After Amazon...

13434418
Luke Killoran
Tony Veale
SQL & Relational Databases

Contents

List of Important Figures

Section 1: Introduction

The intended purpose of this report is to outline a way to support both the database and the main mechanisms of a major online bookseller's catalogue and recommendation system. The domain of this project applies to any business premised on recommendations. However, it is quite limited in that it does not heavily analyze the product content, like many recommendation systems do today (Netflix being one).

The goal was to expound on a database plan which is scalable, simple, and most importantly (asymptotically) flawless. Of course, any company with a faulty database runs the risk of poor operations; it cannot act effectively on the data it has. Today more than ever, businesses depend on the fidelity of their data. And this could not be truer for a business whose success hinges on the quality of its recommendations. Therefore, it is an absolutely essential that the database be designed well.

And this involves simplicity and practicality. A simple design will make the finding of bugs much easier. Though simplicity will not be easy to maintain against large and relentless waves of data, such is a problem that a major online bookseller –and a database administrator (maybe)– welcomes. A preponderance of data is a necessary starting point in businesses like these. It also means that recommendations will naturally be better. But in order to obtain these benefits, the database must first be able to cope with such frequent transactions, storing their information in normalized tables (to avoid anomalies), with checks (or triggers) in place to spot flaws in the processes or the data. It should too come with suite of views on to the database. These facilitate the production of more intuitive and useful (virtual) tables that undo the normalization which separated the data out so much. As well as that, these views furnish ways to restrict the database user's access to only that and exactly that which they should see.

But the database is not there solely to prevent problems. It is also crucial in providing mechanisms which underpin the site. These include a procedure to output a list of bestsellers in place of recommendations for the customers which the business has no data on. This will soften the blow of the cold start problem. Another procedure should facilitate the listing of titles, authors, themes, and qualities by rating or sales with clickable links to filter by category. This permits customers to browse the marketplace.

The buck does not stop there. The database will also be very influential in the production of recommendations. It should also leverage SQL's superior query speed and database operations (e.g., Selection, Join) to go beyond traditional recommendation applications like "Recommend to Tony ten movies", but instead introduce flexible and customizable recommendation capabilities like filtering by a certain category ("Recommend ten computer science books to Tony") or providing rare combinations ("Recommend to Tony ten books that are both computer-science-themed and exciting"). Once the recommendation functionality lives inside the database kernel, the recommendation application takes advantage of the DBMS-inherent features (e.g., query optimization, materialized views, indexing) provided by the storage manager and query execution engine[1]. This will ensure that the recommendation application does not perform unnecessary work, incurring high latency, especially when only a subset of

the recommendation answer is required. It will too eschew the tremendous overhead of extracting the data from the database, loading it to a specialized recommendation engine, and then loading the produced recommendation back to the database.

The database should support both content-based filtering and collaborative filtering. The content-based filtering works on the idea that if a customer has bought many books in some category they will continue to do so. Thus, the database should have a mechanism which queries the customer's collection, finds their favorite book category, and recommends them the bestselling books in that category which they have not already purchased. Another procedure should label each of the customers book by their attributes: author(s), themes, and qualities. Then it should aggregate these to find the top attributes. After which it should return the books with the most matches, and therefore the most similar books in content to those the customer enjoys.

Collaborative filtering –and particularly item-to-item filtering (because we will have many more users than items, so it's computationally simpler to find similar items than it is to find similar users)– is important as well. It ascribes similarity scores to books based on purchasing and rating patterns. More specifically, it imputes a high similarity score to two books if customers tend to either own both or neither (but not only one of them) and if the customers that (dis)like one tend to (dis)like the other.

This method is appropriate for our purposes because a major online bookseller does indeed have the substantial data pool required to support such a system. And it is no surprise that massive companies such Amazon, YouTube, and Netflix use this system[2]; the benefits of this approach are multifarious. For instance, the sparsity of the user-item preference matrix significantly reduces the computational complexity. The expensive batch update of the table housing the similarity scores can be done semi-regularly and offline. Then the online component simply involves querying that table for certain set of books, a process which could easily be implemented inside the DBMS using a procedure which runs in sub-seconds. This scales independently of the catalog size or the total number of customers; it is dependent only on how many titles the user has purchased and rated. Therefore, this functionality will remain effective even as the business grows.

These recommendations would also be interpretable by the customer since they will cite which book spawned the recommendation. For example, the recommendations will be of the form

> Recommended Title: "Metaphor: A Computational Perspective"
> *because of your purchase of "Exploding The Creativity Myth"*

Interpretability also enhances the developers' ability to understand and debug the system. And in general, the algorithm performs well with limited user data, producing high-quality recommendations based on as few as two or three items[2].

That said, the database should not try to do too much with recommendations. It should not attempt to produce time-costly recommendations that do not scale well. Any procedures that output to the customer screen should be efficient because they will be used in real-time. The

database, above all, must carry out its prime responsibility: store the data in a usable, robust, and safe fashion. Any other capacities are bonuses. They can be implemented using other software. But if the database fails, they fail. Hence, importance of the design. And that is what we will discuss next.

Section 2: Database Plan: A Schematic View

The company's business plan is solely to sell books to customers. As such, it makes sense that the principal entities are the books and the customers. Authors and cities could be entities, but they act more like attributes since they do not have any attributes themselves. For this reason, they are considered as such.

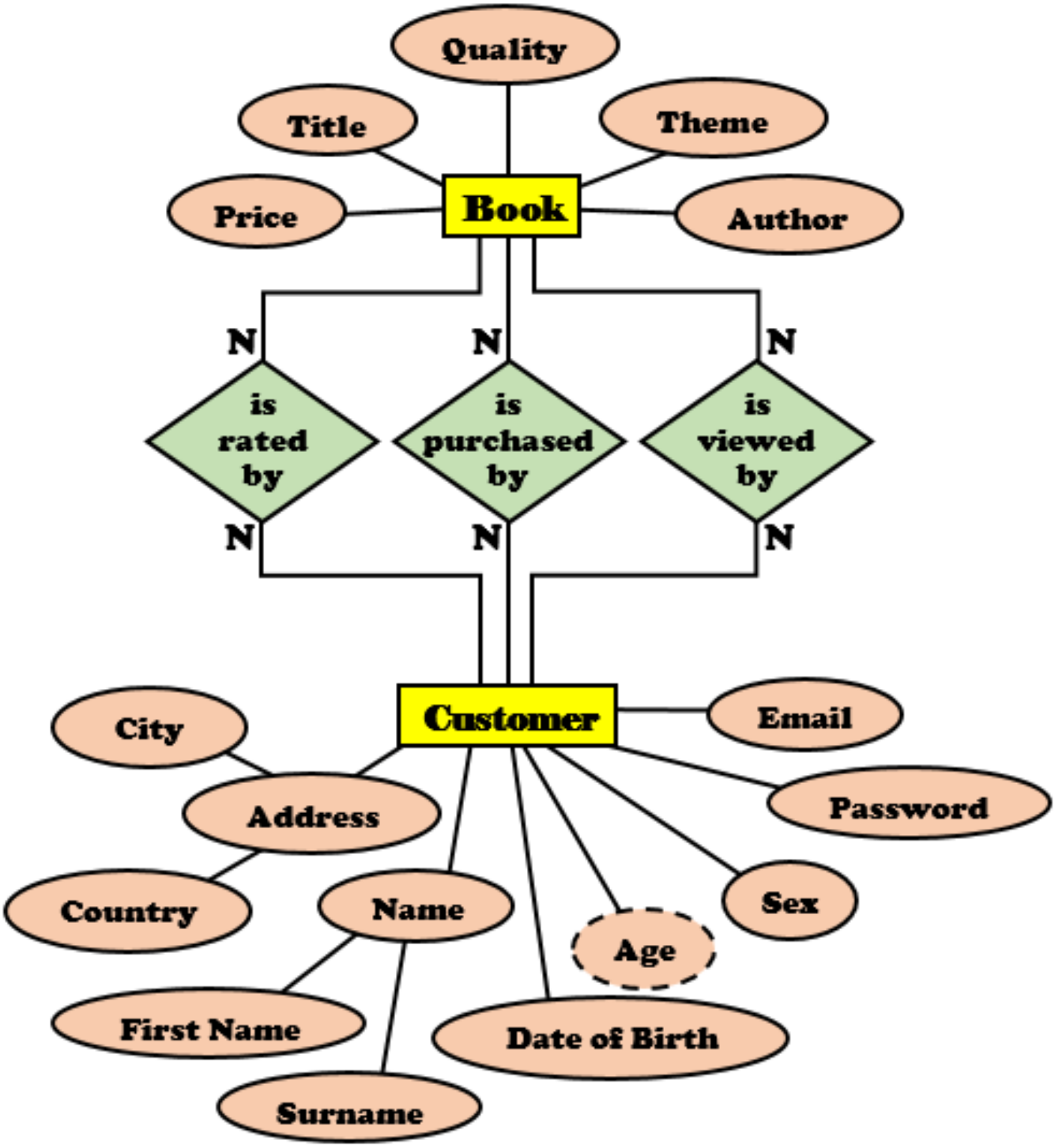
Books have two intrinsic attributes: their title and their author(s). The other ones are external, either imposed on them for classification purposes (themes and qualities) or for selling purposes (price). Customers have two composite attributes: their name (which is comprised of a first name and a surname) and their address (which is composed of a city and a country). Of course, an address is more complicated, especially since the business delivers its products, so a postcode would be more appropriate. I did not implement that, however. I instead favored the simpler approach of just retaining the customer's nearest city. This will help with city-wide analyses. In practice, though, both would be recorded.

