Пользовательские функции

Раздаточный материал к 5-му уроку, посвященному изучению User Defined Functions.

|  |  |
| --- | --- |
| СОДЕРЖАНИЕ | |
| 1 | Определение UDF |
| 2 | Слои хранения и предоставления данных |
| 3 | Варианты ELT |
| 4 | Последовательности |
| 5 | Логирование |
| 6 | Слой витрин и представлений для отчётности |
| 7 | Транзакционность и обработка исключений |
| 8 | Источники информации (ссылки) |

Определение User Defined Functions

User-Defined Function - часть кода, написанная на одном из доступных языков, которую можно выполнять с помощью SQL запроса.

Они могут принимать обязательные и необязательные параметры, указав в таком случае значение по умолчанию или добавив дополнительную обработку в теле функции, если параметр на вход не указывается при вызове функции. Функции возвращают одну или несколько строк и могут выполняться на мастере и на сегментах.

Изображение выглядит как текст

Автоматически созданное описание

По команде CREATE FUNCTION можно создать новую функцию. CREATE OR REPLACE создаст новую или заменит существующую функцию. Имя новой функции не должно совпадать ни с одной существующей функцией с теми же типами входных параметров в той же схеме. Однако функции с разными типами параметров могут иметь общее имя. Стоит учитывать то, что, чтобы поменять имя или тип входных параметров, необходимо удалить и заново создать функцию. Пользователь, который создаёт функцию – является её владельцем (owner).

Необходимо указать тип возвращаемых данных. Тип возвращаемого значения может быть базовым (int, text, date, bool и другие), композитным или сложным (иметь структуру строки, то есть содержать список полей с их типами), может быть указан с использованием констрейнтов not null или check или может ссылаться на тип столбца таблицы. Также функция может ничего не возвращать, в таком случае нужно указать следующее значение.

Далее следует указать имя языка, на котором реализована функция. Например plpgsql, python, c и другие.

Затем прописывается класс функции и само её определение с блоком объявления переменных и телом функции, которое будет содержать в себе необходимые преобразования данных.

Определение прописывается с помощью команды AS, после которой прописывается строка, определяющая функцию — это может быть команда SQL или текст на процедурном языке.

Изображение выглядит как текст

Автоматически созданное описание

Если указать атрибут **WINDOW** – то в таком случае наша функция будет являться оконной. Используется для функций, написанных на С. Атрибут **CALLED ON NULL INPUT** – ставится по умолчанию и обозначает, что функция будет вызываться нормально, когда некоторые из ее параметров не принимают значения. В этом случае автор функции обязан проверять пустые значения, если это необходимо, и выдавать корректные сообщения в логах и результаты. **RETURNS NULL ON NULL INPUT** или **STRICT** – указывает, что функция всегда возвращает значение NULL, когда любой из ее параметров не принимает значение. То есть, если данный атрибут указан, функция не выполняется при наличии пустых параметров; вместо этого автоматически возвращается NULL результат. **SECURITY INVOKER** – ставится по умолчанию и означает, что функция выполняется с правами пользователя, который ее вызывает. **SECURITY DEFINER** указывает, что функция может выполняться пользователем с правами создавшего ее пользователя. С атрибутом **COST** указывается положительное число, определяющее расчетную стоимость выполнения функции в единицах cpu\_operator\_cost. Если функция возвращает набор строк, execute\_cost определяет стоимость возвращаемой строки. Если стоимость не указана, для языка C и внутренних функций по умолчанию используется 1 единица, а для функций на других языках — 100 единиц. Атрибутом **SET** можно применить значение к параметру конфигурации сессии при входе в функцию. Параметр конфигурации восстанавливается до своего предыдущего значения при выходе из функции. **SET FROM CURRENT** сохраняет значение параметра, которое является текущим при выполнении CREATE FUNCTION, как значение, которое будет применяться при входе в функцию. Другая форма атрибута **AS** используется для динамически загружаемых функций языка C. С помощью describe\_function мы можем передать имя callback функции, которая будет вызываться внутри внешней функции после выполнения SQL запроса с вызовом внешней функции для реализации какой-либо подпрограммы или более сложной логики процесса.

