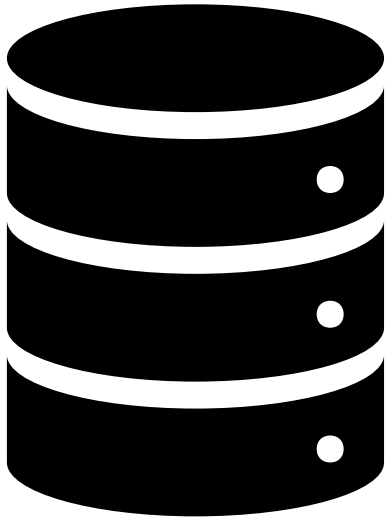


Базы данных

Лекция 8

Дополнительные возможности SQL

Меркурьева Надежда
✉ merkurievanad@gmail.com
@merkurievanad

Хранимые процедуры

– объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере

- Похожи на обыкновенные процедуры языков высокого уровня:
  - входные параметры
  - выходные параметры
  - локальные переменные
  - числовые вычисления и операции над символьными данными
- Могут выполняться стандартные операции с базами данных (как DDL, так и DML)
- Возможны циклы и ветвления

Хранимые процедуры

- позволяют повысить производительность
- расширяют возможности программирования
- поддерживают функции безопасности данных

- Вместо хранения часто запроса, достаточно ссылаться на соответствующую хранимую процедуру

- Рассматриваем на примере PostgreSQL

Хранимые процедуры

PostgreSQL

```
CREATE [ OR REPLACE ] FUNCTION
name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
[ RETURNS rettype |
  RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | WINDOW
  | IMMUTABLE
  | STABLE
  | VOLATILE
  | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT
  | RETURNS NULL ON NULL INPUT
  | STRICT
  | [ EXTERNAL ] SECURITY INVOKER
  | [ EXTERNAL ] SECURITY DEFINER
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
 | AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]
```

Хранимые процедуры

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;

SELECT add(20, 22) AS answer;

answer
------
42
```

Хранимые процедуры

```
CREATE OR REPLACE FUNCTION increment(i integer)
RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;

SELECT increment(41) AS answer;

answer
------
42
```

Хранимые процедуры

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);

f1 | f2
---+----------
42 |42 is text
```

Хранимые процедуры

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);

f1 | f2
---+-----------
42 |42 is text
```

Хранимые процедуры

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);

f1 | f2
---+----------
42 |42 is text
```

Хранимые процедуры

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);

foo
-----
60
```

Хранимые процедуры

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20);

foo
-----
33
```

Хранимые процедуры

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10);

foo
-----
15
```

Хранимые процедуры

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo();
```

Получаем ошибку, т.к. дефолтное значение для первого атрибута не задано

Хранимые процедуры

```
CREATE FUNCTION tf1 (accountno integer, debit numeric)
RETURNS numeric AS $$
    UPDATE bank
       SET balance = balance - debit
     WHERE accountno = tf1.accountno;
    SELECT balance
      FROM bank
     WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

Изменение процедуры

```
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) action [ ... ] [RESTRICT]
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) RENAME TO new_name
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) OWNER TO new_owner
ALTER FUNCTION name ([[argmode][argname] argtype [, ...]]) SET SCHEMA new_schema

action in:
    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE | [NOT] LEAKPROOF
    [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL

DROP FUNCTION [IF EXISTS] name ([[argmode][argname] argtype [, ...]])[CASCADE | RESTRICT]
```

Преимущества хранимых процедур

- Скорость
- Сокрытие структуры данных
- Гибкое управление правами доступа
- Меньшая вероятность SQL injection
- Повторное использование SQL
- Простая отладка SQL

Недостатки хранимых процедур

- Размазывание бизнес-логики
- Скудность языка СУБД
- Непереносимость хранимых функций
- Отсутствие необходимых навыков у команды и высокая «стоимость» соответствующих специалистов

Триггер

— хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по модификации данных: добавлением (INSERT), удалением (DELETE) строки в заданной таблице, или изменением (UPDATE) данных в определённом столбце заданной таблицы реляционной базы данных.

Триггер

- применяется для обеспечения целостности данных и реализации сложной бизнес-логики
- запускается сервером автоматически при попытке изменения данных в таблице, с которой он связан
- в случае обнаружения ошибки или нарушения целостности данных может произойти откат транзакции

Типы триггеров

- Уровень срабатывания
  - Row level – для каждой отдельной строки в таблице
  - Statement level – для всех строк одной инструкции
- Событие срабатывания
  - Update
  - Delete
  - Insert
- Время срабатывания
  - Before
  - After
  - Instead of

Назначение триггеров

- Реализация обновляемых представлений
- Реализация бизнес логики
- Вспомогательные расчеты
- Системные процессы
  - Репликация, например
    - Для тех СУБД, которые не умеют
- Всё, что угодно
  - Накопление истории
  - Логирование

Триггер

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables	Tables and views
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

Триггер

```
CREATE [CONSTRAINT] TRIGGER name {BEFORE | AFTER
| INSTEAD OF} { event [OR ...] }
    ON table
    [FROM referenced_table_name]
    [NOT DEFERRABLE | [DEFERRABLE] {INITIALLY
IMMEDIATE | INITIALLY DEFERRED}]
    [FOR [EACH] {ROW | STATEMENT} ]
    [WHEN (condition) ]
    EXECUTE PROCEDURE function_name (arguments)
```

Триггер

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

Триггер

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

Триггер

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM
NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

Триггер

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

Триггер

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE PROCEDURE view_insert_row();
```

```sql
CREATE TABLE emp (
  empname text PRIMARY KEY,
  salary integer
);

CREATE TABLE emp_audit(
  operation char(1) NOT NULL,
  userid text NOT NULL,
  empname text NOT NULL,
  salary integer,
  stamp timestamp NOT NULL
);

CREATE VIEW emp_view AS
SELECT e.empname,
       e.salary,
       max(ea.stamp) AS last_updated
  FROM emp e
  LEFT JOIN emp_audit ea
    ON ea.empname = e.empname
 GROUP BY e.empname,
       e.salary;
```

```sql
CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
 BEGIN
    -- Perform the required operation on emp, and create a row in emp_audit
    -- to reflect the change made to emp
    IF (TG_OP = 'DELETE') THEN
        DELETE FROM emp WHERE empname = OLD.empname;
        IF NOT FOUND THEN RETURN NULL; END IF;

        OLD.last_updated = now();
        INSERT INTO emp_audit VALUES('D', user, OLD.*);
        RETURN OLD;
    ELSIF (TG_OP = 'UPDATE') THEN
        UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
        IF NOT FOUND THEN RETURN NULL; END IF;

        NEW.last_updated = now();
        INSERT INTO emp_audit VALUES('U', user, NEW.*);
        RETURN NEW;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp VALUES(NEW.empname, NEW.salary);

        NEW.last_updated = now();
        INSERT INTO emp_audit VALUES('I', user, NEW.*);
        RETURN NEW;
    END IF;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
FOR EACH ROW
EXECUTE PROCEDURE update_emp_view();
```

Изменение триггера

```
ALTER TRIGGER name ON table RENAME TO new_name
```

```
ALTER TRIGGER emp_stamp ON emp RENAME TO
emp_track_chgs;
```

```
DROP TRIGGER [IF EXISTS] name ON table [CASCADE
| RESTRICT]
```

