ЭК

`EXISTS` и `NOT EXISTS` - это операторы SQL, используемые в подзапросах. Они используются для проверки наличия или отсутствия записей, возвращаемых подзапросом.

`EXISTS` возвращает `true`, если подзапрос возвращает хотя бы одну запись. В противном случае, он возвращает `false`.

`NOT EXISTS` делает противоположное - возвращает `true`, если подзапрос не возвращает ни одной записи, и `false` в противном случае.

Вот примеры их использования:

```
-- EXISTS
SELECT * FROM orders
WHERE EXISTS (SELECT 1 FROM products WHERE products.id = orders.product_id
AND products.price > 100);

-- NOT EXISTS
SELECT * FROM customers
WHERE NOT EXISTS (SELECT 1 FROM orders WHERE orders.customer_id =
customers.id);
```

В первом примере мы выбираем все заказы, для которых существуют продукты с ценой больше 100. Во втором примере мы выбираем всех клиентов, которые не сделали ни одного заказа.

`EXISTS` и `NOT EXISTS` работают быстро, потому что SQL-сервер останавливает выполнение подзапроса, как только найдено соответствующее условию значение.

тригеры

Триггеры в SQL - это специальные процедуры, которые автоматически запускаются или "срабатывают" при выполнении определенного типа операций, таких как `INSERT`, `UPDATE` или `DELETE`, на определенной таблице или представлении.

Триггеры могут быть полезны для поддержания целостности данных, выполнения сложной бизнес-логики и аудита изменений данных.

Существуют два основных типа триггеров: `AFTER` и `INSTEAD OF`.

AFTER Triggers (или FOR Triggers)

`AFTER` триггеры (в некоторых системах управления базами данных они называются `FOR` триггеры) запускаются после того, как операция `INSERT`, `UPDATE` или `DELETE` была выполнена.

Пример `AFTER` триггера:

```
CREATE TRIGGER after_insert
AFTER INSERT ON employees
```

```
FOR EACH ROW
BEGIN
   INSERT INTO audit_table (employee_id, action) VALUES (NEW.id,
'INSERT');
END;
```

В этом примере триггер `after_insert` будет срабатывать после каждой операции `INSERT` на таблице `employees`, записывая информацию об этом в `audit_table`.

INSTEAD OF Triggers

`INSTEAD OF` триггеры запускаются вместо триггерной операции (`INSERT`, `UPDATE` или `DELETE`). Они часто используются с представлениями, когда вы хотите изменить данные в представлении, которое не поддерживает обновления напрямую.

Пример `INSTEAD OF` триггера:

```
CREATE TRIGGER instead_of_insert
INSTEAD OF INSERT ON view_employees
FOR EACH ROW
BEGIN
   INSERT INTO employees (id, name) VALUES (NEW.id, NEW.name);
END;
```

В этом примере триггер `instead_of_insert` будет срабатывать вместо каждой операции `INSERT` на представлении `view_employees`, вставляя данные непосредственно в таблицу `employees`.

7.sql

```
-- студентов четвертого факультета, не выступавших на конференциях
USE conference_db_lab1;

SELECT s.name
    FROM student s
             JOIN uni_group g ON s.group_id = g.id
             JOIN faculty f ON g.faculty_id = f.id
    WHERE f.number = 4
      AND NOT EXISTS (SELECT 1
                      FROM authorship a
                      WHERE a.author_id = s.id);
```

```
SELECT s.name
```

Эта строка выбирает имена студентов из таблицы `student`.

```
FROM student s
```

Эта строка указывает, что мы будем использовать таблицу `student` и дает ей псевдоним `s` для упрощения ссылок на нее в дальнейшем коде.

```
JOIN uni_group g ON s.group_id = g.id
```

Эта строка объединяет таблицу `student` с таблицей `uni_group` по полю `group_id`. Таблице `uni_group` присваивается псевдоним `g`.

```
JOIN faculty f ON g.faculty_id = f.id
```

Эта строка объединяет результат предыдущего объединения с таблицей `faculty` по полю `faculty_id`. Таблице `faculty` присваивается псевдоним `f`.

```
WHERE f.number = 4
```

Эта строка фильтрует результаты, оставляя только те, где номер факультета равен 4.

```
AND NOT EXISTS (SELECT 1
                      FROM authorship a
                      WHERE a.author_id = s.id);
```

Эта строка дополнительно фильтрует результаты, исключая студентов, которые выступали на конференциях. Это достигается путем проверки на отсутствие записей в таблице `authorship`, где `author_id` совпадает с `id` студента.

