

1. Архитектура ANSI-SPARC.

Архитектура СУБД включает в себя 3 уровня:

Модель ANSI/SPARC



1. Внешний (пользовательский).

- *Внешнее представление* – это содержимое БД, каким его видят определенный конечный пользователь или группа пользователей.
- Внешних представлений обычно бывает несколько.
- Отдельного пользователя обычно интересует только некоторая часть всей БД.
- Пользовательское представление данных может существенно отличаться от того, как они хранятся.

2. Промежуточный (концептуальный).

- *Концептуальное представление* – это представление всей информации БД в несколько более абстрактной по сравнению с физическим способом хранения данных.
- Состоит из одного представления
- Этот уровень описывает то, *какие* данные хранятся в базе данных, а также *связи*, существующие между ними.
- Не содержит никаких сведений о методах хранения данных – описание сущности должно содержать сведения о типах данных атрибутов и их длине, но не должно включать сведений об организации хранения данных, например об объеме занятого пространства в байтах.

3. Внутренний (физический).

- *Внутреннее представление* описывает все подробности, связанные с хранением данных в базе.
- Описывает физическую реализацию базы данных.
- Содержит описание структур данных и организации отдельных файлов, используемых для хранения данных в запоминающих устройствах.
- На внутреннем уровне хранится следующая информация:
 - распределение дискового пространства для хранения данных и индексов;
 - описание подробностей сохранения записей (с указанием реальных размеров сохраняемых элементов данных);
 - сведения о размещении записей;
 - сведения о сжатии данных и выбранных методах их шифрования.

Почти все современные СУБД соответствуют принципам ANSI-SPARC.

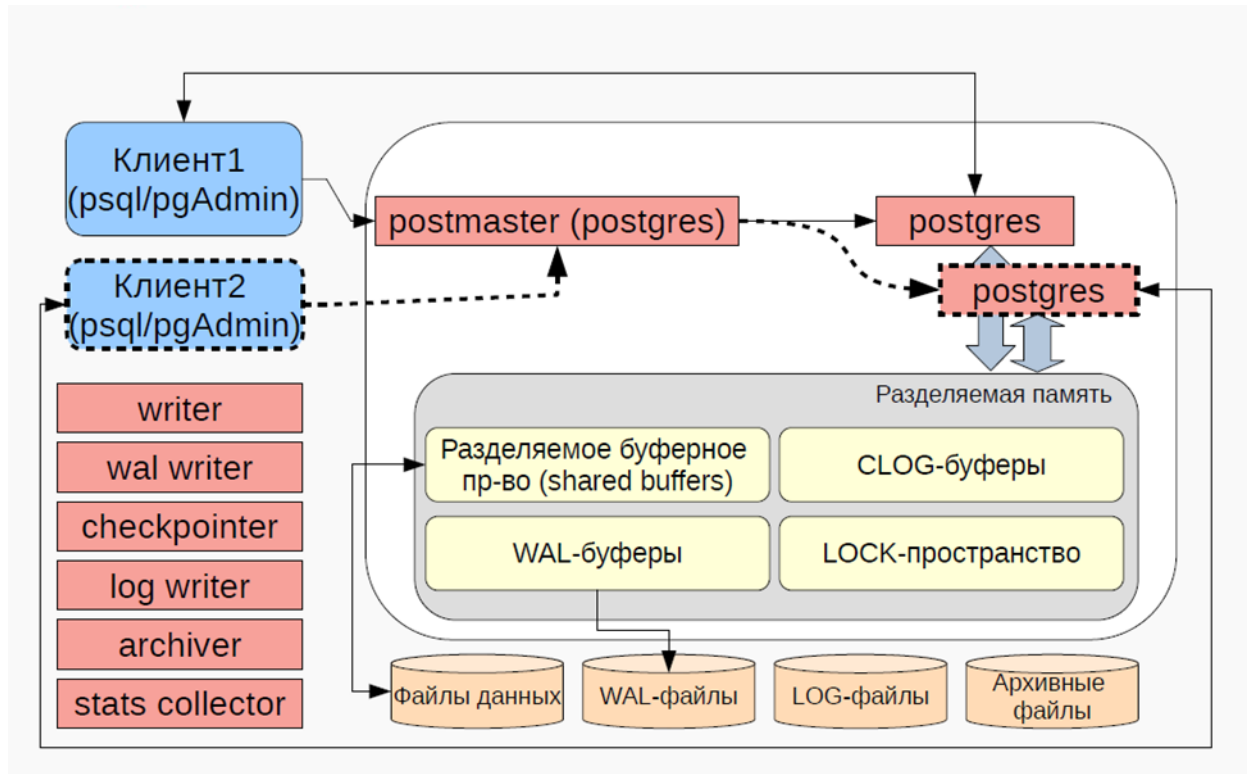
Общее описание базы данных называется *схемой базы данных*:

1. Совокупность схем внешнего уровня – каждая описывает конкретное представление данных
2. Схема концептуального уровня – концептуальная схема – описывает концептуальное представление (все элементы данных и связи между ними, с указанием необходимых ограничений поддержки целостности данных. Для каждой базы данных имеется только одна концептуальная схема.).
3. Внутренняя схема содержит определения хранимых записей, методы представления, описания полей данных, сведения об индексах и выбранных схемах хеширования. Для каждой базы данных существует только одна внутренняя схема.

Независимость данных – изменения на нижних уровнях никак не влияют на верхние уровни:

1. Логическая – обеспечивается за счёт отделения внешнего уровня от концептуального. Изменения концептуальной схемы (добавление, изменение, удаление сущностей и связей) не должны приводить к изменениям внешних схем или переписыванию прикладных программ, для которых эти изменения не предназначены.
2. Физическая – обеспечивается за счёт отделения концептуального уровня от внутреннего. Изменения внутренней схемы (использование различных файловых систем или структур хранения, разных устройств хранения, модификация индексов или хеширование) должны осуществляться без необходимости внесения изменений в концептуальную или внешнюю схемы.

2. Архитектура PostgreSQL.



А что ещё??

3. Разделяемая память PostgreSQL.

Разделяемая память – совместно используется всеми фоновыми и пользовательскими процессами экземпляра PostgreSQL.

Состоит из разделяемого буферного пространства (shared buffers), CLOG-буферов, WAL-буферов и LOCK-пространства.

Shared buffers:

- Хранит копии блоков данных (страниц), считанных из файлов данных.
- Служит для минимизации числа операций обмена с диском.
- Если нужного блока (страницы) данных нет в SB, он читается с диска и помещается в SB.
- Совместно используется всеми фоновыми и пользовательскими процессами экземпляра.
- Можно настроить через shared_buffers.
- По умолчанию 128 MB.

WAL buffers:

- WAL — Write Ahead Log
- Хранит информацию об изменениях данных в БД — записи XLOG.
- Изменениям присваивается LSN — log sequence number.
- Эта информация используется для воссоздания актуального состояния данных в случае восстановления базы данных (например, после сбоя).
- Настраивается через параметр wal_buffers.

CLOG buffers:

- CLOG — Commit Log.
- Хранится статус транзакций: IN_PROGRESS, COMMITTED, ABORTED, SUB_COMMITTED

...	...
350	COMMITTED
351	ABORTED
352	ABORTED
353	COMMITTED
...	...

xid | status

- Размер автоматически устанавливается СУБД.
- Доступен серверным процессам.
- Файлы — в директории `pg_hact`.

Lock space:

- Хранит данные о блокировках, используемых экземпляром БД.
- Данные о блокировках доступны всем серверным процессам.
- Можно настроить через `max_locks_per_transaction`.

4. Буферная память процессов PostgreSQL.

Буферное пространство процессов БД (неразделяемая память):

- Для каждого серверного пользовательского процесса выделено пространство для осуществления операций.
- По умолчанию — 4 MB.
- Может быть различных видов:
 - Vacuum buffers - сборка мусора и опциональный анализ базы данных. Vacuum восстанавливает память, занятую мертвыми кортежами. В обычной работе PostgreSQL кортежи, удаленные или устаревшие в результате обновления, физически не удаляются из их таблицы; они остаются до тех пор, пока не будет выполнено vacuum. Поэтому это нужно делать периодически, особенно на часто обновляемых таблицах.
 - Рабочая память (work_mem) - DISTINCT, ORDER BY, JOIN; Память, которая будет использоваться для внутренних операций сортировки и хеш-таблиц, прежде чем будут задействованы временные файлы на диске. Значение по умолчанию — четыре мегабайта (4MB). В сложных запросах одновременно могут выполняться несколько операций сортировки или хеширования, и при этом указанный объем памяти может использоваться в каждой операции, прежде чем данные начнут вытесняться во временные файлы. Кроме того, такие операции могут выполняться одновременно в разных сеансах. Таким образом, общий объем памяти может многократно превосходить значение work_mem; это следует учитывать, выбирая подходящее значение. Операции сортировки используются для ORDER BY, DISTINCT и соединений слиянием. Хеш-таблицы используются при соединениях и агрегировании по хешу, а также обработке подзапросов IN с применением хеша.
 - Вспомогательная рабочая память (maintenance_work_mem) — REINDEX; Памяти для операций обслуживания БД, в частности VACUUM, CREATE INDEX и ALTER TABLE ADD FOREIGN KEY. По умолчанию его значение — 64 мегабайта (64MB). Так как в один момент времени в сеансе может выполняться только одна такая операция и обычно они не запускаются параллельно, это значение вполне может быть гораздо больше work_mem. Увеличение этого значения может привести к ускорению операций очистки и восстановления БД из копии.
 - Temp_buffer — для работы со временными таблицами. По умолчанию объем временных буферов составляет восемь мегабайт (1024 буфера). Этот параметр можно изменить в отдельном сеансе, но только до первого обращения к временным таблицам; после этого изменить его значение для текущего сеанса не удастся. Сеанс

выделяет временные буферы по мере необходимости до достижения предела, заданного параметром `temp_buffers`. Если сеанс не задействует временные буферы, то для него хранятся только дескрипторы буферов, которые занимают около 64 байт (в количестве `temp_buffers`). Однако если буфер действительно используется, он будет дополнительно занимать 8192 байта (или `BLCKSZ` байт, в общем случае).

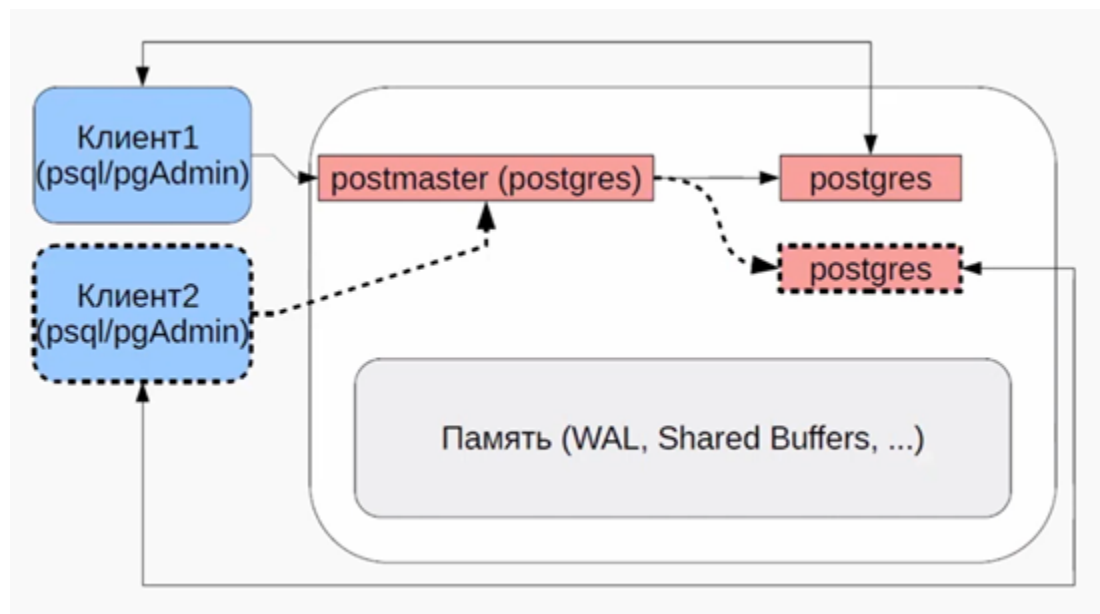
5. Процессы, обеспечивающие работу PostgreSQL.

Два вида:

1. Клиентские пользовательские процессы – запускаются в момент подключения пользователя к БД
2. Процессы СУБД (серверные)
 - Серверные пользовательские процессы: запускаются при установлении сеанса пользователем. Нужны для обработки клиентских запросов.
 - Фоновые процессы: запускаются при запуске экземпляра PostgreSQL. Нужны для поддержки основных функций СУБД.

Процессы, обеспечивающие работу PostgreSQL:

1. postmaster (postgres) – главный процесс: слушает внешние подключения, проводит аутентификацию, создает (форкает) серверные процессы, которые будут обрабатывать запросы клиентов.
2. postgres – серверный пользовательский процесс, обрабатывающий запросы клиента.



Фоновые:

3. writer process (background writer) – процесс записи для синхронизации страниц в shared buffers с файлами данных
 - a. Периодически записывает измененные (заполненные, «грязные») страницы из SB на диск в файлы данных.
 - b. Помечает записанные страницы «чистыми».

- c. «Облегчает» работу процесса checkpointer.
 - d. Bgwriter_delay
- 4. wal writer – периодически проверяет WAL буфер и записывает все незаписанные записи XLOG в сегменты WAL
- 5. checkpointer – работа с контрольными точками:
 - a. Контрольная точка (checkpoint) — точки в последовательности транзакций, в которые произведена синхронизация результатов выполненных операций с файлами на диске.
 - b. Создание контрольной точки:
 - i. WAL-буфер синхронизируется с диском.
 - ii. «Грязные страницы» записываются на диск.
 - iii. Контрольная точка фиксируется в логах.
 - c. Контрольная точка создается, когда заполнен max_wal_size или через время checkpoint_timeout (по умолчанию — 300 секунд) — в зависимости от того, что будет раньше.
- 6. logging collector – записывает сообщения, отправленные в stderr, в лог-файлы (для включения требует установки параметра logging_collector).
- 7. archiver – копирует созданные WAL-файлы в указанное место (по умолчанию выключен)
- 8. stats collector – служит для сбора различных статистических данных о БД (количество данных в таблицах, длительность некоторых операций, а также о работе БД в целом).
 - a. Статистика хранится в промежуточных файлах и через них используется другими процессами.
 - b. Директория с временными файлами определяется параметром stats_temp_directory
 - c. Можно посмотреть в системных каталогах pg_stat_*

6. Системный каталог, организация, способы взаимодействия.

Системный каталог:

- В PostgreSQL есть возможность получения данных о хранимых данных — метаданных:
 - Когда была создана таблица?
 - Сколько в ней атрибутов и какого они типа?
 - Какие индексы связаны с данной таблицей?
- Для доступа к метаданным используются таблицы и представления — системные каталоги;
- У каждой БД — есть схема `pg_catalog`, в ней — каталоги, относящиеся к этой БД.

Работа с системными каталогами:

- Использование системных каталогов напрямую (`SELECT * FROM pg_*`).
- Использование `INFORMATION_SCHEMA`
 - стандартизированный способ работы с системными каталогами (для переносимости кода в разных СУБД).
 - «Под капотом» используются те же системные каталоги
 - Не обладает полной информацией в отличие от использования напрямую (исключаются специфичные для конкретной СУБД данные)
 - Так как по своей сути `INFORMATION_SCHEMA` — это набор дополнительных представлений, которые строятся на основе системных каталогов, — а значит, запросы через неё работают медленнее, чем обращения напрямую.
 - `SELECT Table_Name FROM information_schema.TABLES;`
- Мета-команды `psql (\d, \di...)`.

