Лабораторна робота 3. (2 год.)
ПРЕДСТАВЛЕННЯ. ТРИГЕРИ. ЗБЕРЕЖЕНІ ПРОЦЕДУРИ. ТРАНЗАКЦІЇ.

Представлення

Створення представлень - ***CREATE VIEW***

В SQL Представлення це віртуальна таблиця, що базується на результуючих запитах SQL-виразів.

Представлення має рядки та колонки, як справжня таблиця. Поля Представлення це поля з однієї чи декількох реальних таблиць Бази Даних.

Ви можете додавати SQL-функції, WHERE та JOIN-вирази, щоб переглянути та вивести дані з Представлення.

Синтаксис створення Представлення CREATE VIEW.

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name
WHERE condition
```

Зауважте: Представлення завжди показує обновлені дані! Механізм Бази Даних оновлює дані, кожного разу як лише користувач виконав запит.

Приклад створення Представлення CREATE VIEW

Представлення "Current_Product_List" перелічує всі активні продукти з таблиці "Products"

```
CREATE VIEW Current_Product_List AS

SELECT ProductID,ProductName

FROM Products

WHERE Discontinued=No
```

Тепер ми можемо вибрати дані з Представлення наступним чином:

```
SELECT * FROM Current_Product_List
```

Інший приклад Представлення з Бази Даних вибирає кожен продукт з таблиці "Products" з ціною товару вище середньої ціни даних товарів.

```
CREATE VIEW Products_Above_Average_Price AS

SELECT DISTINCT ProductName,UnitPrice

FROM Products

WHERE UnitPrice>(SELECT AVG(UnitPrice) FROM Products)
```

Тепер ми можемо вибрати дані з Представлення наступним чином:

```
SELECT * FROM [Products Above Average Price]
```

Оновлення таблиці Представлення - Updating View

Ми можемо оновити Представлення використовуючи наступний SQL синтаксис:

```
CREATE OR REPLACE VIEW view_name AS

SELECT column_name(s)

FROM table_name

WHERE condition
```

Якщо необхідно добавити стовбчик "Category" до Представлення " Current_Product_List " , тоді можна оновити слідуючим чином:

```
CREATE VIEW Current_Product_List AS

SELECT ProductID,ProductName,Category

FROM Products

WHERE Discontinued=No
```

Видалення Представлення:

```
DROP VIEW view_name
```

Тригер – це відкомпільована SQL-Процедура, виконання якої обумовлене настанням певних подій усередині реляційної бази даних. Застосування *тригерів* здебільшого досить зручне для користувачів бази даних. І все-таки їхнє використання часто пов'язане з додатковими витратами ресурсів на операції введення/виведення. У тому випадку, коли тих же результатів (з набагато меншими непродуктивними витратами ресурсів) можна домогтися за допомогою збережених процедур або прикладних програм, застосування *тригерів* недоцільно.

Тригери – особливий інструмент SQL-Сервера, використовуваний для підтримки цілісності даних у базі даних. За допомогою обмежень цілісності, правил і значень за замовчуванням не завжди можна домогтися потрібного рівня функціональності. Часто потрібно реалізувати складні алгоритми перевірки даних, гарантуюча їхня вірогідність і реальність. Крім того, іноді необхідно відслідковувати зміни значень таблиці, щоб потрібним образом змінити зв'язані дані. *Тригери* можна розглядати як свого роду фільтри, що вступають у дію після виконання всіх операцій відповідно до правил, стандартними значеннями й т.д.

Тригер являє собою спеціальний тип збережених процедур, що запускаються сервером автоматично при спробі зміни даних у таблицях, з якими *тригери* зв'язані. Кожний *тригер* прив’язується до конкретної таблиці. Всі виконувані їм модифікації даних розглядаються як одна транзакція. У випадку виявлення помилки або порушення цілісності даних відбувається відкат цієї транзакції. Тим самим внесення змін забороняється. Відміняються також всі зміни, уже зроблені *тригером*.

Створює *тригер* тільки власник бази даних. Це обмеження дозволяє уникнути випадкової зміни структури таблиць, способів зв'язку з ними інших об'єктів і т.п.

Тригер являє собою досить корисний і у той же час небезпечний засіб. Так, при неправильній логіці його роботи можна легко знищити цілу базу даних, тому *тригери* необхідно дуже ретельно налагоджувати.

На відміну від звичайної підпрограми, *тригер* виконується неявно в кожному випадку виникнення *тригерної події*, до того ж він не має аргументів. Приведення його в дію іноді називають запуском *тригера*. За допомогою *тригерів* досягаються наступні цілі:

- перевірка коректності введених даних і виконання складних обмежень цілісності даних, які важко, якщо взагалі можливо, підтримувати за допомогою обмежень цілісності, встановлених для таблиці;
- видача попереджень, що нагадують про необхідність виконання деяких дій при відновленні таблиці, реалізованому певним чином;
- нагромадження аудиторської інформації за допомогою фіксації відомостей про внесені зміни й тих осіб, які їх виконали;
- підтримка реплікації.

Основний формат команди CREATE TRIGGER показаний нижче:

```
<Визначення_тригера>::=
 CREATE TRIGGER ім'я_тригера
 BEFORE | AFTER <тригерна_подія>
 ON <ім'я_таблиці>
 [REFERENCING
   <список_старих_або_нових_псевдонімів>]
 [FOR EACH { ROW | STATEMENT}]
 [WHEN(умова_тригера)]
 <тіло_тригера>
```

тригерні події складаються зі вставки, видалення й оновлення рядків у таблиці. В останньому випадку для *тригерної події* можна вказати конкретні імена стовпців таблиці. Час запуску *тригера* визначається за допомогою ключових слів BEFORE (*тригер* запускається до виконання пов'язаних з ним подій) або AFTER (після їхнього виконання).

Виконувані *тригером* дії задаються для кожного рядка (FOR EACH ROW), охопленого даною подією, або тільки один раз для кожної події (FOR EACH STATEMENT).

Позначення <список_старих_або_нових_псевдонімів> ставиться до таких компонентів, як старий або новий рядок (OLD / NEW) або стара або нова таблиця (OLD TABLE / NEW TABLE). Ясно, що старі значення не застосовні для подій вставки, а нові - для подій видалення.

Збережені процедури являють собою групи зв'язаних між собою операторів SQL, застосування яких робить роботу програміста більш легкою і гнучкою, оскільки виконати *збережену процедуру* часто виявляється набагато простіше, ніж послідовність окремих операторів SQL. Збережені процедури являють собою набір команд, що складає з одного або декількох операторів SQL або функцій і, що зберігається в базі даних у відкомпільованому виді. *Виконання* в базі даних *збережених процедур* замість окремих операторів SQL дає користувачеві наступні переваги:

- необхідні оператори вже містяться в базі даних;
- всі вони пройшли етап *синтаксичного аналізу* й перебувають у виконуємому форматі; перед *виконанням збереженої процедури* SQL Server генерує для неї *план виконання*, виконує її оптимізацію й компіляцію;
- *збережені процедури* підтримують *модульне програмування*, тому що дозволяють розбивати великі завдання на самостійні, більш дрібні й зручні в управлінні частини;
- *збережені процедури* можуть викликати інші *збережені процедури* й функції;
- *збережені процедури* можуть бути викликані із прикладних програм інших типів;
- як правило, *збережені процедури* виконуються швидше, ніж послідовність окремих операторів;
- *збережені процедури* простіше використовувати: вони можуть складатися з десятків і сотень команд, але для їхнього запуску досить указати всього лише ім'я потрібної *збереженої процедури*. Це дозволяє зменшити розмір запиту, що посилається від клієнта на сервер, а виходить, і навантаження на мережу.

Зберігання процедур у тому ж місці, де вони виконуються, забезпечує зменшення обсягу переданих по мережі даних і підвищує загальну продуктивність системи. Застосування *збережених процедур* спрощує супровід програмних комплексів і внесення змін у них. Звичайно всі обмеження цілісності у вигляді правил і алгоритмів обробки даних реалізуються на сервері баз даних і доступні кінцевому додатку у вигляді набору *збережених процедур*, які й представляють інтерфейс обробки

даних. Для забезпечення цілісності даних, а також з метою безпеки, додаток звичайно не отримує прямого доступу до даних – вся робота з ними ведеться шляхом виклику тих або інших *збережених процедур*.

Подібний підхід робить досить простою модифікацію алгоритмів обробки даних, що негайно ж стають доступними для всіх користувачів мережі, і забезпечує можливість розширення системи без внесення змін у сам додаток: досить змінити *збережену процедуру* на сервері баз даних. Розроблювачу не потрібно перекомпілювати додаток, створювати його копії, а також інструктувати користувачів про необхідність роботи з новою версією. Користувачі взагалі можуть не підозрювати про те, що в систему внесені зміни.

Збережені процедури існують незалежно від таблиць або яких-небудь інших об'єктів баз даних. Вони викликаються клієнтською програмою, іншою *збереженою процедурою* або тригером. Розроблювач може управляти правами доступу до *збереженої процедури*, дозволяючи або забороняючи її *виконання*. Змінювати код *збереженої процедури* дозволяється тільки її власникові або члену фіксованої ролі бази даних. При необхідності можна передати права володіння нею від одного користувача до іншого.

Створення, зміна й видалення збережених процедур

Створення збереженої процедури передбачає рішення наступних завдань:

- визначення типу створюваної *збереженої процедури*: тимчасова або користувальницька. Крім цього, можна створити свою власну системну *збережену процедуру*, призначивши їй ім'я із префіксом `sp_` і помістивши її в системну базу даних. Така процедура буде доступна в контексті будь-якої бази даних локального сервера;
- планування прав доступу. При *створенні збереженої процедури* варто враховувати, що вона буде мати ті ж права доступу до об'єктів бази даних, що і її користувач, що її створив;
- визначення *параметрів збереженої процедури*. Подібно процедурам, що входять до складу більшості мов програмування, *збережені процедури* можуть мати *вхідні* й вихідні *параметри*;
- розробка коду *збереженої процедури*. Код процедури може містити послідовність будь-яких команд SQL, включаючи виклик інших *збережених процедур*.

Створення нової й зміна наявної *збереженої процедури* здійснюється за допомогою наступної команди:

```
<визначення_процедури>::=
{CREATE | ALTER } PROC[EDURE] ім'я_процедури
   [;номер]
[{@ім'я_параметра тип_даних } [VARYING ]
  [=default][OUTPUT] ][,...n]
[WITH { RECOMPILE | ENCRYPTION | RECOMPILE,
  ENCRYPTION }]
[FOR REPLICATION]
AS
  sql_оператор [...n]
```

Розглянемо *параметри* даної команди.

Використовуючи префікси sp_, #, ##, створювану процедуру можна визначити в якості системної або тимчасовий. Як видно із синтаксису команди, не допускається вказувати ім'я власника, якому буде належати створювана процедура, а також ім'я бази даних, де вона повинна бути розміщена. Таким чином, щоб розмістити створювану *збережену процедуру* в конкретній базі даних, необхідно виконати команду CREATE PROCEDURE у контексті цієї бази даних. При *обігу* з тіла *збереженої процедури* до об'єктів тієї ж бази даних можна використовувати вкорочені імена, тобто без вказівки ім'я бази даних. Коли ж потрібно звернутися до об'єктів, розташованим в інших базах даних, вказівка ім'я бази даних обов'язково.

Номер в ім'ї – це ідентифікаційний номер *збереженої процедури*, однозначно визначальний її в групі процедур. Для зручності керування процедурами логічно однотипні *збережені процедури* можна групувати, привласнюючи їм однакові імена, але різні ідентифікаційні номери.

Для передачі вхідних і вихідних даних у створюваній *збереженій процедурі* можуть використовуватися *параметри*, імена яких, як і імена локальних змінних, повинні починатися із символу @. В одній *збереженій процедурі* можна задати безліч *параметрів*, розділених комами. У тілі процедури не повинні застосовуватися локальні змінні, імена яких збігаються з іменами *параметрів* цієї процедури.

Для визначення типу даних, що буде мати відповідний *параметр збереженої процедури*, годяться будь-які типи даних SQL, включаючи визначені користувачем. Однак тип даних CURSOR може бути використаний тільки як *вихідний параметр збереженої процедури*, тобто із вказівкою ключового слова OUTPUT.

Наявність ключового слова OUTPUT означає, що відповідний *параметр* призначений для повернення даних зі *збереженої процедури*. Однак це зовсім не означає, що *параметр* не підходить для передачі значень у *збережену процедуру*. Вказівка ключового слова OUTPUT пропонує серверу при виході зі *збереженої процедури* привласнити поточне значення *параметра* локальної змінної, котра була зазначена при виклику процедури як значення *параметра*. Відзначимо, що при зазначенні ключового слова OUTPUT значення відповідного *параметра* при виклику процедури може бути задано тільки за допомогою локальної змінної. Не дозволяється використання будь-яких виразів або констант, припустиме для звичайних *параметрів*.

Ключове слово VARYING застосовується разом з *параметром* OUTPUT, що має тип CURSOR. Воно визначає, що *вихідним параметром* буде результуюча множина.

Ключове слово DEFAULT являє собою значення, що буде приймати відповідний *параметр за замовчуванням*. Таким чином, при виклику процедури можна не вказувати явно значення відповідного *параметра*.

Тому що сервер кэшує *план виконання* запиту й компільований код, при наступному виклику процедури будуть використовуватися вже готові значення. Однак у деяких випадках все-таки потрібно виконувати перекомпіляцію коду процедури. Вказівка ключового слова RECOMPILE пропонує системі створювати *план виконання збереженої процедури* при кожному її виклику.

Параметр FOR REPLICATION затребуваний при реплікації даних і включенні створюваної *збереженої процедури* як стаття в публікацію.

Ключове слово ENCRYPTION наказує серверу виконати шифрування коду *збереженої процедури*, що може забезпечити захист від використання авторських алгоритмів, що реалізують роботу *збереженої процедури*.

Ключове слово AS розміщається на початку власне тіла *збереженої процедури*, тобто набору команд SQL, за допомогою яких і буде реалізовуватися та або інша дія. У тілі процедури можуть застосовуватися практично всі команди SQL, оголошуватися транзакції, встановлюватися блокування й викликатися інші *збережені процедури*. Вихід зі *збереженої процедури* можна здійснити за допомогою команди RETURN.

Видалення збереженої процедури здійснюється командою:

```
DROP PROCEDURE {ім'я_процедури} [,...n]
```

Виконання збереженої процедури

Для *виконання збереженої процедури* використовується команда:

```
[[ EXEC [ UTE] ім'я_процедури [;номер]
[[@ім'я_параметра=]{значення | @ім'я_змінної}
  [OUTPUT ]|[DEFAULT ]][,...n]
```

Якщо виклик *збереженої процедури* не є єдиною командою в пакеті, то присутність команди EXECUTE обов'язково. Більше того, ця команда потрібна для виклику процедури з тіла іншої процедури або тригера.

Користувальницькі функції подібні *збереженим процедурам*, але, на відміну від них, можуть застосовуватися в запитах так само, як і системні *вбудовані функції*. *Користувальницькі функції*, що повертають таблиці, можуть стати альтернативою переглядам. Перегляди обмежені одним виразом SELECT, а *користувальницькі функції* здатні включати додаткові вирази, що дозволяє створювати більш складні й потужні конструкції.

Створення й зміна *функції* даного типу виконується за допомогою команди:

```
<визначення_скаляр_функції>::=
{CREATE | ALTER } FUNCTION [власник.]
   ім'я_функції
( [ { @ім'я_параметра скаляр_тип_даних
   [=default]}[,...n]])
RETURNS скаляр_тип_даних
[WITH {ENCRYPTION | SCHEMABINDING}
   [,...n] ]
```

[AS]
BEGIN
<тіло_функції>
RETURN скаляр_вираз
END

Розглянемо призначення параметрів команди.

Функція може містити один або кілька *вхідних параметрів* або не містити жодного. Кожний параметр повинен мати унікальне в межах створюваної *функції* ім'я й починатися із символу "@". Після ім'я вказується тип даних параметра. Додатково можна вказати значення, що буде автоматично привласнюватися параметру (DEFAULT), якщо користувач явно не вказав значення відповідного параметра при виклику *функції*.

За допомогою конструкції RETURNS скаляр_тип_даних вказується, який тип даних буде мати значення, що повертається функцією.

Додаткові параметри, з якими повинна бути створена *функція*, можуть бути зазначені за допомогою ключового слова WITH. Завдяки ключовому слову ENCRYPTION код команди, використовуваний для створення *функції*, буде зашифрований, і ніхто не зможе переглянути його. Ця можливість дозволяє сховати логіку роботи *функції*. Крім того, у тілі *функції* може виконуватися звертання до різних об'єктів бази даних, а тому зміна або видалення відповідних об'єктів може привести до порушення роботи *функції*. Щоб уникнути цього, потрібно заборонити внесення змін, указавши при створенні цієї *функції* ключове слово SCHEMABINDING.

Між ключовими словами BEGIN...END вказується набір команд, вони й будуть тілом *функції*.

Коли в ході виконання коду *функції* зустрічається ключове слово RETURN, виконання *функції* завершується і як результат її обчислення вертається значення, зазначене безпосередньо після слова RETURN. Відзначимо, що в тілі *функції* дозволяється використання кількох команд RETURN, які можуть повертати різні значення. В якості повертаємого значення допускаються як звичайні константи, так і складні вираження. Єдина умова – тип даних значення, що повертається, повинен збігатися з типом даних, зазначеним після ключового слова RETURNS.

Транзакції

Транзакція — набір команд, що виконується як єдине ціле. У транзакції або всі команди будуть виконані, або жодна з них не виконається. Якщо хоча б одна з команд транзакції не може бути виконана, здійснюється відкочування (відновлюється стан системи, в якому вона перебувала до початку виконання транзакції).

