Ca' Foscari University of Venice

Department of Environmental Sciences,
Informatics and Statistics

MCS Computer Science and Information Technology CM-90

[CM0481] Software Performance and Scalability

IMDb: Performance and Scalability Analysis

Authors:
Michele Lotto - 875922
Daniel Jader Pellattiero - 903837

Rev. 26/06/2024

Contents

List of Tables

List of Figures

Abstract

This report presents a comprehensive account of the implementation and performance testing process conducted on a web application that emulates the IMDb website[1].

The report begins with an overview of the system implementation, in which the technologies utilized are outlined. Subsequently, a theoretical analysis of the queueing system network is presented, in which the implemented system is characterised. The load test conducted on the system will be subjected to a thorough examination, including an analysis of its theoretical and empirical findings.

In conclusion, a potential architectural solution is evaluated for its potential to enhance the scalability of the tested system.

It should be noted that the implementation of the system, along with all automated programs and processes referenced in this report, is fully documented and accessible on the GitHub repository[2].

[1] https://www.imdb.com/

[2] https://github.com/danieljaderpellattiero/unive-imdb

Chapter 1

System implementation

The project entails the development of a three-tiered web application, comprising a database, a RESTful API and a website. The following sections will present an analysis of each of these components, with a focus on the technologies and design choices implemented.

1.1 Database – MongoDB

MongoDB is a popular NoSQL document-oriented database that is designed to store and manage large volumes of structured and unstructured data.

The database was selected as the optimal choice for managing the IMDb Non-Commercial Datasets[1] due to its adoption of a schemaless data modelling, which enables the management of any non-normalised records or fields within the data. Furthermore, the usage of JSON format for storing the records allows for enhanced efficiency in interactions with the backend.

1.1.1 Data analysis

Each dataset, comprising a UTF-8-encoded tab-separated values (TSV) file, was subjected to analysis using the Python Pandas library. The primary steps of the data analysis process were as follows:

1. The replacement of missing values (denoted by `'\N'`) with default values of a type consistent with the column domain to which they belong.

2. The removal of records that lack some data fields.

3. The normalisation of fields containing arrays of elements.

4. The potential replacement of table indexes.

Further modifications were implemented to the tables, however, as these affect the database schema, they will be addressed subsequently.

Once the preliminary phase of preparing the datasets was complete, it was decided that the files should be exported in the JSON format, allowing them to be imported into the database. Additionally, the files were exported also in the Parquet format, enabling

[1] https://developer.imdb.com/non-commercial-datasets/

them to be uploaded to the GitHub repository via the GitHub Large File Storage (LFS) facility[2].

1.1.2 Collections' schema

MongoDB represents objects using BSON (Binary JSON) types, which are binary-encoded serialisations of documents that adhere to the JSON format. As each table in MongoDB is translated into the concept of a document collection, the diagrams of the respective collections imported from the datasets obtained post data analysis are presented below. Furthermore, supplementary notes on the refactoring of the tables are provided.

Table 1.1: '*title.basics*' collection schema

Field	BSON type	Index	Index type
_id	String	yes	unique, ascending
titleType	String	no	-
name	String	no	-
nameEng	String	no	-
isAdult	Boolean	no	-
genres	Array[String]	no	-
startYear	32-bit integer	no	-
endYear	32-bit integer	no	-
runtime	32-bit integer	no	-
rating	Double	no	-
votes	32-bit integer	no	-

N.d.R.: A collection was created ad hoc for records with the field *'titleType'* equal to *'tvEpisode'*.

Table 1.2: '*title.akas*' collection schema

Field	BSON type	Index	Index type
_id	ObjectId	yes	unique, ascending
titleId	String	no	-
ordering	32-bit integer	no	-
region	String	no	-
name	String	no	-
nameLower	String	yes	ascending

N.d.R.: *'language'*, *'types'*, *'attributes'*, *'isOriginalTitle'* fields dropped.

[2]https://git-lfs.com/

Table 1.3: '*title.crew*' collection schema

Field	BSON type	Index	Index type
_id	String	yes	unique, ascending
writers	Array[String]	no	-
directors	Array[String]	no	-

Table 1.4: '*title.episodes*' collection schema

Field	BSON type	Index	Index type
_id	String	yes	unique, ascending
titleId	String	yes	ascending
name	String	no	-
nameEng	String	no	-
season	32-bit integer	no	-
episode	32-bit integer	no	-
isAdult	Boolean	no	-
genres	Array[String]	no	-
startYear	32-bit integer	no	-
endYear	32-bit integer	no	-
runtime	32-bit integer	no	-
rating	Double	no	-
votes	32-bit integer	no	-

Table 1.5: '*name.basics*' collection schema

Field	BSON type	Index	Index type
_id	String	yes	unique, ascending
fullName	String	no	-
birth	32-bit integer	no	-
death	32-bit integer	no	-
professions	Array[String]	no	-

N.d.R.: *'knownForTitles'* field dropped.