The customer will have a date-of-birth, from which their age can be derived. They will also have a sex, email, and password. Some of these –such as sex– may not be provided by the customer, but others –such as email and password– are mandatory.

There are three main relations between books and customers: customers can buy the books, customers can rate the books, and customers can view the books. All three of these relations are many-to-many. Multiple customers can own the same book, and the same customer can own multiple books, and ditto for rating and viewing. To present this information in a more intuitive form, an Entity-Relationship (or E-R) diagram is displayed on the next page.

Customers have many other attributes: hair color, race, job, education level, to name a few. But these are either irrelevant or difficult to obtain at a large enough scale that they would improve recommendations. Moreover, it is usually considered a mistake to require a lot of information off the customer when they sign up because this will deter them. Instead, the company would probably favor asking for only the most worthwhile information, which are the attributes listed above. As for books, there are other attributes –most notably page length and published year– that would be useful for customers to know. This is because some people have preferences for shorter books and some, more recent ones. Unfortunately, however, these attributes were not known, so they could not be added. Nonetheless, there other unlisted attributes that will be added. These are ones the company can derive from its own data. They include average rating and bestsellers rank for books, and favorite author, theme, and quality for customers.

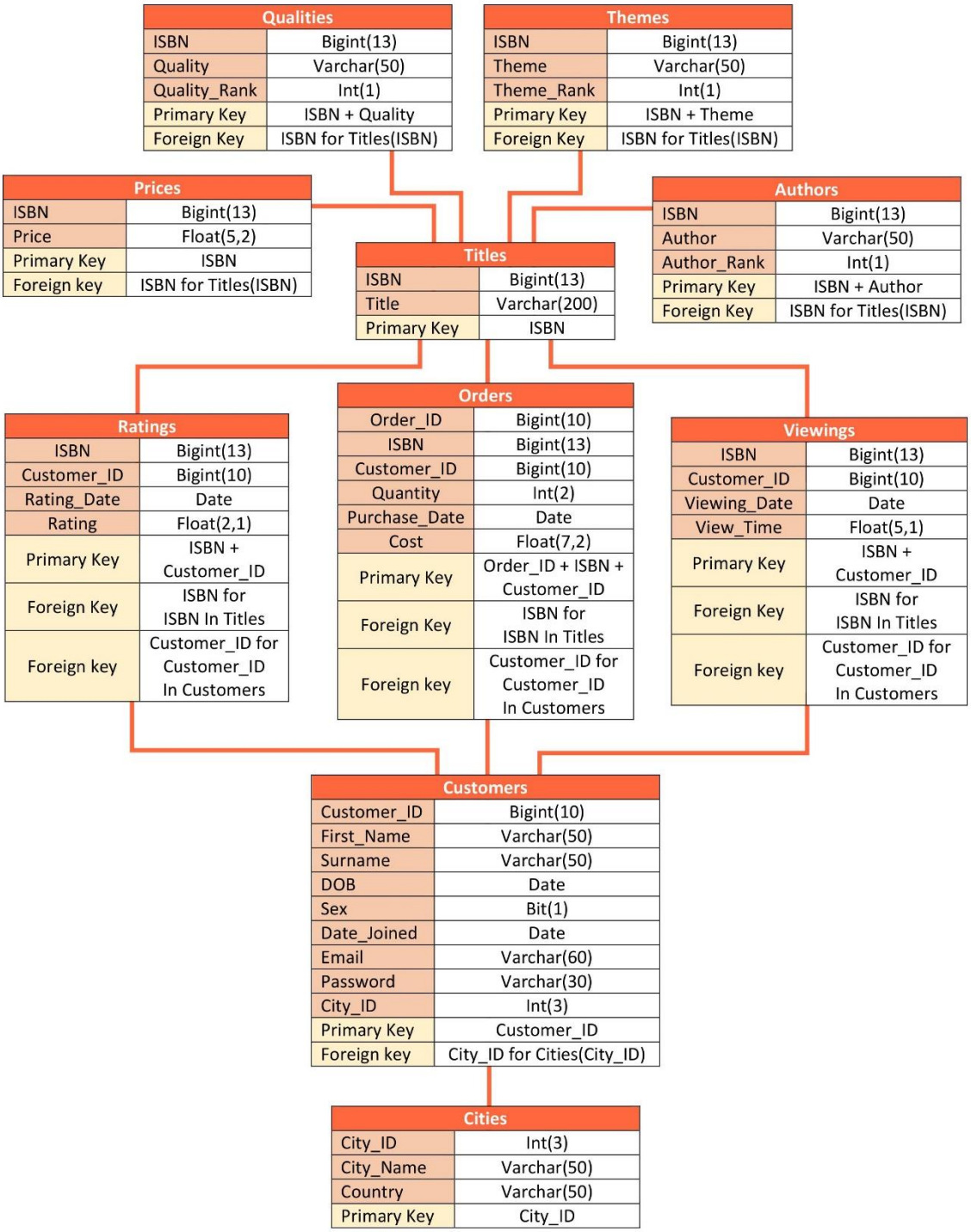
FIGURE 1: THE E-R DIAGRAM



Section 3: Database Structure: A Normalized View

The tables in the database are shown below.

Figure 2: The Database Plan



It is easy to see how this table layout resembles the E-R diagram. The books as entities are represented by their ISBN in the titles table. The attributes of the books such as authors, themes, and qualities (but not titles) are all stored in tables that reference the ISBNs in the titles table. It is a foreign key for their table. In fact, the lines joining the tables signify exactly this: that one table has a foreign key in the other. The customers are represented by their Customer_ID in the customers table. And the relations between the entities are captured in their own tables: ratings, orders, and viewings. Each references both entities, the ISBNs in the titles table and the Customer_IDs in the customers table.

Unlike most of the book attributes, the customer attributes are stored in the same table as their unique identifier. This is because there is a 1-to-N relationship between these values and the Customer_ID. That is, the same customer can only have one first name, surname, city, sex, date of birth, email, and password. (Multiple customers can, nevertheless, have the same first name, surname, city, sex, date of birth, and password.) This in stark contrast to the books, which can have multiple authors, themes, and qualities. That is why authors, themes, and qualities get their own table and first names, surnames, and date of births do not.

The only exceptions are prices and cities, which do get their own table even though they too have a 1-to-N relationship with their entities (books and customers respectively). The motivating reason for the cities table is that cities can be described by their name and their country, but there are some cities that have the same name and country. The town in which the Simpsons live, Springfield, is a prime example. There are 41 Springfields in the US alone. Therefore, it is necessary that we have an ID for each city so that cities with the same name and country will not be considered the same. Assigning a key to a city becomes easier if there is a cities table. Then if there is a new city, a new key and entry will be made. Moreover, putting the cities in their own table will remove repeating redundant information (such as the city name and country name) from the customers table.

The prices table, on the other hand, was constructed because some titles may not be for sale, especially old books whose stock will be no longer replenished. In this case, it is important that a record of the book be maintained (in the titles table) so that the orders table is not referencing a book which does not exist in the database. However, it is not important that a price also be maintained.

