

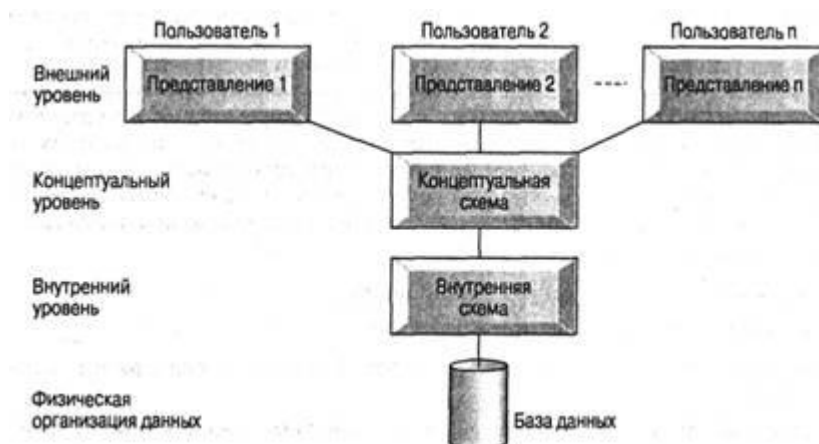
Оглавление

1. Архитектура ANSI-SPARC.....	3
2. Архитектура PostgreSQL.....	5
3. Разделяемая память PostgreSQL.....	5
4. Буферная память процессов PostgreSQL.....	6
5. Процессы, обеспечивающие работу PostgreSQL.....	6
6. Системный каталог, организация, способы взаимодействия.....	10
7. Схема, особенности работы со схемами в PostgreSQL, search_path.....	10
8. Управление доступом к данным в PostgreSQL, привилегии, пользователи, роли.	11
9. Работа с ролями, INHERIT, NOINHERIT.....	13
10. Установка и запуск PostgreSQL.....	14
11. PostgreSQL: template0, template1, назначение, особенности работы.....	15
12. Подключение к PostgreSQL, настройка, особенности.....	16
13. Виды и методы подключений к PostgreSQL, их настройка.....	17
14. PostgreSQL, создание Базы Данных.....	18
15. Файловая структура и конфигурация PostgreSQL.....	19
16. Табличные пространства. Назначение. Организация и настройка.....	20
17. Транзакции. Назначение. ACID.....	21
18. PostgreSQL: явные, неявные транзакции. Транзакционный DDL.....	22
19. PostgreSQL: время начала транзакции, время внутри транзакции.....	22
20. PostgreSQL: идентификация транзакции, особенности работы xid.....	22
21. PostgreSQL: MVCC, SSI.....	24
22. Виды и особенности конфликтов при параллельном доступе к данным.....	25
23. Изоляция транзакций. Режимы для организации доступа к данным.....	27
24. VACUUM.....	29
25. Восстановление данных. Базовые понятия. Организация. Контрольные точки.....	31
26. (Adv.) UNDO-журнал.....	33
27. (Adv.) REDO-журнал.....	35
28. (Adv.) UNDO/REDO-журнал.....	38
29. (Adv.) Восстановление данных. Подходы (steal/no-steal, force/no-force). ARIES.	41
30. Восстановление данных в PostgreSQL. WAL. LSN.....	41
31. Резервное копирование. Базовые понятия. Виды.....	43
32. Логическое резервное копирование. pg_dump, pg_dumpall, pg_restore.....	44
33. Физическое резервное копирование. Способы организации в PostgreSQL.....	46

34. PostgreSQL: непрерывное архивирование.....	46
35. Репликация данных в PostgreSQL. Базовые понятия.	49
36. Виды репликации. Особенности.	50
37. Физическая репликация. Реализация в PostgreSQL.	52
38. Настойка физической репликации. wal_keep_size, слоты.	53
39. (Adv.) Синхронная и асинхронная репликация в PostgreSQL.....	54
40. (Adv.) Ступенчатая (каскадная) репликация.....	56
41. (Adv.*) Логическая репликация в PostgreSQL.	57

1. Архитектура ANSI-SPARC.

Архитектура СУБД включает в себя 3 уровня:



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.

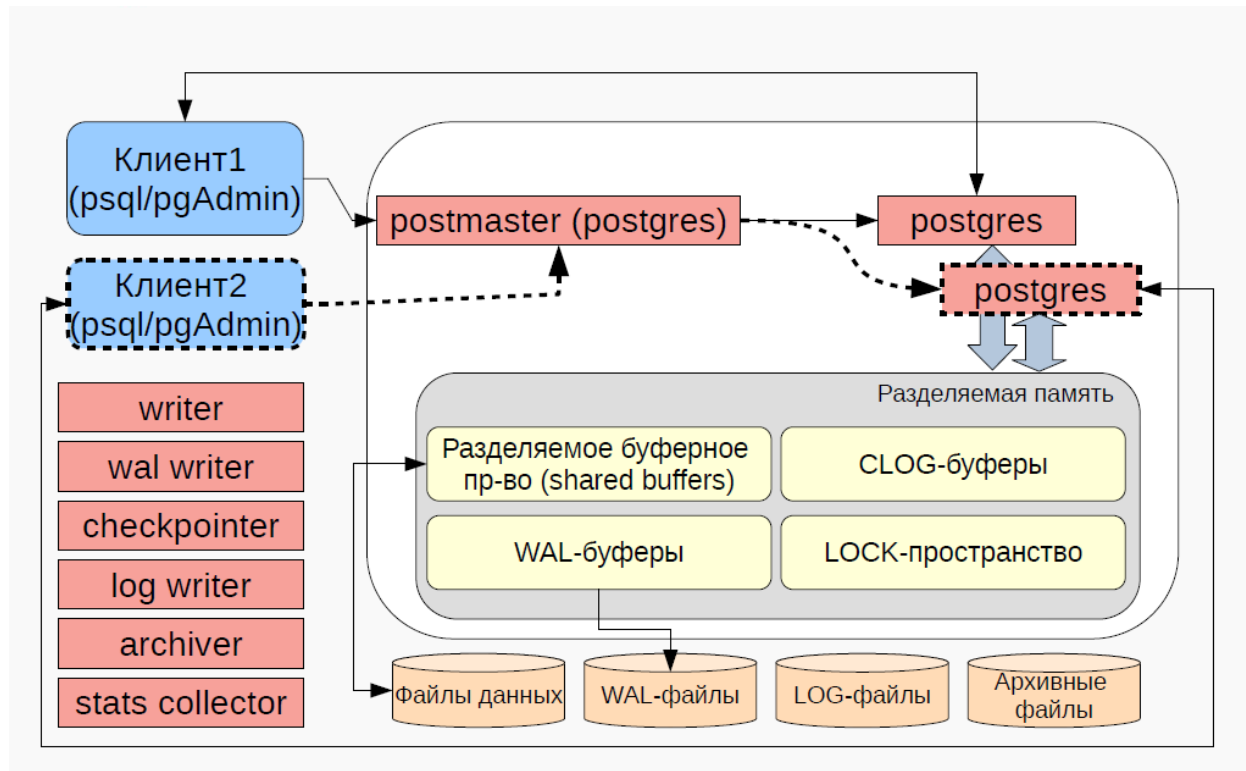
Общее описание базы данных называется *схемой базы данных*:

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 предназначаются.
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.



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 восстановления 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

- Файлы — в директории pg_xact;
- Размер автоматически устанавливается СУБД.
- Доступен серверным процессам.

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):

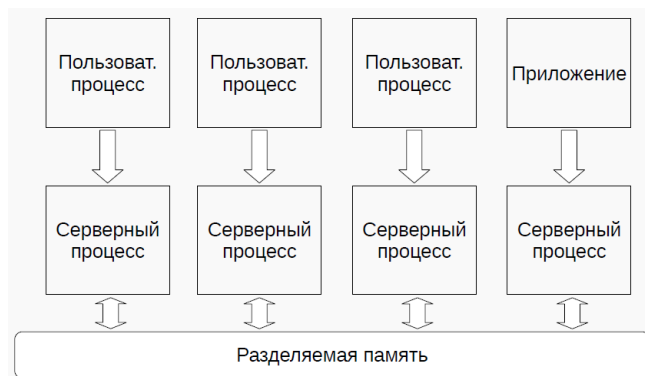
- Для каждого серверного пользовательского процесса выделено пространство для осуществления операций.
- По умолчанию — 4 MB.
- Может быть различных видов:
 - vacuum buffers;
 - рабочая память (work_mem) - DISTINCT, ORDER BY, JOIN;
 - вспомогательная рабочая память (maintenance_work_mem) — REINDEX;
 - temp_buffer — для работы со временными таблицами.

