ChatGPT

Подготовка к экзамену по дисциплине «Распределённые системы хранения данных»

Теоретические вопросы

1. Архитектура PostgreSQL (p12-s6)

Вопрос: *Опишите архитектуру системы управления базами данных PostgreSQL – из каких основных компонентов она состоит? Как организованы процессы и память сервера?*

Ответ: PostgreSQL реализует многопроцессную **клиент-серверную архитектуру**. При запуске сервера стартует процесс **postmaster** (он же *postgres* – главный процесс), который слушает подключения. Каждый новый клиент (например, через `psql` или pgAdmin) при подключении порождает отдельный **серверный процесс** для обслуживания этого соединения (модель *dedicated server*, один процесс на клиента). Таким образом, в рамках одного инстанса СУБД PostgreSQL работает **несколько процессов**: один мастер-процесс и отдельные процессы на каждого подключившегося пользователя. Кроме них, PostgreSQL запускает ряд **фоновых процессов** (background processes), отвечающих за обслуживание системы: - **writer process** (background writer) – периодически сбрасывает «грязные» страницы (изменённые данные) из буферов на диск; - **WAL writer** – записывает изменения из WAL-буфера в лог транзакций на диске; - **checkpointer** – создает контрольные точки (checkpoint) — моменты синхронизации буферов с диском; - **archiver** – копирует сегменты WAL в архив (если включено архивирование); - **stats collector** – собирает статистику работы (например, для `pg_stat` представлений); - **autovacuum launcher** – следит за необходимостью автомактического вакуума таблиц и запускает работники autovacuum; - **logging collector** – перенаправляет лог-сообщения в файлы логов.

Таким образом, **инстанс PostgreSQL** состоит из набора процессов ОС, объединённых общим доступом к общим областям памяти (Shared Memory).

Память PostgreSQL поделена на: - **Разделяемую память** – доступна всем процессам инстанса. К ней относится прежде всего область **shared buffers** (разделяемое буферное пространство), где хранятся копии страниц данных, считанных с диска. Shared buffers используются для кэширования данных и уменьшения числа дисковых операций: если требуемой страницы данных нет в буфере, она читается с диска и помещается туда для дальнейшего использования. Размер shared buffers настраивается параметром `shared_buffers` (по умолчанию 128 MB). - В разделяемой памяти также выделены специальные буферы журналов: **WAL-буфер** (для буферизации записей журнала изменений до их записи на диск) и **CLOG-буфер** (для фиксации статусов транзакций). - **WAL-буфер** хранит записи журнала предзаписи; размер его по умолчанию составляет 1/32 от shared buffers и настраивается параметром `wal_buffers`. - **CLOG** (Commit LOG) – журнал состояний транзакций, содержащий пометки о подтверждении (COMMITTED), откате (ABORTED) и пр. транзакций; соответствующие страницы хранятся в pg_xact и кешируются в памяти (CLOG buffers). - **Lock space** – область shared memory для данных о блокировках (таблиц, строк и пр.), доступна всем процессам; её размер определяется параметром

`max_locks_per_transaction`. - **Неразделяемая (локальная) память процессов** – у каждого серверного процесса есть своя область для операций: например, рабочая память запросов (`work_mem`) для сортировок, хеширований при JOIN, память под временные таблицы (`temp_buffers`), буферы VACUUM и пр.. По умолчанию на процесс отводится ~4MB такой памяти, настройка зависит от типа операций.

Структура хранения данных в PostgreSQL организована следующим образом: - **Кластер баз данных** – совокупность всех баз, управляемых одним запущенным экземпляром сервера (процессами PostgreSQL). Каждому кластеру соответствует своя директория с данными (**PGDATA**). - **Экземпляр** (инстанс) – совокупность процессов СУБД, обслуживающих кластер, и выделенная им разделяемая память.

Работа сервера происходит по правилу: клиентские процессы обращаются к shared buffers (разделяемому кешу) для чтения/записи данных; фоновые процессы заботятся о периодической выгрузке данных на диск и обслуживании журнала. Перед выполнением запроса СУБД проверяет права доступа пользователя, указанные в pg_hba.conf (см. ниже), затем создаёт процесс для клиента, который выполняет SQL-команды, взаимодействуя с памятью и файлами через внутренние механизмы.

2. Файловая структура PostgreSQL (p3-s14)

Вопрос: *Как организована файловая структура кластера PostgreSQL? Какие основные каталоги и файлы входят в PGDATA?*

Ответ: После инициализации PostgreSQL создаётся **директория кластера данных** (каталог PGDATA), содержащая все файлы базы данных. Основные поддиректории и файлы в ней следующие:

- `global/` – содержит глобальные системные таблицы кластера, общие для всех баз данных. Например, файл с таблицей `pg_database` (список всех БД кластера) хранится здесь. Эти файлы имеют уникальные имена-идентификаторы (OID) объектов.
- `base/` – здесь находятся подкаталоги для **каждой отдельной базы данных** в кластере. Название подкаталога – это OID базы данных (уникальный числовой идентификатор из `pg_database`). Например, обычно присутствуют каталоги:
- `base/1/` – база *postgres* (создаётся по умолчанию);
- `base/13201/` – *template0*;
- `base/13202/` – *template1*.

Внутри каждого такого каталога находятся файлы данных таблиц и индексов, принадлежащих соответствующей базе. Каждый объект (таблица или индекс) представлен одним либо несколькими файлами: - Если размер объекта $\leq$ 1 GB, ему соответствует один файл. - Если объект более 1 GB, он разбивается на несколько файлов по ~1 GB (с именами cuffix ...1, ...2 и т.д.).

Имена файлов – это *OID (идентификатор)* объекта или *relfilenode*. Связь OID с именем таблицы можно определить через системный каталог (`pg_class.relfilenode`) или с помощью утилиты `oid2name`. Например, утилита `oid2name` выведет соответствие OID баз данных их именам:

```
Oid    | Database Name | Tablespace
------+---------------+------------
 13202 | template1     | pg_default
```

```
 13201 | template0     | pg_default
 1     | postgres      | pg_default
```

(где видно, что `1` – это postgres, `13201` – template0, `13202` – template1). - `pg_wal/` – *Write-Ahead Log* (ранее назывался pg_xlog). Здесь хранятся файлы журнала предзаписи (WAL) – последовательность бинарных журналов изменений, используемых для восстановления данных после сбоев. WAL-файлы имеют фиксированный размер (16 MB по умолчанию) и именуются по порядковым номерам (16-ричным). Содержимое и назначение WAL подробнее раскрыто ниже. - `pg_xact/` (ранее pg_clog) – хранит специальные файлы журнала статусов транзакций (коммит/откат), так называемый CLOG (Commit Log). По сути это битовые карты, помечающие каждую транзакцию как зафиксированную или нет, используемые для быстрого определения видимости транзакций. - `pg_multixact/`, `pg_commit_ts/`, `pg_subtrans/` – дополнительные журналы транзакционной системы (для мульти-транзакций, временных меток коммитов, информации о под-транзакциях), детали которых выходят за рамки базового рассмотрения. - `pg_tablespace/` – каталоги табличных пространств (если создавались дополнительные помимо стандартных `pg_default` и `pg_global`). - `pg_twophase/` – файлы для поддержки двухфазных транзакций (PREPARE TRANSACTION). - `PG_VERSION` – текстовый файл с версией PostgreSQL данного кластера (используется для проверки соответствия версий при старте). - **Конфигурационные файлы** (обычно в корне PGDATA): - `postgresql.conf` – основной файл настройки параметров сервера; - `pg_hba.conf` – настройки доступа клиентов (см. следующий вопрос); - `pg_ident.conf` – опциональный файл отображения идентификаторов пользователей (используется для *ident/peer*-аутентификации).

Резюме: Файловая структура PostgreSQL организована по принципу: каталог PGDATA содержит общие данные кластера (global, WAL и пр.) и отдельные подкаталоги для каждой базы данных (base/<OID>). Такая структура позволяет легко управлять множеством баз в одном инстансе и обеспечивает изоляцию хранения данных разных БД, а также ведение журнала изменений для надежности.

3. Template0 и Template1 (p3-s10)

Вопрос: *Что такое базы данных-шаблоны template0 и template1 в PostgreSQL? Для чего они нужны и чем отличаются?*

Ответ: После инициализации кластера (`initdb`) PostgreSQL создает две особые базы данных-шаблоны: **template1** и **template0**. Их назначение: - **template1** – основная база-шаблон, используемая по умолчанию при создании новых баз данных. Когда вы выполняете команду `CREATE DATABASE имя;` без явного указания шаблона, PostgreSQL **клонирует** содержимое template1 в новую базу данных. Template1 содержит минимально необходимую структуру (стандартные системные каталоги, типы, функции и т.д.), и изначально практически пуста (кроме служебных объектов). - **template0** – резервная копия чистой базы. Изначально template0 является **точной копией template1** (сразу после инициализации они идентичны). Template0 предназначена для случаев, когда нужно создать новую базу, **игнорируя изменения**, которые могли быть внесены в template1. Сам PostgreSQL не использует template0 по умолчанию, но пользователь может указать `TEMPLATE template0` при создании базы, чтобы получить **совершенно чистую базу** (например, без локальных расширений).

Отличия и особенности: - Template1 **доступна для подключения** пользователям (обычно только суперпользователь). Администратор может **изменять template1** – например, создавать в ней общие для всех будущих баз функции, расширения, настройки по умолчанию. Все такие

изменения будут автоматически присутствовать во **всех новых БД**, создаваемых на основе template1. *Важно:* Если внести изменения в template1, они не затрагивают уже существующие базы, но новые базы из template1 их унаследуют. - Template0 **не доступна для подключения** (PostgreSQL запрещает подключаться к template0). Эта база всегда остается в изначальном состоянии, её предназначение – служить эталонной "чистой" базой. Параметр `datistemplate` в `pg_database` указывает, что template0 и template1 помечены как шаблоны (их могут использовать роли с CREATEDB для создания новых БД), а `datallowconn` обычно запрещает подключения к template0. - При создании новой базы **можно указать любой шаблон**: `CREATE DATABASE newdb TEMPLATE template0;` – база будет скопирована с template0. По умолчанию используется template1, эквивалентно `CREATE DATABASE newdb TEMPLATE template1;`.

Вывод: template1 и template0 – системные базы-шаблоны. Template1 – основная настраиваемая база-шаблон для создания новых БД, template0 – эталонная неизменная копия (чистая база). Они упрощают процесс создания баз данных, позволяя быстро клонировать существующую структуру без необходимости заново создавать системные объекты.

4. Конфигурация pg_hba.conf (p3-s16)

Вопрос: *Что представляет собой файл pg_hba.conf? Как он используется для управления доступом клиентов и каков его формат?*

Ответ: **pg_hba.conf** (Host-Based Authentication) – это файл конфигурации PostgreSQL, определяющий **правила аутентификации клиентов** при подключении к базе данных. Проще говоря, в pg_hba.conf задаётся *кто, откуда и как* может подключаться. Этот файл создаётся при инициализации кластера (initdb) и по умолчанию располагается в каталоге PGDATA (путь можно изменить параметром `hba_file`).

Формат pg_hba.conf: файл состоит из последовательности записей (строк), каждая из которых определяет разрешённый метод доступа. Строки читаются сервером **последовательно**, сверху вниз, и первая подходящая под параметры соединения строка определяет разрешение или запрет. Если ни одна запись не подошла, доступ отклоняется. Поскольку порядок записей важен, администратору нужно тщательно упорядочивать правила (более специфичные – выше, общие – ниже).

Каждая запись имеет колонки, разделённые пробелами:

```
# Тип подключения  БД        Пользователь    Адрес/подсеть       Метод
[Опции]
```

Например, стандартный pg_hba.conf содержит записи:

```
# "local" is for Unix domain socket connections only
local   all   all               peer

# IPv4 local connections:
host    all   all   127.0.0.1/32   md5
```

Значения полей:

- **Тип подключения:** `local` (соединение через UNIX-сокет на локальной машине, без сети) или `host` (TCP/IP соединение). Для `host` также различаются `hostssl` (только SSL) и `hostnossl` (только без SSL).
- **DATABASE:** к каким базам данная строка применима. Можно указать имя базы, `all` (все БД), `sameuser` (БД с именем, совпадающим с именем пользователя), `samerole` или `replication` (специальный флаг для подключения репликации).
- **USER:** какие роли (пользователи) могут подключаться. Указывают имя роли или `all` (любая роль), либо список через запятую, либо @группа (если настроены группы через pg_ident.conf).
- **ADDRESS:** для `host` – IP-адрес клиента (или диапазон subnet CIDR). Например, `127.0.0.1/32` для localhost IPv4, `::1/128` для IPv6 localhost, или более широкие сети (`192.168.0.0/24`). Можно написать `all` для всех адресов. Для `local` этот столбец не используется.
- **METHOD:** способ аутентификации. Основные методы:
  - `trust` – безпарольный доверенный доступ для указанных адресов/пользователей. **Не рекомендуется** для незнакомых подключений, т.к. разрешает вход без проверки (подходит разве что для локального dev-сервера).
  - `password` – простой пароль (передаётся в открытом виде – не безопасно по нешифрованному соединению).
  - `md5` – требовать пароль (хранится хеш MD5 от пароля); **устаревший** но всё ещё используемый метод хеш-аутентификации.
  - `scram-sha-256` – современный безопасный метод: пароль проверяется через SCRAM (Salted Challenge Response Authentication Mechanism) с SHA-256. Начиная с PostgreSQL 13 по умолчанию используется `scram-sha-256` (если пароль задан). **Примечание:** Чтобы использовать SCRAM, параметр `password_encryption` должен быть `scram-sha-256` и у пользователя должен быть пароль в этом формате.
  - `peer` – для локальных сокетов: **сопоставляет имя системного пользователя ОС с именем роли PostgreSQL**. Работает только для `local` подключения на Unix. Например, если пользователь ОС "postgres" подключается через `psql` без указания user, PostgreSQL принимает его как роль "postgres" без пароля, при настроенном `peer`. Можно настроить сопоставления имен через файл pg_ident.conf (опция `map`).
  - `ident` – похож на peer, но для удалённых подключений по TCP: сервер обращается к **ident-сервису** на клиентской машине, чтобы узнать имя пользователя ОС и сопоставить его ролям PG (требует настройки, редко используется).
  - `reject` – специальный метод, означающий явный запрет подключения, если запись совпала (можно в конце списка правил поставить reject для отсеивания прочих).
  - (Также существуют `gss`, `sspi`, `ldap`, `radius`, `cert` – интеграция с Kerberos/GSSAPI, Active Directory, LDAP и сертификатами SSL, но они применяются в специфичных ситуациях).

Пример процесса проверки доступа: клиент пытается подключиться к БД. PostgreSQL рассматривает pg_hba.conf:
1. Сопоставляет тип соединения: локальное или TCP, ищет подходящие записи.
2. Проверяет имя базы данных и пользователя против записей.
3. Если находит первую подходящую строку, применяет указанный метод. Например, если это `md5`, сервер запрашивает у клиента пароль и проверяет

```
его.
4. Если проверка успешна – соединение разрешается; если провалена – отказ.

**Применение изменений:** файл pg_hba.conf читается **только при запуске сервера**. Если вы изменили pg_hba.conf, чтобы применить новые правила **не нужен рестарт**, достаточно выполнить команду перезагрузки конфигурации (SIGHUP):
- либо вызвать SQL-функцию `SELECT pg_reload_conf();`,
- либо выполнить в консоли `pg_ctl reload` от имени OS-пользователя postgres.

**pg_ident.conf:** упомянутый в pg_hba.conf параметр `map` позволяет сопоставлять системные логины и роли PostgreSQL. Эти соответствия задаются в файле **pg_ident.conf**. Он содержит именованные карты, где прописывается, какой системный пользователь соответствует какой роли PG. Это используется только для методов peer/ident, когда нужно логины несоответствуют 1:1.

**Вывод:** pg_hba.conf – ключевой файл безопасности PostgreSQL. Благодаря ему администратор может гибко ограничивать доступ к БД по адресам, пользователям и способам аутентификации. Типовая настройка: локальные сокеты – `peer` (доверять пользователю ОС postgres), локальные TCP 127.0.0.1 – `md5` (пароль), внешние адреса – только разрешённые сети и строгие методы (scram-sha-256 или cert). Грамотная конфигурация pg_hba.conf – необходимое условие безопасной эксплуатации PostgreSQL.

### 5. Системный каталог и мета-команды psql (p12-s28)

**Вопрос:** *Что такое системный каталог PostgreSQL? Какие существуют способы получения метаданных о структурах базы данных? Расскажите о таблицах каталога, представлениях INFORMATION_SCHEMA и мета-командах psql.*

**Ответ:** **Системный каталог** PostgreSQL – это совокупность специальных служебных таблиц и представлений, хранящих **метаданные** о всех объектах базы данных: таблицах, столбцах, типах, индексах, пользователях, правах и т.д. Проще говоря, PostgreSQL сам хранит информацию о своём содержимом в виде обычных таблиц. Например, информация о всех таблицах находится в `pg_class`, о столбцах – в `pg_attribute`, о базах данных – в `pg_database` и т.п..

Каждая база данных имеет своё собственное множество системных таблиц (в схеме `pg_catalog` этой базы) – они содержат информацию об объектах **именно данной базы**. Наряду с ними есть **кластерные каталоги** – некоторые таблицы существуют единожды на весь кластер (в каталоге `global/`), например `pg_database` (список баз в системе) или `pg_authid` (список ролей/паролей).

**Способы доступа к метаданным:**
1. **Прямой запрос к системным таблицам (pg_*):** вы можете писать SQL-запросы к каталогам. Например:
   ```sql
   SELECT * FROM pg_database;
   SELECT datdba, datname FROM pg_database;
   ```
   или получить число активных сессий:
```

```sql
SELECT count(*) FROM pg_stat_activity;
```
(где `pg_stat_activity` – представление со списком подключений). Прямой запрос позволяет получить самую полную информацию, однако каталоги порой имеют нелёгкую для понимания структуру (связи через OID и т.д.).

2. **Стандартная информационная схема (INFORMATION_SCHEMA):** это набор представлений (views), определённых стандартом SQL, которые отображают содержимое системных каталогов в переносимом виде. Они находятся в схеме `information_schema` (доступной в каждой базе). Примеры:
- `information_schema.tables` – список таблиц (и представлений) в текущей базе данных;
- `information_schema.columns` – список столбцов и их типы;
- и др.

Пример запроса:
```sql
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public';
```
получит имена всех таблиц в публичной схеме.

INFORMATION_SCHEMA облегчает перенос SQL между СУБД и скрывает специфичные детали (например, в PostgreSQL столбцы `attname`, `attnum` из `pg_attribute` отображаются как `column_name`, `ordinal_position` и т.д. по стандарту).

3. **Мета-команды psql:** интерактивный терминал `psql` предоставляет удобные короткие команды (начинающиеся с обратного слеша `\`) для просмотра структуры. Они позволяют не запоминать сложные запросы. Наиболее полезные:
- `\d [имя]` – отображает определение объекта (таблицы, представления, последовательности, индекса, функции...). Например, `\d students` выведет схему таблицы *students* (столбцы, типы, индексы, проверки и пр.).
- `\d` без указания имени – покажет список всех таблиц и последовательностей в текущей базе.
- `\dt` – список таблиц (только таблиц) в текущей схеме; `\dv` – список представлений; `\ds` – последовательностей; `\di` – индексов.
- `\l` – список всех баз данных (отображает имя, владелец, кодировку, колlation, размер и комментарий).
- `\dn` – список схем; `\df` – список функций; `\du` – список ролей (пользователей).
- `\z` – список привилегий (права доступа ко всем таблицам/схемам и т.д.).
- `\?` – покажет справку по всем мета-командам.
- `\x` – переключить расширенный формат вывода (для удобного просмотра широких таблиц).

