Практические работы по дисциплине «Технологии обработки транзакций клиент-серверных приложений» направления подготовки бакалавриата 09.03.04 «Программная инженерия»

Практическая работа №10

Транзакции. Оптимизация запросов

Теория для понимания практики:

Темы:

- Общие подходы к оптимизации
- Простые запросы и этапы их обработки
- Расширенные запросы
- Подробнее о планировании

Подходы к оптимизации

Настройка параметров

- подстройка под имеющуюся нагрузку
- глобальное влияние на всю систему
- мониторинг

Оптимизация запросов

- уменьшение нагрузки
- локальное воздействие (запрос или несколько запросов)
- профилирование

Эта практика посвящена оптимизации запросов. Вообще «оптимизация» — очень широкое понятие; задумываться об оптимизации необходимо еще на этапе проектирования системы и выбора архитектуры. В данной практике рассматриваются о те работы, которые выполняются при эксплуатации уже существующего приложения.

Можно выделить два основных подхода. Первый состоит в том, чтобы отслеживать состояние системы и добиваться того, чтобы она справлялась с имеющейся нагрузкой. Для этого можно настраивать параметры СУБД, а также настраивать операционную систему. Если настройки не помогают, при таком подходе остается только модернизировать аппаратуру (что тоже помогает не всегда).

Другой подход, который в основном мы и будем рассматривать далее,

состоит в том, чтобы не приспосабливаться под нагрузку, а уменьшать ее. «Полезная» нагрузка формируется запросами. Если удается найти узкое место, то можно попробовать тем или иным способом повлиять на выполнение запроса и получить тот же результат, потратив меньше ресурсов. Такой способ действует более локально (на отдельный запрос или ряд запросов), но уменьшение нагрузки благоприятно сказывается и на работе всей системы.

Мы начнем с того, что в деталях разберем механизмы выполнения запросов, а уже после этого поговорим о том, как распознать неэффективную работу и о конкретных способах воздействия.

Этапы обработки запроса

- Разбор
- Переписывание (трансформация)
- Планирование (оптимизация)
- Выполнение

Сначала рассмотрим, как выполняется запрос в простом случае – например, если в psql написать команду SELECT.

Разбор

Рисунок 1 – Этап разбора

Обработка обычного запроса выполняется в несколько этапов.

Во-первых, запрос разбирается (parse).

Сначала производится синтаксический разбор: текст запроса представляется в виде дерева — так с ним удобнее работать.

Затем выполняется семантический анализ, в ходе которого определяется, на какие объекты БД ссылается запрос и есть ли у пользователя доступ к ним (для этого анализатор заглядывает в системный каталог).

Синтаксический разбор

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

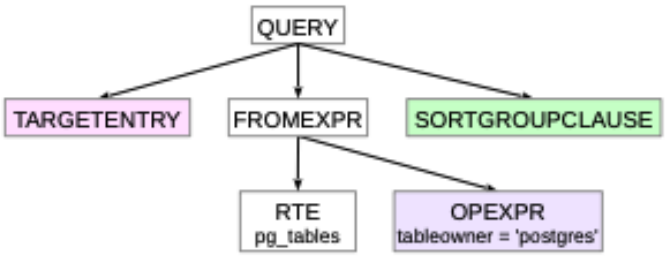


Рисунок 2 – пример синтаксического разбора

Рассмотрим простой пример: запрос, приведенный на рисунке.

На этапе синтаксического разбора в памяти серверного процесса будет построено дерево, упрощенно показанное на рисунке ниже запроса. Цветом показано примерное соответствие частей текста запроса и узлов дерева.

RTE — неочевидное сокращение от Range Table Entry. Этим именем внутри PostgreSQL называются таблицы, подзапросы, результаты соединений — иными словами, наборы строк, с которыми может оперировать SQL.

Если любопытно, то реальное дерево можно увидеть, установив параметр debug_print_parse и заглянув в журнал сообщений сервера. Практического смысла в этом нет (если, конечно, вы не разработчик ядра PostgreSQL).

Семантический разбор

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

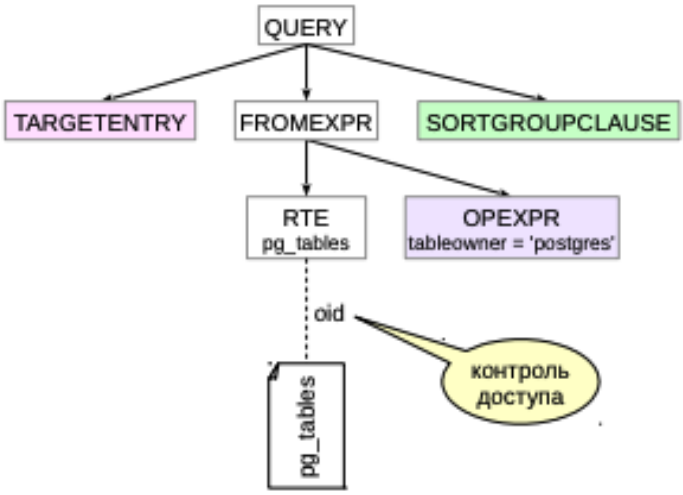


Рисунок 3 – пример семантического разбора

На этапе семантического разбора анализатор сверяется с системным каталогом и связывет имя «pg_tables» с представлением, имеющим определенный идентификатор (oid) в системном каталоге. Также будут проверены права доступа к этому представлению.

Переписывание

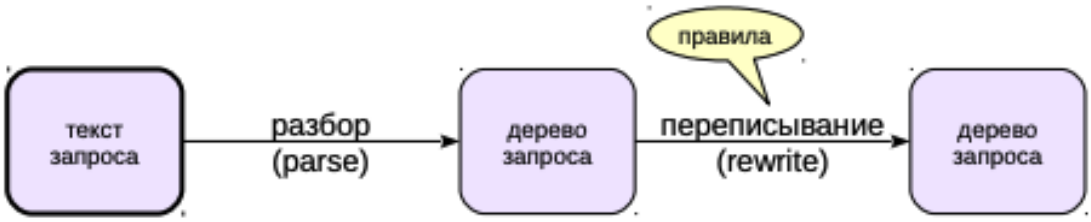


Рисунок 4 – этап переписывания

Во-вторых, запрос переписывается или трансформируется (rewrite) с учетом правил. Важный частный случай — подстановка текста запроса вместо имени представления. Заметим, что текст представления опять необходимо разобрать, поэтому мы несколько упрощаем, говоря, что первые два этапа происходят друг за другом.

