Описание технической архитектуры (версия Postgre SQL)

Оглавление

Введение.

В процессе многолетней эксплуатации Системы диспетчеризации на платформе Oracle был использован целый ряд проверенных, зарекомендовавших себя архитектурных решений, обеспечивающих эффективную работу системы и опирающихся не только на свойства СУБД «Oracle», но и на свойства аппаратуры – Oracle Exadata. При переводе Системы Диспетчеризации с платформы Oracle на платформу PostgreSQL для обеспечения эффективной работы системы потребовался пересмотр существующей архитектуры.

В данном документе описывается и обосновывается выбранная архитектура на платформе PostgreSQL.

1. Архитектурные решения и их обоснование

1.1. Предварительные работы, выполненные для обоснованного принятия архитектурных решений

№	Выполненная работа	Принятое решение
1	Измерение времени выполнения отчетов на таблицах, заполненных тестовыми данными за 1 год Целью данного тестирования являлась первая приближенная проверка «тяжелых» повторяющихся запросов системы, а также запросов, демонстрирующих использование с целью оценки работы конкретных наиболее критичных элементов системы на разных платформах. Специальным средствам оптимизации запросы не подвергались, работали штатные оптимизаторы. Тестирование проводилось на двух классах данных – аналоговые параметры и перечислимые параметры.	Обобщая результаты, можно отметить следующее: Времена выполнения запросов в целом сравнимы при условии, что проводится анализ планов выполнения запросов. На таблицах большого объема PostgreSQL несколько проигрывает, но не сильно. Результаты тестирования говорят о том, что в случаях, когда логика работы приложения определяется не дополнительными расчетами, а исключительно сравнениями с данными внутри базы, логику можно эффективно реализовать

	Выполнялись как обычные, так и групповые запросы, в том числе с использованием агрегатных функций. Подчеркнем, что целью тестирования была не оценка времени выполнения на реальном серверном оборудовании, а сравнительная оценка быстродействия конкретных запросов на разных платформах. Тестирование проводилось на следующих максимальных объемах: Перечислимые параметры - 74 814 952 400 строк; Аналоговые параметры - 999 738 400 строк	в виде хранимой процедуры, как это было сделано в АС «Диспетчеризация» на платформе Oracle. ПО модульные решения относительно распределения функций между Backend и базой данных см. п. 1.2 Количественные результаты тестирования представлены в приложении 1.
2.	**Фоновые процессы и планировщик заданий.** Параллелизм вычислений. В системе Диспетчеризации целый ряд расчетов производится по расписанию, например, в ночное время или после завершения предшествующего процесса. Эти задачи выполняются в фоновом режиме. Реализованы данные процессы с помощью пакетов dbms_job и dbms_scheduler, отсутствующих в PosgreSQL. Причем, пакет dbms_scheduler можно использовать для планирования системных сценариев, либо запуска внешних программ, что позволяет сделать систему Диспетчеризации минимально зависимой от операционной системы. Через эти же механизмы в Oracle реализованы параллельные вычисления, позволяющие резко увеличить быстродействие. Они используются, в	Для реализации параллельных вычислений были приняты следующие решения: а) установка расширения PostgreSql dbms_job (на языке Perl) б) разработана программа на plPython, позволяющая вызвать процедуру базы данных в отдельной сессии (из операционной системы).

	частности, для получения отчетов, формирования вторичных баз и материализованных представлений.	
3.	Проверка взаимодействия баз данных.	Решено отказаться от механизмов межбазового взаимодействия
4.	Фрагментация таблиц	Проведен сравнительный анализ таких решений как: TimeScaleDb, InfluxDB и встроенного в PostgreSQL. Принято решение использовать TimeScaleDb Подробнее в разделе Оптимизация для хранения данных временного ряда.
5.	**Эмулятор обработчика данных.** Целью создания данного эмулятора являлась проверка скорости обработки данных «на лету», т. е. проверка возможности обработки данных перед их закладыванием в таблицу для хранения. При создании эмулятора обработчика в первую очередь обращалось внимание на количество процедур обработки и выполнение необходимого количества обращений к таблицам нормативно-справочной информации. Целью тестирования была сравнительная оценка быстродействия конкретных запросов на разных платформах.	Эмулятор обработчика данных представляет собой развитие и конкретизацию работ, описанных в п.1 данной таблицы. В результате исследования работы эмулятора обработки данных была обоснована возможность определения состояния параметра и объекта «на лету», непосредственно в процессе получения первичных измерений при отсутствии межбазового взаимодействия (см.п.3 данной таблицы).
6.	Оптимизация с помощью использования технологии Citus	Проведя установку, настройку и тестирование, было решено использовать архитектурное решение на основе Patroni, так

		как после ряда тестов было выявлено, что Citus не подходит из-за недостаточного функционала и соответственно более плохой производительности. Более подробный сравнительный анализ описан в разделе кластерные технологии.

1.2. Помодульные решения по распределению Backend/БД

Решения о распределении нагрузки принимались для каждой задачи индивидуально с учетом ее особенностей.

№ п	Модуль	Распределение нагрузки
1	Администрирование	БД
2	Паспортизация	БД
3	Отчеты (включая ежедневный журнал)	Внешнее java-приложение
4	Матрица проблем	Backend
5	Оперативный контроль	БД
6	Аналитические задачи	Справочники - БД, расчеты - Backend.
7	Процедура расчета Тнв полигонов.	БД
9	Процедуры расчета для план-фактного анализа.	Справочники - БД, расчеты - Backend.
10	Процедура расчета суточных значений параметров.	БД
11	Процедура расчета месячных значений параметров.	БД
12	Процедура расчета расстояния до источника теплоснабжения.	Бэк-енд

1.3. Кластерные технологии

Ниже приведена таблица возможностей различных решений:

Тип	Разделяемый диск	Репликация файловой системы	Трансляция журнала предзаписи	Логическая репликация	Триггерная репликация	Репликация SQL	Асинхронная репликация	Синхронная репликация
Известные примеры	NAS	DRBD	встроенная потоковая репликация	встроенная логическая репликация, pglogical	Londiste, Slony	pgpool-II	Bucardo, citus	
Метод взаим.	разделяемые диски	дисковые блоки	WAL	логическое декодирование	Строки таблицы	SQL	Строки таблицы	Строки таблицы и блокировки строк
Не требуется специального оборудования		✓	✓	✓	✓	✓	✓	✓
Допускается несколько ведущих серверов				✓		✓	✓	✓
Нет доп. нагрузки на ведущем	✓		✓	✓		✓		

Нет задержки при нескольких серверах	✓		без синхр.	без синхр.	✓		✓	
Отказ ведущего сервера не может привести к потере данных	✓	✓	с синхр.	с синхр.		✓		✓
Сервер реплики принимает читающие запросы			с горячим резервом	✓	✓	✓	✓	✓
Репликация на уровне таблиц				✓	✓		✓	✓
Не требуется разрешение конфликтов	✓	✓	✓		✓	✓		✓

В данном документе не рассматриваются архитектуры для отказоустойчивости с использованием DRBD и NAS, т. к. реализация данных систем приближена к железному уровню и разнится от системы к системе. Стоит отметить, что данные компоненты необходимы и дополняют архитектуру, предложенную в данном документе.

Выбирая между популярными кластерными решениями на рынке, которые используются в корпоративной среде, были выделены следующие архитектуры:

- Citus — расширение с открытым исходным кодом по лицензии AGPL. Позволяет сделать «распределённый» кластер PostgreSQL, где все сервера являются ведущими и работают в режиме горячего резерва.
- Patroni — программа с открытым исходным кодом по лицензии MIT. Обеспечивает «оркестрацию» PostgreSQL, т. е. Занимается мониторингом состоянием баз данных и переключает сервера при failover.
- PgPool-II — программа с открытым исходным кодом по лицензии схожей с MIT, которая обеспечивает пул подключений, балансировку нагрузки, автоматического аварийного переключения и репликации.

Ниже приведена таблица с возможностями решений:

	Citus	Patroni	PgPool-II
Лицензия	AGPL	MIT	MIT-like
Архитектура	Extension	Orchestring	Proxy-like
Тип репликации	Репликация на основе операторов и потоковая репликация	WAL + логическая репликация	SQL

Мультимастер	✓	–	✓
Распределённая нагрузка	✓	–	✓
Переработка Backend приложения	–	✓	–
Требуется дополнительное программное обеспечение	PGBouncer	etcd, HAProxy, PGBouncer, TimeScaleDB	–
Простота настройки	~	–	–
Использование в компаниях	Cisco, Microsoft, Pex, ConvertFlow	1C, IBM Compose, TimescaleDB, Gitlab	Fujitsu
Распространён-ность	Средняя	Высокая	Низкая

По результатам таблицы и использованию решений в корпоративной среде, выбор сужается до двух решений — Patroni и Citus.

Главной проблемой при выборе из этих решений стоит надёжность системы:

- Конфликтные ситуации:

    - Для разрешения конфликтных ситуаций, Patroni использует систему etcd, используемую в k8s. Etcd — самое надёжное и протестированное временем хранилище key-value.
    - Citus так же, как и patroni, использует разрешение конфликтов, но уже со встроенным компонентом, именуемым «координирующем узлом». Поскольку citus используется для создания multimaster, такие конфликтные ситуации могут привести к катастрофическим последствиям, если такой координирующей узел содержит в себе ошибки в программном коде или же иные уязвимости.
- Распределённая нагрузка:
    - Patroni не позволяет создать multimaster, поэтому в такой архитектуре возможен единственный ведущий сервер. Использование HAProxy, обеспечивающий распределение нагрузки READ транзакций, приведёт к переработке логики обращения к базе данных на стороне приложения-Backend.
    - С другой стороны, Citus распределяет нагрузку между узлами самостоятельно, без дополнительных программ и компонентов. Стоит обратить внимание, что открытых данных с эффективностью данного распределения в открытом доступе найти не удалось.
- Простота конфигурации:
    - Patroni использует дополнительные компоненты etcd и Haproxy для обеспечения разрешения конфликтов и распределения нагрузки соответственно. В связи с этим, количество конфигураций и архитектурных решений значительно усложняется, что приводит к более сложной поддержки серверов.
    - Citus значительно проще в архитектурном плане, но требует более тонкой настройки.
- Использование в корпоративной среде:

- Архитектура Patroni уже давно используется в крупных компаниях, таких как 1C, Gitlab и IBM, в связи с чем данная архитектура является более надёжной, поскольку прошла проверку временем.
- Citus достаточно прогрессивная технология, которая должна обеспечить более надёжное хранение данных. Тем не менее, это новая технология, которая не столь широко используется среди компаний, имеет менее подробную и доступную документацию и пока что не прошла проверку временем.

Архитектура кластера мультимастер (Citus) перспективна и имеет смысл реализации в будущем, когда будут решен необходимый функционал для работы системы диспетчеризации.

Плюсы:

1. высокий уровень масштабируемости;
2. использование функционала, расширений PostgreSQL без сторонних программ;
3. распределённая нагрузка write;
4. более простая установки и поддержка.

