
 }
```

Statistics of author regarding the books:

```
"authorID_stats": {
  "total_books": Int,
  "total_num_pages": Int,
  "total_ratings_count": Int,
  "avg_ratings_count": Double,
  "avg_num_pages": Double,
  "avg_rating_author": Double
}
```

All_keys:

All sessions:

```
{"all_sessions_keys": ["sess_id": ObjectID]}
```

All books:

```
{"all_books_keys": ["bookID": ObjectID]}
```

All authors:

```
{"all_authors_keys": ["author": String]}
```

Users:

```
{"all_users": [
   {
     "user_id": ObjectID,
     "name": {
       "f_name": String,
       "l_name": String
     }
   }
  ]
}
```

Totals:

Total number of books:

```
{"total_number_books": Int}
```

Total number of sessions:

```
{"total_number_sessions": Int}
```

Description for each data model by name:

Book	Contains detailed information pertaining to a single book. The bookID is used as a key to retrieve the information in json format.
Books	Contains detailed information about every book in the database.
Books by author	Contains detailed information of alle the books regarding an author. Books_by_author is the author's name and the value are a list of all the books written by the author. Along with detailed information about the book.
User	Has information pertaining to a single user. UserID is the hashed password and is used to retrieve non sensitive details about a user.
User session	Contains session information pertaining to each instance using the site. The key here is a cookie set on the landing page.
All users:	Contains information about every user in the database.

All keys models	Contains information about every key for their respective names. Further the All authors model contains only the names for each author in the database.
Totals models	Contains the totals pertaining to their respective names.

The User-model contains the first name and last name which can be retrieved with the UserID. This model is used to get information about one single user, while the all_users model can be used to extract information about every user in the database.

The all_books_bucket_X model contains all information correlating to all books as this is the most common use case for this data. In addition to this a user may request fine granular information about a single book. Something that is accounted for by the book-model.

From values in the books_by_author model there are two models with statistic aggregations regarding the author. One is the model all_author_stats_bucket_X, which is a bucket containing all statistics about the books written by the authors in the system. A user can also get statistics regarding a specific author by using the authorID_stats model.

The All_keys-models consists of three models that contains lists of all sessions, books, and authors in the system. With these models a user may see if a certain book or author is in the system. From the all_book_keys and all_sessions_keys the totals-models are calculated. And could be used to get an overview of total database size.

Description of the various operations and their concrete execution

Data search based on column value

Is implemented by the program that interacts with the database, as this is both easier and more practical than having individual data models already sorted and keeping that list sorted.

Summation of total number of books

Done during set intervals:

1. Retrieve all keys for books, using all_books_keys
2. Summarize the total number of books
3. Store new value under key: total_number_books

Done during each transaction:

1. Retrieve data using total_number_books
2. Either increment or decrement accordingly. Given that the book was newly inserted or newly deleted.

Insertion of a new book

Step 1: Add single book entry

a. Add new book and corresponding information under key: bookID.
b. Store the key in the database under the key: bookID

Step 2: Add book to all_book_keys

a. Retrieve all book keys using key: all_book_keys.
b. Add new bookID key to array
c. Store the key array in database under the key: all_books_keys.

Step 3: Add new entry in the array for all books

a. Retrieve all books array using key: all_books_bucket_X
b. Add new book information to said array
c. Store said array under key all_books_bucket_X

Step 4: Add book to the author's bibliography

a. Retrieve books by author using key books_by_author, given author
b. Add new book to array
c. Then store new book correlating to author in the database.

Step 5: Calculate statistics of all the authors

a. Retrieve all_author_stats_bucket_x
b. Program searches through the array for author
c. Calculate new values for current author, either by using pre-existing values or from start
d. Store new values into the database.

Step 6: Calculate statistics of a single author

a. Retrieve authorID_stats
b. Calculate new values, either using pre-existing values or from start
c. Store new values into the database under the same key: authorID.

Step 7: Increment total number of books

a. Retrieve total_number_books
b. Increment value
c. Store new value under the key total_number_books.

Deletion of book

Step 1: Remove book by key

a. Retrieve book keys using bookID as key
b. Remove bookID
c. Store the new book keys to the database

Step 2: Remove entry from the list of all books

a. Retrieve all_books_bucket_X array
b. Search through array and remove bookID and corresponding information from the array
c. Store the new array all_books_bucket_X to the database

Step 3: Remove entry from all_book_keys

a. Retrieve array all_book_keys
b. Remove corresponding bookID from all_book_keys
c. Store the new array all_books_keys to the database

Step 4: Remove book to author's bibliography