EXECUTE ON атрибут – означает то, где будет выполняться функция, когда она вызывается в процессе выполнения запроса.

Первый способ ON ANY (который указывается по умолчанию) означает, что функция может выполняться как на мастере, так и на сегментах. База данных Greenplum автоматически определяет, где выполняется функция. Если в вашей функции есть запросы к распределённым таблицам, то рекомендуется использовать способ EXECUTE ON MASTER. ON ALL SEGMENTS способ указывает, что функция должна выполняться на сегментах, но не на мастере. Общий результат функции в таком случае — это объединение всех результатов работы функции на всех сегментах.

Итак, функции, которые выполняют запросы к разным распределённых таблицам, должны выполняться только на мастере. Однако функции, которые выполняют только запросы SELECT к реплицированным таблицам, могут выполняться и на сегментах. Если функция обращается к таблице с хэш-распределением или к таблице со рандомным распределением, функция должна быть определена с атрибутом EXECUTE ON MASTER. Если не следовать этим рекомендациям по правильному объявлению функции со сложными запросами к распределённым таблицам, то функция может вернуть некорректный результат. В случае EXECUTE ON ANY оптимизатор может самостоятельно определить, что было бы полезно передать вызов функции на сегменты. В основном используется EXECUTE ON ANY атрибут, и мы будем использовать только его.

Есть ограничения для функций, определенных с помощью атрибута EXECUTE ON MASTER или EXECUTE ON ALL SEGMENTS:

* В таких случаях функция должна возвращать множество.
* Функция не может находиться в операторе FROM запроса.
* Функция не может быть в списке SELECT запроса с оператором FROM.
* Запрос, включающий эту функцию, использует оптимизатор Postgres.

Теперь опишем атрибут изменчивости или волатильности функции (можно называть также классом функции), который будет означать видимость изменений.

Другими словами функции, написанные на SQL, имеют важное свойство, определяемое характеристикой изменчивости, а именно видимостью изменений в БД. Это поведение реализуется посредством снэпшотов в MVCC. IMMUTABLE и STABLE функции используют снэпшот БД, полученный в начале вызывающего запроса, тогда как функции VOLATILE получают свежий снэпшот в начале каждого запроса, который они выполняют.

Класс IMMUTABLE – это класс постоянных функций, которые не могут модифицировать базу данных и гарантированно всегда возвращают одинаковые результаты для одних и тех же аргументов. Эта характеристика позволяет оптимизатору предварительно вычислить функцию, когда она вызывается в запросе с постоянными аргументами. То есть, она не обращается к базе данных и не использует информацию, не переданную ей явно в списке аргументов.

Стабильная (STABLE) функция так же не может модифицировать базу данных и в рамках одного сканирования таблицы она всегда возвращает один и тот же результат для определённых значений параметров, но от транзакции к транзакции значение уже может меняться в отличие от IMMUTABLE функции. Оптимизатор заменит множество вызовов этой функции одним (то есть целевое значение вычисляется только один раз, а не для каждой строки), и именно поэтому выражение, содержащее такую функцию, можно безопасно использовать в условии поиска по индексу.

Напомним, что функции VOLATILE получают свежий снэпшот БД в начале каждого запроса, который они выполняют. Эта изменчивая функция может делать всё, что угодно, в том числе, модифицировать базу данных. Она может возвращать различные результаты при нескольких вызовах с одинаковыми аргументами. Оптимизатор не делает никаких предположений о поведении таких функций, то есть в запросе, использующем изменчивую функцию, она будет вычисляться заново для каждой строки, когда потребуется её результат.