Недостатки:

1. Экспериментальная архитектура, не распространённая в крупных компаниях;
2. Не совместим с расширением TimeScaleDB (см. раздел оптимизация для хранения данных временного ряда.)
3. Имеет ряд ограничений, связанных с основными функциями SQL: подзапросы и присоединение(join).
4. Поддерживает объединение между базами данных путем перераспределения данных в соответствии с полем "Объединение". Этот процесс называется MapMerge. Этот метод поддерживает только

естественное соединение, другие соединения по-прежнему не поддерживаются;

5. Установка citus очень проста, но для реального использования в продакшене требуются некоторые усилия. Например, как расширить емкость - это проблема, которую нельзя избежать после производства, а версия citus для сообщества не поддерживает расширение емкости.

Вывод: Принято решение использовать архитектура кластера мультирезерв (Patroni).

1.4. Обеспечение отказоустойчивости. Архитектура кластера мультирезерв

На схеме ниже показана общая архитектура системы на основе Patroni, PgBouncer и HAProxy:

Рисунок 1 Архитектура кластера мультирезерв.

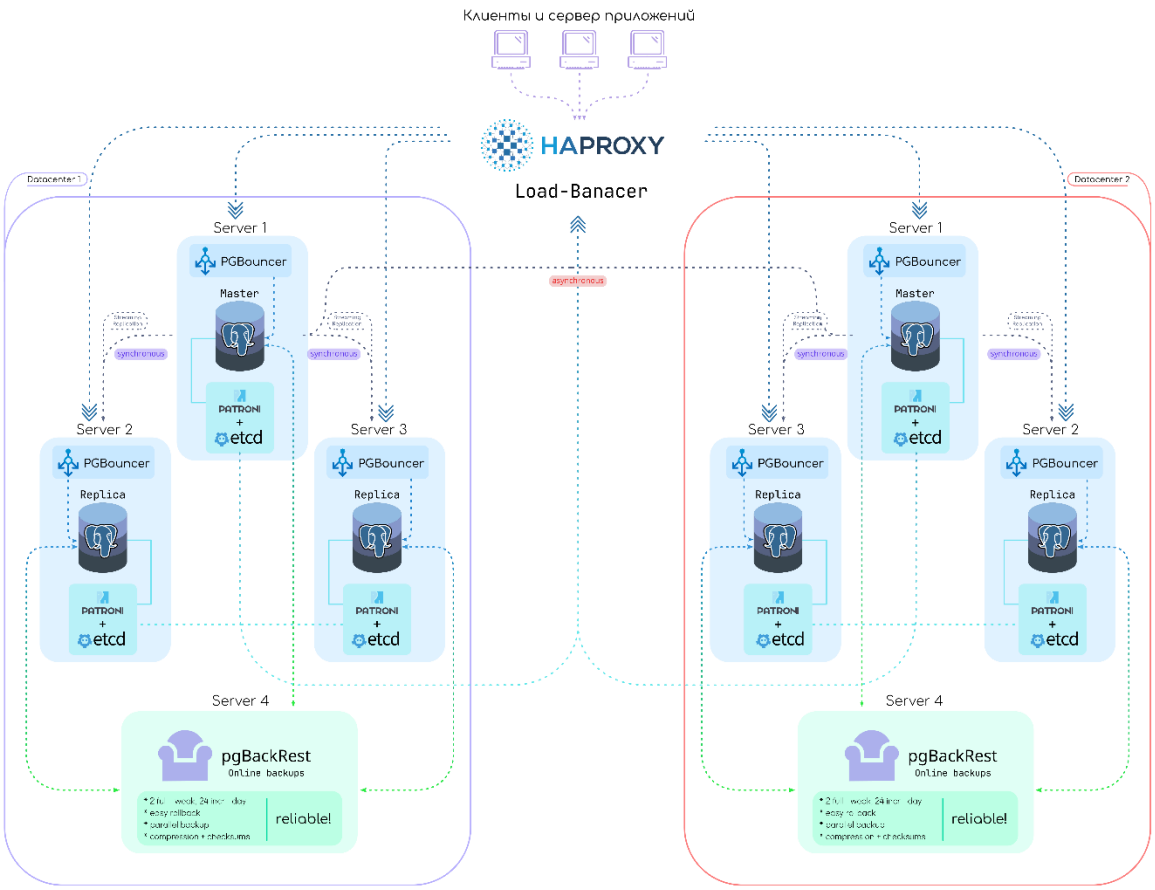


Рисунок 1 Архитектура кластера мультирезерв.

Данная архитектура состоит из следующих компонентов:

- HAProxy - серверное программное обеспечение для обеспечения высокой доступности и балансировки нагрузки для приложений, посредством распределения входящих запросов на несколько обслуживающих серверов;
- PGBouncer - программа, управляющая пулом соединений Postgres;
- Patroni – программа для автоматизации построения кластеров высокой доступности;
- etcd - надёжная и устойчивая к сбоям key-value база данных для разрешения конфликтов (определения main и replica).

Данная архитектура предполагает установку и настройку как минимум трёх серверов на каждом из двух различных дата-центрах (1 и 2). На отдельных дата-центрах (3 и 4) устанавливается HAProxy и кластер etcd. Синхронизация базы данных между серверами в одном дата-центре — синхронная, для обеспечения более надёжной целостности данных. Синхронизация баз данных между дата-центрами — асинхронная, для более быстрой передачи данных. В каждом из серверов в датацентрах 1 и 2 устанавливается PGBouncer, PostgreSQL и Patroni. В кластерах в датацентрах 3 и 4 устанавливается etcd и Haproxy. Репликация между базами данных происходит за счёт встроенной логической репликации PostgreSQL.

Ниже приведена таблица с распределением установки компонентов архитектуры на различных датацентрах и серверах:

Датацентр		Серверы				
		Haproxy	PgBouncer	PostgreSQL	Patroni	Etcd
Datacenter 1	Server 1	–	✓	✓	✓	–
	Server 2	–	✓	✓	✓	–
	Server 3	–	✓	✓	✓	–
Datacenter 2	Server 1	–	✓	✓	✓	–
	Server 2	–	✓	✓	✓	–
	Server 3	–	✓	✓	✓	–
Datacenter 3	Cluster	✓	–	–	–	–

Datacenter 4	Cluster	–	–	–	–	✓

В данной архитектуре единая точка отказа — Haproxy, поскольку к этому компоненту архитектуры идёт обращение на стороне клиентов и приложений. Так же, наиважнейшим компонентом архитектуры для работы Patroni необходимо обеспечить отказоустойчивость etcd.

Для обеспечения отказоустойчивости Haproxy, предлагается сделать системный кластер на основе k8s. Конфигурация и работа компонента k8s не будет описана в данном документе.

Для простоты разворачивания системы возможно использование Ansible — набор программных инструментов, позволяющих использовать инфраструктуру как код. Конфигурация и работа компонента Ansible не будет описана в данном документе.

1.5. Оптимизация для хранения данных временного ряда.

Для решения с хранением и оптимизацией данных временного ряда мы установили и протестировали такие решения как: TimeScaleDb, InfluxDB

TimescaleDB - база данных с открытым исходным кодом, оптимизированная для хранения данных временного ряда.

InfluxDB- система управления базами данных с открытым исходным кодом для хранения временных рядов; написана на языке Go и не требует внешних зависимостей.

Проведя сравнительные тесты TimeScaleDb, InfluxDB было принято решение использовать TimeScaleDb:

- Реализуется, как расширение PostgreSQL
- Более универсальна, чем модель InfluxDB, и обеспечивает больше функций, гибкости и контроля. Это особенно важно по мере развития системы.
- Поддерживает SQL
- Поддерживает автоматическое сегментирование
- Поддерживает автоматическое разбиение временных и пространственных измерений.
- Поддерживает несколько Параллельный запрос одного сервера и нескольких чанков, внутренняя оптимизация записи.

Характеристика	**InfluxDB**	**TimescaleDB**
Доступные ОС сервера	Linux, OS X	Linux, OS X, Windows
Схема данных	Не нужна	Нужна
Типы данных	Числа и строки	Числа, строки, логический тип данных (boolean), массивы, JSON, BLOB, геопространственные измерения, валюты, бинарные данные и другие сложные типы данных
Поддержка XML	Нет	Есть
Вторичные индексы	Нет	Есть
SQL	SQL-подобный язык	Есть

Виды API	HTTP API, JSON over UDP	Библиотека на C, потоковые API для больших объектов, ADO.NET, JDBC, ODBC
Поддерживаемые языки	.Net, Clojure, Erlang, Go, Haskell, Java, JavaScript, JavaScript (Node.js), Lisp, Perl, PHP, Python, R, Ruby, Rust, Scala	.Net, C, C++, Delphi, Java, JavaScript, Perl, PHP, Python, R, Ruby, Scheme, Tcl
Серверные скрипты	Нет	Функции пользователя, PL/pgSQL, PL/Tcl, PL/Perl, PL/Python, PL/Java, PL/PHP, PL/R, PL/Ruby, PL/Scheme, PL/Unix shell
Триггеры	Нет	Есть
Методы разделения	Шардинг	Разделение по времени и пространству атрибутов (хэшированием)
Методы репликации	Выбираемый фактор репликации	Мастер-слейв репликация с горячим чтением и резервированием на слейвах
Внешние ключи	Нет	Есть
Концепт транзакций	Нет	ACID
Контроль доступа	Просто управление с помощью аккаунтов пользователей	Детальные права доступа в соответствии с SQL стандартом

TimescaleDB реализуется как расширение PostgreSQL, и поэтому операции с данными временного ряда не будут отличаться от операций в реляционной базе данных.

Первичной точкой взаимодействия с данными является гипертаблица (hypertable). Гипертаблица – это таблица, партицированная по заданному

столбцу на временные чанки, причем каждый чанк соответствует определенному временному интервалу и области пространства. Эти разделы не пересекаются, что позволяет планировщику запросов минимизировать набор чанков, которые он должен затронуть при выполнении запроса.

Для того, чтобы обеспечить оптимальную нагрузку системы, необходимо пересоздать самые нагруженные части системы, таблицы, в гипертаблицы. Данный процесс состоит из двух этапов: создание таблицы и пересоздание таблицы в гипертаблицы.

Список гипертаблиц в БД:

dz_calc_value_past
dz_day_deviation
dz_day_deviation_obj
dz_day_deviation_struct
dz_deviation
dz_deviation_gtr
dz_deviation_obj
dz_deviation_struct
dz_eco_data_10m
dz_eco_data_20m
dz_eco_data_60m
dz_hist_data
dz_hist_data1
dz_hist_data_per
dz_hist_data_tnv
dz_hist_day_data
dz_hist_day_data_gtr
dz_hist_month_data
dz_hist_month_data_gtr
dz_hist_year_data

dz_hist_year_data_gtr

dz_notice

dz_p_hist_data

dz_pf_day_data

dz_plan_month_data

dz_pump_day_data

dz_trash_data

Проведенные тесты с гипертаблицами (смотреть приложение 2)

1.6. Применяемые программные продукты и лицензионная поддержка.

