SWD406 AE1 report

James Dunsmore

Contents

*Introduction*--------------------------------------------------------------------------3

*Practical work*--------------------------------------------------------------------4-5

*Evaluation report*----------------------------------------------------------------6-8

*Appendix*-------------------------------------------------------------------------9-12

*References*--------------------------------------------------------------------------13

Introduction

Relational databases are an integral part of modern IT systems and play a crucial role in how data is managed, they are what many every day services rely on, from bank transfers to “A relational database is a type of database that stores and provides access to data points that are related to one another” (Anon(Oracle). 2020) They are based on the relational model; a standard method of representing data. The defining characteristic of a relational database is that each record in a table has a unique key, the columns of a table define attributes of the data and the records collection of values for said attributes provide a simple method to establish connections between data points (Anon(Oracle). 2020).

The main benefits of using a relational database is its versatility and consistency. No matter the application, the rules by which the database follows are the same regardless. Relational databases also reduce redundancy, with information appearing in one entry of a table, as tables are linked to rather than the data being stored in multiple tables (Anon(IBM). 2020).

The main drawbacks of a relational model are centred in its biggest strengths; transactions and joins. Whilst these features provide integrity, reduce redundancy and improve simplicity; they add latency to the overall use of the system. With each node that data must be transferred across, latency is added and compounds; resulting in an ever-growing lag for the end user.

SQL is at its core, a language based on relational algebra. What that means for it’s structuring is its rules are consistent and based on mathematics, ensuring that the information it produces is simple and consistent. To understand why SQL’s explicit rules are so useful, first the difference between data and information must be explained. Data is the raw, disorganised random recorded attributes, that lacks any context with which it should be interpreted. Information, however, is that data organised and given context to be interpreted with. The algebraic root of SQL gives a simple mathematical set of rules by which data is interpreted as information.

2.Practical Development Work

1. Retrieving data using SQL
2. *see appendix 1 for output*
3. **SELECT**
4. Shopper\_first\_name **AS** [**first** **name**]
5. ,Shopper\_surname **AS** [surname]
6. ,Shopper\_email\_address **AS** [email address]
7. ,(strftime('%Y', 'now') - strftime('%Y', Date\_of\_birth) ) - (strftime('%m-%d', 'now') < strftime('%m-%d', Date\_of\_Birth) ) **AS** [age]
8. ,date\_joined **AS** [**date** joined]
9. **FROM** Shoppers
10. **WHERE** Date\_joined >= '2020-01-01' OR Date\_of\_birth > '1990-01-01'
11. **ORDER** **BY** Date\_of\_birth, Shopper\_surname
12. ;
    1. *See appendix 2 for output*
13. **SELECT**
14. s.shopper\_first\_name **AS** [**First** **Name**]
15. ,s.shopper\_surname **AS** [Surname]
16. ,so.Order\_id **AS** [**Order** ID]
17. ,so.Order\_date **AS** [**Order** **Date**]
18. ,p.product\_description **AS** [Product Description]
19. ,sell.Seller\_name **AS** [Seller **Name**]
20. ,op.Quantity **AS** [Quantity]
21. ,op.Ordered\_product\_status **AS** [**Order** Status]
22. ,PRINTF("£%.2f",ps.Price) **AS** [Price]
23. **FROM** shopper\_orders so
24. **INNER** JOIN
25. shoppers s **ON** s.shopper\_id = so.shopper\_id
26. **INNER** JOIN
27. ordered\_products op **ON** op.order\_id = so.order\_id
28. **INNER** JOIN
29. products p **ON** p.product\_id = op.product\_id
30. **INNER** JOIN
31. product\_sellers ps **ON** ps.product\_id = op.product\_id
32. **INNER** JOIN
33. sellers sell **ON** sell.seller\_id = ps.seller\_id
34. **WHERE** s.shopper\_id = '10000'
35. **ORDER** **BY** so.Order\_date **DESC**
36. ;

c)

