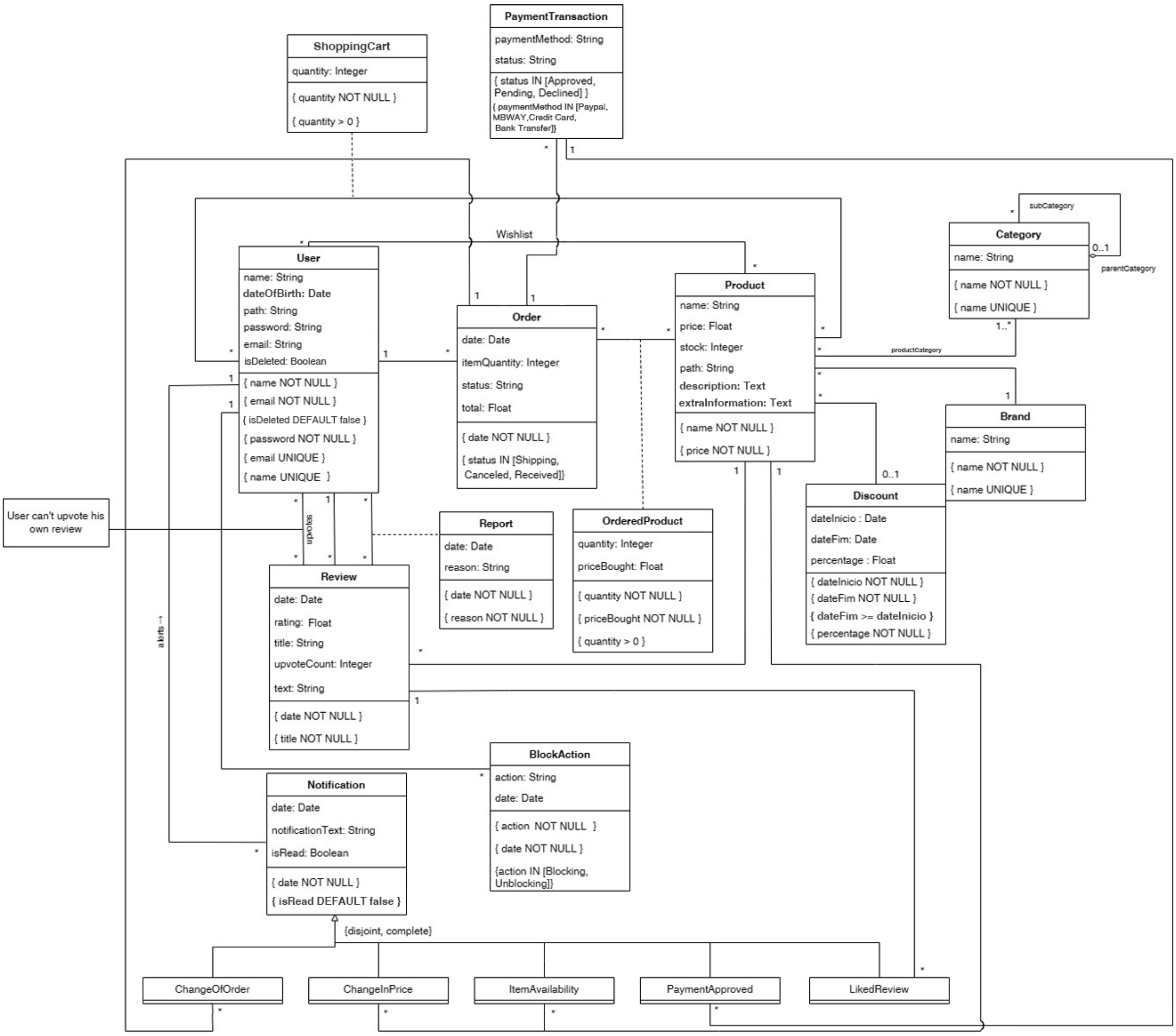
A4: Conceptual Data Model

This section contains the description of Cappuccino's entities and relationships, alongside its database specification.

1. Class diagram

The UML diagram below represents the main classes present in Cappuccino, alongside their attributes, associations with their respective multiplicity and constraints.



2. Additional Business Rules

Identifier	Description
BR08	Customers can only upvote a review once
BR09	A blocked customer cannot see any reviews nor make any more reviews
BR10	Customers can only report a review once

Identifier	Description
BR11	Customers can't report their own review

A5: Relational Schema, validation and schema refinement

1. Relational Schema

This section contains the relational schema derived from the conceptual data model. It shows the attributes, domains, primary and foreign keys from each class, alongside necessary constraits such as *unique*, *default*, *not null* and *check*.