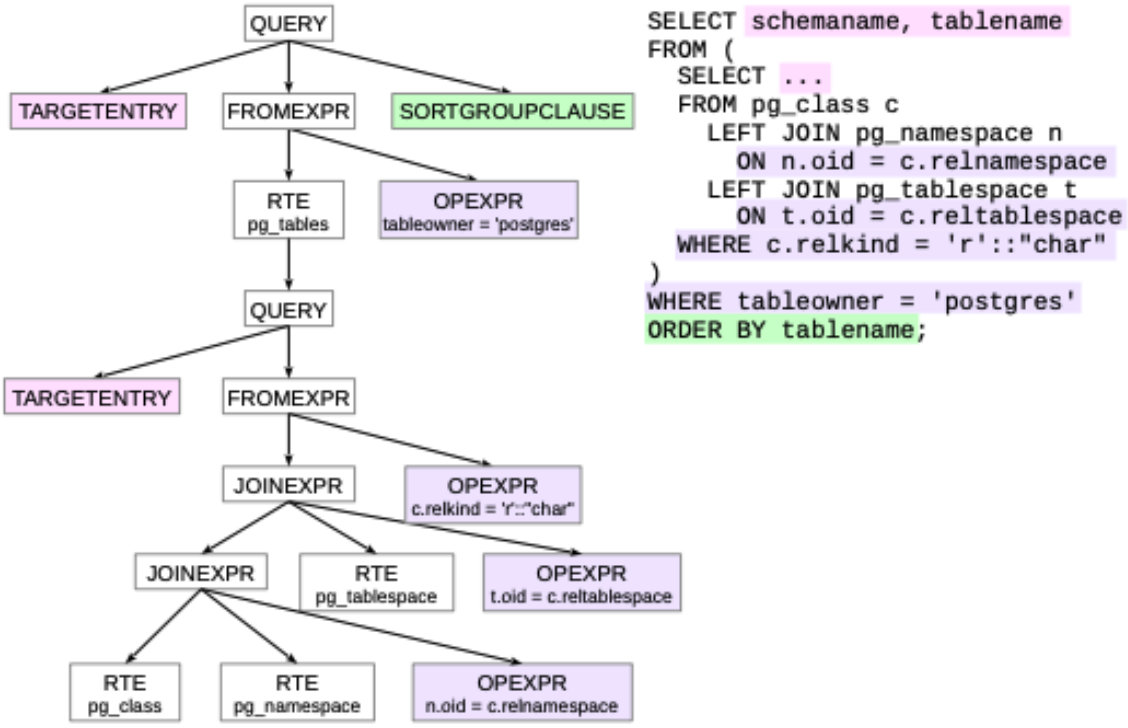


Рисунок 5 – условный разбор запроса

На рисунке приведен запрос с подставленным текстом представления (это условность: реально в таком виде запрос не существует — вся работа по переписыванию происходит только с деревом запроса).

Поддерево, соответствующее подзапросу, имеет своим родителем узел, ссылающийся на представление. На рисунке в этом поддереве хорошо видна древовидная структура запроса.

Трансформированное дерево можно увидеть в журнале сообщений сервера, установив параметр debug_print_rewritten.

Планирование

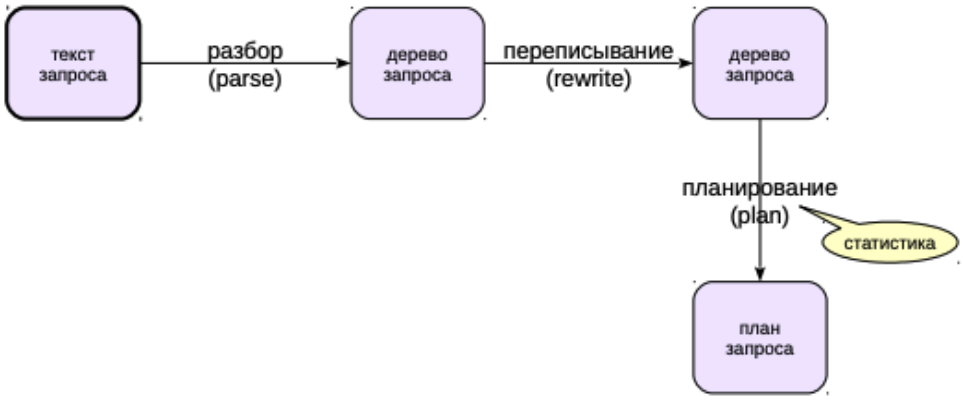


Рисунок 6 – этап планирования

В-третьих, запрос планируется (plan). S

QL — декларативный язык, и один запрос можно выполнить разными способами. Планировщик (он же оптимизатор) перебирает различные способы выполнения и оценивает их. Оценка дается на основе некоторой математической модели исходя из информации об обрабатываемых данных (статистики).

Тот способ выполнения, для которого прогнозируется минимальная стоимость, представляется в виде дерева плана.

```
Sort  (cost=19.59..19.59 rows=1 width=128)
  Sort Key: c.relname
  ->  Nested Loop Left Join  (cost=0.00..19.58 rows=1 width=128)
        Join Filter: (n.oid = c.relnamespace)
        ->  Seq Scan on pg_class c  (cost=0.00..18.44 rows=1 width=72)
              Filter: ((relkind = 'r'::"char") AND
                       (pg_get_userbyid(relowner) = 'postgres'::name))
        ->  Seq Scan on pg_namespace n  (cost=0.00..1.06 rows=6 width=68)
```

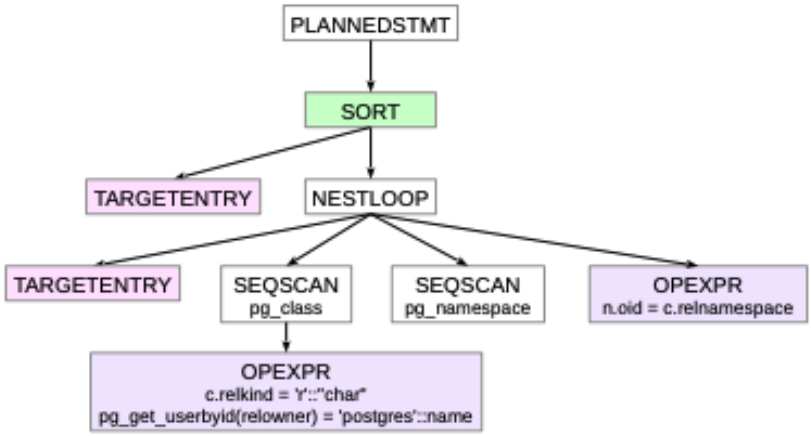


Рисунок 7 – пример дерева плана

На рисунке приведен пример дерева плана, которое представляет способ выполнения запроса.