Для реализации требований ТЗ по объёмам обрабатываемой информации, быстродействию и отказоустойчивости необходимо использовать ряд свободно распространяемых программных продуктов, имеющих открытую лицензию и не запрещенных Минцифры России для применения в Российском ПО.

Продукт	Лицензия
PostgreSQL	PostgreSQL License
plpython3u	PostgreSQL License
pg_dbms_job	PostgreSQL License
TimescaleDB	Apache License 2.0
etcd	Apache License 2.0
Patroni	MIT
pgbouncer	ISC License
HAProxy	GPLv2

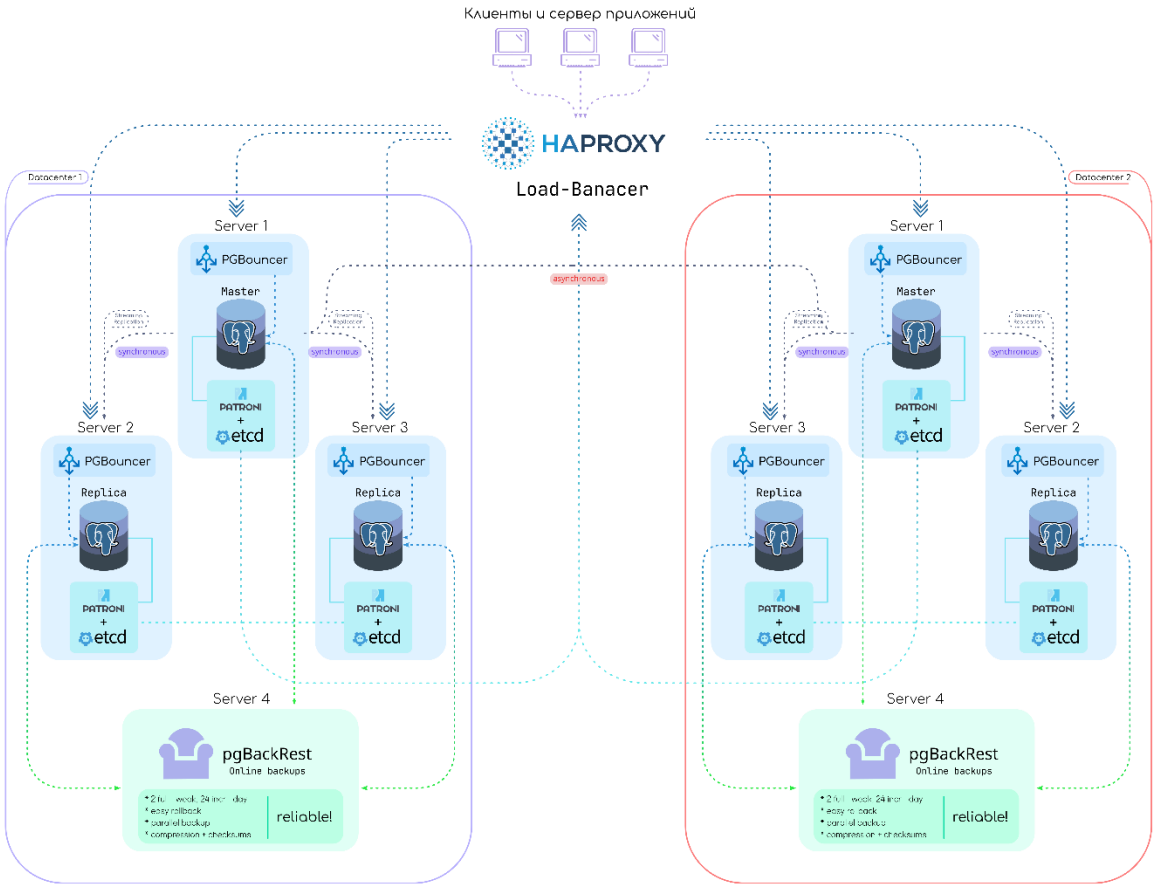
Лицензия	Описание
PostgreSQL License	Такая же, как MIT
ISC License	Такая же, как MIT
Apache License 2.0	**Разрешает:** * Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав

	Требует: * Упоминания авторства и лицензии в работе * Указывать изменения, внесённые в работу **Запрещает:** * Никаких обязательств * Никакой гарантии * Не передаются права на торговые марки
MIT	**Разрешает:** * Коммерческое использование * Распространение * Изменение * Личное использование **Требует:** * Упоминания авторства и лицензии в работе **Запрещает:** * Отказ от ответственности * Никакой гарантии
GPLv2	**Разрешает:** * Коммерческое использование * Распространение * Изменение * Личное использование * Предоставление патентных прав **Требует:** * Распространять исходный код вместе с продуктом * Упоминания авторства и лицензии в работе *Указывать изменения, внесённые в работу * Производные продукта необходимо выпускать под той же лицензией **Запрещает:** * Отказ от ответственности * Никакой гарантии

В случае положительных результатов НИОКР по выбору программной архитектуры АС «ТЕКОН-Диспетчеризация» с применением СУБД PostgreSQL планируется после завершения этапа опытной эксплуатации модуля Системы по модульная аккредитация разработанного ПО в Минцифры России как Российского программного обеспечения (https://reestr.digital.gov.ru/news/315949/).

2. Информационные потоки АС «Диспетчеризация»

Рисунок 2 Общая схема архитектуры системы.



На рис. 2 представлена общая схема архитектуры системы. Одним из важнейших процессов в системе является процесс обработки результатов измерений, приходящих с объектов. Результаты доставляются специальными драйверами client – поставщиками данных, которые вызывают функции, обеспечивающие эффективную обработку этих данных – расчет состояния параметра на основании нормативно-справочной информации, расчет вычислимых параметров, закладку исходных (измеренных) значений и определенного состояния в таблицы базы FIL – DZ_HIST_DATA, DZ_HIST_DATA_PER и др.

На основе этих данных с помощью фоновых процессов осуществляется расчет агрегированной (по времени) информации, которая в дальнейшем будет использоваться во множестве отчетов и в таких модулях, например, как «Ведомость технологических параметров». Для обработки данных в разрезе временных рядов широко используется TimeScaleDB; будучи расширением PostgreSQL, TimeScaleDB обеспечивает необходимое быстродействие при обработке больших объемов данных в разрезе временных рядов.

Пользователи взаимодействуют с системой через программы Backend, работающие на отдельных серверах приложений. В соответствии с политикой безопасности, при входе каждого пользователя в систему прежде всего проверяются права этого пользователя. За эту проверку отвечает модуль «Администрирование пользователей». Он также обеспечивает предоставление/лишение прав пользователя на работу, управление политиками безопасности.

3. Логическая модель данных

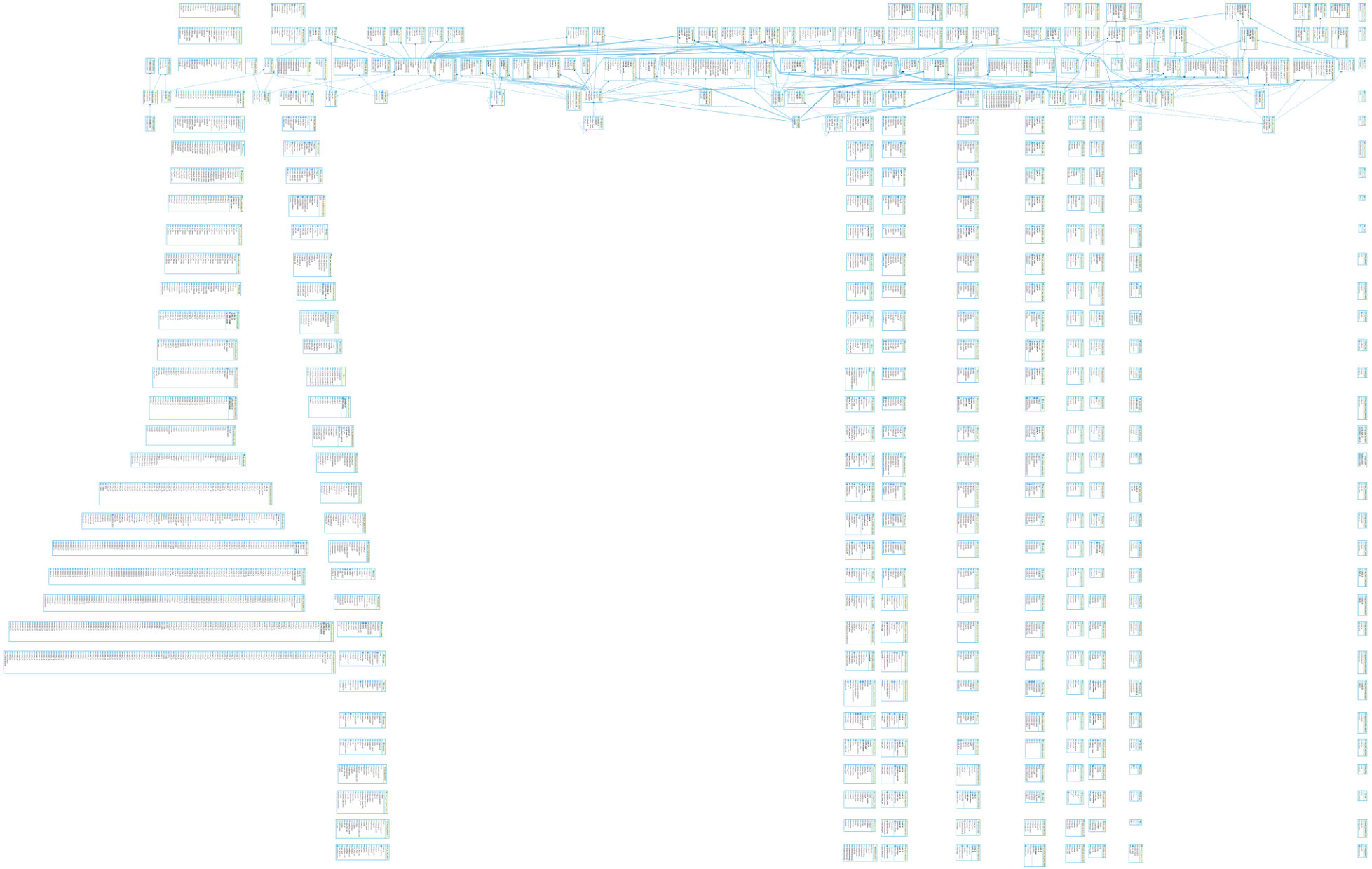


Рис. 3. Логическая модель данных (ER-диаграмма всех таблиц объединённой БД main)