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.

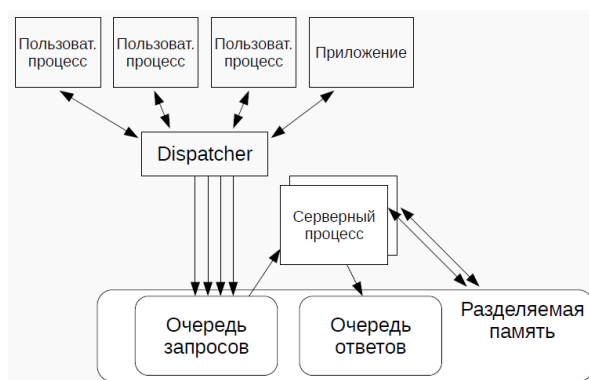
При работе в режиме **выделенного сервера (Dedicated server)** для каждого пользовательского процесса создаётся свой серверный.



Недостатками этого режима является проблема с ресурсами при одновременной работе множества пользователей (ресурсы конечны, а процессы достаточно дорогостоящие). Количество пользователей можно ограничить – новые пользователи будут получать отказ.

Режим выделенного сервера характерен для PostgreSQL и используется в нём по умолчанию.

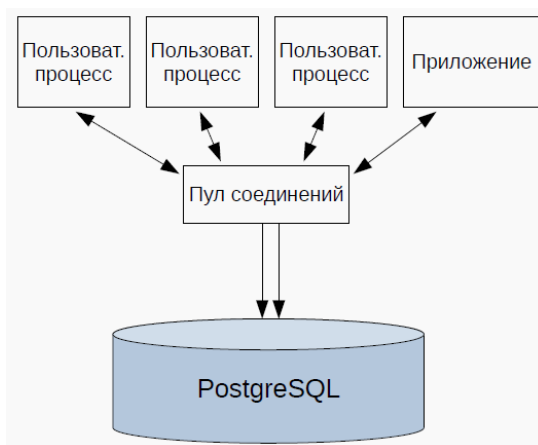
При работе в режиме **разделяемого сервера (Shared server)** для каждого пользовательского процесса серверный процесс выделяется диспетчером из специального пула.



Ограниченное количество серверных процессов. Диспетчер отдаёт задачу серверному процессу, серверный процесс выполняет её, кладёт в очередь ответов, тем самым освобождаясь. После диспетчер может отдать уже другую задачу на выполнение освободившемуся серверному процессу.

Недостаток в том, что снижает отзывчивость системы – при большом количестве пользователей могут быть бОльшие задержки, чем при работе в режиме выделенного сервера.

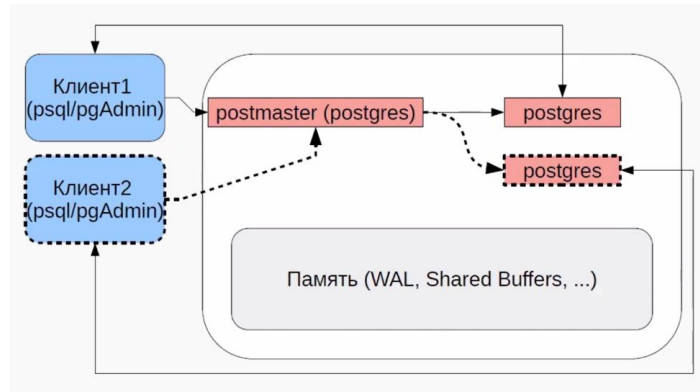
Режима разделяемого сервера в прямом виде в PostgreSQL нет. Но есть возможность организации внешнего пула соединений – с помощью сторонних утилит (pgBouncer, pgpool-II)



В этом случае диспетчер заменяется отдельным внешним сервисом, который принимает подключения, распределяет их и отправляет PostgreSQL, который уже навешивает на них свои процессы.

Процессы, обеспечивающие работу PostgreSQL:

1. postmaster (postgres) – главный процесс: слушает внешние подключения, проводит аутентификацию, создаёт (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 checkpoint.
 - d. bgwriter_delay
4. wal writer – периодически проверяет WAL буфер и записывает все незаписанные записи XLOG в сегменты WAL
5. checkpoint – 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 'somsdfsld';

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.

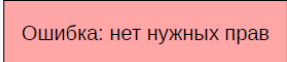
INHERIT — участники группы получают права ролей-групп их окружающих. Параметр **INHERIT** добавляется по умолчанию.

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

```
CREATE ROLE s123457 WITH LOGIN PASSWORD
'somsdfsls' 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 ...;
```

Значения добавлены

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] TO user1, user2, user3;
```

Пример:

```
REVOKE UPDATE, DELETE, TRUNCATE ON STUDENT FROM s4343453;
```

10. Установка и запуск PostgreSQL.

2 варианта:

- Графический установщик.
- Использование утилит:
 - использование подготовленных бинарных файлов
 - разные сборки для разных ОС;
 - компиляция исходных кодов (make)
 - нет сборки для используемой ОС;
 - нужна самая последняя версия\версия с внесенными изменениями;

Базовые компоненты PostgreSQL

- сервер (postgresql);
- клиент (postgresql-client);
- contrib, docs и другие компоненты.

Процесс установки:

1. Установка (сборка) базовых компонентов (пакетов):

```
sudo apt install postgresql-XX postgresql-client-XX ...
```

2. Создание системного пользователя (postgres):

```
adduser postgres
```

```
mkdir [PGDATA]
```

```
chown postgres [PGDATA]
```

3. Задание переменных окружения: PGDATA, ...
4. `initdb [-D PGDATA_path]`
5. Запуск экземпляра кластера БД (один из нижеперечисленных вариантов):
 - a. `postgres [-D PGDATA_path]`
 - b. `pg_ctl [-D PGDATA_path] -l logfile start`
 - c. Сервис: `sudo service postgresql start`

Инициализация кластера – `initdb`:

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

`initdb [-D PGDATA_path]`

`pg_ctl initdb`

Базы данных, доступные после установки:

- пользовательская БД (`postgres`) – принадлежит администратору БД - пользователю `postgres`;
- создается `template1`;
- создается клон `template1` — `template0`;

Создается суперпользователь — по умолчанию совпадает с именем пользователя, который запустил `initdb` (можно задать имя через `-U`: `initdb -U имя`).

Запуск сервера базы данных:

- Можно заранее установить переменную PGDATA:
 - `PGDATA = ...`
 - `export PGDATA`
 - `postgres` или `pg_ctl start -l logfile`
- PGDATA – должна быть проинициализирована перед запуском PostgreSQL

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

- 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).

12. Подключение к PostgreSQL, настройка, особенности.

Клиенты:

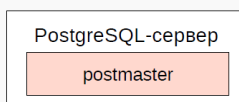
- psql:
 - интерактивный терминал PostgreSQL;
 - стандартный клиент для работы с PostgreSQL;
- pgAdmin:
 - графический клиент;
 - доступен для Windows, Unix, macOS;
- Клиенты, использующие libpq.

Для подключения к БД роль должна быть:

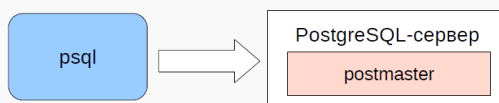
1. LOGIN;
2. содержать привилегию CONNECT на нужную БД;
3. разрешение в pg_hba.conf;

Подключение к серверу БД:

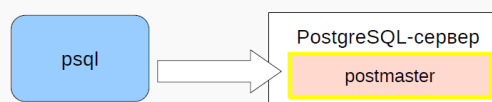
1) Запуск сервера PostgreSQL:
`postgres -p 2378 -D /home/myuser/pgd/data`



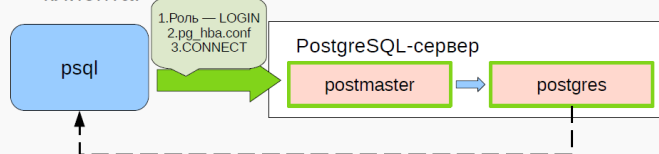
2) Подключаем клиент: `psql -p 2347`



3) экземпляр проверяет, может ли клиент получить доступ:



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



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

- Создается при работе initdb.
- По умолчанию располагается в PGDATA.
- Можно поменять через hba_file параметр:
 - при запуске postgres;
 - в postgresql.conf;
- 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()`
 - `pg_ctl reload`

Пример 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
```

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

Виды подключений:

- Локальный UNIX-сокеты (local):
 - 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`
- TCP/IP соединение (host):
 - указывается хост, порт:
 - `psql -h host -p port database`
 - `psql -h host -p port -U username database`
 - Записи `host` соответствуют подключениям с SSL и без SSL.
- `hostssl`
 - управляет подключениями, устанавливаемыми по TCP/IP с применением шифрования SSL.
 - чтобы использовать эту возможность, сервер должен быть собран с поддержкой SSL.
 - более того, механизм SSL должен быть включён параметром конфигурации `ssl`.
- `hostnossl`
 - этот тип записей противоположен `hostssl`, ему соответствуют только подключения по TCP/IP без шифрования SSL.