8.sql

```
-- студентов, выступивших на всех конференциях;
USE conference_db_lab1;

SELECT s.id AS student_id,
       s.name AS student_name
FROM student s
WHERE NOT EXISTS (
    SELECT 1 FROM conference c
    WHERE NOT EXISTS (
        SELECT 1 FROM authorship a
        JOIN topic t ON a.topic_id = t.id
        JOIN conference_session cs ON t.session_id = cs.id
```

```
        WHERE a.author_id = s.id AND cs.conference_id = c.id
    )
);
```

```
USE conference_db_lab1;
```

Эта строка указывает базу данных, которую мы будем использовать для нашего запроса. В данном случае, это `conference_db_lab1`.

```
SELECT s.id AS student_id,
       s.name AS student_name
```

Эта строка выбирает идентификаторы и имена студентов из таблицы `student`.

```
FROM student s
```

Эта строка указывает, что мы будем использовать таблицу `student` и дает ей псевдоним `s` для упрощения ссылок на нее в дальнейшем коде.

```
WHERE NOT EXISTS (
    SELECT 1 FROM conference c
    WHERE NOT EXISTS (
        SELECT 1 FROM authorship a
        JOIN topic t ON a.topic_id = t.id
        JOIN conference_session cs ON t.session_id = cs.id
        WHERE a.author_id = s.id AND cs.conference_id = c.id
    )
);
```

Этот блок кода выбирает студентов, которые выступали на всех конференциях. Это достигается путем проверки на отсутствие конференций, на которых не выступал данный студент. Для каждой конференции проверяется наличие выступления студента на ней, что достигается путем проверки наличия записи в таблице `authorship`, связанной с данной конференцией и данным студентом.

В результате выполнения этого запроса вы получите список идентификаторов и имен студентов, которые выступали на всех конференциях.

9.sql

```
-- пары студентов, всегда выступающие вместе.
USE conference_db_lab1;
```

```
SELECT a1.author_id AS student1, a2.author_id AS student2
  FROM authorship a1
         JOIN authorship a2 ON a1.topic_id = a2.topic_id AND a1.author_id
< a2.author_id
  WHERE NOT EXISTS (SELECT 1
                    FROM authorship a
                    WHERE a.author_id = a1.author_id
                      AND NOT EXISTS
                      (SELECT 1
                         FROM authorship
                         WHERE author_id = a2.author_id AND topic_id =
a.topic_id))
    AND NOT EXISTS (SELECT 1
                    FROM authorship a
                    WHERE a.author_id = a2.author_id
                      AND NOT EXISTS
                      (SELECT 1
                         FROM authorship
                         WHERE author_id = a1.author_id AND topic_id =
a.topic_id))
  GROUP BY a1.author_id, a2.author_id;
```

```
USE conference_db_lab1;
```

Эта строка указывает базу данных, которую мы будем использовать для нашего запроса. В данном случае, это `conference_db_lab1`.

```
SELECT a1.author_id AS student1, a2.author_id AS student2
```

Эта строка выбирает идентификаторы авторов из таблицы `authorship`.

```
FROM authorship a1
JOIN authorship a2 ON a1.topic_id = a2.topic_id AND a1.author_id <
a2.author_id
```