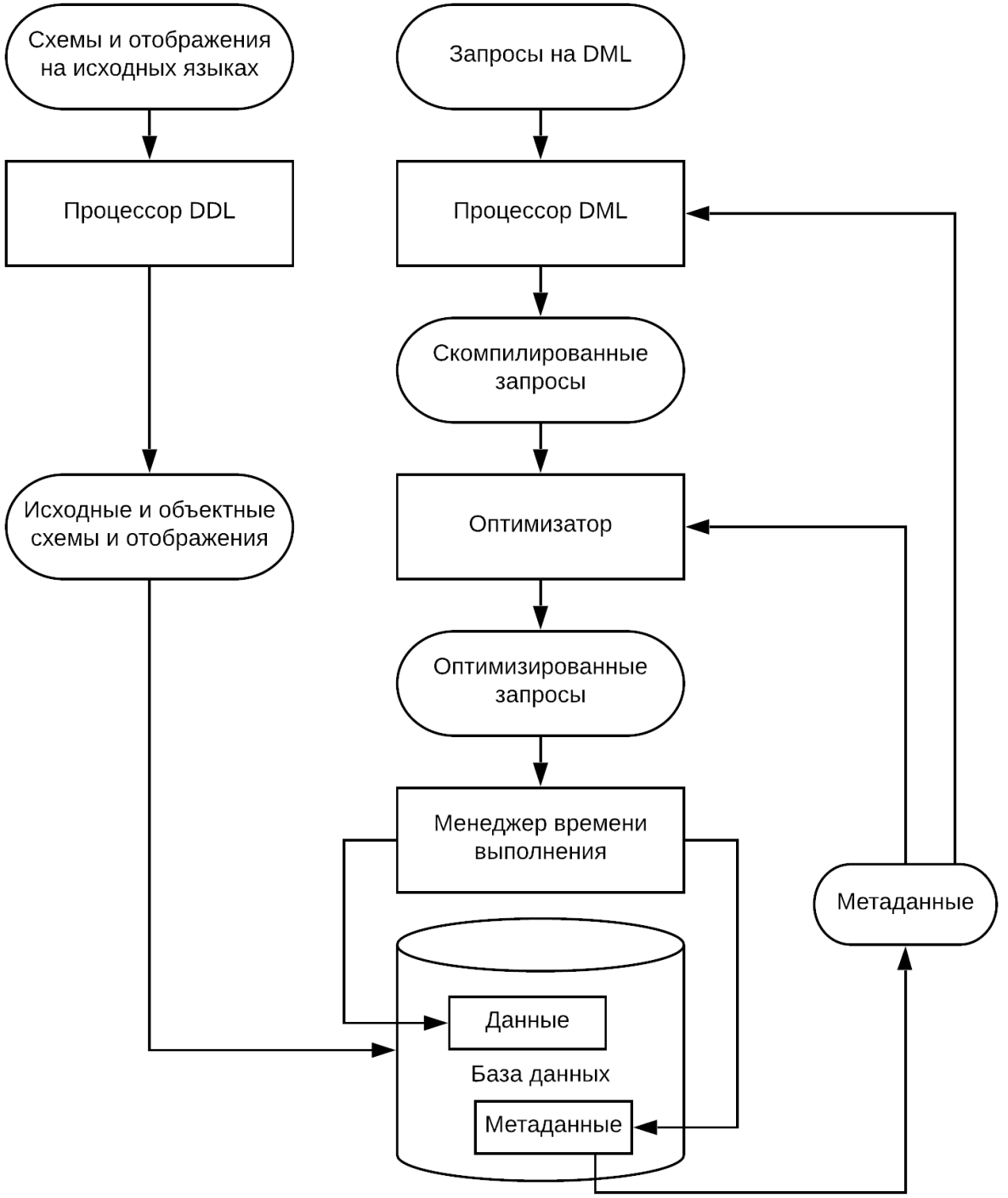
```
DROP TRIGGER if_dist_exists ON films;
```