Методы подключений:

- `trust` — предоставить доступ всем из данной категории (опасно!);
- `reject` – отклоняет подключение безусловно
- по паролю (`scram-sha-256`, `md5`, `password`). Смена пароля: изменение роли или `\password`
- `Ident` — похоже на `peer`, но для `tcp/ip` (host).
- `peer` — сравнивается пользователь ОС с пользователем БД:
 - можно задать правила отображения пользователей в **pg_ident.conf** и параметр `map` в `pg_hba.conf`;
 - только `local`;

Пример правила с pg_ident.conf:

```
# Позволяет пользователям с любого компьютера 192.168.x.x подключаться
# к любой базе данных, если они проходят проверку ident. Если же ident
# говорит, например, что это пользователь "bryanh" и он запрашивает
# подключение как пользователь PostgreSQL "guest1", подключение
# будет разрешено, если в файле pg_ident.conf есть сопоставление
# "omicron", позволяющее пользователю "bryanh" подключаться как "guest1".
#
# TYPE DATABASE USER ADDRESS METHOD
host all all 192.168.0.0/16 ident map=omicron
```

pg_hba.conf

# MAPNAME	SYSTEM-USERNAME	PG-USERNAME
# bryanh также может подключаться как guest1		
omicron	bryanh	guest1

pg_ident.conf

14. PostgreSQL, создание Базы Данных.

Создание БД:

- CREATE DATABASE:
CREATE DATABASE newDb1 [WITH опции];
- createdb — утилита в директории [/postgresql]/bin:
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.
- Можно использовать в качестве шаблона другую базу:
 - CREATE DATABASE newDb3 TEMPLATE newDB1;
 - createdb -T newDB2 newDB4

Требования к базе-шаблону:

Чтобы база могла быть шаблоном нужно, чтобы к ней не было подключений других пользователей.

- Системный каталог pg_database:
 - *datistemplate* — БД создана, чтобы быть шаблоном;
 - может быть использована (как шаблон) только владельцем и суперпользователями (если флаг не установлен) и (если флаг установлен) пользователями с CREATEDB;
 - обычно установлен на template1 и template0
 - *datallowconn* — запрещаются новые подключения.
 - обычно установлен на template0 для избегания модификаций

15. Файловая структура и конфигурация PostgreSQL.

Файловая структура:

PGDATA – путь к директории данных кластера, установлена заранее.

- `base` – содержит подкаталоги для каждой базы данных с файлами данных: (1, 13201, 13202)
- `global` – содержит общие таблицы кластера, такие как `pg_database`
- `pg_wal` – директория с WAL-файлами.
- `pg_xact` – директория с данными о коммитах транзакций (CLOG).

Утилита `oid2name` — для просмотра имен (по `oid` или `filenode`):

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 размером в 1Gb: файл первого сегмента называется по номеру файлового узла (`filenode`), а последующие сегменты получают имена `filenode.1`, `filenode.2` и т. д.

Находятся в `$PGDATA/base` в поддиректории, соответствующей БД.

Конфигурация:

- Стандартный путь, по которому располагаются конфигурационные файлы: `/etc/postgresql/версия/main/`
- `postgresql.conf` — базовый файл для хранения настроек.
 - Создается при работе `initdb`.
 - По умолчанию располагается в PGDATA.
 - Представляет из себя набор параметров (имя, значение):
 - `search_path = 'someSchema, public'`
 - `shared_buffers = 128MB`
 - Определяют значения для всего 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 файлов:

1. postgresql.conf/ALTER SYSTEM — на уровне кластера;
2. ALTER DATABASE — изменить 1. на уровне базы данных;
3. ALTER ROLE — переписать 1. и 2. на уровне пользователя.
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.

Значения применяются при запуске новой сессии

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_тп')`
- Системный каталог: `pg_tablespace`
- `$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;
```

Табличное пространство можно задать для БД целиком:

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

Табличное пространство может быть изменено:

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

Зачем нужны табличные пространства?

- 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.

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;
 - pg_database, pg_authid, pg_tablespace, некоторые другие каталоги и индексы.

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.

ACID:

- 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 (Устойчивость, долговечность)
 - при успешном завершении транзакции результаты ее работы должны остаться в системе независимо от возможных сбоев оборудования, системы и т. д.

Пример:

BEGIN;

UPDATE BANK_ACCOUNT SET AccValue = AccValue+100 WHERE Client_ID = 1;

UPDATE BANK_ACCOUNT SET AccValue = AccValue-100 WHERE Client_ID = 2;

COMMIT;

- ROLLBACK – прерывает транзакцию, откатывает её, аннулируя все изменения;
- SAVEPOINT – точка сохранения в текущей транзакции (сохраняет состояние на момент установки);
- ROLLBACK TO SAVEPOINT – позволяет откатить состояние данных до состояния на определённой точке сохранения

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

Транзакции:

- Неявные (implicit) — СУБД начинает без явного указания;
 - любое действие осуществляется в контексте некоторой транзакции.

```
SELECT * FROM STUDENTS;  
  
↓  
  
BEGIN;  
SELECT * FROM STUDENTS;  
COMMIT;
```

- Явные (explicit) — транзакции, которые задает пользователь самостоятельно (BEGIN, COMMIT).

Если нет явного указания начала транзакции, любое предложение выполняется в рамках неявной транзакции;

Транзакционный DDL: транзакции в PostgreSQL поддерживаются не только для DML предложение но и для DDL предложений (например, несколько CREATE TABLE...)

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

Функции для получения текущего времени (CURRENT_TIME, CURRENT_TIMESTAMP, now()) возвращают время начала транзакции (так же, как и transaction_timestamp). Чтобы получить «настоящее» текущее время, следует использовать функцию clock_timestamp().

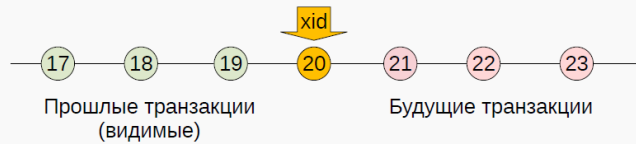
20. PostgreSQL: идентификация транзакции, особенности работы xid

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

- xid — уникален, 32 бита;
- беззнаковое число;
- При создании/модификации записи хранится xid транзакции.
- В простейшем случае: чем больше xid транзакции — тем позже она началась:
 - xid до: прошлое
 - xid после: будущее
- xid — начинается с 3 (значения 0, 1, 2 — зарезервированы).

Могут ли xid закончиться?

- Начинается выдача значений заново (с 3);



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

Проблема: что делать, когда перешли через MAX: **xid wraparound problem**

xid wraparound problem:

- Установка статуса неактуальных записей:
 - статус FROZEN — для неактуальных записей (~транзакции для которых уже давно завершены).
- Экономия xid:
 - если в транзакции нет операций, изменяющих состояние БД, ей выдается виртуальный xid.
 - xid выдаётся не сразу, а лишь тогда, когда будет выполняться операция, меняющая данные.