Выражения без оператора FROM всегда выполняются на мастере: например, вызов встроенной VOLATILE функции setval(), или вызов пользовательской функции по вставке новых строк в таблицу. Выражения с оператором FROM выполняются на сегментах. Например, встроенная агрегатная функция суммирования показателя из распределенной по сегментам таблицы может выполняться на сегментах. Или если наша пользовательская функция возвращает набор строк и если мы её используем в выражении с оператором FROM, то допускается запустить её на сегментах.

В некоторых случаях Greenplum не поддерживает использование функций. Если выполняется ВСЕ из следующих условий:

* Данные таблицы (FROM) распределены по сегментам.
* Функция читает или изменяет данные в распределённых таблицах.
* Функция возвращает более одной строки или принимает на вход столбец из таблицы (FROM).

PL/pgSQL – это процедурный блочно-структурированный язык, который позволяет сгруппировать блок вычислений и последовательность запросов внутри сервера базы данных. Он используется для создания функций. Добавляет к SQL коду управляющие структуры (возврат значений в функциях, различные условные операторы, циклы, обработку ошибок).

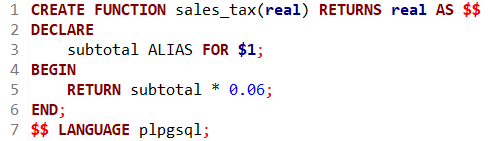
Текст тела функции должен быть блоком. Как говорилось ранее, все переменные функции объявляются в специальной для этого секции в блоке. Переменные могут иметь любой тип и задаются указанным способом.

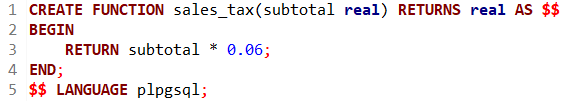


Предложение DEFAULT задаёт начальное значение, которое присваивается переменной при входе в блок. Если оно отсутствует, то переменная инициализируется SQL-значением NULL. Указание CONSTANT предотвращает изменение значения переменной после инициализации, таким образом, значение остаётся постоянным в течение всего блока. Параметр COLLATE определяет правило сортировки, которое будет использоваться для этой переменной. Если указано NOT NULL, то попытка присвоить NULL во время выполнения приведёт к ошибке. Все переменные, объявленные как NOT NULL, должны иметь непустые значения по умолчанию. Можно использовать знак равенства (=) вместо совместимого с PL/SQL :=.

Значение по умолчанию вычисляется и присваивается переменной каждый раз при входе в блок (не только при первом вызове функции). Так, например, если переменная типа timestamp имеет функцию now() в качестве значения по умолчанию, это приведёт к тому, что переменная всегда будет содержать время текущего вызова функции, а не время, когда функция была предварительно скомпилирована.

Передаваемые в функцию параметры могут объявляться идентификаторами или псевдонимами, как было показано ранее. В первом случае мы дали алиас заданному параметру $1 с типом real в секции Declare. Во втором дали этому же параметру псевдоним сразу при его объявлении.





Вместо явной команды RETURN для возвращаемого значения можно указать выходной параметр. Он стартует с NULL и получает значение во время выполнения функции.

Типы возвращаемых значений перечислялись ранее. Подробнее с особенностями этого процедурного языка можете познакомиться по ссылке на документацию в разделе Источники информации.

Слои хранения и предоставления данных

Перед изучением вариантов ELT вспомним слои хранения и предоставления данных. На начальном этапе инкремент исходных данных - дельта относительно уже загруженных в ХД данных попадает в **STG** стейджинг слой, который является временным и ежедневно или в некоторых случаях несколько раз в день перезагружается.

Историю сырых, необработанных данных, которые полностью соответствуют данным в таблице исходной системы, можно хранить в отдельном **RAW** слое. Эта история может использоваться для перегрузки данных в целевые таблицы ХД или для анализа реализованных потоков данных.

**Operational Data Store** хранит операционные данные, разложенные по сущностям и которые являются результатом предобработки данных из RAW перед загрузкой в детальный DDS слой. Иногда этот слой можно не использовать, а необходимые преобразования делать в потоке из RAW сразу в DDS.