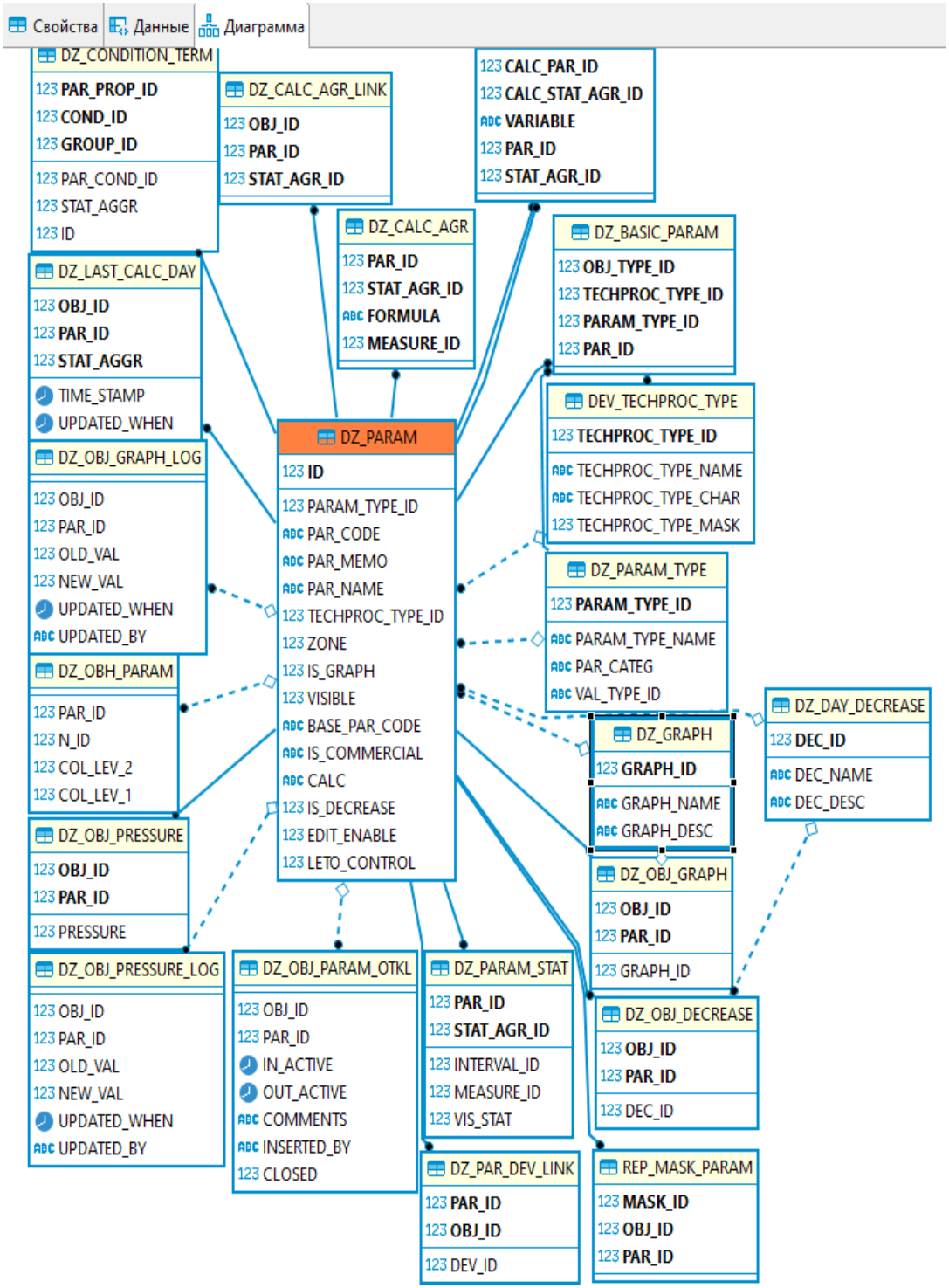


Рисунок 4. Представление таблицы DZ_PARAM и связи с другими таблицами из базы в виде ER-диаграммы.

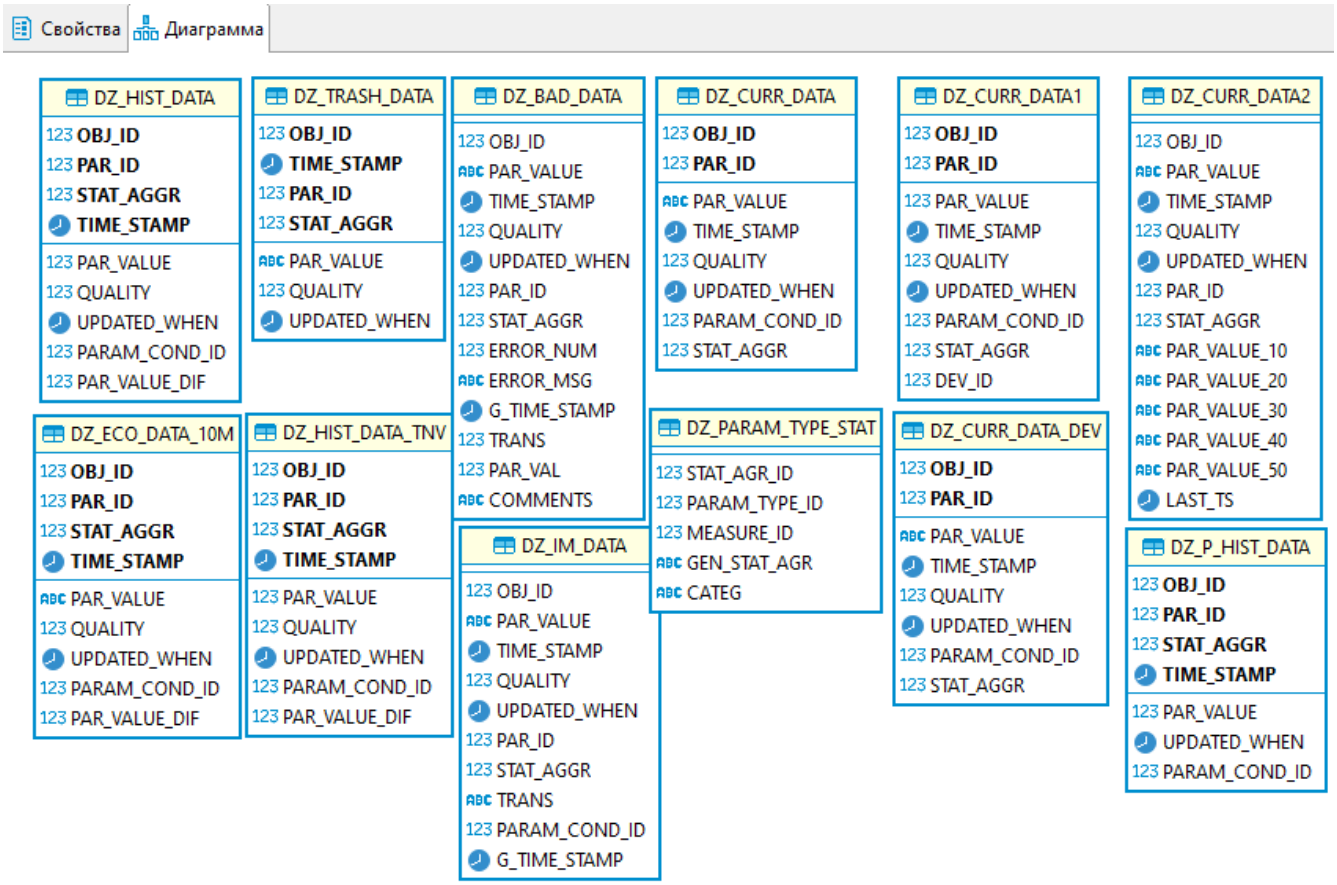


Рисунок 5 ER-диаграмма таблиц, в которые процедуры после обработки записывают данные с определенными состояниями.

4. Аппаратное обеспечение

Функции	Компьютер	Примечание
Серверы БД Postgre SQL (Pg Bouncer)	172.16.4.37, 172.16.3.32 – 172.16.4.34,	СУБД, среды разработки, расширения.
HAProxy	172.16.4.35	Балансировщик нагрузок
Сервер приложений	172.16.4.29	Back-end, Front-end
Дополнительные компьютеры	172.16.4.30-172.16.4.31, 172.16.4.36	Тестирование Citus

Для проверки и тестирования принятых решений использовался следующий аппаратный комплексы:

IP	CPU	RAM	OS
172.16.4.29	Intel xeon 2,3ghz 4 ядра	32гб	RHEL 7.9
172.16.4.30	Intel xeon 2,3ghz 16 ядер	128гб	RHEL 7.9
172.16.4.31	Intel xeon 2,3ghz 16 ядер	64гб	RHEL 7.9
172.16.4.32	Intel xeon 2,6ghz 10 ядер	32гб	Red OS 7.3.1
172.16.4.33	Intel xeon 2,6ghz 10 ядер	32гб	Red OS 7.3.1
172.16.4.34	Intel xeon 2,6ghz 10 ядер	64гб	Red OS 7.3.1
172.16.4.35	Intel xeon 2,6ghz 10 ядер	64гб	Red OS 7.3.1
172.16.4.36	Intel xeon	64гб	Red OS 7.3.1

	2,6ghz 10 ядер		
172.16.4.37	Intel xeon 2,6ghz 10 ядер	64гб	Red OS 7.3.1

БД: PostgreSql:

Установлены следующие расширения:

- pg_dbms_job - это расширение PostgreSQL для создания, управления и использования запланированных запросов DBMS_JOB.
- Plpython3u представляет реализацию PL/Python, основанную на вариации языка Python 3.
- Модуль uuid-ossp предоставляет функции для генерирования универсальных уникальных идентификаторов (UUID) по одному из нескольких стандартных алгоритмов. В нём также есть функции, выдающие специальные UUID-константы.

5. Выводы. Основные принятые решения.

На основании проведенных работ по анализу особенностей PostgreSQL были приняты следующие глобальные архитектурные решения:

- Наличие одной объединенной базы данных Main, содержащей в себе всю структуру баз: NSI (Нормативно – справочной информации), FIL (база первичных данных) и SEC (базы расчетных данных).
  Обработка первичных данных должна осуществляться «максимально близко» к нормативно-справочным данным, которые она использует. По этой причине было принято решение объединить все базы в одну.
- Таблицы (потоки данных) и программы (пакеты, библиотеки) структурируются в базах данных с помощью понятия «схема» PostgreSQL. Понятие «схема» в PostgreSQL принципиально отличается от аналогичного понятия «Oracle» и позволяет логично структурировать программы и данные.

- Доступ к данным осуществляется через процедуры и функции, логически сгруппированные в схемы PostgreSQL. Этот принцип реализует требование безопасности взаимодействия программ Backend с базами данных.
- OLAP- функции и возможности системы реализуются с помощью расширения TimeScaleDB, позволяющего в среде PostgreSQL производить эффективную обработку временных рядов данных. Это касается больших таблиц.
- Для обеспечения отказоустойчивости использовать специальное программное обеспечение (PgBouncer, Patroni, кластерная организация).