Мета-команды не требуют прописывать SQL – `psql` сам отправляет соответствующие запросы к системному каталогу и форматирует результат для удобства. Например, `\d students` фактически выбирает данные из `pg_catalog.pg_tables`, `pg_catalog.pg_columns` и других, чтобы отобразить схему таблицы, индексы и пр.

```
Кроме структуры данных, через системные каталоги доступны и **статистические
и служебные сведения**:
- представления `pg_stat_*` (например, `pg_stat_user_tables` – статистика по
таблицам: число читанных/модифицированных строк и т.п.; `pg_stat_activity` –
текущие соединения и их состояние);
- таблицы настройки `pg_settings` (текущие параметры конфигурации и их
значения);
- и др.

**Примеры системных таблиц:**
- `pg_class` – одна из ключевых таблиц каталога: содержит по строке на каждый
объект класса (таблица, индекс, последовательность и пр.) в базе. Важные
поля: `relname` (имя объекта), `relkind` (тип: r=table, i=index, v=view,
S=sequence,...), `relpages`, `reltuples` (оценка размера), `relowner` (OID
владельца-ролии) и др.
- `pg_attribute` – столбцы таблиц: каждая строка – колонка. Поля: `attname`
(имя колонки), `atttypid` (OID типа данных, ссылается на `pg_type`),
`attrelid` (OID таблицы, ссылается на `pg_class`), `attnotnull` (NOT NULL?),
`adsrc` (по умолчанию, если есть) и др..
- `pg_type` – содержит записи обо всех типах данных (встроенных и
пользовательских).
- `pg_proc` – о всех хранимых функциях (процедурах).
- `pg_constraint` – о всех ограничениях (первичные/внешние ключи, проверки).
- `pg_authid` – роли (пользователи и группы), с хешами паролей (доступна
только суперпользователю, так как содержит пароли).
- `pg_roles` – представление, удобный "человеческий" вид объединения
`pg_authid` и `pg_auth_members` (список ролей с указанием прав SUPERUSER,
CREATEDB, и членств).

**OID и символьные типы:** во многих каталогах объекты ссылаются друг на
друга через OID (идентификаторы). Для удобства PostgreSQL вводит *тип-алиас*
например `regclass` – который при выводе отображает OID таблицы как имя
таблицы. Например:
```sql
SELECT 12345::regclass;
```

если 12345 – OID таблицы, вы получите ее имя. Это упрощает запросы к каталогам (можно писать `WHERE attrelid = 'students'::regclass` вместо подзапроса по имени).

Вывод: системный каталог PostgreSQL – это "внутренняя кухня" СУБД, хранящая структуру всех объектов. Доступ к ней возможен напрямую (запросами SQL), через стандартные представления (information_schema) или удобными командами консоли (`\d`, `\dt`, `\l` etc.). Для полного понимания происходящего в базе администратору важно знать о существовании этих каталогов: например, чтобы узнать структуру или статистику, не обязательно иметь внешний инструментарий – всё доступно изнутри самой базы данных.

6. Транзакции и свойства ACID (p4-s2)

Вопрос: *Что такое транзакция в контексте СУБД? Каковы основные свойства транзакций (ACID) и что они означают?*
```

Ответ: Транзакция – это логическая единица работы с базой данных, объединяющая одну или несколько операций (например, несколько SQL-выражений) в одно **атомарное** целое. Транзакция гарантирует, что все ее изменения данных будут зафиксированы *либо* все вместе, *либо* ни одного (при откате). Промежуточные результаты транзакции невидимы другим транзакциям до момента фиксации (COMMIT). Если произошла ошибка и транзакция откатывается, система возвращает базу в исходное состояние, как будто никаких операций этой транзакции не было.

Транзакции обладают набором свойств, известных как **ACID**: - **Atomicity (атомарность):** Транзакция неделима – либо выполняется целиком, либо не выполняется совсем. Никакие частичные изменения не должны сохраняться при неудаче. Например, если транзакция переводит деньги со счёта A на счёт B (две операции UPDATE), то при сбое после первого обновления система отменит первое обновление, чтобы не было “пропажи” или “создания” денег из воздуха. - **Consistency (согласованность целостности):** После завершения транзакции база данных должна оставаться в корректном, согласованном состоянии, удовлетворяющем всем заданным ограничениям целостности (уникальности, внешних ключей и др.). Если исходные данные были согласованы и транзакция корректна с точки зрения бизнес-логики, то после COMMIT все инварианты должны сохраняться. *Примечание:* согласованность относится к целостности данных – СУБД не гарантирует сама по себе логическую правильность операции, это ответственность разработчика. В процессе выполнения (до commit) транзакция может временно нарушать ограничения (например, создать временно дубли или несвязанные данные), но к финалу транзакции все нарушения должны быть устранены, иначе транзакция откатывается. - **Isolation (изолированность):** Одновременное выполнение нескольких транзакций не должно приводить к взаимному влиянию на их результат. Каждая транзакция **видит данные в определённом состоянии**, как если бы другие транзакции выполнялись либо раньше, либо позже нее, но не “впутывались” в середину. В реальных СУБД полная изоляция достигается в строгом уровне Serializable, тогда как на практических уровнях допускается ограниченное влияние (см. уровни изоляции далее). Но в любом случае, эффекты должны быть такими, будто транзакции выполнялись последовательно в некотором порядке, а не вперемежку. - **Durability (надёжность, долговечность):** Результаты успешно завершённой транзакции **гарантированно сохранены** на постоянном носителе и не потеряются даже в случае сбоя системы, отключения питания и т.п.. Это свойство обеспечивается механизмами журнала транзакций (WAL): когда транзакция фиксируется, все её изменения сначала надежно записываются в журнал на диск, и только после этого пользователю сообщается об успешном завершении. Таким образом, даже если сразу после commit сервер упадёт, при следующем запуске он сможет восстановить состояние БД, применив записанные в журнал изменения (см. WAL).

Приведём простой пример транзакции (перевод денег между счетами):

```
BEGIN;
UPDATE accounts SET balance = balance - 1000 WHERE acc_id = 1;
UPDATE accounts SET balance = balance + 1000 WHERE acc_id = 2;
COMMIT;
```

Здесь обе операции обновления баланса будут зафиксированы атомарно – либо обе применятся (снимая 1000 со счета 1 и зачисляя на счет 2), либо ни одна (в случае любой ошибки мы можем выполнить `ROLLBACK;`, и тогда баланс останется как был). Пока транзакция не зафиксирована, никакой другой сеанс не увидит промежуточного состояния (например, списанных денег со счета 1 без зачисления на счет 2).

PostgreSQL поддерживает **неявные** транзакции: если вы не написали `BEGIN`, каждая отдельная SQL-команда выполняется как транзакция сама по себе (авто-commit). Если вы хотите объединить несколько команд – используйте явный `BEGIN ... COMMIT`. Также есть возможность сохранять промежуточные точки отката внутри транзакции – **savepoints** (SAVEPOINT X / ROLLBACK TO X) – для частичного отката.

Таким образом, транзакции и ACID-гарантии защищают целостность данных и делают работу с БД надежной: даже при сбоях или параллельных изменениях от разных пользователей данные не будут потеряны и останутся консистентными.

7. MVCC в PostgreSQL и VACUUM (p4-s8)

Вопрос: *Как PostgreSQL реализует управление параллельностью транзакций? Что такое MVCC? Почему в PostgreSQL необходим периодический VACUUM?*

Ответ: Для обеспечения изолированности транзакций PostgreSQL использует механизм **MVCC (Multi-Version Concurrency Control)** – многоверсионное управление параллелизмом. В PostgreSQL конкретно реализована разновидность MVCC под названием **Serializable Snapshot Isolation (SSI)** для уровня сериализуемости. Суть MVCC: **каждая транзакция оперирует над своей "версией" данных**, а обновления не блокируют чтения и наоборот.

Основные идеи реализации MVCC в PostgreSQL: - **Каждая запись (строка таблицы) хранится с метками версий** – невидимыми системными полями: `xmin`, `xmax` и др. Когда транзакция вставляет новую строку, этой строке присваивается `xmin = <ID этой транзакции>`; когда транзакция удаляет или обновляет строку, она помечается `xmax = <ID транзакции>` (а при обновлении вдобавок создаётся новая версия – новая строка с собственным xmin). - **Не происходит перезаписи "на месте"**: при `UPDATE` PostgreSQL логически выполняет DELETE + INSERT – помечает старую версию как удалённую (проставляет ей xmax) и создаёт новую копию строки с новыми значениями (у новой версии свой xmin). Таким образом, на одну пользовательскую строку в таблице может существовать несколько версий одновременно (напр., старая и новая). - **Изоляция посредством версий:** каждая выполняющаяся транзакция имеет свой "снимок" (snapshot) данных – набор номеров транзакций, изменения которых она должна видеть. Все версии с `xmin` (идентификатором создающей транзакции) больше, чем текущее "видимое" для нас, будут транзакцией игнорироваться. Проще: транзакция видит только те версии строк, которые были созданы до начала её работы (или которые сами транзакции ранее создали), и не видит версий, появившихся после начала. Также транзакция не видит версии строк, помеченные `xmax` активных на момент её снимка транзакций (то есть удалённые "параллельно"). За счёт этого читающая транзакция не блокируется пишущей: пишущая создаёт новую версию, а читающая продолжает видеть старую, соответствующую её моменту начала. - **Блокировки на уровне строк используются только для конфликтующих изменений.** Благодаря MVCC, обычные SELECT не блокируются на изменения, и разные транзакции могут одновременно менять разные строки. Даже при попытке изменить **одну и ту же строку** параллельно действует правило: вторая транзакция увидит, что существует новая версия, и при попытке обновления либо будет ждать завершения первой (если та еще не завершилась), либо обнаружит конфликт (если первая уже зафиксирована, в зависимости от изоляции может произойти ошибка сериализации). Например, две транзакции, обновляющие одну строку, – вторая будет ждать коммита/отката первой и при необходимости откатится (при сериализуемой изоляции).

Пример:
Таблица STUDENTS, изначально:

```
Stud_ID | Name | Group
1       | Ivan | 33313
```

Транзакция 1 выполняет `UPDATE students SET Group = 33314 WHERE Stud_ID = 1;`
После этого в таблице две версии записи ID=1: - старая: Group=33313, помеченная как старая (имеет xmax = TX1) - новая: Group=33314, с xmin = TX1
Другие транзакции, начатые до коммита TX1, будут видеть старую версию (33313). После коммита транзакции 1 новые транзакции будут видеть новую версию (33314), а старая версия считается **“мертвой записью”** («dead tuple»).

Зачем нужен VACUUM: Так как обновления и удаления не физически удаляют строку, а лишь помечают старую версию как удалённую, *в таблице накапливаются невидимые “мёртвые” строки*. Эти неподчищенные версии занимают место и раздувают объем хранения. Команда **VACUUM** сканирует таблицы и **очищает “мёртвые” записи**, которые уже не нужны ни одной транзакции. Это освобождает пространство (оно помечается как пригодное для повторного использования) и предотвращает бесконечный рост файлов базы. Кроме того, VACUUM выполняет важную функцию предотвращения **переполнения идентификаторов транзакций (XID wraparound)**: каждая транзакция имеет 32-битный номер, который со временем может переполниться. VACUUM помечает очень старые “замороженные” версии (FREEZE) – т.е. указывает, что строки настолько старые, что их можно считать всегда видимыми (присваивает специальный xmin = FrozenXID), предотвращая переполнение счетчика.

В PostgreSQL работает служба **autovacuum** – фоновый процесс, который автоматически вызывает VACUUM (и ANALYZE) по таблицам по мере накопления в них достаточного количества изменений. Но также администратор может вручную выполнять `VACUUM [VERBOSE]` для очистки, или `VACUUM ANALYZE` для одновременного сбора статистики.

Итак, **MVCC** обеспечивает параллельность без жестких блокировок на чтение, а **VACUUM** необходим как “уборщик мусора”, очищающий старые версии данных и поддерживающий производительность и здоровье системы.

8. Уровни изоляции транзакций в PostgreSQL (p4-s13)

Вопрос: *Какие уровни изоляции транзакций определены стандартом SQL и какие из них поддерживаются в PostgreSQL? Какие феномены (аномалии) параллельности предотвращает каждый уровень? Какой уровень используется в PostgreSQL по умолчанию?*

Ответ: Стандарт SQL определяет **4 уровня изоляции** транзакций, характеризующихся тем, какие нежелательные эффекты (аномалии) параллельного доступа они предотвращают: 1. **Read Uncommitted** (*чтение незавершённых данных*) – самый слабый уровень. Допускает все типы аномалий, вплоть до чтения “грязных” данных (из несостоявшихся транзакций). По сути, это отсутствие изоляции. 2. **Read Committed** (*чтение фиксаций*) – предотвращает «грязное чтение», т.е. не позволяет транзакции видеть незавершённые изменения другой транзакции. Однако другие эффекты возможны: повторное чтение может дать разные результаты (nonrepeatable read), могут появиться “фантомы” – новые строки, удовлетворяющие тому же условию, вставленные параллельно. 3. **Repeatable Read** (*повторяемое чтение*) – предотвращает грязное и

неповторяющееся чтение. Транзакция на уровне Repeatable Read гарантированно будет видеть *одинаковые результаты* для одного и того же запроса, выполненного дважды в ее рамках (никакие другие подтверждённые транзакции не влияют на её ранее прочитанные данные). Однако стандарт допускает на этом уровне феномен «фантомного чтения» – когда при повторном запросе в рамках той же транзакции появляются новые строки, удовлетворяющие условию (потому что другой транзакцией они были вставлены и зафиксированы). 4. **Serializable** (*полная сериализуемость*) – самый строгий уровень. Полностью исключает все предыдущие аномалии, включая фантомы, обеспечивая такое поведение, как если бы транзакции выполнялись строго последовательно одна за другой. Любые ситуации, которые могли бы нарушить сериализуемость, приводят к откату (ошибке сериализации) одной из конфликтующих транзакций.

Поддержка в PostgreSQL: - **Read Uncommitted** на практике **не поддерживается** в PostgreSQL, и если выбран, то фактически работает как **Read Committed** (PostgreSQL трактует RU как RC, поскольку грязное чтение у него не допускается ни при каких настройках). - **Read Committed** – **уровень по умолчанию** в PostgreSQL. Каждая отдельная SQL-команда внутри транзакции видит только данные, подтверждённые до начала этой команды. После выполнения любой инструкции транзакция может видеть изменения, которые другие транзакции зафиксировали к тому моменту. То есть snapshot берётся заново каждую команду. Этот уровень предотвращает грязное чтение: никакие незафиксированные данные не видны. Однако неповторяющиеся и фантомные чтения возможны, т.к. между двумя SELECT в одной транзакции другая транзакция может внести изменения, которые первая “заметит” во второй SELECT. - **Repeatable Read** – вплоть до версии PostgreSQL 9.0 соответствовал эффекту, что транзакция получает **единый snapshot на всё время** выполнения (как Serializable snapshot isolation). Начиная с PostgreSQL 9.1, уровень Repeatable Read по сути обеспечивает **Snapshot Isolation**. Он предотвращает грязное и неповторяющееся чтение, а также фантомы в том классическом смысле, что после взятия снимка транзакция не увидит *новых строк*, добавленных другими (потому что работает со snapshot). Однако существует тонкая аномалия “write skew” (ошибка при одновременной записи связанных данных), которая не ловится на уровне Repeatable Read. Таким образом, PostgreSQL Repeatable Read **очень близок к сериализуемому**, и для большинства приложений его достаточно как эквивалент Serializable (без накладных расходов SSI). Тем не менее, формально фантомное чтение в терминах стандарта у PostgreSQL Repeatable Read отсутствует, так как snapshot фиксирован. - **Serializable** – PostgreSQL реализует его через **SSI (Serializable Snapshot Isolation)**. Это значит, что транзакции также работают с снимками (как в Repeatable Read), но система дополнительно отслеживает конфликтующие операции (например, ситуация write skew, когда две транзакции читают набор данных, а потом каждая меняет разные части, нарушая целостность в совокупности). Если такая конфликтная ситуация обнаруживается, PostgreSQL **прерывает** одну из транзакций с ошибкой *serialization failure* («could not serialize access due to read/write dependencies among transactions»). Разработчик может повторно выполнить транзакцию. Таким образом достигается полный эффект сериализуемости – гарантируется эквивалентность некоторому последовательному выполнению.

Феномены параллельности и их предотвращение: - *Dirty read (грязное чтение)* – чтение несохранённых данных другой транзакции. **Не допускается** уже на уровне Read Committed (и выше). - *Non-repeatable read (неповторяющееся чтение)* – ситуация, когда транзакция дважды читает одну и ту же строку и получает разные результаты (потому что между чтениями другая транзакция изменила и зафиксировала эту строку). **Предотвращается** начиная с Repeatable Read (т.к. там второй SELECT увидит старое значение из своего снимка). - *Phantom read (фантомное чтение)* – ситуация, когда повторный запрос, выбирающий множество строк по условию, возвращает другой набор строк, чем раньше (например, другая транзакция добавила новые строки, удовлетворяющие условию). **Предотвращается** на уровне Serializable. PostgreSQL’s Repeatable Read также не покажет новые строки, добавленные после начала транзакции (они не

входят в её snapshot), таким образом класические фантомы не появляются. Однако возможны более сложные эффекты, которые решаются только сериализацией (напр., обе транзакции вставляют по строке, каждая не видит “фантом” другой, но вместе нарушают ограничение – этот случай SSI тоже ловит).

Итого, в PostgreSQL доступны три уровня: Read Committed (по умолчанию), Repeatable Read и Serializable. Для установки уровня используют команды:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
-- или для текущей транзакции:
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Поменять уровень **после начала** транзакции нельзя – только до или при начале.

В повседневной работе чаще всего используется **Read Committed**, который обеспечивает достаточную изоляцию при минимуме блокировок. Когда требуется абсолютная последовательность – включают Serializable (с обработкой возможных откатов). Уровень Repeatable Read может использоваться для долгих отчетных транзакций, которым важно консистентное “снимковое” состояние данных.

9. WAL, журналы REDO и UNDO (p6-s3)

Вопрос: *Что такое WAL (Write-Ahead Log) в PostgreSQL? Как реализован журнал транзакций и восстановление данных? В чем разница между подходами UNDO/REDO и какой из них применяется в PostgreSQL?*

Ответ: WAL (Write-Ahead Log) – это журнал предзаписи в PostgreSQL, фиксирующий все изменения данных перед тем, как они применяются к основным файлам базы. Принцип *write-ahead* означает: *сначала в журнал, потом в данные*. Это ключевой механизм, гарантирующий свойство долговечности (Durability) транзакций.

Запись изменений (REDO логика): PostgreSQL использует **стратегию REDO-журнала**. Это значит, что в журнал **записываются действия, которые нужно выполнить повторно при восстановлении**. Основные правила WAL (правило предварительной записи): - Любое изменение данных (например, изменение страницы таблицы) *до* записи на диск должно быть сначала записано в WAL на диск. - Запись о **коммите транзакции** также пишется в WAL *до* того, как подтвердить пользователю успешное завершение. Только после этого изменения из буферов могут сбрасываться в файлы данных. - Таким образом, последовательность: при `COMMIT` PostgreSQL **форсирует flush WAL на диск**, включая все записи о модификациях, а уже потом позволяет завершить транзакцию и асинхронно сбросить грязные страницы на диск.

Благодаря этому, если сервер упадёт, то при следующем запуске он будет просматривать WAL и **повторит (redo) все изменения**, для которых не успели записаться основные данные. WAL хранится на диске в директории `pg_wal/` сегментами по 16MB (размер можно изменить при инициализации, параметр `wal_segment_size`). WAL общий на весь кластер (для всех баз данных инстанса).

Структура WAL-записи: содержит заголовок (идентификатор транзакции XID, длина, контрольная сумма, ссылки и пр.) и полезные данные – описание изменения. Каждая WAL-запись

идентифицируется уникальным номером **LSN (Log Sequence Number)** – по сути это адрес в журнале. LSN позволяет отслеживать до какого момента данные применены: каждый блок данных в таблицах хранит метку LSN последнего изменения, и при восстановлении система сравнивает LSN блока и журнала, чтобы понять, нужно ли применять журнал к этому блоку.