Здесь операции Seq Scan — это чтение соответствующих таблиц, а Nested Loop — способ соединения двух таблиц. В виде текста на рисунке приведен план выполнения в том виде, как его показывает команда EXPLAIN. Подробно о методах доступа к данным, о способах соединения и об EXPLAIN мы будем говорить в следующих темах.

Пока имеет смысл обратить внимание на два момента:

- из трех таблиц осталось только две: планировщик сообразил, что одна из таблиц не нужна для получения результата и ее можно

безболезненно удалить из дерева плана;

- каждый узел дерева снабжен информацией о предполагаемом числе строк (rows) и о стоимости (cost).

Для интересующихся — увидеть «настоящее» дерево плана можно, установив параметр debug_print_plan.

Выполнение

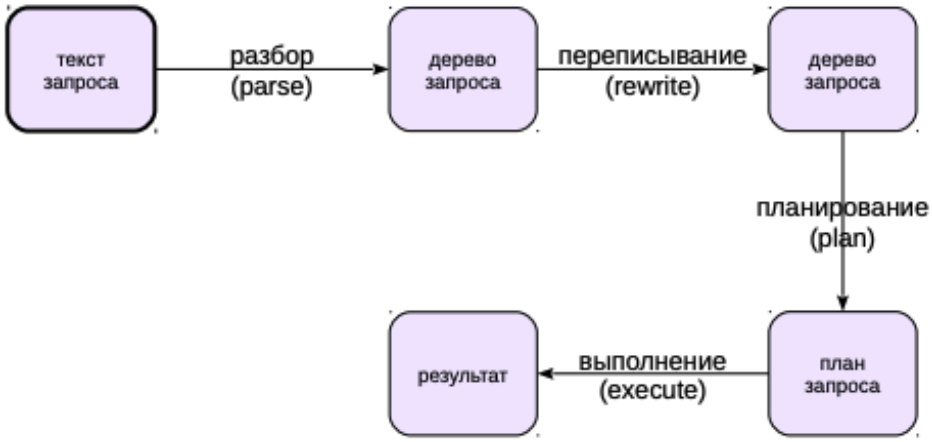


Рисунок 8 – этап выполнения

В-четвертых, запрос выполняется (execute) в соответствии с выбранным планом, и результат возвращается клиенту

Путь запроса

Ниже мы кратко опишем этапы, которые проходит запрос для получения результата.

1. Прикладная программа устанавливает подключение к серверу PostgreSQL. Эта программа передаёт запрос на сервер и ждёт от него результатов.
2. На этапе разбора запроса сервер выполняет синтаксическую проверку запроса, переданного прикладной программой, и создаёт дерево запроса.
3. Система правил принимает дерево запроса, созданное на стадии разбора, и ищет в системных каталогах правила для применения к этому дереву. Обнаружив подходящие правила, она выполняет преобразования, заданные в теле, правил.
   Одно из применений системы правил заключается в реализации представлений. Когда выполняется запрос к представлению (т. е. виртуальной таблице), система правил преобразует запрос пользователя в запрос, обращающийся не к представлению, а к базовым таблицам из определения представления.
4. Планировщик/оптимизатор принимает дерево запроса (возможно, переписанное) и создаёт план запроса, который будет передан

исполнителю.
Он выбирает план, сначала рассматривая все возможные варианты получения одного и того же результата. Например, если для обрабатываемого отношения создан индекс, прочитать отношение можно двумя способами. Во-первых, можно выполнить простое последовательное сканирование, а во-вторых, можно использовать индекс. Затем оценивается стоимость каждого варианта и выбирается самый дешёвый. Затем выбранный вариант разворачивается в полноценный план, который сможет использовать исполнитель.

5. Исполнитель рекурсивно проходит по дереву плана и получает строки тем способом, который указан в плане. Он сканирует отношения, обращаясь к системе хранения, выполняет сортировку и соединения, вычисляет условия фильтра и, наконец, возвращает полученные строки.

Выполнение

Конвейер

- обход дерева от корня вниз
- данные передаются наверх — по мере поступления или все сразу

Доступ к данным

- чтение таблиц, индексов

Соединения

- всегда попарно
- важен порядок

Другие операции

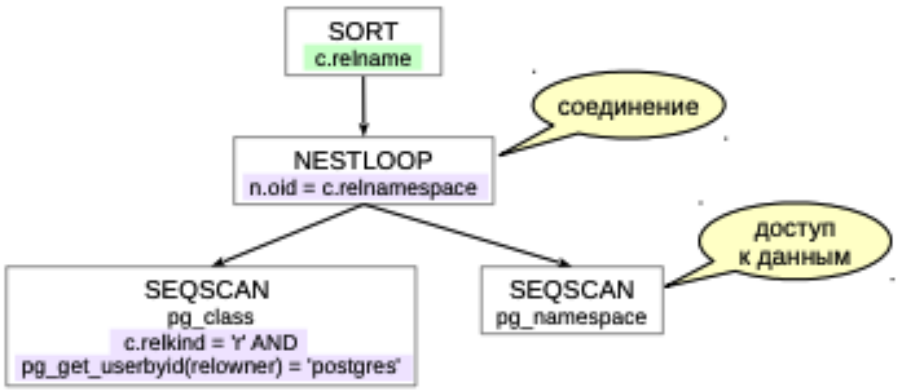


Рисунок 9 – упрощенное дерево плана

На этапе выполнения дерево плана (на рисунке оно перерисовано так, чтобы показать только самое основное) работает как конвейер.

Выполнение начинается с корня дерева. Корневой узел (в нашем случае это операция сортировки SORT) обращается за данными к нижестоящему узлу; получив данные, он выполняет свою работу (сортировку) и отдает данные наверх (то есть клиенту).

Некоторые узлы (как NESTLOOP на рисунке) соединяют данные, полученные из разных источников. Здесь узел обращается по очереди к двум дочерним узлам (соединение всегда осуществляется попарно) и, получив от них строки, соединяет их и возвращает наверх (то есть узлу сортировки).

Два нижних узла представляют обращение к таблице за данными. Они читают строки из соответствующих таблиц и отдают наверх (то есть узлу соединения).

Некоторые узлы могут вернуть результат, только получив от нижестоящих узлов все данные. К таким узлам относится сортировка — нельзя отсортировать неполную выборку. Другие узлы могут возвращать данные по мере поступления. Например, доступ к данным с помощью чтения таблицы может отдавать наверх данные по мере чтения (это позволяет быстро получить первую часть результата — например, для страничного отображения на веб-странице).

Чтобы разобраться с планами выполнения, нужно понять, какие существуют методы доступа к данным, какие есть способы соединения этих данных, и посмотреть некоторые другие операции.