Приложение 1. Таблица результатов тестирования времени выполнения запросов

Таблица 1. Результаты сравнительного тестирования быстродействия выполнения запросов на платформах Oracle 11g и PostgreSQL 13.

<table>
<tr><td rowspan="2">Цель запроса</td><td rowspan="2">Текст запроса с комментариями</td><td colspan="2">Время выполнения запроса (в сек.)</td></tr>
<tr><td>PostgreSQL</td><td>ORACLE</td></tr>
<tr><td rowspan="8">Один параметр по одному объекту за дату</td><td colspan="2">Таблица – dz_hist_data (перечислимые параметры)</td><td colspan="2">Количество записей – 999 738 400</td></tr>
<tr><td>Объект – 17281</td><td colspan="2">Параметр объекта – 3823</td><td colspan="2">Дата – 2018-02-01 17:33:36</td></tr>
<tr><td colspan="2">select *
from admin.dz_hist_data
where UPDATED_WHEN = TO_DATE('2018-02-01 17:33:36', 'YYYY-MM-DD HH24:MI:SS') and PAR_ID = '3823' and OBJ_ID = '17281'</td><td>1,6</td><td>1,19</td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per (аналоговые параметры)</td><td colspan="2">Количество записей – 74 814 952 400</td></tr>
<tr><td>Объект – 15336</td><td colspan="2">Параметр объекта – 2078</td><td colspan="2">Дата – 2017-12-01 05:43:48</td></tr>
<tr><td colspan="2">select *
from admin.dz_hist_data_per
where UPDATED_WHEN = to_date('2017-12-01 05:43:48', 'YYYY-MM-DD</td><td>11,3</td><td>5,1</td></tr>
</table>

<table>
<tr><td></td><td colspan="3">HH24:MI:SS') and PAR_ID = '2078' and
OBJ_ID = '15336'</td><td></td><td></td></tr>
<tr><td rowspan="10">Среднее значение параметра по одному объекту</td><td rowspan="7">За дату</td><td colspan="2">Таблица – dz_hist_data
(перечислимые параметры)</td><td colspan="2">Количество записей – 999 738 400</td></tr>
<tr><td colspan="4">Параметр объекта – 1972</td></tr>
<tr><td colspan="3">SELECT EXTRACT(year FROM
time_stamp) AS "YEAR", EXTRACT(month FROM time_stamp) as "month", obj_id AS
obj, par_id AS par, AVG(par_value) as
avg_par
FROM admin.dz_hist_data
where par_id = '1972'
group BY EXTRACT(year FROM
time_stamp), EXTRACT(month FROM
time_stamp), obj_id, par_id
ORDER BY EXTRACT(month FROM
time_stamp) asc;</td><td>11,9</td><td>2,7</td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per
(аналоговые параметры)</td><td colspan="2">Количество записей – 74 814 952 400</td></tr>
<tr><td colspan="4">Параметр объекта – 2078</td></tr>
<tr><td colspan="3">SELECT EXTRACT(year FROM
time_stamp) AS "YEAR", EXTRACT(month FROM time_stamp) as "month", obj_id AS
obj, par_id AS par, AVG(par_value) as
avg_par
FROM admin.dz_hist_data_per
where par_id = '2078'
group BY EXTRACT(year FROM
time_stamp), EXTRACT(month FROM
time_stamp), obj_id, par_id
ORDER BY EXTRACT(month FROM
time_stamp) asc;</td><td>21,1</td><td>5,8</td></tr>
<tr><td rowspan="3">За период</td><td colspan="2">Таблица – dz_hist_data
(перечислимые параметры)</td><td colspan="2">Количество записей – 999 738 400</td></tr>
<tr><td>Параметр объекта – 1972</td><td>Начальная дата - 2018</td><td colspan="2">Конечная дата – 2019</td></tr>
</table>

<table>
<tr><td rowspan="5"></td><td rowspan="5"></td><td colspan="2">

```
SELECT EXTRACT(year FROM
time_stamp) AS "YEAR",
EXTRACT(month FROM time_stamp) as
"month", obj_id AS obj, par_id AS par,
AVG(par_value) as avg_par
FROM admin.dz_hist_data
where par_id = '1972' and EXTRACT(year
FROM time_stamp) between '2018' and
'2019'
group BY EXTRACT(year FROM
time_stamp), EXTRACT(month FROM
time_stamp), obj_id, par_id
ORDER BY EXTRACT(month FROM
time_stamp) asc;
```

</td><td>18,4</td><td>4,8</td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per (аналоговые параметры)</td><td colspan="2">Количество записей – 74 814 952 400</td></tr>
<tr><td>Параметр объекта – 2078</td><td>Начальная дата – 2017</td><td colspan="2">Конечная дата – 2019</td></tr>
<tr><td colspan="2">

```
SELECT EXTRACT(year FROM
time_stamp) AS "YEAR",
EXTRACT(month FROM time_stamp) as
"month", obj_id AS obj, par_id AS par,
AVG(par_value) as avg_par
FROM admin.dz_hist_data_per
where par_id = '2078' and EXTRACT(year
FROM time_stamp) between '2017' and
'2019'
group BY EXTRACT(year FROM
time_stamp), EXTRACT(month FROM
time_stamp), obj_id, par_id
ORDER BY EXTRACT(month FROM
time_stamp) asc;
```

</td><td>34,4</td><td>10,3</td></tr>
<tr><td rowspan="5">Общее количество параметров по одному объекту</td><td rowspan="5">За дату</td><td colspan="2">Таблица – dz_hist_data (перечислимые параметры)</td><td colspan="2">Количество записей – 999 738 400</td></tr>
<tr><td colspan="4">Дата – 2018</td></tr>
<tr><td colspan="2">

```
select EXTRACT(month FROM
time_stamp) as time_stamp_month,
count(distinct obj_id) as count
from admin.dz_hist_data
where extract(YEAR FROM time_stamp) =
'2018'
group by EXTRACT(month FROM
time_stamp);
```

</td><td>360</td><td>181</td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per</td><td colspan="2">Количество записей –</td></tr>
</table>

<table>
<tr><td rowspan="11"></td><td rowspan="3"></td><td colspan="2">(аналоговые параметры)</td><td colspan="2">74 814 952 400</td></tr>
<tr><td colspan="4">Дата – 2017</td></tr>
<tr><td colspan="2">select EXTRACT(month FROM time_stamp) as time_stamp_month,
count(distinct obj_id) as count
from admin.dz_hist_data_per
where extract(YEAR FROM time_stamp) = '2017'
group by EXTRACT(month FROM time_stamp);</td><td>1002</td><td>561</td></tr>
<tr><td rowspan="8">За период</td><td colspan="2">Таблица – dz_hist_data
(перечислимые параметры)</td><td colspan="2">Количество записей – 999 738 400</td></tr>
<tr><td colspan="2">Начальная дата – 2018</td><td colspan="2">Конечная дата – 2019</td></tr>
<tr><td colspan="2">select EXTRACT(month FROM time_stamp) as time_stamp_month,
count(distinct obj_id) as count
from admin.dz_hist_data
where extract(YEAR FROM time_stamp)
between '2018' and '2019'
group by EXTRACT(month FROM time_stamp);</td><td>480</td><td>260</td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per
(аналоговые параметры)</td><td colspan="2">Количество записей – 74 814 952 400</td></tr>
<tr><td colspan="2">Начальная дата – 2018</td><td colspan="2">Конечная дата – 2019</td></tr>
<tr><td colspan="2">select EXTRACT(month FROM time_stamp) as time_stamp_month,
count(distinct obj_id) as count
from admin.dz_hist_data_per
where extract(YEAR FROM time_stamp)
between '2017' and '2019'
group by EXTRACT(month FROM time_stamp);</td><td>1530</td><td>839</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td rowspan="4">Значение параметров каждого объекта за период</td><td colspan="2">Таблица – dz_hist_data
(перечислимые параметры)</td><td colspan="2">Количество записей – 999 738 400</td></tr>
<tr><td colspan="2">Начальная дата – 2018</td><td colspan="2">Конечная дата – 2019</td></tr>
<tr><td colspan="2">select EXTRACT(month FROM time_stamp) as time_stamp_month, obj_id, par_id, par_value
from admin.dz_hist_data
where extract(YEAR FROM time_stamp)
between '2018' and '2019'
group by EXTRACT(month FROM time_stamp), obj_id, par_id, par_value
ORDER BY EXTRACT(month FROM</td><td>1811</td><td>565</td></tr>
</table>

<table>
<tr><td rowspan="5"></td><td colspan="2">time_stamp) asc;</td><td></td><td></td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per
(аналоговые параметры)</td><td colspan="2">Количество записей –
74 814 952 400</td></tr>
<tr><td colspan="2">Начальная дата – 2017</td><td colspan="2">Конечная дата – 2019</td></tr>
<tr><td colspan="2">select EXTRACT(month FROM
time_stamp) as time_stamp_month, obj_id,
par_id, par_value
from admin.dz_hist_data_per
where extract(YEAR FROM time_stamp)
between '2017' and '2019'
group by EXTRACT(month FROM
time_stamp), obj_id, par_id, par_value
ORDER BY EXTRACT(month FROM
time_stamp) asc;</td><td>3200</td><td>1442</td></tr>
<tr><td colspan="5"></td></tr>
<tr><td rowspan="5">Максимальное значение параметров по всем объектам за период</td><td colspan="2">Таблица – dz_hist_data
(перечислимые параметры)</td><td colspan="2">Количество записей – 999
738 400</td></tr>
<tr><td colspan="2">Начальная дата – 2018</td><td colspan="2">Конечная дата – 2019</td></tr>
<tr><td colspan="2">select EXTRACT(month FROM
time_stamp) as time_stamp_month, obj_id,
par_id, MAX(par_value)
from admin.dz_hist_data
where extract(YEAR FROM time_stamp)
between '2018' and '2019'
group by EXTRACT(month FROM
time_stamp), obj_id, par_id
ORDER BY EXTRACT(month FROM
time_stamp) ASC, obj_id ASC, par_id asc;</td><td>1108</td><td>565</td></tr>
<tr><td colspan="2">Таблица – dz_hist_data_per
(аналоговые параметры)</td><td colspan="2">Количество записей –
74 814 952 400</td></tr>
<tr><td colspan="2">Начальная дата – 2017</td><td colspan="2">Конечная дата – 2019</td></tr>
</table>

	select EXTRACT(month FROM time_stamp) as time_stamp_month, obj_id, par_id, MAX(par_value) from admin.dz_hist_data_per where extract(YEAR FROM time_stamp) between '2017' and '2019' group by EXTRACT(month FROM time_stamp), obj_id, par_id ORDER BY EXTRACT(month FROM time_stamp) ASC, obj_id ASC, par_id asc;	2875	1430
Один параметр по всем объектам структуры за период	Таблица – dz_hist_data (перечислимые параметры)	Количество записей – 999 738 400	
	Структура - 2399	Начальная дата – 2017	Конечная дата – 2018
	select A.* from admin.dz_hist_data A where A.obj_id in (select B.obj_id from admin.obj_object B where B.struct_id = '2399') and extract(YEAR FROM A.time_stamp) between '2017' and '2018'	13,7	0,03
	Таблица – dz_hist_data_per (аналоговые параметры)	Количество записей – 74 814 952 400	
	Структура - 825	Начальная дата – 2017	Конечная дата – 2019
	select A.* from admin.dz_hist_data A where A.obj_id in (select B.obj_id from admin.obj_object B where B.struct_id = '825') and extract(YEAR FROM A.time_stamp) between '2017' and '2019'	74	11

Приложение 2. Результаты тестирование гипертаблиц.

