CLI (Bash) and Data Management for Big Data

Project 1, COMP30770 Programming for Big Data (Spring 2021)

Rajit Banerjee, 18202817

Introduction

This project explores the practical applications of the command line, Bash, and data management for Big Data. The report covers the following sections:

- Cleaning a Dataset with Bash
  - Processing the Reddit dataset: a snapshot of posts (a CSV file containing 68745 lines), to prepare it for analysis using database management systems.
- Data Management
  - Using the cleaned dataset for data management (MySQL and MongoDB).
  - Performing a range of SQL and NoSQL queries.
- Reflection
  - CLI for Big Data
  - Relational (SQL) vs. non-relational (NoSQL) database systems
  - Review on 'Dynamo: Amazon's highly available key-value store'

The scripts and data files have been submitted alongside this report, as described in README.md.

1. Cleaning a Dataset with Bash

Raw dataset: `data/reddit_2021.csv`
Tools: Bash on Ubuntu 20.04 (WSL2 on Windows 10)

1.0. Dealing with protected commas

Script: `./00-replace-protected-commas.sh` (`real` execution time: ~`0m0.151s`)
The original raw CSV file has a single cell (in *title*) that contains a comma protected by double-quotes (`"1,760"`). However, this interferes with the comma delimitation of the file, and hence the script searches (with `grep`) the dataset for any commas surrounded by double quotes, and replaces them with semi-colons using `sed` in-place (`-i`). All cleaning steps are applied to a copy of the dataset (`data/reddit_2021_clean.csv`).

1.1. Dropping index variables and NSFW check column

Script: `./01-drop-index-and-nsfw.sh` (`real` execution time: ~`0m0.198s`)
A simple `cut` command, using a comma delimiter and the `--complement` flag, is used to drop the unnecessary columns, numbered 1, 2, 3 (index variables) and 34 (*over_18*, i.e., NSFW check).

1.2. Dropping empty columns

Script: `./02-drop-empty-cols.sh` (`real` execution time: ~`0m6.974s`)
Iterating through the (comma-separated) columns in the dataset, the non-empty cells for every column are extracted with the `cut` and `tail` commands. If this variable is itself empty (check with `-z` condition) for any column, then the column number is appended to a string containing a list of columns to be removed. This string of comma-separated numbers is passed to the `-f` (field) flag of `cut`, which is used with the `--complement` flag to

drop all such empty columns. In total, 11 columns (#1, 2, 16, 17, 22, 30, 36, 37, 38, 48, 49) are removed here, and the `data/reddit_2021_clean.csv` file is rewritten.

1.3. Dropping uninformative (single-valued) variables

Script: `./03-drop-single-val-cols.sh` (`real` execution time: ~`0m6.946s`)
Similar to the previous part, we iterate through the column numbers. The `cut` and `tail` commands extract a column (without heading); `sort` and `uniq` are used to group by value, then `wc -l` counts lines, i.e. the number of unique values in the column. A blank cell is considered as a value as well. In case a single unique value is found, the column number is appended to a comma-separated list which is used to remove all such uninformative columns (#15, 18, 27, 32) using `cut` and `--complement`. After this step, the clean CSV file should be left with a total of 39 columns.

1.4. Converting date columns to month name

Script: `./04-sec-to-month.sh` (`real` execution time: ~`0m1.055s`)
The columns *created_utc* and *retrieved_on* contain Unix timestamps. Both column numbers are identified via linear search (*created_utc*: #8, *retrieved_on*: #27), and then the powerful `awk` is used to replace all epoch times in these columns. Column substitution is made simple by `awk`, and its built-in time formatting function `strftime()` is used to convert the seconds into full month names (format string: `"%B"`).

```
$ head -n 3 ./data/reddit_2021.csv | cut -d, -f13,43 | column -ts,
created_utc   retrieved_on
1504224000    1507285925
1504224013    1507285925
```

```
$ head -n 3 ./data/reddit_2021_clean.csv | cut -d, -f8,27 | column -ts,
created_utc   retrieved_on
September     October
September     October
```

As shown in the sample rows above, the raw dataset contains the dates as epoch times (columns #13, 43), which have been converted to month name in the cleaned CSV file (columns #8, 27).