Расширенные запросы

Возможности:

- Уточнение схемы обработки запроса
- Подготовленные операторы
- Курсоры

PostgreSQL также предусматривает протокол расширенных запросов. На практике это означает возможность использования подготовленных операторов и курсоров.

Простые запросы

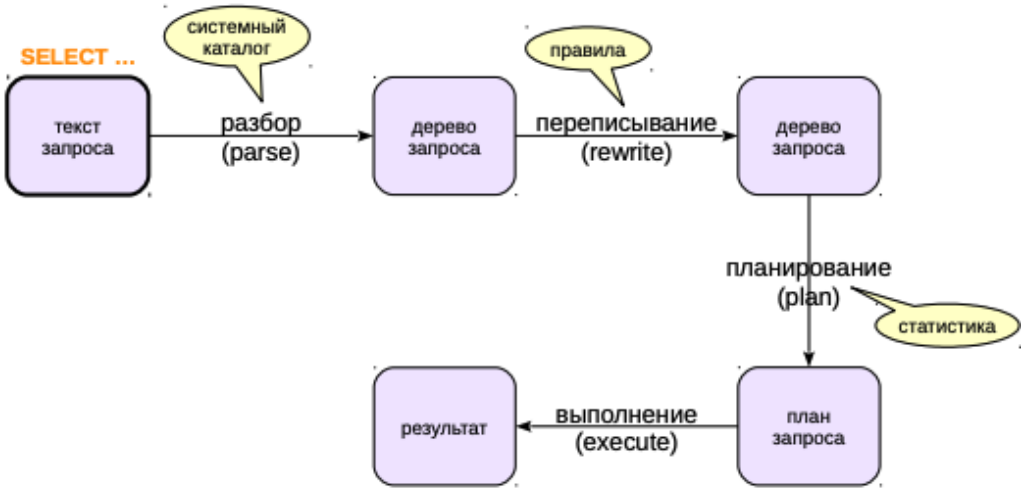
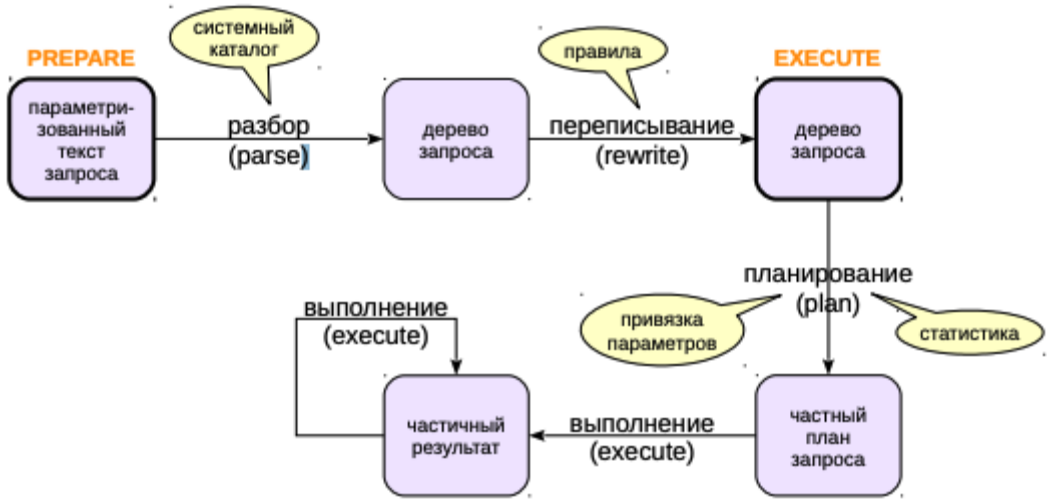


Рисунок 10 – схема обработки простых запросов

На рисунке повторно приведена полная схема обработки простых запросов, которую мы уже рассмотрели.

Расширенный протокол



курсоры

подготовленные операторы

Рисунок 11 – схема обработки с использованием возможностей расширенного протокола

Протокол расширенных запросов разбивает протокол простых запросов на несколько шагов. Результаты подготовительных шагов можно неоднократно использовать повторно для улучшения эффективности. Кроме того, он открывает дополнительные возможности, в частности, возможность передавать значения данных в отдельных параметрах вместо того, чтобы внедрять их непосредственно в строку запроса.

В расширенном протоколе клиент сначала передаёт сообщение Parse с текстовой строкой запроса и, возможно, некоторыми сведениями о типах параметров и именем целевого объекта подготовленного оператора (если имя пустое, создаётся безымянный подготовленный оператор). Ответом на это сообщение будет ParseComplete или ErrorResponse. Типы параметров указываются по OID; при отсутствии явного указания анализатор запроса пытается определить типы данных так же, как он делал бы для нетипизированных строковых констант.

Примечание. Тип данных параметра можно оставить неопределённым, задав для него значение ноль, либо сделав массив с OID типов параметров короче, чем набор символов параметров ($***n***), используемых в строке запроса. Другой особый случай — передача типа параметра как void (то есть передача OID псевдотипа void). Это предусмотрено для того, чтобы символы параметров можно было использовать для параметров функций, на самом деле представляющих собой параметры OUT. Обычно параметр void нельзя использовать ни в каком контексте, но если такой параметр фигурирует в списке параметров функции, он фактически игнорируется. Например, вызову функции foo($1,$2,$3,$4) может соответствовать функция с аргументами IN и двумя OUT, если аргументы $3 и $4 объявлены как имеющие тип void.

Примечание. Строка запроса, содержащаяся в сообщении Parse, не может содержать больше одного оператора SQL; иначе выдаётся синтаксическая ошибка. Это ограничение отсутствует в протоколе простых запросов, но присутствует в расширенном протоколе, так как добавление поддержки подготовленных операторов или порталов, содержащих несколько команд, неоправданно усложнило бы протокол.

В случае успешного создания именованный подготовленный оператор продолжает существовать до завершения текущего сеанса, если только он не будет уничтожен явно. Безымянный подготовленный оператор сохраняется только до следующей команды Parse, в которой целевым является безымянный оператор. (Заметьте, что сообщение простого запроса также уничтожает

безымянный оператор.) Именованные операторы должны явно закрываться, прежде чем их можно будет переопределить другим сообщением Parse, но для безымянных операторов это не требуется. Именованные подготовленные операторы также можно создавать и вызывать на уровне команд SQL, используя команды PREPARE и EXECUTE.