Relation Reference	Relation Compact Notation
R01	user(id, name **NN**, path, dateOfBirth, password **NN**, email **UK NN**, isDeleted **DF** false)
R02	order(id, date **NN**, userId → User, itemQuantity, status **IN** order_status, total)
R03	review(id, userId → User, productId → Product, date **NN**, rating, title **NN**, upvoteCount, text)
R04	notification(id, userId → User, date **NN**, notificationText, isRead **DF** false)
R05	block_action(id, userId → User, action **NN**, date **NN**)
R06	product(id, name **NN**, price **NN**, stock, description, extraInformation, brandId → Brand, discountId → Discount)
R07	ordered_product(productId → Product, orderId → Order, quantity **NN CK** quantity > 0, priceBought **NN**)
R08	payment_transaction(id, orderID → Order, paymentMethod **IN** payment_method, status)
R09	brand(id, name **UK NN**)
R10	discount(id, dateInicio **NN**, dateFim **NN**, percentage **NN**, **CK** dateFim >= dateInicio)
R11	category(id, name **UK NN**, parent_categoryID → Category)
R12	change_of_order(id, orderId → Order, notificationId → Notification)
R13	change_on_price(id, notificationId → Notification, productId → Product)
R14	item_availability(id, notificationId -> Notification, productId -> Product)
R15	payment_approved(id, orderId → Order, notificationId -> Notification, paymentTransactionID → paymentTransaction)
R16	liked_review(id, reviewId → Review, notificationId -> Notification)
R17	wishlist(userId → User, productId → Product)
R18	shopping_cart(userId → User, productId → Product, quantity **NN CK** quantity > 0)
R19	upvotes(userId → User, reviewId -> Review)

Relation Reference	Relation Compact Notation
R20	report(userId → User, reviewId -> Review, date **NN**, reason **NN**)
R21	product_category(productId → Product, categoryId → Category)

Legend:

- **UK** = UNIQUE KEY
- **NN** = NOT NULL
- **DF** = DEFAULT
- **CK** = CHECK

2. Domains

Specification of additional domains.

Domain Name	Domain Specification
order_status	ENUM (‘Shipping’, ‘Payment Approved’, ‘Canceled’, ‘Received’)
blocked_status	ENUM (‘Blocking’, ‘Unblocking’)
payment_status	ENUM ('Approved', 'Pending', 'Declined')
payment_method	ENUM ('Paypal', 'MBWAY', 'Credit Card', 'Bank Transfer')

3. Schema Validation

This section contains the relational schema obtained from the conceptual data model, validated through the identification of all functional dependencies (FDs) and the normalization of each relation schema.

TABLE R01	**user**
Keys	{ id }
Functional Dependencies:	
FD0101	id → {name, path, dateOfBirth, password, isDeleted}
FD0102	email → {id, name, path, dateOfBirth, password, isDeleted}
NORMAL FORM	BCNF

TABLE R02	**order**
Keys	{ id }
Functional Dependencies:	
FD0201	id → {date, userId, itemQuantity, status, total}
FD0202	userId → {id, name, email}
NORMAL FORM	BCNF

TABLE R03	**review**
Keys	{ id }
Functional Dependencies:	
FD0301	id → {userId, productId, date, rating, title, upvoteCount, text}
FD0302	userId → {id, name, email}
FD0303	productId → {id, name}
NORMAL FORM	BCNF

TABLE R04	**notification**
Keys	{ id }
Functional Dependencies:	
FD0401	id → {userId, date, notificationText, isRead}
FD0402	userId → {id, name, email}
NORMAL FORM	BCNF

TABLE R05	**block_action**
Keys	{ id }
Functional Dependencies:	
FD0501	id → {userId, action, date}
FD0502	userId → {id, name, email}
NORMAL FORM	BCNF

TABLE R06	**product**
Keys	{ id }
Functional Dependencies:	
FD0601	id → {name, price, stock, brandId, discountId, description, extraInformation}
FD0602	brandId → {id, name}
FD0603	discountId → {id, dateInicio, dateFim, percentage}
NORMAL FORM	BCNF

TABLE R07	**ordered_product**
Keys	{ productId, orderId }
Functional Dependencies:	
FD0701	{productId, orderId} → {quantity, priceBought}
NORMAL FORM	BCNF

TABLE R08	**payment_transaction**
Keys	{ id }
Functional Dependencies:	
FD0801	id → {orderID, paymentMethod, status}

NORMAL FORM BCNF

TABLE R09	**brand**
Keys	{ id }
Functional Dependencies:	
FD0901	id → {name}

NORMAL FORM BCNF

TABLE R10	**discount**
Keys	{ id }
Functional Dependencies:	
FD1001	id → {dateInicio, dateFim, percentage}

NORMAL FORM BCNF

TABLE R11	**category**
Keys	{ id }
Functional Dependencies:	
FD1101	id → {name, parent_categoryID}