```
BEGIN;
SELECT * FROM STUDENTS;
UPDATE STUDENTS SET Group = 3100 ... ;
COMMIT;
```

Выдача xid:
 SELECT txid_current_if_assigned();
 -- xid текущей транзакции
 -- (если xid не назначен — NULL)

- WARNING: database test must be vacuumed within [...] transactions ...
 - Предупреждение, что доступные xid заканчиваются

Чтобы узнать xid текущей транзакции (bigint):

- SELECT pg_current_xact_id(); – 7435
- txid_current() – старая версия

Чтобы узнать xid операции, изменившей данные: SELECT xmin, * FROM STUDENTS;

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

xid транзакции

- xmin — хранит xid транзакции, в рамках которой запись (копия записи) была создана;
- xmax — хранит xid транзакции, в рамках которой запись была удалена;

- `ctmin` — идентификатор команды (порядковый номер команды в рамке транзакции), создавшей запись;
- `ctmax` — идентификатор команды (порядковый номер команды в рамке транзакции), удалившей запись;

21. PostgreSQL: MVCC, SSI

Реализация транзакций:

- Транзакции должны удовлетворять требованиям ACID.
- СУБД должна обеспечивать многопользовательский доступ:
 - разные пользователи могут запрашивать и изменять одни и те же данные в одно время.
- Существуют разные способы реализации транзакций:
 - могут основываться на блокировании ресурсов, синхронизации доступа к данным, создании копий данных, ...

Реализация транзакций (DML) в PostgreSQL — MVCC:

- MVCC — Multi-Version Concurrency Control
 - в PostgreSQL: Serializable Snapshot Isolation (SSI).
- 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;

```

BEGIN;
UPDATE STUDENTS SET GROUP=33316;
SELECT pg_current_xact_id_if_assigned(); COMMIT;

```

xmin	xmax	Stud_ID	Name	Surname	Group
400	410	1	Egor	Egorov	33313
410	0	1	Egor	Egorov	33316

«Мертвая» запись

Snapshot Isolation (SI):

- 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()`.
- `pg_current_snapshot()` (`txid_current_snapshot()`) — возвращает текущий снимок:
 - Формат: **`xmin:xmax:xid1,xid2`** (пример: 10:20:10,14,15)
 - `xmin` — минимальный идентификатор транзакции среди всех *активных*. Все транзакции, идентификаторы которых меньше `xmin`, уже либо зафиксированы и видимы, либо отменены и «мертвы».

- x_{max} – идентификатор на один больше идентификатора *последней завершённой* транзакции. Все транзакции, идентификаторы которых больше или равны x_{max} , на момент получения снимка ещё не завершены и не являются видимыми.
- $xid1, xid2 \dots$ – транзакции, *выполняющиеся в момент получения снимка*. Транзакции с такими идентификаторами, для которых $x_{min} \leq \text{ид} < x_{max}$, не попавшие в этот список, были уже завершены на момент получения снимка.
 - Верхняя граница (x_{max} — не включается);

Snapshot: предоставляется менеджером транзакций:

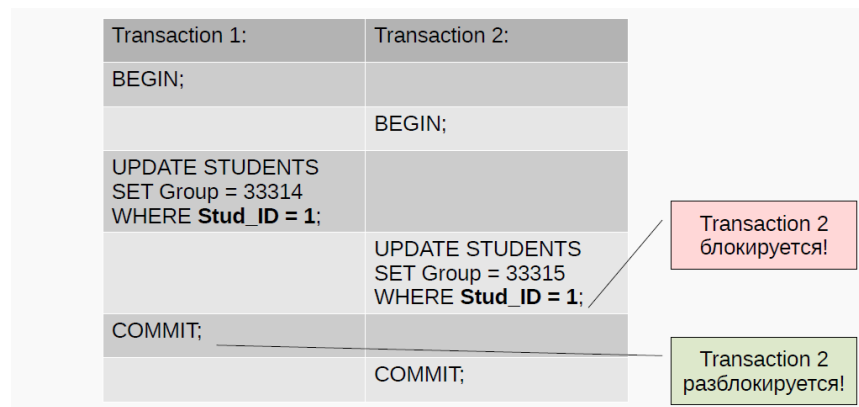
- READ COMMITTED: свой снимок для каждого выполняемого SQL-предложения.
- REPEATABLE READ, SERIALIZABLE: снимок — при выполнении только первого SQL-предложения.

Выбор подходящей версии записи определяется по **правилам проверки видимости**, которые используют:

- снимок;
- значения x_{min} , x_{max} , c_{min} , c_{max} , ...

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

Даже несмотря на использование MVCC в некоторых случаях исполнение транзакции блокируется:



Write-Write (WW) конфликт


Виды конфликтов:

- W-W (запись – запись). Первая транзакция изменила объект и не закончилась. Вторая транзакция пытается изменить этот объект. Результат – потеря обновления («lost update»).
- R-W (чтение – запись). Первая транзакция прочитала объект и не закончилась. Вторая транзакция пытается изменить этот объект. Результат – «неповторяющееся чтение» («nonrepeatable read»).

- W-R (запись – чтение). Первая транзакция изменила объект и не закончилась. Вторая транзакция пытается прочесть этот объект. Результат – «грязное чтение» («dirty read») или «фантомное чтение» («phantom read»).

Понятно, что конфликты типа R-R (чтение – чтение) отсутствуют, так как при чтении данные не изменяются.

«Грязное чтение»: Проблема возникает, если одна транзакция видит измененные данные 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								
		<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							
UPDATE STUDENTS SET Group = 33314 WHERE Stud_ID = 1;										
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>33314</td></tr></table>	Stud_ID	Name	Surname	Group	1	Ivan	Ivanov	33314
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:

Transaction 1:	Transaction 2:									
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><th>Stud_ID</th><th>Name</th><th>Surname</th><th>Group</th></tr><tr><td>2</td><td>Viktor</td><td>Viktorov</td><td>33315</td></tr></table>	Stud_ID	Name	Surname	Group	2	Viktor	Viktorov	33315
Stud_ID	Name	Surname	Group							
2	Viktor	Viktorov	33315							

Сами данные не изменяются

Сами данные не изменяются

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

Режимы для организации доступа к данным PostgreSQL:

Уровень изоляции	«Грязное» чтение	Неповторяемое чтение	Фантомное чтение	Аномалия сериализации
Read uncommitted (Чтение незафиксированных данных)	Допускается, но не в PG	Возможно	Возможно	Возможно
Read committed (Чтение зафиксированных данных)	Невозможно	Возможно	Возможно	Возможно
Repeatable read (Повторяемое чтение)	Невозможно	Невозможно	Допускается, но не в PG	Возможно
Serializable (Сериализуемость)	Невозможно	Невозможно	Невозможно	Невозможно

Можно запросить любой из четырёх уровней изоляции транзакций, однако внутри реализованы только три различных уровня, то есть режим Read Uncommitted в PostgreSQL действует как Read Committed. Также реализация Repeatable Read в PostgreSQL не допускает фантомное чтение.

Для выбора нужного уровня изоляции транзакций используется команда **SET TRANSACTION**. Также можно прописать уровень изоляции с командой **BEGIN**:

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

Уровень изоляции транзакции определяет, какие данные может видеть ЭТА транзакция, когда параллельно с ней выполняются другие 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

Команда SET TRANSACTION устанавливает характеристики текущей транзакции. На последующие транзакции она не влияет. **SET SESSION CHARACTERISTICS** устанавливает характеристики транзакции по умолчанию для последующих транзакций в рамках сеанса. Режимы транзакции для сеанса по умолчанию можно также задать в конфигурационных переменных default_transaction_isolation, default_transaction_read_only и default_transaction_deferrable.

Snapshot: предоставляется менеджером транзакций:

- READ COMMITTED: свой снимок для каждого выполняемого SQL-предложения.
- REPEATABLE READ, SERIALIZABLE: снимок — при выполнении только первого SQL-предложения.

Примеры режимов со snapshot:

1. READ COMMITTED

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;				


2. REPEATABLE READ

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: 321:321				
	SELECT * FROM STUDENTS;	Stud_ID	Name	Surname	Group
		1	Ivan	Ivanov	33313
	COMMIT;				

Snapshot для Tr2 (из-за режима Repeatable read) был сделан при первом SQL-предложении и больше не менялся (p.s. на слайде опечатка: перепутаны режимы Tr1 и Tr2)

3. SERIALIZABLE, REPEATABLE READ на примере LOST UPDATE

Запрещено, чтобы сторонние транзакции изменяли данные, использующиеся в текущей транзакции:

Tx 1:	Tx 2:
BEGIN ISOLATION LEVEL SERIALIZABLE;	
	BEGIN ISOLATION LEVEL SERIALIZABLE;
UPDATE STUDENTS SET Name = 'Roman' WHERE Stud_ID = 1;	Блокировка Tx 2 До завершения Tx 1
	UPDATE STUDENTS SET Name = Upper(Name) WHERE Stud_ID = 1;
COMMIT;	ERROR: could not serialize access...
Tx 1 завершится успешно	COMMIT; 

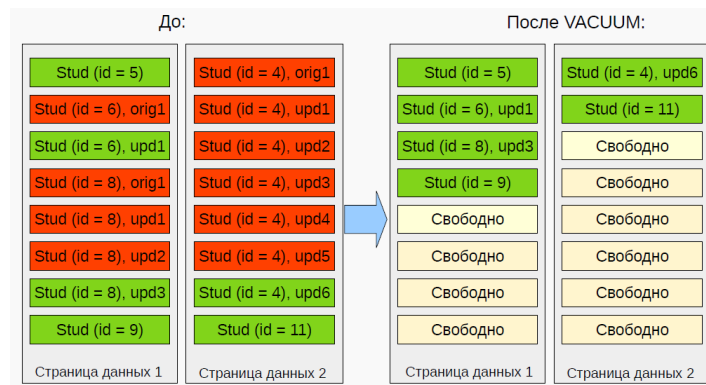
Tx2 будет ожидать завершения Tx1 (из-за блокировки на UPDATE), а после не сможет закоммититься (из-за режима SERIALIZABLE), тем самым не изменив данные.

24. VACUUM

Из-за особенностей реализации транзакций в PostgreSQL – MVCC (SSI), а именно: при изменении данных в транзакции — *создается новая копия данных*, а существующие копии данных *не изменяются*, с накопленными неактуальными данными нужно что-то делать (т. к. они занимают место на диске). Для периодической очистки старых версий данных (мертвых записей) используется **VACUUM**.

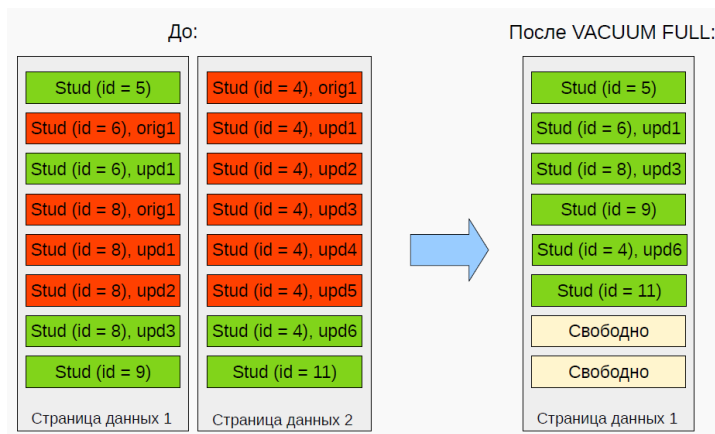
- **VACUUM:**
 - Можно запустить самому: `VACUUM table[, table2, ...];`
 - Периодический запуск настроен по умолчанию;
 - `autovacuum = true` (по умолчанию) – меняется в `postgresql.conf`;
 - `autovacuum_vacuum_threshold` – задаёт минимальное число изменённых или удалённых кортежей, при котором будет 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_cost_limit` – задаёт предел стоимости, который будет учитываться при автоматических операциях `VACUUM`.

- Режимы:
 - VACUUM



- Очищает ненужные записи и перемещает их в *рамках страницы* (количество страниц остаётся прежним – размер БД в памяти не меняется, но внутри страниц появляется свободное место для новых записей)

- VACUUM FULL;



- Разница с обычным VACUUM заключается в том, что с VACUUM FULL происходит *перераспределение страниц* – страницы оставшиеся пустыми освобождаются (место занимаемое базой в памяти уменьшается)
- Эта форма работает намного медленнее и запрашивает исключительную блокировку для каждой обрабатываемой таблицы.

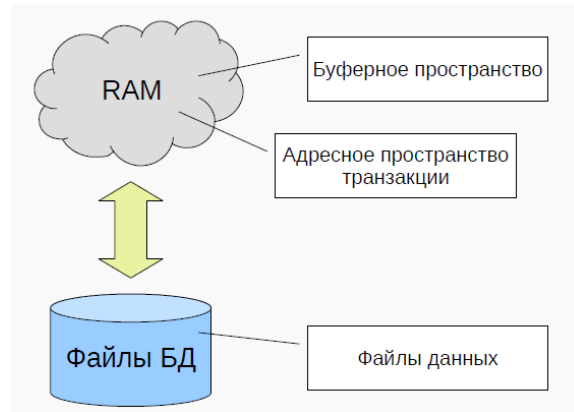
- VACUUM FREEZE;

- помечает содержимое таблицы специальной отметкой времени транзакции, которая сообщает postgres, что ее никогда не нужно очищать. При следующем обновлении этот замороженный идентификатор исчезнет.
- Время от времени демон autovacuum проверяет базу данных и ее таблицы, чтобы увидеть, что нужно очистить. Если таблица заморожена демон autovacuum просто пропустит ее.

Слишком короткий промежуток очистки может привести к тому, что она будет задерживать пользовательские операции изменения данных (так как на время очистки, все операции изменения прерываются), а также изменение данных в базе редко происходит равномерно (скорее скачкообразно), поэтому большая часть мертвых записей появляется в базе в довольно-таки короткий промежуток времени. Поэтому частый запуск VACUUM делает его менее эффективным, чем редкий запуск, позволяющий удалить большой объем накопившихся записей.

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

Память:



При взаимодействии с БД создаются различные транзакции и у каждой транзакции есть своё адресное пространство для выполнения различных операций и сохранения промежуточных результатов.

Операции для синхронизации различных пространств в памяти:

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

Пример:

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;
```