Примеры:

- `pg_database` — информация о базах данных в кластере БД; создается один для кластера:
- `SELECT datdba FROM pg_database WHERE datname = 'MYDB';`
- `pg_class` — информация о таблицах, представлениях, индексах и тд.
- `pg_tables` — информация о таблицах текущей БД — у каждого БД свой:
- `SELECT * FROM pg_catalog.pg_tables;`
- Функции: `current_user`, `current_schema`, ...

7. Схема, особенности работы со схемами в PostgreSQL, search_path.

Схемы: используются для логической группировки объектов в БД.

- У каждой БД в PostgreSQL есть схема public.
- Полное имя в PostgreSQL: dbName.schemaName.objectName

search_path – последовательность схем, которая будет использована для идентификации объекта, когда используется неполное имя.

- Объекты можно использовать без полного имени — тогда используется search_path.
- Схемы рассматриваются в порядке из search_path.
- Первая схема из search_path — используется для создания объектов — текущая.
- pg_temp и pg_catalog автоматически добавляются в search_path перед первой указанной схемой (порядок можно переопределить).
- Изменять search_path нужно осторожно — меняется контекст выполнения запросов (разрешение имен).
- show search_path; set search_path to myschema, public;

8. Управление доступом к данным в PostgreSQL, привилегии, пользователи, роли.

Пользовательские права:

- Разным категориям пользователей должны предоставляться:
 - разные возможности (в зависимости от их потребностей);
 - для управления различных объектов БД.
- Возможности — обеспечение доступа (или выполнения другой операции) с таблицами, представлениями; создание пользователей.

Предоставляемые возможности определяются **привилегиями**, они могут быть двух видов:

- системные — описывают возможность осуществления операций над БД;
- для взаимодействия с объектами — операции над различными объектами (контроль над данными в объектах БД);
 - с различными объектами связаны различные привилегии.

Работа с привилегиями:

```
GRANT privilegeName1, privilegeName2, ... [ON table1] TO user1, user2, user3;
```

```
GRANT CREATE ON SCHEMA someSchema TO sXXXXXX;
```

- Привилегии: SELECT, INSERT, UPDATE, DELETE, CREATE, EXECUTE, CONNECT, REFERENCES, ...
- ALL PRIVILEGES — для выдачи всех привилегий (в зависимости от контекста).

Владелец объекта — обычно пользователь (роль), создавший объект. Обладает некоторыми привилегиями для созданного объекта по умолчанию (например: ALTER, DROP)

Роли — именованные наборы привилегий. Если привилегии отражают конкретную возможность, то роли позволяют управлять объектами и БД на более «высоком» уровне. Могут выступать в качестве пользователя (роль, имеющая привилегию LOGIN).

Роль — конфигурируется на уровне кластера:

- назначение и привилегии могут отличаться для разных БД;
- Имя роли — уникально для кластера.

Действия с ролями:

- CREATE ROLE:

- CREATE ROLE STUDENT;
- CREATE ROLE STUDENT WITH LOGIN PASSWORD 'somePwd';
- CREATE ROLE STUDADMIN CREATEROLE;
- ALTER ROLE
- DROP ROLE

Эти команды могут использовать суперпользователи, а также роли с параметром CREATEROLE (на не суперпользователях).

Пользователи: роли с параметром LOGIN.

- При установке кластера — создается администратор (postgres).
- Созданы для отображения именованных пользователей системы.
- Могут быть созданы суперпользователем и пользователем с параметром CREATEROLE.

Пример: CREATE USER s234XXX WITH PASSWORD 'somePwd';

При создании БД создается роль public: она назначается всем пользователям и ролям — определяет права пользователей и ролей по умолчанию к разным объектам. По умолчанию имеет привилегии для подключения к любой БД, создавать объекты в схеме public и использовать её объекты, обращаться к системным каталогам, выполнять функции.

Посмотреть список ролей можно в системного каталоге pg_roles.

Настройка ролей:

- SUPERUSER — пользователь может действовать как суперпользователь кластера (все права на все объекты);
 - можно создать нескольких суперпользователей
- NOSUPERUSER — убирает возможности SUPERUSER.
- CREATEROLE — позволяет создавать роли.
- CREATEDB — позволяет создавать базы в кластере.
- PASSWORD - установить пароль:
- PASSWORD NULL - запретить пользователю вход по паролю.
- CONNECTION LIMIT n — задать ограничение по числу подключений для пользователя.
- VALID UNTIL — установить срок действия роли (срок действия пароля этой роли).

Группы ролей: роли можно объединять в группы для более гибкого управления ими:

CREATE ROLE STUDENTS;

CREATE ROLE s123456 WITH LOGIN PASSWORD 'somsdfslld';

GRANT STUDENTS TO s123456;

или CREATE ROLE s123457 WITH LOGIN PASSWORD 'xfsewrew' IN ROLE students;

Роль-администратор группы может добавлять новых участников в группу (как и суперпользователь кластера).

- Участник группы (или несколько участников группы) могут быть администраторами:
- CREATE ROLE students WITH NOLOGIN ADMIN s123456;
- Или с помощью GRANT:
- GRANT students TO s123456 WITH ADMIN OPTION;
- Роль должна быть создана, чтобы быть добавлена в качестве администратора.

9. Работа с ролями, INHERIT, NOINHERIT.

Работа с ролями:

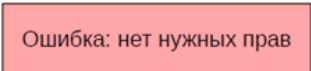
- WITH GRANT OPTION — указывается возможность дальнейшей передачи роли от того, кому она назначена:
- GRANT UPDATE ON STUDENT TO s458455 WITH GRANT OPTION;
- INHERIT/NOINHERIT — наследование привилегий при работе с группами.
- SET ROLE

Удаление группы не удаляет ее участников.

INHERIT — участники группы получают права ролей-групп их окружающих. Параметр INHERIT добавляется по умолчанию.

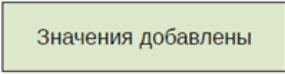
NOINHERIT — следует задавать явно (у участника группы), тогда привилегии группы роли-группы наследоваться не будет.

```
CREATE ROLE s123457 WITH LOGIN PASSWORD  
'somsdfslid' NOINHERIT;  
GRANT INSERT ON stud_comments TO STUDENTS;  
GRANT STUDENTS TO s123457;  
  
psql -U s123457 ucheb  
  
INSERT INTO stud_comments VALUES ...;
```



Но даже при наличии NOINHERIT, если нам нужно получить привилегии группы, в которой мы состоим, можно явно прописать **SET ROLE**:

```
psql -U s123457 ucheb  
  
SET ROLE TO STUDENTS;  
  
INSERT INTO stud_comments VALUES ...;
```



Поиск привилегий:

1. Поиск среди привилегий роли.
2. Поиск среди родителей (если у изначальной роли INHERIT). Поиск среди прародителей (если у родителей INHERIT).
3. Есть ли привилегия для роли public.

Отмена привилегий:

REVOKE privilegeName1, privilegeName2, ... [ON table1] FROM user1, user2, user3;

Пример:

REVOKE UPDATE, DELETE, TRUNCATE ON STUDENT FROM s4343453;

10. Установка и запуск PostgreSQL.

Есть 2 варианта установки:

- Через графический установщик
- С использованием утилит
 - Использование подготовленных бинарных файлов - разные сборки для разных ОС
 - Компиляции исходных кодов (make) - если для используемой операционной системы нет бинарников либо если нужна самая последняя версия / версия с внесенными изменениями

С точки зрения Postgres есть несколько базовых компонентов:

- сервер (postgresql)
 - клиент (postgresql-client)
 - contrib, docs и другие вспомогательные компоненты
1. *Установка Postgresql* - Установка (сборка) базовых компонентов (пакетов): `sudo apt install postgresql-XX postgresql-client-XX ...`
 2. *Подготовка места для кластера баз данных* - Создание системного пользователя (postgres):
`adduser postgres`
`mkdir [PGDATA]` - создание директории PGDATA, где будут расположен кластер базы данных (все связанные с этим файлы)
`chown postgres [PGDATA]` - установка соответствующих прав для пользователя postgres
 3. Задание переменных окружения: PGDATA (путь до директории PGDATA), ...
 4. Существует специальный скрипт `initdb`, с помощью которого создаются каталоги и подкаталоги кластера бд `[/pg_path]/bin/initdb [-D PGDATA_path]`
 5. Запуск экземпляра кластера БД, несколько вариантов:
 - a. `[/pg_path]/bin/postgres [-D PGDATA_path]`
 - b. `[/pg_path]/bin/pg_ctl [-D PGDATA_path] -l logfile start` - неявно утилита использует предыдущую команду `postgres`, но по умолчанию работает в фоне
 - c. Сервис: `sudo service postgresql start` - можно добавить в сервисы для автозапуска на старте или запускать/перезапускать сервис бд стандартными средствами ОС

Инициализация кластера через **initdb**

- Создает структуру директорий PGDATA
- Создает стандартные БД, которые можно использовать для создания своих баз
- Задаёт локаль и кодировку, которая будет использоваться кластером БД

Варианты запуска:

1. `initdb [-D PGDATA_path]`
2. `pg_ctl initdb [-D PGDATA_path]`

Утилита **pg_ctl**

Для упрощения взаимодействия с postgres.

- Можно запускать (по умолчанию в фоне), останавливать, перезапускать кластер:
pg_ctl start -l logfile
pg_ctl status [-D PGDATA_path]
- Можно использовать для инициализации кластера: pg_ctl [-D PGDATA_path] initdb

11. PostgreSQL: template0, template1, назначение, особенности работы.

Базы данных, доступные после установки:

- **пользовательская БД (postgres)** - принадлежит администратору БД - пользователю postgres;
- **создается template1** - это шаблон, на основе которого создаются новые базы. Если добавить объекты в template1, то впоследствии они будут копироваться в новые базы данных. Это позволяет внести изменения в стандартный набор объектов.
- **создается клон template1 — template0** - при инициализации она содержит те же самые объекты, что и template1, предопределённые в рамках устанавливаемой версии PostgreSQL. Не нужно вносить никаких изменений в template0 после инициализации кластера. Если в команде CREATE DATABASE указать на необходимость копирования template0 вместо template1, то на выходе можно получить «чистую» пользовательскую базу данных без изменений, внесённых в template1. Это удобно, когда производится восстановление из дампа данных с помощью утилиты pg_dump: скрипт дампа лучше выполнять в чистую базу, во избежание каких-либо конфликтов с объектами, которые могли быть добавлены в template1. Также можно указать новые параметры локали и кодировку при копировании template0, в то время как для копий template1 они не должны меняться.

Создается суперпользователь — по умолчанию совпадает с именем пользователя, который запустил initdb;

- можно задать имя через -U: initdb -U имя

Изначально template1 и template0 — идентичны.

template0 — служит в виде резервной копии.

template1 — используется в качестве шаблона при создании других БД.

Можно создать свою БД, которая будет использоваться в качестве шаблона.

Если внести изменения в template1 — они будут применены и для баз, созданных на его основе.

К template0 — нельзя подключиться (она резервная).

Особенности использования шаблонов:

- Чтобы база могла быть шаблоном нужно, чтобы к ней не было подключений других пользователей
- Для гарантии этого используются два атрибута БД в системном каталоге **pg_database**:
 - **datistemplate** (bool) — показывает была ли создана БД, чтобы быть шаблоном;
может быть использована (как шаблон) владельцем, суперпользователями и пользователями с CREATEDB;

Если true, базу данных сможет клонировать любой пользователь с правами CREATEDB; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.

- Обычно установлен на template1 и template0
- **dataallowconn** (bool) — запрещаются новые подключения. Если false, никто не сможет подключаться к этой базе данных
 - Обычно установлен на template0 для избегания модификаций

12. Подключение к PostgreSQL, настройка, особенности.

Для подключения существуют несколько видов клиентов:

1. psql:
 - интерактивный терминал PostgreSQL;
 - стандартный клиент для работы с PostgreSQL;
2. pgAdmin:
 - графический клиент;
 - *доступен для Windows, Unix, macOS;
3. Клиенты, использующие libpq

Для подключения к БД роль должна быть:

- LOGIN;
- содержать привилегию CONNECT на нужную БД;
- разрешение в pg_hba.conf;

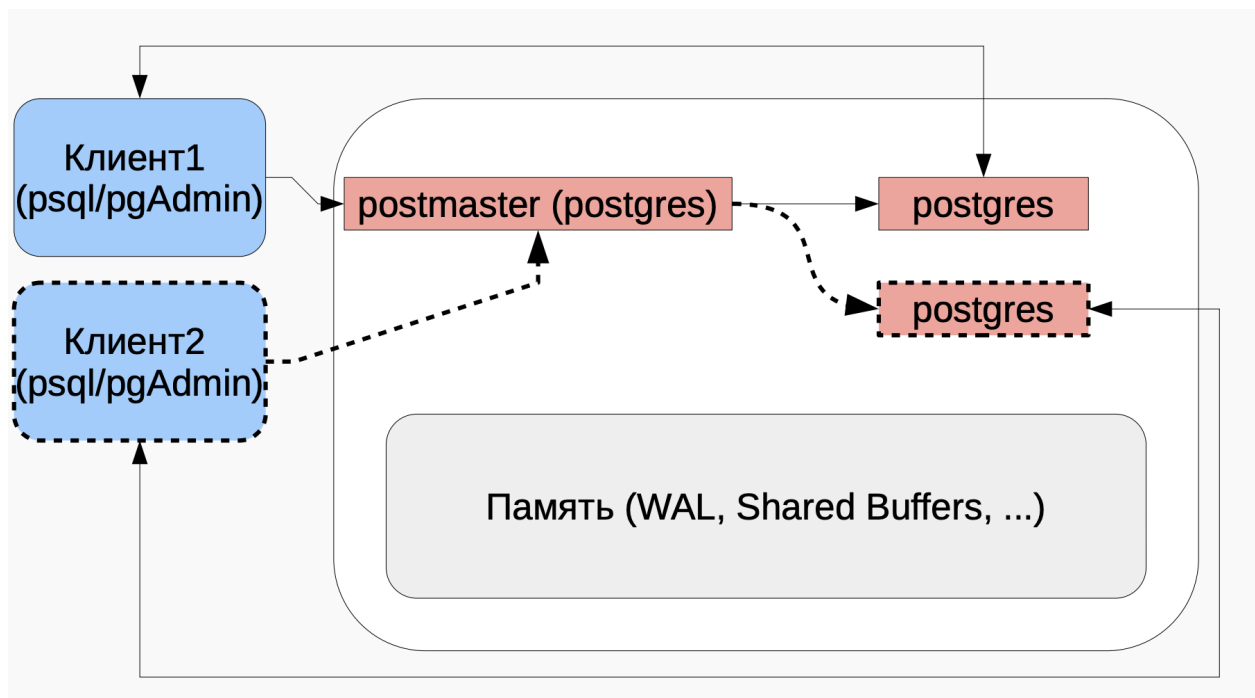
Конфигурационный файл **pg_hba.conf**

В pg_hba.conf задаются способы подключения для различных пользователей к различным базам данных.

Файл создается при работе initdb. По умолчанию располагается в PGDATA.