a. Retrieve books_by _author by using the author’s full name as key
b. Remove book
c. Store the new book array to the database

Step 5: Remove entry from the list of statistics of all authors

a. Retrieve all_author_stats_bucket_x
b. Calculate new values for the author of the book
c. Store new values into the database

Step 6: Remove entry from authorID_stats

a. Retrieve authorID_stats
b. Calculate new values for the author
c. Store the values under authorID in the database

Step 7: Decrease the total number of books

a. Retrieve the total_number_books key
b. Decrease this value by one
c. Store the new value to the database under the key total_number _books

Updating a specific book

Step 1: Change bookID values

a. Optional: Retrieve book information using the bookID or use already defined information from book object. Which is stored inside the program at this point.
b. Change the value that is to be updated. BookID is not allowed to be changed
c. Store book information under the key: bookID.

Step 2: Change information in the list of all books

a. Retrieve the array all_books_bucket_X
b. Optional: retrieve book information regarding the bookID
c. Change the value that is to be updated. BookID cannot be changed
d. Store the new array all_books_bucket_X to the database

Step 3: Update the author's bibliography

a. Retrieve books by author using the key books_by_author
b. Optional: retrieve book information regarding the book to be updated by using the bookID
c. Update the value that is to be changed and add that to the array.
d. Then store the updated book correlating to the already existing author in the database.

Step 4: Calculate new values for the statistics of all authors

d. If any integer values have been updated, then all_author_stats_bucket_x are calculated with the new values.
a. Then those new values are stored into the database.

Step 5: New values for authorID_stats

a. If any integer values have been updated, then authorID_stats are calculated with the new values.
b. Then store those new values into the database under the key authorID.

Distribution models theory

Choosing a distribution model for the website depends on the purpose and requirements of the data. Sharding enhances both reading and writing performance as it spreads and cluster similar data together. Something that is done by specifying a shard key. Though each node would then only contain a subset of the whole data making it prone to failure. As this model is not fault tolerant.

Fault tolerance could be gained by using replication to replicate data x times. Something that would further add availability as data is now present on multiple nodes. Though this would decrease the overall data storage in the cluster. Meaning that a high replication factor would impact the overall storage. Whilst making it too low, might make the data more prone to loss via corruption or node failure.

Communication between nodes could be done either by a Master-Slave or a Peer-to-Peer architecture. Master-Slave excels at handling large amount of read requests as they are routed to the slaves. Which makes it read-resilient in the case a master fails. Though not write-resilient as if the master fails so does the writing transaction as another master must be appointed. In addition to this all changes are propagated from the master, making this model unsuitable for write intensive tasks. But it does prevent write conflicts.

Peer-Peer is both suitable for write and read intensive tasks. As it is possible to both write and read from the various servers. All changes are propagated from each node to the other nodes, which might lead to redundant inter node connection. In addition to this it decreases consistency as clients may read different data from different nodes, as the data has yet to be propagated to all nodes. This could be mitigated by using quorums, which is based on majority voting. Meaning that data is written to most of the nodes and read from most of the nodes.

Use case of distribution model and consistency discussed

Data is both read intensive as data is often accessed and write intensive, as public users can rate books. Though the data that requires computations, like author_stats, does not require to be updated regularly due to it not being critical. Thereby this database only requires it to have a weak consistency. Something that is gained by updating those models regularly instead of during each transaction. This would also reduce overhead related to each transaction.

The data is read intensive as it is often accessed. It is also write intensive due to the public view being able to rate books. Which requires updating all instances that contains said rating. Therefore, a Master-Slave replicating architecture combined with sharding would be the best fit here. As Master-Slave allows for controlling consistency related to writing. Which would prevent scenarios where a write could happen to the same key in a Peer-to-Peer architecture. Though this would make it somewhat slower for writing operations. Which in fairness is not a big problem for this site, given that it uses a key value database. Which is not a perfect fit for this kind of data usage.

Using replication would increase the availability of the data though decrease data consistency. This would also make the data more robust as data is replicated across multiple nodes. Combining this with sharding would further distribute subsets of the data which would cluster similar data together whilst spreading the overall load balance for transactions.

Three concrete examples of the operations

Example 1: Insert "Papertowns" by John Green