NORMAL FORM BCNF

TABLE R12	**change_of_order**
Keys	{ id }
Functional Dependencies:	
FD1201	id → {orderId, notificationId}

NORMAL FORM BCNF

TABLE R13	**change_in_price**
Keys	{ id }
Functional Dependencies:	
FD1301	id → {notificationId, productId}

NORMAL FORM BCNF

TABLE R14	**item_availability**

TABLE R14	**item_availability**
Keys	{ id }
Functional Dependencies:	
FD1401	id → {notificationId, productId}

NORMAL FORM BCNF

TABLE R15	**payment_approved**
Keys	{ id }
Functional Dependencies:	
FD1501	id → {orderId, notificationId, paymentTransactionID}

NORMAL FORM BCNF

TABLE R16	**liked_review**
Keys	{ id }
Functional Dependencies:	
FD1601	id → {reviewId, notificationId}

NORMAL FORM BCNF

TABLE R17	**wishlist**
Keys	{ userId, productId }
Functional Dependencies:	
FD1701	{userId, productId} → {}

NORMAL FORM BCNF

TABLE R18	**shopping_cart**
Keys	{ userId, productId }
Functional Dependencies:	
FD1801	{userId, productId} → {quantity}

NORMAL FORM BCNF

TABLE R19	**upvotes**
Keys	{ userId, reviewId }
Functional Dependencies:	
FD1901	{userId, reviewId} → {}

NORMAL FORM BCNF

TABLE R20	**report**

TABLE R20	**report**
Keys	{ userId, reviewId }
Functional Dependencies:	
FD2001	{userId, reviewId} → {date, reason}
NORMAL FORM	BCNF

TABLE R21	**product_category**
Keys	{ productId, categoryId }
Functional Dependencies:	
FD2001	{productId, categoryId} → {}
NORMAL FORM	BCNF

Since every relation is in the Boyce-Codd Normal Form (BCNF), the relational schema itself is also in the BCNF (no need for further normalization).

A6: Indexes, triggers, transactions and database population

This artifact contains the SQL code (both the database and its population), alongside the indexes, triggers and transactions.

1. Database Workload

The following table outlines the workload for various database relations. It provides insights into the scale and growth estimates for each relation, offering an understanding of the data volume and usage patterns.

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	users	10 k	10 / day
R02	orders	1 k	10 / day
R03	review	100	1 / day
R04	notification	10 k	100 / day
R05	blockAction	10	no growth
R06	product	100	1 / month
R07	orderedProduct	10 k	100 / day
R08	paymentTransaction	1 k	10 / day
R09	brand	100	1 / month
R010	discount	10	10 / month
R011	category	100	no growth

Relation reference	Relation Name	Order of magnitude	Estimated growth
R012	changeOfOrder	1 k	10 / day
R013	changeInPrice	100 k	10 / day
R014	itemAvailability	1 k	1 / day
R015	paymentApproved	1 k	10 / day
R016	likedReview	100	1 / day
R017	wishlist	100	10 / day
R018	shoppingCart	100	10 / day
R019	upvotes	100	1 / day
R020	report	10	1 / month

2. Proposed Indices

2.1. Performance Indices

The following indexes have been designed to enhance query performance and data retrieval speed for various tables. While these indexes optimize SELECT queries, it's important to note that there might be a trade-off with INSERT, UPDATE, and DELETE operations.

Index	**IDX01**
Relation	Notification
Attribute	userID
Type	B-tree
Cardinality	Medium
Clustering	No
Justification	Given the large size of the Notification table, this index is essential to efficiently retrieve notifications for a specific user. It allows for quick access to user-specific notifications without scanning the entire table, which becomes increasingly important as the table size grows.

SQL CODE

```
CREATE INDEX notification_user_id_idx ON "notifications" (userId);
```

Index	**IDX02**

Index	**IDX02**
Relation	Product
Attribute	price
Type	B-tree
Cardinality	Medium
Clustering	No
Justification	This index can be used to efficiently retrieve and filter products based on their price. Searches that have price as a filter, searches using a range of prices, and ordering by price will be common in our website.

SQL CODE

```sql
CREATE INDEX product_price_idx ON "product" (price);
```

Index	**IDX03**
Relation	Review
Attribute	productId
Type	B-tree
Cardinality	Medium
Clustering	No
Justification	This index will improve the performance of queries that retrieve reviews for a specific product. Users often seek reviews for a particular product, and this index will make these searches more efficient.

SQL CODE

```sql
CREATE INDEX review_product_id_idx ON "review" (productId);
```

2.2. Full-text Search Indices

The following FTS indexes have been made to give users advanced search capabilities, allowing them to discover products based on matching attributes such as product names, brand names, categories, and additional information. This enables rapid and accurate search results across multiple data dimensions.