1.5. How many posts have been made in each month?

Script: `./05-count-posts-per-month.sh` (`real` execution time: ~`0m0.458s`)
Upon identifying the date column numbers, the `tail` and `cut` commands extract the column values (for *created_utc* and *retrieved_on*). Then `sort` allows the `uniq -c` command to group by value and count the frequency of Reddit posts by month. On running the script, we see that all 68744 posts in the dataset were created in September.

1.6. Cleaning the *title* column

Script 1: `./06ab-title-lower-no-punc.sh` (`real` execution time: ~`0m0.227s`)
First, the *title* column is `cut` out and written to a separate file (`data/titles.txt`) for processing. Next, `tr` changes all titles to lowercase (`"[:upper:]" "[:lower:]"`) and removes punctuation marks (`-d "[:punct:]"`).

Script 2: `./06c-remove-stop-words.sh` (`real` execution time: ~`0m26.841s`)
A list of stop words (`data/stopwords.txt`) (1298 lines) is obtained from:
https://raw.githubusercontent.com/igorbrigadir/stopwords/master/en/alir3z4.txt.

These words, which have little or no real meaning, are removed from all titles by building a regex group all stop words from the downloaded file, using `tr` to separate words with `|`. Then, the `sed` command is used to remove all occurrences of these words from titles. The file (`data/titles.txt`) now contains all modified post titles.

Script 3: `./06d-reduce-to-stem.sh` (`real` execution time: ~`0m52.508s`)
The stem words dictionary file (`data/diffs.txt`) (29417 lines) is obtained from:
http://snowball.tartarus.org/algorithms/english/diffs.txt

First, this dictionary file is processed, and each word and its "stem" are stored in a Bash associative array for fast lookup in the next step (note: only words which are not identical to their stems are inserted). Then, iterating through all the titles (`data/titles.txt`) (68744 lines), as well as the words in every title, the "stem word" (from the dictionary) is appended to the output file if it exists, else the original word is kept. The output is written to a separate file called `data/stemmed_titles.txt`.

Script 4: `./06e-place-clean-titles.sh` (`real` execution time: ~`0m0.773s`)
Finally, the stemmed titles need to be placed back in the full cleaned dataset. The column number for *title* in `data/reddit_2021_clean.csv` is identified by linear search. Then, an `awk` substitution is used to extract the first (and only) column from `data/stemmed_titles.txt`, which replaces the existing *title* column in `data/reddit_2021_clean.csv`. The post titles have now been processed and updated in the clean CSV file!

Comparison of the *title* column, before and after cleaning in question 6:
(lowercase, no punctuation, no stop words, words stemmed wherever possible)

```
$ head -n 5 ./data/reddit_2021.csv | cut -d, -f56
title
Mix vodka & Kahlua & youll have this colorful classic
Like painters modern artists in this form turned to abstraction as in David Smiths work in steel
A rock variety of game hen is named for this southwestern county of England
A civil suit brought because of damage to the left side of a ship

$ head -n 5 ./data/reddit_2021_clean.csv | cut -d, -f37
title
mix vodka kahlua colorful classic
painter modern artist form abstract david smiths steel
rock varieti game hen name southwestern counti england
civil suit brought damag left ship
```

The line count for all files in the `data/` directory is shown below.

```
$ ls | xargs wc -l
   29417 diffs.txt
   68745 reddit_2021_clean.csv
   68745 reddit_2021.csv
   68745 stemmed_titles.txt
    1298 stopwords.txt
   68745 titles.txt
  305695 total
```

In particular,

- `reddit_2021.csv` (original dataset),
- `reddit_2021_clean.csv` (processed and cleaned data),
- `titles.txt` (*title* column extracted from CSV file, in lowercase, with no punctuation marks or stop words), and
- `stemmed_titles.txt` (titles reduced to stem words),

all have the same number of lines, i.e., 68745 (1 heading + 68744 entries).