1. **SELECT**
2. s.Seller\_account\_ref **AS** [seller Account Ref]
3. ,s.Seller\_name  **AS** [Seller **Name**]
4. ,p.product\_code **AS** [Product Code]
5. ,p.product\_description **AS** [Product Description]
6. ,PRINTF(“£%.2f”,(op.quantity\*op.price)) **AS** [total Sales]
7. ,op.quantity
8. **FROM** sellers s
9. **INNER** JOIN
10. ordered\_products op **ON** op.seller\_id = s.seller\_id
11. **INNER** JOIN
12. products p **ON** p.product\_id = op.product\_id
13. **INNER** JOIN
14. shopper\_orders so **ON** so.order\_id = op.order\_id
15. **WHERE** so.order\_date > ‘2019-05-01’ OR op.quantity = NULL
16. ;

d)

1. **SELECT**
2. category\_description **AS** [category description]
3. ,product\_description **AS** [product description]
4. ,product\_code **AS** [product code]
5. ,IFNULL(AVG(op.quantity),'0' ) **AS** [average quantity sold]
6. **FROM** Products p
7. **INNER** JOIN
8. categories c **ON** c.category\_id = p.category\_id
9. **INNER** JOIN
10. ordered\_products op **ON** op.product\_id = p.product\_id
11. **GROUP** **BY** category\_description
12. **HAVING** AVG(op.quantity) > op.quantity
13. ;

2. database design, implementation and integrity

a) design

1. **CREATE** **TABLE** seller\_reviews (
2. seller\_id     **INTEGER** **REFERENCES** sellers (seller\_id),
3. seller\_rating TEXT    NOT NULL,
4. seller\_review TEXT    NOT NULL,
5. shopper\_id    **INTEGER** **REFERENCES** shoppers (shopper\_id),
6. review\_number **INTEGER** **PRIMARY** **KEY** AUTOINCREMENT
7. );
8. **CREATE** **TABLE** product\_reviews (
9. product\_id     **INTEGER** **REFERENCES** products (product\_id)
10. NOT NULL,
11. product\_rating TEXT    NOT NULL,
12. product\_review TEXT    NOT NULL,
13. shopper\_id     **INTEGER** **REFERENCES** shoppers (shopper\_id)
14. NOT NULL,
15. review\_number  **INTEGER** **PRIMARY** **KEY** AUTOINCREMENT
16. NOT NULL
17. );
18. **CREATE** **TABLE** product\_QnA (
19. product\_id      **INTEGER** **REFERENCES** products (product\_id)
20. NOT NULL,
21. question\_number **INTEGER** **PRIMARY** **KEY** AUTOINCREMENT
22. **UNIQUE**
23. NOT NULL,
24. question        TEXT    NOT NULL,
25. answer          TEXT    NOT NULL
26. );

*See appendix 5 for ERD*

I concluded that I would need 3 tables to complete the task: a table for seller reviews, a table for product reviews, and a table for the product questions.   
the reason I came to this conclusion, is that the two kinds of reviews both need to be anonymous, and unrelated; in order for them to be unrelated, they needed to be separated.   
the third table, came from the need for anonymity, as specified in the brief.  
the first table, seller\_reviews, uses seller\_id and shopper\_id from the sellers and shoppers tables respectively as foreign keys. This is to maintain integrity and provide a relationship to the rest of the database. The primary key is the review\_number. I originally intended to use seller\_id as the primary key, but realised that could cause an issue with multiple reviews for the same seller, as they would have the same value. The attributes are mostly not null, as they have to be populated for the subsequent data to be worthwhile.  
I could have used a 4th table for answers and reserved the QnA table entirely for questions; allowing a question to be answered multiple times. However, this would add a somewhat redundant table for an unnecessary feature and would only lengthen the design process.

To populate the tables, I used ‘INSERT INTO’ commands, like these below:

1. **INSERT** **INTO** product\_reviews
2. **VALUES**(3005955,'\*\*\*','could be worse',10000,null);
4. **INSERT** **INTO** product\_QnA
5. **VALUES**(3005955,null,'does the packaging contain carcinogens?','i am not a doctor, i do not know');
7. **INSERT** **INTO** seller\_reviews
8. **VALUES**(200010,'\*','thief, product never arrived',10017,null);

First the table is declared in the INSERT INTO line, then the values in order of the columns of the table. Those left as NULL, were left as such for the auto increment feature, declared in the tables creation. This means that wit each row added, it will automatically assign each row an id, which can be confidently used as a primary key. I had to enter the product ID manually each time, by consulting the tables in SQLiteStudio, and picking a random one each time. I did this for each ID for each tables’ foreign key.

