**CS2102 - Database Systems**

**Project (Part 2)**

**CS2102 Team 56**

| Guo Ai | | A0177850W |
| --- | --- | --- |
| Leeroy Liu | | A0218491B |
| Qu Mingsi | | A0185084Y |
| Zhao Pengfei | | A0164637X |

**Table of Contents**

[**Responsibilities**](#_3225z3zdov9k) **3**

[**Triggers**](#_l10n647sxmfd) **4**

[**Routines**](#_nhrlakli3k2z) **8**

[Procedure](#_xzj5en6o2yd6) 8

[Functions](#_6xg6n6w1e94r) 10

[**Difficulties encountered & lessons learned**](#_rhgh2zbpyqrb) **13**

# Responsibilities

| **Name** | **Responsibilities** |
| --- | --- |
| Guo Ai | Trigger (1-6) Implementation, Review, Testing & Report Writing |
| Pengfei | Trigger (7-12) Implementation, Review, Testing & Report Writing |
| Leeroy | Procedure (1-3) Implementation, Review, Testing & Report Writing |
| Mingsi | Function (1-3) Implementation, Review, Testing & Report Writing |

# Triggers

We set all our triggers to be deferrable initially deferred because the insertion statements can be run in transactions and the triggers can be fired at the end of the entire transaction. Since deferred triggers only work with after triggers and row-level triggers, we set all our triggers to be after triggers. For after triggers, since the return value does not matter, we cannot use the return value to control whether the new tuple will be inserted or not. Therefore, when the trigger check fails, we will raise an exception inside the trigger so that the entire transaction can rollback and the tuple failing the trigger check will not be inserted into the database.

1. Name of trigger: shop\_at\_least\_one\_product

Basic Idea: The trigger executes the trigger function shop\_at\_least\_one\_product\_function() after every tuple is to be inserted into the shop table. From the sells table, we can use a SQL query to count the number of distinct products sold by the shop identified by new.id. If the number of distinct products sold by the shop is >= 1, we return new and the new tuple will be inserted into the shop table, otherwise, we raise an exception and the tuple will not be inserted into the shop table.

1. Name of trigger: order\_one\_or\_more\_product\_shop

Basic Idea: The trigger executes the trigger function order\_one\_or\_more\_product\_shop\_function() after every tuple is to be inserted into the orders table. From the orderline table, we can count the number of distinct products and distinct shops that the order identified by new.id involves. If both the number of distinct products and distinct shops are >= 1, then we return new and the new tuple will be inserted into the orders table. Otherwise, we raise an exception and the tuple will not be inserted into the orders table.

1. Name of trigger: coupon\_min\_order\_amount

Basic Idea: We retrieve the minimum order amount from the coupon\_batch table, and we may identify the minimum order amount for a particular coupon based on the coupon id; On the other hand, we may retrieve the total amount of an order from the orderline table left joining the sells table, which allows us to retrieve the price and quantity of the order of interest, as matched by the order id. As such, we may hence compare the total amount of an order with the minimum order amount of the coupon, and the trigger may hence raise an exception if the coupon id is not null and the total amount of the order is less than the minimum order amount of the coupon.

1. Name of trigger: check\_refund\_quantity

Basic Idea: The trigger executes the trigger function check\_refund\_quantity\_function() after every tuple is to be inserted into the refund\_request table. From the refund\_request table, we can get the total quantity for the orderline identified by new.order\_id, new.shop\_id, new.product\_id, and new.sell\_timestamp that have been requested for refund and not rejected (i.e., old\_refund\_quantity). Since this is an after trigger, the old\_refund\_quantity already contains the quantity of the new refund request tuple inserted. From the orderline table, we can get the quantity of the orderline identified by new.order\_id, new.shop\_id, new.product\_id, and new.sell\_timestamp (i.e., order\_quantity). If old\_refund\_quantity <= order\_quantity, then we return new and the new tuple will be inserted into the refund\_request table. Otherwise, we raise an exception and the tuple will not be inserted into the refund\_request table.

1. Name of trigger: refund\_within\_30\_days

Basic Idea: The trigger executes the trigger function refund\_within\_30\_days\_function() after every tuple is to be inserted into the refund\_request table. From the orderline table, we can get the delivery date of the order identified by the new.order\_id. From the refund\_request table, we can get the refund request date of the refund request identified by the new.id. If refund\_request\_date >= delivery\_date and refund\_request\_date <= delivery\_date + 30, then we return new and the new tuple will be inserted into the refund\_request table. Otherwise, we raise an exception and the tuple will not be inserted into the refund\_request table.