UNDO vs REDO: В теории журналирования транзакций бывают две стратегии: - **UNDO/REDO (двухфазный журнал)**: СУБД хранит как действия для повторного выполнения (redo), так и противодействия на случай отката (undo). Обычно требуется два типа записей: before-image (исходные данные до изменения) для возможности отката, и after-image (новые данные) для повтора. Пример такой СУБД – Oracle (rollback segments) или MS SQL Server (transaction log). - **Только REDO (протокол WAL как в PostgreSQL):** делается упор на сохранение только необходимых для повтора операций данных. Для отката же PostgreSQL не использует отдельный UNDO-журнал; вместо этого применяется механика MVCC – "старые версии" данных хранятся прямо в таблице, и откат транзакции просто помечает ненужные версии как невидимые (см. MVCC выше). То есть фактически, роль *Undo* частично выполняют старые строчки + механизм CLOG, который фиксирует факт отмены транзакции.

В PostgreSQL WAL содержит **в основном REDO-информацию**: новые данные или изменения, которые нужно применить заново. При откате транзакции в WAL пишется только запись о rollback (Abort) транзакции, чтобы при восстановлении знать, что изменения этой транзакции не должны применяться. Сами "undo" операции не сохраняются – отмена транзакции происходит за счет того, что ее незавершенные изменения просто игнорируются (бросаются) и помечаются невидимыми.

Восстановление после сбоя: При старте после аварии PostgreSQL запускает **процесс восстановления (crash recovery)**: - Определяется граница **последней успешной контрольной точки (checkpoint)** – в WAL периодически записываются спец. записи о чекпоинтах, содержащие информацию о том, какие транзакции завершены к тому моменту и что сброшено на диск. - Далее WAL просматривается с этой точки (REDO point) до конца журнала. При восстановлении отделяют две группы транзакций: - Те, у которых есть запись `COMMIT` в журнале – их изменения **нужно повторить** (redo), т.к. они могли не попасть в файлы данных. - Те, у которых **нет** `COMMIT` (транзакции прервались или откатились) – их изменения **нужно проигнорировать** (по сути "undo", но поскольку они не записаны, достаточно не применять их). - Проигрывание WAL производится последовательно, применяя after-image ко всем страницам (при необходимости) до самого конца журнала. Таким образом, подтвержденные транзакции повторяются, неподтвержденные – отбрасываются. После этого БД возвращается к консистентному состоянию.

Контрольные точки (checkpoint): Чтобы ускорить восстановление и не хранить бесконечно длинный журнал, PostgreSQL периодически делает checkpoint: сбрасывает на диск все изменённые страницы (чтобы данные синхронизировались с журналом) и записывает в WAL специальную метку. Параметры `max_wal_size` или `checkpoint_timeout` регулируют частоту checkpoints (по умолчанию каждые 5 минут или при накоплении 1 GB WAL). При восстановлении можно начинать redo с последней контрольной точки, не читая более старый WAL.

Какие операции не пишутся в WAL: изменения во **временных таблицах** (они видны только текущему сеансу и при сбое все равно бы потерялись) не журналируются, также как операции с **UNLOGGED таблицами** (специальный тип таблиц, чьи изменения не входят в WAL). Это означает, что в случае краха содержимое unlogged-таблиц потеряется или станет недостоверным – при старте PostgreSQL помечает их как требующие сброса. До PostgreSQL 10 также **хеш-индексы** не логировались (по этой причине при сбое или репликации они могли рушиться; начиная с v10 хеш-индексы тоже WAL-логируются).

Резюме: WAL – это последовательный лог изменений, позволяющий восстановить транзакции после сбоев. PostgreSQL использует только редо-журналирование: *“сначала запись – потом действие”*. Подход MVCC избавляет от необходимости хранить подробный undo-журнал – отмена изменений достигается хранением предыдущих версий строк до VACUUM. WAL является также основой для механизмов **репликации и архивирования** (см. далее), так как он содержит полную историю изменений базы данных.

10. Резервное копирование: виды и стратегии (p7-s2)

Вопрос: *Какие существуют виды резервного копирования (backup) баз данных? Классифицируйте виды бэкапов по различным признакам. Какие варианты поддерживаются в PostgreSQL?*

Ответ: **Резервное копирование** – создание копии данных, предназначенной для восстановления базы данных в случае потери или повреждения основной копии. Существует несколько классификаций видов резервного копирования:

1. **По полноте охвата данных:**
2. **Полное резервное копирование** – копируется *вся* база данных (либо весь кластер, либо указанные базы). Это наиболее надежный вариант восстановления *с нуля*, но и самый объёмный по данным.

3. **Частичное резервное копирование** – копируется *часть* базы данных. Например, только определенные таблицы, схемы или только одна база из кластера. Применяется, когда полная копия слишком велика или нужна только отдельная часть данных.

4. **По характеру данных в копии:**

5. **Физическая копия** – представляет собой копирование на уровне файлов системы (бинарных файлов БД). Получается побайтно идентичная копия файлов PostgreSQL. Такая копия зависит от версии СУБД, ОС и архитектуры, но восстанавливается быстрее (просто заменой файлов).

6. **Логическая копия** – представляет данные в виде экспорта объектов БД (таблиц, схем, данных) в некий переносимый формат (например, SQL-скрипт с командами `CREATE TABLE` и вставками `INSERT`). Логическая копия независима от конкретного представления на диске и может быть восстановлена на другой версии СУБД, другой платформе, но восстановление требует выполнения всех SQL-операций заново.

7. **По состоянию работы СУБД во время бэкапа:**

8. **“Холодное” резервное копирование** – выполняется при остановленном сервере БД. Гарантируется консистентность файлов, но требует простоя системы на время копирования.

9. **“Горячее” резервное копирование** – выполняется на работающей базе без прерывания обслуживания пользователей. Пример – логическое копирование утилитой `pg_dump` или физическое копирование через `pg_basebackup`/снимок файловой системы. Горячее копирование сложнее, т.к. нужно обеспечить согласованность — PostgreSQL решает это либо посредством MVCC (логический бэкап читает установленный snapshot без блокировок длительных), либо с помощью механизма WAL (физический бэкап дополняется

журналами, позволяющими довести файлы до согласованного состояния при восстановлении).

10. **По характеру накопления изменений (актуально в системах с поддержкой incremental backup):**

11. **Полный бэкап (уровень 0)** – копия всех выбранных данных целиком.

12. **Инкрементный бэкап** – копия *только изменений* с момента последнего бэкапа. Может быть:

    - **дифференциальным** – содержит изменения с момента последнего *полного* бэкапа (уровня 0). Например, “разница” от последней полной копии.
    - **кумулятивным** (нарастающим) – содержит изменения с момента последнего бэкапа *предыдущего уровня* (т.е. цепочка инкрементов). Восстановление может потребовать применения нескольких последовательных инкрементных копий. *Примечание:* PostgreSQL из коробки не предоставляет простых средств для инкрементного бэкапа данных (кроме как посредством WAL-архивирования, см. непрерывное архивирование). Однако сторонние инструменты (pgBackRest, Barman) реализуют инкрементные физические копии, отслеживая изменённые блоки.

13. **Специальные виды:**

14. **Резервное копирование настроек** – обычно просто копирование конфигурационных файлов (`postgresql.conf`, `pg_hba.conf`, etc.), т.к. они не входят в данные БД.
15. **Бэкап только структуры / только данных** – разновидность логического: можно экспортировать только схему БД (DDL без данных) или только данные (DML). Например, `pg_dump -s` (schema-only) или `pg_dump -a` (data-only).
16. **Снимок (snapshot) файловой системы** – некоторые СУБД/системы хранения поддерживают мгновенный снимок дискового тома. PostgreSQL можно бэкапить “физически” сделав snapshot каталога данных – такой бэкап считается *логически горячим, но физически мгновенным*. Однако при восстановлении обязательно нужны WAL за период snapshot-а, чтобы привести БД к согласованности.

Поддержка в PostgreSQL: - **Логическое резервное копирование** – стандартные утилиты `pg_dump` (копия одной базы) и `pg_dumpall` (копия всего кластера сразу). Позволяют создавать дамп в форматах SQL (текстовый) или custom (сжатый бинарный формат), а также directory/tar формат (для параллельного выгрузки) – см. подробности в следующем вопросе. - **Физическое резервное копирование** – утилита `pg_basebackup`, а также возможность вручную копировать файлы кластера (с вынужденным остановом сервера или в сочетании с `pg_start_backup` / `pg_stop_backup` для горячего режима). Физический бэкап всегда полный (копия всего PGDATA). PostgreSQL не умеет выбирать отдельные таблицы на файловом уровне. - **Непрерывное архивирование WAL (continuous archiving)** – особый режим, позволяющий достраивать полный бэкап до любого момента времени путем хранения последовательности WAL-журналов. По сути, это *инкрементальное* дополнение к физическому бэкапу: делается базовый полный бэкап, а затем сохраняются все WAL-сегменты, появившиеся после него. В случае восстановления, сначала восстанавливается базовый бэкап, затем проигрываются все журналы из архива – и база возвращается в состояние вплоть до последней транзакции в журнале. Этот механизм позволяет организация **Point-In-Time Recovery (PITR)** – восстановления на конкретный момент времени, например, на момент перед ошибочным удалением данных.

Подытоживая, администратор PostgreSQL может выбирать между логическими бэкапами (гибкость, переносимость, можно брать отдельные объекты) и физическими (скорость восстановления, возможность делать PITR). В идеале в стратегии резервирования сочетают: периодически делают полный физический бэкап + постоянно архивируют WAL (для PITR), или регулярно делают логические дампы критичных частей как дополнительную страховку.

11. Логическое резервное копирование: pg_dump, pg_restore (p7-s8)

Вопрос: *Как выполнить логическое резервное копирование базы данных PostgreSQL? Какие инструменты используются и в каких форматах могут создаваться дампы? Как восстановить базу из логического бэкапа? Приведите примеры команд.*

Ответ: В PostgreSQL для логического резервного копирования предусмотрены утилиты `pg_dump` (для копирования одной базы данных) и `pg_dumpall` (для копирования всего инстанса, всех баз сразу). Также утилита `pg_restore` используется для восстановления из сжатых/специальных форматов бэкапов.

pg_dump: утилита создаёт бэкап указанной базы данных (по умолчанию – подключается к локальному серверу). Основной синтаксис:

```
pg_dump [опции] имя_базы > файл_дампа.sql
```

Без опций `pg_dump` выведет SQL-скрипт, содержащий команды создания всех объектов и вставки данных, соответствующие состоянию базы на момент запуска утилиты. Например:

```
pg_dump mydb > mydb_dump.sql
```

создаст текстовый файл с дампом базы *mydb*. Можно указать пользователя, хост, порт, как при обычном подключении:

```
pg_dump -h localhost -p 5432 -U postgres mydb > backup.sql
```

Либо задавать опции через параметры: - `-f filename` – вывести дамп в указанный файл (иначе печатает в stdout). - `-U username` – подключиться под указанной ролью. - `-W` – потребовать ввод пароля (если нужен). - `-h host` / `-p port` – хост и порт (при бэкапе удаленного сервера).

Форматы вывода pg_dump: задаются опцией `-F` (format): - **plain (по умолчанию)** – простой текст SQL (тот же, что через перенаправление в файл). Преимущество: человеческо-читаем, можно открыть/отредактировать, восстановление – просто выполнить SQL-скрипт через `psql`. Недостаток: обычно большой по размеру, восстановление может быть медленнее (выполняются все INSERTы последовательно). Формат plain можно сжимать внешними утилитами (gzip). - **custom (опция `-Fc`)** – собственный двоичный формат PostgreSQL. Обычно сжимается (если компиляция с zlib). Этот формат несовместим с `psql` напрямую – **нужен pg_restore**. Преимущество: поддерживает параллельное восстановление, можно выбирать отдельные объекты для восстановления, содержит индекс оглавления. Хорош для средних и больших баз. - **directory (опция `-Fd`)** – вывод в виде каталога: создаётся папка, внутри много файлов (каждый объект – отдельный файл) + файл `toc.dat` (оглавление). Тоже требует `pg_restore` для восстановления. Удобен для параллельного создания и восстановления (pg_dump / pg_restore

могут работать в несколько потоков с этим форматом), а также позволяет легко вручную разделять/просматривать дамп по частям. Можно сжать каждую часть раздельно. - **tar (опция `-Ft`)** – дамп в tar-архив, содержащий примерно то же, что directory-format (набор файлов), но объединённых в один tar. Удобно для переноски, также требует `pg_restore` для работы.

Примеры команд с форматами:

```
pg_dump -F c -f mydb.dump mydb          # custom format (.dump file)
pg_dump -F d -j 4 -f dump_dir mydb      # directory format, используя 4 потока
pg_dump -F t -f mydb.tar mydb           # tar format
```

(*Примечание:* В Windows консоли `-Fc` нужно без пробела: `-Fc`.)

Содержимое дампа: логический дамп включает DDL (`CREATE TABLE/VIEW/...`) и данные. По умолчанию данные сохраняются командой COPY (для эффективности). Можно переключить на генерацию INSERT:

```
pg_dump --inserts mydb > data.sql         # каждую строку как INSERT
pg_dump --column-inserts mydb > data.sql # INSERT с указанием списка колонок
```

Но это увеличивает размер и снижает скорость восстановления.

Что pg_dump не копирует: По умолчанию *не* включается команда `CREATE DATABASE` в дамп – считается, что вы создадите БД вручную или укажете другое место восстановления. Если нужно, чтобы дамп сам создавал базу при восстановлении, используют опцию `--create` (или `-C`). Эта опция добавит в дамп:

```
CREATE DATABASE mydb;
\connect mydb;
```

в начале файла, чтобы при `psql` восстановлении сразу создать нужную БД и подключиться к ней.

Дамп отдельных объектов: `pg_dump` позволяет гибко выбрать, что именно бэкапить: - `-t <table>` – бэкап только указанной таблицы (или шаблону имени). - `-T <table>` – исключить указанную таблицу из бэкапа. - `-n <schema>` – бэкап только данной схемы. - `-a` / `--data-only` – **только данные** (без схемы). - `-s` / `--schema-only` – **только структура** (без строк таблиц). - `-N <schema>` – исключить схему. - `-E encoding` – указать encoding (по умолчанию UTF8). - `--if-exists` – добавлять `DROP ... IF EXISTS` перед созданием объектов (для удобства перезаписи при восстановлении). - `-v` – verbose, показывать ход работы (полезно, если дамп долгий). - `-j N` – выполнять дамп в N потоков (только для directory формата).

pg_dumpall: это утилита, которая **обходит все базы** и делает их дамп последовательно в один поток вывода. Она полезна для бэкапа *всего* сервера сразу, включая роли и конфигурацию:

```
pg_dumpall -U postgres > all.sql
```

Этот скрипт будет содержать `CREATE DATABASE ...` для каждой базы, все роли (`CREATE ROLE`), привилегии, и потом данные каждой базы. *Примечание:* pg_dumpall может делать только текстовый формат (plain). Если нужен custom/tar – придется бэкапить каждую базу `pg_dump`-ом отдельно.

Восстановление из логического бэкапа: - **Если дамп в plain SQL:** просто выполняется через `psql`. Например:

```
createdb newdb       # сперва создать пустую базу, если дамп без -C
psql -d newdb -f mydb_dump.sql
```

Либо напрямую: `psql mydb < mydb_dump.sql`. Можно импортировать и частями (например, только нужные команды). - **Если дамп в custom/directory/tar:** необходимо использовать **pg_restore**. Эта утилита умеет читать бинарный формат и воссоздавать объекты. Её плюсы: можно выбирать что восстанавливать (по объектам), выполнять параллельно и т.д. Пример:

```
createdb newdb
pg_restore -d newdb mydb.dump        # восстановить в базу newdb из дампа
```

Основные опции pg_restore: - `-d <dbname>` – куда восстановить (можно указать соединение как у psql: -h, -U и пр.). - `-C` – создать базу перед восстановлением, если дамп содержит `CREATE DATABASE` (pg_restore для custom формата сам может создать). - `-j N` – параллельное восстановление в N потоков (для directory/custom). - `-l` – вывести список содержимого дампа (показывает объекты и их порядковый номер). - `-L <listfile>` – восстановить по списку (можно взять вывод -l, отредактировать и указать -L, чтобы восстановить не всё). - `-t <table>`, `-n <schema>` – фильтр, восстановить только указанную таблицу/схему из дампа. - `-x` – не восстанавливать привилегии (если не нужно права переносить). - `--data-only`, `--schema-only` – аналогично, можно выбрать только данные или схему из полного дампа.

Пример восстановления custom-файла:

```
pg_restore -C -d postgres mydb.dump
```

Это прочитает `mydb.dump`, *создаст базу* (имя возьмет из дампа) и наполнит её. Флаг `-C` работает только если pg_dump был с `--create` или pg_dumpall – иначе в дампе нет `CREATE DATABASE` и база должна быть создана вручную до восстановления.

Соответствие версий: Обычно, версия pg_dump должна соответствовать версии сервера (или быть новее), иначе возможны несовместимости в дампе. Рекомендуется делать бэкап `pg_dump`ом версии сервера или выше (pg_dump понимает старые сервера). Восстанавливать лучше на такой же версии PostgreSQL или обновлять дамп (есть утилиты для апгрейда, но это отдельный вопрос).

Итак, **алгоритм логического бэкапа:** 1. Выполнить `pg_dump` с нужными опциями, получить файл дампа. 2. Для восстановления – либо прогнать SQL через psql (если plain), либо использовать pg_restore (если non-plain формат).

12. Физическое резервное копирование: pg_basebackup (p7-s15)

Вопрос: *Как выполняется физическое резервное копирование PostgreSQL? Какие средства предоставляет СУБД для копирования файлов базы данных и в чем их особенности?*

Ответ: Физическое резервное копирование подразумевает копирование файлов базы данных на уровне файловой системы. Цель – получить идентичную копию каталога данных PGDATA, пригодную для последующего запуска сервера. В PostgreSQL есть несколько способов сделать физический бэкап:

- **Холодный бэкап вручную:** остановить сервер (`pg_ctl stop` или через сервис), после чего скопировать целиком каталог PGDATA (и все файлы табличных пространств, если использовались) обычными системными средствами (cp, tar, rsync). Плюс – простота, гарантия целостности (сервер не работает, файлы на диске согласованы). Минус – требует downtime на время копирования.

- **Горячий бэкап при помощи snapshot ОС:** если хранилище/ФС поддерживает создание моментального снимка (например, LVM snapshots, файловые системы типа ZFS, или снапшоты дисков в виртуализации), можно сделать снимок тома с данными PostgreSQL без остановки сервера, а затем снять копию с него. Однако, такой способ требует, чтобы snapshot был *point-in-time* консистентным: обычно сочетание с архивированием WAL для последующей докрутки журнала при восстановлении (т.к. на момент снимка некоторые транзакции могли быть не завершены). PostgreSQL сам не предоставляет средств для управления снапшотами, это внешний подход.

- **Горячий бэкап через** `pg_basebackup`**:** это утилита, входящая в PostgreSQL, которая автоматизирует процесс получения файловой копии с работающего сервера. Она подключается к серверу как реплика (с использованием репликационного протокола) и выгружает данные кластера. Пример запуска:

  ```
  pg_basebackup -h <host> -p <port> -U <repl_user> -D /path/to/backup_dir
  -F tar -X stream -P
  ```

  Здесь:

- `-D` указывает директорию назначения для копии (может быть пустой каталог или опция -F для формата).
- `-F` формат: `p` (plain directory) или `t` (tar архив).
- `-X` метод включения WAL в бэкап: `fetch` (по завершении скопировать оставшиеся WAL), `stream` (стриминг WAL параллельно копированию). `-X stream` наиболее часто – позволяет сразу получить нужные WAL.
- `-P` показывает прогресс копирования.

Пример:

```
pg_basebackup -D /backups/pgbase -F tar -U backup_user -v -P
```

Скопирует весь PGDATA в тар-архив `/backups/pgbase/base.tar` (и pg_wal тоже, либо xlog.tar, или отдельные WAL-файлы в WAL архив, в зависимости от опций).

`pg_basebackup` требует наличия роли с правами REPLICATION и доступа по pg_hba. Он может работать как локально, так и по сети на удаленном сервере. Утилита обеспечит согласованность: на время бэкапа инициирует создание специальной метки (backup label) и контрольной точки, и захватит все необходимые WAL (если указан `-X`).