I tested the tables function with some simple queries:

1. **SELECT**
2. product\_id
3. ,question\_number
4. ,question
5. ,answer
6. **FROM** product\_QnA
7. **ORDER** **BY** question\_number
8. ;
10. **SELECT**
11. product\_id
12. ,product\_rating
13. ,product\_review
14. ,shopper\_id
15. ,review\_number
16. **FROM** product\_reviews
17. **ORDER** **BY** review\_number
18. ;
20. **SELECT**
21. seller\_id
22. ,seller\_rating
23. ,seller\_review
24. ,shopper\_id
25. ,review\_number
26. **FROM** seller\_reviews
27. **ORDER** **BY** review\_number
28. ;

*See appendix 6-8 for the output results.*

These showed that the tables functioned as intended, with the autoincremented keys displaying properly, had populated as intended; and with no null values.

Below is the SQL I used to create the views to show my tables related to those that they relate to in the ERD.  
  
there are:

* products and reviews: which links to both shoppers and products
* products and QnA: which links to products, as it is anonymous, there is no other table to link to.
* Sellers and reviews: this links to sellers and shoppers; it is effectively identical to products and reviews, as it serves much the same function, except for sellers or products respectively.

1. **CREATE** **VIEW** [products and reviews] **AS**
2. **SELECT**
3. product\_id **AS** [product id]
4. ,product\_rating **AS** [rating]
5. ,product\_review **AS** [review]
6. ,shopper\_id **AS** [shopper id]
7. ,shopper\_first\_name **AS** [reviewed **by**:]
8. **FROM** products p
9. **INNER** JOIN
10. product\_reviews pr **ON** pr.product\_id = p.product\_id
11. **INNER** JOIN
12. shoppers s **ON** s.shopper\_id = pr.shopper\_id
13. **ORDER** **BY** shopper\_first\_name
14. ;
16. **CREATE** **VIEW** [products and QnA] **AS**
17. **SELECT**
18. product\_description AS [description]
19. ,product\_id AS [product id]
20. ,question
21. ,answer
22. ,question\_number AS [question number]
23. **FROM** products p
24. **INNER** JOIN
25. product\_QnA pq **ON** pq.product\_id = p.product\_id
26. **ORDER** **BY** question\_number
27. ;
29. **CREATE** **VIEW** [sellers and reviews] **AS**
30. **SELECT**
31. seller\_id **AS** [seller id]
32. ,seller\_name
33. ,seller\_rating **AS** [rating]
34. ,seller\_review **AS** [review]
35. ,shopper\_id **AS** [shopper id]
36. ,shopper\_first\_name **AS** [reviewed **by**:]
37. **FROM** sellers s
38. **INNER** JOIN
39. seller\_reviews sr **ON** sr.seller\_id = s.seller\_id
40. **INNER** JOIN
41. shoppers sh **ON** sh.shopper\_id = sr.shopper\_id
42. **ORDER** **BY** seller\_name
43. ;

*See appendix 9 – 11 for the output results*

3. evaluation report

Introduction

Practical work

a)

* firstly, I selected the necessary data and named the respective columns appropriately, using the SELECT function, then subsequently the AS function for the naming. The columns selected were those from the brief, and those I thought relevant to the interpretation of the data.
* The date of birth was ‘converted’ to age by subtracting the year from the date of birth, from the current year; this is then displayed as ‘age’, again using the AS function.
* Then the table it should be drawn from is declared using the FROM function.
* Using the WHERE function, the results are filtered to only include those who joined after 2019 or were 29 and older.
* Finally, the results are ordered first by age, then by surname alphabetically.

Question ‘a’ was completed to specification, aside from translating the date format to DD-MM-YY, as I could not execute the method I used to determine age whilst keeping with the specified format. If I were to reattempt this question, I would alter the date format before performing calculations on it, preventing myself from being unable to find a workaround all together.

b)

* as with question a) the SELECT function is used, alongside PRINTF(), to print the price with the necessary ‘£’. The columns are organised in a logical order where possible(e.g. first name first, second name second)
* FROM is used to determine where the data should come from, I took Shop orders first, as it has the most connections, and the most primary keys; making it easier to connect to the other necessary tables.
* Using inner joins, I consolidated the necessary columns to the specification, joining shopper orders to shoppers, ordered products, then product sellers to ordered products, then sellers to product sellers.
* Then using ‘WHERE’, I filtered only by shopper ids equalling 10000, as to specification.
* Finally, I ordered the results by order date, in descending order.