1. Name of trigger: check\_refund\_delivered\_product

Basic Idea: The trigger executes the trigger function check\_refund\_delivered\_product\_function() after every tuple is to be inserted into the refund\_request table. From the orderline table, we can get the status of the orderline identified by new.order\_id, new.product\_id, new.shop\_id, and new.sell\_timestamp. If the status of the product is ‘delivered’, then we return new and the new tuple will be inserted into the refund\_request table. Otherwise, we raise an exception and the tuple will not be inserted into the refund\_request table.

1. Name of trigger: check\_product\_review\_identity

Basic Idea: The idea is to trace back and retrieve the user id in two ways and compare the user id to check if they match. One of the user ids comes from the comment entity, which is the superclass in an ISA hierarchy with the review entity. From this side, we are hence identifying the user who is making the product review; The other user id comes from the orderline table left joining the orders table on the order id, which hence contains the user id as contributed from the orders table. The orderline table contributes the order id, shop id, product id as well as the sell timestamp, which we use to identify the product being purchased. From this side, we are hence identifying the user who purchased the product. If the two users are the same, then the user making the product review has been checked to be indeed the one that purchased the product. If not, then the trigger raises an exception.

1. Name of trigger: comment\_is\_review\_or\_reply

Basic Idea: We count the total number of appearances of the comment in the review table as well as the reply table, and if the total number is exactly equal to 1, then that means the comment is indeed one and only one of the 2 possibilities: review or reply. In other words, the comment is either a review or a reply, but not both. Otherwise, the trigger would raise an exception.

Additionally, to guard against the possibility of inserting a review or reply again after inserting them in the comment table as well as the respective review or reply table, we check the children's table (i.e. review’s or reply’s) as well and enforce that the review or the reply can only appear once in the comment table and cannot appear in the other child table (i.e. reply or review respectively). Otherwise, the trigger would raise an exception.

1. Name of trigger: reply\_has\_at\_least\_one\_reply\_version

Basic Idea: We examine the reply\_version table and count the number of entries that have matching reply id with the reply concerned. If there are indeed some entry or entries present, then that means the reply has indeed at least one reply version. Otherwise, the trigger would raise an exception.

1. Name of trigger: review\_has\_at\_least\_one\_review\_version

Basic Idea: We examine the review\_version table and count the number of entries that have matching reply id with the review concerned. If there are indeed some entry or entries present, then that means the review has indeed at least one review version. Otherwise, the trigger would raise an exception.

1. Name of trigger: check\_delivery\_complaint\_on\_delivered\_product

Basic Idea: Similar to (6), we utilise the orderline table to check on the product delivery status, by matching the order id, shop id, product id, as well as the sell timestamp to pinpoint the unique entry. If the product delivery status is not ‘delivered’, then making a delivery complaint would cause the trigger to raise an exception.

1. Name of trigger: complaint\_is\_delivery\_or\_shop\_or\_comment\_related

Basic Idea: We count the total number of appearances of the complaint in the shop\_complaint, comment\_complaint as well as the delivery\_complaint table, and if the total number is exactly equal to 1, then that means the complaint is indeed one and only one of the 3 possibilities: a shop-related complaint, a comment-related complaint or a delivery-related complaint. In other words, the complaint is either a delivery-related complaint, a shop-related complaint or a comment-related complaint (non-overlapping and covering). Otherwise, the trigger would raise an exception.

Additionally, to guard against the possibility of inserting a delivery-related complaint, a shop-related complaint or a comment-related complaint after inserting in the complaint table as well as the respective children's table (i.e. shop\_complaint’s, comment\_complaint’s or delivery\_complaint’s table), we check the children table (i.e. shop\_complaint, comment\_complaint or delivery\_complaint table) as well and enforce that the delivery-related complaint, the shop-related complaint or the comment-related complaint can only appear once in the complaint table and cannot appear in the other children tables. Otherwise, the trigger would raise an exception.