Эта строка объединяет таблицу `authorship` с самой собой по полю `topic_id`, исключая пары, где идентификаторы авторов равны или второй идентификатор больше первого (чтобы исключить дубликаты пар).

```
WHERE NOT EXISTS (SELECT 1
                  FROM authorship a
                  WHERE a.author_id = a1.author_id
                  AND NOT EXISTS
                  (SELECT 1
```

```
                    FROM authorship
                    WHERE author_id = a2.author_id AND topic_id =
a.topic_id))
```

Этот блок кода исключает авторов, которые выступали на какой-либо теме без другого автора. Это достигается путем проверки на отсутствие записей в таблице `authorship`, где первый автор выступал на теме, на которой второй автор не выступал.

```
AND NOT EXISTS (SELECT 1
                FROM authorship a
                WHERE a.author_id = a2.author_id
                AND NOT EXISTS
                (SELECT 1
                 FROM authorship
                 WHERE author_id = a1.author_id AND topic_id =
a.topic_id))
```

Этот блок кода исключает авторов, которые выступали на какой-либо теме без другого автора. Это достигается путем проверки на отсутствие записей в таблице `authorship`, где второй автор выступал на теме, на которой первый автор не выступал.

```
GROUP BY a1.author_id, a2.author_id;
```

Эта строка группирует результаты по идентификаторам авторов.

В результате выполнения этого запроса вы получите список пар авторов, которые всегда выступали вместе.

diff.sql

```
USE conference_db_lab1;

DROP TEMPORARY TABLE IF EXISTS temp_students;
CREATE TEMPORARY TABLE temp_students AS
SELECT s.id, s.name
  FROM student s
  WHERE s.id <= 5
UNION
SELECT NULL, 'Аноним';

SELECT  * FROM temp_students;

SELECT s.id, s.name
  FROM student s
  WHERE NOT EXISTS (SELECT 1 FROM temp_students ts WHERE ts.id = s.id);
```

```
SELECT s.id, s.name
  FROM student s
  WHERE s.id NOT IN (SELECT ts.id FROM temp_students ts);
```

```
USE conference_db_lab1;
```

Эта строка указывает базу данных, которую мы будем использовать для нашего запроса. В данном случае, это `conference_db_lab1`.

```
DROP TEMPORARY TABLE IF EXISTS temp_students;
```

Эта строка удаляет временную таблицу `temp_students`, если она существует.

```
CREATE TEMPORARY TABLE temp_students AS
SELECT s.id, s.name
  FROM student s
  WHERE s.id <= 5
UNION
SELECT NULL, 'Аноним';
```

Этот блок кода создает временную таблицу `temp_students` и заполняет ее данными из таблицы `student`, где идентификатор студента меньше или равен 5, а также добавляет одну запись с именем 'Аноним' и идентификатором NULL.

```
SELECT  * FROM temp_students;
```

Эта строка выбирает все записи из временной таблицы `temp_students`.

```
SELECT s.id, s.name
  FROM student s
  WHERE NOT EXISTS (SELECT 1 FROM temp_students ts WHERE ts.id = s.id);
```

Этот запрос выбирает идентификаторы и имена студентов из таблицы `student`, которые не присутствуют во временной таблице `temp_students`.

```
SELECT s.id, s.name
  FROM student s
  WHERE s.id NOT IN (SELECT ts.id FROM temp_students ts);
```

Этот запрос делает то же самое, что и предыдущий, но использует другой подход: он выбирает идентификаторы и имена студентов из таблицы `student`, идентификаторы которых не присутствуют во временной таблице `temp_students`.

---

after.sql

```
USE conference_db_lab1;

-- AFTER INSERT триггер
CREATE TRIGGER after_university_insert
  AFTER INSERT
  ON university
  FOR EACH ROW
BEGIN
  INSERT INTO faculty(university_id, number)
    VALUES (NEW.id, 1);
END;

-- AFTER UPDATE триггер
CREATE TRIGGER after_uni_group_update
  AFTER UPDATE
  ON uni_group
  FOR EACH ROW
BEGIN
  UPDATE student
  SET name = CONCAT(name, ' (', NEW.name, ')')
    WHERE group_id = NEW.id;
END;

-- AFTER DELETE триггер
CREATE TRIGGER after_university_delete
  AFTER DELETE
  ON university
  FOR EACH ROW
BEGIN
  DELETE
    FROM faculty
    WHERE university_id = OLD.id;
END;
```

```
USE conference_db_lab1;
```