Можно поменять через hba_file параметр:

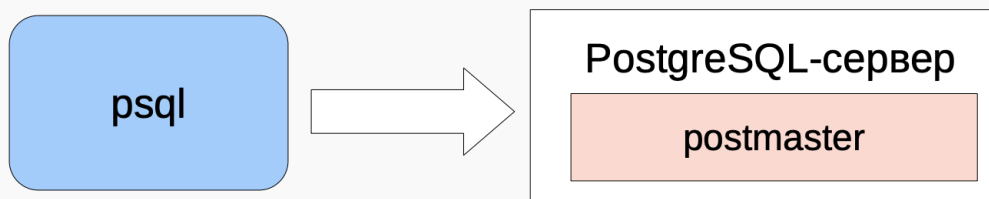
- при запуске postgres;
- в postgresql.conf;



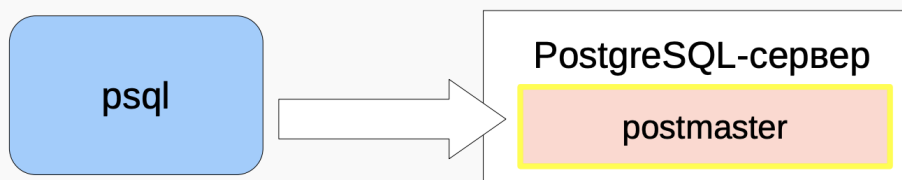
- 1) Запуск сервера PostgreSQL:
`postgres -p 2378 -D /home/myuser/pgd/data`



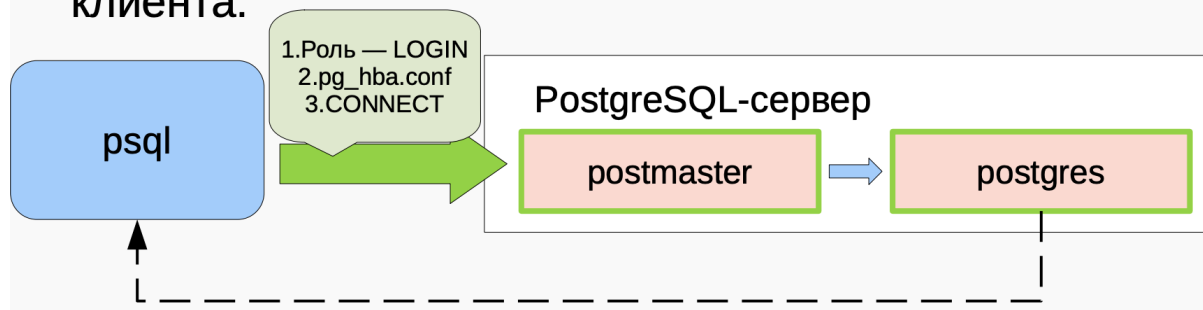
- 2) Подключаем клиент: `psql -p 2347`



- 3) экземпляр проверяет, может ли клиент получить доступ:



- 4) Если проверка прошла успешно, создается серверный процесс для обработки соединения данного клиента:



13. Виды и методы подключений к PostgreSQL, их настройка.

В конфигурационном файле **pg_hba.conf** задаются способы подключения для различных пользователей к различным базам данных.

Файл создается при работе initdb. По умолчанию располагается в PGDATA.

Можно поменять через hba_file параметр:

- при запуске postgres;
- в postgresql.conf;

Файл представляет собой набор записей. Каждая запись в файле определяет вид подключения для разных категорий пользователей. Порядок расположения записей влияет на права:

- чтение происходит последовательно;
- pg_hba.conf читается во время запуска;
- Если файл изменен:
 - pg_reload_conf() - перезагружает конфигурационные параметры
 - pg_ctl reload - перезагрузка Postgresql

Пример конфигурации pg_hba.conf

```
# PostgreSQL Client Authentication Configuration File
...
# local   DATABASE USER METHOD      [OPTIONS]
# host    DATABASE USER ADDRESS    METHOD [OPTIONS]
...
# "local" is for Unix domain socket connections only
local    all     all peer
# IPv4 local connections:
host     all     all 127.0.0.1/32 md5
...
```

1. Вид подключения

- Локальный UNIX-сокет (local) - там же, где и установлен postgresql:
 - psql (без имени хоста), используется libpq:
 - подключается пользователь системы к БД с тем же именем, а также есть пользователь БД с тем же именем.
 - клиент и сервер на одной машине;

- параметр `unix_socket_directories` - позволяет разместить файл сокета в должным образом защищенном каталоге
- TCP/IP соединение (host):
 - указывается хост, порт:
`psql -h host -p port database`
`psql -h host -p port -U username database`
 - Предполагается, что задан параметр PGDATA.
- 2. База данных
- 3. Пользователь
- 4. IP-адрес (для host подключения)
- 5. Метод доступа
- 6. Опции

Методы подключения

- **trust** — предоставить доступ всем из данной категории (опасно!). Любой подключающийся к серверу авторизован для доступа к базе данных вне зависимости от указанного имени пользователя базы данных (даже если это имя суперпользователя). Можно использовать только для локальных подключений
- **по паролю** (`scram-sha-256`, `md5`, `password`):
 - Эти методы действуют похожим образом; отличие состоит только в том, как передается пароль по каналу связи, а именно: в виде хеша `scram-sha-256`, MD5, или открытым текстом, соответственно
 - База данных паролей PostgreSQL отделена от паролей пользователей операционной системы. Пароль для каждого пользователя базы данных хранится в системном каталоге `pg_authid`.
 - Смена пароля:
 - изменение роли - ALTER и тогда нужно очищать логи, чтобы там не остался пароль
 - `\password` - команда в psql
- **ident** — похоже на `peer`, но для tcp/ip (host). Метод аутентификации `ident` работает, получая имя пользователя операционной системы клиента от сервера `ident` и используя его в качестве разрешенного имени пользователя базы данных (с возможным сопоставлением имени пользователя). Для этого добавляется параметр конфигурации `map` в `pg_hba.conf`.
- **peer** — для `local`, сравнивается пользователь ОС с пользователем БД:
 - можно задать правила отображения пользователей в `pg_ident.conf` и параметр `map` в `pg_hba.conf`;

14. PostgreSQL, создание Базы Данных.

Есть два варианта:

- С помощью команды CREATE DATABASE:
CREATE DATABASE newDb1 [WITH опции];
- Использование createdb — утилита в директории [/postgresql]/bin - чтобы не подключаться к стандартной бд postgres:
createdb -h host -p port -U user [опции] newDb2;

Чтобы создать базу пользователь должен быть:

- суперпользователем;
- обладать ролью CREATEDB;

Создание базы из указанного шаблона

По умолчанию при создании новой базы данных копируется template1.

Но можно использовать в качестве шаблона другую базу, указав ключевое слово template или опцию -T:

1. **CREATE DATABASE** newDb3 **TEMPLATE** newDB1;
2. **createdb** -T newDB2 newDB4

Особенности использования шаблонов:

- Чтобы база могла быть шаблоном нужно, чтобы к ней не было подключений других пользователей
- Для гарантии этого используются два атрибута БД в системном каталоге **pg_database**:
 - **datistemplate** (bool) — показывает была ли создана БД, чтобы быть шаблоном;
может быть использована (как шаблон) владельцем, суперпользователями и пользователями с CREATEDB;
Если true, базу данных сможет клонировать любой пользователь с правами CREATEDB; в противном случае клонировать эту базу смогут только суперпользователи и её владелец.
 - **dataallowconn** (bool) — запрещаются новые подключения. Если false, никто не сможет подключаться к этой базе данных

15. Файловая структура и конфигурация PostgreSQL.

PGDATA – путь к директории данных кластера, установлена заранее. Помимо того, что в этой директории лежат конфигурационные файлы - например, postgresql.conf, pg_hba.conf, а ней существует определенная структура подкаталогов:

- **global** – содержит таблицы уровня кластера (pg_database) - файлы, которые связаны с системными каталогами, относящимися к кластеру postgres целиком
- **pg_wal** – директория с WAL-файлами.
- **pg_xact** – директория с данными о коммитах транзакций (CLOG).
- **base** – файлы данных базы данных кластера (количество поддиректорий в этой директории равно количеству существующих баз данных). Список oid, соответствующих базам. По факту числа равны filenode'ам. Если посмотреть после инициализации содержимое директории, то, например:
 - 1 (соответствует postgres)
 - 13201 (template 0)
 - 13202 (template 1)

Для просмотра имен (по oid или filenode) можно использовать утилиту oid2name:

oid2name

Oid	Database Name	Tablespace

13202	template1	pg_default
13201	template0	pg_default
1	postgres	pg_default

Объектам БД (PostgreSQL) — соответствуют **файлы данных**:

- если до 1Gb (по умолчанию): объекту соответствует 1 файл данных;
- если > 1Gb: объекту соответствуют файлы-сегменты;

Находятся в *\$PGDATA/base* в директории соответствующей БД.

Базовые конфигурационные файлы

Стандартный путь, по которому располагаются конфигурационные файлы:

/etc/postgresql/версия/main/

- **postgresql.conf** — базовый файл для хранения настроек. Создается при работе initdb. Представляет из себя набор параметров (имя, значение):
`search_path = 'someSchema, public'`
`shared_buffers = 128MB`
 Определяют значения для всех БД кластера.
- **postgresql.auto.conf** — динамически изменяемые настройки (через ALTER SYSTEM).
- **pg_hba.conf** — конфигурация подключений к БД.
- **pg_ident.conf** — файл отображения имен.

Изменение конфигурационных параметров

1. postgresql.conf/ALTER SYSTEM - на уровне кластера;
2. ALTER DATABASE - изменить 1. на уровне базы данных; *// Значения применяются*
3. ALTER ROLE - переписать 1.и 2. на уровне пользователя/*при запуске новой сессии*
4. SET - изменить для сессии: *SET param TO value/DEFAULT;*
5. Параметры можно задать в команде запуска сервера БД (postgres) - для переписи параметров postgresql.conf.

16. Табличные пространства. Назначение. Организация и настройка.

- Табличные пространства — позволяют задать пути (директорию в файловой системе), где будут храниться объекты БД (вне PGDATA):
 - позволяют управлять физическим расположением объектов БД.
- У табличных пространств есть имя.
- Просмотр размера табличного пространства:
 - Функция `pg_tablespace_size('имя_мп')`
- Системный каталог: `pg_tablespace` содержит информацию о табличных пространствах
- `$PGDATA/pg_tblspc` содержит символические ссылки на директории внешних табличных пространств.

Табличные пространства можно задать для разных объектов БД: таблиц, индексов, последовательностей, представлений.

```
CREATE TABLESPACE newTablespace LOCATION '/diskA/someDir';
```

```
CREATE TABLE STUDENT (  
    id integer NOT NULL  
) TABLESPACE newTablespace;
```

Стандартные табличные пространства

- **pg_default** — соответствует директории `base` в PGDATA;
 - по умолчанию здесь хранятся файлы данных, соответствующие объектам БД.
- **pg_global** — соответствует директории `global` в PGDATA - общие для всех баз данных кластера;
 - `pg_database`, `pg_authid`, `pg_tablespace`, некоторые другие каталоги и индексы

Табличное пространство можно задать для БД целиком:

```
CREATE DATABASE TestDB TABLESPACE newTablespace;
```

Табличное пространство может быть изменено (пересоздано в новом табличном пространстве):

```
ALTER TABLE STUDENT SET TABLESPACE pg_default;
```

Назначение табличных пространств

- Можно контролировать что где хранится.
- Критически важные объекты можно размещать на более быстрых физических дисках.
- Если заканчивается место на жестком диске, можно использовать другой диск для размещения объектов.

17. Транзакции. Назначение. ACID.

- Транзакции объединяют последовательность действий в одну операцию.
- Промежуточные состояния внутри последовательности операций не видны другим транзакциям.
- Если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Транзакция - группа последовательных операций с базой данных, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще, и тогда она не должна произвести никакого эффекта.

Стандартный **пример**: перевод средств с одного банковского счета на другой.

Одним из наиболее распространённых наборов требований к транзакциям и транзакционным системам является набор ACID (Atomicity, Consistency, Isolation, Durability).

Atomicity (атомарность):

- гарантирует, что результаты работы транзакции не будут зафиксированы в системе частично;
- будут либо выполнены все операции в транзакции, либо не выполнено ни одной.

Consistency (согласованность):

- после выполнения транзакции база данных должна быть в согласованном (целостном) состоянии;
- во время выполнения транзакции (после выполнения отдельных операций в рамках транзакции) согласованность не требуется.

Isolation (изолированность):

- во время выполнения транзакции другие транзакции (выполняющиеся параллельно) не должны оказывать влияние на результат транзакции.

Durability (долговечность):

- при успешном завершении транзакции результаты ее работы должны остаться в системе независимо от возможных сбоев оборудования, системы и тд.

Пример:

BEGIN;

UPDATE BANK_ACCOUNT SET AccValue = AccValue+1000 WHERE Client_ID = 1;

UPDATE BANK_ACCOUNT SET AccValue = AccValue-1000 WHERE Client_ID = 2;

COMMIT;

Также можно использовать ROLLBACK и SAVEPOINT

18. PostgreSQL: явные, неявные транзакции. Транзакционный DDL.

- **Неявные (implicit)** — СУБД начинает без явного указания;
 - Все sql-выражения (DML) выполняются в контексте некоторой транзакции:

```
BEGIN;  
SELECT * FROM STUDENTS; -> SELECT * FROM STUDENTS;  
COMMIT;  
  
BEGIN;  
INSERT INTO STUDENTS VALUES; -> INSERT INTO STUDENTS VALUES;  
COMMIT;
```
 - Есть транзакционный DDL (в отличие от других СУБД)

```
BEGIN;  
CREATE TABLE STUDENTS (StudId int, Name text);  
CREATE TABLE EXAMS (ExamId int, ExName text);  
ROLLBACK;
```

Можно поменять структуру базы данных в рамках транзакции и откатить изменения, можно создавать несколько таблиц или не создавать ни одной за одну транзакцию
- **Явные (explicit)** — транзакции, которые задает пользователь самостоятельно (BEGIN, COMMIT).
- Если нет явного указания начала транзакции, любое предложение выполняется в рамках неявной транзакции;

19. PostgreSQL: время начала транзакции, время внутри транзакции.

Время можно вывести с помощью системной функций **CURRENT_TIME**, **CURRENT_TIMESTAMP**, **now()**

Пример:

```
SELECT CURRENT_TIME as my_time
```

Если функция вызвана внутри транзакции, то выводится время начала транзакции:

```
BEGIN;  
SELECT CURRENT_TIME as my_time_1;  
-- 13:30:49. ...  
-- [ ... ONE HOUR LATER ... ]  
SELECT CURRENT_TIME as my_time_2;  
-- 13:30:49. ...  
COMMIT;
```

- **CURRENT_TIME** — возвращает время начала транзакции (CURRENT_TIMESTAMP, transaction_timestamp);
- **clock_timestamp()** — возвращает текущее время (timestamp);

Поэтому, если нужно знать конкретно текущее время в транзакции, то:

```
BEGIN;  
SELECT CURRENT_TIME as my_time_1;  
-- 13:30:49. ...  
-- [ ... ONE HOUR LATER ... ]  
SELECT clock_timestamp() as my_time_2;  
-- 2022-04-25 14:30:49. ...  
COMMIT;
```

20. PostgreSQL: идентификация транзакции, особенности работы xid

Идентификатор транзакции (xid) — назначается СУБД для каждой новой транзакции;

- xid — уникален
- 32 бита, беззнаковое число
- xid — начинается с 3 (значения 0, 1, 2 — зарезервированы)

При создании/модификации записи хранится xid транзакции.