2. Data Management

Clean dataset: `data/reddit_2021_clean.csv`
Tools: Bash, MySQL and MongoDB in Docker (`registry.gitlab.com/roddhjav/ucd-bigdata/db`)

The following scripts were run to set up a Docker container with MySQL and MongoDB support.

- `./docker-create.sh`: Create a new container called `comp30770-db` for the image mentioned above (`.../ucd-bigdata/db`), after stopping and removing any existing containers with the same name.
- `./docker-cp-files.sh`: Copy the relevant MySQL and MongoDB database creation and population scripts, as well as the cleaned dataset `data/reddit_2021_clean.csv` over to the container.
- `./docker-start.sh`: Start a Bash prompt in the container's `/root/` directory.

2.1. Database creation in MySQL

Script: `./07-mysql-create-db.sh` (`real` execution time: ~`0m0.338s`)
The script creates a new database called *reddit* after dropping any existing namesakes, using the `CREATE DATABASE` and `DROP DATABASE IF EXISTS` statements for SQL. Then, the following three tables are set up using `CREATE TABLE`:

- user(author id, author, author_cakeday)
- subreddit(subreddit)
- post(id, author_id↑, subreddit↑, created_month, title).

All fields in these tables are set to accept alpha-numeric values (`VARCHAR`) of varying lengths (which were calculated upon exploring the maximum length of the dataset's cells for the columns in question). Some fields like user.author_cakeday and post.title are nullable, and don't have the `NOT NULL` setting. `PRIMARY KEY` was used to set the unique table identifiers (underlined above), and `FOREIGN KEY...REFERENCES` was used to establish links between the tables (foreign keys marked by ↑). The script also uses `DESCRIBE` to display the table details.

2.2. Populating the *reddit* database

Script: `./08-mysql-populate-db.sh` (`real` execution time: ~`0m20.018s`)
First, the column numbers for all the required fields for the 3 SQL tables are identified by iterating through column headings. The column *created_utc* from the cleaned dataset is renamed to *created_month* for MySQL. The results (column numbers) are stored in an associative array.

Next, for each table variable (`user`, `subreddit`, `post`), custom formatting is applied to match the requirements for MySQL `INSERT INTO` statements. The required columns are extracted from the CSV with `awk`, then using `sed`: every field is double-quoted, empty strings are replaced with `NULL`, and finally, every line is enclosed in brackets `()`, with a trailing comma. Theoretically, it should be possible to insert these formatted strings into their respective MySQL tables. However, due to Bash argument list length limitations, the row insertion is processed in batches of 1000 lines from the CSV.

By iterating through the number of rows in the CSV file, the beginning and ending row numbers for the current batch are calculated. Next, `sed` and `tr` are used to extract the specific rows from the full table strings created earlier. Then, `mysql`'s `INSERT INTO` statements are used to insert the current batch of entries into the *user*, *subreddit* and *post* tables of the *reddit* database. `IGNORE` is added to `INSERT` statements for the *subreddit* and *post* tables to avoid duplicate primary key errors.

Formatting the table strings as necessary for `INSERT INTO`, then inserting in batches as described above, ensures far better execution speeds compared to a brute force approach of iterating through every row in the CSV file and using a `mysql` client access for each.

2.3. MySQL queries

The following questions about the database were answered with SQL queries. The `LIMIT` clause can be used when executing these queries to display a subset of the results.

- List of all author names.

  ```
  SELECT author FROM user;
  ```

  All rows are displayed for the *author* field belonging to the *user* table.