Ratings, orders, and viewings get different tables for obvious reasons, namely that they have different fields and are quite unrelated. The primary key in nearly all the tables is very obvious. In the titles and prices tables, it is the ISBN. In the authors, themes, and qualities it is the ISBN together with the author/theme/quality. In ratings and viewings tables, it is the ISBN together with the customer ID. And in the cities and customers tables, it is the city ID and the customer ID respectively. All of these primary keys make intuitive sense and provide indexes that will be used often. The only difference is the orders table, for which the primary key is the combination of order ID, ISBN, and customer ID. Why is not simply the order ID? Well the reason for this is that the same order can contain multiple books. Of course, one could pack the book ISBNs into one row with the order ID, but this would break the atomic value rule, which states that each

entry in a row-and-column intersection should be single-valued. And even if the ISBNs are packed into three separate columns, searching for books would be a lot more difficult. (This is also the reason the authors, themes, and qualities are not packed into one row for one ISBN but span many.) Another option is to simply consider different book purchases in the one transaction as different orders, but this would not be good practice because it is important to know the total transaction amount in order to detect cases of fraud. Also, it would be helpful to know which books are frequently bought together, so that the company can flash a suggestion at customers before they finish their payment. This is why the primary key is as it is. Although, it will not be useful as an index, so I simply created another; this time it is on customer ID.

It may seem that the rank columns in the authors, themes, and qualities tables are redundant, but they are not. The rank furnishes an ordering of the attributes for a given book. This is because, for instance, some books have primary authors who contribute more than the secondary author. In all books, some themes stand out more than others. And so on.

The table layout has hopefully been justified adequately so far. Now it must be shown that the tables are normalized. For the first normal form, this is quite easy. The column names in each table are unique. Each set of related data has its own table, with its own primary key. Interventions such as creating the cities table have removed any repeated information. There is a set of attribute values which uniquely determine every row in all tables, not least in practice but also in theory. The primary keys ensure they will be no duplicate rows in each table not only by preventing duplicate inserts but also by ensuring no null values are entered in a key column. Indeed there can be repeated important information like the same customer purchasing the same book at the same cost on the same date. It won't be a duplicate, however, because the order ID will be different. (If there are multiple books of the same ISBN in the same transaction, they will be grouped in the same row using the field quantity. That way, there will never be an entry with the same order ID, ISBN, and customer ISBN.) There is no ordering of the data. That is, we will not lose information if the rows or the columns are reordered. Every row-and-column intersection contains exactly one value. There is no mixing of data types in any column. In fact, even when different columns reference each other, the data type remains constant. An example of this is ISBN, which is a BIGINT(13) in every table which it appears. The same is true for customer ID and city ID.

So it seems the database is in 1NF. For 2NF, it must be demonstrated that there is no prime-attribute dependencies. These are cases some non-prime attribute is functionally dependent on some subset of prime-attributes (which is not the whole set of prime-attributes). Functionally dependent means the attributes are not just correlated with each other, but that one value of the first attribute is associated with precisely one value of the second attribute. An example of a violation of 2NF would be if each customer gave all their books the same rating. Here, rating, a non-prime attribute (because it does not appear in any candidate keys) is functionally dependent on customer ID, a prime attribute. The way to solve this would be to first, add a customer's rating value to their entry in the customers table, and second, to have only three columns in the ratings table: ISBN, Customer_ID, and Rating_Date. If an entry was made, then that book was rated by that customer. Of course, it is possible for a small database of books

and authors to fall into this situation, especially if almost all the customers had only rated one book. However, it would be a mistake to alter the database in the way described above because it is quite possible that at least one customer will give two of his/her books different ratings. Then the database is in trouble. Thus, what matters is whether the attributes are functionally dependent for all cases, or in theory. And there seems to be no places in this database where that is the case. (Customers usually give different ratings to different books.)

For 3NF, the question is whether some non-prime attributes are functionally dependent on other non-prime attributes, or do they depend on the key only. An example of what this could be is if every book cost the same amount but there were deals such as three of the same book for the price of two. Then the cost field in the orders table, a non-prime attribute, would be functionally dependent on the quantity field, another non-prime attribute, and so the database would violate the 3NF. In this case, it would be better to remove the cost column from the orders table and create a new table which details the costs for various quantities. Of course, this is not the case here. (Nor, by the way, is it the case that quantity and ISBN together functionally determine the cost. The prices table only holds current prices, so the price of the book at other times of purchase may have been different.) There seems to be no instances of the sort in this database. Thus, we can conclude that the database is in third normal form.

For Boyce-Codd normal form, a table must satisfy the condition that any functional dependency of the form X -> Y must either be itself a trivial one (Y is a subset of X) or have X as a superkey (a set of attributes which uniquely specify the rows). Clearly, a 3NF table that does not have multiple overlapping candidate keys is guaranteed to be in BCNF. This is because there is no column that is functionally dependent on anything but the key. And most tables in the database satisfy this constraint. In fact, very few tables have even multiple candidate keys. For instance, the cities table has only city ID as a candidate key; the customers table has only customer ID or potentially email; the prices and titles tables have only ISBN; the ratings and viewings tables have only ISBN together with customer ID; and the orders table has only order ID together with customer ID and ISBN. On the other hand, the authors, themes, and qualities all have an option of the ISBN with author/theme/quality or the ISBN with the rank. These candidate keys are obviously overlapping. Despite this, though, the authors, themes, and qualities tables are, in fact, in BCNF. This is because for every one of its dependencies X->Y (ISBN+Author -> Rank or ISBN+Rank -> Author), the X term is a superkey. That is, it uniquely specifies the rows: knowing the ISBN and the author/rank would always direct you to one and only one row.

(Note that ISBN has a BIGINT(13) data type because all ISBNs are thirteen digits long when one excludes the dashes. These are optimally stored as BIGINTs because they take up less memory than Varchars.)

Section 4: Views

The first view I created, “book_metrics”, was a perfect candidate for a view. It consolidated information from five tables (titles, prices, orders, ratings, viewings) into one by appending many performance metrics of each book to its row in the titles table. The DB user need only search for the ISBN of the book to attain its name, price, sales figure, recent sales figure, total-income figure, average rating, rating volume, view volume, and recent view volume. These metrics are usually all one needs to know to understand how well the book is faring with customers. The “recent” period referenced in two of those indicators was the period until now from the first day of 2018. It was only chosen as so because of the nature of my dataset; ideally it would be a shorter interval (perhaps a month).

This is what the view looks like:

title	isbn	price	total_spent	num_sales	num_recent_sales	num_customers	num_ratings	avg_rating	num_views	num_recent_views
Beyond Good and Evil	9780521779135	59.52	1249.92	21	7	14	4	2.87500	1	0
The Alchemist	9780062416216	64.98	1104.66	17	1	10	4	2.37500	0	0
Waiting for Godot	9780802198822	68.44	1095.04	16	10	9	4	2.87500	1	0
Digital Forensics	9781119262411	62.41	1060.97	17	0	11	3	2.16667	0	0
The Man Who Mistook His Wife for a Hat	9780684853949	56.03	1008.54	18	5	11	6	2.50000	1	0
Silence of the Lambs	9781446439746	68.97	965.58	14	3	8	3	3.33333	0	0
The Name of the Rose	9780544176560	68.94	965.16	14	6	7	2	1.75000	2	0
Ways of Looking: How to Experience Contempo...	9781780671932	63.87	894.18	14	2	9	1	0.00000	1	1

The view pulls in some summarized data and some single-valued data, each from very different tables, and it combines them into a coherent and intuitive format designed so that the user can query quicker and easier. The user could find the most expensive books, most sold books (bestsellers), most highly rated books, and highest trending books (most recently viewed books) all from a simple ORDER BY clause. The other benefit of combining data in this way is that the DB user can effortlessly use many disparate values (values derived through different means and from different tables) in a single query. For example, they could find the sales and rating figures for both the cheapest books and the most expensive to learn whether the firm’s consumer base is interested in bargains, and whether this affects their expectations of book quality. Similarly, the user could investigate whether strong ratings correlate with high sales volume. There are countless options. One need only click the “Export” icon above the result grid to open the data in Excel. That way the correlation between sales volume and price, say, could be calculated using only one preset function. In sum, all these tasks are much easier and quicker now that the data is merged into one view.