Функции для получения xid:

- **pg_current_xact_id()** - возвращает xid текущей транзакции (bigint):
 - *SELECT pg_current_xact_id();* -- 7435
 - txid_current() и txid_current_if_assigned() - старые названия
 - если у текущей транзакции ещё нет идентификатора (она не успела выполнить какие-либо изменения в базе), он будет ей назначен
- **pg_current_xact_id_if_assigned()** - выдаёт идентификатор текущей транзакции или NULL, если она ещё не имеет идентификатора

Например, здесь выводятся разные xid в рамках новых транзакций для каждого select запроса

```
SELECT Column1, pg_current_xact_id() FROM TABLE1; -- (1) ???
```

```
SELECT Column2, pg_current_xact_id() FROM TABLE2; -- (2) ???
```

В таблицах хранятся не только пользовательские столбцы, которые выводятся с помощью селектора *, но и скрытые системные. Некоторые из них:

- **xmin** — хранит xid транзакции, в рамках которой запись (копия записи) была создана;
- **xmax** — хранит xid транзакции, в рамках которой запись была удалена;
- **cmin** — идентификатор команды, создавшей запись;
- **cmx** — идентификатор команды, удалившей запись;

Пример:

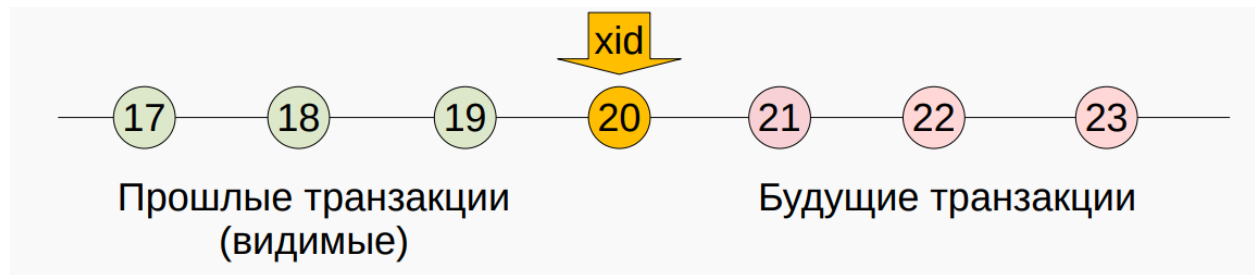
```
SELECT xmin, * FROM STUDENTS;
```

STUDENTS

xmin	Stud_ID	Name	Surname	Group
345	1	Ivan	Ivanov	33313
345	2	Petr	Petrov	33314

xid транзакции

В простейшем случае: чем больше xid транзакции — тем позже она началась:



Но если перешли через MAX значение, то возникает xid wraparound problem.

Решение - представление последовательности xid в виде кольца, где часть записей - текущие для будущих транзакций, а часть архивные:

- Установка статуса неактуальных записей:
 - Статус FROZEN — для неактуальных записей (~ транзакции для которых уже давно завершены) - можно переиспользовать для новых транзакций
- Экономия xid:
 - если в транзакции нет операций, изменяющих состояние БД, ей выдается виртуальный xid
- WARNING: database test must be vacuumed within [...] transactions ...
 - Можно освободить xid'ы с помощью vacuum

Пример:

```
BEGIN;  
SELECT * FROM STUDENTS;  
UPDATE STUDENTS SET Group = 3100 ... ; //xid назначается только здесь  
COMMIT;
```

Проверить, назначен ли xid, можно с помощью команды `txid_current_if_assigned()`. Если не назначен - выведет NULL

21. PostgreSQL: MVCC, SSI

MVCC - Multi-Version Concurrency Control (Многоверсионное управление конкурентным доступом).

В **PostgreSQL** - это **SSI** (Serializable Snapshot Isolation).

Каждый **SQL-оператор** видит снимок данных (версию базы данных) на определённый момент времени, вне зависимости от текущего состояния данных. Это защищает операторы от несогласованности данных, возможной, если другие конкурирующие транзакции внесут изменения в те же строки данных, и обеспечивает тем самым изоляцию транзакций для каждого сеанса баз данных. **MVCC**, отходя от методик блокирования, принятых в традиционных СУБД, снижает уровень конфликтов блокировок и таким образом обеспечивает более высокую производительность в многопользовательской среде.

Основное преимущество использования модели **MVCC** по сравнению с блокированием заключается в том, что блокировки **MVCC**, полученные для чтения данных, не конфликтуют с блокировками, полученными для записи, и поэтому чтение никогда не мешает записи, а запись чтению. PostgreSQL гарантирует это даже для самого строгого уровня изоляции транзакций, используя инновационный уровень изоляции **SSI** (Serializable Snapshot Isolation, Сериализуемая изоляция снимков).

Особенности **MVCC (SSI)**:

- При изменении данных в транзакции - создается новая копия данных
- Существующие копии данных не изменяются
- При выборке данных берется подходящая для данной транзакции версия данных

Снимок (snapshot) - характеризует временное окно видимости данных для транзакции:

- Каждая транзакция видит свой набор данных, определяемый связанным с ней снимком
- Снимок: какие xid (версии данных) видит данная транзакция
- Можно узнать через `pg_current_snapshot()`
 - Возвращает текущий снимок
 - Формат: **xmin:xmax:xid1,xid2**
 - **xmin** - минимальный идентификатор транзакции среди всех активных. Все транзакции, идентификаторы которых меньше xmin, уже либо зафиксированы и видимы, либо отменены и «мертвы».
 - **xmax** - идентификатор на один больше идентификатора последней завершённой транзакции. Все транзакции, идентификаторы которых больше или равны xmax, на момент получения снимка ещё не завершены и не являются видимыми.
 - **xid1, xid2** - транзакции, выполняющиеся в момент получения снимка. Транзакции с такими идентификаторами, для которых xmin <= ид <

хтах, не попавшие в этот список, были уже завершены на момент получения снимка.

- Верхняя граница (хтах - не включается)

Изолированность. MVCC



MVCC

MultiVersion Concurrency Control

- Разные пользователи могут одновременно работать с одними и теми же данными;
- Каждый пользователь видит свой изолированный срез данных;
- Изменения, вносимые пользователем, никому не видны до завершения транзакции.


Tr 1 (xid = 321):	Tr 2 (xid = 322):				
BEGIN REP. READ;					
	BEGIN READ COM.;				
Snapshot: 321:321	Snapshot: 321:321				
SELECT * FROM STUDENTS;	SELECT * FROM STUDENTS;	Stud_ID	Name	Surname	Group
Snapshot: 321:321		1	Ivan	Ivanov	33313
UPDATE STUDENTS SET Name = 'Egor' WHERE Stud_ID = 1;					
	Snapshot: 321:321				
	SELECT * FROM STUDENTS;	Stud_ID	Name	Surname	Group
		1	Ivan	Ivanov	33313
COMMIT;					
	Snapshot: 322:322				
	SELECT * FROM STUDENTS;	Stud_ID	Name	Surname	Group
		1	Egor	Ivanov	33313
	COMMIT;				

22. Виды и особенности конфликтов при параллельном доступе к данным.

При организации параллельного доступа к данным могут возникнуть такие проблемы как:

- Грязное чтение ("Dirty read")
- Неповторяющееся чтение ("Non-repeatable read")
- Фантомное чтение ("phantom read")
- Потерянное обновление ("Lost update") - доп
- Аномалии сериализации - доп

Грязное чтение - проблема возникает, если одна транзакция видит изменения данных другой (незавершенной транзакции).

Transaction 1:	Transaction 2:				
BEGIN;					
	BEGIN;				
UPDATE STUDENTS SET Group = 33314 WHERE Stud_ID = 1;					
	SELECT * FROM STUDENTS;	STUDENTS			
ROLLBACK;		Stud_ID	Name	Surname	Group
		1	Ivan	Ivanov	33314
	COMMIT;				

Неповторяющееся чтение - проблема возникает, если один и тот же запрос в рамках транзакции возвращает разные результаты.

Transaction 1:	Transaction 2:				
BEGIN;					
	BEGIN;				
	SELECT * FROM STUDENTS;	STUDENTS			
		Stud_ID	Name	Surname	Group
		1	Ivan	Ivanov	33313
UPDATE STUDENTS SET Group = 33314 WHERE Stud_ID = 1;					
COMMIT;		STUDENTS			
	SELECT * FROM STUDENTS;	Stud_ID	Name	Surname	Group
		1	Ivan	Ivanov	33314
	COMMIT;				

Фантомное чтение - проблема возникает, если один и тот же запрос в рамках транзакции возвращает разное число записей (сами данные не изменяются, а добавляются).

Transaction 1:	Transaction 2:	<div>Сами данные не изменяются</div>								
BEGIN;										
	BEGIN;									
	SELECT * FROM STUDENTS;	STUDENTS								
INSERT INTO STUDENTS VALUES (2, 'Viktor', 'Ivanov', 33315);		<table><tr><th>Stud_ID</th><th>Name</th><th>Surname</th><th>Group</th></tr><tr><td>1</td><td>Ivan</td><td>Ivanov</td><td>33313</td></tr></table>	Stud_ID	Name	Surname	Group	1	Ivan	Ivanov	33313
	Stud_ID	Name	Surname	Group						
1	Ivan	Ivanov	33313							
COMMIT;		STUDENTS								
	SELECT * FROM STUDENTS;	<table><tr><th>Stud_ID</th><th>Name</th><th>Surname</th><th>Group</th></tr><tr><td>1</td><td>Ivan</td><td>Ivanov</td><td>33313</td></tr></table>	Stud_ID	Name	Surname	Group	1	Ivan	Ivanov	33313
Stud_ID	Name	Surname	Group							
1	Ivan	Ivanov	33313							
	COMMIT;	<table><tr><td>2</td><td>Viktor</td><td>Viktorov</td><td>33315</td></tr></table>	2	Viktor	Viktorov	33315				
2	Viktor	Viktorov	33315							

Дополнительно:

Потерянное обновление - проблема возникает в случае пере затирания изменений другой транзакцией до завершения транзакции сделавшей изменения.

Транзакция 1	Транзакция 2
UPDATE tbl1 SET f2=f2+20 WHERE f1=1;	
	UPDATE tbl1 SET f2=f2+25 WHERE f1=1;

Аномалии сериализации - ситуация, когда параллельное выполнение транзакций приводит к результату, невозможному при последовательном выполнении тех же транзакций.

Транзакция 1	Транзакция 2
SELECT SUM(value) FROM mytab WHERE class = 1; ----- 30 (1 row)	SELECT SUM(value) FROM mytab WHERE class = 2; ----- 300 (1 row)
INSERT INTO mytab (value, class) VALUES (30, 2)	INSERT INTO mytab (value, class) VALUES (300, 1)
COMMIT;	COMMIT;

23. Изоляция транзакций. Режимы для организации доступа к данным.

Уровень изолированности транзакций — условное значение, определяющее, в какой мере в результате выполнения логически параллельных транзакций в СУБД допускается получение несогласованных данных. Шкала уровней изолированности транзакций содержит ряд значений, ранжированных от наимизшего до наивысшего; более высокий уровень изолированности соответствует лучшей согласованности данных, но его использование может снижать количество физически параллельно выполняемых транзакций. И наоборот, более низкий уровень изолированности позволяет выполнять больше параллельных транзакций, но снижает точность данных.

Таким образом, выбирая используемый уровень изолированности транзакций, разработчик информационной системы в определённой мере обеспечивает выбор между скоростью работы и обеспечением гарантированной согласованности получаемых из системы данных.

По **SQL-стандарту** существует 4 уровня изоляции (чем ниже уровень, тем строже)

1. **Read Uncommitted** (не поддерживается в PostgreSQL) - чтение незафиксированных данных
2. **Read Committed** (по-умолчанию) - чтение фиксированных данных
3. **Repeatable Read** - повторяемость чтения
4. **Serializable** - упорядочиваемость

	Грязное чтение	Неповторяющееся чтение	Фантомное чтение	Аномалии сериализация
Read Uncommitted	+	+	+	+
Read Committed	-	+	+	+
Repeatable read	-	-	+ (стандарт) - (PostgreSQL)	+ (write skew / read-only transaction skew)
Serializable	-	-	-	-

Уровень изоляции	Потерянное обновление	«Грязное» чтение	Неповторяющееся чтение	Фантомное чтение	Аномалии сериализации
Read uncommitted	не возможно	допускается	возможно	возможно	возможно
Read committed	не возможно	не возможно	возможно	возможно	возможно
Repeatable read	не возможно	не возможно	не возможно	допускается	возможно
Serializable	не возможно	не возможно	не возможно	не возможно	не возможно

Как выбрать режим изоляции транзакции:

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Или

BEGIN;

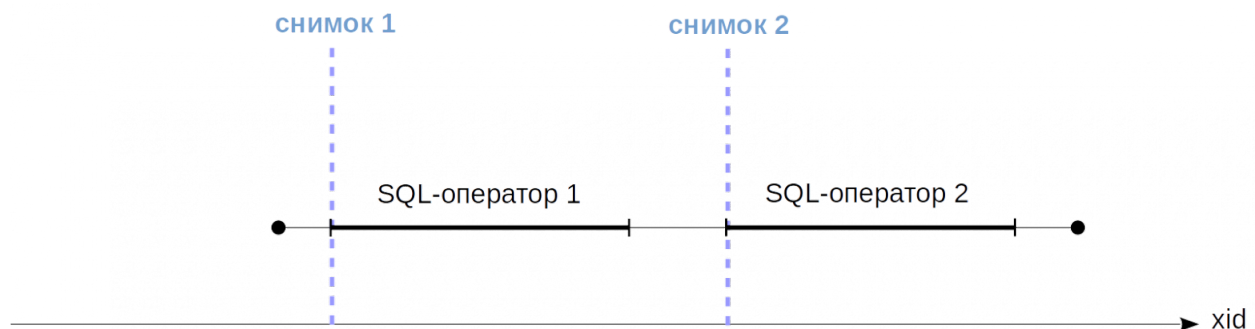
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

(нельзя менять по ходу транзакции!)

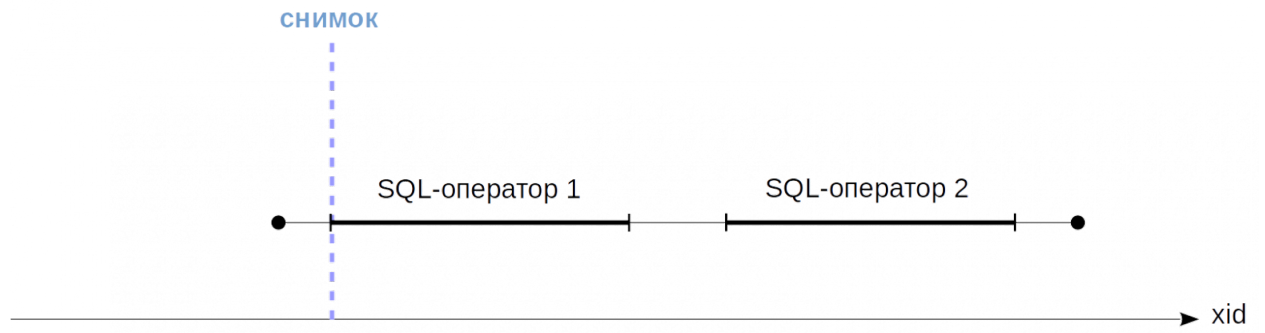
Уровень изоляции транзакции нельзя изменить после выполнения первого запроса на выборку или изменение данных (SELECT, INSERT, DELETE, UPDATE, FETCH или COPY) в текущей транзакции.