*Выполнение операции UPDATE сильно упрощено

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

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

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

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

COMMIT;

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

Незапланированный отказ работы экземпляра:



$D \leftarrow M(R_{374})$	1330	$R_{374}=1331, R_{375}=1330$	$R_{374}=1331, R_{375}=375$
$D \leftarrow M(R_{375})$	1330	$R_{374}=1331, R_{375}=1330$	$R_{374}=1331, R_{375}=1330$

Если система «упадет» в данный момент — проблема!

Несогласованное состояние данных

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;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	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
$D \leftarrow M(R_{374})$	$R_{374}=1331, R_{375}=375$	

Произошел сбой!

Операция	Диск	Журнал
		BEGIN T1;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	Возвращаем первоначальное значение
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	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
$D \leftarrow M(R_{374})$	$R_{374}=1331, R_{375}=375$	

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:

1. В журнал добавляется START CHKn: T_n, \dots, T_m , фиксируется на диске через WriteLog.
 - а. $T_n..T_m$ — ещё не завершённые (на момент начала создания контрольной точки) транзакции
2. *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 CHKn, фиксируется на диске через WriteLog.
4. Записи UNDO LOG перед START CHKn, у которой есть END CHKn, могут быть удалены.

Действия при восстановлении данных UNDO LOG с учетом контрольных точек:

- СУБД сканирует журнал и встречает запись END CHKn: T_n, \dots, T_m (а не START CHKn):
 - Значит установка контрольной точки n завершена;
 - Сканирование журнала до записи START CHKn, так как незавершенные транзакции могут быть только после записи START CHKn:
 - среди транзакций, которые начали работу после начала установки контрольной точки.
 - Остальные действия как раньше.
- СУБД сканирует журнал 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 (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 CHK_n: T_n, ..., T_m – начало создания контр. точки
- END CHK_n – 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;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(Room_Num) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	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
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, незавершенные транзакции – т. к. не завершены, по любому изменения не сохранены на диск.
- Когда в журнале встречается запись изменения (Tx: R, old_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.

Примеры:

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

Произошел сбой!

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

Нет COMMIT: транзакция не завершена. ABORT.

«WAL логирование с REDO связанная штука»

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 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;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	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
		COMMIT T1
WriteLog		

Произошел сбой!

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

Транзакция COMMIT и зафиксирована → повторяем сначала

Не смотря 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;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	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
		COMMIT T1

Произошел сбой!