```
{
    "45641": ObjectID,
    "Paper Towns": String,
    "John Green": String,
    "3.8": Double,
    "014241493X ": String,
    "9780142414934 ": String,
    "eng": String,
    "305": Int,
    "921561": Int,
    "09/22/2009": String,
    "Bloomsbury": String
}
```

1. Add book by key:
    a. Add new book information by bookID key: 45641
    b. Store new key in the database
2. Add book to all_book_keys:
    a. Retrieve all book keys using all_book_keys
    b. Add book with bookID "45641" as key
    c. Store the key array in database under the key: all_books_keys.
3. Add new entry in the array all_books_bucket_X:
    a. Retrieve data from the database with key: all_books_bucket_X
    b. Add new entry with information about bookID: 45641
    c. Store the new array in the database under key: all_books_bucket_X
4. Add book to the author's bibliography:
    a. Retrieve books by author using key: "John Green"
    b. Add new bookID 45641 and corresponding information to array
    c. Store the new book to John Green in the database
5. Calculate all_author_stats_bucket_x:
    a. Retrieve the all_author_stats_bucket_x array
    b. Program the searches through the array for John Green
    c. Then calculates new stat values for "John Green"
    d. Then stores these new values into the database
6. Calculate authorID_stats:
    a. Retrieve authorID_stats by using key: "John Green"
    b. Calculate new values
    c. Store these new values into the database under the same key: "John Green"
7. Increment total number of books
    a. Retrieve total_number_books
    b. Increment value
    c. Store new value under the key total_number_books

Example 2: Deleting "War and peace" by Tolstoy

1. Remove book by key:
    a. Remove the book from the database with bookID: 656

    b. Store the new keys in the database

2. Remove entry from the array all_books_bucket_X:
    a. Retrieve data from the database with key: all_books_bucket_X
    b. The program then searches through the array for key: 656
    c. Remove array entry
    d. Store the new array in database under key: all_books_bucket_X
3. Remove entry from all_book_keys:
    a. Retrieve data from database with key: all_book_keys.
    b. Search through array with bookID: 656.
    c. Remove array entry.
    d. Store new array information in database under key all_book_keys.
4. Remove entry from author's bibliography:
    a. Retrieve dataset of Tolstoy’s books using "Leo Tolstoy" as key.
    b. The program then searches through the dataset for the bookID: 656
    c. Remove the corresponding entry
    d. Store the new dataset to the database
    e. Repeat step a-d for co-authors; "Henry Gifford", "Aylmer Maude" and "Louise Maude"
5. Remove entry from all_author_stats_bucket_x:
    a. Retrieve data from database with key: all_author_stats_bucket_x
    b. The program then searches through the dataset for author stats with key being "Leo Tolstoy"
    c. Then calculates new values
    d. Store the new values under "Leo Tolstoy" in the database
    e. Repeat step a-d for co-authors authors; "Henry Gifford", "Aylmer Maude" and "Louise Maude"
6. Remove entry from authorID_stats
    a. Retrieve data from authorID_stats using the key being "Leo Tolstoy"
    b. Calculate new values
    c. Store the new values under "Leo Tolstoy" in the database
    d. Repeat step a-c for the co-authors; "Henry Gifford", "Aylmer Maude" and "Louise Maude"
7. Decrease total_number_books
    a. Decrease value
    b. Store the new value to the database under the key total_number _books

Example 3: Updating "Paper Towns" by John Green's ratings count and average rating

In the case of more user ratings for a specific book, it is natural to update the average rating and ratings count of a book. This shows how this is done.