I had the same issue with date as I did with a, with b. Alongside this, I performed the joins wrong. I should have instead joined sellers and products to ordered products, following the provided ERD. My solution was clumsy and involved more tables than necessary and did not follow the best relations shown in the ERD. This could be resolved by joining sellers to product\_sellers, and then product\_sellers to ordered\_products and Products respectively. I also failed to consider that product\_sellers has a composite key consisting of two columns, meaning I only joined to one column, failing to use AND to include both parts of the table’s key.

c)

* Much like b, I selected the columns I needed using SELECT, adding the ‘s.’ or ‘p.’ to denote where they are from, to later be used in the declarations of the inner joins.
* I used PRINTF() alongside basic arithmetic to produce the total sales column, multiplying total quantity by the individual item price.
* Then, much like b, I declared which table I would join to initially, and selected sellers.
* I joined the columns together, using the prefixes I had stated in the SELECT argument.
* The WHERE function dictates that only orders newer than 2019-05-01 or where quantity returned NULL should be shown.

The problems with ‘c’ all stem from my entire lack of any use of outer joins. Because of this mistake, the information created by my query had no null values and failed to report any products that are offered but unsold. I also, despite not displaying any, did not account for the NULL values, and if they could occur, would display as ‘NULL’ as opposed to ‘0’ as specified.

d)

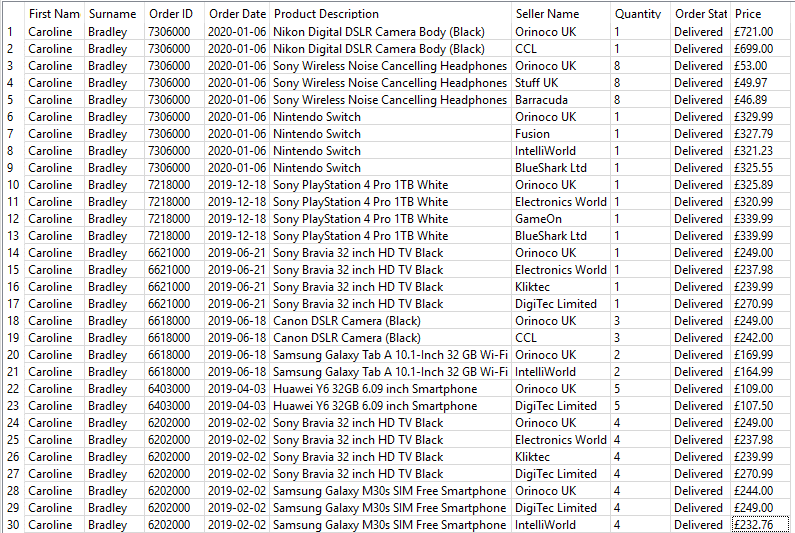
* As with the other tasks, the columns required are selected.   
  the specification required the results to show any null results as ‘0’, and to comply to this, I used IFNULL() and AVG(). IFNULL is a Boolean argument, the first entry being if it is not a null value, the latter being if it is. AVG() creates an average of whatever value it is given; in this case the quantity sold.
* FROM is used to declare what table from which the data is taken, I chose to use products, as it is relevant and closely related to the other relevant tables.
* The inner joins join categories and ordered products to products.
* The results are ordered by category description as per the specification
* The HAVING argument filters the results by all rows that have a total quantity sold lower than the average quantity sold.

The areas where D fail to meet specification are a continuation of the issues found in C. The lack of outer joins, and the subsequent misrepresentation of data means that the information provided is incomplete. Whilst it does provide a short list of products that have a quantity sold lower than the average for that given category, it still fails to account for NULL values. An improvement on C can be found in that, even though it would not be possible for them to occur; the query would account for the NULL values, displaying them as a ‘0’ as specified in the brief.