Table 1.6: '*title.principals*' collection schema

Field	BSON type	Index	Index type
_id	ObjectId	yes	unique, ascending
titleId	String	yes	ascending
ordering	32-bit integer	no	-
personId	String	no	-
job	String	no	-
characters	Array[String]	no	-

N.d.R.: *'category'* field used to fill empty *'job'* fields; then dropped.

1.1.3 Schema indexes and bulk data insert

As can be seen from the tables above, in addition to the standard unique indexes, it was decided to add additional ones to optimise the queries requested by the backend. Given the significant use of wildcard searches based on regular prefix expressions, it was decided to implement ascending indexes on several *'String'* type fields to drastically reduce search times without having to resort to a text index, which would take up much more space due to the tokenisation and stemming of the fields.

In order to perform a bulk insert of the approximately 42.5 million records contained in $\sim$ 1.6 GB of JSON files, it was decided to write a script in Python that would take advantage of a multithreaded execution of the `mongoimport` command-line tool provided by MongoDB. The overall script is responsible for generating the database's collections, populating them and finally creating indexes.

1.1.4 Technical details

The database was configured as a locally managed instance of MongoDB, thus avoiding the utilisation of the cloud version of the service (Atlas). The service's local deployment facilitated the monitoring and control of its resources, as well as the measurement of its performance. These factors collectively contributed to the generation of more reliable results during the load tests that were conducted.

Technical insights into the environment are presented below.

- MongoDB 7.0.11 (Community Server service)
- MongoDB Compass 1.43.3 (database UI)
- MongoDB Shell 2.2.9 (database CLI)

1.2 RESTful API – Express

The web application's backend was developed using the Express framework, a Node.js web application framework that was identified as the most widely utilised in the "State of JavaScript 2023" survey[3].

[3]https://2023.stateofjs.com/en-US/other-tools/#backend_frameworks

A variety of endpoints were incorporated into the API, enabling a constrained yet still functional utilisation of the web interface. The user-accessible API calls are enumerated below, along with a concise description of each.

1. `localhost:3000/search/preview/:title` – Returns the top 4 most voted titles that match the search query prefix.
2. `localhost:3000/search/:title` – Returns the most voted titles that match the search query prefix; the results are paginated in groups of 8.
3. `localhost:3000/search/episodes/:title` – Returns the episodes of a specific title that matches the *'titleId'* parameter.
4. `localhost:3000/title/:id` – Returns the details of a specific title that matches the *'_id'* parameter.
5. `localhost:3000/episode/:id` – Returns the details of a specific episode that matches the *'_id'* parameter.

The official MongoDB driver for Node.js (Typescript) `mongodb@6.8` is responsible for handling queries from the backend to the database. We deliberately decided not to utilise any form of Object-Relational/Document Mapping (e.g. `Mongoose` or `Prisma`) due to the suitability of the JSON record format to the application domain model and potential performance overheads.

Aggregation pipelines are used to facilitate the execution of queries to the database. These pipelines allow the definition of queries through the use of a sequential list of stages, thereby simplifying the grouping and sorting of data and providing control over the execution times of the individual stages within the pipeline.

1.3 Website – Vue and Tailwind CSS

The web interface of the system was realised using Vue.js "The Progressive JavaScript Framework", a well-known framework used for building SPAs (Single Page Applications) renowned for its component-based architecture and reactive data binding system. The aesthetic component was addressed through the utilisation of Tailwind CSS, a utility-first CSS framework that can be suitably integrated with Vue. A bundle of the website, optimised for deployment on a static hosting service, was created at the end of the development process using Vite[4]. This design choice ensures that the frontend is efficiently packaged into static files, which can be served directly to users without requiring server-side processing. Consequently, during load testing with JMeter, the focus has been exclusively on the backend API endpoints, as the frontend's static nature does not impose additional load on the server.

Thumbnails of the web interface are available on GitHub[5].

[4]A modern front-end build tool that facilitates optimized production builds through the utilization of ES hot module replacement (HMR).

