COMP 47470

Big Data Programming

Project 1

Conor Murphy

20205251

01/03/2021

Part 1

Q1 The top line of the file is piped into the awk command which when using the -F option will iterate over every value in the file, the print NR will then return the total number or records iterated over, in this case each record is a column, so the returned value is the total number of columns.

```
head -1 nasa.csv | awk -F, '{print NF}'
```

The same result can be achieved using the sed to count the number of commas in the header as shown below.

```
head -1 nasa.csv | sed 's/[^,]//g' | wc -c
```

for the remainder of the scripts, I opted for the awk approach to count the number of columns.

Answer : 222

Q2 Using the wc command with the -l option for lines the number of lines can easily be found. This value will include the title column.

```
cat nasa.csv | wc -l
```

Answer: 409

Q3 This can be achieved Using the cut command with the -d option set to "," as this is a csv file and the -f option set to "2-" which will cut everything from the second column on as the sample_year column is the first column, the output is then redirected to NewNasa.csv.

```
cut -d"," -f2- nasa.csv > newNasa.csv
```

Q4 First the last 408 columns are selected as the top one is the column names, the desired column which is the 6th index is then cut . Using the awk command a variable total increment itself by the current row, once finished adding all of the values it then returns the total divided by the number of items it iterated over so in this case 408 which is the average value of the column.

```
tail -n 408 nasa.csv|cut -d"," -
f6 | awk '{ total += $1 } END { print total/NR }'
```

The same result can be achieved more succinctly using the awk -F option which can skip the cut step and instead be used to reference the 6th column directly in the awk statement.

```
tail -n 408 nasa.csv| awk -
F',' '{ total += $6 } END { print total/NR }'
```

Answer: 29.723

Q5 firstly, the column headers are removed as they will interfere with the sort that we need to do later. the desired column is then cut and these values are sorted with the arguments -r (reverse sort) so the largest will be first and -n (numeric sort) as the values of the columns are numbers and without this the returned value is 990

```
tail -n 408 nasa.csv | cut -d, -f12 | sort -rn | head -n1
```

Answer: 1830

Q6

```
#!/bin/bash

inputFile=nasa.csv
numcolumns=$(head -1 $inputFile | awk -F, '{print NF}')

cols_to_keep=""

for i in $(seq 1 $numcolumns); do
        uniqVals=$(tail -n408 $inputFile | cut -d',' -f$i | uniq | wc -
l)
        if [ $uniqVals -gt 1 ]; then
                cols_to_keep=${cols_to_keep}$i',';
        fi
done

#remove last comma
cols_to_keep=${cols_to_keep: : -1}

#create new dataset
cut -d"," -f $cols_to_keep $inputFile > removed_vals.csv
```

Firstly, the input file is set to a variable as it will be reused thought the script. An empty string is declared outside of the loop as this will hold all of the values of the columns that we intend to keep. A loop is declared looping from 1 to the total length of the columns with an increment of one. The loop will check each column individually and count the number of unique values in the column, now all of the columns with only one value will return one. If the result of the function is greater than one (signifying that it has a least two unique values) then the index of the column which is being tracked by I in the for loop is added to the variable cols_to_keep. Initially I had tried implementing this with an array but that cannot be used with cut and if I iterated thought the array and cut at each position the index positions would be changing so this would not be a great approach. Fortunately, the cut command will take the list with all of the desired columns to keep and can then redirect the output into a new csv file called removed_vals.csv .