Эта строка указывает базу данных, которую мы будем использовать для нашего запроса. В данном случае, это `conference_db_lab1`.

```

```

```
CREATE TRIGGER after_university_insert
  AFTER INSERT
  ON university
  FOR EACH ROW
BEGIN
  INSERT INTO faculty(university_id, number)
    VALUES (NEW.id, 1);
END;
```

Этот блок кода создает триггер `after_university_insert`, который срабатывает после каждой операции вставки в таблицу `university`. Когда новая запись добавляется в таблицу `university`, триггер автоматически добавляет новую запись в таблицу `faculty` с `university_id`, равным `id` новой записи, и `number` равным 1.

```
CREATE TRIGGER after_uni_group_update
  AFTER UPDATE
  ON uni_group
  FOR EACH ROW
BEGIN
  UPDATE student
  SET name = CONCAT(name, ' (', NEW.name, ')')
    WHERE group_id = NEW.id;
END;
```

Этот блок кода создает триггер `after_uni_group_update`, который срабатывает после каждой операции обновления в таблице `uni_group`. Когда запись в таблице `uni_group` обновляется, триггер автоматически обновляет имя каждого студента в этой группе, добавляя к нему новое имя группы в скобках.

```
CREATE TRIGGER after_university_delete
  AFTER DELETE
  ON university
  FOR EACH ROW
BEGIN
  DELETE
    FROM faculty
    WHERE university_id = OLD.id;
END;
```

Этот блок кода создает триггер `after_university_delete`, который срабатывает после каждой операции удаления в таблице `university`. Когда запись в таблице `university` удаляется, триггер автоматически удаляет все записи из таблицы `faculty`, где `university_id` равен `id` удаленной записи.

check_trigger.sql

```
-- Проверка триггера after_university_insert
INSERT INTO university(name) VALUES ('Test University');

-- Проверьте, создался ли новый факультет с university_id, соответствующим
ID нового университета
SELECT * FROM faculty WHERE university_id = (SELECT id FROM university
WHERE name = 'Test University');

-- Проверка триггера after_uni_group_update
UPDATE uni_group SET name = 'Updated Group' WHERE id = 1;

-- Проверьте, обновлены ли имена студентов, принадлежащих этой группе
SELECT * FROM student WHERE group_id = 1;

-- Проверка триггера after_university_delete
DELETE FROM university WHERE name = 'Test University';

-- Проверьте, удалены ли факультеты, связанные с удаленным университетом
SELECT * FROM faculty WHERE university_id = (SELECT id FROM university
WHERE name = 'Test University');
```

Этот SQL-скрипт проверяет работу трех триггеров: `after_university_insert`, `after_uni_group_update` и `after_university_delete`.

```
-- Проверка триггера after_university_insert
INSERT INTO university(name) VALUES ('Test University');
```

Эта строка добавляет новую запись в таблицу `university`. Триггер `after_university_insert` должен автоматически добавить новую запись в таблицу `faculty`.

```
-- Проверьте, создался ли новый факультет с university_id, соответствующим
ID нового университета
SELECT * FROM faculty WHERE university_id = (SELECT id FROM university
WHERE name = 'Test University');
```

Эта строка проверяет, была ли добавлена новая запись в таблицу `faculty` с `university_id`, соответствующим `id` нового университета.

```
-- Проверка триггера after_uni_group_update
UPDATE uni_group SET name = 'Updated Group' WHERE id = 1;
```

Эта строка обновляет запись в таблице `uni_group`. Триггер `after_uni_group_update` должен автоматически обновить имена студентов в этой группе.

```
-- Проверьте, обновлены ли имена студентов, принадлежащих этой группе
SELECT * FROM student WHERE group_id = 1;
```