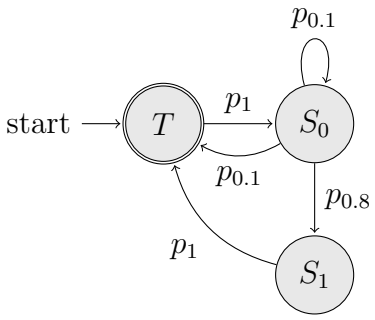
[5]https://github.com/danieljaderpellattiero/unive-imdb/tree/frontend/thumbnails

Chapter 2

Test case identification

2.1 High-level test case design

This brief chapter aims to present the test case designed to perform the load test of the system. In order to illustrate the user behaviour modelled by the test, we define a trivial finite-state automata with reference to the endpoints implemented in the RESTful API.



The above automata has to be interpreted in the following manner:

- The user initiates a title search by name match, regardless of the customer language locale. (`search/:title`)
  - In 80% of cases, the search continues with a precise request for the title information. (`/title/:id` *or* `/episode/:id`)
  - In 20% of cases, either the user terminates the search process pre-emptively, without finding the title, thus entering a dormant (thinking) state; or it continues the search by hitting the same endpoint though the use of pagination. (`search/:title?page=n`)
- Upon receipt of the searched title information, the user is satisfied and returns to a dormant (thinking) state before repeating a new search.

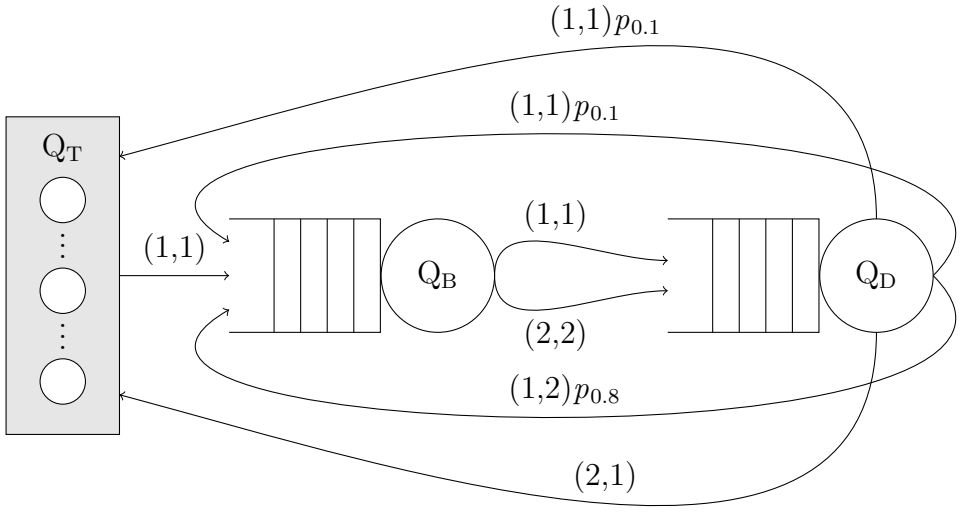
Chapter 3

Queueing network model

3.1 Queueing network components

This chapter presents a comprehensive theoretical analysis of the implemented model. The following illustration depicts the system components involved in the load test in the context of queuing theory.

For the purposes of this illustration, let Q_T represent the thinking station, Q_B represent the backend, and Q_D represent the database.



It is possible to derive several characteristics of the system under consideration from the illustration above.

The system is characterised by a closed and interactive nature, free of stability conditions, and a constant number of users that dictate the nature of the workload. A further aspect that emerges relates to the multiclass nature of the jobs within the system, as indicated by the labels placed at the station entry and exit arcs.

Subsequently, an in-depth examination will be conducted on the distinct service time/demand and probabilistic routing for the two classes of jobs.

In conclusion, the network under analysis can be classified as a BCMP, exhibiting a closed topology and utilising "Processor Sharing" (PS) as its scheduling discipline.

Prior to examining the nature of the queues that characterise our network and their associated scheduling disciplines, we proceed with the formalisation and resolution of the network's traffic equations. These equations will then be essential in producing the theoretical analysis of the system.