There are other reasons to justify the view’s existence, especially in opposition to other mechanisms. It has two main benefits over storing the query results in an actual table. First, it automatically updates (unlike the table) and second, it will not be called at an extremely high clip (like multiple times per second) such that running the query again and again would be vastly slower than executing it once in a while and storing the results. Rather, the only possible worry with this view is that there is no capability to create an index on a view in MySQL, so storing the results in a table would actually be worth it should there be constant use of WHERE clauses when querying the view. This, however, is not the case here.
As for stored procedures, their output cannot be queried from. But this is crucial functionality for actually using the view to generate insights. The “book_metrics” view is not a static

summary report but an expansive virtual table that functions as a tool to ease and quicken the user's interactions with the data. The view is also not burdened with parameters or variables, so it need not be encapsulated as a stored procedure.

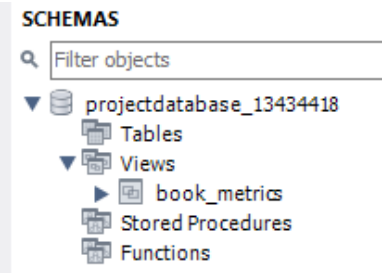
Finally, the view's advantages over simply rewriting the query again is that the query is long (35 lines long, in fact), so continually rewriting it would waste a lot of time. And this query is indeed useful enough that it would be used quite often.

Moreover, the view can be used to provide the common DB user will all the information they need to conduct analysis whilst maintaining the security of sensitive information on customers and orders, which is of utmost importance, especially in light of new EU (GDPR) regulations. This way the user can get sales figures without ever needing to access the orders table. And although alterations to the view can alter the basis tables, this can be prevented by only granting SELECT permissions to that user for this view.

The code below creates a user with only SELECT privileges on the "book_metrics" view:

```
create user 'Restricted_User'@'localhost'
identified by 'password';
grant select on ProjectDatabase_13434418.book_metrics to 'Restricted_User'@'localhost';
```

Logging in as this user shows that they only have access to the "book_metrics" view (and none of its basis tables):



If they try to edit the view, they will be met with an error:

```
update book_metrics set title = "Rodion's Reason" where title = "Crime and Punishment"
```

Error Code: 1142. UPDATE command denied to user 'Restricted_User'@'localhost' for table 'book_metrics'

The next views, which are named "author_metrics", "theme_metrics", and "quality_metrics", are very similar to the book one except that they group the books by some common attribute. There is only one new metric and that is sales per book (which tries to normalize so that categories with more books do not get inflated sales figures). Apart from that these views could be justified in exactly the same way as the books one. They should too be afforded to common users to obviate the need to access the orders or customers table, both of which should be cordoned off. The only new charge against their existence as views is that their queries are so similar that replacing "themes", say, with "table_name" and "theme" with "attribute_name"

could form the basis of a stored procedure which outputs the authors view if “attribute_name” was selected to “author”. But this does not help. The results of the stored procedure cannot be queried, so it is actually advantageous to instead write the same code multiple times and have three separate views.

This is what the authors one looks like (with a surprise author in 5th place):

author	total_spent	avg_price	sales_per_book	num_books	num_sales	num_recent_sales	num_customers	num_ratings	avg_rating	num_views	num_recent_views
Ian Fleming	3408.43	31.020833	9.0000	12	108	23	56	23	1.93478	9	6
William Shakespeare	2453.56	38.715714	9.4286	7	66	6	45	28	2.69643	11	4
Stephen King	2129.90	28.014000	7.7000	10	77	19	46	20	2.52500	9	2
Thomas Harris	2055.48	61.136667	11.0000	3	33	9	21	7	2.71429	3	2
Tony Veale	1909.61	50.690001	12.3333	3	37	8	24	9	2.50000	2	1
Neil Gaiman	1653.42	41.085001	10.0000	4	40	8	25	10	2.05000	7	3
J.R.R. Tolkien	1547.90	37.330000	8.8000	5	44	14	28	13	2.42308	5	3

The themes one:

theme	total_spent	avg_price	sales_per_book	num_books	num_sales	num_recent_sales	num_customers	num_ratings	avg_rating	num_views	num_recent_views
romance	13396.93	41.590000	8.2368	38	313	68	139	86	2.69767	42	16
adventure	13138.29	32.610625	8.6250	48	414	101	145	104	2.29327	55	36
science	13033.19	38.965128	8.3846	39	327	93	138	87	2.62069	53	24
love	11407.91	36.415128	7.8462	39	306	87	128	90	2.85000	52	22
mystery	10857.25	41.198333	9.1000	30	273	57	115	59	2.09322	35	18

And the qualities one:

quality	total_spent	avg_price	sales_per_book	num_books	num_sales	num_recent_sales	num_customers	num_ratings	avg_rating	num_views	num_recent_views
thrilling	26947.61	38.369500	8.8625	80	709	158	188	170	2.30000	91	45
exciting	26495.82	38.401316	8.9079	76	677	165	178	160	2.42188	95	46
suspenseful	22552.15	41.058636	8.1970	66	541	113	176	132	2.30682	80	35
dark	22339.37	38.771739	8.0435	69	555	135	172	124	2.52823	68	31
philosophical	21787.31	39.443750	9.5893	56	537	137	174	127	2.31102	78	42

These views allow one to easily find the authors/themes/qualities which are most popular for the firm’s customer base using the sales-per-book metric. This metric could advise the firm which books to invest more in (ones with the top performing categories) and which books to invest less (ones with the worst performing categories). At the same time, it is also helpful to know the total-amount-spent (or “total_spent”) metric because there may be categories of books with low sales-per-book but high sales in general. So this is an exception to the advice given from the previous metric: the business should not simply pull investment over the whole category which has low returns per book. Instead, the company should find the poorly-performing books within that category that drag the mean down. This nuanced advice is a great example of why having more metrics congregated together can produce more intricate strategies – and thus why views such as these are useful.

As well as informing the firm about what books to invest it, these views can also illuminate the business’s performance in targeting certain types of customer. For example, say there were many books in the online store geared towards teenage readers. If themes such as young adulthood, teenager, youth, school, social-anxiety, heartbreak, and nonconformity were all performing badly then perhaps the advertising to young people has been poor or the

recommendations have been poor, assuming that it is not just the same set of books in all of those categories.

Other useful measures include the number of recent sales and the number of recent views, which together give a proxy measure of current (or "trending") customer desires. Knowing which categories are now trending can help the business stay ahead of the curb, prepare its stock, and advertise more effectively.

Generally, though, the usefulness of these views is really the ability to have many measures available to condition on at once. There is no difficulty for the user to find poorly-rated authors that have sold well despite having high book prices. The query below illustrates this.

```
select * from author_metrics
where num_ratings > 5 and num_books > 3 and avg_rating < 2.5 and avg_price > 40
order by sales_per_book desc;
```

author	total_spent	avg_price	sales_per_book	num_books	num_sales	num_recent_sales	num_customers	num_ratings	avg_rating
Neil Gaiman	1653.42	41.085001	10.0000	4	40	8	25	10	2.05000

So even though Neil Gaiman has a terrible rating and high average book prices, his sales-per-book ratio is great. Perhaps this means his novels are controversial; many read out of curiosity, but few like them. Whatever the case may be, the point is that query took 3 lines to write yet it involved 4 tables (authors, orders, prices, ratings) and 3 different group by summaries (prices, ratings, sales all had different book sets). Thus insights are much more possible since so much *useful* data is at the DB user's fingertips.

The next view that I constructed was the "customer_metrics" one. It is shown below.

customer_id	first_name	surname	dob	sex	date_joined	city_id	num_purchases	num_recent_purchases
33	Eddy	Gordon	1997-10-15	0	2011-06-17	26	40	7
162	Donald	Peters	1975-07-11	0	2013-07-27	17	44	6
177	Alan	Boucher	1983-04-11	0	2011-06-03	46	40	11
8	Ramona	Stanley	1974-09-03	0	2013-01-19	21	37	10
195	Jeanette	Gamble	1984-02-22	0	2010-04-13	46	39	5
70	Lorna	Kavanaugh	1999-01-08	1	2009-09-11	5	42	6
124	Matthew	Gonzales	1990-02-03	0	2010-01-24	35	34	7

total_spent	num_ratings	avg_rating	num_views	num_recent_views	purchases_band
1886.26	9	3.27778	5	2	31-40
1776.95	10	2.35000	4	2	Over 40
1730.07	13	1.76923	7	1	31-40
1664.68	13	3.30769	3	1	31-40
1654.44	9	2.94444	3	2	31-40
1642.05	11	2.95455	2	1	Over 40
1594.33	9	2.72222	1	0	31-40

It helps the firm understand the relationships between various activities (viewing, rating, and buying) and the demography of the customers engaged in them. Answering questions like "what is the age range and sex of the company's most active customers?" could improve how the company markets itself by better targeting its audience. The buck does not stop there, however; there are countless other questions that could be answered using this view. For instance, is the company struggling to get male twenty-year-olds to buy more than a few books each? Are the most prolific raters reflective of the whole customer base, or is it mostly women and older men? Which type of customer is viewing many items but not buying any, and why? And again, like the other views, having the data altogether can really improve the user's ease and efficiency when interacting with the database to generate insights, especially for specific searches (which often involve conditioning).