Q7

Firstly, the script checks to see if the value is the correct number of arguments have been passed in, if they have not the user will receive an error message. The input file is once again set to a variable and the number of columns to be iterated over is also stored in a variable. A for loop is used with the sed command, on each iteration the column name is checked with the input provided and if they match then the desired row is found and returned using the head and tail commands with pipes. Note the arithmetic operator is used to add one on to the users input as the 10th column will really be the 11th with the header included.

```
#!/bin/bash
if [ $# -ne 2 ]; then
    echo "Please provide two arguments Column name and row number"
else
inputFile='nasa.csv'
numcolumns=$(head -1 $inputFile | awk -F, '{print NF}')

for i in $(seq 1 $numcolumns); do
        col_name=$(head -n1 $inputFile | cut -d"," -f$i)
        if [ "$col_name" = "$1" ]; then
            cut -d"," -f$i $inputFile | head -n $(($2 + 1)) | tail -n 1
            break;
        fi
done
fi
```

Q8 The nasa.csv file is copied to a new file as the word replacements will be done in place so to preserve the nasa.csv file a new one was created for the question. The dict file is iterated over and using the sed command with the -i option (in place) and the global option at the end of the command indicated by the g, this will change every instance of the word and not only the first instance which is the default with sed.

```
#!/bin/bash
cp nasa.csv q8Nasa.csv

while read -r var;
do
    word=$(echo $var | cut -d, -f1)
    replace_with=$(echo $var | cut -d, -f2)
    sed -i s/$word/$replace_with/g q8Nasa.csv
done < dict.txt
```

Part 2 SQL

The two tables were populated suing the following SQL, the

```
create table table1(
    sam_index smallint Not Null AUTO_INCREMENT,
    sample_id varchar(20) not null,
    sample_latitude decimal(8,4) not null,
    sample_longitude decimal(8,4) not null,
    sample_month tinyint not null,
    primary key (sam_index)
);

create table table2(
    sam_index smallint Not Null AUTO_INCREMENT,
```

```
    sample_id varchar(20) not null,
    sample_altitude decimal(8,2) not null,
    sample_seconds tinyint not null,
    primary key (sam_index)
);
```

The sam_index field was set to an auto incrementing smallint which is sufficient for the dataset as there are only 408 rows. The sample id was set to a var char as it takes the form “00/00/0000” latitude, longitude and altitude were set to decimals to preserve their floating point values. sample_months and sample_seconds were set to tinyints as their maximums are 12 and 60 respectively. The primary key was set to the sam_index as the sample _index repeats values and thus is not good primary key.

The script sql_insert script was used to populate the tables and the database name is asgn1.

2.1

1) without the round function the result is 29.7230 but as all seconds in the dataset are given as whole numbers the round is helpful. ANSWER : 30

```
select ROUND(AVG(sample_seconds)) from table2;
```

2) The distinct keyword signifies unique values. Answer : 131

```
select COUNT(DISTINCT(sample_id)) from table1;
```

3) Either all entries can be printed and the first and last retrieved or using the limit keyword set to 1 and the order way switched we can get the values. The first way show prints all values.

```
select sample_id, sample_month from table1 order by sample_month ASC li
mit 1;
```

or with the limit included and the direction of the sort switched. The first one is minimum and the second one the maximum.

```
select sample_id, sample_month from table1 order by sample_month ASC li
mit 1;
```

```
select sample_id, sample_month from table1 order by sample_month DESC l
imit 1;
```

Answer: Min = 0101/3921, MAX = 01/12/3921

4) this is achieved with the > operator. ANSWER: 204

```
Select Count(*) from table2 where sample_seconds > 30;
```

5) the two tables are joined on using the sample_index colum and all items that meet the where clause are then counted and the total returned. Answer : 186

```
Select DISTINCT(COUNT(*)) from table1 t1
inner join table2 t2 on t1.sam_index = t2.sam_index
```

```
where t1.sample_month > 5
and t2.sample_seconds > 30;
```

6) The relevant columns are selected a simple join is done between the tables with table alisas set to t1 and t2. the specified items are returned where altitude equals 5036.82.

