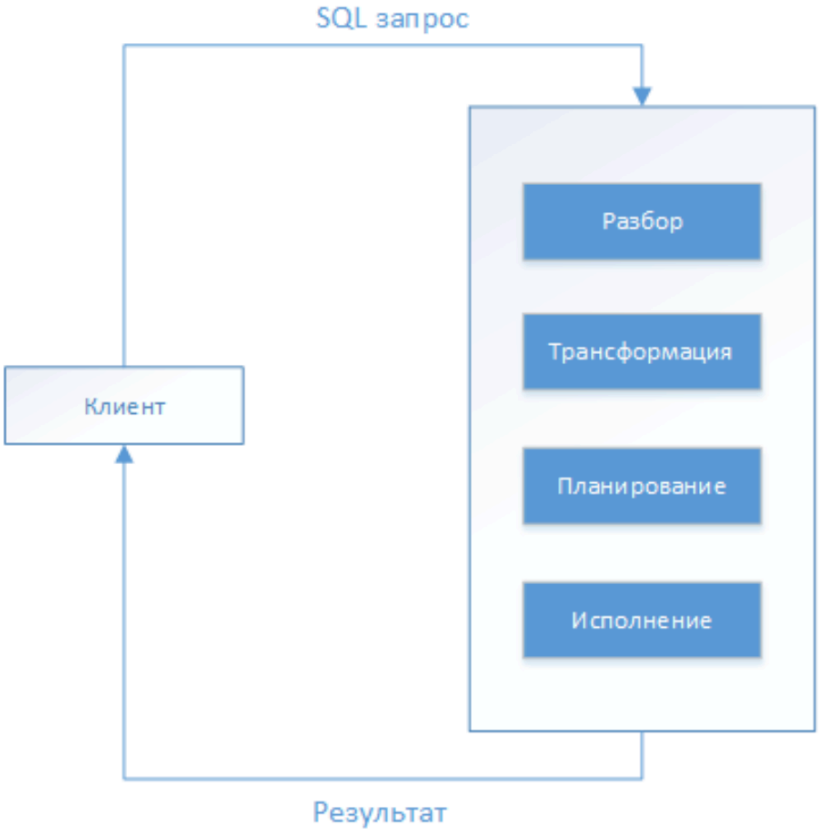
лаба 4

Как выполняется запрос?



1. Парсер (parser) преобразовывает текст на языке SQL в дерево разбора;
2. Анализатор запроса (analyzer) генерирует из дерева разбора начальный логический план запроса (реляционное выражение);
3. Система правил (rewriter) принимает разобранный запрос, одно дерево запроса, а также определённые пользователем правила перезаписи, представленные деревьями с некоторой дополнительной информацией, и создаёт определенное количество деревьев запросов;
4. Планировщик (planner) создает наиболее оптимальный план выполнения запроса на основе множества факторов;

5. Исполнитель (executor) выполняет запрос, обращаясь к таблицам и индексам в том порядке, который был создан деревом плана, и формирует результирующий набор строк, возвращаемый клиенту в виде серии сообщений.

Парсер

> 💡 Парсер генерирует дерево разбора, которое может быть прочитано последующими подсистемами из SQL-оператора в виде обычного текста.

Дерево разбора - это дерево, корневым узлом которого является структура SelectStmt.

При генерации дерева разбора парсер проверяет только синтаксис входного запроса. Поэтому он возвращает ошибку только в том случае, если в запросе есть синтаксическая ошибка.

Семантику входного запроса парсер не проверяет. Например, даже если запрос содержит несуществующее имя таблицы, парсер не сможет выдать соответствующую ошибку.

Анализатор запроса

> 💡 Анализатор выполняет семантический анализ дерева разбора, сгенерированного парсером, и генерирует дерево запросов.

Семантический разбор. Задача *семантического анализа* — определить, есть ли в базе данных таблицы и другие объекты, на которые запрос ссылается по имени, и есть ли у пользователя право обращаться к этим объектам. Вся необходимая для семантического анализа информация хранится в системном каталоге.

Семантический анализатор получает дерево разбора, дополняя ссылками на конкретные объекты базы данных, информацией о типах данных.

Корнем дерева запросов является структура Query, содержащая метаданные соответствующего запроса, такие как тип команды (SELECT, INSERT или другие), а также несколько листьев. Каждый лист образует список или дерево и содержит данные для каждого пункта.

Трансформация

На этапе трансформации происходит преобразование имён представлений в узлах дерева запроса в новые поддеревья, которые соответствуют этим представлениям. Для этого применяется система правил, которая ищет в системных каталогах подходящие правила и выполняет преобразование дерева запросов. Трансформация необходима для предоставления планировщику полной информации о таблицах и их связях между собой для построения оптимального плана выполнения запроса.

При необходимости система правил преобразует дерево запросов в соответствии с правилами, хранящимися в каталоге pg_rules .

Представления в PostgreSQL реализованы на основе системы правил. В случае, если представление задано командой CREATE VIEW, автоматически генерируется соответствующее правило, сохраняемое в каталоге.

Планирование

Задача планировщика — построить наилучший план выполнения.