Изоляция в PostgreSQL строится на основе снимков данных (snapshot): каждая транзакция работает со своим снимком данных, который «содержит» данные, которые были зафиксированы до момента создания снимка, и не «содержит» еще не зафиксированные на этот момент данные.

На уровне изоляции Read Committed снимок создается в начале каждого оператора транзакции. Такой снимок активен, пока выполняется оператор. На рисунке момент создания снимка (который, как мы помним, определяется номером транзакции) показан синим цветом.



На уровнях Repeatable Read и Serializable снимок создается один раз в начале первого оператора транзакции. Такой снимок остается активным до самого конца транзакции.



24. VACUUM

VACUUM - занимается высвобождением пространства на диске, занимаемое “мертвыми” записями.

Для удаления таких записей используется такая команда:

VACUUM table[, table2, ...];

(Без параметра команда VACUUM обрабатывает все таблицы в текущей базе данных, которые может очистить текущий пользователь)

VACUUM:

- **По умолчанию;** - если записи хранятся на нескольких страницах, то все “мертвые” записи в обеих страницах будут удалены. В итоге мы получим такое же количество страниц как и было прежде, только с пустыми пространствами, где раньше хранились наши “мертвые записи”. (делаем пространство для повторного использования, место не освобождается на диске при этом)
- **VACUUM FULL;** - переписывает всё содержимое таблицы в новый файл на диске, не содержащий ничего лишнего, что позволяет вернуть неиспользованное пространство операционной системе. Эта форма работает намного медленнее и запрашивает исключительную блокировку для каждой обрабатываемой таблицы.
- **VACUUM FREEZE;** - помечает, что данную таблицу не нужно очищать до следующего обновления таблицы.

VACUUM нельзя выполнять внутри блока транзакции!

Команды VACUUM в PostgreSQL должны обрабатывать каждую таблицу по следующим причинам:

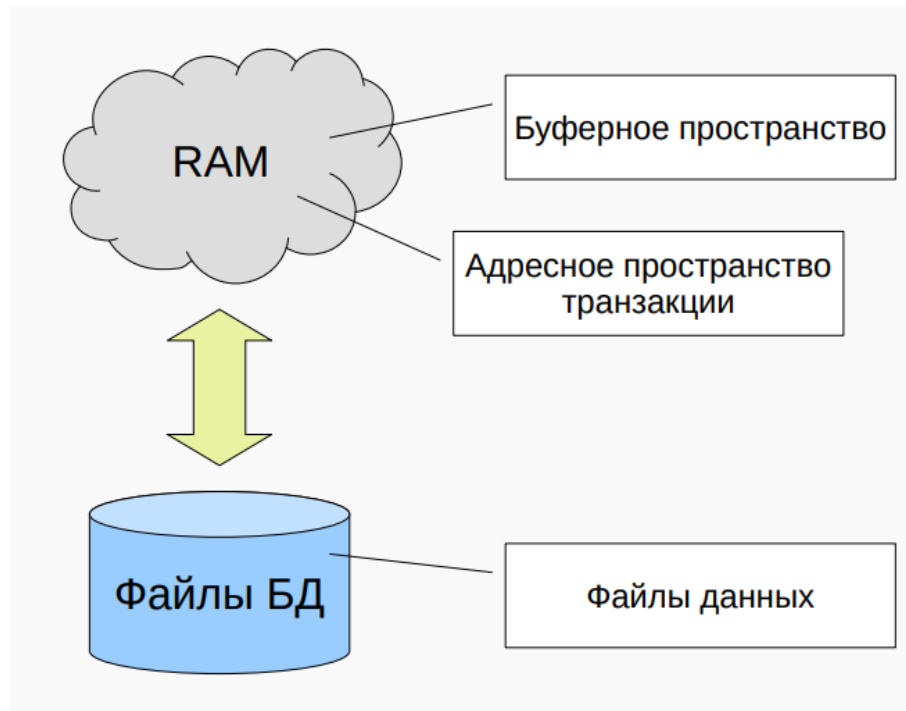
- Для высвобождения или повторного использования дискового пространства, занятого изменёнными или удалёнными строками.
- Для обновления статистики по данным, используемой планировщиком запросов PostgreSQL.
- Для обновления карты видимости, которая ускоряет сканирование только индекса.
- Для предотвращения потери очень старых данных из-за закливания идентификаторов транзакций или мультитранзакций.

VACUUM можно настроить через файл **postgresql.conf**

- **autovacuum = true** (по умолчанию)
- **autovacuum_vacuum_threshold** - Задаёт минимальное число изменённых или удалённых кортежей, при котором будет выполняться VACUUM для отдельно взятой таблицы. Значение по умолчанию — 50 кортежей.
- **autovacuum_vacuum_scale_factor** - Задаёт процент от размера таблицы, который будет добавляться к autovacuum_vacuum_threshold при выборе порога срабатывания команды VACUUM. Значение по умолчанию — 0.2 (20% от размера таблицы)

- **autovacuum_vacuum_cost_limit** - Задаёт предел стоимости, который будет учитываться при автоматических операциях VACUUM. При значении -1 (по умолчанию) применяется обычное значение vacuum_cost_limit.

25. Восстановление данных. Базовые понятия. Организация. Контрольные точки.



Восстановление базы данных — функция СУБД, которая в случае логических и физических сбоев приводит базу данных в актуальное и консистентное состояние.

Буферное пространство – буферы, в которых хранятся копии данных.

Адресное пространство транзакции – локальное пространство, в котором выполняются различные операции с данными данной транзакции.

Для предотвращения ситуаций, связанных с появлением несогласованных данных можно использовать — **журнал (лог)**.

СУБД сохраняет информацию об изменениях и фиксациях данных в журнале:

- первоначально запись сохраняется в соответствующем буфере (буфере журнала);
- буфер журнала синхронизируется с файлами журнала на диске (WriteLog);

В случае логического отказа или сигнала отката одной транзакции журнал изменений сканируется в обратном направлении, и все записи отменяемой транзакции извлекаются из журнала вплоть до отметки начала транзакции. Согласно извлеченной информации выполняются действия, отменяющие действия транзакции. Этот процесс называется откат (rollback).

В случае физического отказа, если ни журнал изменений, ни сама база данных не повреждены, то выполняется процесс прогонки (rollforward). Журнал сканируется в прямом направлении, начиная от предыдущей контрольной точки. Все записи извлекаются из журнала вплоть до конца журнала. Извлеченная из журнала информация вносится в блоки данных внешней памяти, у которых отметка номера изменений меньше, чем записанная в журнале. Если в процессе прогонки снова возникает сбой, то сканирование журнала вновь начнётся сначала, но восстановление фактически продолжится с той точки, где оно прервалось.

Операции для синхронизации различных пространств памяти:

- $M \leftarrow D(Obj)$ — загрузить страницу, содержащую **Obj**, с диска в буферное пространство.
- $D \leftarrow M(Obj)$ — сохранить страницу, содержащую **Obj**, на диск из буфера ОП.
- $T(v, Tx) \leftarrow M(Obj)$ — копировать **Obj** в переменную **v** транзакции **Tx**:
 - Если **Obj** не в **M**, то сначала $M \leftarrow D(Obj)$.
- $M(Obj) \leftarrow T(v, Tx)$ — копировать значение **v** из транзакции **Tx** в **Obj**:
 - Если **Obj** не в **M**, то сначала $M \leftarrow D(Obj)$.
- **WriteLog** — сохранить лог на диск из соответствующего буфера.

Контрольные точки (checkpoint) – точка синхронизации, во время которой происходит синхронизация данных из буферов с соответствующими файлами на диске. Для восстановления данных нужна какая-то стартовая точка, чтобы не анализировать транзакции с самого начала работы с БД.

Задача: переименовать аудитории

Room

Room_ID	Address	Room_Num	Size
1	Kronverksky, 49	374	20
2	Kronverksky, 49	375	25

```
BEGIN;
UPDATE ROOM SET Room_Num = 1331
WHERE Room_Num = 374;
UPDATE ROOM SET Room_Num = 1330
WHERE Room_Num = 375;
COMMIT;
```

BEGIN;

```
UPDATE ROOM
SET Room_Num = 1331
WHERE Room_Num = 374;
```

$T(v_{Room}, Tx) \leftarrow M(Room_{374});$
 $V_{Room}(Room_Num) = 1331;$
 $M(Room_{374}) \leftarrow T(v_{Room}, Tx);$

```
UPDATE ROOM
SET Room_Num = 1330
WHERE Room_Num = 375;
```

$T(v_{Room}, Tx) \leftarrow M(Room_{375});$
 $V_{Room}(Room_Num) = 1330;$
 $M(Room_{375}) \leftarrow T(v_{Room}, Tx);$

$D \leftarrow M(Room_{374}), M(Room_{375})$

COMMIT;

*Выполнение операции
UPDATE сильно упрощено

Операция	Пам. тр.	Буферное пространство	Диск
$T(v_{R_i}, Tx) \leftarrow M(R_{374})$	374	$R_{374}=374, R_{375}= -$	$R_{374}=374, R_{375}=375$
$V_{R_i}(Room_Num) = 1331;$	1331	$R_{374}=374, R_{375}= -$	$R_{374}=374, R_{375}=375$
$M(R_{374}) \leftarrow T(v_{R_i}, Tx);$	1331	$R_{374}=1331, R_{375}= -$	$R_{374}=374, R_{375}=375$
$T(v_{R_i}, Tx) \leftarrow M(R_{375});$	375	$R_{374}=1331, R_{375}=375$	$R_{374}=374, R_{375}=375$
$V_{R_i}(Room_Num) = 1330;$	1330	$R_{374}=1331, R_{375}=375$	$R_{374}=374, R_{375}=375$
$M(R_{375}) \leftarrow T(v_{R_i}, Tx);$	1330	$R_{374}=1331, R_{375}=1330$	$R_{374}=374, R_{375}=375$
$D \leftarrow M(R_{374})$	1330	$R_{374}=1331, R_{375}=1330$	$R_{374}=1331, R_{375}=375$
$D \leftarrow M(R_{375})$	1330	$R_{374}=1331, R_{375}=1330$	$R_{374}=1331, R_{375}=1330$

26. (Adv.) UNDO-журнал.

Логирование:

Для предотвращения ситуаций, связанных с появлением несогласованных данных можно использовать — журнал (лог).

- СУБД сохраняет информацию об изменениях и фиксациях данных в журнале:
 - первоначально запись сохраняется в соответствующем буфере (буфере журнала);
 - буфер журнала синхронизируется с файлами журнала на диске (WriteLog);

Журнал отмены:

Один из вариантов ведения журнала — журнал отмены, UNDO LOG.

Возможные записи в журнале отмены:

- BEGIN Tx – начало транзакции;
- Tx: R, old_val – изменение: транзакция Tx переписывает значение old_val в R;
- COMMIT Tx – успешное завершение транзакции;
- ABORT Tx – преждевременное завершение транз-ии;
- START CHK_n: T_n, ..., T_m – начало создания контр. точки
- END CHK_n – завершение создания контр. точки

Правила записи в UNDO LOG:

- Запись изменения данных (Tx: R, old_val) записывается в журнал (на диск) **ДО сохранения обновленного значения на диске.**
- COMMIT Tx** должен быть добавлен в журнал (на диск) **ПОСЛЕ обновления всех файлов данных**, связанных с изменениями данной транзакции (после синхронизации буферного пространства данных с файлами данных).

Пример:

Операция	Диск	Журнал
		BEGIN T1;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₄ , 374
$T(v_R, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₅ , 375
WriteLog		-- записи на диске
$D \leftarrow M(R_{374})$	$R_{374}=1331, R_{375}=375$	
$D \leftarrow M(R_{375})$	$R_{374}=1331, R_{375}=1330$	
		COMMIT T1
WriteLog		-- COMMIT на диске

P.S. столбец «Журнал» – буфер журнала в памяти, запись в журнал на диске происходит после операции WriteLog.

Использование журнала отмены при восстановлении данных:

- СУБД сканирует журнал от новых записей до старых.
- Отдельно фиксируются транзакции:
 - Группа 1: для которых есть запись COMMIT Tx;
 - Группа 2: транзакции для которых есть запись ABORT Tx, незавершенные транзакции.
- Когда в журнале встречается запись изменения (Tx: R, old_val), анализируется Tx.
 - Если Tx:
 - из группы 1 действия не предпринимаются.
 - из группы 2: для R восстанавливается old_val (происходит отмена изменений).
- Во время восстановления СУБД записывает ABORT Tx для каждой незавершенной транзакции из группы 2 (после отмены её изменений).
- Запускается операция WriteLog, чтобы записи вида ABORT Tx оказались в журнале.

Операция	Диск	Журнал
		BEGIN T1;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₄ , 374
$T(v_R, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₅ , 375
WriteLog		-- записи на диске
D ← M(R ₃₇₄)	$R_{374}=1331, R_{375}=375$	

Произизошел сбой!

Операция	Диск	Журнал
		BEGIN T1;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1331;$	$R_{374}=374, R_{375}=375$	Возвращаем первоначальное значение
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₄ , 374
$T(v_R, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₅ , 375
WriteLog		-- записи на диске
D ← M(R ₃₇₄)	$R_{374}=1331, R_{375}=375$	

Возвращаем первоначальное значение

Контрольные точки:

- Для восстановления данных нужна какая-то стартовая точка, чтобы не анализировать транзакции с самого начала работы с БД.
- Контрольная точка (checkpoint) — точка синхронизации, во время которой происходит синхронизация данных из буферов с соответствующими файлами на диске.

Создание контрольной точки не одномоментно:

- В журнал добавляется START CHKn: Tn, ..., Tm, фиксируется на диске через WriteLog
 - Tn..Tm – ещё не завершённые (на момент начала создания контрольной точки) транзакции
- Ожидание завершения транзакций, которые работали в момент создания

- контрольной точки (T_n, \dots, T_m). В это время могут начинаться другие транзакции.
3. В журнал добавляется END CHK n , фиксируется на диске через WriteLog
 4. Записи UNDO LOG перед START CHK n , у которой есть END CHK n , могут быть удалены.

Действия при восстановлении данных UNDO LOG с учетом контрольных точек:

- СУБД сканирует журнал и встречает запись END CHK n : T_n, \dots, T_m (а не START CHK n):
 - Значит установка контрольной точки n завершена;
 - Сканирование журнала до записи START CHK n , так как незавершенные транзакции могут быть только после записи START CHK n :
 - среди транзакций, которые начали работу после начала установки контрольной точки
 - Остальные действия как раньше.
- СУБД сканирует журнал и встречает запись START CHK n : T_n, \dots, T_m (а не END CHK n):
 - Значит установка контрольной точки n не завершена (проблема во время установки);
 - Ищем END CHK $n-1$, дальнейшие действия, как в пред. пункте, только для CHK $n-1$ (так как транзакции T_n, \dots, T_m на момент старта предыдущей контрольной точки гарантировано ещё не были начаты).

Недостатки UNDO LOG:

- Нельзя завершить транзакцию (зафиксировать COMMIT в журнале) до записи изменений данных в файлы данных.
- Много операций ввода-вывода с диском, так как надо синхронизировать данные для каждой транзакции

27. (Adv.) REDO-журнал.