В результате выполнения pg_basebackup, помимо самих данных, в pg_wal создается файл `backup_label` (в tar архиве или директории), содержащий информацию о бэкапе (метка, LSN начала/конца). При восстановлении этот файл сигнализирует, что база была получена путем бэкапа и может потребовать проиграть WAL.

- **Горячий бэкап вручную через** `pg_start_backup/pg_stop_backup`**:** В старых версиях PostgreSQL (до 9.6) использовался такой подход:
- Выполняется `SELECT pg_start_backup('метка');` – это создаёт контрольную точку и “метку бэкапа”. После этого сервер продолжает работать, но фиксирует границу LSN, с которой WAL нужно сохранить.
- Администратор копирует файлы PGDATA любым способом (cp, tar и т.д.) пока сервер работает. Поскольку изменения продолжаются, они журналируются в WAL.
- Выполняется `SELECT pg_stop_backup();` – сервер пометит конец бэкапа, возможно, выдаст название архивного WAL.
- Собираются также все WAL-сегменты с LSN pg_start_backup до pg_stop_backup (если архивирование настроено – убедиться что все необходимые WAL ушли в архив).

Этот метод сложнее и сейчас `pg_basebackup` фактически делает то же за вас. В новых версиях (начиная с 9.6, особенно с v13), вместо `pg_start_backup/pg_stop_backup` для неархивируемого бэкапа используют *репликационный* вариант или `pg_basebackup`. В v14+ и далее ручной способ упрощён с появлением `pg_backup_start()` / `pg_backup_stop()`.

Особенности физического бэкапа: - **Консистентность данных:** Файловая копия может получиться не полностью согласованной (если делать без остановки) – например, разные файлы таблиц скопированы с разницей во времени. Но за счёт **WAL** при восстановлении PostgreSQL способен довести базу до консистентного состояния (проиграв или отменив транзакции). Поэтому крайне важно сохранить нужные WAL-сегменты вместе с бэкапом. `pg_basebackup` при `-X stream` включит WAL автоматически. - **Размер:** физический бэкап = размер всех данных. Он, как правило, больше, чем логический дамп (который текст, без пустого места). Зато физический бэкап восстанавливается простым копированием – очень быстро для больших баз. - **Совместимость:** физическую копию обычно нельзя “применить” на другую версию PG или платформу. Она должна восстанавливаться на той же версии (или очень близкой, pg_upgrade might reuse it, но это отдельное). Логические бэкапы более переносимы.

Восстановление физического бэкапа: - Если это просто копия PGDATA, сделанная при выключенном сервере, восстановление тривиально: убедиться, что в `postgresql.conf` указаны правильные пути (если использовались tablespaces – создать нужные каталоги), и запустить сервер на этом каталоге. Он стартует как обычно. - Если это бэкап с WAL (для PITR или после pg_basebackup), нужно: 1. Остановить сервер (если вдруг запущен). 2. Распаковать/

скопировать файлы бэкапа на место PGDATA. 3. Удалить из PGDATA старые WAL (директорию pg_wal очистить или заменить на ту, что шла с бэкапом). 4. Поместить сохранённые WAL-сегменты (например, архив) в отдельную папку или в `pg_wal` (если решено вручную). 5. Настроить recovery: в современных версиях создать **файл-флажок** `recovery.signal` в PGDATA. А в `postgresql.conf` прописать параметры: - `restore_command = 'cp /archive_path/%f "%p"'` – как брать сегменты из архива (например, копировать из каталога архива в pg_wal). Эта команда будет вызываться при старте для каждого WAL-сегмента; она должна вернуть 0 при успехе или ненулевой код при отсутствии файла (что сигнализирует, что архив кончился). - Опционально `recovery_target_time` или `recovery_target_xid` или `recovery_target_lsn`, если хотим PITR не до конца WAL, а до конкретного момента (например, `recovery_target_time = '2025-06-01 10:00:00'`). - Рекомендуется *на время восстановления* запретить внешний доступ к базе: можно в `pg_hba.conf` временно прописать `reject` для всех, или установить параметр `pause_at_recovery_target` и т.д. – чтобы случайно не подключились. (В older versions, use `recovery.conf` with similar settings). 6. Стартовать сервер. Он увидит `recovery.signal` и войдёт в режим восстановления (standby-mode). Будет читать сегменты WAL (через restore_command) и применять их, пока не кончатся или не достигнут целевого момента. 7. По завершении восстановления (достижении end-of-WAL или заданного recovery target) сервер: - переименует `recovery.signal` (или удалит) и продолжит работу в обычном режиме (как основная база). В логах будет запись “database system is ready” и отметка о том, до какого LSN восстановлено. - Если использовался PITR с конкретным target, то неприменённые WAL после цели будут игнорированы. - В PostgreSQL до 12 для запуска recovery использовался отдельный файл `recovery.conf`. Сейчас всё слилось в postgresql.conf + флаг-сигнал.

- Если настроена **синхронная реплика** (standby), то после `pg_basebackup` вместо `recovery.signal` создаётся `standby.signal` и прописывается `primary_conninfo` – но это уже про репликацию.

Итого: Физический бэкап – это быстрый способ склонировать базу на бинарном уровне. PostgreSQL предлагает удобную утилиту `pg_basebackup` для этих целей, а для поддержки актуальности бэкапа – механизм архивирования WAL. Обычно в продакшене применяют комбинацию: периодически `pg_basebackup` (например, раз в сутки) + постоянно архивировать WAL (continuous archiving). Это даёт возможность восстановить базу на любой момент и минимизировать потерю данных.

13. Непрерывное архивирование и Point-In-Time Recovery (PITR) (p7-s13)

Вопрос: *Что такое непрерывное архивирование WAL в PostgreSQL? Как настроить архивирование WAL-журналов и выполнить восстановление базы данных до определённого момента времени (PITR)? Опишите необходимые параметры конфигурации и шаги процесса.*

Ответ: Непрерывное архивирование (continuous archiving) – это режим работы PostgreSQL, при котором каждый завершённый сегмент WAL (журнала транзакций) автоматически сохраняется в стороннем хранилище (каталог, удалённый сервер, облако и т.д.). Цель – накопить последовательность WAL, позволяющую восстановить базу данных *после* последнего имеющегося бэкапа вплоть до момента, когда произошёл сбой или другая проблема. Сочетание **полного физического бэкапа** и **архива всех WAL после него** образует непрерывную цепочку изменений, из которой можно восстановить состояние базы на любой момент между бэкапом и текущим временем.

Настройка архивирования WAL включает следующие шаги:

1. **Включить режим архивации:** В конфигурационном файле `postgresql.conf` задать:

   ```
   wal_level = replica
   archive_mode = on
   archive_command = 'команда_архивирования %p %f'
   ```

2. `wal_level` – уровень детализации WAL. Должен быть как минимум `replica` для архивирования (это по умолчанию в современных версиях). `logical` тоже подходит (включает всё необходимое, и для логической репликации).
3. `archive_mode = on` – включает сам механизм архивирования. (Это нельзя менять без перезапуска: require restart).
4. `archive_command` – команда, которую сервер будет выполнять каждый раз, когда очередной WAL-сегмент заполнен и готов к архивации. В этой строке можно использовать плейсхолдеры:

   - `%p` – абсолютный путь к архивируемому файлу WAL (в каталоге pg_wal).
   - `%f` – имя файла (только имя).

   Команда должна возвращать 0 при успехе, иначе PostgreSQL **повторно попытается** архивировать сегмент (через некоторое время). Примеры:

   ```
   archive_command = 'cp %p /mnt/server_archives/%f'
   ```

   Этот пример для Unix: копировать файл WAL из pg_wal (путь подставится вместо %p) в указанный каталог архивов (с именем %f). Для Windows нужно экранировать, обычно используют builtins copy или утилиты вроде robocopy.

   Важный момент: если команда завершается с ненулевым кодом, PostgreSQL **останавливает удаление старых WAL** – т.е. пока сегмент не будет успешно скопирован, он не удалится из pg_wal. Таким образом обеспечивается надежность: либо архив есть, либо WAL остаётся на диске.

5. (Опционально) `archive_timeout = 600` – таймаут принудительной ротации WAL в секундах. Если база неактивна, WAL-сегмент может заполняться долго; этот параметр заставит PostgreSQL каждые N секунд начинать новый сегмент, даже если предыдущий не полон, чтобы он ушёл в архив. Слишком маленький timeout создаст много пустых WAL-файлов, но слишком большой может означать потерю данных до N секунд активности при сбое (если сегмент не закрыт, и диск погиб – последние изменения потеряются). 5-15 минут обычно.

После изменения этих параметров нужно перезапустить сервер (archive_mode).

1. **Организовать место хранения архивов:** Это может быть директория на другом диске/сервере, репозиторий, облачный бакет и т.д. Главное – чтобы `archive_command` могла туда поместить файл. Распространённый вариант: NFS-смонтированный сетевой ресурс или локальный каталог (который потом бэкапится внешней системой).

После настройки, при работе сервера мы будем получать последовательность файлов в архивной директории, например:

```
000000010000000A000000FE
000000010000000A000000FF
000000010000000B00000000
...
```

и т.д. – имена WAL-сегментов.

Чтобы **восстановиться до определенного момента (PITR)**, требуется: - Иметь базовый **полный бэкап** базы данных (обычно физический) на некоторую точку времени T0. - Иметь **все WAL-сегменты с момента T0 до целевого момента Target**.

Процесс восстановления (PITR) в PostgreSQL: 1. Остановить работающий сервер (если ещё не). **Важно:** PITR производится офлайн. 2. Развернуть (восстановить) полный бэкап в целевое место PGDATA. Например, распаковать архив PGDATA или развернуть snapshot. 3. Собрать необходимые WAL. У нас архив WAL находится отдельно – либо скопировать их в какой-то каталог. 4. Настроить recovery.conf-параметры. Начиная с PostgreSQL 12, вместо отдельного файла используется `postgresql.conf` + наличие файла-флага: - Создать `recovery.signal` в папке PGDATA (пустой файл). Это скажет при старте: "войти в режим восстановления с архивом". - В `postgresql.conf` указать, откуда брать архивные журналы:

```
restore_command = 'cp /mnt/server_archives/%f "%p"'
```

(обратная команда archive_command: здесь мы копируем файл из архива (`%f` подставится именем сегмента) в место, ожидаемое сервером (`%p` полное имя в pg_wal)). Можно использовать альтернативы: если архив сжат – команду распаковки, если на удалённом – скрипт копирования по ssh и т.д. - Указать **целевой момент восстановления** (не обязательно, если хотим до конца WAL): - `recovery_target_time = '2025-06-17 02:00:00+03'` – время (с таймзоной) до которого откатывать. Или: - `recovery_target_xid = '123456'` – до конкретного XID транзакции. - `recovery_target_name = 'метка'` – до named point (если использовалась команда pg_create_restore_point, которая ставит маркер в WAL). - `recovery_target = immediate` – если хотим сразу завершить восстановление, как только применили весь WAL. - Опционально: `recovery_target_inclusive = true/false` (включать ли транзакцию на указанном моменте).

```
 Если эти параметры не задать, восстановление пойдёт до самого конца
имеющихся WAL (т.е. максимально возможное состояние).
```

- (В старых версиях эти параметры писались в файл *recovery.conf*).
- Настроить политику при достижении точки: по умолчанию, как только цель достигнута или WAL кончатся, сервер **переходит в обычный режим** (и начинает принимать запросы). Если хотим, можно задать `pause_at_recovery_target = on` чтобы сервер **поставился на паузу** (не завершал recovery), давая возможность администратору проверить состояние прежде чем завершить recovery.
- Запустить сервер. Процесс **recovery**:

- PostgreSQL прочитает `backup_label` (из бэкапа) – узнает откуда начинать WAL.
- Начнёт вызывать `restore_command` для последовательных WAL-сегментов, применять все записи из них. Он будет продолжать, пока не дойдет до условия остановки.
- Если указан recovery_target_time, он применит WAL вплоть до первой транзакции, временной штамп commit которой превышает заданное время (лишнее откатит).

- Как только цель достигнута (или кончились WAL):

    - Сервер либо останавливается (pause), либо удаляет файл `recovery.signal` и выходит из режима восстановления.
    - При выходе он фиксирует специальный WAL-сегмент `restore done` и может сменить timeline (увеличивает номер timeline, начиная новую ветку истории – архивный WAL относится к старой timeline).
    - База данных открывается для обычной работы.

- По завершении можно подключаться и убедиться, что состояние соответствует желаемому моменту. Например, если случайно удалили таблицу в 10:00, можно откатиться на 09:59 и увидеть ее на месте.

Не забыть: - **Отключить архивирование на время recovery** (или настроить так, чтобы восстановленный экземпляр случайно не начал писать в тот же архив!). Обычно для PITR делают восстановление на **другой сервер** или в отдельный каталог, чтобы не мешать продакшену. - Проверить, что **pg_hba.conf** блокирует подключение обычных пользователей, пока идет recovery (PostgreSQL 12+ сам запрещает внешние коннекты во время recovery, кроме реплик). - После успешного восстановления можно сохранить старые WAL (на всякий случай) и почистить архив, если нужно, либо продолжить архивирование уже с новой timelines.

Пример настройки (postgresql.conf фрагмент):

```
archive_mode = on
archive_command = 'gzip < %p > /mnt/backup/wal/%f.gz'

# ... при восстановлении:
restore_command = 'gunzip < /mnt/backup/wal/%f.gz > %p'
recovery_target_time = '2025-06-17 02:00:00+03'
```

В этом примере архивы WAL хранятся сжатыми (`.gz`), а при восстановлении команда gunzip распаковывает их обратно.

Итого: Непрерывное архивирование + PITR дают мощный инструмент защиты данных. Можно выполнять бэкапы раз в неделю, а все изменения между ними сохранять через WAL – тогда максимальная потеря данных в случае сбоя определяется лишь интервалом между последним архивированным WAL и моментом сбоя (как правило, секунды). Настройка этого требует включения archive_mode и написания корректного archive_command, а восстановление – приготовления базы из бэкапа, указания restore_command и, при необходимости, цели восстановления. PostgreSQL автоматизирует применение WAL, что существенно упрощает жизнь DBA.

14. Репликация Master-Slave, Multi-Master, репликация без мастера (p9-s5)

Вопрос: *Какие архитектурные варианты репликации данных существуют в распределённых системах? Опишите модели Master/Slave, Master/Master и репликацию без единого мастера (masterless). Как они реализуются и в чем их преимущества и недостатки?*

Ответ: В контексте распределённых баз данных **репликацией** называют поддержание нескольких копий одних и тех же данных на разных узлах (серверах) кластера. Это делается ради повышения отказоустойчивости (если один узел выйдет из строя, данные не потеряны, т.к. есть копии) и/или для распределения нагрузки на чтение. Существует несколько основных моделей организации репликации:

- **Модель Master-Slave (ведущий/ведомый)** – (она же Primary/Standby). Предполагается, что один узел является **главным (master)** – он принимает все запросы на изменение данных (INSERT/UPDATE/DELETE и DDL). Остальные узлы – **ведомые (slaves, или standbys)** – получают от мастера копии изменений и применяют их у себя, поддерживая актуальную копию базы. При этом, как правило, ведомые узлы работают в режиме **только чтение** (можно отправлять SELECT-запросы, называются *hot standby*, если они активно обслуживают чтение). Запросы на запись на slave либо запрещены, либо перенаправляются на master.

Как реализуется: мастер пересылает транзакционные изменения на слейвы. В PostgreSQL это **физическая потоковая репликация** через WAL: master транслирует WAL-записи, а standby применяет их к своим файлам данных, повторяя все действия. Аналогичные подходы в MySQL (binlog replication), Oracle Data Guard и т.д. Мастер обычно ведёт очередь изменений, чтобы отставшие слейвы могли догнать.

Преимущества: простота концепции, однозначность источника данных (нет конфликтов, так как изменения идут только с одного узла). Проще обеспечить согласованность – на мастере данные последовательно изменяются, а слейвы просто воспроизводят. Масштабирование чтения – можно распределить читающие запросы по слейвам. Отказоустойчивость – при падении мастера можно произвести **failover**: один из слейвов продвигается в мастера (promotion), и работа продолжается (с некоторым временем простоя на переключение).

Недостатки: мастер – единственная точка отказа (Single Point of Failure) до выполнения failover. При выходе мастера из строя требуется вручную или полуавтоматически переключить роли. Масштабирование записи отсутствует – всё упирается в возможности одного сервера (остальные – пассивные реплики). Также есть **задержка репликации**: слейвы отстают от мастера на доли секунды или секунды; если мастер внезапно выходит, последние транзакции могли не дойти до слейвов (потенциальная потеря данных – можно уменьшить синхронной репликацией, см. далее).

- **Модель Multi-Master (Multi-Primary)** – несколько узлов одновременно выступают в роли главных, все они принимают запросы на запись (и чтение). Копии данных распределены на всех, и изменения, сделанные на одном узле, реплицируются на остальные. Эта модель намного сложнее, т.к. возникают проблемы **конфликтов изменений**: если два мастера одновременно изменяют одну и ту же запись по-разному – как разрешить конфликт? Требуются дополнительные протоколы и договорённости (временные метки, победитель/проигравший, last-write-wins, либо предотвращение конфликтов через мьютексы или разделение данных). Multi-master может работать по принципу: каждый узел фиксирует

изменения и отсылает другим, иногда через централизованное хранилище WAL или через peer-to-peer.

Примеры реализации: PostgreSQL не поддерживает multi-master репликацию “из коробки”. Однако существуют расширения (Postgres-BDR, Bucardo, citus), которые позволяют нескольким узлам PG работать как мульти-мастер, синхронизируя изменения через логическую репликацию и отложенные конфликты. MySQL имеет режим circular replication (каждый мастер – слейв другого, образуя кольцо). СУБД Oracle RAC – частично multi-master (общая диск-система, координация через interconnect). В NoSQL это более распространено (Cassandra – каждый узел может принимать запись, затем используя кворум для согласования, хотя можно сказать, что там вообще нет понятия "мастер").

Преимущества: нет единой точки отказа – любой узел может принимать записи, поэтому система устойчива к падению отдельных серверов (клиент может переподключиться к другому). Масштабируется запись – нагрузка распределяется по нескольким узлам (но не линейно, т.к. нужен оверхед на синхронизацию). Пользователю ближе может быть локальный мастер (географическое распределение) – уменьшение задержек.

Недостатки: крайне сложная реализация согласованности. Конфликтующие транзакции могут приводить к разным копиям данных. Нужно решать: либо предварительно как-то синхронизировать (что замедляет систему, например, двухфазный коммит между мастерами), либо разрешать конфликты постфактум (что может нарушать целостность с точки зрения приложения). Часто multi-master системы жертвуют строгой консистентностью ради доступности (см. CAP). Многие СУБД предпочитают режим одного Primary, так надёжнее. Multi-master обычно используется либо для территориально распределённых систем, где задержки большие (каждый регион пишет локально, а потом реплицирует – мирится с временной несогласованностью), либо в специализированных системах с ограниченным набором операций.

- **Репликация без мастера (Masterless, децентрализованная)** – это архитектура, где **нет выделенного ведущего узла**, все узлы равноправны, и данные распределены среди них с избыточностью. Запрос на запись может быть обработан любым узлом, он сам распространяет изменения на остальные по какому-то протоколу согласования. Такая система, как правило, базируется на принципах **Eventually Consistent** (конечной согласованности) и **кворумного согласования** (см. CAP, кворум ниже).

Примеры: Cassandra, Riak, Amazon Dynamo – популярные NoSQL-хранилища, используют ring-кластер без мастера. Данные шардуются между узлами по ключам (consistent hashing), и для каждого ключа хранится N реплик на разных узлах. Любой узел может получать запросы, перенаправлять к ответственным узлам или сам применяет (если он хранит эту часть данных).