Index	**IDX04**

Index	**IDX04**
Relation	Product
Attribute	name, extraInformation, description, brand.name
Type	GIN
Clustering	No
Justification	To provide full-text search features for products, enabling users to search based on matching product names, brand names and extra information. The GIN index type is chosen for efficiency and performance. The indexed fields are not expected to change frequently.

```sql
ALTER TABLE product
ADD COLUMN product_tsv TSVECTOR;

CREATE OR REPLACE FUNCTION product_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.product_tsv = setweight(to_tsvector('english', NEW.productName), 'A') ||
                          setweight(to_tsvector('english', (SELECT brandName FROM brand WHERE id = NEW.brandId)), 'B') ||
                          setweight(to_tsvector('english', NEW.extraInformation), 'D');
    END IF;

    IF TG_OP = 'UPDATE' THEN
        IF (NEW.productName <> OLD.productName OR NEW.brandId <> OLD.brandId OR NEW.extraInformation <> OLD.extraInformation) THEN
            NEW.product_tsv = setweight(to_tsvector('english', NEW.productName), 'A') ||
                              setweight(to_tsvector('english', (SELECT brandName FROM brand WHERE id = NEW.brandId)), 'B') ||
                              setweight(to_tsvector('english', NEW.extraInformation), 'D');
        END IF;
    END IF;

    RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER product_search_update
BEFORE INSERT OR UPDATE ON product
FOR EACH ROW
EXECUTE PROCEDURE product_search_update();

CREATE INDEX product_search_idx ON product USING GIN (product_tsv);
```

Index	**IDX05**
Relation	productCategory
Attribute	product.name, product.extraInformation, product.description, name, brand.name
Type	GIN
Clustering	No
Justification	We also needed to update the tsv for when the product category got changed so we opted to do this FTS index to enable the user to search for categories and get the related products.

```sql
CREATE OR REPLACE FUNCTION product_category_search_update() RETURNS TRIGGER AS $$
BEGIN
  IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
    UPDATE Product
    SET product_tsv = (
      setweight(to_tsvector('english', P.productName), 'A') ||
      setweight(to_tsvector('english', B.brandName), 'B') ||
      setweight(to_tsvector('english', C.categoryName), 'C') ||
      setweight(to_tsvector('english', P.extraInformation), 'D')
    )
    FROM product P
    JOIN brand B ON P.brandId = B.id
    JOIN productCategory PC ON P.id = PC.productId
    JOIN category C ON PC.categoryId = C.id
    WHERE PC.productId = NEW.productId;
  END IF;

  RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER product_category_search_update
AFTER INSERT OR UPDATE ON productCategory
FOR EACH ROW
EXECUTE PROCEDURE product_category_search_update();

CREATE INDEX product_category_search_idx ON product USING GIN (product_tsv);
```

3. Triggers

Triggers are automated actions that respond to specific events in our database. In this section, we present a concise overview of the triggers used to enhance data consistency and streamline essential processes.

Trigger	**TRIGGER01**
Description	Add notification based on an item availability

SQL CODE

```
CREATE OR REPLACE FUNCTION add_notificationAvailability() RETURNS TRIGGER AS
$BODY$
DECLARE notificationId integer;
BEGIN
  IF OLD.stock = 0 AND NEW.stock > 0 THEN
        INSERT INTO notifications (notificationDate, notificationText, userId,
isRead) SELECT NOW(), 'There is stock available right now', userId, false FROM
Wishlist WHERE Wishlist.productId = NEW.id RETURNING id INTO notificationId;
        INSERT INTO itemAvailability (notificationId, productId) VALUES
(notificationId, NEW.id);
  ELSIF NEW.stock = 1 THEN
        INSERT INTO notifications (notificationDate, notificationText, userId,
isRead) SELECT NOW(), 'LAST ITEM AVAILABLE', userId, false FROM Wishlist WHERE
Wishlist.productId = NEW.id RETURNING id INTO notificationId;
        INSERT INTO itemAvailability (notificationId, productId) VALUES
(notificationId, NEW.id);
  END IF;
  RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER notificationAvailability
AFTER UPDATE ON product
FOR EACH ROW
EXECUTE PROCEDURE add_notificationAvailability();
```

Trigger	TRIGGER02
Description	Add notification based on a like on a review

SQL CODE

```
CREATE OR REPLACE FUNCTION add_notificationLike() RETURNS TRIGGER AS $BODY$
DECLARE notificationId integer;
BEGIN
  IF OLD.upvoteCount < NEW.upvoteCount THEN
    INSERT INTO notifications (notificationDate, notificationText, userId, isRead)
VALUES (NOW(),'Someone Liked Your Review', NEW.userId, false) RETURNING id INTO
notificationId;
    INSERT INTO likedReview (notificationId, reviewId) VALUES (notificationId,
NEW.id);
  END IF;
  RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;
```

```
CREATE TRIGGER notificationLike
AFTER UPDATE ON review
FOR EACH ROW
EXECUTE PROCEDURE add_notificationLike();

update review set upvoteCount = 11 where review.id = 1;
```