Транзакції мають задовольняти вимоги ACID (Atomicity, Consistency, Isolation, Durability — атомарність, несуперечність, ізольованість, довговічність), що гарантують правильність і надійність роботи системи.

Атомарність передбачає:

- виконуються всі операції транзакції або жодна з них не виконується;
- якщо виконання транзакції було перерване, то всі зроблені транзакцією зміни мають бути скасовані

Дії, спрямовані на підтримку атомарності транзакції під час її аварійного завершення у зв’язку з помилками введення/виведення, перевантаженнями системи чи блокуваннями, називаються відновленням транзакції.

Дії, спрямовані на забезпечення атомарності під час виходу з ладу системи, називаються відновленням у разі виникнення перебоїв.

Несуперечність означає, що транзакція, яка працює з несуперечною базою даних, після завершення роботи залишає її також у несуперечному стані. Транзакція не повинна порушувати цілісності бази даних

Для вирішення проблем одночасного доступу інститут ANSI розробив спеціальний стандарт, який визначає чотири рівні блокування (кожний вищий рівень передбачає виконання умов усіх нижчих рівнів)

Рівень 0. Заборона «забруднення» даних. На цьому рівні вимагається, щоб змінювати дані могла лише одна транзакція. Якщо іншій транзакції необхідно змінити ці ж дані, то вона має очікувати завершення першої транзакції.

Рівень 1. Заборона некоректного зчитування. Якщо певна транзакція розпочала змінювати дані, то жодна інша транзакція не зможе зчитати ці дані доти, доки перша не завершиться.

Рівень 2. Заборона неповторюваного зчитування. Якщо певна транзакція зчитує дані, то жодна інша транзакція не зможе їх змінити. Отже, під час повторного зчитування дані перебуватимуть у початковому стані.

Рівень 3. Заборона «фантомів». Якщо транзакція звертається до даних, то жодна інша транзакція не зможе додати чи видалити рядки, що можуть бути зчитані під час виконання транзакції. Реалізація цього рівня блокування виконується блокуванням діапазону ключів. Це блокування накладається не на конкретні рядки таблиці, а на рядки, що відповідають певній логічній умові.

Властивість ізольованості означає, що на роботу транзакції не мають впливати інші транзакції. Транзакція «бачить» дані в тому стані, в якому вони перебували до початку роботи іншої транзакції, або в тому стані, в якому вони перебувають після її завершення. Одна транзакція не може переглядати проміжні стани даних, що використовуються іншими транзакціями. Якщо транзакція зчитує кілька разів ті самі дані, то вона повинна отримувати їх щоразу в тому стані, в якому вони були під час першого зчитування. Ще одним аспектом ізольованості є неможливість роботи з неповними результатами — незавершена транзакція не може передавати свої результати іншим транзакціям до підтвердження свого успішного завершення.

Після того, як було підтверджено успішне завершення роботи транзакції (*Commit*), система має гарантувати, що її результати не будуть втрачені, незважаючи на можливі перебої. Це й називається довговічністю. Для забезпечення довговічності застосовуються механізми відновлення бази даних. У випадку, якщо транзакція не була успішною, відбувається її відкат *Rollback*.

Фіксація транзакції (COMMIT):