- List of all posts' title with their author's name and the subreddit they were posted in.

  ```
  SELECT post.title, user.author, post.subreddit
  FROM post, user
  WHERE post.author_id = user.author_id;
  ```

  The *title*, *subreddit* (from *post*) and *author* name (from *user*) are selected by joining the two tables (Cartesian product). The *post* and *user* tables are joined on the *author_id* attribute, which is a primary key for *user* and a foreign key in *post.*

- List of (subreddit, month) pairs, and the number of posts made in the subreddit during this month.

  ```
  SELECT subreddit, created_month, COUNT(*)
  FROM  post
  GROUP BY subreddit, created_month;
  ```

  The *subreddit* and *created_month* attributes are selected from the *post* table, and grouping is done by the same *subreddit*, *created_month* pair. This displays only the unique (subreddit, month name) pairs, and selecting `COUNT(*)` displays the number of posts for each such pair. This effectively counts the number of posts in every subreddit, grouped by month.

2.4. Database creation and population in MongoDB

Script: `./09-mongo-populate-db.sh` (`real` execution time: ~`12m5.684s` (full CSV), ~`0m21.884s` (10K rows)) Due to the NoSQL design of MongoDB, we don't need to worry about separate tables, and can insert the full clean dataset into a 'collection'. Similar to the MySQL database population script, row insertion is done in batches to ensure faster speeds. First, any preexisting collection named *allposts* is removed from the *reddit* database. Then, the CSV column headings are read into an array. Since `awk` was useful in formatting the rows for MySQL, a similar approach is used to build up the `awk` command for MongoDB, by iterating through CSV column numbers, and setting up the possibility of inserting the column name and a colon before every cell in the dataset.

Using the command portion built above, `awk` is used to format each row like a JSON object, so that `mongo` can insert it as a document. The `sed` command is used again to double-quote every key and value, then enclose every row in braces `{}`. A few more formatting steps are applied to ensure successful insertion into the database, which are described as comments in the script. The documents are now organised in batches of 100 (due to Bash argument list length limits), and `mongo`'s `db.allposts.insert()` statement is used to insert them.

An alternative and much faster method to load the full CSV file into a MongoDB collection is the following:

```
# mongoimport --type csv -d reddit -c allposts --headerline --drop ./data/reddit_2021_clean.csv
```

2.5. Translating SQL queries to NoSQL for MongoDB

The MySQL queries from earlier were translated into MongoDB queries as follows:

- List of all author names.

  ```
  db.allposts.find({}, {_id: 0, author: 1})
  ```

  From the *allposts* collection (which contains all rows and columns from the clean CSV), the command fetches all the documents (rows), but then only displays the *author* field (by setting it to 1). MongoDB's generated ObjectID field is ignored by setting `_id` to 0.

- List of all posts' title with their author's name and the subreddit they were posted in.

  ```
  db.allposts.find({}, {_id: 0, title: 1, author: 1, subreddit: 1})
  ```

  Similar to the previous query, all documents in the collection are queried, but this time, it's the *title*, *author* and *subreddit* column on display. Unlike SQL, there is no concept of joining tables, since a single collection contains all the CSV records here.

- List of (subreddit, month) pairs, and the number of posts made in the subreddit during this month.

  ```
  db.allposts.aggregate({$group: {
      _id: {"subreddit": "$subreddit", "month": "$created_utc"},
      "count": {$sum: 1}
  }})
  ```

  An aggregation operation is applied to the *allposts* collection, where grouping is done to get all the unique (subreddit, month name) pairs. The `$sum` operation is applied in the grouping stage to count the number of posts in every subreddit, grouped by month.