Когда подготовленный оператор существует, его можно подготовить к выполнению сообщением Bind. В сообщении Bind задаётся имя исходного подготовленного оператора (пустая строка подразумевает безымянный подготовленный оператор), имя целевого портала (пустая строка подразумевает безымянный портал) и значения для любых шаблонов параметров, представленных в подготовленном операторе. Набор передаваемых значений должен соответствовать набору параметров, требующихся для подготовленного оператора. (Если вы объявили параметры void в сообщении Parse, передайте для них значения NULL в сообщении Bind.) Bind также принимает указание формата для данных, возвращаемых в результате запроса; формат можно указать для всех данных, либо для отдельных столбцов. Ответом на это сообщение будет BindComplete или ErrorResponse.

Примечание. Выбор между текстовым и двоичным форматом вывода определяется кодами формата, передаваемыми в Bind, вне зависимости от команды SQL. При использовании протокола расширенных запросов атрибут BINARY в объявлении курсоров не имеет значения.

Планирование запроса обычно имеет место при обработке сообщения Bind. Если подготовленный оператор не имеет параметров, либо он выполняется многократно, сервер может сохранить созданный план и использовать его повторно при последующих сообщениях Bind для того же подготовленного оператора. Однако он будет делать это, только если решит, что можно получить универсальный план, который не будет значительно неэффективнее планов, зависящих от конкретных значений параметров. С точки зрения протокола это происходит незаметно.

В случае успешного создания объект именованного портала продолжает существование до конца текущей транзакции, если только он не будет уничтожен явно. Безымянный портал уничтожается в конце транзакции или при выполнении следующей команды Bind, в которой в качестве целевого выбирается безымянный портал. (Заметьте, что сообщение простого запроса также уничтожает безымянный портал.) Именованные порталы должны явно закрываться, прежде чем их можно будет явно переопределить другим сообщением Bind, но это не требуется для безымянных порталов. Именованные порталы также можно создавать и вызывать на уровне команд SQL, используя команды DECLARE CURSOR и FETCH.

Когда портал существует, его можно запустить на выполнение сообщением Execute. В сообщении Execute указывается имя портала (пустая строка подразумевает безымянный портал) и максимальное число результирующих строк (ноль означает «выбрать все строки»). Число результирующих строк имеет значение только для порталов, которые содержат команды, возвращающие наборы строк; в других случаях команда всегда выполняется до завершения и число строк игнорируется. В ответ на Execute могут быть получены те же сообщения, что описаны выше для запросов, выполняемых через протокол простых запросов, за исключением того, что после Execute не выдаются сообщения ReadyForQuery и RowDescription.

Если операция Execute оканчивается до завершения выполнения портала (из-за достижения ненулевого ограничения на число строк), сервер отправляет сообщение PortalSuspended; появление этого сообщения говорит клиенту о том, что для завершения операции с данным порталом нужно выдать ещё одно сообщение Execute. Сообщение CommandComplete, говорящее о завершении исходной команды SQL, не передаётся до завершения выполнения портала. Таким образом, фаза Execute всегда заканчивается при появлении одного из сообщений: CommandComplete, EmptyQueryResponse (если портал был создан из пустой строки запроса), ErrorResponse или PortalSuspended.

В конце каждой серии сообщений расширенных запросов клиент должен выдать сообщение Sync. Получив это сообщение без параметров, сервер закрывает текущую транзакцию, если команды выполняются не внутри блока транзакции BEGIN/COMMIT (под «закрытием» понимается фиксация при отсутствии ошибок или откат в противном случае). Затем он выдаёт ответ ReadyForQuery. Целью сообщения Sync является обозначение точки синхронизации для восстановления в случае ошибок. Если при обработке сообщений расширенных запросов происходит ошибка, сервер выдаёт ErrorResponse, затем считывает и пропускает сообщения до Sync, после чего выдаёт ReadyForQuery и возвращается к обычной обработке сообщений. (Но заметьте, что он не будет пропускать следующие сообщения, если ошибка происходит в процессе обработки Sync — это гарантирует, что для каждого Sync будет передаваться в точности одно сообщение ReadyForQuery.)

Примечание. Сообщение Sync не приводит к закрытию блока транзакции, открытого командой BEGIN. Выявить эту ситуацию можно, используя информацию о состоянии транзакции, содержащуюся в сообщении ReadyForQuery.

В дополнение к этим фундаментальным и обязательным операциям, протокол расширенных запросов позволяет выполнить и несколько дополнительных операций.

В сообщении Describe (в вариации для портала) задаётся имя существующего портала (пустая строка обозначает безымянный портал). В ответ передаётся сообщение RowDescription, описывающее строки, которые будут возвращены при выполнении портала; либо сообщение NoData, если портал не содержит запроса, возвращающего строки; либо ErrorResponse, если такого портала нет.

В сообщении Describe (в вариации для оператора) задаётся имя существующего подготовленного оператора (пустая строка обозначает безымянный подготовленный оператор). В ответ передаётся сообщение ParameterDescription, описывающее параметры, требующиеся для оператора,

за которым следует сообщение RowDescription, описывающее строки, которые будут возвращены, когда оператор будет собственно выполнен (или сообщение NoData, если оператор не возвратит строки). ErrorResponse выдаётся, если такой подготовленный оператор отсутствует. Заметьте, что так как команда Bind не выполнялась, сервер ещё не знает, в каком формате будут возвращаться столбцы; в этом случае поля кодов формата в сообщении RowDescription будут содержать нули.

Подсказка. В большинстве случаев клиент должен выдать ту или иную вариацию Describe, прежде чем выдавать Execute, чтобы понять, как интерпретировать результаты, которые он получит.

Сообщение Close закрывает существующий подготовленный оператор или портал и освобождает связанные ресурсы. При попытке выполнить Close для имени несуществующего портала или оператора ошибки не будет. Ответ на это сообщение обычно CloseComplete, но может быть и ErrorResponse, если при освобождении ресурсов возникают проблемы. Заметьте, что при закрытии подготовленного оператора неявно закрываются все открытые порталы, которые были получены из этого оператора.

Сообщение Flush не приводит к генерации каких-либо данных, а указывает серверу передать все данные, находящиеся в очереди в его буферах вывода. Сообщение Flush клиент должен отправлять после любой команды расширенных запросов, кроме Sync, если он желает проанализировать результаты этой команды, прежде чем выдавать следующие команды. Без Flush сообщения, возвращаемые сервером, будут объединяться вместе в минимальное количество пакетов с целью уменьшения сетевого трафика.

Примечание. Простое сообщение Query примерно равнозначно последовательности сообщений Parse, Bind, Describe (для портала), Execute, Close, Sync, с использованием объектов подготовленного оператора и портала без имён и без параметров. Одно отличие состоит в том, что такое сообщение может содержать в строке запроса несколько операторов SQL, для каждого из которых по очереди автоматически выполняется последовательность

Bind/Describe/Execute. Другое отличие состоит в том, что в ответ на него не приходят сообщения ParseComplete, BindComplete, CloseComplete или NoData.