# Routines

## Procedure

**place\_order( user\_id INTEGER, coupon\_id INTEGER, shipping\_address TEXT, shop\_ids INTEGER[], product\_ids INTEGER[], sell\_timestamps TIMESTAMP[], quantities INTEGER[], shipping\_costs NUMERIC[] )**

Basic idea: There are 2 insertions required: order and orderline, and the new columns required are payment amount and order\_id.

Firstly, we check if the arrays are of the same length (input validation) and store the discount and min amount in 2 variables, if there is a coupon attached to the order.

Next, we calculated the payment amount in total, for all the order\_item. But before adding, we checked whether the product belongs to the shop and the item is in stock (input validation).

Then, the formula is used: **total\_amount = price \* quantity + shipping\_cost.**

Finally, the coupon is applied. (Trigger 3 is raised if coupon minimum amount is not reached)

The insertion is then executed. Firstly, the order is inserted using the total amount calculated. Then, the orderline items are inserted with the order\_id stored from the previous insertion of the order into the orders table.

**review( user\_id INTEGER, order\_id INTEGER, shop\_id INTEGER, product\_id INTEGER, sell\_timestamp TIMESTAMP, content TEXT, rating INTEGER, comment\_timestamp TIMESTAMP)**

Basic idea: inserts a review, which is a comment, into the comment, review and review\_version tables, storing the comment\_id and review\_id to be used as reference. Insertion into the comment table only requires user\_id provided, while the review table requires comment\_id and other parameters provided. Finally, insertion into review\_version requires review\_id and other parameters provided. These 3 tables have to be inserted together, failing which would result in constraint trigger 8 or 10 to raise an exception. The user has to be the one who purchased the product, else the trigger 7 will be activated.

**reply( user\_id INTEGER, other\_comment\_id INTEGER, content TEXT, reply\_timestamp TIMESTAMP )**

Basic idea: inserts a reply, which is also a comment, into the comment, reply and reply\_version tables. Similar to the review procedure above, except that other\_comment\_id is used as it is replying to another comment, unlike review which is reviewing a product order. Similarly, this is connected to the trigger 8 and 9, that reply must have at least one reply version.

## Functions

1. View\_comments( shop\_id\_input INTEGER, product\_id\_input INTEGER, sell\_timestamp\_input TIMESTAMP )

Note: We changed the input variable name because in the orderline table, there is an attribute named “shop\_id”.

Basic idea:

Firstly, we collect the ids of comment related to the given (shop\_id\_input, product\_id\_input, sell\_timestamp\_input) recursively where the base case is the ids of direct review to the given (shop\_id\_input, product\_id\_input, sell\_timestamp\_input), and the inductive step is selecting ids of reply which replies to the existing comments, including reviews and replies. In this way, we can get the entire reply chain including reviews, replies to reviews, and replies to replies.

Then we select the corresponding (user\_id, comment\_id, content, rating (set null if the type of comment is reply), comment\_timestamp) from the latest review and reply versions which are obtained from the review versions and reply versions by comparing the timestamp of rows with the same review\_id (or reply\_id) and selecting the row with the latest timestamp.

Lastly, we obtain the output table by joining the above table and the ‘users’ table to get corresponding usernames, and in case ‘account\_closed’ was set to be True for some user, the username is replaced by ‘A Deleted User’.

1. get\_most\_returned\_products\_from\_manufacturer ( manufacturer\_id INTEGER, n INTEGER)

Assumption: Since the question defines the denominator of return\_rate as “sum of the quantity delivered across all orders”, we only include the ordered products for which the orderline status is ‘delivered’. Ordered products with orderline status ‘being\_processed’ and ‘shipped’ are not included in the quantity delivered.

Basic idea:

Firstly, we get the ids and names of products that are manufactured by the given ‘manufacturer\_id’ and for each, the number of products that are successfully delivered to the customer, from the ‘orderline’ table by filtering rows that satisfy the conditions of status being ‘delivered’ and manufacturer matching input, and then grouping the rows on product\_id and selecting sum of quantity.

Then we get the ids of the product that is manufactured by the given ‘manufacturer\_id’ and for each, the number of product that is successfully returned and refunded, from the ‘refund\_request’ table similarly.

Lastly, we obtain the output table by selecting product\_id and the ratio of ‘the number of products successfully returned and refunded’ to ‘the number of products delivered’ as return\_rate from the join result of the above two tables, where the return\_rate is set to be 0.00 if there does not exist product that is delivered or there does not exist product that is successfully refunded. Getting N products with the highest return\_rate is achieved by ordering the table by return\_rate descending and limiting N rows. If the return\_rate ties, we order the output tuples in ascending order of product\_id.