Answer :

Sample_latitude	Sample_longitude
67.9232	-160.0902

```
Select t1.sample_latitude, t1.sample_longitude
from table1 t1
Inner join table2 t2 on t1.sam_index =t2.sam_index
where t2.sample_altitude = 5036.82;
```

7) The round is once again used as it does not make much sense for seconds to have decimal places,

Answer: 30

```
select ROUND(AVG(t2.sample_seconds))
from table2 t2
inner join table1 t1 on t2.sam_index = t1.sam_index
where t1.sample_month > 5;
```

8) this is the same as Q7 with the addition of another condition in the where clause.

Answer: 40

```
select ROUND(AVG(t2.sample_seconds))
from table2 t2
inner join table1 t1 on t2.sam_index = t1.sam_index
where t1.sample_month > 5
and t2.sample_seconds > 20;
```

2.2 Mongo

The mongo_insert.sh script was used to populate the MongoDB collection with the data from the nasa.csv file. The database name that was used is called asgn1 and the collection is called nasa.

1) The count function is used while retrieving only the items with over 30 seconds using the $gt comparison operator.

Answer: 204

```
db.nasa.count({sample_seconds:{$gt:30}});
```

2) using the find function and setting the sample latitude and longitude to the desired value the desired result can be found.

Answer: "sample_id" : "01/08/3915"

```
db.nasa.find({sample_latitude:68.5582, sample_longitude:-155.7573},{sample_id:1})
```

3) the decimal values can be ignored simply by using a combination of the greater than and less than operators, all values that have a 67 in them are greater than or equal to 67 and less than 68. Initially I tried achieving the desired result using regex but fond this approach to be far simpler.

Answer: 36

```
db.nasa.count({sample_latitude: {$gte:67, $lt:68}});
```

4) the MongoDB aggregration framework was used to get the average sample seconds, the items are grouped together and a sample Id value of 1 is set as it cannot be left blank, then using the $avg operator and specifying the sample second field we can get the desired figure

Answer: { "_id" : "1", "avg_seconds" : 29.723039215686274 }

```
db.nasa.aggregate( [{ $group:{_id: "1",avg_seconds:{$avg: "$sample_seconds"}}} ] );
```

5) Now the group value is instead set to the month field which will allow for the grouping of the sample seconds per month, the month is also displayed in the output.

Answer:

```
{ "_id" : 10, "avg_seconds" : 28.885714285714286 }
{ "_id" : 11, "avg_seconds" : 30.652173913043477 }
{ "_id" : 9, "avg_seconds" : 30.26027397260274 }
{ "_id" : 8, "avg_seconds" : 32.385714285714286 }
{ "_id" : 7, "avg_seconds" : 27.617283950617285 }
{ "_id" : 6, "avg_seconds" : 30.7375 }
{ "_id" : 5, "avg_seconds" : 27.74074074074074 }
{ "_id" : 4, "avg_seconds" : 25.789473684210527 }
```

```
db.nasa.aggregate([{ $group:{_id: "$sample_month",avg_seconds:{$avg: "$sample_seconds"}}} ]);
```

6) This was a very similar approach to question 5 but het instead of returning the average the maximum value is instead returned, the remainder of the code remains unchanged.

Answer:

```
{ "_id" : 10, "max_seconds" : 59 }
{ "_id" : 11, "max_seconds" : 59 }
{ "_id" : 9, "max_seconds" : 59 }
{ "_id" : 8, "max_seconds" : 59 }
{ "_id" : 7, "max_seconds" : 59 }
{ "_id" : 6, "max_seconds" : 58 }
{ "_id" : 5, "max_seconds" : 59 }
{ "_id" : 4, "max_seconds" : 56 }
```

```
db.nasa.aggregate([{ $group:{_id: "$sample_month",max_seconds:{$max: "$sample_seconds"}}} ]);
```

3.1 Reflection

Relation database models and NoSQL database models differ greatly. Both have their own use cases, and one is not inherently better than the other but depending on the situation one may be preferable to the other. Before comparing the two it is useful to understand each model.