Why have this functionality as a view? Well, a table would be of no use since customer information changes significantly in even short time periods when the customer base is of a major online bookseller. So dynamism is important. But a stored procedure is not the solution since querying the results is instrumental in gaining any insights. A rewritten query wouldn't suffice due to the time wasting. Thus, a view is the firm's best option.

There are no security benefits to this view since it simply adds even more sensitive data onto one screen. Therefore only users who need to know customer information should have access to this view.

The "city_breakdown" view, on the other hand, removes all sensitive information. It acts a report that presents all the summary statistics for cities. This was be ideal for the executives to get a sense of which locations are vital, and what types of people live there. This is what it looks like:

city_name	city_id	country	total_spent	num_book_sales	num_customers	avg_customer_value	num_elderly	num_adults	num_teenagers	proportion_male
Bremen	46	USA	26055.32	659	26	1002.127693	1	25	0	0.0385
Flint	26	USA	20329.38	513	19	1069.967369	0	19	0	0.0000
West Burke	5	USA	19767.33	508	19	1040.385788	1	0	18	1.0000
Jacksonville	37	USA	17575.26	429	18	976.403335	0	17	1	0.0556
Culver City	45	USA	17292.17	446	17	1017.186469	16	1	0	0.9412

The hope is that managers would be given access to this view so that they obtain all relevant city information without ever needing to know customer information. This view will then give them a sense of which cities are most profitable as measured by the average customer value (total_spent / num_customers). Also, it could give the managers an indication of how best to market the company in different cities, especially considering how the demography changes (through shifts in proportions of elders, adults, and teenagers, and males and females).

This view will not be called all that often, so it need not have its results stored in a table. It should not, however, be rewritten again and again because it is also quite a long query (at 21 lines). And it also should not be coded as a stored procedure since managers would still want to query it, and changing the output through grouping by country or ordering by different metrics or filtering by certain conditions. The list currently has only 14 cities with customers, so it seems

like such capacities are unnecessary, but for a major online bookseller there would be a copious amount of cities to consider, which would then render this view invaluable.

All views I have included are intended to be non-editable, this works well because update and delete statements do not work on them.

Section 5A: Triggers

The first trigger I made was the "before_orders_changes" one. It tracks alterations to the orders table, recoding the values in each field prior to and after the change. It dumps this information into the orders_changes table. For example, this is the result of a simple change:

```
update orders set cost=30.14 where order_id=1;
select * from orders_changes;
```

change_id	old_order_id	old_isbn	old_customer_id	old_quantity	old_cost	new_order_id	new_isbn	new_customer_id	new_quantity	new_cost	change_date	action
1	1	9780300210798	77	1	12.44	1	9780300210798	77	1	30.14	2019-04-18 15:15:59	update

The "before_prices_changes" one does the same job.

```
update prices set price=500 where isbn=9780007289486;
select * from prices_changes;
```

change_id	old_price	old_isbn	new_price	new_isbn	change_date	action
1	22.08	9780007289486	500.00	9780007289486	2019-04-18 15:21:46	update

These triggers are useful in monitoring important tables. Prices of books stored in the prices table will probably be used in determining payments on the site. For this reason, it is crucial that the firm keep a close watch on the prices table, noting any changes whatsoever, lest there be any skullduggery or bungling that goes unnoticed. But the uses of monitoring do not stop there. Recording price changes can also inform the company how successful the discount offerings or flash sales have been. With the price changes stored, one can find the books that have endured a significant cut over some selected period of time by filtering the prices_changes_table. Then join this with the orders table and compare the number of sales before and after the price slash. Doing this for all books could suggest further avenues for the business: which discounts work and which don't.

The orders table is of comparable importance. It is the core record of the firm's business transactions, and as such, it is a target of foul play. For legal reasons if nothing else, the company should observe any changes to these records.

The purpose of the "before_insert_viewings" trigger is to prevent already-purchased books from entering the viewings table for a certain customer. This is because the viewings table is intended to guide the firm on potential purchases, but most customers will not want to buy a book they already own. The trigger will throw an error before the bought item is inputted to the viewings table.

```
3  13:12:32  insert into viewings values (9780300210798,77,'2018-07-08',578)
Error Code: 1644. Don't add a bought item to viewings_table
```

This is because the customer with ID 77 has already purchased the book with ISBN

```
select * from orders where isbn = 9780300210798 and customer_id = 77;
```

order_id	isbn	customer_id	quantity	purchase_date	cost
1	9780300210798	77	1	2017-10-24	30.14

The “before_insert_ratings” trigger is the same in every respect except that it checks if a book is *not* purchased rather than checking if a book *is* purchased. The motivating idea is that ratings of unpurchased books may not be genuine for many reasons. Firstly, the reviews produced by users with many ratings often come earlier in the comments thread. This gives them the incentive to produce more ratings; the more, the better. And since the quality of the ratings is irrelevant to the ranking there is no real need to even purchase the book so that they produce a fair and informed review. With this in mind, then, the best strategy for them to pursue is to tile the site with ratings and reviews. Secondly, authors and publishers will want to shape public opinion as much as possible and therefore will naturally have an incentive to create fake positive reviews or smear their rivals. Thus, it is clear that there will be fake reviews, which will undermine the firm’s recommendation model. However, the requirement of having to purchase the book to be able to rate it does alleviate the problem. That is why this trigger is in place.

So quite like the viewings trigger, this one will throw an error to prevent the execution of the insert:

```
7  13:26:08  insert into ratings values (9780300210798,78,'2018-07-08',4)  Error Code: 1644. Don't add rating if not bought
```

This is because customer with ID 78 has not purchased the book with ISBN 9780300210798.

```
select * from orders where isbn = 9780300210798 and customer_id = 78;
```

order_id	isbn	customer_id	quantity	purchase_date	cost
NULL	NULL	NULL	NULL	NULL	NULL

The final trigger I included was the “after_orders_insert” one, which checks if a purchased item has been inputted into the viewings table. If it has, then it will be deleted. The code below shows the disappearance of a view from the viewings table, after the book with ISBN 9780006473299 was purchased by that customer with ID 55.

```
select * from viewings where isbn = 9780006473299 and customer_id = 167;
```

isbn	customer_id	viewing_date	view_time
9780006473299	167	2017-06-20	2935.8

```
insert into orders values (2919,9780006473299,167,1,'2019-01-01',100.00);
select * from viewings where isbn = 9780006473299 and customer_id = 167;
```

isbn	customer_id	viewing_date	view_time
NULL	NULL	NULL	NULL

Section 5B: Other Procedures

The stored procedures described below did not use any views in their code. If that were the case, queries would be inefficient on account of all the extra joins which are irrelevant to the purpose of the procedure. Another thing to note is that, where possible, the asterisk which denotes all columns in a table or view, was often omitted from the stored procedures. This is simple because it is faster in SQL if all the column names are listed instead. I did this for some parts of the views but not all because they won't used as frequently as the stored procedures. Finally, the purpose of this project was to build a database that supports a bookseller's catalogue and recommendation system. But there is a fine line between some functionalities that support recommendations (outputting bestsellers in the customer's favorite category) and those that do the recommendation itself (outputting books which have similar themes and qualities to the customer's collection). Moreover, SQL has many benefits –such as its efficient query mechanisms and indexing– which can be leveraged to improve or produce recommendations. For these reasons, I did construct some procedures to provide recommendations.

The first six procedures are all intended to be used by the customer. Numbers 3-6 all involve outputting a list of book titles to the customer's home screen, whereas the first one is a page that can be accessed on the website home page, and the second one is used when a customer clicks on the links provided one that page.

"Best-Selling or Best-Rated Titles / Authors / Themes / Qualities"

The first procedure takes as input the attribute to output (book titles, authors, themes, or qualities), the page number (page 0 lists the top ten; page 1 lists numbers 11-20; page 2, 21-30, etc.), and an indicator for whether sorting should be by purchases or ratings. (Also, when sorting by ratings the procedure will only consider books with a minimum number of ratings.) Its output for various inputs is shown below.