Trigger	TRIGGER03
Description	Add notification based on a change on a price of a product in wishlist

SQL CODE

```
CREATE OR REPLACE FUNCTION add_notificationPrice() RETURNS TRIGGER AS $BODY$
DECLARE notificationId integer;
BEGIN
  IF OLD.price < NEW.price THEN
     INSERT INTO notifications (notificationDate, notificationText, userId,
isRead) SELECT NOW(), 'The price is higher on ' || NEW.productName, userId, false
FROM Wishlist WHERE Wishlist.productId = NEW.id RETURNING id INTO notificationId;
     INSERT INTO changeInPrice (notificationId, productId) VALUES (notificationId,
NEW.id);
  ELSIF OLD.price > NEW.price THEN
     INSERT INTO notifications (notificationDate, notificationText, userId,
isRead) SELECT NOW(), 'The price is lower on ' || NEW.productName, userId, false
FROM Wishlist WHERE Wishlist.productId = NEW.id RETURNING id INTO notificationId;
     INSERT INTO changeInPrice (notificationId, productId) VALUES (notificationId,
NEW.id);
  END IF;
  RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER notificationPrice
AFTER UPDATE ON product
FOR EACH ROW
EXECUTE PROCEDURE add_notificationPrice();
```

Trigger	TRIGGER04
Description	Add notification based on the status of a order

SQL CODE

```sql
CREATE OR REPLACE FUNCTION add_notificationOrder() RETURNS TRIGGER AS $BODY$
DECLARE notificationId integer;
BEGIN
    INSERT INTO notifications (notificationDate, notificationText, userId, isRead)
VALUES (NOW(), 'Order Status: ' || NEW.orderStatus, NEW.userId, false) RETURNING
id INTO notificationId;
    INSERT INTO changeOfOrder (notificationId, orderId) VALUES (notificationId,
NEW.id);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER notificationOrder
AFTER INSERT OR UPDATE ON orders
FOR EACH ROW
EXECUTE PROCEDURE add_notificationOrder();
```

Trigger	**TRIGGER05**
Description	Add notification based on payment status

SQL CODE

```sql
CREATE OR REPLACE FUNCTION add_notificationPayment() RETURNS TRIGGER AS $BODY$
DECLARE notificationId integer;
BEGIN
    INSERT INTO notifications (notificationDate, notificationText, userId, isRead)
SELECT NOW(), 'Payment Status: ' || NEW.paymentStatus, userId, false FROM orders
WHERE orders.id = NEW.orderId RETURNING id INTO notificationId;
    INSERT INTO paymentApproved (notificationId, paymentTransactionId) VALUES
(notificationId, NEW.id);
    RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER notificationPayment
AFTER INSERT OR UPDATE ON paymentTransaction
FOR EACH ROW
EXECUTE PROCEDURE add_notificationPayment();
```

Trigger	**TRIGGER06**
Description	Verify if is voting its own review (business rule BR07)

SQL CODE

```
CREATE OR REPLACE FUNCTION verify_vote() RETURNS TRIGGER AS $BODY$
BEGIN
  IF EXISTS (SELECT * FROM review WHERE review.userID = NEW.userID AND review.id =
NEW.reviewId) THEN
     RAISE EXCEPTION 'A user cant vote in his own review';
  END IF;
  RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER verify_vote
BEFORE INSERT ON upvote
FOR EACH ROW
EXECUTE PROCEDURE verify_vote();
```

Trigger	**TRIGGER07**
Description	Verify if a user has already reviewed that product (business rule BR04)

SQL CODE

```
CREATE OR REPLACE FUNCTION verify_review() RETURNS TRIGGER AS $BODY$
BEGIN
  IF EXISTS (SELECT * FROM review WHERE review.userId = NEW.userId AND
review.productId = NEW.productId) THEN
     RAISE EXCEPTION 'This user already reviewed this product';
  END IF;
  RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER verify_review
BEFORE INSERT ON review
FOR EACH ROW
EXECUTE PROCEDURE verify_review();
```

Trigger	**TRIGGER08**
Description	Verify if a order still can be canceled (business rule BR02)