```
DELETE FROM CUSTOMERS
   WHERE AGE = 25;
COMMIT;
```

Створення точки відкату (SAVEPOINT Command):

```
SAVEPOINT SAVEPOINT_NAME;
ROLLBACK TO SAVEPOINT_NAME;
```

```
SAVEPOINT SP1;
DELETE FROM CUSTOMERS WHERE ID=1;

SAVEPOINT SP2;
DELETE FROM CUSTOMERS WHERE ID=2;

SAVEPOINT SP3;
DELETE FROM CUSTOMERS WHERE ID=3;

ROLLBACK TO SP2;
```

Для видалення точки відкату (RELEASE SAVEPOINT):

Команда RELEASE SAVEPOINT використовується для видалення SAVEPOINT, який ви створили. Синтаксис RELEASE SAVEPOINT виглядає наступним чином:

RELEASE SAVEPOINT *SAVEPOINT_NAME*;

Завдання:

1) Використовуючи загальне завдання Лабораторної роботи №2, придумати 2 таблиці "Представлення" та реалізувати загалом 5 запитів на отримання даних з таблиць "Представлення". Перевірити оновлюваність таблиць "Представлень", шляхом внесення змін в базові таблиці та виконання запитів до таблиць "Представлень".

2) Використовуючи БД із Лабораторної роботи №2, ввести числові атрибути. Придумати та реалізувати 2 тригери (на додавання даних та оновлення даних).

3) Використовуючи БД із Лабораторної роботи №2, ввести придумати та реалізувати 1 користувацьку функцію та 1 процедуру.

ПРИКЛАДИ

ТРИГЕРИ

```
use supermarket;
CREATE TRIGGER supermarket.get_t_cost
AFTER INSERT   ON supermarket.`order`
    FOR EACH ROW
       UPDATE supermarket.`check`
       SET total_cost = (
         SELECT SUM(cost)
         FROM supermarket.`order`
         WHERE id_check = NEW.id_check)
       WHERE id_check = NEW.id_check;

use supermarket;
CREATE TRIGGER get_cost
BEFORE UPDATE   ON supermarket.`order`
    FOR EACH ROW
       SET NEW.cost = NEW.count  *
       (SELECT Вартість_од
         FROM supermarket.`supply`
         WHERE supermarket.`supply`.id_Склад = NEW.id_Склад);