Триггер

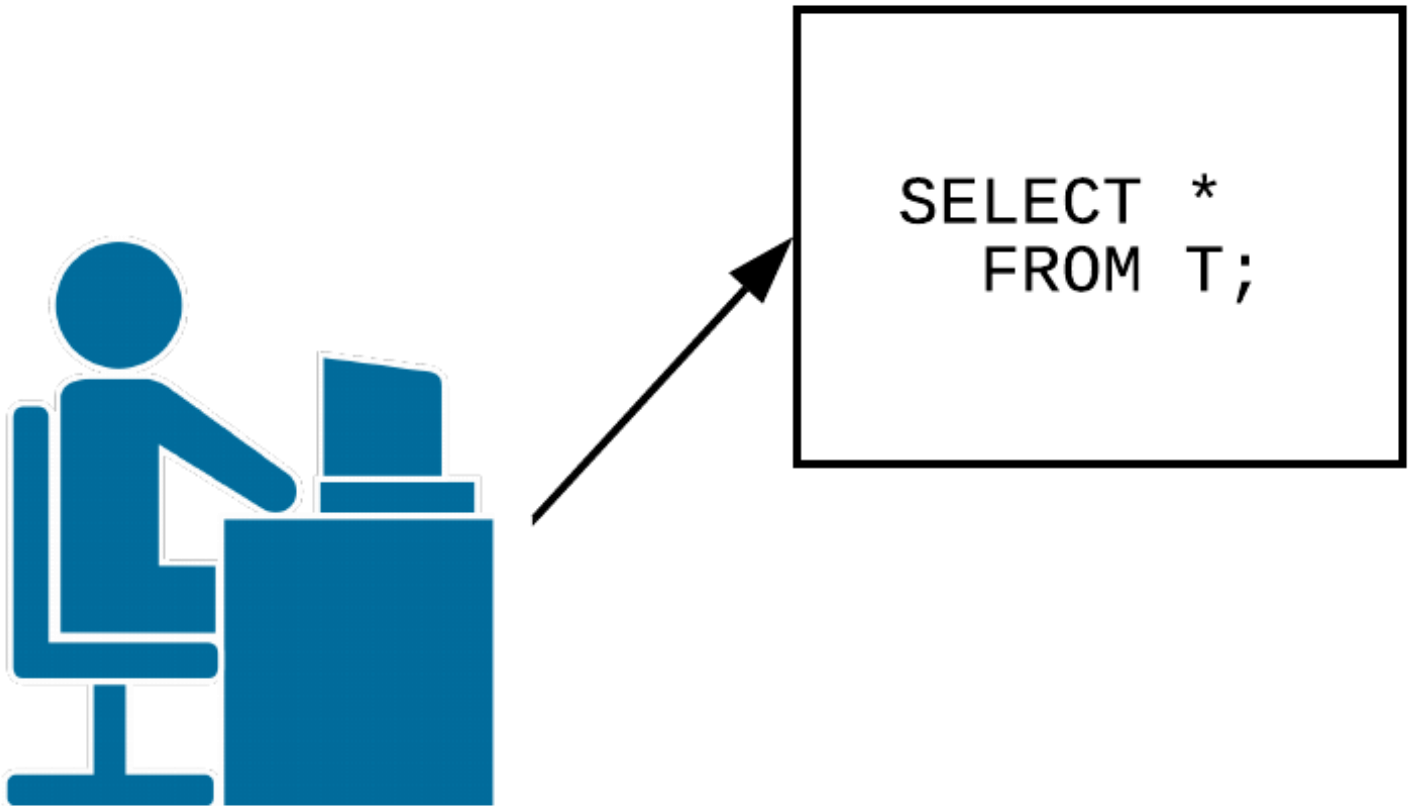
- Достоинства:
  - Реализация сложной, событийно-ориентированной логики
  - Сокрытие алгоритмов обработки
  - Возможность вносить корректировки в работы системы не затрагивая основные механизмы

- Недостатки:
  - При сложной схеме данных логика растягивается на множество триггеров
  - Увеличение числа зависимостей между объектами
  - Усложнение отладки