Другой способ ведения журнала — журнал повторений, REDO LOG (повторяются завершённые транзакции).

Возможные записи в журнале повторений:

- BEGIN Tx – начало транзакции;
- Tx: R, new_val – изменение: транзакция Tx изменяет значение на new_val в R;
- COMMIT Tx – успешное завершение транзакции;
- ABORT Tx – преждевременное завершение транзакции;
- START CHKn: Tn, ..., Tm – начало создания контр. точки
- END CHKn – завершение создания контр. точки

Правило записи в REDO LOG (правило Write Ahead Logging):

- Запись изменения данных (Tx: R, new_val) записывается в журнал (на диск) ДО сохранения обновленного значения на диске.
- COMMIT Tx должен быть добавлен в журнал (на диск) ДО обновления любого из файлов данных, связанных с изменениями данной транзакции (до синхронизации буферного пространства данных с файлами данных).
- То есть в WAL (на диске) вносятся записи изменения, которые произошли в Tx, и запись COMMIT Tx, и только после этого — идет синхронизация буферов с диском.

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R_1}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₄ , 1331
$T(v_{R_1}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₅ , 1330
		COMMIT T1
WriteLog		
$D \leftarrow M(R_{374}), D \leftarrow M(R_{375})$	$R_{374}=1331, R_{375}=1330$	

Лог уже на диске, а буферы не синхронизированы с файлами данных

Использование журнала повторений при восстановлении данных:

- СУБД сканирует журнал от старых записей до новых. Отдельно фиксируются транзакции:
 - Группа 1: для которых есть запись COMMIT Tx – не известно записаны изменённые данные на диск или нет, а если записаны, полностью ли;
 - Группа 2: транзакции для которых есть запись ABORT Tx, незавершённые транзакции – т. к. не завершены, по любому изменения не сохранены на диск.

- Когда в журнале встречается запись изменения (Tx: R, new_val), анализируется Tx. Если Tx:
 - из группы 1: для R перезаписывается new_val (происходит «повтор», перезапись изменений).
 - из группы 2:
 - СУБД записывает ABORT Tx для каждой незавершенной транзакции из группы 2 (у которой не было ABORT).
 - Запускается операция WriteLog, чтобы записи вида ABORT Tx оказались в журнале.

Пример:

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R_1}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{374} , 1331
$T(v_{R_1}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{375} , 1330

Произошел сбой!

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R_1}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{374} , 1331
$T(v_{R_1}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{375} , 1330

Нет COMMIT: транзакция не завершена. ABORT.

«WAL логирование с REDO связанная штука» – **Николаев**

Записи о транзакции есть только в WAL-буфере, в журнале на диске о транзакции записей нет – журнал даже не знает о транзакции, поэтому ничего делать не нужно.

Возможно следующее: другой процесс инициализировал синхронизацию WAL буфера с WAL-файлами на диске – тогда 3 записи о транзакции из буфера попадут на диск – транзакция не завершена (нет COMMIT) – делаем ABORT.

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R_1}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{374} , 1331
$T(v_{R_1}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{375} , 1330
		COMMIT T1
WriteLog		

Произошел сбой!

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R_1}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{374} , 1331
$T(v_{R_1}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_{R_1}(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R_1}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{375} , 1330
		COMMIT T1
WriteLog		

Транзакция COMMIT и зафиксирована → повторяем сначала

Несмотря на то, что файлы данных могут быть не синхронизированы с shared buffers, записи о транзакции после WriteLog из WAL-буфера перенеслись в WAL-файлы – а значит, в журнале на диске есть завершённая транзакция с COMMIT – повторяем её с начала.

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R^*}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R^*}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{374} , 1331
$T(v_{R^*}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R^*}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{375} , 1330
		COMMIT T1

Произошел сбой!

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_{R^*}, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_{R^*}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{374} , 1331
$T(v_{R^*}, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_{R^*}, Tx);$	$R_{374}=374, R_{375}=375$	T1: R_{375} , 1330
		COMMIT T1

Зависит от того попал ли в WAL на диске COMMIT. WriteLog мог выполнить другой процесс

Записи о транзакции есть только в WAL-буфере, в журнале на диске о транзакции записей нет – журнал даже не знает о транзакции, поэтому ничего делать не нужно.

Возможно следующее: другой процесс инициировал синхронизацию WALбуфера с WAL-файлами на диске – тогда, если COMMIT на диске – повторяем (пример 2), иначе – делаем ABORT (пример 1).

Контрольные точки (REDO):

Создание контрольной точки не одномоментно:

1. В журнал добавляется START CHKn: T_n, \dots, T_m , фиксируется на диске через WriteLog.
2. Сохранение всех изменений («грязных страниц») на диск, измененных завершёнными транзакциями на момент START CHKn
3. В журнал добавляется END CHKn, фиксируется на диске через WriteLog

Действия при восстановлении данных REDO LOG с учетом контрольных точек:

- СУБД сканирует журнал в поисках данных о последней контрольной точке, и встречает запись END CHKn (а не START CHKn: T_n, \dots, T_m):
 - Остальные действия как раньше, но только для транзакций (T_n, \dots, T_m) или тех, которые были начаты после записи START CHKn: T_n, \dots, T_m
- СУБД сканирует журнал в поисках данных о последней контрольной точке и встречает запись START CHKn: T_n, \dots, T_m (а не END CHKn):
 - Значит установка контрольной точки n не завершена (проблема во время установки);
 - Ищем END CHKn-1, дальнейшие действия, как в пред. пункте, только для CHKn-1.

Недостатки REDO LOG:

Нельзя «сбросить» данные из буферов на диск до завершения транзакции → требуется много памяти под буферы.

28. (Adv.) UNDO/REDO-журнал.

Еще один способ — журнал отмены/повторений, UNDO/REDO LOG.

Возможные записи в журнале повторений:

- BEGIN Tx – начало транзакции;
- Tx: R, old_val, new_val – изменение: транзакция Tx изменяет значение с old_val на new_val в R;
- COMMIT Tx – успешное завершение транзакции;
- ABORT Tx – преждевременное завершение транзакции;
- START CHKn: Tn, ..., Tm – начало создания контр. точки
- END CHKn – завершение создания контр. точки

Правило записи в UNDO/REDO LOG:

- Запись изменения данных (Tx: R, old_val, new_val) записывается в журнал (на диск) ДО сохранения обновленного значения на диске.
- COMMIT Tx может быть добавлен в журнал (на диск) как до, так и после обновления любого из файлов данных, связанных с изменениями данной транзакции.

Использование UNDO/REDO для восстановления данных:

- повторить операции для завершенных транзакций (от ранних к поздним);
- отменить операции для незавершенных транзакций (от поздних к ранним);
- запись COMMIT должна быть сразу синхронизирована с журналом;

Пример:

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₄ , 1331
$T(v_R, Tx) \leftarrow M(R_{375});$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1330;$	$R_{374}=374, R_{375}=375$	
$M(R_{375}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	T1: R ₃₇₅ , 1330
WriteLog		
$D \leftarrow M(R_{374}), D \leftarrow M(R_{375})$	$R_{374}=1331, R_{375}=1330$	
		COMMIT T1
WriteLog		

Изменения значений в R фиксируются до записи данных на диск (первый WriteLog).
Порядок COMMIT'а и операций записи данных на диск – произвольный.

Контрольные точки (UNDO/REDO):

Создание контрольной точки не одномоментно:

1. В журнал добавляется START CHK_n: T_n, ..., T_m, фиксируется на диске через WriteLog
2. Сохранение всех изменений («грязных страниц») на диск.
 - а. Если транзакция может прервать работу (ROLLBACK), ее изменения не должны фиксироваться в буферах данных – должны использоваться специальные хранилища, потому что мы сохраняем на диск изменения всех транзакций (работающих и завершённых на момент START CHK_n), а возможный ROLLBACK работающей транзакции может всё попортить.
3. В журнал добавляется END CHK_n, фиксируется на диске через WriteLog.

Действия при восстановлении данных UNDO/REDO LOG с учетом контрольных точек:

Состоит из шагов для UNDO и REDO.

- Благодаря тому, что при установке контрольной точки происходит сохранение всех изменений («грязных страниц») на диск, REDO нужно возвращаться только к последней START CHK_n, для которой есть END CHK_n:
 - если на момент START CHK_n были незавершенные транзакции, их изменения зафиксированы при создании контрольной точки:
 - Если транзакция в итоге завершиться успешно — REDO (с момента начала этой контр. точки);
 - Если неуспешно — UNDO — как обычно.

29. (Adv.) Восстановление данных. Подходы (steal/no-steal, force/no-force). ARIES.

no-steal — подход, при котором страница из буфера данных не может быть записана на диск до завершения (COMMIT) транзакции. В ходе которой она была изменена. (сначала COMMIT – потом синхронизация данных буфера с диском)

steal — если позволено записывать в файлы данных данные еще незавершенных транзакций (для оптимизации запросов к диску или если буферная память заканчивается).

force — если требуется, чтобы все страницы, измененные транзакцией, записывались сразу перед COMMIT. (сначала синхронизация данных буфера с диском – потом COMMIT)

no-force — изменения, осуществленные завершенными транзакциями могут быть не зафиксированы на диске.

ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) — алгоритм для восстановления данных. Базируется на:

- WAL
- steal
- no-force

ARIES происходит в 3 этапа (3 фазы):

- Анализ — поиск «грязных» страниц и незавершенных транзакций;
- REDO — повторение всех действий;
- UNDO — откат действий незавершенных транзакций;

Для работы требуется LSN, таблица транзакций, таблица «грязных страниц».

30. Восстановление данных в PostgreSQL. WAL. LSN.

WAL (Write Ahead Log) — хранит информацию об изменениях данных в БД. Эта информация используется для воссоздания актуального состояния данных в случае восстановления базы данных (например, после сбоя). Настраивается через параметр `wal_buffers` — размер wal-буфера.

WAL — общий для всего кластера.

WAL-записи фиксируют различные изменения состояния данных в БД (INSERT, UPDATE, DELETE, COMMIT)

В PostgreSQL:

- Поддерживается REDO-лог.
- Запись об изменении попадает в журнал (на диск) раньше, чем данные на диск и clog.
- PGDATA/pg_wal:
 - журнал разбит на сегменты;
 - `wal-segsize` — размер сегмента;
 - `wal-segsize` можно изменить только при инициализации кластера PostgreSQL;

Особенности работы журнала:

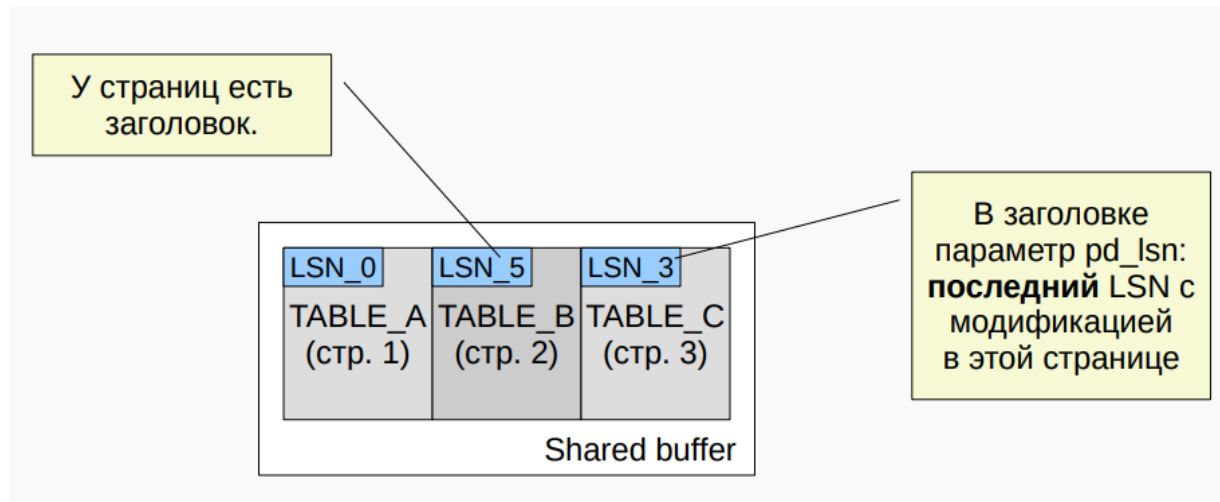
- В журнал не попадают:
 - действия над временными таблицами;
 - действия над UNLOGGED-таблицами;
 - до версии 10: хэш-индексы.
- UNLOGGED TABLE: изменения для данных в такой таблице не хранятся в WAL:
 - `CREATE UNLOGGED TABLE Students ...`
 - **UNLOGGED** можно использовать, только когда данные в таблице не важны (не страшны потери), и при этом требуется минимизировать число обменов IO

Основные составляющие записи в журнале:

- Заголовок:
 - размер всей записи;
 - номер транзакции;
 - размер данных;
 - ссылка на предыдущую запись;
 - контрольная сумма;
- Данные об изменении — могут быть по-разному представлены.

У каждой WAL-записи есть **LSN (Log Sequence Number)** – уникальный идентификатор WAL-записи. Он используется для нахождения позиции записи в WAL (LSN - смещение до записи от начала журнала).

LSN последней WAL-записи, относящейся к некоторой странице хранится в заголовке страницы. Для определения того, актуальна или нет некоторая страница (например, на момент восстановления) её LSN сравнивается с LSN в журнале.



Восстановление:

- 1) **WAL (REDO-лог)** — журнал последовательно читается и перевыполняются операции:
 - а) для определения записей, которые нужно заново «выполнить» используется LSN.
- 2) Чтобы сократить число записей для просмотра: используются контрольные точки:
 - а) процесс checkpointer;
 - б) начало последней контрольной точки — REDO point

wal_level - Параметр позволяет задать уровень детализации операций в WAL:

- **minimal** — только для восстановления после сбоя или immediate shutdown;
- **replica** — minimal + поддержка архивирования WAL и физической репликации;
- **logical** — replica + поддержка логической репликации (логического декодирования WAL);

31. Резервное копирование. Базовые понятия. Виды.

Стратегия резервного копирования бывает:

- полная — охватывает всю базу данных (или весь кластер);
- частичная — охватывает только часть базы данных (только набор объектов, схема, данные и т.д.)

Обычно нам недостаточно одной резервной копии, значит нужно делать их периодически. В таком случае копии могут по-разному соотноситься друг с другом. Резервная копия может содержать:

- все страницы в пределах выбранных файлов – часть копии может совпадать с предыдущей, и тогда мы будем напрасно тратить память
- Инкрементная копия – сначала создается полная копия, а затем инкременты – наборы страниц с изменениями относительно этой полной копии. При этом с определенной периодичностью делается полная копия. При этом инкременты могут быть:
 - Кумулятивными – каждый инкремент соотносится относительно полной копии
 - Дифференциальными – каждый инкремент отчитывается относительно предыдущего инкремента

Виды резервного копирования:

- «Холодное» — создание резервной копии, когда экземпляр сервера БД остановлен. Это простой способ, но хочется чтобы база всегда была доступна.
- «Горячее» — создание резервной копии во время работы экземпляра сервера БД.

Другая классификация:

- Логическое – копия объектов БД и структур данных создается в виде команд/предложений языка SQL, сами файлы не копируются
- Физическое – при создании резервной копии происходит копирование файлов исходного кластера

32. Логическое резервное копирование. `pg_dump`, `pg_dumpall`, `pg_restore`.

Копия объектов БД и структур данных создается в виде команд/предложений языка SQL, сами файлы не копируются.