2.6. Discussing modifications to the database structure

If we were to add multiple subreddits per post:

- Do the stored records in the database need to be altered?
  - **MySQL**: Yes. Currently, subreddits and posts have a one-to-many relationship, since a post can only be in one subreddit, but a subreddit can have multiple posts. However, the database schema would need to be changed to accommodate a many-to-many relationship, which can be achieved by creating a third table, with both subreddit and post id as foreign keys.
  - **MongoDB**: No. MongoDB being a NoSQL database doesn't have a rigid schema. Hence, multiple documents with all fields identical, except the subreddit related fields (to accommodate multiple subreddits), can be inserted. The preexisting records in the database do not need to be altered.
- Will the previously discussed queries still work?
  - **MySQL**: No (except query #1, to list author names). The associative table which enables the many-to-many relationship between posts and subreddits will need to be joined in order to access the subreddit name. Moreover, the foreign key in the post table will need to change to reference an entry in the associative table.
  - **MongoDB**: Yes. All three queries will work since no change was made to the database structure.

3. Reflection

3.1. CLI for Big Data

More often than not, we make the error of focussing on flashy tools, instead of looking for the simplest approach to solving a problem. Although the command line is now about half a century old, it is still the fastest tool for certain use cases such as stream processing tasks. CLI offers several desirable advantages, including simplicity, stability, as well as high modularity in design. Even with Big Data tools like Hadoop widely accessible now, it would be incorrect to assume that the most sophisticated tools are the best at solving every data-driven problem. In his article [1], Adam Drake demonstrates how a single pipeline using shell tools was over 200 times faster at computing win/loss statistics for chess matches, compared to a Hadoop cluster. It is certainly true that distributed processing frameworks using MapReduce have been proven to be instrumental at approaching problems involving Big Data, but before rushing into it, we should always take a moment to question the necessity. One may often find that the data they are working with can be cleaned and analysed with much greater efficiency using well-known shell tools in a command-line interface.

3.2. Relational (SQL) vs non-relational (NoSQL) database systems

- Relational and non-relational systems are both widely used database management models, so the context and specific problem scenario are vital to making the choice between SQL and NoSQL. There are no definitive answers to "which one is better?", since they both have their merits and pitfalls. The choice must be guided by factors such as the type of information needed to be stored, constraints dictated by the client, and the kind of data access required.
- Relational databases have been prevalent for decades, and have the advantages of being mature, proven and widely implemented [2]. SQL is extremely powerful when there are preexisting relationships between different entities in data, hence the data is stored in an organised, yet rigid manner, where much of the focus is on reducing duplication.
- SQL requires a predefined schema and database structure before use, and all entries must follow this same structure. With the ACID transaction model [3], relational databases offer the following guarantees:
  - *Atomicity*: The validity of all data is ensured by the atomic property of every transaction, i.e., partial success is not permitted, and an unsuccessful transaction would revert the database to its previous state.
  - *Consistency*: The structural integrity of the database is maintained at all times by a rigid schema, hence no processed transaction can change this.
  - *Isolation*: While in progress, a transaction cannot interact with another transaction and compromise its integrity.
  - *Durability*: Once a transaction is marked successful, the data will persist in the system even in case of power or network failure.
- On the other hand, No-SQL databases excel in scalability, resilience and availability characteristics. They are much younger than SQL systems and were developed to cater to rapid application changes [4]. Unlike tables in SQL, data in non-relational databases is stored in the form of documents, key-value pairs, wide columns or graphs.
- NoSQL systems are also highly flexible and horizontally scalable, i.e., instead of increasing resources on a single server (vertically scalable relational databases), traffic is handled by distributing load across multiple servers. Moreover, NoSQL queries are often faster than in SQL, since the latter usually has data in normalised tables, requiring multiple complex joins to access the necessary attributes. This can become an expensive operation with large amounts of data. Hence NoSQL optimises access time, and is preferable when the data is in unstructured like documents or JSON.
- Instead of ACID properties, most NoSQL systems follow the BASE model:
  - *Basically Available*: Data availability is the focus, using replication across a cluster.
  - *Soft State*: No responsibility is taken to enforce immediate consistency, since data may change over time.
  - *Eventually Consistent*: Although immediate consistency is not ensured, it is expected that eventually, it will attain a consistent state, however without any guaranteed deadline.
- To summarise, SQL systems (e.g. Oracle, MySQL, PostgreSQL) are ideal when the data is relational, predictable and highly structured; ACID properties are required; and queries are complex. NoSQL databases (e.g. MongoDB, DynamoDB, Cassandra) should be used when ACID guarantees are not necessary; data is non-relational, dynamic and frequently changing; workloads require large scale; or fast writes are a priority over data safety.