```
{
    "45641": ObjectID,
    "Paper Towns": String,
    "John Green": String,
    "4.2": Double,
    "014241493X ": String,
    "9780142414934 ": String,
    "eng": String,
    "305": Int,
    "921750": Int,
    "09/22/2009": String,
    "Bloomsbury": String
}
```

1. Change bookID values:
    a. Change the values in the bookID. Here ratings count and average rating has increased.
    b. Overwrite the already existing key "45641" with the new information
    c. Store book information under the key: "45641"
2. Change information in array all_books_bucket_X:
    a. Retrieve the array all_books_bucket_X
    b. Overwrite the key "45641" with the new values
    c. Store the new updated array all_books_bucket_X to the database
3. Update the author's bibliography:
    a. Retrieve books by using "John Green" as key
    b. Overwrite the key "45641" with the new values and add that to the array
    c. Tore the updated book to the key: "John Green"
4. New values for all_author_stats_bucket_x
    a. Since the values changed are integers, then all_author_stats_bucket_x is calculated with the new values
    b. Then these values are stored into the database
5. New values for authorID_stats
    a. Since the values changed are integer values, then authorID_stats are calculated with the new values.
    b. Then these are stored into the database under the key: "John Green"

Document database

Migrating from a key value directly to a document database requires little effort when it comes to altering the data models. As the key becomes the unique ID in a document database. In addition to this, some of the models may become redundant as inherent functionality in the database already exists thereby replacing those models. Though this would, in most require a full rewrite of the program interacting with the document database.

The key-value database created in the last task could be migrated as this:

Data models(Collection models)

Book:

```
{
    "bookID": ObjectID,
     "title": String,
     "authors": ["authorID": {
                        "authorID": ObjectID,
                        "name": {
                          "f_name": String,
                          "l_name": String
                         }
                      }
                   ],
     "average_rating": Double,
     "isbn": String,
     "isbn13": String,
     "language_code": String,
     "num_pages": Int,
     "ratings_count": Int,
     "text_reviews_count": Int,
     "publication_date": String,
     "publisher": String
}
```

User:

```
{
  "user_id": ObjectID,
  "name": {
      "f_name": String,
      "l_name": String
    }
}
```

User session:

```
{
    "sess_id": ObjectID,
    "session_info": String
}
```

Author:

```
{
  "authorID": ObjectID,
  "name": {
        "f_name": String,
        "l_name": String
        }
  "books": [{ "bookID": ObjectID, "title": String}]
  "highest_rated_book": ObjectID,
  " highest_num_pages_book": ObjectID
}
```

Materialized views:

Totals:

Total number of books:
{"total_number_books": Int}

Total number of sessions:
{"total_number_sessions": Int}

Statistics of author regarding the books:

```
{
  "total_books": Int,
  "total_num_pages": Int,
  "total_ratings_count": Int,
  "total_text_reviews_count": Int,
  "avg_text_reviews_count": Double,
  "avg_ratings_count": Double,
  "avg_num_pages": Double,
  "avg_rating_author": Double
}
```

A document database supports CRUD operations thereby allowing the database itself to search by reference, in the various models. It further allows for some aggregations though this is a relative slow operation and should be avoided in the database design. Which has been done here as the most important and relevant information is condensed inside a single data element as much as possible. Making for ease of access, thereby reducing number of required joins.

This entails several changes for each major operation which has been defined:

Insertion of a new book:

1. Insert new book information the following collections: book and author.
2. Insert new author information into collection: author, if this author does not exist from before.
3. If author exists in the database, then update author information with new book details and possible update relevant information such as highest_rating_book, if need be.

Deletion of a book:

1. Delete where bookID is equal to bookID in collection: Book.
2. Remove corresponding information from the collection author, where bookID is equal to bookID. Might be necessary to recalculate the highest rating as well as highest_num_pages_book. It further might be necessary to remove the entry corresponding to the author, as removing the book entry might mean that the books array is empty. Thereby making author information redundant.

Updating a book:

Updating IDs are not allowed as this would require several other changes that would be overall redundant. The steps are:

1. Update information where bookID is equal to bookID in collection: Book.
2. Updates to either author name or additional authors, requires an additional update to the author collection. In the respective combined sections.

Searching is inherently available in a document database as it supports all CRUD operations. Thereby making searching for specific item/items in specified collections, given certain criteria's, possible. In addition to this, all updates, deletions or insertions that may impact the information in the various material views, must be updated either after a transaction has finished or regularly. Both of which depends on overall uses of the website. As updating the materialized views after each transaction may impact overall performance if there are many such transaction but does add consistency. Though consistency is lost if one chose to update the materialized views regularly, though this increases performance.

More servers

An increase in traffic to this website would require additional horizontal scaling by introducing more nodes. In order to divide the load over multiple nodes as well as increasing the amount of total storage capacity. Something that would require the use of different distribution models in order to satisfy an increase in the amount of reading and writing transactions.

This is done by deploying sharding. Which distributes subsets of data across many nodes, by utilizing horizontal partitioning. Thereby making access and queries local to each node. However, sharding is prone to loss off data due to corruption or damaged nodes.

Hard loss could be mitigated by using replication. A technique that replicates data across nodes. Which contributes to availability and fault tolerance. This is done by having copies of the data across multiple servers. In mongoDB for example config servers and shards as replica sets are deployed. Where the sharded clusters can still perform partial read and writes if a shard replica set turns out to be unavailable.

However, it is important to consider the increased complexity of a sharded cluster infrastructure. This will require more planning and maintenance to the website. Also, if a cluster has already been sharded as the website grows, document databases does not grant any easy method to unshard a sharded collection.

MILESTONE 2

Theory

Key-value database

In a key value database, each entry is identified by a key-value pair where the key is unique. Something that allows for fast reading and writing and simple data formats. Which in turn leads to higher availability. In addition to this it is also highly horizontally scalable. Making it most useful for data that is highly user specific and required to be both read and accessed fast.

The only operations allowed are: Read, insert and update. Thereby limiting overall functionality and operations like aggregations and searching. And as such, makes this database useful for cases like session storage, shopping carts, user defined preferences and user recommendations.

Document database

Data is stored and indexed into documents accessed by unique keys, which pertains to a specific collection. It supports all CRUD operations and can handle both read and write intensive tasks as well. In addition to allowing for direct mapping between data objects in program to various formats like JSON, GSON or XML.

The database allows for dynamic adding and removing columns on demand as there are no enforcement of any concrete schema, as this only exists in the program. Thereby also saving space whilst allowing the data storage to be more flexible. Something that allows the models to reflect program usage more closely. In addition to this, data that are accessed together are stored together making the data more available.

Though this leads to redundant copies of data as the collections are often denormalized as much as possible. Which in turn makes the program handle any reference integrity and makes this non-existent in the database. Something that might lead to inconsistent objects as each field could be different inside the same collection.

Interlinking between collections leads to complex data structures which again slows down the database and the various queries. In addition to this, joins are not always possible and should be avoided. As this would lead to accessing data across the entire cluster.

Columnar database

Each stores data in column family models, which contains rows with similar columns, where each column is identified by a key and value pair. It further allows for dynamic schemas as each column could be added or removed on demand. Further allowing for ad hoc queries as well as optimizing storage size. As only necessary data is stored.

The aggregations are fast and efficient due to the data being clumped together as columns that are accessed together are stored together. Further a columnar database will contain a table comprised of basic metadata about individual blocks on a disk. Thereby leading to quicker computation time as there's no reason to access each stored data element at a time. Aka, it allows for skipping several, which lessens redundant disk access. Though this also means that the database does not/should not utilize joins on data. As this would either be impossible or slow to execute. Due to data also being stored across multiple disks over multiple nodes. However, this type of database is not ideal for writing single rows. Therefore, it is best to write in batches to the database.

Graph database

Stores connections between data, where data might be represented as nodes and connections between nodes as edges. Every node may contain properties specific to that node. Making this structure flexible for showcasing relations between data. Further, this database structure is specialized in traversing all graphs across the database in various ways. Therefore, it is most suited for use cases like: Networks, social relations, IoT relations, and other interconnected data cases.

The data structure makes this type of database slow for aggregations and joins which rely on intimate details contained inside nodes. Therefore, also search queries based on the same details. In addition to this it is also slow for large transactions that span across the whole database including mass analysis queries.

Relational database

A structure that organizes data into tables consisting of columns and rows where a unique key identifies each row. And focuses on using ACID and normalization to increase consistence as well as decreasing required storage space.

Data is structured into categories which in turn makes the database both easy to traverse and search for specific elements. Though, this requires the usage of rigid schemas which makes it difficult to expand and add columns dynamically. In addition to this it is also difficult to scale horizontally, making it only able to scale vertically. Which is costly. Another downside is the lack for complex data types like pictures.

Spark

Spark is and open source cluster-computing framework used for processing batches of data. And as such comes with pre supported integration with most databases. Which means spark will try to unload most of the work to respective databases as is more efficient. In addition to this spark is also fault tolerant as each step in a query could be recomputed on failure.

Spark is meant to be sued for large data batches and not small files; therefore, it is slow when handing many small files. Another problem is that it does not come with its own file management system. As well as keeping data in memory is expensive.

Data models

Our data utilizes the following database structures: Key-value, Document database and Column oriented database.

Memcached Key-Value database

Data models:

```
All sessions:
{"all_sessions_keys": ["sess_id": ObjectID]}
```

```
User session:
{"sess_id": {
    "sess_id": ObjectID,
    "session_info": String
  }
 }
```

MongoDB Document database

Data models:

Book:

```
{
   "bookID": ObjectID,
    "title": String,
    "authors": ["authorID": {
                     "authorID": ObjectID,
                     "name": {
                       "f_name": String,
                       "l_name": String
                      }
                    }
                  ],
    "average_rating": Double,
    "isbn": String,
    "isbn13": String,
    "language_code": String,
    "num_pages": Int,
    "ratings_count": Int,
    "text_reviews_count": Int,
    "publication_date": String,
    "publisher": String
}
```

User:

```
{
  "user_id": ObjectID,
  "name": {
     "f_name": String,
     "l_name": String
   }
}
```

Author:

```
{
  "authorID": ObjectID,
  "name": {
       "f_name": String,
       "l_name": String
  }
  "books": [{ "bookID": ObjectID, "title": String}]
  "highest_rated_book": ObjectID,
  " highest_num_pages_book": ObjectID
}
```

Materialized views:

Detailed Books pr year:

```
"book_pr_year": {
   "bookID": ObjectID,
    "title": String,
    "authors": ["authorID": {
                    "authorID": ObjectID,
                    "name": {
                      "f_name": String,
                      "l_name": String
                     }
                  }
                ],
    "average_rating": Double,
    "isbn": String,
    "isbn13": String,
    "language_code": String,
    "num_pages": Int,
    "ratings_count": Int,
    "text_reviews_count": Int,
    "publication_date": String,
    "publisher": String
}
```

Cassandra Columnar database

Data model:

```
                    Books:
{
"bookID": {
        "authors": Set<ObjectID>,
        "average_rating": Double,
        "num_pages": Int,
        "ratings_count": Int,
        "text_reviews_count": Int,
        "publication_date": Date
        }
}
```

Materialized Views:

```
           All statistics of author:
  {
   "total_books": Int,
   "total_num_pages": Int,
   "total_ratings_count": Int,
   "total_text_reviews_count": Int,
   "avg_text_reviews_count": Double,
   "avg_ratings_count": Double,
   "avg_num_pages": Double,
   "avg_rating_author": Double
  }
```

```
        Individual statistics of author:

{ "total_books": Int}
{"total_num_pages": Int}
{"total_ratings_count": Int}
{"total text_reviews_count": Double}
{"avg_ratings_count": Double}
{"avg_num_pages": Double}
{"avg_rating_author": Double}
```