Одинаковые таблицы с 289 408 289 строк.

	Текст запроса с комментариями	Время выполнения запроса	
		Гипертаблица	**Встроенное секционирование**
	Таблица – dz_hist_data (перечислимые параметры)	Количество полученных записей - 12	

<table>
<tr><td></td><td>

```
select EXTRACT(month FROM time_stamp)
as time_stamp_month, count(distinct
obj_id) as count

from admin.dz_hist_data

where extract(YEAR FROM time_stamp) =
'2018'

group by EXTRACT(month FROM
time_stamp);
```

</td><td>2m 54s</td><td>5m 6s</td></tr>
<tr><td rowspan="2">За период</td><td>Таблица – dz_hist_data (перечислимые параметры)</td><td colspan="2">Количество полученных записей - 200</td></tr>
<tr><td>

```
select EXTRACT(month FROM time_stamp)
as time_stamp_month, obj_id, par_id,
par_value

from admin.dz_hist_data_test

where extract(YEAR FROM time_stamp)
between '2018' and '2019'

group by EXTRACT(month FROM
time_stamp), obj_id, par_id,
par_value

ORDER BY EXTRACT(month FROM
time_stamp) asc;
```

</td><td>10m 39s</td><td>11m 23s</td></tr>
<tr><td rowspan="2"></td><td>Таблица – dz_hist_data (перечислимые параметры)</td><td colspan="2">Количество полученных записей - 200</td></tr>
<tr><td>

```
SELECT EXTRACT(year FROM time_stamp)
AS "YEAR", EXTRACT(month FROM
time_stamp) as "month", obj_id AS obj,
par_id AS par, AVG(par_value) as
avg_par
FROM admin.dz_hist_data_test

where par_id = '1972' and time_stamp >
TIMESTAMPTZ '2018-01-01' AND
time_stamp  < TIMESTAMPTZ '2019-01-01'

group BY EXTRACT(year FROM
time_stamp), EXTRACT(month FROM
time_stamp), obj_id, par_id

ORDER BY EXTRACT(month FROM
time_stamp) asc;
```

</td><td>44.2s</td><td>38.787s</td></tr>
<tr><td>последни</td><td>Таблица – dz_hist_data (перечислимые параметры)</td><td colspan="2">Количество полученных записей - 10</td></tr>
</table>

е 10 строк	`SELECT * FROM admin.dz_hist_data_test LIMIT 10;`	69ms	648ms
последние значение для каждой	Таблица – dz_hist_data (перечислимые параметры)	Количество полученных записей - 200	
	`select obj_id , last(345805, "time_stamp") FROM admin.dz_hist_data_test GROUP BY obj_id ;`	2m 59s	3m 53s
Один параметр по одному объекту за дату	Таблица – dz_hist_data (перечислимые параметры)	Количество полученных записей - нет	
	`select * from admin.dz_hist_data_test where UPDATED_WHEN = TO_DATE('2018-02-01 17:33:36', 'YYYY-MM-DD HH24:MI:SS') and PAR_ID = '3823' and OBJ_ID = '17281'`	3m 29s	3m 53s

Максимальное значение параметров по всем объектам за период	Таблица – dz_hist_data (перечислимые параметры)	Количество полученных записей - 200	
	`select EXTRACT(month FROM time_stamp) as time_stamp_month, obj_id, par_id, MAX(par_value) from admin.dz_hist_data where extract(YEAR FROM time_stamp) between '2018' and '2019' group by EXTRACT(month FROM time_stamp), obj_id, par_id ORDER BY EXTRACT(month FROM time_stamp) ASC, obj_id`	4m 16s	10m 27s

	ASC, par_id **asc**;		

Приложение 3. Инструкция установки архитектуры на основе Patroni.

Для простоты конфигурации, на серверах будет установлен следующий файл hosts:

```
/etc/hosts
```

```
172.16.4.32    etcd1   pg1
172.16.4.33    etcd2   pg2
172.16.4.34    etcd3   pg3
172.16.4.35    haproxy
```

etcd

Установка

1. Устанавливаем etcd с помощью пакетного менеджера:

```
sudo dnf install etcd
```

2. Проверяем установку:

```
etcd --version
```

Настройка первого узла

1. Используя текстовый редактор создаём новый файл настроек:

```
/etc/etcd/etcd.conf
```

```
ETCD_NAME=etcd1
ETCD_DATA_DIR=/var/lib/etcd
```

```
ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379
ETCD_LISTEN_PEER_URLS=http://0.0.0.0:2380
ETCD_ADVERTISE_CLIENT_URLS=http://etcd1:2379
ETCD_INITIAL_ADVERTISE_PEER_URLS=http://etcd1:2380
ETCD_INITIAL_CLUSTER=etcd1=http://etcd1:2380
ETCD_INITIAL_CLUSTER_STATE=new
ETCD_INITIAL_CLUSTER_TOKEN=etcd-cluster
```

- ETCD_DATA_DIR - указывает расположение каталога данных кластера
- ETCD_LISTEN_PEER_URLS - задаёт схему и точку подключения для остальных узлов кластера, по шаблону scheme://IP:port. Схема может быть http, https. Альтернатива, unix:// или unixs:// для юникс сокетов. Если в качестве IP адреса указано 0.0.0.0, то указанный порт будет прослушиваться на всех интерфейсах.
- ETCD_LISTEN_CLIENT_URLS - задаёт схему и точку подключения для клиентов кластера. В остальном совпадает с ETCD_LISTEN_PEER_URLS.
- ETCD_NAME - человекочитаемое имя этого узла кластера. Должно быть уникально в кластере. Для первого узла может быть любым. Для последующих должно совпадать с именем, указанным при добавлении узла.
- ETCD_HEARTBEAT_INTERVAL - время в миллисекудах, между рассылками лидером оповещений о том, что он всё ещё лидер. Рекомендуется задавать с учётом сетевой задержки между узлами кластера.
- ETCD_ELECTION_TIMEOUT - время в миллисекундах, которое проходит между последним принятым оповещением от лидера кластера, до попытки захватить роль лидера на ведомом узле. Рекомендуется задавать его в несколько раз большим, чем
- ETCD_HEARTBEAT_INTERVAL. Более подробно о этих параметрах можно прочесть в документации.
- ETCD_INITIAL_ADVERTISE_PEER_URLS - Список равноправных URL-адресов, по которым его могут найти остальные узлы кластера. Эти адреса используются для передачи данных по кластеру. По крайней мере, один из этих адресов должен быть маршрутизируемым для всех членов кластера. Могут содержать доменные имена. Используется только при первом запуске нового узла кластера.
- ETCD_ADVERTISE_CLIENT_URLS - Список равноправных URL-адресов, по которым его могут найти остальные узлы кластера. Эти адреса используются для передачи данных по кластеру. По крайней мере, один из этих адресов должен быть маршрутизируемым для всех членов кластера. Могут содержать доменные имена.
- ETCD_INITIAL_CLUSTER - Список узлов кластера на момент запуска. Используется только при первом запуске нового узла кластера.
- ETCD_INITIAL_CLUSTER_TOKEN - Токен кластера. Должен совпадать на всех узлах кластера. Используется только при первом запуске нового узла кластера.
- ETCD_INITIAL_CLUSTER_STATE - может принимать два значения "new" и "existing". Значение "new" используется при первом запуске первого узла в кластере. При значении "existing", узел при старте будет пытаться установить связь с остальными узлами кластера.

Файл конфигурации /etc/default/etcd используется для бутстрапа кластера etcd, т. е. параметры, описанные в нём, применяются в момент инициализации (первого запуска) процесса etcd. После того, как кластер инициализирован, конфигурация читается из рабочего каталога, заданного параметром ETCD_DATA_DIR.

2. Запускаем демон etcd:

```
sudo systemctl start etcd.service
```

3. Проверяем успешность запуска:

```
sudo systemctl status etcd.service
? etcd.service - Etcd Server
Loaded: loaded (/usr/lib/systemd/system/etcd.service; disabled; vendor preset:
disabled)
Active: active (running) since Wed 2020-03-04 07:39:43 UTC; 18s ago Main PID:
16423 (etcd)
CGroup: /system.slice/etcd.service
 ??16423 /usr/bin/etcd --name=core --data-dir=/var/lib/etcd --listen-client-
urls=http://0.0.0.0:2379 Mar 04 07:39:43 core.example etcd[16423]:
18cd9dc4a590c73e became leader at term 6
Mar 04 07:39:43 etcd[16423]: raft.node: 18cd9dc4a590c73e elected leader
18cd9dc4a590c73e at term 6
Mar 04 07:39:43 etcd[16423]: setting up the initial cluster version to 3.3
Mar 04 07:39:43 etcd[16423]: published {Name:core
ClientURLs:[http://etcd1:...eb8af0
Mar 04 07:39:43 etcd[16423]: ready to serve client requests
Mar 04 07:39:43 etcd[16423]: serving insecure client requests on [::]:2379, this
is strongly discouraged!
Mar 04 07:39:43 systemd[1]: Started Etcd Server.
```

4. Если запуск прошёл успешно, добавляем etcd.service в автозапуск:

```
sudo systemctl enable etcd.service
```

Добавление нового узла

Добавление нового узла в кластер etcd происходит в два этапа. На первом этапе кластер предупреждается о появлении нового узла. На втором запускается сам новый узел. Следующие действия необходимо последовательно выполнить на всех оставшихся серверах:

- 172.16.4.33 etcd2
- 172.16.4.34 etcd3

1. На первом узле выполняем оповещение кластера о появлении нового узла:

```
etcdctl member add etcd2 http://etcd2:2380
```

```
Added member named dbtwo with ID 871ff309aeb9cd1 to cluster ETCD_NAME="etcd2"
ETCD_INITIAL_CLUSTER="etcd2=http://etcd2:2380,etcd1=http://etcd1:2380"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

2. Устанавливаем etcd на новый сервер (см. пункт 1 в Установка).
3. Используя текстовый редактор создаём новый файл настроек:

```
ETCD_NAME=etcd3
ETCD_DATA_DIR=/var/lib/etcd
ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379
ETCD_LISTEN_PEER_URLS=http://0.0.0.0:2380
ETCD_ADVERTISE_CLIENT_URLS=http://etcd3:2379
ETCD_INITIAL_ADVERTISE_PEER_URLS=http://etcd3:2380
ETCD_INITIAL_CLUSTER=etcd3=http://etcd1:2380,etcd2=http://etcd2:2380
ETCD_INITIAL_CLUSTER_STATE=existing
ETCD_INITIAL_CLUSTER_TOKEN=etcd-cluster
```

Значение параметра "ETCD_NAME" необходимо поменять на новое имя (в данном случае имя хоста). Параметр "ETCD_INITIAL_CLUSTER" необходимо изменить на «existing», что означает наличие существующего кластера.

4. Запускаем демон etcd на новом узле:

```
sudo systemctl start etcd.service
```

5. Проверяем успешность запуска:

```
sudo systemctl status etcd.service
```

6. В списке узлов кластера новый узел должен быть в состоянии "healthy":

```
etcdctl cluster-health

member 871ff309aeb9cd1 is healthy: got healthy result from http://etcd1:2379
member 99789c1c8817dff1 is healthy: got healthy result from http://etcd2:2379
```

```
etcdctl member list

871ff309aeb9cd1: name=etcd1 peerURLs=http://etcd1:2380
clientURLs=http:etcd1:2379 isLeader=true
99789c1c8817dff1: name=etcd2 peerURLs=http://etcd2:2380
clientURLs=http://etcd2:2379 isLeader=false
```

7. Если запуск прошёл успешно, добавляем etcd.service в автозапуск:

```
sudo systemctl enable etcd.service
```

Завершение установки кластера

После установки и успешного запуска etcd на всех серверах, следует привести содержание файла /etc/etcd/etcd.conf в окончательное состояние. Для этого необходимо изменить следующие параметры в этом файле на всех серверах:

- Параметр "ETCD_INITIAL_CLUSTER" должен быть одинаковым на всех узлах:

```
ETCD_INITIAL_CLUSTER="etcd1=http://etcd1:2380,etcd2=http://etcd2:2380,etcd3=http
://etcd3:2380"
```

- Параметр "ETCD_INITIAL_CLUSTER_STATE" следует установить в значение "existing":

```
ETCD_INITIAL_CLUSTER_STATE="existing"
```

Завершение установки etcd

1. Добавим авторизацию по логину и паролю при обращениях на клиентский интерфейс etcd:

```
etcdctl user add root
New password:
User root created
```

Это первый пользователь и поэтому ему автоматически назначается роль "root":

```
etcdctl user get root

User: root
Roles: root
```

2. Включаем проверку логина и пароля:

```
etcdctl auth enable

Authentication Enabled
```

3. Проверяем, что изменения вступили в силу:

```
etcdctl --username root user get root
```

```
Password:
User: root
Roles: root
```

Полную документацию etcd можно получить по следующему адресу: `https://etcd.io/docs/`

PostgreSQL

Установка

1. Обновляем пакеты репозиториев:

```
sudo dnf update
```

2. Устанавливаем последнюю версию PostgreSQL:

```
sudo dnf install postgresql
```

Настройка

Поскольку за остановкой, запуском и конфигурацией будет отвечать Patroni, инициализировать кластер не является обходимым. Так же, запрещаем автоматический запуск PostgreSQL при старте операционной системы:

```
sudo systemctl disable postgresql-14
```

PGBouncer

Установка

1. После проведения процедур в 1 пункте Установки PostgreSQL, в операционный системе будет установлен репозиторий PostgreSQL, в который входит PGBouncer. Поэтому, устанавливаем PGBouncer с помощью пакетного менеджера:

```
sudo dnf install pgbouncer
```

2. Проверяем установленный пакет:

```
pgbouncer --version
```

Настройка

1. При большом количестве соединений необходимо увеличить лимит на количество открытых файлов в операционной системе. Для
   задания нового лимита редактируем конфигурационный файл:

```
/etc/systemd/system/pgbouncer.service.d/override.conf
```

```
[Service]

LimitNOFILE=8192
```

2. Обновляем новую конфигурацию сервиса:

```
sudo systemctl daemon-reload
```

3. Редактируем конфигурационный файл pgouncer:

```
[databases]
* = host=localhost port=5432
[pgbouncer]
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid
listen_addr = *
listen_port = 6432
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
reserve_pool_size = 10
max_client_conn = 1500
application_name_add_host = 1
server_round_robin = 1
```

```
server_reset_query =
```

Главные параметры:

- [databases] — пункт настроек подключения к базе. В данном случае выставлено значение «*», означающие, что можно подключаться ко всем базам в PostgreSQL. В переменную входят следующие значения:
  - host — адрес локального хоста
  - port — порт, на котором работает PostgreSQL
  - dbname — имя базы данных
  - auth_user — аутентификация к базе данных
- listen_port — порт, на котором будет работать pgbouncer.
- auth_type — метод аутентификации
- auth_file — файл аутентификации для подключения к базе данным
- pool_mode — метод пулинга соединений. Для эффективности обратки
- max_client_conn — максимальное количество клиентских соединений

4. Для аутентификации pgbouncer к PostgreSQL необходимо создать следующий файл с именем пользователя и паролем:

```
/etc/pgbouncer/userlist.txt
```

```
"postgres" "testtest"
"replicator" "testtest2"
"rewind_user" "testtest3"
```

5. Запускаем сервис PGBouncer:

```
systemctl start pgbouncer.service
```

6. Проверяем успешность запуска сервиса:

```
systemctl status pgbouncer.service
```

7. После проверки успешного запуска сервиса, включаем его по-умолчанию:

```
systemctl enable pgbouncer.service
```

Полную документацию pgbouncer можно получить по следующему адресу:
`https://www.pgbouncer.org/usage.html`

Patroni

Установка

1. Первоначально, устанавливаем модуль psycopg2 для корректной работы Patroni:

```
python3 -m pip install psycopg2-binary
```

2. С помощью пакетного менеджера pip, устанавливаем Patroni с модулем etcd:

```
python3 -m pip install patroni[etcd]
```

3. Проверяем установку:

```
patroni --version
```

Настройка первого узла

1. Создаём каталог настроек Patroni на 172.16.4.32:

```
sudo mkdir /etc/patroni

sudo chown postgres:postgres /etc/patroni
sudo chmod 700 /etc/patroni
```

2. Создаём файл настроек со следующей конфигурацией:

```
/etc/patroni/patroni.yml
```

```
name: pg1
scope: postgres
restapi:
 listen: 0.0.0.0:8008
 connect_address: pg1:8008
 authentication:
  username: patroni
  password: patroni
etcd3:
hosts: localhost:2379
username: root
password: testtest
bootstrap:
dcs:
 ttl: 30
 loop_wait: 10
 retry_timeout: 10
 maximum_lag_on_failover: 1048576
 master_start_timeout: 10
```

```
 postgresql:
  use_pg_rewind: true
  use_slots: true
  parameters:
   hot_standby: "on"
   wal_keep_segments: 8
   max_wal_senders: 5
   max_replication_slots: 5
   checkpoint_timeout: 30
initdb:
- auth-host: md5
- auth-local: peer
- encoding: UTF8
- data-checksums
- locale: en_US.UTF-8
pg_hba:
- local all postgres trust
- local all all md5
- host all all 0.0.0.0/0 md5
- host all all ::/0 md5
- host replication replicator samenet md5
- host replication all 127.0.0.1/32 md5
- host replication all ::1/128 md5
users:
 postgres:
  password: testtest
  options:
   - superuser
postgresql:
listen: 0.0.0.0:5432
connect_address: pg3:5432
data_dir: /data/postgres/14
config_dir: /data/postgres/14
bin_dir: /usr/pgsql-14/bin/
pgpass: /tmp/pgpass0
authentication:
  superuser:
    username: postgres
    password: testtest
  replication:
    username: replicator
    password: testtest2
  rewind:
    username: rewind_user
    password: testtest3
parameters:
  lc_time: 'en_US.UTF-8'
  lc_numeric: 'en_US.UTF-8'
  lc_monetary: 'en_US.UTF-8'
  lc_messages: 'en_US.UTF-8'
  default_text_search_config: 'pg_catalog.english'
  timezone: 'Europe/Moscow'
  datestyle: 'iso, dmy'
  dynamic_shared_memory_type: posix
```

```
  listen_addresses: '*'
  password_encryption: md5
  max_connections: 500
  shared_buffers: 5GB
  effective_cache_size: 15GB
  maintenance_work_mem: 2GB
  checkpoint_completion_target: 0.9
  wal_buffers: 16MB
  default_statistics_target: 500
  random_page_cost: 1.1
  effective_io_concurrency: 300
  work_mem: 13MB
  min_wal_size: 4GB
  max_wal_size: 16GB
  max_worker_processes: 8
  max_parallel_workers_per_gather: 4
  max_parallel_workers: 8
  max_parallel_maintenance_workers: 4
tags:
 nofailover: false
 noloadbalance: false
 clonefrom: false
 nosync: false
```

Настройки Patroni разбиты на несколько категорий:

- restapi — здесь указываются параметры Patroni, его адрес, порт и параметры аутентификации
- etcd — здесь описываются хосты с etcd
- bootstrap — здесь происходит первичная установка и конфигурации узла Patroni к PostgreSQL
- postgresql — здесь описывается подключение к базе данным, параметры аутентификации и директории
- tags — здесь описываются параметры тэгов, которые влияют на работу категории restapi (то есть на работу всего узла Patroni)

В приведённом примере настроек, есть ряд параметров, влияющих на выполнение переключения на резервный сервер.

- maximum_lag_on_failover - максимальное количество байт, на которые может отставать резервный сервер от ведущего, участвующий в выборах нового лидера.
- master_start_timeout - задержка в секундах, между обнаружением аварийной ситуации и началом отработки переключения на резервный сервер. По умолчанию 300 секунд. Если задано 0, то переключение начнётся немедленно. При использовании асинхронной репликации (как в приведённом примере) это может привести к потере последних транзакций. Максимальное время переключения на реплику равно "loop_wait" + "master_start_timeout" + "loop_wait". Если "master_start_timeout" установленно в 0, то это время становится равно значению параметра "loop_wait".
- nofailover - значение "true" запрещает выбирать этот узел в качестве ведущего

- clonefrom - значение "true" рекомендует выбирать этот узел для создания резервной копии при развёртывании нового узла Patroni. Если значение "true" установлено у нескольких узлов, будет случайным образом выбран один из них
- noloadbalance - устанавливает HTTP код возврата 503 для запроса GET /replica REST API
- nosync - значение true запрещает выбирать этот узел для синхронной репликации

3. Создаём сервис для запуска демона Patroni:

```
/etc/systemd/system/patroni.service
```

```
[Unit]
Description=Runners to orchestrate a high-availability PostgreSQL
After=syslog.target network.target
[Service]
Type=simple
User=postgres
Group=postgres
ExecStart=/usr/local/bin/patroni /etc/patroni/patroni.yml
ExecReload=/bin/kill -s HUP $MAINPID
KillMode=process
TimeoutSec=30
Restart=no
[Install]
WantedBy=multi-user.target
```

В приведённых настройках запуска особенно важны два параметра:

- TimeoutSec - время в секундах, которое система будет ожидать при запуске и остановке сервиса, перед тем как произвести попытку его внештатного завершения.
- Restart - может принимать значения: no, on-success, on-failure, on-abnormal, on-watchdog, on-abort, или always. Определяет политику перезапуска сервиса в случае, если он завершает работу не по команде от systemd.

4. Обновляем системные настройки:

```
sudo systemctl daemon-reload
```

5. Проверяем успешность запуска:

```
sudo systemctl status patroni.service

? patroni.service - Runners to orchestrate a high-availability PostgreSQL
Loaded: loaded (/etc/systemd/system/patroni.service; disabled; vendor preset:
disabled)
Active: active (running) since Wed 2020-03-04 09:22:32 UTC; 7s ago
```

```
Main PID: 19572 (patroni)
CGroup: /system.slice/patroni.service
 ??19572 /usr/bin/python3 /usr/local/bin/patroni /etc/patroni/patroni.yml
 ??19581 /usr/pgsql-11/bin/pg_ctl initdb -D /var/lib/pgsql/11/data -o --
encoding=UTF8 --data-checksums --locale=ru_RU.UTF-8 --use...
 ??19583 /usr/pgsql-11/bin/initdb -D /var/lib/pgsql/11/data --encoding=UTF8 --
data-checksums --locale=ru_RU.UTF-8 --username=post...
Mar 04 09:22:33 dbone.example patroni[19572]: Data page checksums are enabled.
Mar 04 09:22:33 dbone.example patroni[19572]: fixing permissions on existing
directory /var/lib/pgsql/11/data ... ok
Mar 04 09:22:33 dbone.example patroni[19572]: creating subdirectories ... ok
Mar 04 09:22:33 dbone.example patroni[19572]: selecting default max_connections
... 100
Mar 04 09:22:33 dbone.example patroni[19572]: selecting default shared_buffers
... 128MB
Mar 04 09:22:33 dbone.example patroni[19572]: selecting default timezone ... UTC
Mar 04 09:22:33 dbone.example patroni[19572]: selecting dynamic shared memory
implementation ... posix
Mar 04 09:22:33 dbone.example patroni[19572]: creating configuration files ...
ok
Mar 04 09:22:33 dbone.example patroni[19572]: running bootstrap script ... ok
Mar 04 09:22:34 dbone.example patroni[19572]: performing post-bootstrap
initialization ... ok
```

6. Если запуск прошёл успешно, добавляем patroni.service в автозапуск:

```
sudo systemctl enable patroni.service
```

Настройка patronictl

1. Создаём файл конфигурации patronictl со следующим содержимым:

~/.config/patroni/patronictl.yaml

```
dcs_api:
 etcd://localhost:2379
namespace: /db/
scope: postgres
authentication:
 username: patroni
 password: patroni
```

2. Проверяем результат выполненных настроек:

```
patronictl list
+--------+------+---------+---------+----+-----------+
| Member | Host | Role    | State   | TL | Lag in MB |
+ Cluster: postgres (7146211112078719817) -----------+
```

```
| pg1    | pg1  | Leader  | running |  3 |           |
+--------+------+---------+---------+----+-----------+
```

Добавление нового узла

Данную настройку нужно повторить на каждом сервере:

- 172.16.4.33 pg2
- 172.16.4.34 pg3

Для добавления нового узла в кластер Patroni, выполняются действия из раздела "Настройка первого узла". Различия заключаются в замене "dbone" на имя текущего сервера в файле конфигурации. Поэтому для pg2 содержимое будет следующим:

```
/etc/patroni/patroni.yml
```

```
name: pg2
scope: postgres
restapi:
 listen: 0.0.0.0:8008
 connect_address: pg2:8008
 authentication:
  username: patroni
  password: patroni
etcd3:
hosts: localhost:2379
username: root
password: testtest
bootstrap:
dcs:
 ttl: 30
 loop_wait: 10
 retry_timeout: 10
 maximum_lag_on_failover: 1048576
 master_start_timeout: 10
 postgresql:
  use_pg_rewind: true
  use_slots: true
  parameters:
   hot_standby: "on"
   wal_keep_segments: 8
   max_wal_senders: 5
   max_replication_slots: 5
   checkpoint_timeout: 30
initdb:
- auth-host: md5
- auth-local: peer
- encoding: UTF8
- data-checksums
- locale: en_US.UTF-8
pg_hba:
- local all postgres trust
- local all all md5
```

```
- host all all 0.0.0.0/0 md5
- host all all ::/0 md5
- host replication replicator samenet md5
- host replication all 127.0.0.1/32 md5
- host replication all ::1/128 md5
users:
 postgres:
  password: testtest
  options:
   - superuser
postgresql:
listen: 0.0.0.0:5432
connect_address: pg2:5432
data_dir: /data/postgres/14
config_dir: /data/postgres/14
bin_dir: /usr/pgsql-14/bin/
pgpass: /tmp/pgpass0
authentication:
  superuser:
    username: postgres
    password: testtest
  replication:
    username: replicator
    password: testtest2
  rewind:
    username: rewind_user
    password: testtest3
parameters:
  lc_time: 'en_US.UTF-8'
  lc_numeric: 'en_US.UTF-8'
  lc_monetary: 'en_US.UTF-8'
  lc_messages: 'en_US.UTF-8'
  default_text_search_config: 'pg_catalog.english'
  timezone: 'Europe/Moscow'
  datestyle: 'iso, dmy'
  dynamic_shared_memory_type: posix
  listen_addresses: '*'
  password_encryption: md5
  max_connections: 500
  shared_buffers: 5GB
  effective_cache_size: 15GB
  maintenance_work_mem: 2GB
  checkpoint_completion_target: 0.9
  wal_buffers: 16MB
  default_statistics_target: 500
  random_page_cost: 1.1
  effective_io_concurrency: 300
  work_mem: 13MB
  min_wal_size: 4GB
  max_wal_size: 16GB
  max_worker_processes: 8
  max_parallel_workers_per_gather: 4
  max_parallel_workers: 8
  max_parallel_maintenance_workers: 4
```

```
tags:
 nofailover: false
 noloadbalance: false
 clonefrom: false
 nosync: false
```

После запуска Patroni на резервном сервере выполняются следующие действия в автоматическом порядке:

- Patroni подключается к кластеру на pg1
- Создаётся новый кластер PostgreSQL и заполняется данными с pg1
- Новый кластер PostgreSQL переводится в "slave mode"

В результате, в кластере Patroni должно быть шесть узлов:

```
patronictl list
+--------+------+---------+---------+----+-----------+
| Member | Host | Role    | State   | TL | Lag in MB |
+ Cluster: postgres (7146211112078719817) -----------+
| pg1    | pg1  | Leader  | running |  3 |           |
| pg2    | pg2  | Replica | running |  3 |         0 |
| pg3    | pg3  | Replica | running |  3 |         0 |
+--------+------+---------+---------+----+-----------+
```

Изменение настроек PostgreSQL через Patroni

Так как PostgreSQL в данной архитектуре управляется Patroni, то настройки PostgreSQL задаются через конфигурационный файл Paroni. Рекомендуется поддерживать данные настройки одинаковыми на всех узлах. Для задания настроек PostgreSQL используется параметр "parameters" в секции "postgresql" файла /etc/patroni/patroni.yml:

```
parameters:
 unix_socket_directories: '/var/run/postgresql/'
```

Следующие действия выполняются на серверах PostgreSQL + Patroni:

1. Приводим "parameters" к следующему виду:

```
parameters:
  lc_time: 'en_US.UTF-8'
  lc_numeric: 'en_US.UTF-8'
  lc_monetary: 'en_US.UTF-8'
  lc_messages: 'en_US.UTF-8'
  default_text_search_config: 'pg_catalog.english'
  timezone: 'Europe/Moscow'
  datestyle: 'iso, dmy'
  dynamic_shared_memory_type: posix
  listen_addresses: '*'
  password_encryption: md5
```

```
max_connections: 500
shared_buffers: 5GB
effective_cache_size: 15GB
maintenance_work_mem: 2GB
checkpoint_completion_target: 0.9
wal_buffers: 16MB
default_statistics_target: 500
random_page_cost: 1.1
effective_io_concurrency: 300
work_mem: 13MB
min_wal_size: 4GB
max_wal_size: 16GB
max_worker_processes: 8
max_parallel_workers_per_gather: 4
max_parallel_workers: 8
max_parallel_maintenance_workers: 4
```

Значения параметров приведены только в качестве примера задания значений для этих параметров. Для определения значений параметров следует обратится к полной документации PostgreSQL: https://www.postgresql.org/docs/current/runtime-config.html

2. Применяем настройки:

```
patronictl reload postgres pg1
patronictl restart postgres pg1
patronictl reload postgres pg2
patronictl restart postgres pg2
patronictl reload postgres pg3
patronictl restart postgres pg3
```

3. Проверяем изменения настроек:

```
psql -U postgres -d postgres
```

Полную документацию Patroni можно получить по следующему адресу:
`https://patroni.readthedocs.io/en/latest`

Haproxy

Установка

1. Устанавливаем Haproxy из пакетного менеджера на сервере 172.16.4.35:

```
sudo dnf install haproxy
```

2. Проверяем установленный пакет Haproxy:

```
haproxy --version
```

Настройка

1. Создаём конфигурационный файл Haproxy со следующим содержимым:

```
/etc/haproxy/haproxy.cfg
```

```
global
 maxconn 4500
defaults
 log global
 mode tcp
 retries 2
 timeout client 30m
 timeout connect 4s
 timeout server 30m
 timeout check 5s
listen stats
 mode http
 bind *:7000
 stats enable
 stats uri /
listen postgres
 bind *:6432
 option httpchk
 http-check expect status 200
 default-server inter 3s fastinter 1s fall 2 rise 2 on-marked-down shutdown-sessions
 server pg1 pg1:6432 maxconn 1500 check port 8008
 server pg2 pg2:6432 maxconn 1500 check port 8008
 server pg3 pg3:6432 maxconn 1500 check port 8008
```

- В директиве Global описываются глобальные параметры системы Haproxy.
- В defaults задаются параметры подключения по-умолчанию.
- Stats включает в себя статистику для дальнейшего анализа и отображения (Prometheus + Grafana).
- Listen postgres — основная директива, которая обеспечивает мониторинг хостов Patroni.

Поскольку данная конфигурация контролируется Patroni, а не системным администратором, она является динамически изменяемой и не требует дополнительного описания настройки.

2. Запускаем демон Haproxy:

```
sudo systemctl start haproxy.service
```

3. Проверяем результат запуска:

```
sudo systemctl status haproxy.service
```

Поверяем в интерфейсе haproxy (http://haproxy:7000/), что узлы кластера Patroni найдены и их роли определены корректно. Ведущий сервер будет отмечен как "UP", резервные как "DOWN".

4. Если запуск прошёл успешно, добавляем haproxy.service в автозапуск:

```
sudo systemctl enable haproxy.service
```

Полную документацию HAProxy можно получить по следующему адресу: https://www.haproxy.com/documentation/hapee/latest/onepage/