Relation Database Model

Relational database models operate on using a predefined schema, interactions with a relational model are done using a Structured Query Language called SQL. Relational databases have been around for far longer than the newer NoSQL models. Relational database models operate on the A.C.I.D principle which stands for Atomic, Consistent, Isolated and Durable. Atomic means that a transaction (which is a database operation like inserting data) will complete fully or fail completely i.e., it will succeed in full or not at all. So, if you were inserting a new record and one field was incorrect according to the schema , then the other fields would also not be written. Consistent means that the database will only be brought from one state of consistency to another so after a transaction all integrity constraints will be maintained, this ensures that the database is always correct. Isolation means that transactions are independent of each other and that if transactions are executed concurrently the result would still be the same as if they were executed sequentially. Durability states that the data will consist even if the database fails as the data will be stored in non-volatile memory. The most popular relational databases are MySQL, Microsoft SQL server, PostgreSQL, and oracle database, many of these services are also available and ready to use from cloud providers like AWS and Azure.

NoSQL Database Model

NoSQL stands for “Not only SQL”. NoSQL database models do also not focus on a strict schema and instead are suited for unstructured data. NoSQL databases store data in a variety of ways the most popular ones are Key value pairs, documents, wide column stores and Graph databases. NoSQL database models generally follow the B.A.S.E principle which stands for basically available, Soft state and Eventually consistent. Basically Available meant that the database will appear to work most of the time and follows the CAP theory. Soft state essentially means that the user or developer is responsible for consistency i.e., a value field could be 4 or “four”. Eventually consistent means that the database will at some point be consistent after all write operations but during operations it may not be. BASE was made to purely contrast the ACID paradigm.

Comparison

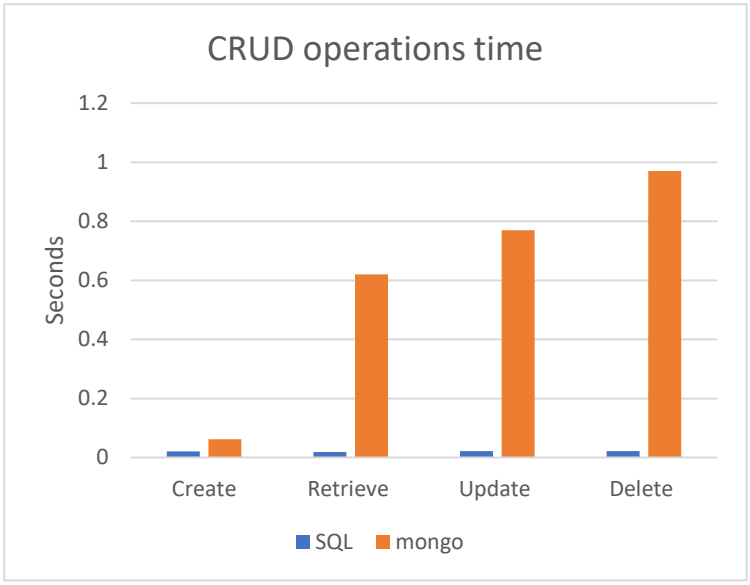
Now a foundational understanding of the two models has been established the two can be compared. As relational databases follows a schema so if a new field is wanted to be captured then the entire table will need to be updated this is not the case for a NoSQL database as the data is unstructured so the new field could easily be added, in this instance possibly a NoSQL database would be preferable over a relational one, however if the structure of the application has a relational structure and has a lot of multi row transactions then a relational approach may be preferable.

Due to the BASE nature of NoSQL, developers have the added job or ensuring that data is consistent. This worry is not present in relational databases so for example if data validity was a priority then potentially a relational database approach is preferable. If Complex queries with multiple joins are necessary, then relations database models are also a better choice.