3.2 Traffic equations

The system's traffic equations are presented below.

$$\begin{cases} e_{B1} = e_{T1} + (0.1 \times e_{D1}) \\ e_{B2} = 0.8 \times e_{D1} \\ e_{D1} = e_{B1} \\ e_{D2} = e_{B2} \\ e_{T1} = e_{D2} + (0.1 \times e_{D1}) \end{cases} \Rightarrow \begin{cases} e_{B1} = 1.1111 \times e_{T1} \\ e_{B2} = 0.8889 \times e_{T1} \\ e_{D1} = 1.1111 \times e_{T1} \\ e_{D2} = 0.8889 \times e_{T1} \end{cases}$$

The system of equations above demonstrates that our routing is irreducible, which implies that any queue can be accessed from any other queue within the network. From an analytical standpoint, our system is under-determined, indicating the existence of an infinite number of solutions. The non-trivial solutions can be derived from the non-zero multiplicative coefficients.

Moreover, it is essential to highlight that within this context, where the system under consideration is an interactive queueing network, it is imperative to impose an additional constraint on the previous system, specifically $e_{T1} = 1$, given that this is the thinking station $\mathrm{Q_T}$.

3.3 Queueing systems analysis

This section will provide a brief overview of the components of our network, with reference to Kendall's notation.

3.3.1 Thinking station

The thinking station can be represented by a queuing system described with the "G/D/∞/IS" notation. The distribution of inter-arrival times for jobs at the thinking station is unknown, but the distribution of service times is known, as the jobs within it remain in service (*thinking state*) for a fixed constant time, known as thinking time. By definition, the thinking station is equipped with an infinite number of servers, indeed an undefined number of jobs can access it. Consequently, the service discipline of this system is of the "Delay center" or "Infinite Servers" type, as all the jobs remain in an idle state in parallel.

3.3.2 Backend station

With regard to the JavaScript runtime environment Node.js, despite its reputation as a "purely single-threaded" execution environment due to its Event Loop, it is in fact capable of offloading I/O blocking operations to separate threads or the operating system, thus allowing the main thread to continue processing other tasks. The library that permits the dispatching of blocking operations to a thread pool is called `libuv` and it belongs to

the Chrome V8 Engine. Furthermore, Node.js has also implemented the `worker_threads` module, which allows for further enhancements of performance by the utilisation of multi-core processors for CPU-bound operations.

In light of the aforementioned considerations, it can be concluded that the RESTful API functions as a "G/G/1/PS" queueing system. This is due to the fact that no manual instantiation of multiple worker threads was performed for the API, and thus a unique main thread is responsible for handling all incoming requests through a "Processor Sharing" scheduling discipline.

3.3.3 Database station

With regard to MongoDB, an examination of the "Production Notes" section reveals that its WiredTiger storage engine is inherently multithreaded. In particular, the total number of active threads (i.e. concurrent operations) relative to the number of available CPUs can impact performance as follows:

- The throughput increases in direct proportion to the number of concurrent active operations, up to the overall number of CPUs.
- The throughput decreases as the number of concurrent active operations exceeds the number of CPUs by a certain threshold amount.

The optimal number of concurrent active operations for a given application can be determined through the measurement of throughput.

In order to ascertain the exactness of the information provided in the documentation, an empirical test was conducted to monitor the thread pool instantiated by the MongoDB service in both idle and stressed states.

The results obtained from the Linux command-line utility `htop` are presented below.