SQL CODE

```
CREATE OR REPLACE FUNCTION verify_order_delete() RETURNS TRIGGER AS $BODY$
BEGIN
```

```
  IF (SELECT orderStatus FROM orders WHERE orders.id = NEW.id) = 'Shipping' AND NEW.orderStatus = 'Canceled' THEN
     RAISE EXCEPTION 'This order is already on its way';
  ELSIF (SELECT orderStatus FROM orders WHERE orders.id = NEW.id) = 'Received' AND NEW.orderStatus = 'Canceled' THEN
     RAISE EXCEPTION 'This order is already done';
  ELSIF (SELECT orderStatus FROM orders WHERE orders.id = NEW.id) = 'Canceled' THEN
     RAISE EXCEPTION 'This order is already canceled';
  END IF;
  RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER verify_order_delete
BEFORE UPDATE ON orders
FOR EACH ROW
EXECUTE PROCEDURE verify_order_delete();
```

Trigger	**TRIGGER09**
Description	Deleted read notification

SQL CODE

```
CREATE OR REPLACE FUNCTION delete_notification_read() RETURNS TRIGGER AS $BODY$
BEGIN
   DELETE FROM notifications WHERE notifications.isRead = true;
   RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER delete_notification_read
AFTER UPDATE ON notifications
FOR EACH ROW
EXECUTE PROCEDURE delete_notification_read();
```

Trigger	**TRIGGER10**
Description	Verify if the time of a new review its after the current time (business rule BR06)

SQL CODE

```
CREATE OR REPLACE FUNCTION verify_time() RETURNS TRIGGER AS $BODY$
BEGIN
   IF NEW.reviewDate > Now() THEN
```

```
     RAISE EXCEPTION 'Problems with date time';
   END IF;
   RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER verify_time
BEFORE INSERT ON review
FOR EACH ROW
EXECUTE PROCEDURE verify_time();
```

Trigger	**TRIGGER11**
Description	Verify the age of the user for alcoholic products and tobacco (business rule BR05)

SQL CODE

```
CREATE OR REPLACE FUNCTION verify_age() RETURNS TRIGGER AS $BODY$
BEGIN
   IF COALESCE((SELECT AGE(NOW(), dateofbirth) FROM users WHERE users.id =
NEW.userId), '0 years') < interval '18 years' AND ((SELECT categoryName FROM
category WHERE category.id = (SELECT categoryId FROM productCategory WHERE
productCategory.productId = NEW.productId)) = 'Tobacco' OR (SELECT categoryName
FROM category WHERE category.id = (SELECT categoryId FROM productCategory WHERE
productCategory.productId = NEW.productId)) = 'Alcohol') THEN
     RAISE EXCEPTION 'You need to be over 18';
   END IF;
   RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER verify_age
BEFORE INSERT ON shoppingCart
FOR EACH ROW
EXECUTE PROCEDURE verify_age();
```

4. Transactions

Transactions are essential database operations that ensure data integrity. This section offers a brief insight into the transactions used to maintain consistent and reliable data within the database.

Transaction	**TRAN01**
Description	Inserting item in shopping cart

Transaction	**TRAN01**
Justification	In order to maintain consistency, it's necessary to use a transaction that everything occurs smoothly and without errors, otherwise a ROLLBACK is issued. The isolation level is Read Commited, since updates to the item availabity can happen by other commited transaction, would result in inconsisting data being stored.
Isolation level	READ COMMITED

SQL CODE

```
DO $$
DECLARE
    stock_available integer;
BEGIN
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
    SELECT stock INTO stock_available
    FROM product
    WHERE id = $product_id
    FOR UPDATE;

    IF stock_available >= 1 THEN
        INSERT INTO shoppingCart (userId, productId, quantity)
        VALUES ($user_id, $product_id, 1);

        UPDATE product
        SET stock = stock_available - 1
        WHERE id = $product_id;

        COMMIT;
    ELSE
        ROLLBACK;
    END IF;

END $$;
```

Transaction	**TRAN02**
Description	Customer deleting account
Justification	The isolation level is serializabe since it may be new rows in the notification table related to the customer that could result in inconsisting data being stored.
Isolation level	SERIALIZABLE

SQL CODE

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

UPDATE users
SET username = 'none',
    userPassword = 'none',
    userPath = 'img/default.png',
    email = $user_id,
    isDeleted = true
WHERE id = $user_id;

DELETE FROM notifications
WHERE notifications.userId = $user_id;