Механизм: обычно применяют **кворумное чтение/запись**. Например, настроены параметры: N – число реплик каждого куска данных, W – сколько реплик должно подтвердить запись для успеха, R – сколько реплик нужно опросить для чтения. Если выбрать $W + R > N$, то можно гарантировать пересечение множества узлов обслуживающих чтение и запись, что обеспечивает консистентность при чтении последнего записанного значения. (Иначе возможны ситуации чтения устаревших данных – что и происходит при eventual consistency, когда $W+R \leq N$). В masterless системах нет центрального координатора: узлы сами договариваются – например, запись клиент отправляет на все N узлов, ждет W подтверждений; чтение – запрашивает у N, ждёт R ответов (может возвращать самый свежий по версии). Со временем, неактуальные копии догоняют (задержка репликации), даже если $W+R \leq N$, система придёт к единому состоянию – это и зовётся **конечной согласованностью**.

Преимущества: высокая **устойчивость к разделению сети** – нет мастер-узла, который при потере связи “распилит” кластер; даже если кластер распадается на части, каждую часть можно продолжать использовать (пусть и с риском рассинхрона). Отличная масштабируемость: добавление узлов прозрачно увеличивает и объем хранения, и совокупную производительность (запись/чтение распределяются). Нет узкого места – нагрузка симметрично распределяется.

Недостатки: *Согласованность данных* сложнее гарантировать. При сетевых разрывах разные узлы могут принять конфликтующие обновления, которые потом тяжело слить. Обычно приходится идти на компромиссы (допускать *eventual consistency*, когда какое-то время разные узлы имеют разные данные, либо выбрасывать один из конфликтующих апдейтов). Сложнее программировать: приложения должны уметь работать с возможностью чтения устаревших данных, с “раздвоением” (например, возвращаются две версии – разрешение возложено на клиента либо на сервер по правилу). Кроме того, выполнение сложных запросов (JOIN, транзакции) крайне ограничено – такие системы зачастую NoSQL-ориентированы, без общего SQL/ACID по всем узлам.

Вывод: - *Master-Slave* – классический подход: один пишущий узел, остальные – реплики для чтения и фейловера. Легко понять, относительно просто поддерживать целостность, но не масштабирует запись. - *Multi-Master* – все узлы пишущие, но нужна синхронизация для консистентности. Реализуется в специальных системах; хороший аптайм, но риск конфликтов. - *Masterless* – полностью распределённая среда, никакого центрального узла. Обеспечивает высокую доступность и масштабирование, но система консистентности “размазывается” на узлы (обычно используют кворум, CAP – см. ниже) и сложно достигается строгая согласованность.

Каждая модель может быть уместна в разных сценариях: реляционные СУБД чаще Master-Slave; мультимастер иногда в геораспределенных или clustered (Galera Cluster для MySQL, Oracle RAC); безмастерные – в больших NoSQL хранилищах, требующих доступности и партиционирования прежде всего.

15. Физическая vs. логическая репликация PostgreSQL (p8-s24)

Вопрос: *Объясните разницу между физической репликацией и логической репликацией в PostgreSQL. Какой подход в каких случаях применяется?*

Ответ: PostgreSQL поддерживает два основных механизма репликации: **физическую** и **логическую**, которые сильно отличаются по уровню и возможностям.

- **Физическая репликация** – это побайтная (блочная) репликация всего кластера. Основана на передаче **WAL (журнала предзаписи)** от главного узла к узлу-реплике и его воспроизведении. Реплика (standby) хранит **точную копию** файлов данных главного сервера, включая всю структуру базы, OID, и т.д., и поддерживает её актуальной, применяя поток WAL-записей [1]. При физической репликации реплика *не может менять своё содержимое самостоятельно* – она *ро-бэка* (read-only), и она имеет идентичную базу данных (те же базы, таблицы, пользователи и пр.).

Технически: для физической репликации нужно, чтобы `wal_level` на мастере был `replica` или `logical` (по умолчанию `replica`). Настраивается максимальное число слотов/отправителей WAL (`max_wal_senders`). На реплике делается base backup, затем запускается как standby (с файлом `standby.signal`), указывают `primary_conninfo` (как достучаться до мастера). Standby подключается по протоколу репликации и **стримит WAL**. Базовые утилиты:

`pg_basebackup` для начальной копии, параметры `primary_conninfo`, `primary_slot_name` (если используются replication slots).

Применение: физическая репликация хороша для горячих standby (резерв на случай отказа) и для распределения нагрузки на чтение. Так как standby – полная копия, можно на нее направлять SELECT запросы (при `hot_standby = on`, что по умолчанию). Физическая репликация также единственный способ реплицировать *все* данные, включая системные каталоги, информацию о ролях, и т.п. – т.е. standby имеет всё, чтобы в случае чего стать новым мастером.

Ограничения: на реплике **нельзя писать**, и **все базы** реплицируются целиком. Невозможно выбрать “реплицируй только одну таблицу” – реплицируется весь кластер. Также физическая репликация требует одинаковой версии PostgreSQL и архитектуры (копируются бинарные данные). Она не очень гибкая при топологии: обычно один мастер – несколько standby. Хотя есть **каскадная репликация**: standby сам может выступать источником WAL для ниже стоящих реплик (например, primary -> standby1 -> standby2; standby2 получает WAL от standby1). Это используется для распределения нагрузки трафика WAL или географического разделения.

- **Логическая репликация** – репликация **избранных таблиц** в виде *потока логических изменений* (на уровне отдельных записей). Логическая репликация появилась в PostgreSQL 10. Она работает через механизм **логического декодирования WAL**: на мастере транзакционные журналы декодируются в ряд логических сообщений (типа “INSERT в таблицу X значение (...), DELETE ..., etc.”). Эти изменения передаются подписчикам, которые выполняют их у себя, воспроизводя данные на уровне содержимого таблиц.

Архитектура: Логическая репликация организована по схеме **публикация-подписка** (*publish-subscribe*): - На источнике (мастере) создается **публикация** (`CREATE PUBLICATION`) – по сути, набор таблиц, изменения в которых должны транслироваться. Например:

```
CREATE PUBLICATION mypub FOR TABLE mytable, myschema.anothertable;
```

Можно публиковать все таблицы (FOR ALL TABLES) или конкретные. Публикация может настраивать, какие операции реплицировать (INSERT, UPDATE, DELETE, обычно все). - На стороне получателя (подписчика) создается **подписка** (`CREATE SUBSCRIPTION`), указывающая строку подключения к источнику и какую публикацию слушать:

```
CREATE SUBSCRIPTION mysub
  CONNECTION 'host=master dbname=mydb user=repl password=...'
  PUBLICATION mypub;
```

При создании подписки PostgreSQL подключается к мастеру, берет снимок текущих данных (инициализация – копирует существующие строки таблиц) и затем начинает получать поток изменений. На подписчике должны существовать аналогичные таблицы (с такой же структурой) – обычно их создают заранее, либо опция `WITH (create_slot = true, copy_data = true)` позволяет скопировать данные.

Особенности логической репликации: - Можно реплицировать *конкретные таблицы*, а не весь кластер. Это полезно, когда нужно синхронизировать только часть данных, или организовать более сложную топологию (например, несколько баз получают данные от одной). - Можно иметь **несколько источников** для разных таблиц (например, объединять потоки – хотя

конфликтующие изменения не разрешаются автоматически). - Подписчик — это **полноценная самостоятельная база**: он может и принимать данные по репликации, и позволять локальные изменения других таблиц. Локально, однако, таблицы, подписанные на логическую репликацию, по умолчанию **read-only** (их нельзя менять, иначе конфликт: можно override, но тогда изменения не реплицируются назад). - Логическая репликация позволяет сценарии **разветвления/ объединения**: например, вы можете настроить несколько подписчиков на одну публикацию (один ко многим), или наоборот – совмещать в одной базе публикации от разных баз (многие к одному). Однако “многие к одному” может привести к конфликтам, PostgreSQL сейчас не умеет их автоматически разруливать (на подписчике при конфликте применяет либо модель last-wins, либо останавливается – обычно стоит избегать). - Логическая репликация **не переносит DDL** изменения (схему нужно менять вручную синхронно). Она реплицирует только данные. Если на мастере ALTER TABLE (добавили колонку), администратор должен сам изменить таблицу на подписчике, иначе поток может остановиться (несоответствие структуры). - Требует чуть больше ресурсов: логическое декодирование – дополнительная нагрузка. Также, в отличие от физической, может добавлять некоторую задержку (но обычно незначительную). - Логическая репликация допускает разнородность версий (напр. можно организовать репликацию из PG10 в PG12, хотя официально поддерживается в пределах близких версий).

Применение: - **Избирательная репликация** – когда не нужна полная копия базы, а только определенные таблицы (например, есть большой кластер, но аналитической базе нужна лишь 2-3 таблицы из него). - **Интеграция данных** – объединение изменений из разных баз в одну (с оговорками о возможных конфликтах). - **Миграции/обновления без простоя** – можно логически реплицировать данные на новую версию PostgreSQL (даже с другим схемам или ОС) и переключить приложение (т.н. logical replication for upgrade). - **Мульти-мастер через логическую** – некоторые системы строят multi-master на основе логической репликации, прикручивая обработку конфликтов. - **Аудит/триггеры замены** – фактически логическая репликация дает поток изменений (подобно триггерам) – можно его читать и делать действия (похож на Kafka-stream).

Физическая репликация не даёт этих гибкостей: она хороша для резервирования и масштабирования чтения, но всё или ничего. Логическая – гибче, но сложнее и не заменяет физическую полностью.

Вывод: - *Физическая репликация:* уровень сегментов WAL, копия всего сервера, read-only реплики. Простая и надежная, используется для горячего резерва и распределения нагрузки чтения, требует одинаковой среды. - *Логическая репликация:* уровень отдельных логических изменений, выборочные таблицы, позволяет изменять структуру и сочетать с локальными данными, но не переносит DDL и может иметь конфликты. Используется когда нужна частичная или трансформированная копия данных, интеграция между системами, или upgrade/подмножество данных.

Оба вида репликации могут сосуществовать: например, можно иметь кластер с физической репликацией для фейловера и на мастер настроить публикацию для логической репликации в аналитический отбор данных.

16. Синхронная и асинхронная репликация (p9-s13)

Вопрос: *В чем разница между асинхронной и синхронной репликацией в PostgreSQL? Как настраивается режим synchronous replication и как он влияет на подтверждение транзакций?*

Ответ: Асинхронная репликация и **синхронная репликация** – это два режима координации между главным сервером (primary) и его репликами (standby) с точки зрения подтверждения транзакций.

- **Асинхронная репликация (async):** по умолчанию PostgreSQL работает в этом режиме. Главный сервер, выполняя транзакцию, **не ждёт подтверждения** от реплик о том, что они получили или применили изменения. WAL-запись посылается на standby, но мастер считает транзакцию завершенной (COMMIT) как только записал данные в свой локальный WAL (и на диск при необходимости), независимо от состояния слейвов. Это означает, что реплики могут **отставать**: если мастер упал сразу после коммита, на некоторые реплики последние транзакции могли не успеть дойти – возможна *потеря данных на реплике*. Но с точки зрения мастера транзакция уже сохранена (на диск WAL), так что при его рестарте она не потеряется (потеряется только если и мастер погибнет без возврата).

Плюсы async: минимальное влияние на производительность мастера – он не ждёт сети/acknowledgments, продолжает работать. Реплики могут хоть сильно отставать, это не тормозит работу основного. Поэтому асинхронный режим наиболее быстр и используется, когда небольшое окно потенциальной потери данных (фактор репликации) не критично.

Минусы: возможна ситуация, когда мастер подтверждает клиенту commit, но упав, ни одна реплика не получила эти данные – тогда эти последние транзакции будут **утрачены** (при переключении на реплику их не окажется). Это называется *небезопасностью по подтверждению:* клиент думает, что данные сохранены, а фактически ни одна постоянная копия их может не иметь (кроме может быть WAL на мастере, но он потерян если диск мастера погиб). Также чтения с реплик при async могут отставать – клиент, совершив запись и потом читая с реплики, может не увидеть свою запись (реплика ещё не получила).

- **Синхронная репликация (sync):** в этом режиме подтверждение транзакции на мастере происходит только после того, как заданное количество реплик подтвердят получение (а возможно и применение) данных. То есть COMMIT задерживается, пока хотя бы одна (или N) реплика не синхронизируется. Это гарантирует, что транзакция не потеряется при падении мастера: по крайней мере одна из реплик уже имеет эти данные в себе (в WAL, а в опционально и применёнными).

PostgreSQL позволяет настроить параметр `synchronous_commit` и `synchronous_standby_names`: - `synchronous_commit` – глобальный или для текущей транзакции параметр, определяющий, как далеко должна зайти транзакция, прежде чем считать COMMIT завершённым: - `on` (по умолчанию при sync mode включен) – ждать, пока хотя бы одна синхронная реплика записывает WAL (write) и *применит его в свою память (flush)*. - `remote_write` – ждать только записи в сеть/память реплики, но не fsync на реплике. - `local` – не ждать реплик (то есть фактически async). - `off` – даже свой WAL не синхронизировать (небезопасно, редко). - `remote_apply` (в новых версиях) – ждать не только записи на реплике, но и подтверждения, что реплика *применил* транзакцию (она видна для запросов на standby). Это самый строгий, но и медленный вариант. - `synchronous_standby_names` – задает список standby-серверов (или число `FIRST N`), которые считаются *синхронными*. Например:

```
synchronous_standby_names = 'FIRST 2 (node_A, node_B, node_C)'
```

означает: из подключенных реплик с именами node_A, node_B, node_C одновременно будут выбраны любые 2 как синхронные. Мастер будет ждать подтверждения от двух. Если какие-то

отвалятся, он выберет следующие из списка, etc. Без FIRST N, список – приоритетный, первая доступная берётся как sync.

Настройка: включить `synchronous_commit = on` (это по дефолту on глобально, но по умолчанию не заданы sync_standbys, поэтому фактически async), и задать `synchronous_standby_names` на мастере. На репликах имена задаются параметром `primary_conninfo` / `application_name`, который должен совпадать с именем в списке мастера. Можно на лету менять через `pg_ctl reload` или `ALTER SYSTEM` + reload.

Поведение: При синхронном режиме, когда транзакция на мастере коммитится, процесс *WAL Sender* будет ждать, пока *WAL Receiver* на реплике не сообщит о записи (и flush при `on`). Только после этого клиенту возвращается ACK на COMMIT. Таким образом, хотя мастер погибнет, по крайней мере одна реплика уже имеет транзакцию на диске (в WAL), и данные не потеряны – можно фейловериться без потерь.

Плюсы sync: гарантия отсутствия потери подтверждённых данных – стойкость к сбоям улучшена. Данные на выбранных репликах почти в реальном времени (опережение на мерцание).

Минусы: увеличивает **время отклика** транзакций, т.к. каждая COMMIT ждёт сети и записи на другом сервере. Если реплика медленная или связь медленная, мастер затормаживается. Если синхронная реплика падает, мастер будет *ждать её или переключения на другую* – т.е. может приостановиться в коммитах (синхронный режим ради консистентности жертвует доступностью: если нет указанного числа реплик, по умолчанию он будет висеть). В PostgreSQL можно смягчить: `synchronous_commit = remote_write` чуть быстрее (не ждёт fsync реплики), либо `quorum_commit` (9.6+) – можно настроить так, что достаточно любого N из M реплик (не конкретных, а кворума) – это немного повышает доступность. Но все равно, sync replication – компромисс: за 0 потерянных транзакций платим скоростью и риском остановки, если реплики недоступны.

Применение: Обычно продакшн PG используют **асинхронную репликацию**, чтобы не замедлять работу. Потенциальная потеря данных ограничивается несколькими последними секундами транзакций в случае полного краха мастера (что редко и, возможно, приемлемо). Синхронный режим применяют, когда критично не потерять ни одной транзакции (финансы, и пр.) или при географическом резервировании, когда готовы платить производительностью. Можно комбинировать: например, 1 реплика синхронная в пределах датацентра (быстрый канал) для безопасности, а удалённые реплики асинхронные.

Настройка примера: Пусть есть master и 2 реплики (node1, node2). Хотим синхронно на одной:

```
synchronous_standby_names = 'ANY 1 (node1, node2)'
synchronous_commit = on
```

`ANY 1` (эквивалент FIRST 1) означает: ждать подтверждения от любой одной реплики из списка. Если node1 или node2 откликнулись – коммит продолжится. Если хотя бы один доступен, мастер работает. Если обе недоступны – все коммиты будут висеть (или, начиная с PG12, по timeout перейдёт в async автоматически, если включить `synchronous_commit = remote_apply` + `vacuum_defer_cleanup_age`, но это детали).

Вывод: асинхронная репликация – быстрее, но с маленьким окном потери данных; синхронная – полностью защитит от потери подтвержденных транзакций, но замедляет систему и делает ее более чувствительной к состоянию реплик. Выбор режима зависит от требований системы: баланс между **C (согласованность данных)** и **A (доступность)** по теореме CAP. PostgreSQL даёт гибкость настроить промежуточные варианты (remote_write, number of standbys, etc.) для тонкой балансировки.

17. Шардирование данных (p9-s?)

Вопрос: *Что такое шардирование (разбиение) базы данных? Зачем оно применяется и какие способы шардирования используются в распределённых системах хранения?*

Ответ: Шардирование – это подход к масштабированию базы данных, при котором данные **разделяются по частям (шардам)** и распределяются на разные узлы (серверы). Вместо того, чтобы хранить всю таблицу целиком на одном сервере, таблица делится – например, по диапазонам значений или по хэш-функции от ключа – и части хранятся отдельно. Каждый такой раздел называется **шард (shard)**, а совокупность узлов, хранящих шарды, образует кластер. В идеале, шардирование позволяет практически **линейно масштабировать объём данных и нагрузку** – добавлением узлов можно увеличить вместимость и параллельную пропускную способность.

Зачем нужно: Когда объёмы данных или нагрузка превышают возможности одной машины (по памяти, диску, CPU или IO), шардирование – способ **горизонтального масштабирования**. В отличие от вертикального (когда улучшают один сервер, что имеет предел), горизонтальное предполагает использование множества относительно обычных серверов, поделив между ними работу. Шардирование также может улучшить доступность: при выходе из строя одного узла недоступна только часть данных (если настроено резервирование, то и этого можно избежать), а не вся база.

Основные методы шардирования: 1. **Шардирование по ключу (хэширование или диапазоны):** выбирается поле или группа полей (shard key), по которым определяется, в какой шард попадёт запись. Например, часто берут первичный ключ или идентификатор клиента. - *Диапазонное шардирование* – каждому шарду соответствует диапазон значений ключа. Например, записи с ID 1-1,000,000 – на узле A, 1,000,001-2,000,000 – на узле B, и т.д. Или по алфавиту: A-M фамилии – на одном шарде, N-Z – на другом. Этот подход понятен, но может страдать от **дисбаланса** (если данные неравномерны, один диапазон может оказаться гораздо больше/активнее). - *Хэш-шардирование* – значение ключа пропускают через хэш-функцию, и по результату (обычно по модулю) определяют номер шарда. Это даёт равномерное распределение записей (если хэш хорош и ключи равномерны). Но при хэшировании теряется локальность по диапазону (например, диапазон запросов или сортировка по ключу затрагивает все шарды). Зато хорошее *balance* нагрузки.

Ключевое здесь – выбор хорошего shard key: он должен равномерно распределять данные и чаще запросы должны содержать этот ключ (чтобы маршрутизатор понимал, на какой шард идти). Например, шардирование по user_id в большой социальной сети: все данные пользователя (посты, сообщения) могут идти на шард, зависящий от user_id.

1. **Статическое vs. динамическое шардирование:**
2. *Статическое (fixed) – число шардов задаётся изначально (например, 16 шардов), возможно, даже больше, чем узлов, а затем на узлы раскладываются эти секции. При добавлении нового узла перетасовывают целые шард-разделы с других узлов на новый, но общее*

количество шардов фиксировано. Такой подход применялся, напр., в Riak, Couchbase: обычно выбирают число шардов с запасом под будущие узлы.
    - *Плюсы:* не надо переразбивать существующие секции – при росте добавляешь узел и переносишь ему некоторые шарды целиком (миграция на уровне целых секций). Это уменьшает объем перебалансировки.
    - *Минусы:* изначально трудно выбрать оптимальное число (если задашь слишком мало – потом масштабироваться проблематично без полного ребаланса; задашь слишком много – overhead на управление шардами). Также, если данные неравномерны, одни шарды могут вырасти больше других.