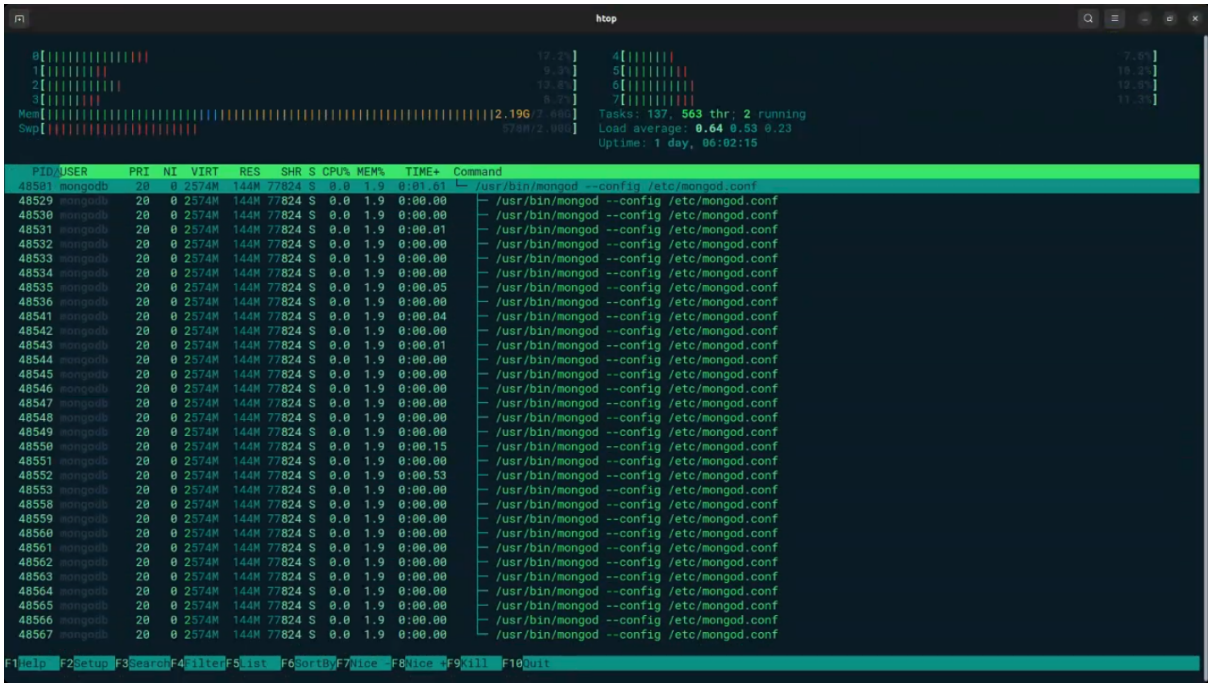


Figure 3.1: MongoDB thread pool in idle state

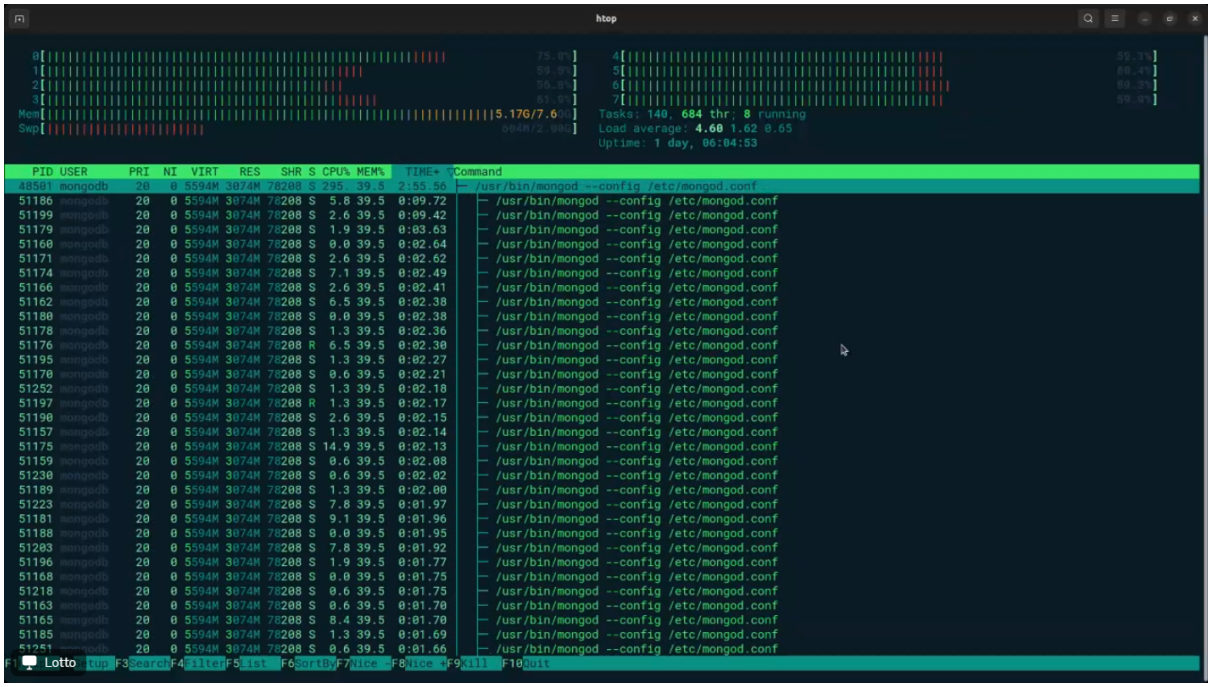


Figure 3.2: MongoDB thread pool under load conditions

In conclusion, the database can be identified as a "G/G/#C/PS" queueing system, where "#C" represents the number of cores. The inter-arrival times and service times are both unknown, however it can be stated that the database leverages all CPU cores with a "Processor Sharing" service discipline.