3.3. Review on ‘Dynamo: Amazon’s highly available key-value store’

Motivation

During the holiday season of 2004 [5], Amazon.com was facing scalability issues with its existing Oracle database. Out of an experiment to build an in-house database to address this issue, engineers at Amazon (DeCandia et al. [6]) designed Dynamo, with the paper being published in 2007. The paper describes a non-relational, highly scalable, eventually consistent, key-value database with the primary focus on being “always writable”. Such a high level of reliability was required since minor outages have significant financial consequences, especially in applications like Amazon’s shopping cart, where availability is essential even in the event of disk failure, network failure or natural disasters at the data centre sites. Rejecting updates from clients could not be afforded since it would lead to poor customer experience and harm the company’s reputation.

Following the CAP theorem (consistency, availability, partition tolerance), Dynamo sacrifices immediate consistency under certain conditions, and combines well-known techniques such as consistent hashing, object versioning, application assisted conflict resolution, distributed failure detection, and so on, to meet the demanding requirements of Amazon’s services. Apart from the availability aspect, Amazon found that the majority of read/write operations were key-based and too simple to require complicated database structure, a relational schema, or the JOIN functionality in SQL. Hence, a NoSQL design supporting horizontal scaling was preferable, and would, in turn, overcome the vertical scalability limitations of an RDBMS.

Related Work

The authors discuss several existing technologies at the time which were related to Dynamo, but not quite something that would meet Amazon’s targets. Peer-to-peer (P2P) systems like Freenet and Gnutella tackled data storage and distribution, but were primarily used for file sharing, where searching would require flooding through the network. Next-generation P2P structured networks like Pastry and Chord used globally consistent protocols. Other examples include Oceanstore, PAST, Farsite, Ficus, Coda, Bayou, as well as the Google File System.

However, in contrast to these systems, Dynamo was designed to optimise storing small objects (less than 1MB) as key-value stores, which were easier to configure on a per-application basis. The problem of data security also doesn’t arise since the intended use case was for Amazon’s internal, trusted services. Even compared to Google’s Bigtable, Dynamo’s primary target was applications requiring only key-value access, high availability, and accepting user updates even in a variety of failure scenarios.

Proposed Solution

The main principles in Dynamo’s design are the following:

- *Incremental scalability*: the ability to scale out nodes without adverse effects on the system.
- *Symmetry*: every node has the same responsibilities.
- *Decentralisation*: avoid outages caused by centralised control, by using a decentralised P2P approach.
- *Heterogeneity*: work distribution according to the capabilities of individual servers.

Moreover, the paper describes how Dynamo deals with system architecture problems:

- *Interface*: Being a non-relational key-value store, Dynamo only exposes two simple operations: `get(key)` and `put(key, context, object)`.

- *Partitioning*: The incremental scalability requirement of Dynamo needs dynamic partitioning of data over storage hosts. Instead of regular hashing techniques, Dynamo applies the consistent hashing algorithm, where adding a new node to the system only affects the immediate neighbours, and the key assignment operation does not need to be repeated for every node. The main idea behind the algorithm is to treat the output range of a hash function as a ring within which nodes are positioned and given responsibility for specific areas. To address performance issues with basic consistent hashing, Dynamo uses virtual nodes.