Утилиты в PostgreSQL для осуществления логического резервного копирования/восстановления:

- `pg_dump`
- `pg_dumpall`
- `pg_restore`

`pg_dump` служит для создания резервной копии БД или ее части. В общем случае создается SQL-файл с командами, которые позволяют воссоздать логическое состояние БД на момент создания бэкапа.

Использование: `pg_dump db_name > db_dump`
`pg_dump -U postgres1 -f db_dump db_name`

Форматы:

- plain text, текстовый формат — в файле ~SQL предложения:
 - можно восстановить, выполнив в `psql`;
 - `pg_dump -f mydbdump mydb`
- custom формат:
 - возможно сжатие (если есть `zlib`);
 - бинарный файл;
 - `pg_dump -Fc -f mydbdump mydb`
- в виде директории — один файл для каждой ~таблицы;
 - `pg_dump -Fd Student -f mydbdump mydb`
- tar — архив PK, в архиве содержимое, как для формата в виде директории;
 - `pg_dump -Ft Student -f mydbdump mydb`

Формат PK в виде директории

Для отдельных объектов БД создаются отдельные файлы:

- возможно сжатие файлов в директории;
- В директории есть файл `toc.dat`:
 - индекс для `pg_restore`;
 - для поиска файлов внутри директории;

Текстовый формат

Используется по умолчанию.

- Содержимое — предложения для восстановления логического состояния БД.
- По умолчанию для добавления данных используется PostgreSQL команда `COPY`.
- Для генерации `INSERT`:
`pg_dump -U postgres1 --inserts -f mydbdump mydb`
`pg_dump -U postgres1 --column-inserts -f mydbdump mydb`

Особенности использования pg_dump

По умолчанию не создаются предложения для создания БД.

- Чтобы восстановить данные, нужно подключиться к существующей БД.
- Для добавления создания БД (CREATE DATABASE) можно использовать опцию --create (-C):
- при использовании опции в бэкап добавляются предложения для подключения к созданной БД:
 - pg_dump -U postgres1 --create -f mydbdump mydb

Получение скрипта для восстановления части БД

Можно задать создание скрипта для восстановления только определенных таблиц, структуры БД, содержимого БД:

- pg_dump -t Student -f mydbdump mydb – только Student
- pg_dump -T Student -f mydbdump mydb – все, кроме Student
- pg_dump -a -f mydbdump mydb -- только данные
- pg_dump -s -f mydbdump mydb -- только схема (DDL)

Восстановление БД

- Использование: psql mydb < mydbdump
- Используемые в бэкапе пользователи, роли должны существовать/быть воссозданы до запуска скрипта.

Ошибки при восстановлении БД

- Чтобы остановить скрипт после возникновения первой ошибки (невыполненного предложения): psql --set ON_ERROR_STOP=on mydb < mydbdump (Результаты прошедших команд останутся в случае возникновения ошибки).
- Можно указать, чтобы восстановление выполнялось в рамках одной транзакции: psql --single-transaction mydb < mydbdump

pg_restore

- Утилита для восстановления данных на основе скриптов, созданных pg_dump: pg_restore -d mydb mydbdump
- Можно посмотреть список команд, которые будут исполнены при восстановлении БД: pg_restore mydbdump -f test.sql
- Если используется директория — можно включить свой toc-файл и сделать частичное восстановление.

pg_dumpall

Утилита для создания скриптов восстановления всех БД в кластере:

pg_dumpall -U postgres1 -f mydbdump

- Создает SQL-скрипт.
- Кроме БД сохраняются глобальные объекты (пользователи, роли, табличные пространства).
- Восстановление БД: psql -f dumpfile postgres

Осуществление резервного копирования

- Возможен параллельный режим: `pg_dump`, опция `-j`:
 - можно указать число параллельных процессов.
- Есть готовые скрипты-шаблоны для организации резервного копирования

33. Физическое резервное копирование. Способы организации в PostgreSQL.

- При создании резервной копии происходит копирование файлов исходного кластера.
- Утилиты в PostgreSQL для осуществления физического резервного копирования/восстановления:
 - pg_basebackup
 - функции pg_start_backup, pg_stop_backup

Резервное копирование на уровне файловой системы

- Заключается в полном копировании директории кластера БД средствами файловой системы.
- Обычно производится в «холодном режиме» - сервер БД должен быть остановлен:
 - «горячий режим» возможен если файловая система поддерживает возможность получения единовременного «снимка» данных.
- Может быть произведен только для всех объектов БД

Можно осуществить через pg_basebackup:

pg_basebackup -D /path/backup -T /old/ts=\$(pwd)/new/ts

- backup_history – файл в pg_wal (***.backup) — имя содержит название первого WAL-сегмента для осуществления восстановления.
- С помощью pg_basebackup можно осуществить создание обособленной копии (без осуществления непрерывного копирования).

Низкоуровневый вызов системных функций pg_*_backup:

- SELECT pg_start_backup('my_label');
 - создается backup_my_label файл — содержит различную информацию о резервной копии (время начала, метку, ...);
 - создается tablespace_map;
 - создается контрольная точка;
- пользователем производится копирование файлов кластера БД;
- SELECT pg_stop_backup();

34. PostgreSQL: непрерывное архивирование.

Базируется на существовании WAL (PGDATA/pg_wal):

- состояние БД может быть восстановлено путем повтора зафиксированных в логе операций;
- конфигурационные файлы не восстанавливаются через WAL (postgresql.conf, ...);
- Резервная копия = полная копия + WAL-файлы

Физическое РК + WAL

- БД необязательно должна быть в полностью согласованном состоянии:
 - Для приведения состояния к согласованному может использоваться WAL;
 - С точки зрения работы WAL — механизмы аналогичны тем, которые используются при восстановлении после сбоев.
- Можно восстановить данные в одно из временных состояний между снятием бэкапа и последним WAL.
- Можно восстановить базу данных целиком (а не отдельную часть).

Этапы для реализации непрерывного архивирования

1. Организация архивирования WAL
2. Создание базовой резервной копии

Этап 1: архивирование WAL

- На диске WAL разделен на сегменты по 16 Мб (по умолчанию).
- Необходимо осуществить копирование добавляемых сегментов на внешнее хранилище.
- Для включения возможности резервного копирования:
 - archive_mode = on
 - wal_level = replica # минимум
- PostgreSQL позволяет задать команду для создания копий WAL по мере их создания (postgresql.conf):
 - archive_command = 'some command'

Команда для архивирования

Задается в postgresql.conf:

archive_command = 'cp %p /path2/%f' # Unix

- %p — заменяется на полное имя файла для архив-я.
- %f — заменяется на имя файла.

Назначение:

- копирование сегментов WAL в path2.
- команда должна возвращать 0 только в случае успешного завершения.

archive_command

- Значение можно изменить, перезагрузив значения конфигурационного файла.
- Частоту создания новых сегментов можно регулировать через `archive_timeout`: если он небольшой, то будет создаваться много незаполненных сегментов и тратиться лишнее место.

Этап 2: создание базовой резервной копии - pg_basebackup

- Можно осуществить через `pg_basebackup`:
`pg_basebackup -D /path/backup -T /old/ts=$(pwd)/new/ts`
- `backup_history`-файл в `pg_wal` (`***.backup`) — имя содержит название первого WAL-сегмента для осуществления восстановления.
- С помощью `pg_basebackup` можно осуществить создание обособленной копии (без осуществления непрерывного копирования).

Этап 2: создание базовой резервной копии — через pg_*_backup

функции `pg_start_backup`, `pg_stop_backup`:

- `SELECT pg_start_backup('my_label');`
 - создается `backup_my_label` файл — содержит различную информацию о резервной копии (время начала, метку, ...);
 - создается `tablespace_map`;
 - создается контрольная точка;
- пользователем производится копирование файлов кластера БД;
- `SELECT pg_stop_backup();`

Восстановление данных на основе бэкапа с непрерывным архивированием

1. Остановить сервер БД
2. Если возможно, сделать копию — хотя бы `pg_wal` (на случай, если есть не заархивированные WAL)
3. Перед шагом: проверить наличие бэкапа. Если бэкап есть — удалить содержимое PGDATA и, если есть, внешних директорий (с табличными пространствами и т. д).
4. Скопировать в PGDATA и внешние директории данные ранее полученного бэкапа.
5. Очистить директорию PGDATA/pg_wal
6. Если есть не заархивированные WAL (с шага 2) — поместить их в PGDATA/pg_wal.
7. Установить настройки для восстановления в `postgresql.conf` (`restore_command`, должна возвращать ненулевой код в случае неудачи) + запретить внешние подключения (для пользователей) в `pg_hba.conf`
8. Создать файл `recovery.signal` — говорит серверу, что надо стартовать в режиме восстановления (PGDATA/data)
9. Запустить сервер → он перейдет в режим восстановления. По завершении `recovery.signal` будет удален
10. Разрешить подключения для пользователей в `pg_hba.conf`

restore_command

Задаёт логику получения архивированных WAL-сегментов:

`restore_command = 'cp /path/%f %p'`

- `%f` - имя файла (WAL-сегмента);
- `%p` - путь, куда копировать файлы (WAL-сегменты).

Сегменты, которые не были найдены в архиве, будут
искаться в `pg_wal`:

- приоритет тем сегментам, которые в архиве.

35. Репликация данных в PostgreSQL. Базовые понятия.

Нам нужен **высокий уровень доступности** — (high availability) способность системы работать без сбоев (непрерывно) в течение определенного периода времени.

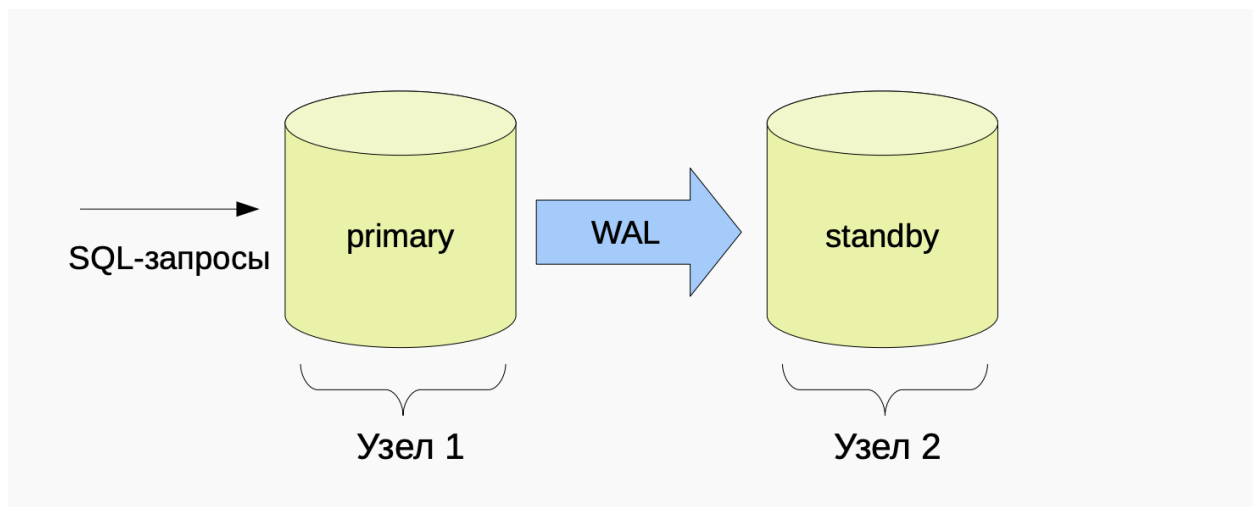
Как обеспечить высокий уровень доступности:

- обеспечить избыточность важных частей системы;
 - репликация данных.
- Failover, switchover.

Репликация данных — средство обеспечения избыточности. Заключается в сохранении и поддержании нескольких копий данных.

Репликация в СУБД:

- Узлы — отдельные серверы БД, обеспечивающие работу. Совокупность узлов — кластер.
- У каждого узла — своя БД (оригинал или копия, реплика). Копии данных синхронизируются в реальном времени.
- Один из серверов БД — главный (master, primary). Режим — чтение/запись. Если несколько master-узлов — multimaster-репликация.
- Остальные узлы — зависимые (slave, standby). Если режим — только чтение — hot standby.



Репликация в PostgreSQL

В PostgreSQL репликация может быть:

- Физическая — пересылаются файлы (WAL). standby — содержит копию primary:
 - асинхронная — данные для синхронизации состояний серверов БД пересылаются без подтверждения получения;
 - синхронная — пересылки данных идут с подтверждением.
- Логическая — пересылаются декодированные из WAL операции.

36. Виды репликации. Особенности.

Можно выделить три основных вида репликации:

- Блочная репликация на уровне системы хранения данных;
- Физическая репликация на уровне СУБД;
- Логическая репликация на уровне СУБД.

Блочная репликация

Каждая операция записи выполняется не только на основном диске, но и на резервном. Таким образом тому на одном массиве соответствует зеркальный том на другом массиве, с точностью до байта повторяющий основной том

Плюсы:

- Универсальность
- Легко настраивается
- Надежность, отказоустойчивость

Минусы:

- Если что-то произошло на основном томе, нужно сразу отключить репликацию и восстанавливать, запуская ее в обратном направлении. Эти действия могут быть автоматизированы, но остается время на переключение

Физическая репликация

Так как журналы redo log и write-ahead-log содержат все изменения, то можно просто выполнить их же в другой базе - реплике.

В Постгресе это называется - Log shipping/**Streaming replication(Потоковая)**

Плюсы:

- Реплике нужно всего около 10% мощности основного сервера, чтобы актуализировать бд.
- Переключение на резервную копию быстрее

- Можно выполнять запросы на реплике, снимая часть нагрузки

Минусы:

- Журналы log не документированы и могут меняться, поэтому использовать физическую репликацию можно только между экземплярами одной СУБД.

Особенности:

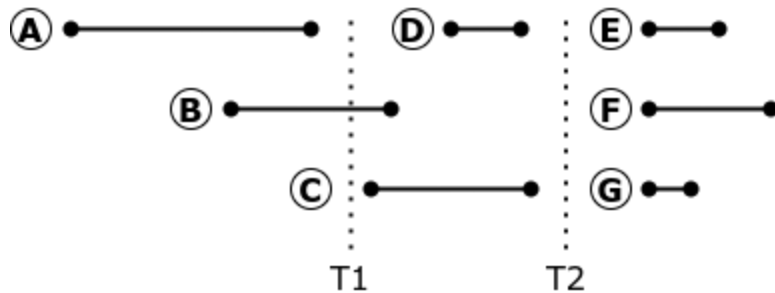
Физическая репликация может быть как синхронной, так и асинхронной. При асинхронной репликации всегда есть некий набор транзакций, которые завершены на основной базе, но ещё не дошли до резервной, и в случае перехода на резервную базу при сбое основной эти транзакции будут потеряны. При синхронной репликации завершение операции commit означает, что все журнальные записи, относящиеся к данной транзакции, переданы на реплику. Важно понимать, что получение репликой журнала не означает применения изменений к данным. Физическая репликация почти во всем лучше блочной.

Логическая репликация

Все изменения в базе данных происходят в результате вызовов её API – например, в результате выполнения SQL-запросов.

Но нужно придерживаться правил -

1. Нельзя начинать транзакцию, пока не завершены все транзакции, которые должны закончиться раньше. Так на рисунке ниже нельзя запускать транзакцию D, пока не завершены транзакции A и B.
2. Нельзя завершать транзакцию, пока не начаты все транзакции, которые должны закончиться до завершения текущей транзакции. Так на рисунке ниже даже если транзакция B выполнялась мгновенно, завершить её можно только после того, как начнётся транзакция C.