На этапе планирования запроса выполняется поиск наиболее оптимального способа выполнения нашего запроса. Для этого планировщик осуществляет перебор возможных планов выполнения, оценивает затраты на исполнение и присваивает каждому определённую стоимость. Оценка затрат основывается на математической модели с использованием статистики об обрабатываемых данных, которая накапливается в процессе анализа и при выполнении некоторых DDL-операций, а также уточняется при очистке.

В первую очередь планировщик вырабатывает планы сканирования для таблиц с учётом наличия у них индексов, для которых могут создаваться отдельные планы. Далее происходит выработка планов соединения таблиц, при этом планировщик рассматривает все возможные способы соединения и выбирает самый выгодный. Существует три основных стратегии соединения: соединение с вложенным циклом, соединение слиянием и соединение по

хэшу. В конечном итоге планировщик формирует дерево плана, состоящее из отдельных операций. Ниже приведены основные из них:

- Seq Scan – последовательное сканирование;
- Index Scan – индексное сканирование;
- Merge Join – соединение слиянием;
- Nested Loop – соединение с вложенным циклом;
- Hash Join – соединение по хэшу;
- Sort – операция сортировки по столбцам;
- Aggregate – вычисление агрегатных функций;

Исполнение

Исполнение плана выполнения начинается с создания в памяти обслуживающего процесса объекта, который хранит в себе состояние выполняющего запроса в виде дерева, повторяющего структуру плана. Такой объект называется порталом, а его работа происходит по принципу конвейера. Выполнение запроса происходит рекурсивно, каждый корневой узел дерева обращается к дочерним узлам для получения входных данных, далее он производит необходимые преобразования и расчёты, назначенные ему планировщиком, и передаёт их вверх по дереву.

В конце концов исполнитель формирует результирующий набор строк, который возвращается клиенту в виде серии сообщений. Например, для простых запросов на выборку данных PostgreSQL отправляет сообщение RowDescription, описывающее структуру столбцов. Далее СУБД для каждой строки результирующего набора формирует сообщений DataRow и отправляет их последовательно клиенту. Завершается передача результатов выборки сообщением CommandComplete. По готовности безопасно принимать новые команды, PostgreSQL отправляет клиенту сообщение ReadyForQuery.

Дальше клиент отправляет серверу сообщение Terminate и закрывает соединение. Со своей стороны, СУБД также закрывает подключение и завершает обслуживающий процесс.

Nested Loops Join

Nested Loops Join (соединение вложенных циклов) сравнивает каждую строку одной таблицы (называемой внешней таблицей) с каждой строкой другой таблицы (называемой внутренней таблицей), ища те строки, которые удовлетворяют предикату соединения.

Hash Join

Хэш-соединение выполняется в две стадии: компоновка и проба. Во время компоновки осуществляется чтение всех строк первого входного потока, хеширование строк по ключам соединения эквивалентности и создание в оперативной памяти хеш-таблицы. Во время пробы осуществляется чтение всех строк второго входного потока (часто его называют правым потоком или пробным потоком), хеширование строк этого потока по тем же ключам соединения эквивалентности, а потом осуществляется просмотр или поиск соответствий строк в хеш-таблице

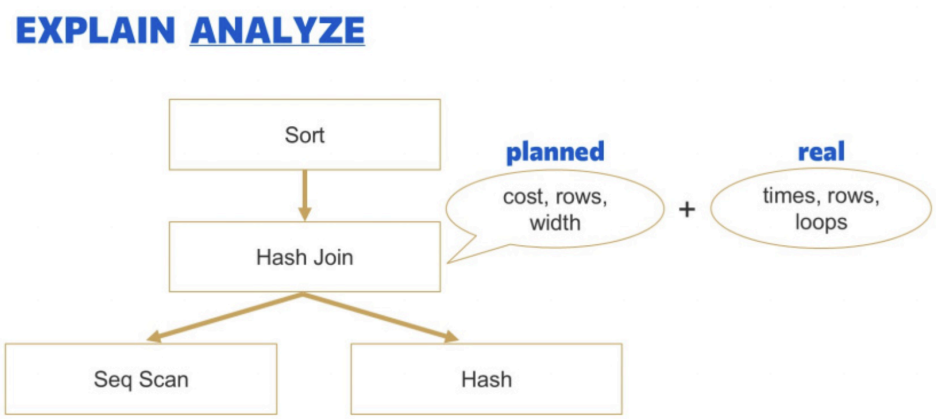
EXPLAIN и EXPLAIN ANALYZE

Команда EXPLAIN нам распечатывает дерево с планом запроса. В каждом узле дерева – операция, которая отвечает за выполнение своего кусочка запроса. Вышестоящие операции используют результаты нижестоящих операций.

В каждом узле плана у нас будет сохранена некоторая информация, такая как:

- cost – оценочная стоимость операции
- rows – количество строк, которые будут обработаны;
- width – ширина.

EXPLAIN мы запускаем до исполнения запроса, но если мы добавим флаг ANALYZE, тогда наш запрос все-таки будет выполнен.