- *High write availability*: A series of design choices enable high availability for writes. Data is replicated across N nodes, with each key having a coordinator node responsible for replication across remaining N - 1 neighbours. Since strong consistency is sacrificed for availability, this means that updates are propagated to all replicas asynchronously. Vector clocks are used to deal with inconsistencies in data versioning, where the clock increments for every version of an object. According to the needs of an application, the responsibility of resolving version conflicts is delegated to the client. Dynamo can afford these inconsistencies, especially in applications like the shopping cart, where the removal of an item from cart due to conflicts is a far greater issue than having an extra item which the user can choose to remove later.

- *Temporary failures*: “Sloppy quorum” and “hinted handoff” techniques are used to overcome temporary failures in the system. Sloppy quorum ensures availability and durability during server/network failures, where a strict quorum is not enforced and all read and write operations are performed on the first N healthy nodes from a preference list. Hinted handoff works in tandem with this technique, and prevents operation failure due to temporary node or network failures.

- *Permanent failures*: To recover from permanent failures, Dynamo utilises anti-entropy and Merkle trees (hash tree with leaves as hashes of individual keys). Replica synchronisation is optimised by using these Merkle trees, which reduce the data transfer in the operation by identifying differences using a tree traversal scheme.
- *Membership and failure detection*: A "gossip-based" protocol guarantees eventual consistency and enables each node in the system to learn about the arrival/departure of other nodes. This preserves symmetry and avoids the concept of a centralised registry for storing membership and node-liveliness information.

Implementation

Each storage node in the Dynamo system has three primary software components, implemented in Java:

- *Request co-ordination*: Built on top of an event-driven messaging substrate, with the message processing pipeline split into multiple stages.
- *Membership and failure detection*
- *Local persistence engine*: Allows for different storage engines to be plugged in, e.g. BDB Java Edition, MySQL, etc.

Results

For the two years before the Dynamo paper was published, it was being used internally at Amazon. They observed extremely promising results, with applications receiving successful responses for 99.9995% of requests, without time-outs or any data loss! Dynamo has also met the high expectations in terms of availability, scalability, performance and failure tolerance. Moreover, Dynamo provides a great advantage in its ability to allow parameter tuning (N - number of hosts, R - minimum number of nodes participating in a successful read, W - minimum number of nodes participating in a successful write), which lets the instance be configured as per requirements.

Thoughts

Dynamo is definitely one of the most influential papers on NoSQL systems, having inspired several databases such as DynamoDB, Apache Cassandra, Riak and Project Voldemort. It was interesting to explore the various algorithms applied to achieve the goal of "always writable", which initially seems far-fetched. In terms of limitations, one aspect to consider would be that Dynamo was designed to be used for Amazon's own services, hence data integrity and security concerns are not addressed.

Conclusion

My primary takeaway from this project is the experience I gained in optimising Bash scripts to run quickly on large datasets. Having previously worked with Python for data cleaning and analysis, I realised that bash can also be a powerful tool for such purposes. Finally, as an incoming intern at AWS, I enjoyed learning about Amazon's history with the Dynamo paper.

References

[1] Adam Drake. 2014. Command-line Tools can be 235x Faster than your Hadoop Cluster.
Retrieved from *https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html.*

[2] Microsoft. 2021. Relational vs. NoSQL data.
Retrieved from *https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data.*

[3] Marko Aleksic. 2020. ACID vs. BASE: Comparison of Database Transaction Models.
Retrieved from *https://phoenixnap.com/kb/acid-vs-base.*

[4] Lauren Schaefer. 2021. NoSQL vs SQL Databases.
Retrieved from *https://www.mongodb.com/nosql-explained/nosql-vs-sql.*

[5] Alex DeBrie. 2018. The Dynamo Paper.
Retrieved from *https://www.dynamodbguide.com/the-dynamo-paper/.*

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (SOSP '07). Association for Computing Machinery, New York, NY, USA, 205–220.