**Detail Data Store** хранит уже детальные данные, распределенные по бизнес-сущностям и объединенные из разных источников. Такие данные необходимы конечному пользователю, но ещё не являются агрегированными.

В слое витрин **CDM** данные необходимо агрегировать под нужды пользователей и рассчитать бизнес-метрики и показатели по определённым формулам. Данные должны соответствовать всем бизнес-потребностям. Каждая витрина, как правило, содержит тематическую информацию по одному из направлений деятельности компании.

**REPORT** слой содержит определённые запросы к витринам, которые могут формировать например рабочую книгу или образовывать в BI системах доступные для сотрудников различных департаментов пользовательские отчёты строящиеся на юниверсах, которые связывают витринные таблицы фактов и справочники. Есть возможность гибкого просмотра данных, например за конкретную дату, для определённого магазина, регионального отделения или товара.

Варианты ELT

Дадим определение ELT процессу и покажем основное отличие от ETL подхода.   
ELT — это процесс переноса данных из разнородных источников в хранилище данных (ХД) с целью их дальнейшего анализа. В целом, процесс ELT выполняет те же функции, что и ETL с той только разницей, что этапы загрузки и преобразования меняются местами. ELT извлекает данные из исходных систем, но вместо перемещения их в промежуточную область для преобразования, загружает необработанные данные непосредственно в приемник.

Основные преимущества ELT подхода:

* быстрота – так как затратный этап преобразования не предшествует загрузке, пользователи получают интересующие данные быстрее;
* разделение задач - поскольку данные преобразуются после загрузки, появляется больше возможностей управления ими;
* масштабирование – есть возможность наращивать число узлов обработки;
* непрерывность работы – к преимуществу можно также отнести постоянство получения данных;

Основными недостатками являются:

* работа с «сырыми данными» требует от пользователей более высокой квалификации;
* поскольку информация загружается в ХД в сыром, необработанном виде, может произойти снижение качества данных;
* сложно обеспечить единый согласованный подход к дальнейшей обработке данных, т. к. любой аналитик имеет возможность применить специфичные алгоритмы обработки;
* отсутствие анонимизации и шифрования данных могут создать проблемы с безопасностью;

Опишем типы загрузки данных для справочников

1. Тип **FULL** подразумевает под собой полную перезапись справочника с использованием DELETE или TRUNCATE с последующим INSERT всех записей.

а) **DELETE + INSERT** – относительно медленно, но без блокировок на чтение.

б) **TRUNCATE + INSERT** – быстрое обновление, но с блокировкой ACCESS EXCLUSIVE

1. С помощью команды **EXCHANGE DEFAULT PARTITION** можно реализовать тип **DELTA\_MERGE**. Он подразумевает под собой полную замену данных дефолтной партиции целевой таблицы данными заранее подготовленной временной таблицы.
2. С помощью типа **DELTA\_UPSERT** мы можем предварительно удалить записи в целевой таблице по ключу, а затем с помощью INSERT обновить, вставив заново, старые записи и вставить новые.

Опишем некоторые типы загрузки данных для таблиц фактов.

1. Тип **DELTA\_PARTITION** является наиболее эффективным. Он подразумевает под собой подмену дневной, недельной или месячной партиции целевой таблицы временной, соответствующей данному периоду и содержащей актуальные данные. Почти всегда большие таблицы-фактов партиционируются по какой-либо дате. Но на практике можно использовать и другие поля для разделения на партиции.
2. Тип **FULL DELETE INSERT ON PARTITION** аналогичен типу DELTA PARTITION. Но вместо операции exchange\_partition будет функция с логикой полной очистки партиции и вставки актуальных данных из подготовленной временной таблицы. Стоит рассмотреть, если в момент обновления предполагается интенсивная нагрузка на чтение обновляемой сущности. Например, внутридневные обновления данных.
3. Тип **DELTA UPSERT** был описан ранее. Стоит учитывать, что данную операцию необходимо выполнять при обновлении небольшого количества данных

Перечисленные ранее названия вариантов ELT (DELTA MERGE, DELTA UPSERT, DELTA PARTTION) – это не общепринятый стандарт, а придуманные термины для логического обозначения процесса. Общие наглядные схемы процессов для всех вариантов ELT можно посмотреть в презентации.