Архитектура СУБД



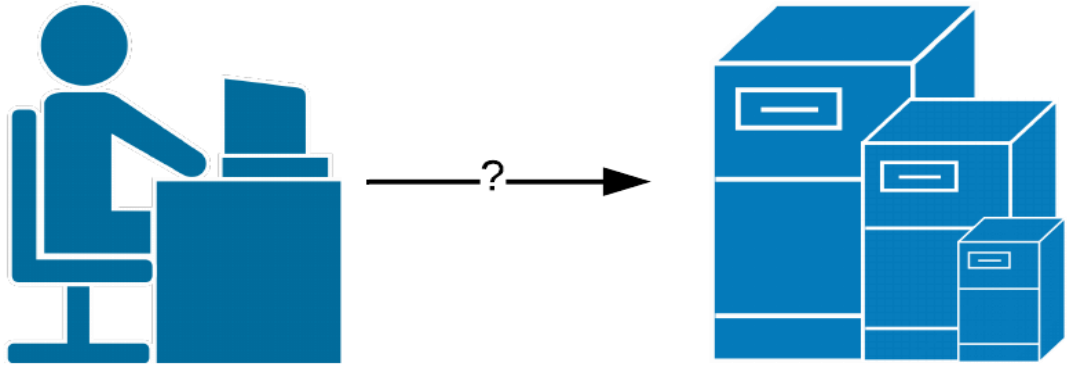
Вы написали запрос. Что дальше?



На примере PostgreSQL

1. Подключение к СУБД

1. Клиентский процесс обращается к главному серверному процессу
2. Главный серверный процесс создает новый при запросе соединения
3. Когда соединение установлено, клиент может отправить на сервер запрос
4. Запрос будет передан в виде обычного текста
5. На этом этапе обработки запроса не происходит



2. Парсинг запроса

1. Парсер проверяет запрос на правильность синтаксиса
2. Парсер возвращает итог работы:
   - Дерево разбора, если синтаксис корректный
   - Ошибка, если синтаксис некорректный. Пользователь получает ошибку
3. Преобразователь принимает на вход дерево разбора и преобразует его в дерево запроса:
   - Формируется понимание, какие таблицы, функции и операторы использует запрос
   - Обращение к базе только в рамках транзакций

Если очень интересно, как происходит разбор:

- *scan.l* – лексер, который отвечает за распознавание идентификаторов и ключевых слов SQL
- *gram.y* – набор правил грамматики и действий, которые запускаются при каждом вызове файла

Дерево запроса

Специальное внутренне представление SQL запросов с полным его разбором по ключевым параметрам:

- Тип команды (SELECT, UPDATE, DELETE, INSERT)
- Список используемых отношений
- Целевое отношение, в которое будет записан результат
- Список полей (* преобразуется в полный список всех полей)
- Список ограничений (которые указаны в WHERE)
- Список джоинов
- Прочие параметры, например, ORDER BY

Система правил

Что делаете вы:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

Что происходит на самом деле:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON
SELECT TO myview DO INSTEAD
 SELECT * FROM mytab;
```

3. Система переписывания запросов

1. На вход принимается дерево запроса
2. Обращения к представлениям заменяются на таблицы с использованием системы правил
3. Система возвращает обновленное дерево запроса

4. Планировщик / оптимизатор

Задача *планировщика/оптимизатора* — построить наилучший план выполнения:

- Один и тот же SQL-запрос может быть выполнен разными способами
- Результат их выполнения будет идентичен
- На разные способы тратится разное число ресурсов

Результат работы планировщика – план запроса

Подробности об оптимизации и планах запроса – на следующей лекции

5. Выполнение запроса

1. На выход принимает план (дерево) запроса
2. Обработчик рекурсивно обходит план запросы
3. При обращению к каждому узлу плана должны быть получены 1 или более строк, либо сообщение о том, что выдача строк завершена

Жизненный цикл запроса

1. Создается подключение к СУБД. В СУБД отправляется запрос
2. Парсер проверяет корректность синтаксиса запроса и создает дерево запроса
3. Система переписывания запросов преобразует запрос
4. Планировщик / оптимизатор создает план запроса
5. Обработчик рекурсивно обходит план и получает строки