CREATE TRIGGER supermarket.get_t_cost2
AFTER UPDATE   ON supermarket.`order`
    FOR EACH ROW
       UPDATE supermarket.`check`
       SET total_cost = (
         SELECT SUM(cost)
         FROM supermarket.`order`
         WHERE id_check = NEW.id_check)
            WHERE id_check = NEW.id_check;
```

id_order	id_Склад	id_check	count	cost
5	1	1	2	60
6	3	2	3	120
7	2	2	2	70
8	1	1	3	90
9	1	2	1	30
NULL	NULL	NULL	NULL	NULL

Рис.1. Відношення "Замовлення" перед застовуванням тригерів

id_check	total_cost	date	time
1	0	2016-04-10	17:05:00
2	220	2016-04-10	19:20:00
3	0	2016-04-11	12:17:00
NULL	NULL	NULL	NULL

Рис.2. Відношення "Чек" перед застовуванням тригерів

id_order	id_Склад	id_check	count	cost
5	1	1	2	60
6	3	2	3	120
7	2	2	4	140
8	1	1	3	90
9	1	2	1	30
NULL	NULL	NULL	NULL	NULL

Рис.3. Відношення "Замовлення" після застовуванням тригерів

id_check	total_cost	date	time
1	150	2016-04-10	17:05:00
2	290	2016-04-10	19:20:00
3	0	2016-04-11	12:17:00
NULL	NULL	NULL	NULL

Рис.4. Відношення "Чек" після застовуванням тригерів

ПРЕДСТАВЛЕННЯ

```
use supermarket;
CREATE VIEW product_data AS
SELECT DISTINCT
    product.Назва as `product`,
    Вага as `weight`,
    provider.Назва as `provider`,
    `Дата_пост` as `date`,
    `К-сть` as `count`,
    `Вартість_од` as `price`,
    SUM(`order`.count) as `sold`
FROM product, supply, provider, `order`
WHERE product.id_Товар = supply.id_Товар
AND supply.id_Пост = provider.id_Постачальник
AND supply.id_Склад = `order`.id_Склад
GROUP BY product.Назва;
```

Запит1 до таблиці Представлення:

```
use supermarket;
SELECT * FROM product_data ;
```

product	weight	provider	date	count	price	sold
ТМ Richfield, чай «Мікс» в пакетиках	37,5 г	Спецторг, ПП	2015-11-25	10	40	3
ТМ Richfield, чай зелений в пакетиках	37,5 г	Спецторг, ПП	2015-12-20	20	35	4
ТМ Richfield, чай чорний в пакетиках	37,5 г	Спецторг, ПП	2015-12-20	25	30	6

Запит2 до таблиці Представлення:

```
use supermarket;
SELECT product, price, sold, (price * sold) as income
FROM product_data
ORDER BY income DESC;
```

product	price	sold	income
ТМ Richfield, чай чорний в пакетиках	30	6	180
ТМ Richfield, чай зелений в пакетиках	35	4	140
ТМ Richfield, чай «Мікс» в пакетиках	40	3	120

Запит3 до таблиці Представлення після додавання нових даних в таблицю order (Автоматичне оновлення):

```
use supermarket;
SELECT * FROM product_data ;
```

id_order	id_Склад	id_check	count	cost
5	1	1	2	60
6	3	2	3	120
7	2	2	4	140
8	1	1	3	90
9	1	2	1	30
11	5	3	3	300
12	6	3	1	105
NULL	NULL	NULL	NULL	NULL

product	weight	provider	date	count	price	sold
Lavazza,Crema e Gusto Forte	1 кг	Coffeez	2016-01-10	15	100	4
ТМ Richfield, чай «Мікс» в пакетиках	37,5 г	Спецторг, ПП	2015-11-25	10	40	3
ТМ Richfield, чай зелений в пакетиках	37,5 г	Спецторг, ПП	2015-12-20	20	35	4
ТМ Richfield, чай чорний в пакетиках	37,5 г	Спецторг, ПП	2015-12-20	25	30	6