Последовательности

Дадим определение последовательностям. Последовательности — это объекты базы данных, который генерируют целые числа в соответствии с правилами, установленными во время их создания. Создать новую последовательность можно командой CREATE SEQUENCE.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Данные объекты, например, можно применять для генерации уникальных id при помощи функции nextval().



Функции для работы с последовательностями:

|  |  |  |
| --- | --- | --- |
| Функция | Тип результата | Описание |
| currval(regclass) | Bigint | Выдаёт значение заданной последовательности, которое было возвращено при последнем вызове функции nextval. |
| lastval() | Bigint | Выдаёт значение любой последовательности, которое было возвращено при последнем вызове функции nextval. |
| nextval(regclass) | Bigint | Продвигает последовательность к следующему значению и возвращает его. |
| setval(regclass, bigint) | Bigint | Устанавливает текущее значение последовательности. |
| setval(regclass, bigint, boolean) | Bigint | Устанавливает текущее значение последовательности и флаг is\_called, указывающий на то, что это значение использовалось. |

Логирование

Можно написать пользовательскую функцию для логирования загрузок. При вызове функции для логов можно передавать в неё тип лога, тип сообщения и «место» или название функции, для которого выполнения логирования.

Изображение выглядит как текст

Автоматически созданное описание

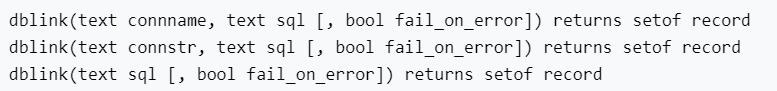
Конечно, хорошо было бы записывать такие сообщения в какую-нибудь таблицу для логов и хранить их там. Например, сообщения о начале вставки записей в таблицу, об окончании, о возникшей ошибке или о другом.

Изображение выглядит как текст

Автоматически созданное описание

Но в связи с особенностями транзакционности системы, в случае падения функции, все выполняемые действия откатываются, соответственно сообщения с логами не запишутся в таблицу. Но можно обойти этот момент с помощью функции dblink, с помощью которой можно выполнять запрос (обычно SELECT, но это может быть и любой другой оператор SQL, возвращающий строки) в удалённой базе данных, передав ей имя сервера и сам SQL скрипт.

Синтаксис:



Когда этой функции передаются два аргумента типа text, первый сначала рассматривается как имя постоянного подключения; если такое подключение находится, команда выполняется для него. Если не находится, первый аргумент воспринимается как строка подключения, и заданное подключение устанавливается только на время выполнения этой команды.

Аргументы:

*Connname* - имя используемого подключения, опустите этот параметр, чтобы использовать безымянное подключение. *Connstr* - строка подключения. *Sql* - SQL-запрос, который вы хотите выполнить в удалённой базе данных. *Fail\_on\_error* - если равен true (это значение по умолчанию), в случае ошибки, выданной на удалённой стороне соединения, ошибка также выдаётся локально. Если равен false, удалённая ошибка выдаётся локально как «Замечание», и функция не возвращает строки. Подробнее функции dblink\* можно изучить по ссылкам разделе с источниками информации.

Слой витрин и представлений для отчётности

Витрина – это таблица, которая содержит узкоспециализированную, тематическую информацию, ориентированную под запросы сотрудников определенного департамента. Содержит в себе агрегированную информацию по определенным тематикам.

Изображение выглядит как диаграмма

Автоматически созданное описание

Виды витрин:

* Логическая таблица (table) – обычная таблица, с возможностью для пользователя указать момент времени, относительно которого требуется извлечь данные таблицы.
* Логическое представление (view) – сохраненный именованный SQL-запрос, к которому можно выполнять запросы.
* Логическая внешняя таблица – виртуальная таблица, являющаяся указателем на источник или приёмник данных.