Операция	Данные на диске	Журнал (WAL)
		BEGIN T1;
$T(v_R, Tx) \leftarrow M(R_{374})$	$R_{374}=374, R_{375}=375$	
$V_R(\text{Room_Num}) = 1331;$	$R_{374}=374, R_{375}=375$	
$M(R_{374}) \leftarrow T(v_R, Tx);$	$R_{374}=374, R_{375}=375$	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
		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 инициализировал синхронизацию WAL-буфера 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 *завершенными* 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 CHK_n: T_n, ..., T_m – начало создания контр. точки
- END CHK_n – 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'a и второго WriteLog'a – 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.
 - i. 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 — WAL-записи (XLOG).
- Эта информация используется для воссоздания актуального состояния данных в случае восстановления базы данных (например, после сбоя).
- Настраивается через параметр `wal_buffers` — размер wal-буфера:
 - по умолчанию 1/32 от разделяемого буфера;

WAL-записи:

- WAL-записи фиксируют различные изменения состояния данных в БД:
 - INSERT, UPDATE, DELETE, COMMIT, ...;
- WAL-записи записываются в:
 - Write Ahead Log;
 - При операции COMMIT — происходит синхронизация с WAL (логом транзакций);
- WAL — общий для всего кластера.

PostgreSQL: WAL-записи:

- Поддерживается REDO-лог (UNDO-лог по сути не нужен, потому что для незавершенных транзакций (и части завершённых) в таблицах хранятся данные со старыми значениями (из-за особенности MVCC)).
- 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 (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

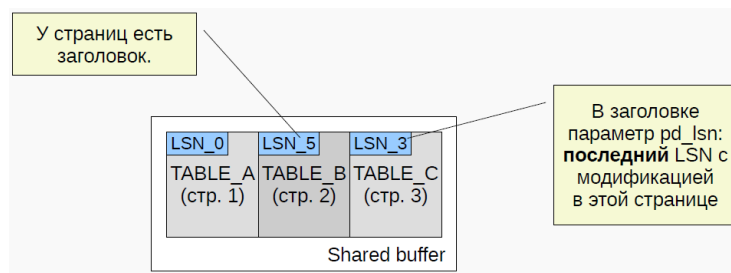
WAL-запись в журнале:

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

LSN:

У каждой WAL-записи есть LSN (Log Sequence Number) — уникальный идентификатор WAL-записи.

- 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:

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

Категории ошибок/сбоев:

- Ошибка при выполнении оператора – ошибка отдельной операции базы данных (выборка, вставка, обновление или удаление)
- Сбой пользовательского процесса – ошибка отдельного сеанса базы данных
- Сбой сети – потеря соединения с базой
- Ошибка пользователя – операция выполняется успешно, но сама операция неверна (удаление таблицы или ввод некорректных данных)
- Сбой экземпляра – непредвиденное завершение работы экземпляра базы данных
- Сбой носителя – потеря одного или нескольких файлов базы данных – файла данных, конфигурационного файла, WAL... (т. е. удаление файлов или сбой в работе диска)

Терминология:

- стратегия резервного копирования бывает:
 - полная — охватывает всю базу данных;
 - частичная — охватывает только часть базы данных.
- Резервная копия может содержать:
 - все страницы в пределах выбранных файлов;
 - ту информацию, которая изменилась с момента последнего резервного копирования (инкрементная):

- кумулятивная (изменения — до последнего уровня 0) – diff относительно полной резервной копии;
- дифференциальная (изменения — до последнего инкрементного резервного копирования) – diff относительно последней инкрементной копии.
- Виды резервного копирования:
 - «холодное» — создание резервной копии, когда экземпляр сервера БД остановлен;
 - «горячее» — создание резервной копии во время работы экземпляра сервера БД;
- Виды резервного копирования:
 - логическое;
 - физическое;

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`

- 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.

`pg_dump mydb > mydbdump`



`pg_dump -U postgres1 -f mydbdump mydb`

- Форматы РК:
 - **plain text**, текстовый формат — в файле ~SQL- предложения:
 - можно восстановить, выполнив в `psql`, *остальные форматы* – `pg_restore`;
 - 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`
 - **custom** формат:
 - возможно сжатие (если есть `zlib`);
 - бинарный файл;
 - в виде **директории** — один файл для каждой ~таблицы;
 - 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;
- **tar** — архив РК, в архиве содержимое, как для формата в виде директории;
- Можно задать создание скрипта для восстановления только определенных таблиц, структуры БД, содержимого БД:
 - pg_dump -t Student -f mydbdump mydb – only Student
 - pg_dump -T Student -f mydbdump mydb – all except Student
 - pg_dump -a -f mydbdump mydb – only data
 - pg_dump -s -f mydbdump mydb – only schema (DDL)
- Восстановление:
 - 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
 - psql mydb < mydbdump
 - Используемые в бэкапе пользователи, роли должны существовать/быть воссозданы до запуска скрипта.
 - При ошибке скрипт восстановления продолжит работу
 - Чтобы остановить скрипт после возникновения первой ошибки (невыполненного предложения):
 - psql --set ON_ERROR_STOP=on mydb < mydbdump
 - Можно указать, чтобы восстановление выполнялось в рамках одной транзакции:
 - psql --single-transaction mydb < mydbdump

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
- Возможен параллельный режим (опция -j)

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 тос-файл и сделать частичное восстановление.

33. Физическое резервное копирование. Способы организации в PostgreSQL.

При создания резервной копии происходит копирование файлов исходного кластера. Используются 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 системный 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.

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`, ...);

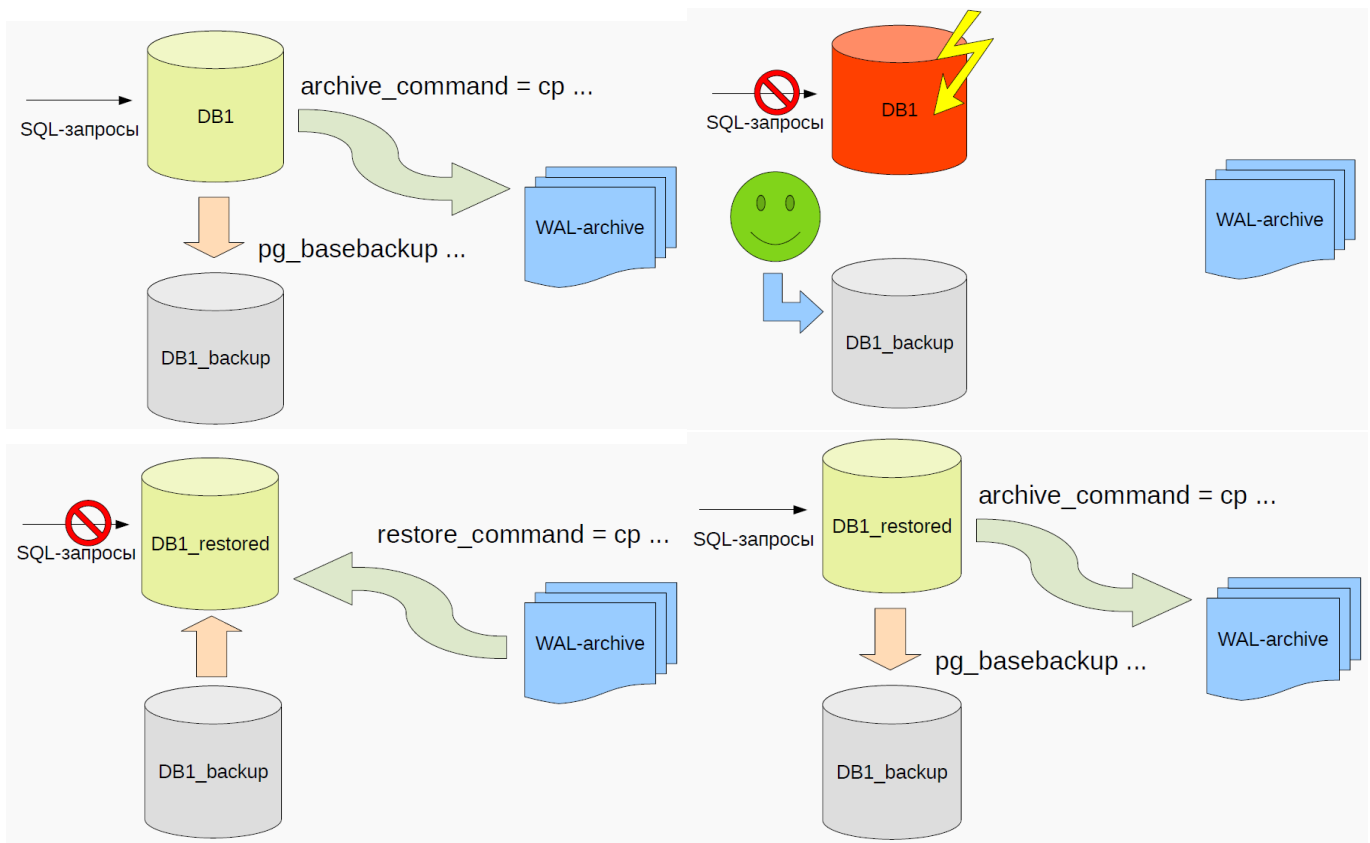
Резервная копия = полная копия + WAL-файлы

- БД не обязательно 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:

- Организация архивирования 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 = 'cp %p /path2/%f'`
 - `%p` — заменяется на полное имя файла для архивирования.
 - `%f` — заменяется на имя файла.
 - копирование сегментов WAL в `path2`.
 - команда должна возвращать 0 только в случае успешного завершения.
 - Частоту создания новых сегментов можно регулировать через `archive_timeout`:
 - если он небольшой будет создаваться много незаполненных сегментов и тратиться лишнее место.
- Создание базовой резервной копии
 - См. предыдущий пункт



Нормальный процесс работы при непрерывном архивировании => сбой и обнаружение проблемы => Восстановление => Состояние восстановлено

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 – задает 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 искать в pg_wal:
 - приоритет тем сегментам, которые в архиве.