Расширенный протокол позволяет более детально управлять обработкой запроса.

Во-первых, запрос может быть подготовлен. Для этого клиент передает запрос серверу (возможно, в параметризованном виде), а сервер выполняет разбор и переписывание и сохраняет подготовленное дерево запроса в локальную память обслуживающего процесса.

Чтобы выполнить подготовленный запрос, клиент называет его имя и указывает конкретные значения параметров. Сервер планирует запрос исходя из переданных параметров и выполняет его.

За счет подготовки удается избежать повторного разбора и переписывания одного и того же запроса, если он выполняется в одном сеансе неоднократно.

Во-вторых, можно использовать курсоры. Механизм курсоров позволяет получать результат выполнения запроса не весь сразу, а построчно. Информация об открытом курсоре также хранится в локальной памяти обслуживающего процесса.

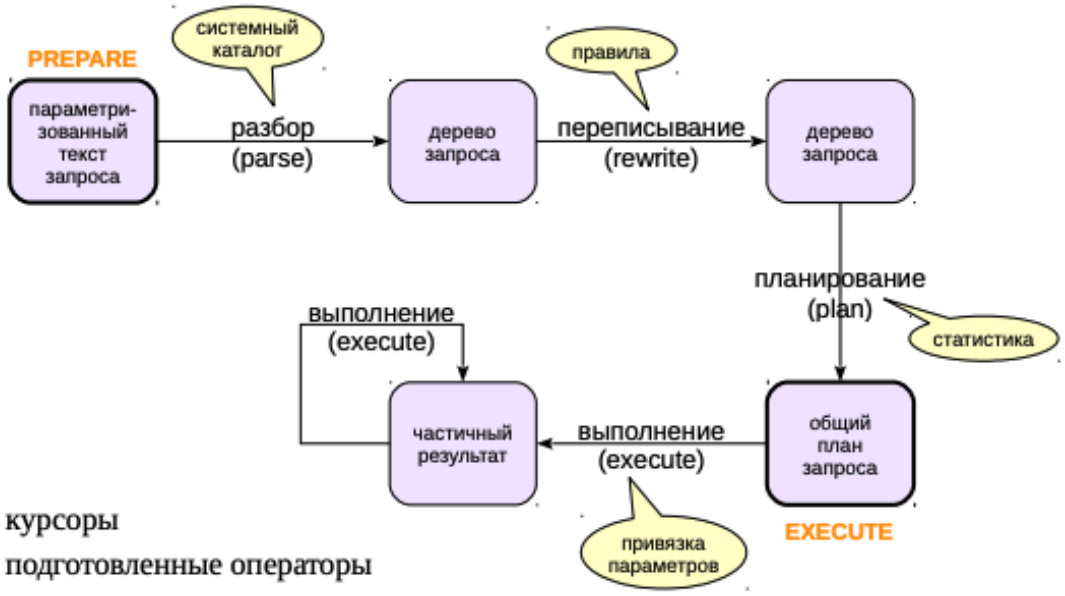


Рисунок 12 – схема обработки с использованием возможностей расширенного протокола

Если запрос не имеет параметров, серверу нет смысла перепланировать запрос при каждом выполнении. В этом случае он сразу запоминает общий (generic) план выполнения. Это позволяет еще больше экономить ресурсы. Если же запрос имеет параметры, то сервер несколько раз (точнее, 5) строит частные (custom) планы в соответствии с переданными параметрами.

Если после этого сервер замечает, что общий (generic) план получается «не хуже» частных, он переключается на него и перестает выполнять перепланирование. Есть и еще один повод использовать подготовленные операторы: гарантировать безопасность от внедрения SQL-кода, если входные данные для запроса получены из ненадежного источника (например, из поля ввода на веб-форме).

Практика 1. Выполняется на вашей БД по аналогии нижеприведенной практике.

Этапы выполнения запроса

Установим параметры, которые покажут примерное время выполнения этапов обработки:

```
=> ALTER SYSTEM SET log_parser_stats = on;
ALTER SYSTEM
=> ALTER SYSTEM SET log_planner_stats = on;
ALTER SYSTEM
=> ALTER SYSTEM SET log_executor_stats = on;
ALTER SYSTEM
=> SELECT pg_reload_conf();
 pg_reload_conf
----------------
 t
(1 row)
```

Обычный удобный способ узнать время выполнения и время планирования (включая разбор и переписывание) - команда EXPLAIN ANALYZE. Ее основное назначение - показать план выполнения запроса, но о

плане как таковом мы будем говорить позже. Пока обратите внимание на две последние строки:

```
=> EXPLAIN (ANALYZE, COSTS OFF, TIMING OFF)
SELECT * FROM ticket_flights;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on ticket_flights (actual rows=8391852 loops=1)
 Planning time: 4.235 ms
 Execution time: 1993.501 ms
(3 rows)
```

Благодаря установленным нами параметрам, из журнала сообщений можно получить более подробную информацию, хотя обычно это и не требуется. Выведем только основные цифры (elapsed):

```
postgres$ tail -n 50 /var/log/postgresql/postgresql-10-main.log
| egrep 'LOG: |elapsed'
2019-02-13 13:02:46.387 MSK [3054] postgres@demo LOG:  PARSER
STATISTICS
     !    0.000000 s user, 0.000000 s system, 0.000029 s elapsed
2019-02-13 13:02:46.392 MSK [3054] postgres@demo LOG:  PARSE
ANALYSIS STATISTICS
     !    0.000000 s user, 0.000000 s system, 0.005306 s elapsed
2019-02-13 13:02:46.392 MSK [3054] postgres@demo LOG:  REWRITER
STATISTICS
     !    0.000000 s user, 0.000000 s system, 0.000001 s elapsed
2019-02-13 13:02:46.397 MSK [3054] postgres@demo LOG:  PLANNER
STATISTICS
     !    0.000000 s user, 0.004000 s system, 0.004176 s elapsed
2019-02-13 13:02:48.391 MSK [3054] postgres@demo LOG:  EXECUTOR
STATISTICS
     !    0.536000 s user, 0.616000 s system, 1.997833 s elapsed
```

Разные способы измерения выдают несколько отличающиеся результаты.