```
              Book statistics:
  {
   "total_number_books": Int,
   "total_books _year": Int,
   "avg_books_year": Double
  }
```

The materialized views: "All statistics of authors" and "Individual statistics of authors" contains statistics for each author. The statistics calculated are:

- Total number of books
- Total number of pages written
- Total ratings and text reviews count
- Average of the total rating and text review counts
- Average number of pages written
- Average rating of each author

The website also calculates statistics for the books in the system and is represented in the materialized view "Book statistics". The following calculations are represented:

- Total number of books in the system
- Total number of books released each year
- Average number of books each year

Discussion

Polyglot structure allows for utilizing specialized databases where required. Thereby increasing both performance and data storage. Tough also leading to a more complex system. As well as increased knowledge requirements about each database application. In addition to higher interconnections between them. Which requires interoperability between different systems. Something that is difficult, though could be solved by utilizing spark. As Spark already comes with functionality for interacting with different databases. Another possibility would be to add the functionality into the backend part of the website.

Interaction between and to databases

Utilizing spark would reduce both overall developer time as well as being easy to both integrate and setup. In addition to being available on multiple platforms like Python, Java and Scala. Furthermore, it is built for clustering meaning it scales horizontal easily. Something that would make the system easier to upgrade as well as handle upcoming changes. Both with regards to the data as well as the internal data models.

Integrating spark as a medium between databases may add transaction time both to processing data and handling data access between multiple databases. Though this would not be noticeable as possible delays would be small. It may also require both additional hardware and external tools for debugging and testing queries before production.

An alternative to spark would be to custom build a program to handle the connections and updates between different databases. Something that could possibly speed up transaction time but might make the system more prone to errors. In addition to having a longer developer time related to the project, as well as securing the program. Therefore, it is recommended to utilize spark for this task.

Key-Value

Key-Value database was chosen to handle session data as this data requires both fast read and fast write access. As well as often created and deleted, and sometimes updated. Where fields may be either added or removed over time making it highly modular data. Though it does not require any complex aggregations nor more functionality than, reading, writing or updating an entry. Thereby making this database structure most suited.

An admin might need to access all sessions id in storage, which means there is a requirement for a model that satisfies this. Thereby making the model: all_sessions_keys needed. The model does not contain intimate details about each session only the session id. Thereby saving space and adding consistency. Though this model might become a larger dataset where it would increase transaction time. In which case one could apply bucketing in order to size per transaction, leading to decreased transaction time and higher availability as data are accessed in smaller buckets.

Document Database

Intimate details about books and authors are stored in a document database. As this allows data that are often accessed together to be stored together in the same collection thereby also located together. In addition to allowing for embedding of other documents inside a document as well as dynamic schema. Which increases availability of the data making it easier to access.

Stored data are searchable either by per entry or by collection field. Moreover, the database allows for queries that operate on per item in the database. It further allows for dynamic schemas inside collections. Making it appropriate for storing detailed data such as books, authors and users.