3. *Динамическое шардирование:* шарды создаются/делятся по мере роста данных. Обычно начально каждый узел имеет по одному (или несколько) шардов, и когда какой-то шард становится слишком большим или узел перегружен – этот шард **разбивается на два** и часть переносится на другой узел. Это похоже на работу B-дерева или на auto-splitting в MongoDB/HBase: шарды – это диапазоны; когда диапазон переполняется (по размеру или числу записей) – он делится, а новый диапазон переносится на другой узел, если есть.

    - *Плюсы:* система **автоматически балансируется** при росте нагрузки, нет необходимости предсказывать заранее. Секции остаются более-менее равного размера.
    - *Минусы:* более сложная реализация. Если данных мало, все может оказаться на одном узле (не разбивается, что может быть нормально). Если плохо настроено – может случаться много делений (thrashing).
    - Пример: MongoDB – шардирование по диапазонам с auto-split + auto-balance, HBase – регионы (region server) аналогично.

4. **Консистентное хеширование (виртуальные узлы):** специальный метод, популярный в NoSQL (Dynamo, Cassandra). Значения ключа отображаются хэш-функцией на *устраненное пространство* (кольцо хешей). Узлы (серверы) тоже получают позиции на этом кольце (каждый узел может иметь множество виртуальных позиций). Данные с ключами, попадающими между позицией узла A и узла B, принадлежат узлу B и т.д. Когда добавляется новый узел, кольцо перераспределяется: некоторым диапазонам, ранее принадлежавшим другим узлам, назначается новый узел. **Количество секций** изначально может быть очень большим (виртуальных), чтобы баланс был гладким.

5. Пример: Cassandra создает 2^N token ranges и распределяет по узлам.
6. *Плюсы:* при добавлении/удалении узла **не нужно перестраивать все** – только соседние диапазоны хэша перераспределяются.
7. *Минусы:* сложнее понять, что где лежит без специального координатора, т.к. нет явного человека-читаемого диапазона. Но для ПО это не проблема.

Многие современные системы используют консистентное хеширование, т.к. оно минимизирует *rebalance* при изменении состава узлов.

Перебалансировка (rebalance): При добавлении нового узла, систему нужно **ребалансировать** – переместить часть данных на новый узел, чтобы нагрузка выровнялась. - В статическом подходе: обычно просто перемещают некоторое количество статических шардов с существующих узлов на новый (в идеале равное количество у каждого отберут). Это более контролируемо: переносим целые части. Но если секций мало, перенос может быть крупнозернистым. - В динамическом: если новый узел пуст, система может начать делить самые большие шарды и переносить половинки на него. - В консистентном хэше: новый узел получает

определенные позиции на кольце, от прежних узлов он "оттянет" их части (соседи отдают диапазоны).

Проблемы шардирования: - **Complex queries:** Запросы, охватывающие данные из нескольких шардов, сложно выполнять – нужно либо направлять на все узлы (scatter/gather), либо иметь механизм *merge*. Например, `SELECT COUNT(*) FROM table` – нужно выполнить на всех шардах и суммировать. СУБД может этого не уметь прозрачно (PostgreSQL vanilla – не умеет, но есть Pgpool-II, Citus extension). - **Distributed transactions:** если одна операция затрагивает несколько узлов (например, трансфер денег между клиентами на разных шардах), требуется распределённая транзакция (двухфазный коммит) или денормализация. Это усложняет согласованность (CAP – сетевые ошибки могут мешать). Многие системы ради простоты избегают транзакций между шардами. - **Hot spots:** если шард ключ неудачный, может возникнуть, что одна часть нагружена сильнее. Например, шардирование по стране: США может иметь 50% данных, а другие страны – меньше, перегруз на одном шарде. Поэтому ключ выбирают так, чтобы разрезать “горячие” группы. - **Operational complexity:** нужно инфраструктуру: маршрутизатор запросов (или встроенные механизмы в БД), мониторинг баланса, процесс ребалансировки с минимальным влиянием.

Применение: Когда данных очень много (в сотни миллионов/миллиарды строк или размеры > нескольких TB) и одна машина не вытягивает, шардирование – решение. Многие NoSQL сразу рассчитаны на шардирование. PostgreSQL же традиционно не имел встроенного шардинга (кроме схемы секционирования, но это не меж-узловое). С появлением PG 10 логическая репликация и PG 11 `pg_partman` etc., а также проектов типа Citus (добавляет распределённость), PG тоже можно шардинговать.

В контексте экзамена, достаточно помнить: *шардирование = разделение данных по узлам для масштабирования и отказоустойчивости; методы – по диапазону, по хэшу; нужно решать проблемы ребаланса и запросов по множеству узлов.*

18. Теорема CAP (p9-s10)

Вопрос: *Сформулируйте теорему CAP для распределённых систем. Как она поясняет компромисс между согласованностью и доступностью в условиях сетевых разделений?*

Ответ: Теорема CAP – фундаментальный результат в теории распределённых вычислений (теорема Брюера), гласящий, что *в любой реализуемой системе разделённых данных невозможно одновременно обеспечить более чем две из трёх свойств: Consistency, Availability, Partition tolerance*. Проще: при проектировании распределённой системы хранения данных приходится **жертвовать одним из трёх** ключевых качеств: - **C (Consistency)** – согласованность (строгая, линейная) данных на всех узлах в любой момент. Это означает, что все узлы видят одни и те же данные одновременно; любой чтущий запрос получает либо последние подтверждённые данные, либо система не отвечает. - **A (Availability)** – доступность: система отвечает на все запросы (которые поступают на доступные узлы) с ненулевым ответом даже во время отказов (возможно, ответы могут быть старые или разные на разных узлах, но ответ придёт). То есть каждый запрос получает некоторый отклик, даже если часть системы упала. - **P (Partition tolerance)** – устойчивость к разделению сети: система продолжает работать даже если сеть разделилась на части (некоторые узлы не могут общаться с другими). Формально – система выдерживает произвольное число потерь/задержек сообщений между узлами (вплоть до полного разделения кластеров) и всё ещё может обрабатывать запросы.

Теорема CAP утверждает: *в присутствии разделения сети* (P – которое рано или поздно случается, если система распределённая), система не может одновременно быть строго согласованной и полностью доступной. При сетевом расколе придётся либо: - **Отключить доступность (A)** для части узлов – т.е. заблокировать систему (некоторые запросы будут ожидать восстановления связи, чтобы обеспечить консистентность), - Либо **пожертвовать консистентностью (C)** – позволить узлам отвечать автономно, даже если это приведёт к расхождению данных (несогласованности) между разделёнными частями.

В отсутствие партиций (P) – когда связь идеальна – можно иметь и C и A. Но гарантировать отсутствие разделений нельзя (в любой момент может оборваться связь). Поэтому система должна быть спроектирована или как CP, или как AP, или (теоретически) CA, но CA-вариант – это по сути нереплицированная, нераспределённая система (single node), т.к. если есть хоть какая-то распределённость, P рано или поздно надо учитывать.

Примеры: - **CP-системы** (Consistency + Partition tolerance, жертва Availability): эти системы предпочитают *согласованность* даже ценой отказа в обслуживании, если что-то не так. Например, классические реляционные СУБД с синхронной репликацией: при падении связи между мастером и синхронной репликой, мастер может остановить приём новых транзакций (т.к. не может гарантировать консистентность с репликой). Или системы типа Zookeeper, etcd (они требуют кворума для операции; если нет кворума, операции блокируются – но данные никогда не расходятся). В CP-системах, пока разделение не устранено или пока недоступные узлы не восстановятся, часть запросов не будет обслуживаться (система может частично “замереть”), но согласованность не нарушится. - **AP-системы** (Availability + Partition tolerance, жертва Consistency): эти системы выбирают *доступность* – они будут отвечать на запросы даже при разделении, но допускают, что разные части кластера могут расходиться в данных. Пример – многие NoSQL (Cassandra, Riak) по умолчанию предоставляют eventual consistency: при разделе сети каждый узел продолжает принимать записи/чтения (высокая доступность), но в разных разделах могут произойти конфликтующие обновления. Затем, когда связь восстановится, система *постепенно приведёт данные к согласованности* (реплицируя и разруливая конфликты) – т.е. согласованность не мгновенная. Такое поведение означает, что на время разделения или сразу после, клиенты могли получить устаревшие или противоречивые данные, но зато сервис не простаивал. - **CA-системы** (Consistency + Availability) теоретически противоречат P, т.е. они возможны только если партиций не происходит. Иногда memcached или single-instance relational DB приводят как CA (т.к. один узел: последняя запись всегда читается (C) и узел отвечает (A), но при разделении – если узел отрезан – система просто целиком недоступна, что можно считать нарушением PTolerance).

Важно понимать, что CAP – грубая модель. В реальности, многие системы стараются подобрать **баланс**: например, SQL-базы с настроенной репликацией могут переключаться (failover) – в момент переключения коротко потерять доступность, но потом восстановить. NoSQL могут позволять настроить разные уровни консистентности (кворумы).

С появлением теоремы CAP ещё ввели понятия: - **BASE** vs **ACID**: BASE (Basically Available, Soft-state, Eventually consistent) – отсыл к AP системам, где “в основном доступно, состояние может быть неустойчивым, но в итоге согласуется”; ACID – скорее CP/CA, строго. - **PACELC**: уточнение CAP, что даже когда нет партиций (P), система все равно может выбирать latency vs consistency (LC): т.е. даже без разделения, либо ответ быстрее, но возможно менее консистентно (к примеру, читать со слейва – быстро, но может чуть устарело) или ждать согласованности (медленнее).

Заключение: Теорема CAP в контексте распределённых хранилищ напоминает архитекторам: *при дизайне репликации нужно выбрать, чем жертвовать при сбоях сети.* Либо при проблемах останавливать часть операций, обеспечив целостность данных (стратегия CP – например,

банковская система, где неверные данные недопустимы), либо продолжать работу даже разобщённо, рискуя временной неконсистентностью (стратегия AP – например, социальная сеть: лучше показать хоть какие-то данные, пусть и с возможными расхождениями, но не “упасть” полностью). Универсального решения "и то и другое" – не существует, по крайней мере, на этапе активного разделения сети.

19. Конечная согласованность и кворум (eventual consistency, quorum) (p9-s12)

Вопрос: *Что означает термин "конечная согласованность" (eventual consistency) в распределённой базе данных? Как механизм кворума чтения/записи (параметры W и R) помогает достичь согласованности в системе без мастера?*

Ответ: Конечная согласованность (eventual consistency) – это модель согласованности в распределённых системах, согласно которой при отсутствии новых обновлений данные на всех узлах в конце концов (через какое-то конечное время) **станут согласованными (одинаковыми)**. Иначе говоря, система может некоторое время содержать **устаревшие данные на некоторых узлах**, но гарантируется, что если ждать достаточно долго после последнего изменения, все реплики выровняются. Eventual consistency – более слабое требование, чем строгая консистентность: она не обещает, что чтение сразу после записи вернёт новое значение (может вернуть старое, если читаешь с отставшей реплики), но обещает, что *в итоге* все реплики применят изменение.

Eventual consistency характерна для **AP-систем** (по CAP), которые предпочитают доступность: они позволяют узлам работать автономно и **реплицируют изменения асинхронно**. В результате возникает **задержка репликации** (*replication lag*): промежуток времени между фиксацией изменения на одном узле и его отражением на другом. В течение этой задержки разные узлы могут отдавать разные ответы – данные противоречивы. Однако имеется механизм рассылки изменений (например, обмен сообщений или периодическая синхронизация), который рано или поздно доставит обновления всем. Отсюда "в конце концов согласованно" – если система в течение достаточно длительного времени работает без новых записей, все узлы придут к одному состоянию.

Пример: DNS – обновление записи на мастере может доходить до вторичных DNS-серверов минуты или часы (TTL кэш), но в итоге все получат новый IP. Пока не дошло – некоторые клиенты видят старый IP, некоторые новый. Или Cassandra: запись на один узел распространяется фоново на реплики; сразу после операции другой узел может вернуть старое значение, но через секунду-две (или по требованию read-repair) обновится.

Кворумное согласование (quorum) для согласованности: В системах без мастера (например, Dynamo, Cassandra) вводят параметры **W** (write quorum) и **R** (read quorum). Обычно имеется *N* – число реплик каждого фрагмента данных. Правило **кворума**: если **W + R > N**, то **читающая и пишущая операции перекрываются минимум на одном узле**, что гарантирует чтение свежих данных.

Объяснение: - **W** – сколько реплик должны подтвердить запись, чтобы она считалась успешно зафиксированной. - **R** – сколько реплик опрашивается при чтении (и часто берётся самый "новый" ответ среди них по метке времени). - Допустим, *N=3*, выбраны *W=2*, *R=2*. Тогда: - При записи система запишет на минимум 2 реплики (может быть все 3, но ждёт 2 подтверждения) – значит хотя бы на 2 узлах актуальные данные. - При чтении она читает минимум с 2 реплик. Если одна из них была обновлена (а при W=2 минимум 2 из 3 хранят новое), то читатель увидит новую версию. - В худшем случае, если запись подтверждена двумя узлами, а чтение случилось с

третьего (отставшего) и одним из обновлённых, чтящая система может обнаружить конфликт (две разные версии). Обычно она выберет самую свежую (по timestamp) – таким образом, клиент получит новую версию.

Если $W+R > N$, обязательно хотя бы один узел попадёт и в множество, подтверждающее запись, и в множество, отвечающее на чтение – на этом узле данные новые, он "перетянет" консистентность.

Если же $W+R \leq N$, то возможна ситуация, что запись разошлась на, скажем, W узлов, а чтение опросило другой непересекающийся поднабор R узлов – ни один из опрошенных не имел новой версии, и чтение вернёт **устаревшие данные**. Это нарушение сильной консистентности. Но иногда так делают ради большей доступности (меньше квоты – быстрее отдает, даже если мало узлов ответили).

Пример настройки: - N=3 реплики. - Если хотим сильную консистентность на чтение: W=2, R=2 (как выше) – это распространённая конфигурация. Она **кворумная** (2+2 > 3). - Можно ещё сильнее: W=3, R=1 – запись должна быть на всех 3 (синхронно), чтение можно с любой (после записи любой имеет новую). Это даёт консистентность, но снижает доступность (все узлы должны принять запись). - W=1, R=3 – наоборот: пишем быстро на один узел (остальные потом), но читаем всегда со всех и агрегируем – тоже кворум (1+3 >3), но чтение медленное, запись быстрая. Cassandra default был W=1, R=1 (что не кворум, eventual by default).

Достижение согласованности: Система с кворумами при нормальной работе стремится к консистентности: - Если запросы используют кворумные R и W, они фактически реализуют читаемую-после-записи консистентность (близкую к последовательной, с оговоркой про concurrent writes). - Если случилось разделение сети: кворум может быть не доступен. Например, 3 узла, разбиение 2+1. Кворум (2) всё равно достижим в части с 2 узлами – она продолжает обслуживать и читать/писать (2 из 3 – ок). Одинокий узел не имеет кворума (нужно 2 для пишущих или читающих), значит либо: - либо он не будет обслуживать (система CP: меньшая часть стопорится), - либо система временно позволяет нарушить условие (AP: одинокий узел тоже принимает записb с W=1, но тогда W+R уже не >N, и консистентность страдает – потом, при слиянии, нужно сливать, т.е. eventual).

В целом, *кворумный протокол* – способ настроить AP-систему ближе к консистентности, жертвуя некоторой доступностью (нужен ответ большинства). Он часто используется: Cassandra, Riak позволяют настраивать *quorum read/write*.

Что при несоблюдении кворума: Если W+R <= N, то возможны **несогласованные чтения**. В примере W=1,R=1,N=3: записали на 1 узел (два нет), сразу читаем с другого – он не знает об обновлении, вернёт старые данные. Однако, при eventual consistency, со временем: - Фоновая система узнает про невыполненный до конца write (например, недостающие реплики отметят, что им надо обновиться), и либо сама реплицирует (“hinted handoff” в Cassandra – откладывает запись для узла офлайн и позже отдаёт), либо при следующем чтении с них, если замечается что у соседа новее, выполняется **read repair**: новое значение отправляется на отставшую реплику чтобы выправить. - Если два узла получили противоречивые записи (split-brain, concurrent updates), при слиянии нужно решить conflict resolution: обычно по timestamp – победит более новый (last write wins), или объединение (в CRDT-структурах).

Eventual consistency вкупе с кворумами: - Если всегда соблюдается W+R > N, то в сущности у вас **строгая консистентность для синхронных операций** (можно считать CP по CAP). - Если иногда

нарушается (из-за partition например), то на время разбиения могут выполняться не-кворумные операции – они потом синхронизируются, но в моменте несогласованность.

Итог: *Конечная согласованность* – свойство, при котором система не гарантирует немедленной консистентности, но обеспечивает, что при отсутствии новых изменений все реплики сойдутся. *Кворумный механизм* – инженерный приём, позволяющий повысить согласованность: устанавливая требование пересечения читающих и пишущих узлов, мы исключаем "разъехавшиеся" ответы. Таким образом, настройкой W,R можно настроить баланс: - Кворум (W,R) – ближе к **сильной консистентности** (но может чуть снизить доступность при сбоях), - Меньшие W,R – выше **доступность и скорость**, но только eventual consistency.

Соблюдение кворума является одним из способов достигать консистентности в masterless системах, не вводя мастер, а распределяя ответственность между группой узлов (большинством). При этом system tolerant к некоторым partition, пока есть кворум, но если раскололось на маленькие части – малые части не смогут прогрессировать (т.е. система становится CP, жертвуя полной доступностью всех узлов ради консистентности большинства).

Практические вопросы

1. Инициализация кластера PostgreSQL: утилита initdb (p3-s8)

Вопрос: *Как создать новый кластер базы данных PostgreSQL "с нуля"? Опишите процесс и команды для инициализации кластера, включая настройку окружения и результаты работы* `initdb`.

Ответ: Для подготовки PostgreSQL к работе необходимо **инициализировать кластер** базы данных – то есть создать структуру каталогов и минимальные служебные данные. Это делается командой `initdb` (англ. initialize database). В Linux/Unix она обычно расположена в директории `bin` PostgreSQL (например, `/usr/pgsql-15/bin/initdb` или при установке по умолчанию `initdb` доступна в PATH).

Шаги и команды:

1. **Создание системного пользователя ОС (при необходимости):** Обычно для PostgreSQL выделяется отдельный системный аккаунт (по умолчанию `postgres`), от имени которого будут запускаться сервер и выполняться initdb. Это пользователь ОС, не путать с ролями PG. Например (под root):

   ```
   adduser postgres
   ```

   Назначим ему каталог данных (например, `/var/lib/pgsql/15/data` или другой).

2. **Подготовка окружения:** определить **каталог PGDATA** – место, где будут храниться все данные кластера. Его можно задать либо переменной окружения `PGDATA`, либо указывать `-D` в командах. Например:

```
export PGDATA="/var/lib/pgsql/15/data"
mkdir -p $PGDATA
chown postgres $PGDATA
```

(Выдаём права владельца каталога пользователю postgres).

3. **Выполнение initdb:** Переключаемся на пользователя postgres (чтобы файлы были его):

```
sudo -iu postgres
```

Запускаем команду:

```
initdb -D "$PGDATA" -U <имя_суперпользователя> --locale=<локаль> --
encoding=<кодировка>
```

4. `-D <dir>` указывает каталог данных (можно опустить, если PGDATA задан).
5. `-U <name>` задаёт имя **суперпользователя PostgreSQL**, который будет создан. По умолчанию, если не указано, берётся имя текущего ОС пользователя. Часто это "postgres" (если initdb выполняется от пользователя postgres).
6. `--locale` определяет локаль (язык/регион) по умолчанию для базы (влияет на сортировку, формат данных). Можно также `--lc-collate`, `--lc-ctype` отдельно указать.
7. `--encoding` – кодировка по умолчанию (например UTF8).
8. `-E` тоже кодировка.
9. Можно добавить `-W` чтобы задать пароль суперпользователя (для peer метод обычно не нужно).
10. Пример: `initdb -D /var/lib/pgsql/15/data --locale=en_US.UTF-8 --encoding=UTF8 -U postgres`

После запуска initdb выводит ряд сообщений:

```
initializing PostgreSQL database at /var/lib/pgsql/15/data
selecting default shared_buffers ... 128MB
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

Success. Database system has been initialized.
```

(Примерно, могут отличаться).