COMMIT;
END TRANSACTION;
```

Transaction	**TRAN03**
Description	Moving shopping cart items to order
Justification	The isolation level is Repeatable Read, because, otherwise, changes to the shopping cart, or the price of a specific item would lead to a inconsistent data storage.
Isolation level	REPEATABLE READ

SQL CODE

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN;
DO $$

DECLARE torder INTEGER;
    BEGIN
    INSERT INTO orders(orderDate,userId)
    VALUES (now(), $user_id)
    RETURNING id INTO torder;

    INSERT INTO orderedProduct(orderId, productId, quantity, priceBought)
    SELECT torder, shoppingCart.productId, shoppingCart.quantity, product.price
    FROM shoppingCart
    INNER JOIN product ON shoppingCart.productId = product.id
    WHERE shoppingCart.userId = $user_id;

    UPDATE orders
    SET itemQuantity = (SELECT SUM(orderedProduct.quantity) FROM orderedProduct
WHERE orderedProduct.orderId = id),
        total = (SELECT SUM(orderedProduct.quantity*orderedProduct.priceBought)
FROM orderedProduct WHERE orderedProduct.orderId = id)
    WHERE userId = $user_id;

    DELETE FROM shoppingCart
    WHERE shoppingCart.userId = $user_id;
END $$;
COMMIT;
```

Transaction	TRAN04
Description	Get the 20 first items from a specific category
Justification	The isolation level is read-only, because their may be new items regarding that category added to the table.
Isolation level	READ ONLY

SQL CODE

```
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY;

SELECT product.productName, category.categoryName
FROM product
INNER JOIN productCategory ON productCategory.productId = product.id
INNER JOIN category ON category.id = productCategory.categoryId
LIMIT 20;

COMMIT;

END TRANSACTION;
```

Annex A. SQL Code

A.1. Database Schema

```
CREATE SCHEMA IF NOT EXISTS lbaw2345;


DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS orders CASCADE;
DROP TABLE IF EXISTS product CASCADE;
DROP TABLE IF EXISTS brand CASCADE;
DROP TABLE IF EXISTS category CASCADE;
DROP TABLE IF EXISTS discount CASCADE;
DROP TABLE IF EXISTS orderedProduct CASCADE;
DROP TABLE IF EXISTS review CASCADE;
DROP TABLE IF EXISTS notifications CASCADE;
DROP TABLE IF EXISTS changeOfOrder CASCADE;
DROP TABLE IF EXISTS changeInPrice CASCADE;
DROP TABLE IF EXISTS itemAvailability CASCADE;
DROP TABLE IF EXISTS productCategory CASCADE;
DROP TABLE IF EXISTS paymentApproved CASCADE;
DROP TABLE IF EXISTS likedReview CASCADE;
DROP TABLE IF EXISTS paymentTransaction CASCADE;
DROP TABLE IF EXISTS report CASCADE;
DROP TABLE IF EXISTS blockAction CASCADE;
DROP TABLE IF EXISTS wishlist CASCADE;
DROP TABLE IF EXISTS shoppingCart CASCADE;
DROP TABLE IF EXISTS upvote CASCADE;

DROP TYPE IF EXISTS order_status;
DROP TYPE IF EXISTS blocked_status;
DROP TYPE IF EXISTS payment_status;
DROP TYPE IF EXISTS payment_method;

CREATE TYPE order_status AS ENUM ('Shipping', 'Payment Approved', 'Canceled', 'Received');
CREATE TYPE blocked_status AS ENUM ('Blocking', 'Unblocking');
CREATE TYPE payment_status AS ENUM ('Approved', 'Pending', 'Declined');
CREATE TYPE payment_method AS ENUM ('Paypal', 'MBWAY', 'Credit Card', 'Bank Transfer');

CREATE TABLE users (
   id SERIAL PRIMARY KEY,
   dateofbirth DATE,
   username VARCHAR(256) UNIQUE NOT NULL,
   userPath VARCHAR(256),
   userPassword VARCHAR(256) NOT NULL,
   email VARCHAR(256) UNIQUE NOT NULL,
   isDeleted BOOL DEFAULT false NOT NULL
);

CREATE TABLE orders (
   id SERIAL PRIMARY KEY,
   orderDate DATE NOT NULL,
   itemQuantity INTEGER,
   orderStatus order_status,
   total REAL,
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE paymentTransaction (
   id SERIAL PRIMARY KEY,
   method payment_method,
   paymentStatus payment_status,
   orderId INTEGER NOT NULL REFERENCES orders (id) ON UPDATE CASCADE ON DELETE
CASCADE
);

CREATE TABLE brand (
   id SERIAL PRIMARY KEY,
   brandName VARCHAR(256) UNIQUE NOT NULL
);

CREATE TABLE category (
   id SERIAL PRIMARY KEY,
   categoryName VARCHAR(256) UNIQUE NOT NULL,
   parentCategoryId INTEGER REFERENCES category(id) ON UPDATE CASCADE ON DELETE
CASCADE
);

CREATE TABLE discount (
   id SERIAL PRIMARY KEY,
   startDate DATE NOT NULL,
   endDate DATE NOT NULL,
   percentage REAL NOT NULL
);

CREATE TABLE product (
   id SERIAL PRIMARY KEY,
   productName VARCHAR(256) NOT NULL,
   description TEXT,
   extraInformation TEXT,
   price REAL NOT NULL,
   productPath VARCHAR(256),
   stock INTEGER,
   brandId INTEGER NOT NULL REFERENCES brand (id) ON UPDATE CASCADE ON DELETE
CASCADE,
   discountId INTEGER REFERENCES discount (id) ON UPDATE CASCADE ON DELETE SET
NULL
);

CREATE TABLE productCategory (
   PRIMARY KEY (productId, categoryId),
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE
CASCADE,
   categoryId INTEGER NOT NULL REFERENCES category (id) ON UPDATE CASCADE ON
DELETE CASCADE
);

CREATE TABLE orderedProduct (
   PRIMARY KEY (orderId, productId),
   quantity INTEGER NOT NULL CHECK ( ( quantity > 0  ) ),
   priceBought REAL NOT NULL,
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE
CASCADE,
   orderId INTEGER NOT NULL REFERENCES orders (id) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