Page 0 (#1-10) of the Bestselling Books:

```
call bestsellers_or_bestrated('title',0,0);
```

title
Full Disclosure
Beyond Good and Evil
Are You There, God? It's Me, Margaret
The Tempest
The Sun Also Rises
The Man Who Mistook His Wife for a Hat
The Metamorphosis
Cold Comfort Farm
The Big Short
The Alchemist

Page 2 (#21-30) of the Best-Rated Themes

```
call bestsellers_or_bestrated('theme',1,1);
```

theme
author
authorship
marriage
gender
dystopia
drugs
youth
crime
science communication
science communicators

This is an important procedure because the company will be without data on many customers who have just joined the site (this is known as the cold-start problem). These customers will receive no recommendations. Therefore, the company needs some other mechanism to spark some interest in the customer. A bestsellers or best-rated list broken down by title, author, theme, and quality is a good start. This provides a kind of catalogue for the customers to

browse through the top titles, authors, themes, or qualities. The next procedure makes the catalogue even more interactive by creating links on each category to filter by that category.

"Best-Selling or Best-Rated Books by Author, Theme, or Quality"
The idea of this procedure is that once the customer sees an author, theme, or quality that they like when browsing the list provided by the previous procedure, they should be allowed to click on that category and have it return a list of the best-selling or best-rated books in that category. For instance, say a customer is on Page 2 of the best-rated themes, and they see the theme "dystopia". They could then click the link and be brought a page where this is shown:

	title	avg_rating
▶	Fahrenheit 451	3.62500
	The Handmaid's Tale	3.20000
	The Planet of the Apes	3.00000
	The Giver	2.25000
	The Chronicles of Thomas Covenant the Unbeliever	2.00000

```
call best_titles_by_category('theme','dystopia',0,1);
```

"What You've Recently Viewed"
The "recently_viewed" procedure simply returns the five most recently viewed books for a specified customer, excluding books which were viewed for only two seconds or less. Here's an example:

```
call recently_viewed(2);
```

	title
▶	James and the Giant Peach
	The Spy Who Loved Me
	The Joy of Sex
	Middlemarch
	Fifty Shades Of Grey

(The only input is the Customer_ID.)

"Other _____ Bestsellers"
The procedure with name "other_bestsellers_in_your_fav_category" finds which attribute (author, theme, or quality) is most common among a specified customer's purchase history. Then it finds the bestsellers with that attribute but were not already purchased by the customer. Of these it returns the top ten. For example, customer with ID 54 would see this on his/her screen:

other_suspenseful_bestsellers
A Reaper at the Gates
Gone Girl
The Day of the Jackal
The Dark Half
American Gods
The Strange Case of Dr Jekyll and Mr Hyde
And Then There Were None
The Thirty-Nine Steps
The Snowman
Where Eagles Dare

```
call other_bestsellers_in_your_fav_category(54);
```

Customer 54 does like suspenseful books (with 7 such purchases), and the list is sorted in descending order by the number of sales.

"Books Similar in Content to Those You've Bought"
The "recommended_for_you_by_content" procedure finds a specified customer's viewed books, and their top 10 favorite themes, qualities, and authors (from looking at their purchase history). It then returns the books that most closely match these attributes (counting each attribute as equal), and it excludes books which have already been purchased. The recommendations provided for the customer with ID 9 are displayed below.

```
call recommended_for_you_by_content(9);
```

title	num_matches
The Lord Of The Rings: The Fellowship of the Ring	6
The Lord Of The Rings: Return of the King	6
LA Confidential	4
Red Dragon	4
Silence of the Lambs	3
The Hobbit	3
A Midsummer Night's Dream	3
A Clockwork Orange	3
London Fields	3
Hannibal	3

As you can see, these recommended books tend to have a lot in common. Three are from the same author (J. R. R. Tolkien wrote "The Lord of the Rings" series and "The Hobbit"). And naturally enough, the customer has actually purchased the "The Lord of the Rings: The Two Towers". The code features an oddity analogous to "SELECT * from (SELECT * from table_name)". That is, there is a seemingly redundant SELECT, but it is purposeful because MySQL forbids using LIMIT in subqueries unless they are used in derived tables. Ideally, this procedure would be executed when the user opens up the site homepage (while logged in). The

output of the procedure would then be shown onscreen to the customer under the banner "Books Similar In Content To Those You've Bought".

"Recommended Books" and "Similar to Those You've Viewed"

First, two similarity functions were constructed: "purchase_similarity" and "rating_similarity". They both take as input two ISBNs and return a similarity score between the two books (one called i and one called j). For the purchase one, this is the cosine similarity $= k * \frac{n_{ij}}{n_i * n_j}$, where n_{ij} denotes the number of customers who have purchased both books i and j; n_i denotes the number of customers who have purchased book i (and similarly for book j); and k is given by the formula $k = \frac{min(n_i\ ,\ n_j)}{15}\ if min(n_i\ ,\ n_j) < 15;\ otherwise\ k = 1$. The idea is the two books are more similar if more customers (as a proportion) buy both books rather than only one of them. k, then, is a penalty term applied when one or both of the books in consideration are rarely purchased. This is because the score may not be reflective of similarity at these small scales. For example, had the two books only had one purchaser that happened to be the same person, they would be awarded a score of 1, representing perfect similarity, but we know this could easily have been a random fluctuation. Only from repeated occurrences can it be inferred that the books are indeed quite this similar.

The rating function, on the other hand, must be more sophisticated since ratings can take 11 different values {0, 0.5, 1, 1.5, ... , 3.5, 4, 4.5, 5} rather than only binary ones {0, 1} like those in customer purchases. The chosen metric was then Pearson's correlation, which is given by the formula $\frac{\sum_{c=1}^{n}({}_cr_i - \overline{r_i})({}_cr_j - \overline{r_j})}{\sqrt{\sum_{c=1}^{n}({}_cr_i - \overline{r_i})^2}\ * \sqrt{\sum_{c=1}^{n}({}_cr_j - \overline{r_j})^2}}$, where only customers who have rated both are considered, and n is the number of these. ${}_cr_i$ denotes the rating of customer c on book i, and $\overline{r_i}$ denotes the mean rating of book i across all customers who have rated both books. These terms are denoted similarly for book j. The resulting score compares how much the ratings of both books vary together for certain customers with how much each book's ratings vary in general. The idea is that if the ratings of both seem to mirror each other, especially compared to their respective means, then the two books are similar. For example, say book i had a mean rating of four stars and book j, two stars. Then the books would be considered similar if customers that thought book i was better than four stars also felt that book j was better than two stars, and customers that rated book i less than four stars also rated book j less than two stars. (Obviously the amount by which they rate the books higher or lower is also considered in the measure.) In terms of the formula, though, this basic situation translates to two positive terms or negative terms multiplied by each other in the numerator (because both or neither ratings are greater than their means). Either way, the result is that the numerator becomes larger, and therefore, the larger the correlation value, the more similar the book.

Then the table "similarities" was created. It takes all the books that have been ordered enough times (in this case 10 times) and stores all the possible pairs of these. For each pair, the purchase similarity and rating similarity is calculated and stored alongside their entry in the

table. Only the books that have a minimum purchase similarity are kept; all others are removed.

Recommendations are bidirectional. That is, for two similar books i and j, "if you've bought i but not j then buy j", but also "if you've bought j but not i then buy i". This can make for awkward searching. Instead of this, the entries in the table are duplicated but with their ISBNs swapped. That way the columns will represent the recommended book ("recommended_title") and the book that spawned the recommendation ("from_title"). Now, searching for recommendations for a certain customer will only require looking row-by-row.

The procedure which then outputs the recommendations for a specified customer is called "recommender". It finds all the rows of the similarities table where the customer has not purchased the recommended book but has purchased the "from_title" book (this search is now very easy due to the aforementioned duplication). The procedure then sorts these results by purchase similarity (in descending order) and returns the top ten recommendations. If the "viewed_indicator" input is set to 1, the "recommender" procedure will find books similar to those the customer has viewed rather than purchased.