Проблема: После сбоя и до восстановления БД недоступна!

Возможное решение: Резервный сервер, принимающий соединения, когда главный лежит

35. Репликация данных в PostgreSQL. Базовые понятия.

Что мы хотим?

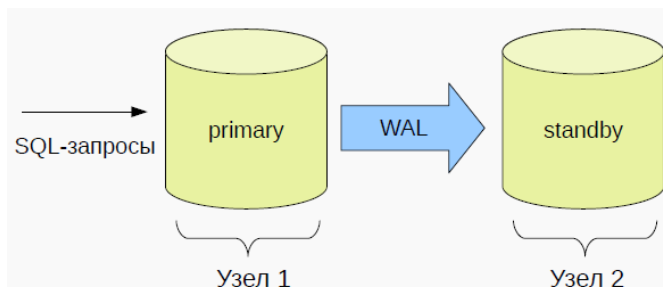
Высокий уровень доступности — (high availability) способность системы работать без сбоев (непрерывно) в течение определенного периода времени.

Как обеспечить высокий уровень доступности:

- обеспечить избыточность важных частей системы;
 - репликация данных.
- Failover/switchover – переключение мастера при падении/переключение без падения (например если у мастера странное поведение или нужно сервисное обслуживание).

Репликация:

- 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.



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.

Отказоустойчивость на разделяемых дисках (не репликация!)

Отказоустойчивость на разделяемых дисках позволяет избежать избыточности синхронизации путём задействования только одной копии базы данных. Она использует единственный дисковый массив, который разделяется между несколькими серверами. Если основной сервер БД откажет, резервный сервер может подключиться и запустить базу данных, что позволит восстановить БД после аварии. Это обеспечивает быстрое переключение без потери данных.

Существенное ограничение этого метода состоит в том, что в случае отказа или порчи разделяемого дискового массива оба сервера: главный и резервный — станут нерабочими. Другая особенность — резервный сервер никогда не получает доступ к разделяемым дискам во время работы главного.

Репликация на уровне файловой системы

Видоизменённая версия функциональности разделяемого оборудования представлена в виде репликации на уровне файловой системы, когда все изменения в файловой системе отражаются в файловой системе другого компьютера. Единственное ограничение: синхронизация должна выполняться методом, гарантирующим целостность копии файловой системы на резервном сервере — в частности, запись на резервном сервере должна происходить в том же порядке, что и на главном.

Трансляция журнала транзакций

Серверы тёплого и горячего резерва могут так же поддерживаться актуальными путём чтения потока записей из журнала изменений (WAL). Если основной сервер отказывает, резервный содержит почти все данные с него и может быть быстро преобразован в новый главный сервер БД. Это можно сделать синхронно или асинхронно, но может быть выполнено только на уровне сервера БД целиком.

Резервный сервер может быть реализован с применением трансляции файлов журналов, или потоковой репликации, или их комбинацией.

Репликация главный-резервный на основе триггеров

При репликации главный-резервный все запросы, изменяющие данные, пересылаются главному серверу. Главный сервер, в свою очередь, асинхронно пересылает изменённые данные резервному. Резервный сервер может обрабатывать запросы только на чтение при работающем главном. Такой резервный сервер идеален для обработки запросов к хранилищам данных.

Асинхронная репликация с несколькими главными серверами

Если серверы не находятся постоянно в единой сети или связаны низкоскоростным каналом, как например, ноутбуки или удалённые серверы, обеспечение согласованности данных между ними представляет проблему. Когда используется асинхронная репликация

с несколькими главными серверами, каждый из них работает независимо и периодически связывается с другими серверами для определения конфликтующих транзакций. Конфликты могут урегулироваться пользователем или по правилам их разрешения.

Синхронная репликация с несколькими главными серверами

При синхронной репликации с несколькими главными серверами каждый сервер может принимать запросы на запись, а изменённые данные передаются с получившего их сервера всем остальным, прежде чем транзакция будет подтверждена. Если запись производится интенсивно, это может провоцировать избыточные блокировки и задержки при фиксации, что приводит к снижению производительности. Запросы на чтение могут быть обработаны любым сервером.

Синхронная репликация с несколькими главными серверами лучше всего работает, когда преобладают операции чтения, хотя её большой плюс в том, что любой сервер может принимать запросы на запись — нет необходимости искусственно разделять нагрузку между главным и резервными серверами.

Репликация запросов в среднем слое

В схеме с репликацией запросов в среднем слое, средний слой перехватывает каждый SQL-запрос и пересылает его на один или все серверы. Каждый сервер работает независимо. Модифицирующие запросы должны быть направлены всем серверам, чтобы каждый из них получал все изменения. Но читающие запросы могут быть посланы только на один сервер, что позволяет перераспределить читающую нагрузку между всеми серверами.

Если запросы просто перенаправлять без изменений, функции подобные `random()`, `CURRENT_TIMESTAMP` и последовательности могут получить различные значения на разных серверах. Это происходит потому что каждый сервер работает независимо, а эти запросы неизбирательные (и действительно не изменяют строки). Если такая ситуация недопустима, или средний слой, или приложение должно запросить подобные значения с одного сервера, затем использовать его в других пишущих запросах. Другим способом является применения этого вида репликации совместно с другим традиционным набором репликации главный-резервный, то есть изменяющие данные запросы посылаются только на главный сервер, а затем применяются на резервном в процессе этой репликации, но не с помощью реплицирующего среднего слоя. Следует иметь в виду, что все транзакции фиксируются или прерываются на всех серверах, возможно с применением двухфазной фиксации.

Тип	Отказоустойчивость через разделяемые диски	Репликация файловой системы	Трансляция журнала транзакций	Репликация главный-резервный на основе триггеров	Репликация запросов в среднем слое	Асинхронная репликация с несколькими главными серверами	Синхронная репликация с несколькими главными серверами
Наиболее типичная реализация	NAS	DRBD	Потоковая репликация	Slony	pgpool-II	Bucardo	
Метод взаимодействия	разделяемые диски	дисковые блоки	WAL	Строки таблицы	SQL	Строки таблицы	Строки таблицы и блокировки строк

37. Физическая репликация. Реализация в PostgreSQL.

Физическая репликация — пересылаются файлы (WAL). Standby — содержит копию primary.

- Требуется — `wal_level = replica` (по умолчанию).
- Поточковая репликация (**streaming replication**) — WAL-записи передаются по соединению между primary и standby:
 - Использует свой протокол для отправки WAL-файлов — WAL-sender, WAL-receiver.
 - standby — осуществляет восстановление на основе полученных WAL для поддержки актуального состояния.