Each data model depicts data that are often accessed together such as a book and it corresponding author, or a single user and correlated data. Moreover, utilizing the hashed password for the user as a key for those values allows for both faster and fail fast access.

User data could've been located inside a relational database as this is data might require ACID in future and does not necessarily change over time. Though in this case it was chosen to store it inside a document database in order to alleviate number of required database structures. Furthermore, the data model is not complex enough for this to matter.

In order to increase further availability and consistency would be to implement sharding and setting a higher degree of replication, as was done in milestone 1. Something that would not affect overall consistency as data in this database is not updated often enough to be a noticeable downside to speed or consistency. Thereby making it have strong consistency.

Columnar Database

Columnar databases specialize in analytical aggregations making it suitable for queries related to statistics of an author's book releases. Along with statistics about the total number of books in the system. As it includes queries that involve reading a larger subset of the cluster data.

The data model named "Books" is designed so that the data is kept close together and not nested. This will lead to a database that is efficient at reading data and has fast query performance. A columnar database is used for the optimalization of performance, not storage place. Hence why the book data model only includes the necessary information needed to calculate what is represented in the materialized views.

In addition to this the data will be written in bulks to the database as it is not updated on a per transaction level. Since it is not crucial to update after every transaction, the materialized views are updated periodically.

Requirements

Utilizing these 3 types of database structures leads to additional requirements related to training and usage of each database. As now each developer must learn how to setup, operate and optimize the various databases. Both Cassandra and MongoDB have certificates for developers, making them require additional studying and developer training for usage. Memcached does not have any certificates, nor is it needed. As it has detailed documentation as well as a large community that could be utilized for help.

All these databases are open source, meaning they are free to use. Thereby not requiring any licensing. In addition to having large user communities for help and being thoroughly documented. Though one does require hardware and as such increase requirements related to hardware and infrastructure maintenance.

Security

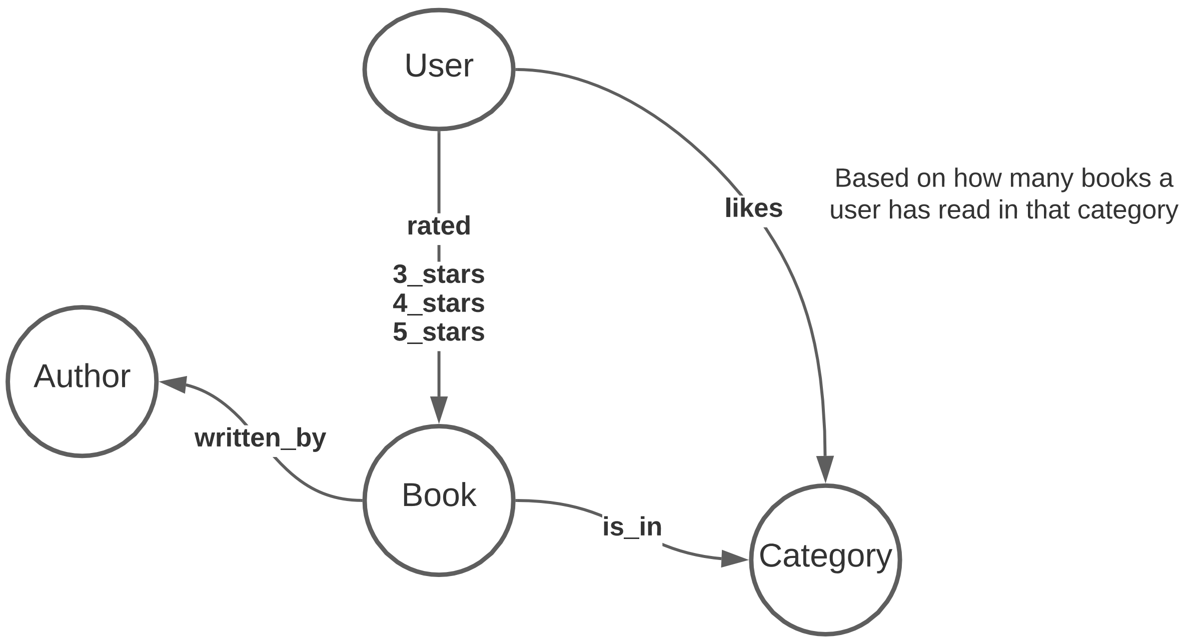
All these databases contain little security compared to relational databases and is something that must be handled in the software applications. Therefore, this step may require additional expertise and work in order to satisfy required security steps.

Maintenance

Infrastructure grows in conjunction with website popularity. Thereby requiring additional hardware, which may require frequent monitoring and switching out various parts.

Further development

Current functionality of the website could be expanded in various ways. One may be allowing users to log in not only as an admin. Where they would be allowed to rate and favourite books. Thereby not only having an admin view as that limit's potential functionality. Custom user views would open for additional functionality like user recommendations. Something that would require utilizing a graph database for linking user rated books to category and author, in order to recommend similar books to the user. Thereby increasing functionality and user traffic to the website. An example of a suited graph database for handling this functionality is visualized under:



Dataset used

Goodreads-books. (2014, June 14). Kaggle.
https://www.kaggle.com/jealousleopard/goodreadsbooks