```
CREATE TABLE review (
   id SERIAL PRIMARY KEY,
   reviewDate DATE NOT NULL,
   rating REAL,
   title VARCHAR(256) NOT NULL,
   upvoteCount INTEGER,
   reviewText TEXT,
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE,
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE notifications (
   id SERIAL PRIMARY KEY,
   notificationDate DATE NOT NULL,
   notificationText VARCHAR(256) NOT NULL,
   isRead BOOL DEFAULT false NOT NULL,
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE changeOfOrder (
   id SERIAL PRIMARY KEY,
   notificationId INTEGER NOT NULL REFERENCES notifications (id) ON UPDATE CASCADE ON DELETE CASCADE,
   orderId INTEGER NOT NULL REFERENCES orders (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE changeInPrice (
   id SERIAL PRIMARY KEY,
   notificationId INTEGER NOT NULL REFERENCES notifications (id) ON UPDATE CASCADE ON DELETE CASCADE,
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE itemAvailability (
   id SERIAL PRIMARY KEY,
   notificationId INTEGER NOT NULL REFERENCES notifications (id) ON UPDATE CASCADE ON DELETE CASCADE,
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE paymentApproved (
   id SERIAL PRIMARY KEY,
   notificationId INTEGER NOT NULL REFERENCES notifications (id) ON UPDATE CASCADE ON DELETE CASCADE,
   paymentTransactionId INTEGER NOT NULL REFERENCES paymentTransaction (id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

```
CREATE TABLE likedReview (
   id SERIAL PRIMARY KEY,
   notificationId INTEGER NOT NULL REFERENCES notifications (id) ON UPDATE CASCADE ON DELETE CASCADE,
   reviewId INTEGER NOT NULL REFERENCES review (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE report (
   PRIMARY KEY (userId, reviewId),
   reportDate DATE NOT NULL,
   reason VARCHAR(256) NOT NULL,
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE,
   reviewId INTEGER NOT NULL REFERENCES review (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE blockAction (
   id SERIAL PRIMARY KEY,
   blockDate DATE NOT NULL,
   blockedAction blocked_status NOT NULL,
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE wishlist (
   PRIMARY KEY (userId, productId),
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE,
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE shoppingCart (
   PRIMARY KEY (userId, productId),
   quantity INTEGER,
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE,
   productId INTEGER NOT NULL REFERENCES product (id) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE upvote (
   PRIMARY KEY (userId, reviewId),
   userId INTEGER NOT NULL REFERENCES users (id) ON UPDATE CASCADE ON DELETE CASCADE,
   reviewId INTEGER NOT NULL REFERENCES review (id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

A.2. Database Population

> First 10 lines of our population script: The rest of the population script can be found here

```
INSERT INTO users (username, userPath, dateofbirth, userPassword, email) VALUES
   ('saul_goodman', '/users/saul_goodman', '1999-10-08', 'bettercallsaul',
'thegoodmansaul@gmail.com'),
   ('pescator', '/users/pescator', '1999-10-08', '1000hrsofCSGO',
'mlgpescator@gmail.com'),
   ('impostor', '/users/impostor', '1978-07-27', 'venting',
'sussyamongus@outlook.com'),
   ('mr-white', '/users/mr-white', '2003-05-10', 'albuquerque',
'walterwhite@outlook.com');

INSERT INTO brand (brandName) VALUES
   ('Brand A'),
   ('Brand B'),
   ('Brand C');
```

Revision History

Changes made to the first submission:

GROUP45, 22/10/2023

- Carlos Manuel da Silva Costa
    - Email: up202004151@up.pt
- João Pedro Rodrigues Coutinho
    - Email: up202108787@up.pt
- Miguel Jorge Medeiros Garrido
    - Email: up202108889@up.pt
- Tomás Henrique Ribeiro Coelho
    - Email: up202108861@up.pt

Editor: Tomás Coelho