Here it what it looks like for the customer with ID 89:
"Similar to Those You've Bought"

```
call recommender(89,0);
```

	recommended_title	from_title
▶	Digital Forensics	Are You There, God? It's Me, Margaret
	Cold Comfort Farm	The Tempest
	A Reaper at the Gates	The Tempest
	Full Disclosure	Are You There, God? It's Me, Margaret
	The World According to Garp	Are You There, God? It's Me, Margaret

"Similar to Those You've Viewed"

```
call recommender(89,1);
```

	recommended_title	from_title
▶	Foucault's Pendulum	Girls & Sex: Navigating the Complicated New La...
	Cold Comfort Farm	Girls & Sex: Navigating the Complicated New La...
	Animal Farm	Girls & Sex: Navigating the Complicated New La...
	The Man Who Mistook His Wife for a Hat	Girls & Sex: Navigating the Complicated New La...

The benefit of these recommendations is that customers get to see why they have been recommended certain titles (using the "from_title" column). These recommendations used only the purchase similarity, but this was only due to the nature of the dataset; there were very few books rated by the same customer. In practice, there would be many, in which case some combination of the two similarities would be used: either a simple linear combination or even one that involves treating purchases without ratings as medium ratings (like three-star ratings). The reason for creating the similarities table is that it would take a lot of time to recalculate all the metrics, so it is better to do it only semi-regularly and store the results in a table, rather

than running it every time a customer logs in. This is an example of item-to-item similarity in collaborative filtering. (MORE ON ITEM-TO-ITEM)

The remaining two procedures are intended for use by the DB user. The first of these is called "customer_favorites". It takes as input the Customer_ID, the attribute (title, author, theme, or quality), the amount of results to return, and an indicator variable which determines whether the favourites will be determined by purchases or ratings. It also appends some information onto the rows. This differs between singular data such as book titles and grouped data such as authors, themes, or qualities. For example, with titles the extra information is the customer's rating of the book and the data of the rating. With themes, on the other hand, the extra information regards the books in that theme category is the number of book purchased, the number of ratings, the average rating, and the most recent purchase date.

Here is it in practice:
Customer 54's Favourite (10) Books by Rating:

```
call customer_favorites(54,'title',10,1);
```

	title	rating	rating_date
▶	Married Lovers	4.5	2018-01-27
	Candide	4.0	2015-10-06
	Pride And Prejudice	4.0	2017-04-07
	The Joy Luck Club	3.0	2015-12-15
	The Handmaid's Tale	1.5	2015-12-01
	Lord of the Flies	1.5	2018-04-28

Customer 54's Favourite (5) Themes Ranked by the Amount of Purchases:

```
call customer_favorites(54,'theme',5,0);
```

	theme	num_purchases	most_recent_purchase_date	num_ratings	avg_rating
▶	dating	4	2017-04-07	1	4.00000
	relationships	4	2017-04-07	1	4.00000
	murder	4	2017-11-16	0	NULL
	courtship	2	2017-04-07	1	4.00000
	philosophy	2	2015-10-06	1	4.00000

NULL is purposely used for the values of the average rating when there has been no ratings. This is so because any number (such as 0) would mislead the DB user into thinking that the specified customer had in fact rated books in that category.

The "city_favorites" procedure is very similar though the results are for specific cities rather than customers, and the addition of the measure "recent_purchases" which considers books purchased in the last year. (Ideally it would be a shorter interval, like those purchased in the last month). The output of this procedure is displayed below.

City 5's (5) Favourite Books by Purchases:

```
call city_favorites(5,'title',5,0);
```

title	num_purchases	num_recent_purchases	total_spent	num_ratings	avg_rating
A Prayer for Owen Meany	10	4	626.70	6	3.75000
The Big Short	10	0	109.50	6	2.75000
The Adventures of Huckleberry Finn	9	0	273.60	9	3.16667
Neverwhere	8	0	147.52	6	4.50000
An Inconvenient Truth	8	0	150.96	6	3.25000

City 5's (3) Favourite Authors by Ratings:

```
call city_favorites(5,'author',3,1);
```

author	num_purchases	num_recent_purchases	num_ratings	avg_rating
Stephen King	13	4	4	3.62500
William Shakespeare	8	3	4	3.00000
Mark Twain	6	0	4	2.87500

Section 6: Example Queries

A customer creates an account with the company. They must be added to the table customers. There is no need to find a unique Customer_ID and assign it to the customer in the insert statement because that column is already set to auto-increment, which will automatically do it for the user.

Adding Luke Killoran to the database looks like so:

```
insert into customers (first_name, surname, dob, sex, date_joined, email, password, city_id)
values ("Luke", "Killoran", "1995-01-01", 1, "2019-04-21", "lk@gmail.com", "password", 5);
```

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
201	Luke	Killoran	1995-01-01	1	2019-04-21	lk@gmail.com	password	5
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46
198	Alma	Schulz	1984-02-22	0	2010-04-13	AlmaSchulz@gmail.com	BY3b8Zpo	46
197	Manuel	Powell	1984-02-22	0	2010-04-13	ManuelPowell@gmail.com	BY3b8Zpo	46

Should he decide to delete his account we could delete him from the database.

```
delete from customers where customer_id = 201;
```

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46
198	Alma	Schulz	1984-02-22	0	2010-04-13	AlmaSchulz@gmail.com	BY3b8Zpo	46
197	Manuel	Powell	1984-02-22	0	2010-04-13	ManuelPowell@gmail.com	BY3b8Zpo	46
196	Ola	Yepsen	1984-02-22	0	2010-04-13	OlaYepsen@gmail.com	BY3b8Zpo	46

Only certain data format errors will be caught. For instance, an input that is too long will be caught. Consider entering “male” instead of 1 for the field “sex”.

```
insert into customers (first_name, surname, dob, sex, date_joined, email, password, city_id)
values ("Luke", "Killoran", "1995-01-01", "male", "2019-04-21", "lk@gmail.com", "password", 5);
```

Error Code: 1406. Data too long for column 'sex' at row 1

But numbers instead of character strings will not throw an error.

```
insert into customers (first_name, surname, dob, sex, date_joined, email, password, city_id)
values ("Luke", 3.5, "1995-01-01", 1, "2019-04-21", "lk@gmail.com", "password", 5);
```

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
202	Luke	3.5	1995-01-01	1	2019-04-21	lk@gmail.com	password	5
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46

If the DB user tried to add a customer and assign them a Customer_ID which has already been assigned, then the database will throw an error.

```
insert into customers (customer_id, first_name, surname, dob, sex, date_joined, email, password, city_id)
values (200, "Luke", "Killoran", "1995-01-01", 1, "2019-04-21", "lk@gmail.com", "password", 5);
```

Error Code: 1062. Duplicate entry '200' for key 'PRIMARY'

The DB user is allowed to exclude some information that the customer did not provide. These will be filled with NULL values.

```
insert into customers (first_name, surname, date_joined, email, password)
values ("Luke", "Killoran", "2019-04-21", "lk@gmail.com", "password");
```

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
203	Luke	Killoran	NULL	NULL	2019-04-21	lk@gmail.com	password	NULL
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46

On the other hand, fields such as “first_name”, “surname”, “email”, and “password” are marked NOT NULL, so if there is no entry for one of them, an error will prevent the insertion. (Passwords are hashed before being stored in the database.)

DB users can edit the entry too if they have access. Say there is a mistake with the original entry. They can correct this using an UPDATE statement.

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
201	Puke	Killoran	1995-01-01	1	2019-04-21	lk@gmail.com	password	5
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46

```
update customers set first_name = "Luke" where customer_id = 201;
```

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
201	Luke	Killoran	1995-01-01	1	2019-04-21	lk@gmail.com	password	5
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46

DB users can also interact with the customers table using SELECT statements. For example, one could find all the customers that are older than Luke yet living in the same city.

```
select * from customers where dob < '1995-01-01' and city_id=5 order by customer_id desc;
```

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
62	Alicia	Robbins	1975-03-20	1	2014-06-19	AliciaRobbins@gmail.com	3TIF8Sdl	5

(Only one such customer.)