Плюсы:

- База-реплика открыта и доступна не только на чтение, но и на запись. Это позволяет использовать реплику для выполнения части запросов
- Настройка набора реплицируемых данных на уровне таблиц (при физической репликации – на уровне файлов и табличных пространств, при блочной репликации – на уровне томов);
- Репликация между разными версиями СУБД или даже между СУБД разных производителей

Минусы:

- Все реплицируемые данные обязаны иметь первичные ключи
- Логическая репликация поддерживает не все типы данных
- Логическая репликация на практике не бывает полностью синхронной: время от получения изменений до их применения слишком велико, чтобы основная база могла ждать
- Логическая репликация создаёт большую нагрузку на реплику;
- При переключении приложение должно иметь возможность убедиться, что все изменения с основной базы, применены на реплике – СУБД зачастую сама не может этого определить, так как для неё режимы реплики и основной базы эквивалентны.

В таблице не смотрим на блочную репликацию агентом, это почти то же самое

	Блочная репликация СХД	Блочная репликация агентом	Физическая репликация	Логическая репликация СУБД
Воспроизведение источника	Побайтно	Побайтно	Побайтно	Логически
Выборочная репликация	На уровне томов	На уровне томов	На уровне файлов	На уровне таблиц и строк
Объём трафика	X	X	X/7..X/5	X/7..X/5
Скорость переключения	5 мин... часы	5 мин... часы	1..10 мин	1..10 мин
Гарантия переключения	+	+	+++	+
Доступность реплики	∅	∅	RO	R/W
Топология репликации	точка-точка	точка-точка broadcast	точка-точка broadcast каскад	точка-точка broadcast каскад встречная* p2p*
Нагрузка на источник	∅	∅	—	— —
Простота настройки	+++	++	++	++
Стоимость дополнительного ПО	— —	— —	—	—

37. Физическая репликация. Реализация в PostgreSQL.

При физической репликации, серверы имеют назначенные роли: один является **ведущим («мастер»)** и один или несколько серверов являются **ведомыми («реплики»)**. Мастер передает на реплику журнальные записи, а реплика применяет эти записи к своим файлам данных.

Применение происходит чисто механически, без «понимания смысла» изменений, и выполняется быстрее, чем работало бы повторное выполнение операторов SQL. Но при этом критична двоичная совместимость между серверами — они должны работать на одной и той же программно-аппаратной платформе и на них должны быть запущены одинаковые основные версии PostgreSQL.

Поскольку журнал является общим для всего кластера, то и реплицировать можно только кластер целиком, а не отдельные базы данных или таблицы. Возможность «отфильтровать» журнальные записи отсутствует. Реплика не может генерировать собственных журнальных записей, она лишь применяет записи мастера. Поэтому при таком подходе реплика может быть доступна только для чтения — никакие операции, изменяющие данные, на реплике не допустимы.

Реализация в PostgreSQL:

Есть два способа доставки журналов от мастера к реплике. Основной, который используется на практике — **поточковая репликация**. В этом случае реплика подключается к мастеру по протоколу репликации и читает поток записей WAL. За счет этого при потоковой репликации отставание реплики от мастера **минимально** (и при необходимости может быть сведено к нулю). На реплике за прием журнальных записей отвечает процесс wal receiver. Запустившись, он подключается к мастеру по протоколу репликации, а мастер запускает процесс wal sender, который обслуживает это соединение. Полученные записи пишутся на диск в виде сегментов WAL так же, как это происходит на мастере. Далее они (сразу или с задержкой) применяются к файлам данных. За применение отвечает процесс startup — тот же самый, который обеспечивает восстановление после сбоев, только теперь работает постоянно. Поточковая репликация может, если нужно, использовать слот репликации

Другой способ доставки журнальных записей состоит в том, чтобы предоставить реплике доступ к **архиву журналов** — точно так же, как при обычном восстановлении из архива. Если реплика по каким-то причинам не сможет получить очередную журнальную запись по протоколу репликации (например, из-за обрыва связи), она попытается прочитать ее из архива с помощью команды `restore_command` из файла `recovery.conf`. При восстановлении связи реплика снова автоматически переключится на использование потоковой репликации.

(Это про реплику все, если что)

Фиксация транзакций может выполняться в двух режимах. В более быстром **асинхронном** режиме есть шанс потерять уже зафиксированные изменения в случае сбоя. В **синхронном** режиме команда COMMIT не завершается, пока запись не дойдет до диска. При наличии нескольких серверов надежность можно еще более увеличить, дожидаясь подтверждения от реплики о получении записи, тогда данные не пропадут даже при выходе из строя носителя.

Синхронная репликация включается тем же параметром `synchronous_commit`, что и синхронная фиксация. Разные значения этого параметра позволяют управлять уровнем надежности.

38. Настройка физической репликации. wal_keep_size, слоты.

Общая инфа про физическую репликацию в прошлом и позапрошлом вопросе
Но еще раз:

Потоковая репликация (Streaming Replication). (Это один из вариантов физической репликации, рассматриваем его, потому что другого в постгресе нет!!)

Это репликация, при которой от основного сервера PostgreSQL на реплики передается WAL. И каждая реплика затем по этому журналу изменяет свои данные. Для настройки такой репликации все серверы должны быть одной версии, работать на одной ОС и архитектуре. Потоковая репликация в Postgres бывает двух видов — асинхронная и синхронная.

- Асинхронная репликация. В этом случае PostgreSQL сначала применит изменения на основном узле и только потом отправит записи из WAL на реплики. Преимущество такого способа — быстрое подтверждение транзакции, т.к. не нужно ждать пока все реплики применят изменения. Недостаток в том, что при падении основного сервера часть данных на репликах может потеряться, так как изменения не успели продублироваться.
- Синхронная репликация. В этом случае изменения сначала записываются в WAL хотя бы одной реплики и только после этого фиксируются на основном сервере. Преимущество — более надежный способ, при котором сложнее потерять данные. Недостаток — операции выполняются медленнее, потому что прежде чем подтвердить транзакцию, нужно сначала продублировать ее на реплике.

До девятой версии в PostgreSQL для создания «теплого» резервного сервера использовался WAL archiving. В версии 9.0 появилась потоковая репликация с возможностью создания «горячего» read-only сервера. В следующей версии PostgreSQL 9.4 появился новый функционал для создания потоковой репликации под названием **replication slots**.

В контексте логической репликации слот представляет собой поток изменений, которые могут быть воспроизведены клиенту в том порядке, в котором они были внесены на исходном сервере. Каждый слот передает последовательность изменений из одной базы данных.

Слот репликации имеет идентификатор, который уникален для всех баз данных в кластере PostgreSQL. Слоты сохраняются независимо от используемого соединения и являются аварийно-безопасными.

Логический слот будет выдавать каждое изменение только один раз при нормальной работе. Текущая позиция каждого слота сохраняется только на контрольной точке, поэтому в случае сбоя слот может вернуться к более раннему состоянию, что затем приведет к повторной отправке последних изменений при перезапуске сервера.

Для одной базы данных может существовать несколько независимых слотов. Каждый слот имеет свое собственное состояние, что позволяет разным потребителям получать изменения из разных точек потока изменений базы данных. Для большинства приложений для каждого потребителя потребуется отдельный слот.

Логический слот репликации ничего не знает о состоянии получателей. Возможно даже, что несколько разных получателей используют один и тот же слот в разное время; они просто получают изменения, следующие за тем, когда последний получатель перестал их использовать. Только один получатель может принимать изменения из слота в любой момент времени.

Wal_keep_size

Задаёт минимальный размер сегментов прошлых файлов журнала, хранящихся в каталоге `pg_wal`, на случай, если резервному серверу потребуется извлечь их для потоковой репликации. Если резервный сервер, подключенный к отправляющему серверу, отстаёт более чем на мегабайты `wal_keep_size`, отправляющий сервер может удалить сегмент WAL, который все ещё необходим резервному серверу, и в этом случае соединение репликации будет прервано. В результате нисходящие соединения также в конечном итоге выйдут из строя. (Однако резервный сервер может восстановить данные, извлекая сегмент из архива, если используется архивирование WAL)

Это устанавливает только минимальный размер сегментов, сохраняемых в `pg_wal`; системе может потребоваться сохранить больше сегментов для архивирования WAL или для восстановления с контрольной точки. Если значение `wal_keep_size` равно нулю (по умолчанию), система не сохраняет никаких дополнительных сегментов для целей ожидания, поэтому количество старых сегментов WAL, доступных резервным серверам, зависит от местоположения

предыдущей контрольной точки и состояния архивирования WAL. Если это значение указано без единиц измерения, оно принимается за мегабайты.

39. (Adv.) Синхронная и асинхронная репликация в PostgreSQL.

По умолчанию в Postgres потоковая репликация **асинхронна**. Если ведущий сервер выходит из строя, некоторые транзакции, которые были подтверждены, но не переданы на резервный, могут быть потеряны. Объем потерянных данных пропорционален задержке репликации на момент отработки отказа.

Синхронная репликация предоставляет возможность **гарантировать**, что все изменения, внесённые в транзакции, были переданы одному или нескольким синхронным резервным серверам. Это повышает стандартный уровень надёжности, гарантируемый при фиксации транзакции.

Для включения синхронной репликации на мастере нужно задать следующие параметры (postgresql.conf):

- *synchronous_commit* = *on* (значение по умолчанию, обычно действий не требуется)
 - значение *on* – каждая транзакция будет ожидать подтверждение того, что на резервном сервере произошла запись транзакции в надёжное хранилище.
 - может иметь значение *remote_writer* – в случае подтверждения транзакции ответ от резервного сервера об успешном подтверждении будет передан, когда данные запишутся в операционной системе, но не когда данные будут реально сохранены на диске
 - снижение надёжности (потеря данных резервным сервером при падении ОС (не PostgreSQL))
 - сокращение времени отклика
 - может иметь значение *remote_apply* – для завершения фиксирования транзакции потребуется дождаться, чтобы текущие синхронные резервные серверы сообщили, что они воспроизвели транзакцию и её могут видеть запросы пользователей

	Долгов-ть локального коммита	Долгов-ть сбой БД	Долгов-ть сбой ОС	standby - согл-ть запросов
remote_apply	+	+	+	+
on	+	+	+	
remote_write	+	+		
local	+			
off				

- *synchronous_standby_names* в непустое значение:
 - *synchronous_standby_names* = 's2, s3, s4' – узлы s2, s3, s4 – работают синхронно, остальные (если есть) – асинхронно.
 - *synchronous_standby_names* = 'FIRST 2 (s1, s2, s3)' – узлы s1, s2 – работают синхронно, s3 – потенциальный синхронный (будет работать синхронно при падении s1 или s2), остальные (если есть) – асинхронно
 - *synchronous_standby_names* = 'ANY 2 (s1, s2, s3)' – транзакция зафиксируется только после получения подтверждений как минимум от 2 резервных синхронных серверов из s1, s2, s3 (ну и разумеется, s1, s2, s3 – работают синхронно, а все остальные – асинхронно).

Преимущества и недостатки синхронной репликации над асинхронной:

- (-) непродуманное использование синхронной репликации приведёт к снижению производительности БД из-за увеличения времени отклика и числа конфликтов (длительное ожидание подтверждения)
- (-) фиксирование транзакции может не завершиться никогда, если один из синхронных резервных серверов выйдет из строя.
- (+) повышение надёжности и отказоустойчивости (потеря данных может произойти только в случае одновременного выхода из строя ведущего и резервного серверов)

40. (Adv.) Ступенчатая (каскадная) репликация.

Свойство каскадной репликации позволяет резервному серверу принимать соединения репликации и потоки WAL от других резервных, выступающих посредниками. Это может быть полезно для уменьшения числа непосредственных подключений к главному серверу, а также для уменьшения накладных расходов при передаче данных в интрасети.

Резервный сервер, выступающий как получатель и отправитель, называется **каскадным резервным сервером**. Резервные серверы, стоящие ближе к главному, называются серверами **верхнего уровня**, а более отдалённые — серверами **нижнего уровня**. Каскадная репликация не накладывает ограничений на количество или организацию последующих уровней, а каждый резервный соединяется только с одним сервером вышестоящего уровня, который в конце концов соединяется с единственным главным/ведущим сервером.

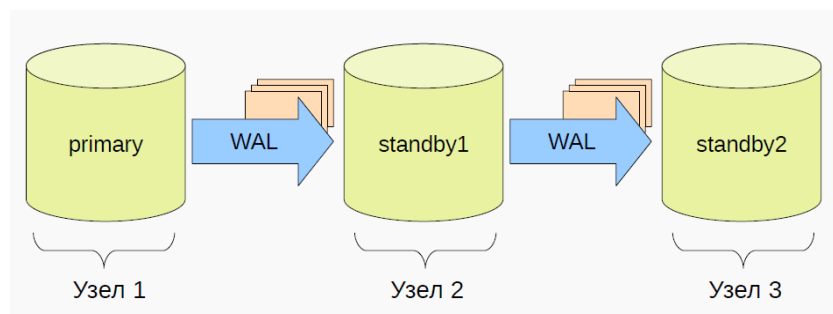
Резервный сервер каскадной репликации не только получает записи WAL от главного, но так же восстанавливает их из архива. Таким образом, даже если соединение с сервером более высокого уровня разорвётся, потоковая репликация для последующих уровней будет продолжаться до исчерпания доступных записей WAL.

Каскадная репликация в текущей реализации **асинхронна**. Параметры синхронной репликации в настоящее время не оказывают влияние на каскадную репликацию.

Распространение обратной связи горячего резерва работает от нижестоящего уровня к вышестоящему уровню вне зависимости от способа организации связи.

Если вышестоящий резервный сервер будет преобразован в новый главный, нижестоящие серверы продолжают получать поток с нового главного при условии, что *recovery_target_timeline* установлен в значение *'latest'*.

Для использования каскадной репликации необходимо настроить резервный каскадный сервер на приём соединений репликации (то есть установить *max_wal_senders* и *hot_standby*, настроить *host-based authentication*). Также может быть необходимо настроить на нижестоящем резервном значению *primary_conninfo* на каскадный резервный сервер.



41. (Adv.*) Логическая репликация в PostgreSQL

Логическая репликация — пересылаются команды, реконструированные из WAL.

- Требуется — `wal_level = logical`.
- `standby` — осуществляет восстановление на основе полученных команд для поддержки актуального состояния.
- Используется модель «публикаций» «подписок»:
 - после логического декодирования команды публикуются на мастере;
 - опубликованные команды могут получить подписанные standby.

Настройка на master:

- `wal_level = logical`
- Пользователь для репликации (master):
 - `CREATE USER LOGICREPL WITH REPLICATION ...`
- `pg_hba.conf`: `host replication logicrepl all md5`.
- Выдать права для LOGICREPL на нужные объекты.
 - `CREATE PUBLICATION all_t_publ FOR ALL TABLES;`

Настройка на slave:

- `wal_level = logical`
- Воссоздать структуру таблиц.
- `CREATE SUBSCRIPTION all_t_subscr CONNECTION 'user=logicrepl ... dbname=db_name' PUBLICATION all_t_publ;`

Особенности:

- Можно реплицировать часть БД (отдельный набор таблиц).
- Не поддерживается репликация DDL.
- Возможна репликация даже для узлов с разными версиями PostgreSQL.
- Standby может работать не в read-only режиме.