Представление – это виртуальная таблица, подразумевающая под собой SQL запрос с оператором SELECT с выборкой данных из других таблиц или представлений. Запрос будет подставлен как подзапрос при использовании VIEW. В отличие от обычных таблиц реляционных баз данных, представление не является самостоятельной частью набора данных, хранящегося в базе.

Команда CREATE VIEW создаёт представление запроса. Оно лишено физической материализации, поэтому указанный запрос будет выполняться при каждом обращении к представлению. Если представление с этим именем уже существует команда CREATE OR REPLACE VIEW заменит его.

Изображение выглядит как текст

Автоматически созданное описание

Транзакционность и обработка исключений

Суть транзакций заключается в том, что они объединяют последовательность DML операций в одну по принципу «Все или ничего». Если не выполняется хотя бы одна операция, то изменения, сделанные предыдущими операциями также откатываются. Промежуточное состояние внутри этой последовательности не видны другим транзакциям.

По умолчанию любая возникающая ошибка прерывает выполнение функции и транзакцию, в которой она выполняется. Использование в блоке BEGIN END секции EXCEPTION позволяет перехватывать и обрабатывать ошибки.

Синтаксис секции EXCEPTION:

Изображение выглядит как текст

Автоматически созданное описание

Если ошибок не было, то выполняются все ***операторы*** блока BEGIN END и управление переходит к следующему оператору после END. Но если при выполнении ***оператора*** происходит ошибка, то дальнейшая обработка прекращается и управление переходит к списку исключений в секции EXCEPTION. В этом списке ищется первое исключение, условие которого соответствует ошибке. Если исключение найдено, то выполняются соответствующие ***операторы\_обработчика*** и управление переходит к следующему оператору после END. Если исключение не найдено, то ошибка передаётся наружу, как будто секции EXCEPTION не было. При этом ошибку можно перехватить в секции EXCEPTION внешнего блока BEGIN END. Если ошибка так и не была перехвачена, то обработка функции прекращается.

В качестве ***условия*** может задаваться имя ошибки Postgres, возможные ошибки можно посмотреть по ссылке в разделе Источники информации. Имена условий воспринимаются без учёта регистра. Можно указать сам код ошибки. Код ошибки указывается при помощи SQLSTATE. Специальному имени условия OTHERS соответствуют все типы ошибок, кроме QUERY\_CANCELED и ASSERT\_FAILURE.

Если при выполнении ***операторов\_обработчика*** возникнет новая ошибка, то она не может быть перехвачена в этой секции EXCEPTION. Ошибка передаётся наружу и её можно перехватить в секции EXCEPTION внешнего блока. При выполнении команд в секции EXCEPTION локальные переменные функции сохраняют те значения, которые были на момент возникновения ошибки. Однако все изменения в базе данных, выполненные в блоке, будут отменены. Приведём пример с обработкой ошибки деления на ноль.

Изображение выглядит как текст

Автоматически созданное описание

При присвоении значения переменной y произойдёт ошибка division\_by\_zero. Она будет перехвачена в секции EXCEPTION. Оператор RETURN вернёт значение x, увеличенное на единицу, но изменения сделанные командой UPDATE будут отменены. Изменения, выполненные командой INSERT, которая предшествует блоку, не будут отменены. В результате таблица будет содержать имя Tom Jones.

Существует системная команда GET DIAGNOSTICS, которая получает информацию о текущем состоянии выполнения кода и команда GET STACKED DIAGNOSTICS, которая выдаёт информацию о состоянии выполнения в момент предыдущей ошибки. Возможные элементы, выдаваемые этими командами можете просмотреть в презентации к уроку.

В Greenplum нет возможности сделать автономную транзакцию внутри родительской. В PostgreSQL, на котором основан Greenplum, такая возможность есть, но в Greenplum она не реализована. Но есть некоторые обходные пути для решения этой проблемы.