1. **Что делает initdb:**

2. **Создаёт структуру каталогов PGDATA** (поддиректории: `base`, `global`, `pg_wal`, `pg_xact`, `pg_multixact`, `pg_notify`, `pg_stat`, `pg_logical`, `pg_replslot`, `pg_tblspc`, `pg_twophase` и др., а также файл `PG_VERSION`).
3. **Создаёт стандартные базы данных**: `template1`, `template0` и `postgres`.
    - `template1` – шаблон для будущих БД (из неё будут копироваться).
    - `template0` – "чистый" резервный шаблон (идентичная копия template1).
    - `postgres` – по умолчанию, пользовательская пустая база, принадлежащая администратору (часто используется как рабочая по умолчанию).
4. **Создаёт системные каталоги** и начальные таблицы. Это происходит при выполнении внутреннего bootstrap-сценария – фактически, initdb запускает postgres в специальном "режиме зарождения" (bootstrap mode), который исполняет SQL-скрипты для создания всех системных таблиц, типов, функций.

5. **Назначает суперпользователя** базы данных. Имя суперпользователя будет либо значение указанное через `-U`, либо (по умолчанию) имя ОС-пользователя, запускавшего initdb. У этого суперпользователя (обычно "postgres") нет пароля, если явно не указан `-W` (в dev-среде это ОК, при peer-auth).

6. **Задаёт локаль и кодировку кластера** – они сохраняются как свойства template0/ template1 и по умолчанию унаследуются всеми базами. Локаль влияет на правила сортировки (LC_COLLATE) и классификации символов (LC_CTYPE), а encoding – на хранение текста. Если требуется другая локаль/кодировка, нужно указывать при initdb (потом поменять для кластера нельзя, только перестраивать или создавать базы с override COLLATE).

7. **Создаёт конфигурационные файлы** в PGDATA: `postgresql.conf` (настройки сервера), `pg_hba.conf` (правила доступа), `pg_ident.conf` (правила отображения идентификаторов). Эти файлы наполняются шаблонными настройками по умолчанию. В частности, `postgresql.conf` будет содержать (закомментированные) default параметры и несколько включенных: `data_directory`, `hba_file`, `ident_file` (с путями, если `-D` нестандартный), и locale/encoding info.

8. **Выполняет начальную настройку параметров**. initdb может выбрать разумные дефолты для `shared_buffers` и `max_connections` исходя из системы (например, "selecting default max_connections ... 100" может появиться). В PG 12+ initdb пытается, но в основном defaults.
9. **Инициализирует WAL**: создает `pg_wal` директорию, первый WAL-сегмент, файл контрольной точки `PG_VERSION` (с версией) и `global/pg_control` (ключевая структура управления кластером).

10. **Выводит инструкции**: Например, после инициализации может сообщить, как запустить сервер: `pg_ctl -D /var/lib/pgsql/15/data -l logfile start` или напомнить про пароли.

11. **Результат:** каталог PGDATA готов к запуску сервера. В нём:

12. База данных `postgres`, `template1`, `template0` (каждая – это подкаталог в `base/` с OID).
13. Роль-суперпользователь (имя postgres) создана.
14. Конфиги на месте (pg_hba.conf по умолчанию зачастую настроен на `local all all peer` и `host all all 127.0.0.1/32 md5`, etc).

15. Файлы и директории как выше.

16. **Дальнейшие действия:** Можно запустить сервер:

```
pg_ctl -D "$PGDATA" -l logfile start
```

или использовать `postgres -D $PGDATA` (прямой запуск). Если это системный инстанс – через `systemctl start postgresql` (который под капотом сделает то же). После запуска можно подключаться:

```
psql -d postgres   # зайти в созданную базу postgres под текущим пользователем (если peer, от пользователя postgres)
```

Итого: `initdb` выполняется один раз для создания нового кластера PostgreSQL. В процессе она делает: - Создание каталогов (global, base, pg_wal, ...), - Заполнение минимально необходимых таблиц, - Создание шаблонных баз, - Настройка локали/кодировки, - Формирование конфигурационных файлов, - Создание superuser-ролей.

После успешного initdb вы увидите сообщение *"Success. You can now start the database server using: ..."*.

2. Запуск сервера и подключение клиента (psql)

Вопрос: *Как запустить сервер PostgreSQL вручную и как подключиться к нему с помощью клиента psql? Какие основные параметры нужно указать?*

Ответ: После инициализации кластера (командой initdb) или установки PostgreSQL, чтобы начать работать, нужно **запустить сервер**. Серверная программа – это `postgres` (или через управляющую `pg_ctl`).

Запуск сервера: - **Прямой запуск:**

```
postgres -D /path/to/PGDATA [options]
```

Например:

```
postgres -D /var/lib/pgsql/15/data
```

Этот процесс будет запущен на переднем плане в текущей консоли (обычно его запускают в фоновом режиме или через сервис менеджер). Он при запуске откроет порт (по умолчанию 5432), и начнет слушать сокеты (локальный Unix-socket и TCP).

Можно указать опции: - `-p <порт>` чтобы указать не стандартный порт. Например, `postgres -D data -p 5433`. - `-c config_item=value` – переопределить конфигурацию (например, `-c shared_buffers=1GB`).

Чтобы не держать процесс, часто используют `&` (background) или `pg_ctl`.

- **Через pg_ctl:**

```
pg_ctl -D /path/to/PGDATA -l logfile start
```

`pg_ctl` – удобная утилита управления:
- `-D` – указывает каталог данных,
- `-l logfile` – файл для вывода логов сервера,
- `start` – команда запустить (есть еще `stop`, `restart`, `status`).

Пример:

```
pg_ctl -D /var/lib/pgsql/15/data -l /var/lib/pgsql/logfile start
```

pg_ctl запускает сервер в фоновом режиме, вывод сервера направляет в указанный логфайл. Оно даст сообщение вида:

```
waiting for server to start.... done
server started
```

- **Через системный сервис:** На системах с systemd, после установки PGSQL, можно:

```
sudo systemctl start postgresql
```

(служба берет каталог данных из настроек, обычно /var/lib/pgsql/<ver>/data). Systemctl под капотом вызывает pg_ctl. Статус сервера – `systemctl status postgresql`.

Проверка запуска: - Лог: в `postgresql.conf` by default logging might be in `pg_log` or `journalctl`. - `pg_ctl status`:

```
pg_ctl -D data status
```

покажет PID если запущен.

Подключение с помощью psql: `psql` – интерактивный CLI-клиент PostgreSQL. Чтобы подключиться, нужно указать: - **Хост** (`-h`), **порт** (`-p`), - **базу данных** (если не указана, берёт $PGDATABASE или имя пользователя), - **пользователя** (`-U` или $PGUSER), - (если требуется) пароль – psql запросит, или можно заранее задать PGPASSWORD env, или настроить .pgpass.

Пример (локальное подключение через UNIX socket):

```
psql -d postgres
```

По умолчанию: - Хост: *локальный сокет* (`/tmp/.s.PGSQL.5432` или `/var/run/postgresql/.s.PGSQL.5432`, зависит от OS). - Порт: 5432. - База: `postgres` (если -d указано postgres). Иначе, если -d не указано, берется имя пользователя ОС. - Пользователь: текущий ОС пользователь (если `-U` не указан).

Если initdb выполнялся от `postgres` и `pg_hba.conf` настроен на peer, то команда:

```
sudo -iu postgres
psql
```

зайдет без пароля, подключившись к `postgres` DB под role `postgres`.

Пример (TCP/IP подключение):

```
psql -h 127.0.0.1 -p 5432 -U myuser mydatabase
```

Это: - Подключиться к хосту 127.0.0.1 (loopback) на порт 5432, - под пользователем (роль) `myuser`, - к базе `mydatabase`. Если pg_hba.conf на `127.0.0.1/32` требует md5/scram, psql выведет `Password:` – нужно ввести пароль пользователя myuser (который задается командой CREATE USER ... PASSWORD).

При успешном соединении psql выдает баннер:

```
psql (15.1)
Type "help" for help.

mydatabase=>
```

И ждет команды.

Основные psql параметры: - `-d, --dbname=DBNAME` – имя базы данных (или строка подключения целиком `postgresql://user:pass@host:port/db`). - `-U, --username=NAME` – имя пользователя. - `-h, --host=HOSTNAME` – хост (имя или IP; `-h /tmp` можно указать путь к нестандартному сокету). - `-p, --port=PORT` – порт (если не стандартный). - `-W` – force prompt for password (psql автоматически запрашивает, если pg_hba говорит password/scram, но -W заставит всегда). - `-c "SQL"` – сразу выполнить SQL команду и выйти. - `-f file.sql` – выполнить скрипт. - `-X` – не выполнять ~/.psqlrc (в случае scripting). - `-1` – wrap entire script in single transaction (nice for big scripts). - `--set=var=val` – set psql variable.

После подключения: - Можно исполнять SQL, - метакоманды `\l` (список DB), `\c otherdb` (переключиться), `\dt` (tables), etc.

Соединение под другим пользователем: Если вы на ОС под user1, но хотите зайти как role user2 via peer, можно:

```
psql -U user2 postgres
```

Поскольку peer проверяет имя ОС=роль, обычно чтобы зайти под другой ролью, либо аутентификация md5/pass, либо `sudo -iu user2` OS switch + peer.

Ошибки подключения: - "Connection refused" – сервер не запущен или firewall/incorrect port. - "could not connect to server: No such file or directory" – пытается через unix-socket, но сокета нет (сервер не запущен или -h path mismatch). If server is on TCP only, use -h. - "FATAL: role 'X' does not exist" – неправильное имя пользователя (сразу на этапе login). - "FATAL: database 'X' does not exist" – указанной базы нет (psql by default tries user name as DB if not specified). - "psql: error: connection to server on socket ... failed: FATAL: password authentication failed for user ..." – пароль неверен.

Установка переменных окружения: psql honors: - `PGHOST`, `PGPORT`, `PGUSER`, `PGDATABASE`, `PGPASSWORD` (though PGPASSWORD not recommended in env for security, better .pgpass file). This can save typing.

Stop server: - `pg_ctl -D data stop [-m fast]` - or `sudo systemctl stop postgresql`.

Вывод: Запуск сервера:

```
pg_ctl -D /path/to/data -l logfile start
```

Подключение:

```
psql -h host -p port -U user dbname
```

Если всё локально и defaults, можно просто `psql` (под своим именем к базе с тем же именем). После этого можно вводить SQL или \commands.

3. Логическое резервное копирование: pg_dump и pg_restore (p7-s8)

Вопрос: *Как создать логическую резервную копию базы данных PostgreSQL с помощью утилиты pg_dump? В каких форматах можно сохранить дамп и как затем восстановить базу из этой копии? Приведите примеры команд и опций pg_dump и pg_restore.*

Ответ: Для логического (структурно-данного) бэкапа PostgreSQL используется утилита `pg_dump`, входящая в дистрибутив. Она подключается к запущенному серверу и извлекает объекты базы данных (схемы, таблицы, данные) в виде SQL-скрипта или в специальном формате.

Создание дампа с помощью pg_dump:

- Простейший способ:

```
pg_dump mydatabase > backup.sql
```

Здесь pg_dump подключится к базе `mydatabase` (по $PGHOST, $PGUSER или peer, etc.) и выведет дамп (SQL-скрипт) в stdout, откуда перенаправляем в файл `backup.sql`. Этот **plain text** дамп – обычный текстовый файл, содержащий команды `CREATE TABLE/INSERT/...`.

- Указание параметров подключения:

```
pg_dump -h localhost -p 5432 -U myuser -F p -f backup.sql mydatabase
```

Опции:

- `-h` хост (не указывать для локального через сокет).
- `-p` порт.
- `-U` пользователь (если не задан, берет OS пользователя).
- `-f <file>` – выходной файл (иначе stdout).
- `-F <format>` – формат дампа (по умолчанию `p` – plain text SQL). Возможные форматы:
  - `p` (plain) – текстовый SQL (то же, что перенаправление).
  - `c` (custom) – бинарный "custom" формат pg_dump.
  - `d` (directory) – каталог с отдельными файлами для каждого блоба/таблицы.
  - `t` (tar) – tar-архив, содержащий файлы (похож на directory, но одним архином).

Пример: `-F c` создаст `.dump` файл (имя как `-f`), который требует `pg_restore` для восстановления.

- Другие ключевые опции pg_dump:
- `-n schema` – дамп только определённой схемы.
- `-t table` – дамп только указанной таблицы (можно несколько `-t` или шаблоны, psql interpret).
  - Важно: `-t` без `-s` / `-a` включает и структуру и данные *только* этой таблицы (и связанных объектов, напр. sequence).
- `-T table` – исключить таблицу (inverse of -t).
- `-s` / `--schema-only` – только DDL (схема, без данных).
- `-a` / `--data-only` – только данные (таблицы без create, только insert/copy).
- `--inserts` – использовать INSERT вместо COPY для данных (INSERT на каждую строку).
- `--column-inserts` – INSERT с перечислением колонок (более портируемо, но verbose).
- `--no-owner` – не включать команды смены владельца (чтобы при восстановлении не требовались суперправа).
- `-x` / `--no-privileges` – не дампить GRANT/REVOKE (привилегии).
- `-O` / `--no-owner` – (alias) игнорировать OWNER.
- `-E encoding` – задаёт кодировку дампа.
- `--section=pre-data|data|post-data` – можно дампить части (например, только pre-data (schema defs), или data, или post-data (indexes, constraints)).
- `--clean` – добавить в дамп команды `DROP ...` перед `CREATE` (удобно если восстанавливать поверх существующей DB, чтобы заменить всё).

Форматы: - **Plain text (SQL):** Плюсы – читаем, можно выполнить частично, отредактировать. Минусы – восстановление требует выполнить все SQL (может быть долго), нельзя легко выбирать части при восстановлении (надо вручную редактировать). - **Custom (binary .dump):** Сжимается, pg_dump -Fc. Требует `pg_restore`. Позволяет **выборочно восстановить** объекты,

поддерживает **параллельное восстановление**. - **Directory (-Fd):** Создаёт папку, внутри: - файлы для схемы, для данных каждой таблицы отдельно, - `toc.dat` (table of contents). Можно сжимать/не сжимать каждый файл. Восстанавливается `pg_restore -Fd`. Поддерживает **параллельный dump/restore** (опция `-j` (jobs) с pg_dump/pg_restore). - **Tar (-Ft):** TAR-архив, содержит похожее на directory (индекс + data files). Можно разархивировать tar и получить структуру, либо скормить pg_restore напрямую tar.

- **Мультипоточность:** `pg_dump` может выгружать в несколько потоков (`-j N`) **только** при формате directory (-Fd). Например:

  ```
  pg_dump -Fd -j 4 -f backup_dir mydb
  ```

  создаст каталог backup_dir, параллельно 4 worker потока будут выгружать разные таблицы одновременно.

- **Пользователи и BLOB'ы:** `pg_dump` в обычном режиме дампит контент *одной* базы. Информацию о ролях (пользователях) и tablespace он не содержит (кроме owner/privilegies). Чтобы бэкапнуть **все базы и роли** – `pg_dumpall`:

  ```
  pg_dumpall > all.sql
  ```

  Этот скрипт начнётся с `CREATE ROLE` для всех ролей, затем `CREATE DATABASE` и `\connect` и дампы каждой DB. `pg_dumpall -g` только глобальные (roles, tablespaces).

Восстановление дампа:

- **Если дамп – SQL (plain text):** Восстановление происходит выполнением этого скрипта через psql:

  ```
  createdb mydatabase    # сначала создать пустую базу (если в дампе нет
  CREATE DATABASE).
  psql -d mydatabase -f backup.sql
  ```

  Если в дампе есть `CREATE DATABASE ...` (например, `pg_dump --create`), можно просто:

  ```
  psql -f backup.sql postgres
  ```

  (он сам создаст базу и переключится). Plain-дамп можно разбивать, редактировать. Минус: долго выполнять, особенно большие INSERTы (но pg_dump по умолчанию использует COPY, что довольно быстро).

- **Если дамп – custom, directory или tar:** Используется утилита `pg_restore`.

```
createdb mydatabase   # создаём БД, если дамп без --create
pg_restore -d mydatabase backup.dump
```

pg_restore сам подключается к БД (по PGHOST, PGUSER env или `-h,-U` аргументами).

Ключевые опции pg_restore: - `-d <dbname>` – сразу указывает базу для восстановления (pg_restore тогда сам подключится). - `-C` – создать базу из дампа (т.е. если дамп включает CREATE DATABASE, pg_restore может подключиться к postgres и создать указанную базу). - `-U, -h, -p` – стандартные параметры подключения. - `-j N` – параллельное восстановление (для -Fd или -Ft, custom). Например:

```
pg_restore -j 4 -d mydatabase backup_dir
```

Это распараллелит восстановление отдельных таблиц/indexes. - `-v` – verbose (полезно видеть прогресс, особенно параллельный). - `-L listfile` – можно указать файл-лист объектов (см. -l). - `-l` – вывести перечень содержимого дампа (список всех объектов с номером и название). Это можно сохранить, отредактировать (например, удалить строку чтобы не восстанавливать тот объект) и подать -L. - `-n schema` – восстановить только определенную схему. - `-t table` – только определенные таблицы. - `-x` – не восстанавливать привилегии (GRANT/REVOKE). - `-O` – игнорировать владельцев (все объекты будут принадлежать тому юзеру, под которым pg_restore подключился). - `--disable-triggers` – отключить триггеры (FK) на время загрузки данных (полезно, если загружаем data-only dump и хотим ускорить, но нужно после вручную ENABLE). - `--clean` – перед созданием объектов выполнять DROP (аналогично pg_dump --clean, но можно и тут).

Особенность: pg_restore можно направлять и в stdout (по умолчанию, если -f не указано), тогда он выводит SQL. Но обычно его используют для actually apply: - For plain dumps, pg_restore не нужен (psql suffice). - For custom/dir/tar, pg_restore выполняет *преобразование* (разжатие, reorder) и посылает в сервер.

Examples: - Восстановить в новую базу с созданием:

```
pg_restore -C -d postgres backup.dump
```

Это подключится к `postgres` DB, прочитает имя базы из дампа (в дампе хранится CREATE DATABASE ...), создаст ее, и наполнит. (Нужно права суперпользователя обычно).

- Восстановить только данные, без схемы:

```
pg_restore -d mydb --data-only backup.dump
```

- Восстановить только схему (структуру):

```
pg_restore -d mydb --schema-only backup.dump
```

• Восстановить только конкретную таблицу:

```
pg_restore -d mydb -t mytable backup.dump
```

• Просмотреть, что в дампе:

```
pg_restore -l backup.dump
```

(выведет список, примерно: `;
123; 1255 TABLE myschema mytable; ...`)

pg_dumpall vs pg_restore: - `pg_dumpall` выдает plain SQL для всех DB, его надо через psql применять, `pg_restore` тут не подходит, потому что pg_restore работает с pg_dump custom/tar/directory output.

Использование parallel: - Перед параллельным восстановлением, БД должна существовать (`-j` нельзя с -C, т.к. -C implies some serial steps). - pg_restore -j N: N<= number of CPUs, expedite large data loads and index creations concurrently.

Резюме примера: 1. Бэкап:

```
pg_dump -Fc -f mydb.dump mydb
```

(Custom формат). 2. Восстановление:

```
createdb mydb_restored
pg_restore -d mydb_restored mydb.dump
```

(Восстановит в новую базу). *Или*

```
pg_restore -C -d postgres mydb.dump
```

(pg_restore сам создаст базу mydb).

4. Физическое резервное копирование: pg_basebackup (p7-s3)

Вопрос: *Как выполнить физический бэкап кластера PostgreSQL на уровне файлов? Что делает утилита pg_basebackup и какие у неё ключевые параметры? Опишите процесс создания полной файловой копии (base backup) и условия её использования.*

Ответ: Физическое резервное копирование предполагает копирование файлов базы данных *как есть*. В PostgreSQL для этого используется утилита `pg_basebackup`, которая упрощает получение *полной копии PGDATA*.

pg_basebackup: Это утилита, работающая по протоколу репликации (т.е. соединяется к серверу как реплика) и забирающая файл за файлом весь каталог данных. Она: - может выполняться *на*

работающем сервере (то есть **"горячий бэкап"**), - обеспечивает согласованность копии, используя механизм контролной точки.