If, however, Luke lives in a new city, the city must first be added before Luke is. This is because the “city_id” field in the cities table is a foreign key to the “city_id” field in the customers table.

```
insert into customers (customer_id, first_name, surname, dob, sex, date_joined, email, password, city_id)
values (201, "Luke", "Killoran", "1995-01-01", 1, "2019-04-21", "lk@gmail.com", "password", 51);
```

Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails

Instead:

```
insert into cities values (51, "Dublin", "Ireland");
insert into customers (customer_id, first_name, surname, dob, sex, date_joined, email, password, city_id)
values (201, "Luke", "Killoran", "1995-01-01", 1, "2019-04-21", "lk@gmail.com", "password", 51);
```

city_id	city_name	country
51	Dublin	Ireland
50	Sugarcreek	USA
49	Denver	USA

customer_id	first_name	surname	dob	sex	date_joined	email	password	city_id
201	Luke	Killoran	1995-01-01	1	2019-04-21	lk@gmail.com	password	51
200	Terra	Goleman	1984-02-22	0	2010-04-13	TerraGoleman@gmail.com	BY3b8Zpo	46
199	Ricardo	Samuel	1984-02-22	0	2010-04-13	RicardoSamuel@gmail.com	BY3b8Zpo	46

The same goes for Customer_ID in the orders, ratings, and viewings tables. There must be referent in the customers table with that ID for the transaction to be added. It is also the case that there must be a referent in the titles table with that ISBN for any record to be added to the prices, authors, themes, qualities, orders, ratings, and viewings tables. This ensures that there is no mismatches and anomalies between tables.

The new customer can now make orders, ratings, and viewings.

```
insert into orders (order_id, isbn, customer_id, quantity, purchase_date, cost)
values (2920, 9781514683682, 201, 1, '2019-04-21', 1*(select price from prices where isbn=9781514683682));
```

order_id	isbn	customer_id	quantity	purchase_date	cost
2920	9781514683682	201	1	2019-04-21	42.59
2919	9780006473299	167	1	2019-01-01	100.00
2917	9781472119742	187	2	2018-09-28	82.46
2916	9781439198612	167	1	2017-12-03	46.89
2915	9780141918921	177	2	2015-05-16	18.74

Here the cost was determined by the quantity times the current price, which was retrieved from the prices table. In practice, this probably would be unnecessary, but this is just an example.

```
insert into ratings (isbn, customer_id, rating_date, rating)
values (9781514683682, 201, '2019-04-21', 3.5);
```

isbn	customer_id	rating_date	rating
9781514683682	201	2019-04-21	3.5
9781250146588	129	2018-12-28	4.5
9780062570611	75	2018-12-28	3.5
9780141394602	178	2018-12-25	3.0
9780803292406	115	2018-12-25	5.0

Of course, the triggers are still in place, so the user cannot rate any books except the one he bought.

```
insert into viewings (isbn, customer_id, viewing_date, view_time)
values (9780385537674, 201, '2019-04-21', 200);
```

isbn	customer_id	viewing_date	view_time
9780385537674	201	2019-04-21	200.0
9780141378534	148	2018-12-28	2749.7
9780743477741	114	2018-12-27	4887.4

Obviously, there is no need to list all the column names when I am inserting information about all of the fields. Nevertheless, listing the column names illustrates it better by providing information to you, dear reader, about which columns are getting which values.

Now that the customer has entered other tables such as orders, ratings, and viewings, it cannot be deleted from the customers table due to the ON DELETE RESTRICT constraint in those three tables. This is because we do not want delete those records or have a NULL customer in them.

```
delete from customers where customer_id=201;
```

Error Code: 1451. Cannot delete or update a parent row: a foreign key constraint fails

However, if for some reason we update the customer information, the changes cascade down to those tables due to the ON UPDATE CASCADE constraint in those three tables. This is because we want the tables to reference the correct customer at all times.

```
update customers set customer_id = 300 where customer_id = 201;
```

order_id	isbn	customer_id	quantity	purchase_date	cost
2920	9781514683682	300	1	2019-04-21	42.59
2919	9780006473299	167	1	2019-01-01	100.00
2917	9781472119742	187	2	2018-09-28	82.46

isbn	customer_id	rating_date	rating
9781514683682	300	2019-04-21	3.5
9781250146588	129	2018-12-28	4.5
9780062570611	75	2018-12-28	3.5

isbn	customer_id	viewing_date	view_time
9780385537674	300	2019-04-21	200.0
9780141378534	148	2018-12-28	2749.7
9780743477741	114	2018-12-27	4887.4

The same goes for ISBN changes and deletes in the titles table, and City_ID changes and deletes in the cities table. Some tables restrict the deletion of the ISBN and some cascade the deletion, but the restriction conditions will overrule the cascade ones.

Another small thing to note is that the orders table has an index that is not its primary key.

```
index(customer_id)
```

It has an index on Customer_ID. This will speed searches that condition for a certain customer. It was necessary to add this index because they crop up quite often. The index that is created as a result of the primary key is of no use, especially since the same order can span multiple rows.

Section 7: Conclusions

In conclusion, I have built a database that should support the business's catalogue and book recommendation system. The database should be able to cope with expansions in the customer and product base. It seems to have no issues, and increasing capacity should not slow the process much. The only functionality which will suffer is the item-to-item recommendation process, but this will alleviated by storing the recommendations in the similarities table, and only updating semi-regularly.

This is not the final version of the project, however; there are plans to improve the database, most notably on nine different frontiers. First, users should be created with the appropriate permissions. City managers, for instance, could be granted SELECT privileges on the "city_breakdown" view as well as the "city_favorites" procedure.

Second, instead of plainly recording one view, it is hoped that the database will store how many times a customer has viewed an item as well as the immediate path whence they came (whether it be through the search function, the bestsellers catalogue, or a recommendation list). This will indicate which avenues the firm should focus on, and more importantly, which recommendations work.

Third, reviews should be captured too, on which the firm could use simple sentiment analysis to extract even more information. More importantly, though, it could filter out reviews which aren't about the book but about irrelevant issues such as delivery time, for example. (These are not irrelevant to the business but irrelevant to the book.) Furthermore, many customers enjoy reading each other's opinions on a book rather than examining the book's mean rating, so this would be a welcome modification from a user-experience point of view.

Fourth, it is hoped that order, rating, and viewing date-times could be recorded rather than only dates. This could be helpful in pinpointing specific transactions. For the similar reasons, postcode rather than only city and country information would be added. This would also help with more granular geographic recommendations.

Fifth, the current swath of procedures should allow the customer to browse the bestsellers catalogue and be presented with recommendations. Other planned procedures were never built. The most pressing of these is the systematic output of a "Frequently Bought With Your Purchase" list for the customer at the point of sale.

Sixth, the date when a customer bought or rated a book should be incorporated into their recommendations on account of their change in taste. Perhaps the solution to this involves sorting the recommendations so that the top ones have a high similarity score and are relatively recent. Whatever the alterations may be, it is imperative that the code to populate the similarities table would be modified to always point to a new version of the table since the process takes a while to execute. In the time that it is running, the business will obviously still need recommendations, and thus the old version. Moreover, the procedures should only be switched to reference the new version once all the checks are in place.

Seventh, a procedure could be constructed which allows the filtering the collaborative-filtering-recommended books by one or multiple categories. This plan was outlined in the introduction, but it was never made into a reality. Similarly, it was planned that I use the fact that the authors, themes, and qualities were ranked by their predominance in each book, but, again, this never made it to the final product. Hopefully that is rectified by the time the system comes online. There is also a major worry about the efficacy of the whole content-filtering approach, and that is the quality of the labels. It would a priority that the business ensure these labels are representative. Otherwise, the system will be based on faulty data.

Eighth, it may be the case that as the company grows it will demand more sophisticated recommendation algorithms. Most of these –and especially regression, machine learning, and matrix factorization ones- will be better implemented in other software. And although the similarity scores currently take quite a short time to calculate, this is imply because only the top-selling books are sampled. This is obviously suboptimal. Ideally we do not want to only recommend popular books. Instead, all books should be considered. But this may well take too long in SQL, and thus the company may be forced to conduct its calculations outside the DBMS. The similarities table could simply be filled from the output of a recommendation system outside the database, and then the procedures to hand out the recommendations would already be in place, integrated into the DMS. However, the recommenders are quite rigid. If these are found not to scale or perform well, it may be awkward and slow to change. For this reason, implementing these mechanisms inside the DBMS may be risky. Perhaps other mechanisms could also be added a-priori in case the chosen ones underperform. Or perhaps the company may choose to export these tasks to another software entirely.

And finally, this adds yet another reason why the recommender systems should be tested. Simple recall and precision metrics could be calculated to assess respectively what proportion of books purchased are recommended (to that customer) and what proportion of recommended books are purchased. The company could adjust its weighting balance between purchase similarity and rating similarity. Extensive tests should be carried out before the company makes a decision one way or the other.

References

[1]: Mohamed Sarwat (2014)
[2]: https://www.computer.org/csdl/magazine/ic/2017/03/mic2017030012/13rRUB6SpQq
[3]: https://towardsdatascience.com/recommender-systems-in-practice-cef9033bb23a