Эта строка проверяет, были ли обновлены имена студентов, принадлежащих обновленной группе.

```
-- Проверка триггера after_university_delete
DELETE FROM university WHERE name = 'Test University';
```

Эта строка удаляет запись из таблицы `university`. Триггер `after_university_delete` должен автоматически удалить все записи из таблицы `faculty`, связанные с удаленным университетом.

```
-- Проверьте, удалены ли факультеты, связанные с удаленным университетом
SELECT * FROM faculty WHERE university_id = (SELECT id FROM university
WHERE name = 'Test University');
```

Эта строка проверяет, были ли удалены все записи из таблицы `faculty`, связанные с удаленным университетом.

conference_history.sql

```
USE conference_db_lab1;

-- Создаем таблицу для хранения истории изменений
CREATE TABLE IF NOT EXISTS authorship_history
(
  id          INT PRIMARY KEY AUTO_INCREMENT,
  author_id   INT         NOT NULL,
  topic_id    INT         NOT NULL,
  change_type VARCHAR(10) NOT NULL,
  change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Создаем триггер для отслеживания изменений в таблице authorship
DELIMITER //
CREATE TRIGGER authorship_after_insert
  AFTER INSERT
  ON authorship
  FOR EACH ROW
BEGIN
  INSERT INTO authorship_history (author_id, topic_id, change_type)
    VALUES (NEW.author_id, NEW.topic_id, 'INSERT');
END;
//

CREATE TRIGGER authorship_after_delete
  AFTER DELETE
```

```
  ON authorship
  FOR EACH ROW
BEGIN
  INSERT INTO authorship_history (author_id, topic_id, change_type)
    VALUES (OLD.author_id, OLD.topic_id, 'DELETE');
END;
//

CREATE TRIGGER authorship_after_update
  AFTER UPDATE
  ON authorship
  FOR EACH ROW
BEGIN
  INSERT INTO authorship_history (author_id, topic_id, change_type)
    VALUES (NEW.author_id, NEW.topic_id, 'UPDATE');
END;
//
DELIMITER ;
```

Этот SQL-скрипт создает таблицу `authorship_history` для отслеживания изменений в таблице `authorship` и три триггера, которые записывают информацию об изменениях в эту таблицу.

```
USE conference_db_lab1;
```

Эта строка указывает базу данных, которую мы будем использовать для нашего запроса. В данном случае, это `conference_db_lab1`.

```
CREATE TABLE IF NOT EXISTS authorship_history
(
  id          INT PRIMARY KEY AUTO_INCREMENT,
  author_id   INT         NOT NULL,
  topic_id    INT         NOT NULL,
  change_type VARCHAR(10) NOT NULL,
  change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Этот блок кода создает таблицу `authorship_history`, если она еще не существует. Таблица содержит следующие поля: `id` (первичный ключ, автоинкремент), `author_id` (идентификатор автора), `topic_id` (идентификатор темы), `change_type` (тип изменения: INSERT, DELETE или UPDATE) и `change_time` (время изменения, по умолчанию текущее время).

```
CREATE TRIGGER authorship_after_insert
  AFTER INSERT
  ON authorship
  FOR EACH ROW
BEGIN
```

```
  INSERT INTO authorship_history (author_id, topic_id, change_type)
    VALUES (NEW.author_id, NEW.topic_id, 'INSERT');
END;
```

Этот блок кода создает триггер `authorship_after_insert`, который срабатывает после каждой операции вставки в таблицу `authorship`. Когда новая запись добавляется в таблицу `authorship`, триггер автоматически добавляет новую запись в таблицу `authorship_history` с информацией об этом изменении.

Аналогично работают триггеры `authorship_after_delete` и `authorship_after_update`, которые срабатывают после каждой операции удаления или обновления в таблице `authorship` соответственно.

instead_og.sql

```
-- Создание представления
CREATE VIEW student_view AS SELECT * FROM student;