1. get\_worst\_shops( n INTEGER )

Basic idea:

Firstly, we count the number of refund requests (for distinct orderlines), number of shop complaints, number of delivery complaints (for distinct orderlines) and number of one-star reviews respectively from ‘refund\_request’ table, ‘shop\_complaint’ table, ‘delivery complaint’ table, and ‘review\_version’ table. The latest review versions are obtained from the review\_version table by comparing the timestamp of rows with the same review\_id and selecting the row with the latest timestamp.

Then we join the above tables one by one to get the shop\_ids and the number of negative indicators of each type, where the corresponding number is set to be 0 in case the shop\_id does not exist in any of the above tables (which means it does not have negative indicators in that type).

Lastly, we obtain the output table by joining the above table and the ‘shop’ table to get the shop name, and selecting the sum of four negative indicator counts. Getting the worst N shops is achieved by ordering the table by the sum of counts descendingly and limiting N rows. If the num\_negative\_indicators ties, we order the output tuples in ascending order of shop\_id.

# Difficulties encountered & lessons learned

| **Difficulties encountered** | **How We Solved the Difficulties/Lessons learned** |
| --- | --- |
| In implementing the procedure, we realised that we had to defer the trigger requirement checking. However, deferred triggers only worked with AFTER, not BEFORE.  However, for AFTER triggers, the return value does not matter. | Use triggers AFTER insert instead of BEFORE insert.  We raise an exception in the trigger if the tuple inserted violates the constraint, which will cause the entire transaction to rollback. |
| When implementing the procedures and the triggers separately, as we split the workload initially, we did not realise that they were interconnected, hence some exceptions were raised when the whole code was pieced together that has to be resolved. | We should communicate with one another and update on what each of us is doing, and also take time to read what others are doing in detail to prevent miscommunication. |
| When implementing the procedure place\_order, we found that a few assumptions had to be made, such as that the order is placed within the validity period of the coupon, if a coupon is used. Although input validation is not needed, since there are some updating required such as quantity in sells table, we decided to add in some validity checks. | Clarify and read the question carefully, and it would be better to be safe and check for valid inputs. |
| It was unclear what sell\_timestamp would refer to, and there was some confusion on whether it is required to check whether the order of the timestamp was within the validity period of coupon, and we thought sell\_timestamp refers to that order timestamp. However, upon further inspection, we realised that it refers to the product timestamp in the particular shop, as the shop could have multiple rows of the same product but listed at different timing. | Look at the ER diagram for clues on unfamiliar terms or attributes(columns) we are unsure of, and check which primary key it belongs to, to get a better idea of its uniqueness. |
| When testing, a lot of syntax error revolves around trying to call the procedure or function with a query as a parameter, which was not possible as subqueries are not allowed. Declaration of variables globally was also not allowed, hence we have to put them inside functions or procedures in order to store the values returned by the query inside variables declared. | In Postgresql, only declare variables inside functions and procedures, and executed tests as functions or procedures to increase the modularity of code. |
| A lot of compilation errors that we get revolve around ambiguous namings, as we often use names similar to that of column names inside our functions or procedures.  This results in ambiguity as to which exceptions have been raised during the debugging process. | Use different local variables or have a naming convention for them e.g. having an underscore as the prefix for local variables to differentiate from column variables. |
| During testing, there were a lot of unexpected bugs because we did not consider carefully all possible cases that may occur. | Modify the code carefully to cover all possible situations, especially for boundary values. |
| For the get\_most\_returned\_products\_from\_manufacturer() function, we need to retrieve the entire reply chain (i.e., reviews, replies to reviews, replies to replies). | We learned how to use recursive queries in SQL and used a recursive query to get the full reply chain. |
| For the get\_most\_returned\_products\_from\_manufacturer() function, there are quite a lot of edge cases, for example, when the product has never been ordered, when there is no refund request about the product, etc. We also need to avoid integer division because the return\_rate is of numeric type and we do not want the decimal places to be truncated. | We used coalesce(value, 0) where appropriate to convert a null value to 0 to deal with the cases when the product was never delivered or returned. We also converted the sum of quantity returned and the sum of the quantity delivered to numeric before the division. |