1. Можно создать WRITABLE внешнюю таблицу с соединением по PXF с подключением на сам Greenplum, то есть указать соединение само в себя. Таким образом можно будет записать данные в таблицу в отдельной транзакции, отличной от текущей. Через внешнюю таблицу отдельной транзакцией также можно записать данные в локальный CSV-файл.
2. Ещё один способ мы рассмотрели ранее, когда разбирали логирование и использовали функцию dblink. В качестве адреса сервера, на котором будет выполняться запрос, можно указать также сам Greenplum. Таким образом, запрос выполнится в отдельной транзакции.
3. Также например, мы можем обернуть каждую SQL команду в пользовательской функции в блок BEGIN END с обработкой всех ошибок через EXCEPTION. Таким образом можно организовать вложенные транзакции. Любой блок сможет выполниться и его результат зафиксируется независимо от результата выполнения других блоков.
4. Также существует возможность указывать Savepoints, которые позволяют создавать точки сохранения внутри транзакции, чтобы можно было откатиться к ним в случае необходимости, не откатывая всю транзакцию. Для создания такой точки используется команда SAVEPOINT, а для отката к ней - команда ROLLBACK TO SAVEPOINT.

В пользовательских функциях нет возможности использовать savepoints. Rollback to savepoint является неким аналогом блока BEGIN EXCEPTION END в функциях, можно реализовать аналогичный сценарий с помощью таких блоков.

Источники информации (ссылки)

1. [Using Functions and Operators | Pivotal Greenplum Docs](https://gpdb.docs.pivotal.io/6-4/admin_guide/query/topics/functions-operators.html)
2. [CREATE FUNCTION | Pivotal Greenplum Docs](https://gpdb.docs.pivotal.io/6-9/ref_guide/sql_commands/CREATE_FUNCTION.html)
3. [DO | Pivotal Greenplum Docs](https://gpdb.docs.pivotal.io/6-4/ref_guide/sql_commands/DO.html#topic1)
4. [Postgres Pro Standard: Документация: 9.5: 35.6. Категории изменчивости функций](https://postgrespro.ru/docs/postgrespro/9.5/xfunc-volatility)
5. [Postgres Pro Standard: Документация: 9.5: 13.1. Введение](https://postgrespro.ru/docs/postgrespro/9.5/mvcc-intro)
6. [Что в глубинах Data Lake? Строим архитектуру, укладываем слои, распределяем ответственность](https://www.pvsm.ru/bazy-danny-h/373051)
7. [Что такое ETL и почему это важно: полное руководство | АНАЛИТИКА ПЛЮС](https://analytikaplus.ru/chto-takoe-etl-polnoe-rukovodstvo/)
8. [PostgreSQL: Документация: 9.6: CREATE FUNCTION](https://postgrespro.ru/docs/postgresql/9.6/sql-createfunction)
9. [Postgres Pro Standard: Документация: 10: dblink](https://postgrespro.ru/docs/postgrespro/10/contrib-dblink-function)
10. [Postgres Pro Standard: Документация: 10: F.12. dblink](https://postgrespro.ru/docs/postgrespro/10/dblink)
11. [Изоляция и многоверсионность](https://edu.postgrespro.ru/dev1-9.6/dev1_03_arch_mvcc.pdf)
12. [CREATE FUNCTION - Документация по СУБД «Квант-Гибрид» 1.5.0](https://repo.granit-concern.ru/qhb/std-1/doc/SQL/commands/create-function.html)
13. [SET - Документация по СУБД «Квант-Гибрид» 1.5.0](https://repo.granit-concern.ru/qhb/std-1/doc/SQL/commands/set.html#%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%8B)
14. [PL/pgSQL — процедурный язык SQL](https://postgrespro.ru/docs/postgresql/9.6/plpgsql)
15. [Управляющие структуры (Обработка ошибок)](https://postgrespro.ru/docs/postgrespro/9.6/plpgsql-control-structures)