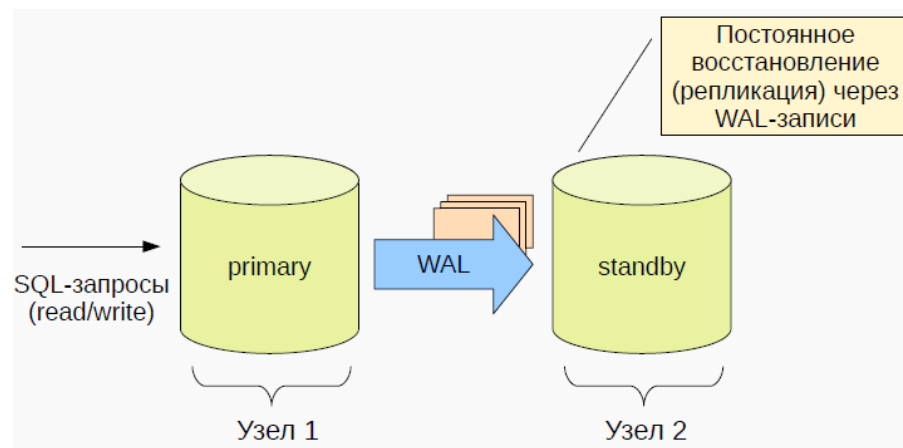
Особенности standby:

warm standby — режим standby, при котором логи (WAL), архивированные primary-сервером, собираются standby и происходит непрерывное восстановление состояния БД по полученным WAL. Warm standby не позволяет осуществлять запросы.


hot standby — сервер работает в режиме чтения — кроме восстановления по WAL, позволяет осуществлять запросы к данным (read-only).

Параметр `hot_standby = true` (по умолчанию).

Потоковая репликация:



38. Настройка физической репликации. wal_keep_size, слоты.



Физическая репликация в PostgreSQL

- Пользователь для репликации:
`CREATE USER PHYSREPL WITH REPLICATION ...`
- pg_hba.conf: host replication physrepl all md5
- backup: pg_basebackup -d 'conn params' -D 'path'

} primary

- standby.signal в PGDATA.
- postgresql.conf:
`primary_conninfo = 'host=... user=physrepl'`

} standby

standby.signal – файл в PGDATA. Позволяет запустить сервер в режиме standby.

Если установлен:

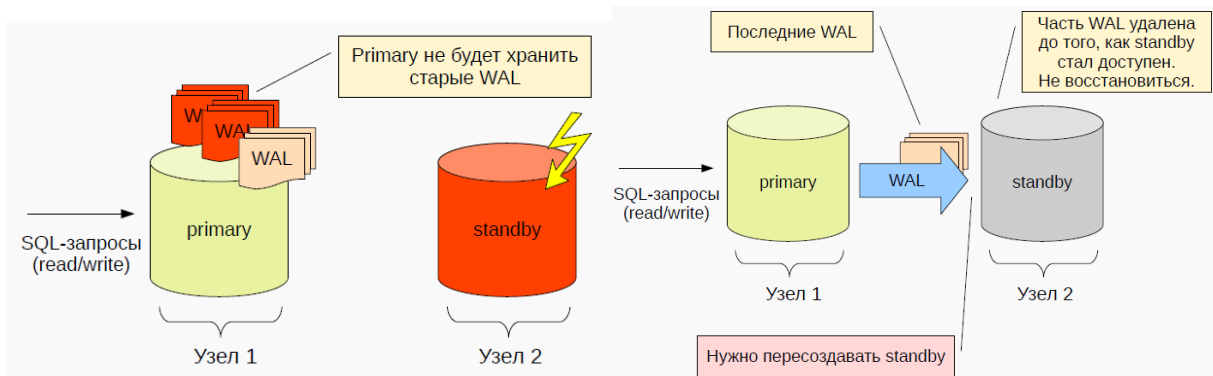
- После обработки архивированных в данный момент WAL, сервер продолжает пытаться получить новые WAL-сегменты через:
- restore_command
- WAL-записи с мастера, используя primary_conninfo

primary_conninfo — используется только, если standby.signal установлен.

```
# - Standby Servers -  
  
# These settings are ignored on a primary server.  
  
primary_conninfo = 'host=10.0.2.7 port=5432 user=postgres'  
#primary_slot_name = ''  
promote_trigger_file = 'failover'  
hot_standby = on
```

Проблема со standby:

При потоковой репликации без постоянной архивации на уровне файлов, сервер может избавиться от старых сегментов WAL до того, как резервный получит их. В этом случае резервный сервер потребует повторной инициализации из новой базовой резервной копии. Этого можно избежать, установив для **wal_keep_size** достаточно большое значение, при котором сегменты WAL будут защищены от ранней очистки, либо настроив слот репликации для резервного сервера. Если с резервного сервера доступен архив WAL, этого не требуется, так как резервный может всегда обратиться к архиву для восполнения пропущенных сегментов.



wal_keep_size:

- Задается в postgresql.conf.
- Определяет минимальное количество ненужных (для мастера) WAL-сегментов, которые должны храниться.
- По умолчанию — 0.
- Эти WAL-сегменты будут доступны standby:
 - НЕТ гарантии, что для standby будет достаточно этих
 - Однако ведомый сервер сможет восстановиться, выбрав эти сегменты из архива, если осуществляется архивация WAL.

Слоты:

- Позволяет хранить сегменты, пока они не отправлены standby.
- Таких сегментов может накопиться много (если standby не доступен).
- max_slot_wal_keep_size — позволяет ограничить число хранимых сегментов.
- Для использования:
 - нужно создать слот: `SELECT * FROM pg_create_physical_replication_slot('node2_slot');`
 - на standby: `primary_slot_name = 'node2_slot'`.

Полезные параметры/команды:

- max_wal_senders — число standby + 1. Если используется — wal_method=stream — еще +1 — для backup.
 - Определяет число процессов для отправки WAL (WAL sender).
- wal_method=stream — при использовании pg_basebackup будет открыто второе соединение для передачи WAL параллельно с созданием копии.
 - Используется по умолчанию.
- Просмотр состояния реплики:
 - представление pg_stat_replication;

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

По умолчанию в Postgres Pro 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
<i>remote_apply</i>	+	+	+	+
<i>on</i>	+	+	+	
<i>remote_write</i>	+	+		
<i>local</i>	+			
<i>off</i>				

- *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 зафиксирована только после получения подтверждений как минимум от 2 резервных синхронных серверов из *s1, s2, s3*.

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.

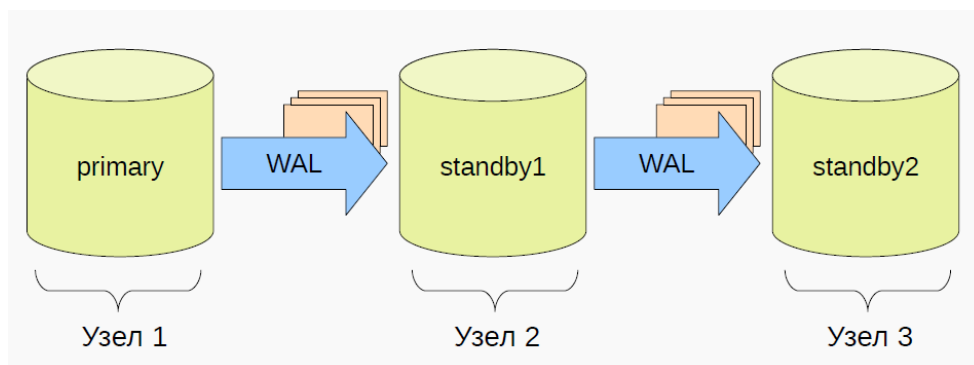
Резервный сервер каскадной репликации не только получает записи WAL от главного, но так же восстанавливает их из архива. Таким образом, даже если соединение с сервером более высокого уровня разорвётся, потоковая репликация для последующих уровней будет продолжаться до исчерпания доступных записей WAL.

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'.

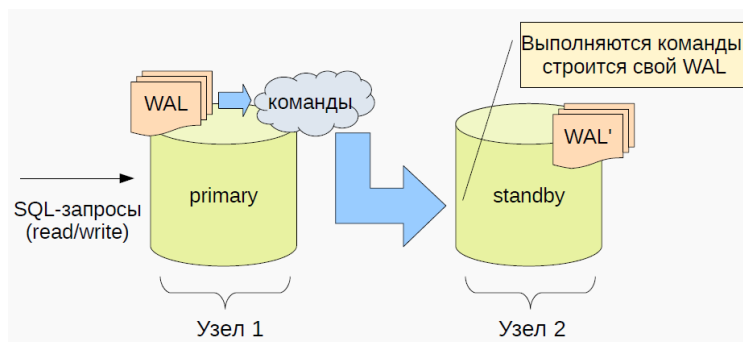
Для использования каскадной репликации необходимо настроить резервный каскадный сервер на приём соединений репликации (то есть установить `max_wal_senders` и `hot_standby`, настроить `host-based authentication`). Так же может быть необходимо настроить на нижестоящем резервном значение `primary_conninfo`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