Appendix

*Appendix 1*

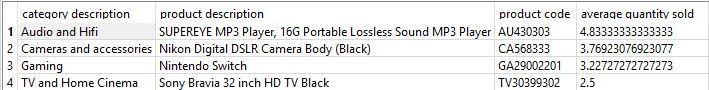
*Appendix 2*



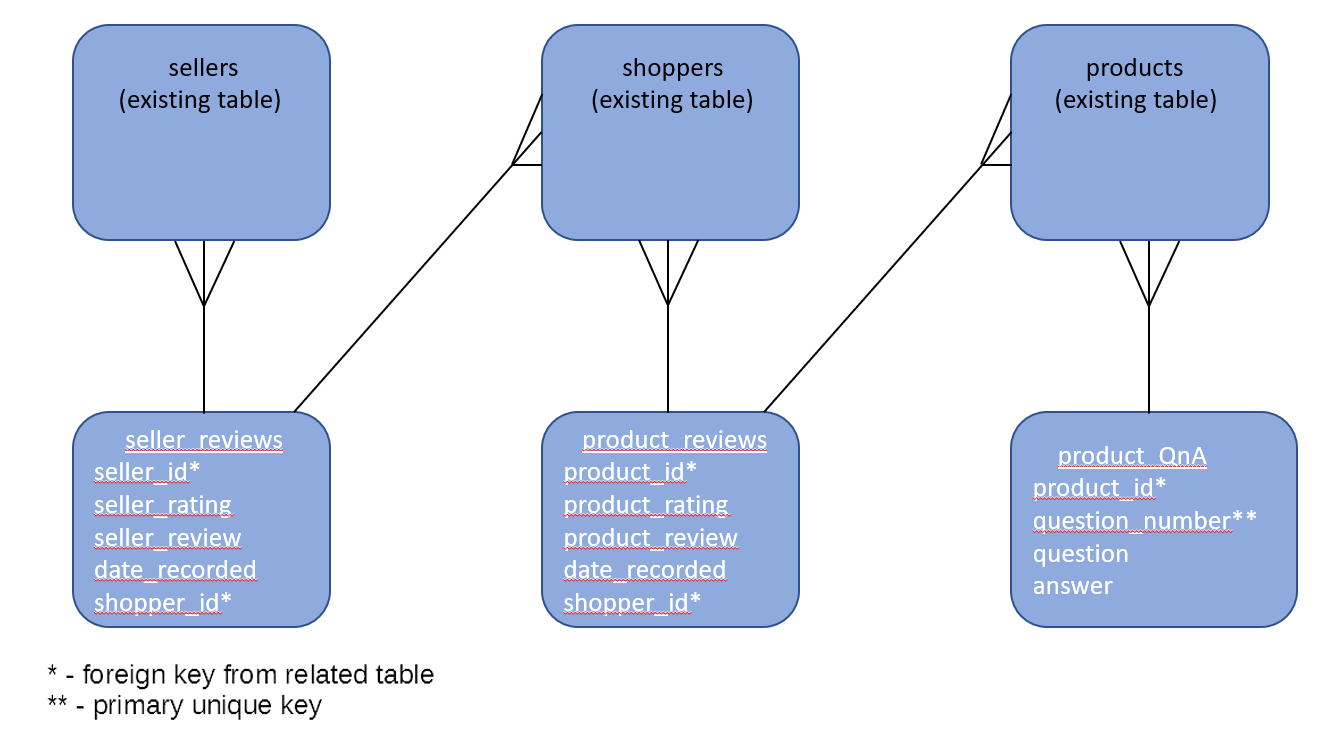
*Appendix 3*



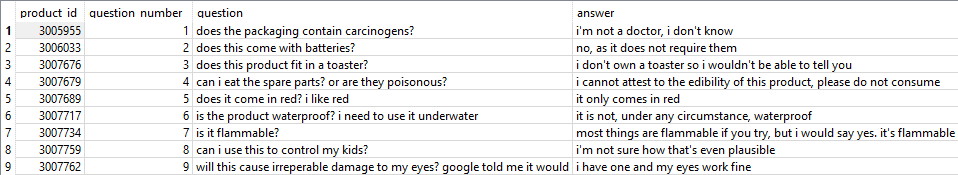
Appendix 4



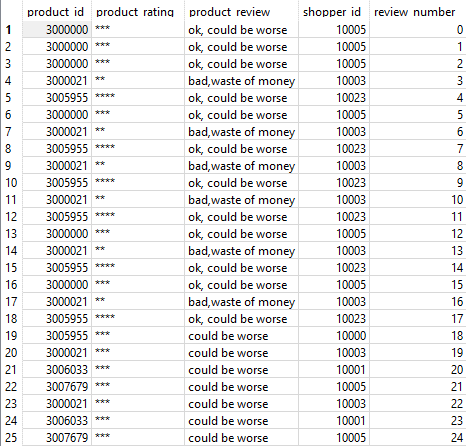
Appendix 5



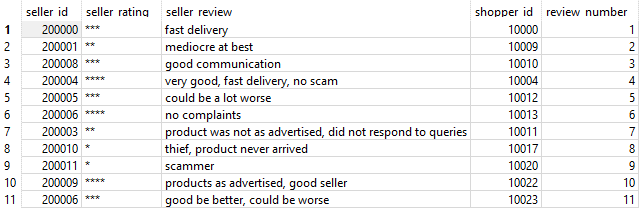
Appendix 6



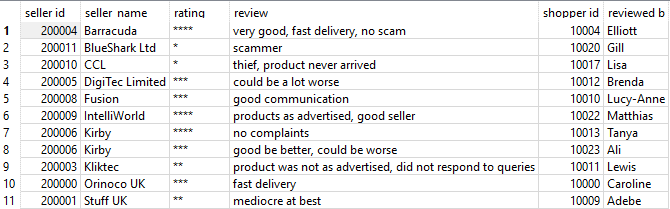
Appendix 7



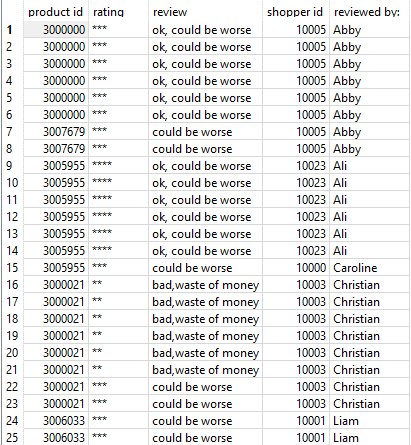
Appendix 8



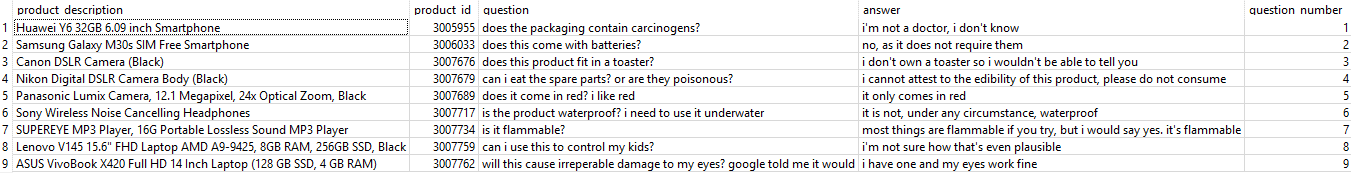
Appendix 9



Appendix 10



Appendix 11



References

References

* ANON., 2020. *What is a relational database?* [viewed 13 May 2020]. Available from: <https://www.oracle.com/uk/database/what-is-a-relational-database/>
* ANON., 2020. *Relational-databases* [viewed 13 May 2020]. Available from: <https://www.ibm.com/cloud/learn/relational-databases>
* ANON., 2020. *Relational Database - an overview | ScienceDirect Topics* [viewed 13 May 2020]. Available from: <https://www.sciencedirect.com/topics/computer-science/relational-database>
* ANON., 2015. *Relational Databases Are Not Designed To Handle Change* [viewed 13 May 2020]. Available from: <https://www.marklogic.com/blog/relational-databases-change/>
* REEVE, A., 2012. Big Data and NoSQL: The Problem with Relational Databases. In: *InFocus Blog | Dell Technologies Services*. 7 September 2012 [viewed 13 May 2020]. Available from: <https://infocus.delltechnologies.com/april_reeve/big-data-and-nosql-the-problem-with-relational-databases/>