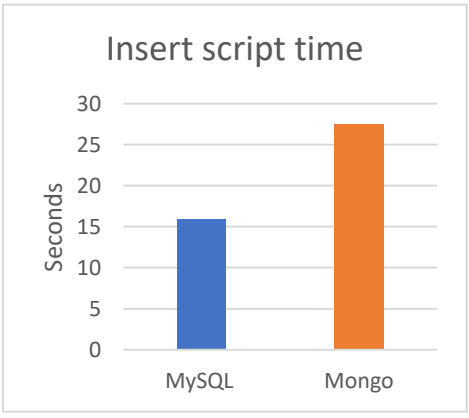
One major difference between the two approaches is how they scale; Relational models scale vertically, and NoSQL models scale horizontally. Scaling vertically mean that the machine itself need to become more powerful for example moving from an intel i3 process to an i5 process or upgrading RAM. Scaling horizontally on the other had means that the instead or upgrading process you simply add another machine and split the work between them, this Is generally easier as adding a few new database servers to keep up with demand on a NoSQL model is simpler than upgrading an entire Relational database server. For this reason, where there is a large amount of data or rapidly increasing dataset then a NoSQL approach may be a better, for example Realtime databases like google firebase are NoSQL systems.

To compare the running times of both the database models I used the time command in bash which returns the total time the operation took to run. I did two sets of experiments the first was the to test the running time of each CRUD (Create, Read, Update, Delete) operation, the results are shown in the graph below.

	SQL	mongo
Create	0.021	0.062
Retrieve	0.018	0.62
Update	0.022	0.77
Delete	0.022	0.97



Overall, the results clearly show that the MySQL database being faster for the queries, however it is worth noting that the database does only have 408 entries and at this size querying would be expected to be faster than if there were hundreds of thousands. Another notable point is that as this is only one command and the results can vary between different tests, so there is a margin of error. The second experiment was running the time command with the full insert script which tracked how long it took to insert all 408 entries into both database models, the results are shown to the right. Here it is interesting to see how MongoDB, is just under double the time of the MySQL script.



In Summary the two systems have their own unique strengths and weaknesses and as mentioned one is not better than the other instead it depends on the use case and the context in which the database model will be used.

Sources

Ganesh Chandra, D., 2015. BASE analysis of NoSQL database. *Future Generation Computer Systems*, 52, pp.13-21.

https://www.geeksforgeeks.org/acid-properties-in-dbms/

https://www.xplenty.com/blog/the-sql-vs-nosql-difference/

https://www.guru99.com/sql-vs-nosql.html

3.2 Introduction

The article I have chosen is Eventual Consistency by Werner Vogels. Initially when reading the authors name, I knew it seems familiar and within the first sentence it clicked that he is the CTO of Amazon. This connection initially made me think to arguable his most famous quote “Everything fails, all the time”, which is coincidentally one of the main themes of the article as he states, “when a system processing trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system”. The paper mainly focuses on consistency in the scope of distributed systems as suggested by the title, it delves into the historical approach to consistency up to the different models of consistency being implemented today. The research consistently emphasised the impact of each model in large distributed systems like AWS. Throughout the paper he discusses the CAP theorem and in particular the trade off between consistency and availability in reliable distributed systems.

Related Solutions

Vogels discusses the ideal consistency model as one where if an update is made all observers would see the update, he mentions historically in a distributed system the goal is to make the make the distributed system appear as if it is a singular system instead of being made up of multiple machines. Interestingly early distributed systems favoured the approach that it was better to fail as a complete system than to break distributed transparency (the internal details of distribution are hidden from the user). As larger distributed systems began to be introduces in the 90’s they realised that availability was now becoming more important and that to prioritise it there would need to be a trade-off. This leads to the idea of CAP theorem whereby you can only have two of these three qualities in a distributed system, either Consistency, Availability, or partition tolerance. He goes on to state that in large, distributed network partitions are a given so that only leave either Availability and Consistency and thus is the source of the trade off between the two.