Подготовка: Чтобы pg_basebackup работал, должны выполняться условия: - На сервере `postgresql.conf`: `wal_level` должен быть не менее `replica` (default есть). - Иметь роль с правом `REPLICATION` и доступ по pg_hba для репликации:

```
CREATE ROLE backup_user REPLICATION LOGIN ENCRYPTED PASSWORD '...';
```

pg_hba.conf строка, например:

```
host  replication  backup_user  192.168.0.100/32  md5
```

- Достаточно места на целевом диске.

Команда:

Запускается на стороне, куда хотим сохранить:

```
pg_basebackup -h <host> -p <port> -U <replication_user> -D /path/to/
backup_dir [options]
```

Пример:

```
pg_basebackup -h 127.0.0.1 -p 5432 -U backup_user -D /backups/pg-full -Fp -
Xs -P -v
```

Что здесь: - `-h` и `-p` – хост и порт сервера. - `-U` – имя репликационного пользователя. - `-D` – *директория назначения*, куда сложить бэкап. Она должна быть пустой или не существовать (pg_basebackup может сам создать). - `-F` – формат: - `-Fp` или `--format=plain` (по умолчанию) – просто files, директория PGDATA копируется в указанную папку. - `-Ft` (`--format=tar`) – создать tar-архив (или несколько, если указаны tablespaces). (С PG15 ещё `-T` for tar with separate WAL, etc). - `-X` – включает способ обращения с WAL: - `-X fetch` – после копирования данных взять все оставшиеся WAL-файлы, необходимые для консистентности. - `-X stream` – запускать параллельно поток для трансляции WAL (с PG 9.3+, лучше, потому что захватывает WAL в реальном времени). - `-X none` – (default до 9.6) не копировать WAL (тогда копия может быть несогласованной без них – не рекомендуется).

Обычно используют `-X stream`.

- `-c fast/slow` (в старых) – уже нет, cp vs tar.
- `-P` – показывать прогресс (процент).
- `-v` – verbose (печатать сообщения).
- `-Z 0-9` – сжатие (для tar).
- `-j N` – с PG13, parallel tar.
- `--wal-dir` – если хотим WAL складывать отдельно (при -Ft можно, tar WAL separate).

- `-T` – можно ремапить таблиспейсы (например, если cluster имел tablespace, куда в backup положить).

Процесс: - pg_basebackup под капотом запускает на сервере команду `START_BACKUP` (с меткой), которая делает контрольную точку и позволяет делать backup-mode. - Затем начинает копировать все файлы из PGDATA: - Он исключает некоторые, которые не нужны: например, `postmaster.pid` (то, что привязано к работающему инстансу), `pg_replslot/*` (можно или нельзя, зависит), `pg_xlog` / `pg_wal` – WAL может копироваться отдельно. - По умолчанию, копирует и конфигурационные файлы (`postgresql.conf`, `pg_hba.conf`, ...). - Если указано `-X stream`, параллельно стартует поток репликации, чтобы считывать WAL; - Все изменения, которые идут во время копирования, фиксируются в WAL; - pg_basebackup пишет WAL-сегменты в `pg_wal` в папке бэкапа по мере их появления. - Когда копирование всех файлов данных завершено, он выполняет `STOP_BACKUP` (команда завершения backup на сервере). `STOP_BACKUP` записывает специальный WAL record (backup-end) и возвращает имя последнего необходимого WAL сегмента. - При `-X fetch`, механизм другой: `STOP_BACKUP` возвращает ласт сегмент, pg_basebackup потом докачивает все WAL файлы, начиная с начального LSN бэкапа до того, что вернул, из pg_wal архива.

- В результате, в каталоге бэкапа получаем:
- Полную копию всех файлов бд (base/, *global/*, и пр.),
- В `pg_wal/` – последние WAL сегменты, нужные для recovery (если -X stream, может быть неполный последний сегмент + `.partial`).
- Файл `backup_label` (в PG < 15) либо `backup_manifest` (PG13+ генерирует manifest) – `backup_label` содержит метку, LSN начала, timeline и т.п.;
- PG 13+ ещё генерирует `backup_manifest` (список файлов, checksums) – можно отключить `--no-manifest`.

Использование бэкапа: - Полученная папка – по сути готовый PGDATA. Чтобы восстановить, обычно: - Остановить работающий кластер (например, для PITR), - Стереть/убрать старые файлы, - Развернуть (скопировать) содержимое бэкапа в PGDATA. - **Важно:** Если бэкап делался как часть PITR + continuous archiving, надо также взять архивированные WAL **после** бэкапа. - Перед запуском восстановленного, если это для PITR: поместить нужные WAL (или настроить restore_command), создать recovery.signal.

- Если цель – создать standby (физическую реплику):
- Запускаем pg_basebackup на реплике:

```
pg_basebackup -h primary_host -U repl_user -D /var/lib/pgsql/15/data -
Fp -Xs -P -R
```

`-R` (или `--write-recovery-conf` PG12-, PG >= 12 adds standby.signal) – сразу создавать конфиг для реплики:

- Раньше (до PG12) `recovery.conf` с primary_conninfo.
- Сейчас (PG12+): создает `standby.signal` файл и appends `primary_conninfo` and `primary_slot_name` (if given) to postgresql.auto.conf.
- Так что после -R, можно просто `pg_ctl start` на реплике, и она подключится к primary.

Ключевые параметры еще: - `-R` – выше, автоконфиг для standby. - `-C` – просто проверить, сколько wal_seg_size / version (list tablespaces). - `--waldir` – указать, куда сложить WAL (в bkp). -

`--slot=NAME` – использовать репликационный слот, чтобы на мастере не убирались WAL до завершения бэкапа (полезно, чтобы он точно получил все нужные WAL). - `--no-sync` – после копирования *не* делать fsync локально (по умолчанию fsync = true). - `--progress` (equiv -P).

Speed: - pg_basebackup single-threaded per tablespace (one tablespace per thread). В PG15 parallel for tar came, but for plain directory parallel still not supported. - You can compress on the fly with -Z.

Пример полной команды:

```
pg_basebackup -U backup_user -h primary.host.com -p 5432 \
  -D /backups/cluster1/`date +%Y%m%d` -Fp -Xs -P -v --label
"full_backup_20250617"
```

- This will produce directory /backups/cluster1/20250617 containing data. - --label sets a friendly label stored in backup_label.

Условия использования: - Бэкап действителен для восстановления в ту же версию PG (major version). Нельзя прямой pg_basebackup PG14 -> восстанавливать на PG13, нужно PG14 сервер. - backup includes all databases cluster-wide. - Should be combined with WAL archiving if want PITR beyond base backup snapshot. - If used for standby, should be taken from a consistent state of primary (pg_basebackup ensures that via start/stop backup LSN logic). - If cluster has multiple tablespaces, by default pg_basebackup copies them all; if needed, can redirect each tablespace location or use tar to handle them.

N.B.: Physical backup can also be done manually: stop server and cp -r PGDATA, or use filesystem snapshot; but pg_basebackup is official online method.

In conclusion, `pg_basebackup` provides an **online, consistent** full copy of the PostgreSQL cluster data directory, easy to restore as a whole or use for replication setup.

5. Непрерывное архивирование WAL и PITR (p7-s13)

Вопрос: *Как настроить непрерывное архивирование WAL в PostgreSQL и восстановить базу данных до нужного момента времени (PITR)? Какие параметры отвечают за архивирование журналов транзакций и как выполняется восстановление с их помощью?*

Ответ: Непрерывное архивирование WAL (continuous WAL archiving) – механизм, при котором PostgreSQL сохраняет каждый завершённый WAL-сегмент во внешнее хранилище (например, файловый архив), чтобы их можно было использовать для восстановления базы данных на любой момент времени после последнего полного бэкапа.

Настройка архивирования: 1. Включить архивирование в конфигурации `postgresql.conf`: - `wal_level = replica` (или `logical`) – должно быть не ниже `replica` для генерации полных WAL-записей (включая для репликации/архивации). По умолчанию сейчас у большинства версий это так. - `archive_mode = on` – включает режим архивирования WAL. (Изменение требует перезапуска сервера). - `archive_command = 'команда'` – задаёт команду, которую сервер будет выполнять для каждого завершённого WAL-сегмента, чтобы скопировать его в архив.

Пример команды (для Linux):

```
archive_command = 'cp %p /mnt/server_archives/%f'
```

Здесь `%p` – полный путь к текущему WAL-файлу, а `%f` – его имя. Эта команда копирует WAL в каталог `/mnt/server_archives/`. Она должна возвращать **0 при успехе**; если код возврата ненулевой, PostgreSQL **повторит попытку** архивации снова (и не удалит этот WAL из pg_wal, пока не заархивируется успешно).

- Альтернативы: можно использовать `gzip` для сжатия:

  ```
  archive_command = 'test ! -f /backup/arch/%f && gzip < %p > /backup/arch/%f.gz'
  ```

  (здесь `test ! -f` гарантирует не перезаписывать, если вдруг существует).

- В Windows команда может быть `copy %p X:\arch\%f`.

- `archive_timeout = 600` (в секундах) – опционально, форсировать переключение WAL-сегмента, если в течение указанного времени он не заполнился полностью. Это нужно, чтобы даже при низкой активности транзакций регулярно отсылать WAL-ы в архив (иначе при простое долго ничего не будет архивировано). Например, 600 секунд (10 минут) – означает, что хотя бы каждые 10 минут будет начинаться новый WAL-сегмент, и текущий отправится в архив, даже если он не полный. Ставить слишком малое значение не стоит (будет много почти пустых WAL-файлов).

- Перезапустить PostgreSQL (так как archive_mode = on требует restart). После этого сервер начнёт выполнять archive_command для каждого завершённого WAL-сегмента:

- WAL-сегмент (обычно 16 MB размер, имя что-то типа `000000010000000000000A1`) записывается в `pg_wal` (текущий), а когда заполняется или при выполнении CHECKPOINT, PostgreSQL завершает его и зовёт archive_command.
- Если команда успешно скопировала файл, сервер помечает его как архивированный и может удалить/перезаписать при нужде.
- Если команда вернула ошибку, сервер **будет пытаться снова** каждые archive_timeout/каждый CHECKPOINT (и WAL файл останется в pg_wal), пока не удастся (это защита от потери архивов).

Организация хранилища WAL: Это может быть локальный каталог (который резервируется, например, на другом диске), NFS-сетка, облачный бакет (тогда команда может быть s3cmd или gsutil). Главное – чтобы archive_command делала *атомарно* копирование: либо полный успех, либо файл не появится (тест на ! -f помогает не перезаписывать).

Создание базового бэкапа: Архивирование WAL, чтобы восстановиться, обычно используется совместно с **полным (base) бэкапом**. Т.е. периодически (например, раз в день/неделю) делается физический бэкап всей базы (pg_basebackup или snapshot), а затем между бэкапами все WAL собираются. В сочетании: *Base Backup + WAL archives* дают возможность PITR (point-in-time recovery).

PostgreSQL позволяет пометить **набор WAL, соответствующий бэкапу**: - Если бэкап делать `pg_basebackup`, он сам позаботится (создаст backup_label). - Если вручную/снапшотом – рекомендуется:

```
SELECT pg_start_backup('label');
-- копировать файлы PGDATA
SELECT pg_stop_backup();
```

При pg_stop_backup PostgreSQL создает `backup_label` (в PG12- <= – файл, <= PG15 – в WAL). Архивирование WAL не прекращается при этом (только метки).

Восстановление (PITR): Чтобы восстановить кластер на определённый момент, нужно: 1. Иметь подходящий **базовый бэкап** (снятый ранее). Выбираем бэкап, предшествующий желаемому моменту во времени. 2. Иметь **WAL-сегменты** начиная с момента, когда этот бэкап был сделан, до целевого времени (а лучше до немного после). - Бэкап вместе с backup_label содержит информацию, с какого LSN начать применять WAL (это обычно LSN точки start_backup, и имя timeline).

1. Подготовить восстановление:
2. Разворачиваем базовый бэкап в новый каталог PGDATA (или существующий удаляем, заменяем).
3. Удостоверяемся, что в папке нет лишних WAL (лучше очистить `pg_wal` и потом добавить нужные).

4. Копируем заархивированные WAL-сегменты (из archive) куда-то, откуда сервер сможет их брать. Есть 2 подхода: а) Копировать все нужные WAL прямо в `PGDATA/pg_wal` (тогда restore_command можно прописать как `cp %f %p` от локальной папки). б) Указать restore_command, чтобы сервер брал каждый WAL по запросу.

5. Создать **файл сигнал** для режима восстановления: В PostgreSQL 12+:

    - Создать пустой файл `recovery.signal` (для PITR, или `standby.signal` для постоянной реплики) в PGDATA. Наличие этого файла при старте говорит серверу: "мы делаем recovery". В старых версиях PG (<=11): нужно создать `recovery.conf` файл с параметрами (сейчас же `postgresql.conf` используется).

6. Указать параметры восстановления в `postgresql.conf` (или recovery.conf если старое):

    - `restore_command = 'cp /mnt/server_archives/%f "%p"'`. Эта команда будет вызвана сервером при попытке получить WAL-сегмент: `%f` заменится на имя сегмента, `%p` – на путь куда положить (в `pg_wal`). Обычно, мы копируем из архива в `pg_wal`. Она должна вернуть 0, если сегмент найден и скопирован, иначе неудача (тогда сервер попробует снова или прекратит восстановление, если сегмента нет).
    - `recovery_target_time = '2025-06-17 02:00:00+03'` например, если хотим восстановиться на конкретный момент времени (PITR). Можно вместо этого `recovery_target_xid` или `recovery_target_lsn` или `recovery_target_name` (для отметки, поставленной командой `pg_create_restore_point('name')`).
    - `recovery_target_inclusive = true/false` – включительно ли брать транзакцию, совпадающую по времени/LSN с target (по умолчанию true для time).

- Если хотим просто до конца WAL (например, после аварии) – можно не задавать recovery_target, тогда восстановится на самый последний возможный.
- `recovery_target_action = 'pause' | 'promote' | 'shutdown'` – что делать после достижения точки. По умолчанию promote (т.е. выход из recovery и продолжение работы). Если указать 'pause', сервер перейдёт в paused-режим, позволяя админу проверить состояние (можно потом продолжить). 'shutdown' – просто завершится.

Пример:

```
restore_command = 'cp /mnt/pgarchive/%f "%p"'
recovery_target_time = '2025-06-17 02:00:00+03'
recovery_target_inclusive = true
```

(Если PG ≤ 11, это записывается в recovery.conf вместе с standby_mode etc., но в ≥12 – в основной конфиг).

Также можно ограничивать timeline: - `recovery_target_timeline = 'current'` или 'latest' – обычно 'latest', чтобы если был failover – перейти на новую timeline.

7. **Закрыть доступ клиентам**: На время восстановления обычно внешний доступ не нужен, и при restore_mode сервер и так не принимает обычных коннектов (в PG12+ он работает read-only). В pg_hba можно заранее restrict.

8. **Запустить сервер** (pg_ctl start).

9. Сервер видит recovery.signal => запускается в режиме восстановления (на консоль/лог пишет: entering standby mode / starting point-in-time recovery).
10. Он прочитает `backup_label` (если есть файл, PG <15, или internal info PG15) – узнает LSN стартовый, timeline бэкапа.
11. Затем начнёт восстанавливать: *Он будет запрашивать WAL-сегменты через restore_command.*
    - Начнёт с сегмента, следующего за тем, что в backup_label (start WAL).
    - Последовательно применяет WAL записи ко всем data files.
    - Когда пройдет указанное `recovery_target_time`, он:
      - либо остановится (если action pause),
      - либо завершит recovery (применит все изменения до точки, нужные для консистентности).
    - Если `recovery_target` не задан, он будет восстанавливать пока есть WAL (restore_command возвращает success). Как только столкнётся, что следующего WAL нет в архиве (restore_command возвратит ненулевой), он поймет, что архив кончился и завершит recovery.
12. При завершении recovery:
    - Создаст **new timeline** (если был старый timeline, например, при PITR, он переключится на новый, +1).
    - Запишет соответствующую запись в WAL (timeline history).
    - Удалит (или переименует) recovery.signal,
    - Перестанет выполнять restore_command,
    - Перейдёт в обычный режим работы (listen for connections).
13. В логах будет:

```
recovery stopping at time 2025-06-17 02:00:00+03
recovery has paused (requested)
```

или

```
reached end of WAL, consistent state
selected new timeline ID: 3
database system is ready to accept connections
```

14. Теперь база данных находится в состоянии на указанный момент.

15. **Проверить результат:** Администратор может подключиться (psql) и убедиться, что данные соответствуют ожидаемому моменту (например, если мы откатывались чтобы "отменить" ошибочное удаление – проверить, что удалённые данные снова на месте). Если recovery_target_action = pause, база ждет `pg_wal_replay_resume()` (SQL) или `pg_ctl promote` (?), а может позволяеть read-only queries. Admin может решить продвинуть дальше или shutdown.

16. **Последствия:** Если мы делали PITR на новый узел/инстанс (например, не заменяя существующий продакшн, а на отдельную машину) – дальше это отдельная разветвившаяся копия. Если мы делали на месте продакшн после аварии – timeline увеличился, старый primary timeline "заморожен".

Параметры пояснение: - `restore_command` – **обязателен** для восстановления, если WAL не лежат локально. Он вызывается при каждом отсутствии сегмента в pg_wal. - Если вернуть ненулевой код – сервер попробует снова после небольшого ожидания (считая, что может сегмент появится). - Если restore_command вернул специальный код 1 (в PG doc: `return 1` for missing file) – тогда PG понимает, что сегментов больше нет (end of archive) и завершает recovery.

- `recovery_target_*`:
- Можно использовать `recovery_target_time` (читает timestamp COMMIT),
- `recovery_target_xid` (определенный XID – редко юзают),
- `recovery_target_name` (связано с `pg_create_restore_point('name')` – DBA мог заранее поставить именованную метку в WAL, например, перед критической операцией, потом можно явно сказать восстановиться к ней).

- `recovery_target_lsn` – до конкретного LSN.

- `recovery_target_action`:

- `pause` – очень удобно: сервер остановится *до* применения транзакции, наступившей после target. В payse режиме можно exam data (в read-only). Потом `pg_wal_replay_resume()` to finish (apply remaining up to target inclusive).
- `promote` (default) – сразу выйти в normal mode (commit target trans if inclusive).
- `shutdown` – просто выключится.

Пример настроек Recovery:

```
restore_command = 'cp /mnt/pgsql_archives/%f "%p"'
recovery_target_time = '2025-06-17 02:00:00'
recovery_target_inclusive = false   # до момента, не включая транзакции после
02:00:00
recovery_target_action = 'promote'
```

CAP/Caveats: - Убедиться, что archive_command работал корректно и у нас действительно есть все WAL от base backup LSN до целевого времени. Если какого-то не хватит – восстановление не сможет продолжиться (error "requested WAL segment ... is not found"). - В postgresql.conf, есть param `hot_standby = on` (default on), который разрешает read during recovery (for replica or paused recovery). - archive_mode=on + archive_command = '...' – overhead: WAL files not removed until archived, ensure archive_command is robust (maybe double-check with `archive_status` files (.done / .ready) in pg_wal). - After PITR, if we want to catch up to current: cannot continue same timeline, unless you do logic to apply more WAL if available.

В итоге: - **Archive WAL**: (wal_level=replica, archive_mode=on, archive_command set). - **Base backup**: initial full snapshot (e.g. pg_basebackup or manual + pg_start/stop_backup). - **Collect WAL files** continuously. - **To recover**: restore base backup, configure restore_command, specify point (time, XID, name, etc), start in recovery mode (recovery.signal). - **Server replays WAL** to reach that point, then stops/promotes, achieving PITR.

---

1 dbms22-p8.pdf
file://file-KKtjApXuhyyXKpbbsAVNmw