-- Триггер INSTEAD OF INSERT
CREATE OR REPLACE FUNCTION student_insert() RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO student(group_id, name) VALUES (NEW.group_id, NEW.name);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER student_insert_trigger
INSTEAD OF INSERT ON student_view
FOR EACH ROW EXECUTE PROCEDURE student_insert();

-- Триггер INSTEAD OF UPDATE
CREATE OR REPLACE FUNCTION student_update() RETURNS TRIGGER AS $$
BEGIN
  UPDATE student SET group_id = NEW.group_id, name = NEW.name WHERE id =
OLD.id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER student_update_trigger
INSTEAD OF UPDATE ON student_view
FOR EACH ROW EXECUTE PROCEDURE student_update();

-- Триггер INSTEAD OF DELETE
CREATE OR REPLACE FUNCTION student_delete() RETURNS TRIGGER AS $$
BEGIN
  DELETE FROM student WHERE id = OLD.id;
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER student_delete_trigger
INSTEAD OF DELETE ON student_view
FOR EACH ROW EXECUTE PROCEDURE student_delete();
```

Этот SQL-скрипт создает представление `student_view` на основе таблицы `student` и три триггера `INSTEAD OF INSERT`, `INSTEAD OF UPDATE` и `INSTEAD OF DELETE` на это представление.

```
-- Создание представления
CREATE VIEW student_view AS SELECT * FROM student;
```

Эта строка создает представление `student_view`, которое включает все записи из таблицы `student`.

```
-- Триггер INSTEAD OF INSERT
CREATE OR REPLACE FUNCTION student_insert() RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO student(group_id, name) VALUES (NEW.group_id, NEW.name);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER student_insert_trigger
INSTEAD OF INSERT ON student_view
FOR EACH ROW EXECUTE PROCEDURE student_insert();
```

Этот блок кода создает функцию `student_insert` и триггер `student_insert_trigger`, который срабатывает вместо операции вставки на представление `student_view`. Когда пытаются вставить новую запись в представление `student_view`, триггер вместо этого вставляет новую запись в таблицу `student`.

Аналогично работают триггеры `student_update_trigger` и `student_delete_trigger`, которые срабатывают вместо операций обновления и удаления на представлении `student_view` соответственно. Когда пытаются обновить или удалить запись в представлении `student_view`, эти триггеры вместо этого обновляют или удаляют соответствующую запись в таблице `student`.

check_trigger.sql

```
-- Вставка нового студента через представление
INSERT INTO student_view (group_id, name) VALUES (1, 'Клон 11');

-- Обновление имени студента через представление
UPDATE student_view SET name = 'Клон 12' WHERE name = 'Клон 11';

-- Удаление студента через представление
DELETE FROM student_view WHERE name = 'Клон 12';
```

```
-- Проверка результата на основной таблице
SELECT * FROM student;
```

Этот SQL-скрипт проверяет работу триггеров `INSTEAD OF INSERT`, `INSTEAD OF UPDATE` и `INSTEAD OF DELETE` на представлении `student_view`.

```
-- Вставка нового студента через представление
INSERT INTO student_view (group_id, name) VALUES (1, 'Клон 11');
```

Эта строка пытается вставить новую запись в представление `student_view`. Триггер `INSTEAD OF INSERT` должен перехватить эту операцию и вместо этого вставить новую запись в таблицу `student`.

```
-- Обновление имени студента через представление
UPDATE student_view SET name = 'Клон 12' WHERE name = 'Клон 11';
```

Эта строка пытается обновить запись в представлении `student_view`. Триггер `INSTEAD OF UPDATE` должен перехватить эту операцию и вместо этого обновить соответствующую запись в таблице `student`.

```
-- Удаление студента через представление
DELETE FROM student_view WHERE name = 'Клон 12';
```

Эта строка пытается удалить запись из представления `student_view`. Триггер `INSTEAD OF DELETE` должен перехватить эту операцию и вместо этого удалить соответствующую запись из таблицы `student`.

```
-- Проверка результата на основной таблице
SELECT * FROM student;
```