Простой запрос

Протокол простых запросов применяется, когда на сервер просто отправляется оператор (и, если это SELECT или команда с фразой RETURNING, то мы ожидаем получение всех строк результата). Например:

```
=> SELECT model FROM aircrafts WHERE aircraft_code = '773';
     model
---------------
 Боинг 777-300
(1 row)
```

Подготовленные операторы

Создадим подготовленный оператор для нашего запроса:

```
=> PREPARE model(varchar) AS
  SELECT model FROM aircrafts WHERE aircraft_code = $1;
PREPARE
```

Теперь мы можем вызывать оператор по имени:

```
=> EXECUTE model('773');
     model
---------------
 Боинг 777-300
(1 row)

=> EXECUTE model('763');
     model
---------------
 Боинг 767-300
(1 row)
```

Все подготовленные операторы можно увидеть в представлении:

```
=> SELECT * FROM pg_prepared_statements \gx
-[ RECORD 1 ]---+--------------------------------------------
---------
name            | model
statement       | PREPARE model(varchar) AS
+
                |   SELECT model FROM aircrafts WHERE
aircraft_code = $1;
prepare_time    | 2019-02-13 13:02:48.603462+03
parameter_types | {"character varying"}
from_sql        | t
```

Если подготовленный оператор больше не нужен, его можно удалить командой DEALLOCATE, но в любом случае оператор пропадет при завершении сеанса.

```
=> \c
You are now connected to database "demo" as user "postgres".
=> SELECT * FROM pg_prepared_statements;
 name | statement | prepare_time | parameter_types | from_sql
------+-----------+--------------+-----------------+----------
(0 rows)
```

Команды PREPARE, EXECUTE, DEALLOCATE - команды SQL. Клиенты на других языках программирования будут использовать операции, определенные в соответствующем драйвере. Но любой драйвер использует один и тот же протокол для взаимодействия с сервером.

Курсоры

Курсоры дают возможность построчной обработки результата. Они часто используются в приложениях и имеют особенности, связанные с оптимизацией.

На SQL использование курсоров можно продемонстрировать следующим образом. Объявляем (и сразу же открываем) курсор и выбираем первую строку:

```
=> BEGIN;
BEGIN
=> DECLARE c CURSOR FOR SELECT * FROM aircrafts;
DECLARE CURSOR
=> FETCH c;
 aircraft_code |     model     | range
---------------+---------------+-------
 773           | Боинг 777-300 | 11100
(1 row)
```

Читаем вторую строку результата и закрываем открытый курсор.

```
=> FETCH c;
 aircraft_code |     model     | range
---------------+---------------+-------
 763           | Боинг 767-300 |  7900
(1 row)

=> CLOSE c;
CLOSE CURSOR
=> COMMIT;
COMMIT
```

Практика 2. Выполняется на вашей БД по аналогии нижеприведенной практике.

Долгий запрос

```
=> \timing on
Timing is on.
```

Обычный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..10 LOOP
```

```
    EXECUTE 'SELECT avg(amount) FROM ticket_flights';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
DO
Time: 21286,115 ms (00:21,286)
```

Подготовленный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..10 LOOP
    PERFORM avg(amount) FROM ticket_flights;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
DO
Time: 20152,028 ms (00:20,152)
```

Время изменилось незначительно - большую часть занимает выполнение запроса.

Быстрый запрос

Обычный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..100000 LOOP
    EXECUTE 'SELECT * FROM bookings WHERE book_ref =
''0824C5''';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
DO
Time: 4680,450 ms (00:04,680)
```

Подготовленный оператор:

```
=> DO $$
BEGIN
  FOR i IN 1..100000 LOOP
```

```
    PERFORM * FROM bookings WHERE book_ref = '0824C5';
  END LOOP;
END;
$$ LANGUAGE plpgsql;
DO
Time: 850,609 ms
```

Время сократилось существенно - разбор и планирование занимает большую часть общего времени.

Подробнее о планировании

Темы:

- Процесс планирования
- Оценка кардинальности
- Оценка стоимости
- Выбор наилучшего плана

Планирование запроса – очень важный и достаточно сложный этап, поэтому остановимся на нем подробнее.

Процесс планирования

Статистика

- данные о размере таблиц и распределении данных

Оценка кардинальности

- селективность условий — доля выбираемых строк,
- кардинальность — итоговое число строк
- для расчета требуется статистика

Оценка стоимости

- в первую очередь зависит от типа узла и числа обрабатываемых строк

Перебор планов оптимизатором

- выбирается план с наименьшей стоимостью

Оптимизатор перебирает всевозможные планы выполнения, оценивает их и выбирает план с наименьшей стоимостью.

Чтобы оценить стоимость узла, нужно знать тип этого узла (понятно, что

стоимость чтения данных напрямую из таблицы или с помощью индекса будет отличаться) и объем обрабатываемых этим узлом данных.

Для оценки объема данных важны два понятия:

- кардинальность — общее число строк;
- селективность — доля строк, отбираемых условиями (предикатами).

Для оценки селективности и кардинальности надо, в свою очередь, иметь сведения о данных: размер таблиц, распределение данных по столбцам.

Таким образом, в итоге все сводится к статистике — информации, собираемой и обновляемой процессом автоанализа или командой ANALYZE.

Если кардинальность оценена правильно, то и стоимость обычно рассчитывается довольно точно. Основные ошибки оптимизатора связаны именно с неправильной оценкой кардинальности. Это может происходить из-за неадекватной статистики, невозможности ее использования или несовершенства моделей, лежащих в основе оптимизатора. Об этом мы будем говорить подробнее.

Оценка кардинальности

Кардинальность метода доступа

$$rows_{A\ where\ cond} = rows_A \cdot sel_{cond}$$

Кардинальность соединения

$$rows_{A\ join\ B\ on\ cond} = rows_A \cdot rows_B \cdot sel_{cond}$$

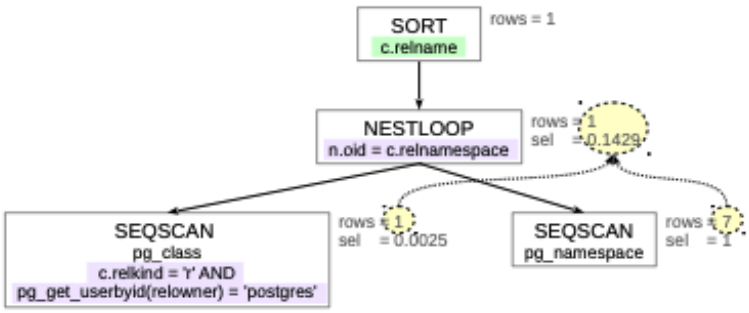


Рисунок 13 – оценка кардинальности

Оценку кардинальности удобно рассматривать как рекурсивный процесс. Чтобы оценить кардинальность узла, надо сначала оценить кардинальности дочерних узлов, а затем — зная тип узла — вычислить на их

основе кардинальность самого узла.

Таким образом, сначала можно рассчитать кардинальности листовых узлов, в которых находятся методы доступа к данным. Для этого нам нужно знать размер таблицы и селективность условий, наложенных на нее. Как конкретно это делается, мы рассмотрим позже.

Пока отметим, что достаточно уметь оценивать селективность простых условий, а селективность условий, составленных с помощью логический операций, расчитываатся по простым формулам:

$sel_{x \text{ and } y} = sel_x \cdot sel_y$

$sel_{x \text{ or } y} = 1 - (1 - sel_x)(1 - sel_y)$

К сожалению, эти формулы предполагают независимость предикатов. В случае коррелированных предикатов такая оценка будет неточной.

Затем можно рассчитать кардинальности соединений. Кардинальности соединяемых наборов данных нам уже известны, осталось оценить селективность условий соединения. Пока будем просто считать, что это как-то возможно.

Аналогично можно поступить и с другими узлами, например, с сортировками или агрегациями.

Важно отметить, что ошибка расчета кардинальности, возникшая в нижних узлах, будет распространяться выше, приводя в итоге к неверной оценке и выбору неудачного плана.

Оценка стоимости

Вычисляется на основе математических моделей

$cost = cost_A + \sum cost_{дочерние\ узлы\ A}$

Две компоненты

подготовительная работа .. получение всех строк

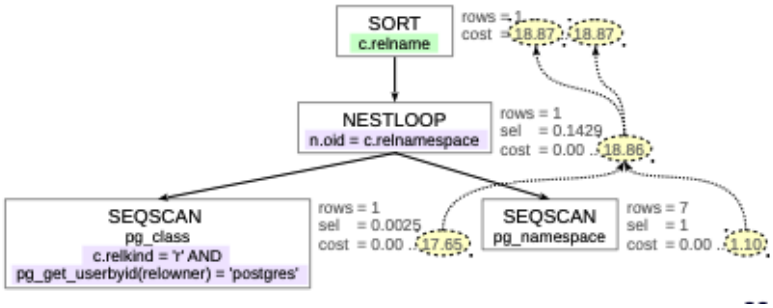


Рисунок 14 – оценка стоимости

Теперь рассмотрим общий процесс оценки стоимости. Он также рекурсивен по своей природе. Чтобы рассчитать стоимость поддерева, сначала надо рассчитать стоимости дочерних узлов, сложить их, и добавить стоимость самого узла.

Стоимость работы самого узла рассчитывается на основе математической модели, заложенной в планировщик, с учетом оценки числа обрабатываемых строк (которая уже рассчитана).

Стоимость состоит из двух компонентов, оцениваемых отдельно. Первый — стоимость подготовительной работы, второй — стоимость получения всех строк выборки.

Некоторые операции не требуют никакой подготовки; у таких узлов первая компонента стоимости будет равна нулю. Некоторые операции, наоборот, требуют выполнения предварительных действий. Например, сортировка в приведенном примере должна сначала получить от дочернего узла все данные, чтобы начать работу. У таких узлов первая компонента будет отлична от нуля — эту стоимость придется «заплатить» независимо от того, сколько строк результата потребуется.

Важно понимать, что стоимость отражает оценку планировщика и может не коррелировать с реальным временем выполнения. Можно считать, что стоимость выражена в неких «условных единицах», которые сами по себе

ни о чем не говорят. Стоимость нужна лишь для того, чтобы планировщик мог сравнивать разные планы одного и того же запроса.

Выбор лучшего плана

Перебор планов

- порядок соединений, способы соединений, методы доступа
- по возможности полный перебор,
- при большом числе вариантов — генетический алгоритм (GEQO)

Простые запросы и подготовленные операторы

- минимальная общая стоимость
- оптимизируется время получения всех строк

Курсоры

- минимальная стоимость для получения cursor_tuple_fraction строк
- оптимизируется время получения части первых строк

Оптимизатор старается перебрать всевозможные планы выполнения запроса, чтобы выбрать из них лучший.

При большом количестве вариантов (в первую очередь на это влияет количество соединяемых таблиц) полный перебор за разумное время становится невозможен. В таком случае PostgreSQL переключается на использование генетического алгоритма (GEQO — Genetic Query Optimization). Это может привести к тому, что оптимизатор выберет не лучший план не из-за ошибок в оценке, а просто потому, что лучший план не рассматривался.

Что же считается «лучшим планом»? Для обычных запросов это план, минимизирующий время получения всех строк, то есть план с минимальной общей стоимостью (вторая компонента).

Однако при использовании курсоров может быть важно как можно быстрее получить первые строки. Поэтому существует параметр cursor_tuple_fraction (значение по умолчанию 0.1), задающий долю строк, которую надо получить как можно быстрее. Чем меньше значение этого параметра, тем больше на выбор плана влияет первая компонента стоимости,

а не вторая.

Итоги

Обработка запроса состоит из нескольких шагов: разбор и переписывание, планирование, выполнение.

Имеются разные способы выполнения:

- простой — непосредственное выполнение и получение результата
- расширенный — подготовленные операторы и курсоры

Время выполнения зависит от качества планирования. Планирование на основе стоимостного оптимизатора.

Задание на практическую работу:

1. Влияние подготовки на выполнение долгого оператора. Вычислите среднюю стоимость одного заказа. Посчитайте среднее время выполнения этого запроса. Подготовьте оператор для этого запроса. Снова посчитайте среднее время выполнения. Во сколько раз ускорилось выполнение?
2. Влияние подготовки на выполнение коротких операторов. Многократно запросите данные о одном и том же заказе. Посчитайте среднее время выполнения. Подготовьте оператор для этого запроса. Снова посчитайте среднее время выполнения. Во сколько раз ускорилось выполнение в этом случае?

Время выполнения одного и того же запроса может отличаться, причем довольно сильно (особенно время первого выполнения). Чтобы сгладить разницу, время надо усреднить, выполнив запрос несколько раз. Для этого удобно использовать язык PL/pgSQL, учитывая, что:

- динамический запрос, выполняемый командой PL/pgSQL EXECUTE (не путать с командой SQL EXECUTE!), каждый раз проходит все этапы;
- запрос SQL, встроенный в PL/pgSQL-код, выполняется с помощью подготовленных операторов.

Пример синтаксиса команды для обычного оператора:

```
DO $$
BEGIN
    FOR i IN 1..10 LOOP
```

```
          EXECUTE 'SELECT ... FROM ...';
     END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Для подготовленного оператора (здесь SELECT заменяется на PERFORM, поскольку нас не интересует результат как таковой):

```
DO $$
BEGIN
     FOR i IN 1..10 LOOP
          PERFORM ... FROM ...;
     END LOOP;
END;
$$ LANGUAGE plpgsql;
```