К информации, которую планировщик предполагал, добавится информация о том, как на самом деле запрос был выполнен:

- times – сколько времени мы потратили на каждом этапе;
- rows – сколько строк получили;
- loops – сколько циклов прошли.

Индексы

https://timeweb.cloud/tutorials/sql/indeksy-v-sql-sozdanie-vidy-i-kak-rabotayut

Индекс - это объект, который создается для одного или более столбцов в таблицах базы данных с целью повышения производительности, а именно для ускорения поиска и извлечения требуемых данных из базы данных

Индекс состоит из **ключей**. Ключи построены из одного или более столбцов в таблице. Сами ключи хранятся в виде структуры сбалансированного дерева (древовидная структура, предназначенная для быстрого доступа к данным). Данная структура берет свое начало с корневого узла на вершине иерархии и далее продолжается в виде конечных узлов, которые располагаются в нижней части сбалансированного дерева

B-Tree индекс

Наиболее распространённая структура индексов в реляционных БД. Эффективен для точного поиска, сортировки и диапазонных запросов.

- **Пример**:

    - Индекс по столбцу `name` в таблице `users` — дерево, где каждое имя связано с соответствующими строками
    - Индекс по дате создания `created_at` в таблице `orders` ускоряет поиск заказов за определённый период
- Подходит для:
    - Точного поиска (`WHERE user_id = 123`)
    - Диапазонных запросов (`WHERE age BETWEEN 18 AND 30`, `WHERE date > '2023-01-01'`)
    - Сортировки (`ORDER BY last_name`)

Hash-индекс

Использует хеш-функцию для отображения ключей. Обеспечивает быстрый доступ по точному совпадению, но не поддерживает диапазонные запросы.

- Пример:
    - Индекс по `email` в таблице `users` позволяет мгновенно найти пользователя (`WHERE email = 'user@example.com'`)
- Подходит для:
    - Точного поиска (`=`, `IN`), но не для `>`, `<`, `BETWEEN` или `ORDER BY`

Составной (композитный) индекс

Включает несколько столбцов. Эффективен, если запросы часто используют несколько полей одновременно.

- Пример:
    - Индекс по `(department_id, salary)` ускоряет запросы вида:

      ```
      WHERE department_id = 5 AND salary > 50000
      ```

    - Индекс по `(last_name, first_name)` улучшает поиск по фамилии и имени
- Подходит для:
    - Запросов с несколькими условиями (`WHERE a = 1 AND b = 2`)
    - Сортировки по нескольким столбцам (`ORDER BY date, time`)

Уникальный индекс (UNIQUE)

Гарантирует **уникальность значений** в столбце или комбинации столбцов.

По умолчанию значения NULL в уникальном столбце считаются не равными друг другу, так что в таком столбце может быть несколько значений NULL

- **Пример**:
  - Уникальный индекс по **email** запрещает регистрацию двух пользователей с одинаковым адресом

```
CREATE UNIQUE INDEX idx_users_email ON users(email);
```

  - Составной уникальный индекс по **(username, domain)** в таблице **email_accounts**

Запретит дублирующие комбинации username+domain

```
CREATE TABLE email_accounts (
    id SERIAL PRIMARY KEY,
    username TEXT,
    domain TEXT,
    UNIQUE (username, domain)
);
```

- **Подходит для**:
  - Обеспечения уникальности (**PRIMARY KEY**, **UNIQUE CONSTRAINT**)

Когда для таблицы определяется ограничение уникальности или первичный ключ, PostgreSQL автоматически создаёт уникальный индекс по всем столбцам, составляющим это ограничение или первичный ключ (индекс может быть составным).

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email TEXT UNIQUE
);
```

Преимущества:

- **Ускорение запросов**: Особенно полезно для запросов на выборку данных.
- **Повышение производительности соединений (JOIN)**: Индексы могут ускорить выполнение запросов, которые объединяют несколько таблиц.

Недостатки:

- **Замедление операций записи**: Индексы требуют дополнительного времени для обновления при добавлении, удалении или изменении строк в таблице.
- **Дополнительное место на диске**: Индексы занимают место в памяти и на диске.

Создание индексов

Чтобы создать индекс в базе данных, существует команда `CREATE INDEX`.

Рассмотрим практические примеры создания индексов в PostgreSQL. Общий синтаксис создания индексов следующий:

```
CREATE INDEX <имя_индекса> ON <название_таблицы> (<имя_столбца1>, <имя_столбца2>);
```

Например, создадим индекс для таблицы `orders` для столбца `order_id`:

```
CREATE INDEX index_for_order ON orders (order_id);
```

Вот так создаются b-tree и hash индексы

```
CREATE INDEX person_name_ind ON "Н_ЛЮДИ" USING btree("ИМЯ");

CREATE INDEX education_NZK_ind ON "Н_ОБУЧЕНИЯ" USING btree("НЗК");
```

Создавать индексы можно для двух столбцов и более. Для этого необходимо указать их через запятую:

```
CREATE INDEX index_for_order ON orders (order_id, client_id);
```

Чтобы создать уникальный индекс, необходимо использовать ключевое слово `UNIQUE`:

```
CREATE UNIQUE INDEX index_for_order ON orders (order_id);
```