Vogels puts an emphasis on the developer being aware of the trade off, for example if the system prioritises consistency then the developer will need to have a plan if the database in not available and conversely if the priority is on availability the developer needs to know what to do when, for example a recent write is not immediately available. Vogels compares the form of consistency found in the acid model for relational databases that takes the database to from one state of consistency to another and uses the example when transferring money from one bank account to another the total value in both accounts should not change. This form of consistency differs from the form of consistency that Vogels is discussing in the article as he is discussing it in the context of a distributed database having all of the exact same info across all nodes when a new transaction takes place.

Different Approaches

Vogels discusses two different ways of looking at consistency one as from the client or developer point of view and the other from the server point of view. Looking at it from the client point of view is seeing how they observe updates and looking at it from the server side is looking at how updates flow through the system and what guarantees the system can give these updates. Vogles then goes on to present the different forms of consistency a short list of a few of the main ones presented are.

- ***Strong Consistency***: After an update, all process will return the updated value.
- ***Weak consistency***: There is no guarantee that all process will have immediate access to an updated value. Three is a period between when an update takes place and when the system can guarantee that all process can see the update is known as the inconsistency window.

- ***Eventual Consistency***: this is a form of weak consistency and has been seen before in the BASE model and discussed in part 3.1, this Is essential when the system guarantees that in the absence of any new entries all process will eventually have access to updated values.
- Eventual consistency has a number of variations which include Casual consistency, read your writes consistency, session consistency, Monotonic consistency, and Monotonic Write consistency.

This problem of eventual consistency might seem like it is only present in large distributed systems, but Vogel's points out that it is also present its replication techniques for relational database model. In a synchronous model's replication, the replica is written to backup as part of the transaction, so the data is strongly consistent. In an asynchronous model the replica is delayed after the transaction and consistency will eventually be achieved when the logs are shipped meaning that is the database fails between when the transaction occurred, and the logs had been shipped then the last backup would not be consistent as some data would have been lost. Vogles then goes on to discuss the server side and how it deals with the different consistency types listed above and runs through different scenarios of with varying numbers of nodes and replicas needed for each type. He uses the terminology N = number of nodes that store replicas of the data, W = The number of replicas that need to acknowledge the receipt of update before the update completes and R= the number of replicas that are contacted when a data object is accesses through a read operation. Some of the main consistency states are. Strong Consistency: W+R > N, Weak Consistency: R+W=N. He does say that there is a problem with these as they abide by a basic quorum principle meaning that, for example if N = 4 and W = 4 with only 3 nodes available then the system will fail to write new entries even though three nodes are available, it is very much so an all or nothing approach.

Vogel's concludes with introducing Amazon Dynamo which is amazons key-value pair storage system and is still extremely popular and in use to this day as part of AWS, he states that dynamo allows customers to choose the level of trade-off between consistency, durability, and availability at different price ranges. The summary states that it should be tolerated to have data inconsistency in large scale distributed systems as it speeds up read and write process during highly concurrent conditions and that if not in partition cases with a majority model the system would state that it is unavailable even when the nodes are functioning. He finished by stating that the developer should decide whether or not inconsistencies are acceptable and to what extent, for example if user perceived consistency is adequate which is a form of eventual consistency where the inconsistency window is small enough to fit in the time between a webpage being submitted and the next one being loaded; here eventual consistency is being used but to the user the systems data seem consistent.

Reflection

Overall, I found the article to be quite intriguing, I had only really been introduced to the idea of consistency in distributed systems in this module, so it was interesting to dive a bit deeper into the different types of consistency and how they are used in distributed systems. Overall, I thoroughly enjoyed the piece and seeing that it came from the CTO of amazon with his wealth of experience was also pretty cool. The paper was written in 2008 and to see the ideas that he is talking about to still be so prevalent today especially in relation to Amazon Dynamo which is modern day Dynamo DB is pretty cool, as it is currently extremely popular in the development of serverless applications, which is an interest area of mine. To be able to understand a bit more about the distributed systems behind services like that was extremely insightful for me. Overall, it was an informative read with clear understandable language that I most certainly benefited from.