

ЗАЩИЩЕННАЯ СИСТЕМА УПРАВЛЕНИЯ
БАЗАМИ ДАННЫХ «ЈАТОВА»

Руководство по настройке. Часть 11.
Высокопроизводительный кластер.
Компонент «ja_Hipe_Cluster»

643.72410666.00067-07 98 01-11

Листов 113

Инв. № подл.	Подп. и дата	Взам. инв. №	Инв. № дубл.	Подп. и дата

АННОТАЦИЯ

Администратор СУБД «Jatoba» должен иметь навыки по работе с системами управления базами данных (СУБД) PostgreSQL или защищенной СУБД «Jatoba» (ООО «Газинформсервис»).

Для СУБД «Jatoba» версии ядра 4 версия компонента — 11.0.4

Для СУБД «Jatoba» версии ядра 5/6 версия компонента — 12.1.1

Руководство содержит следующие разделы:

- Раздел 1, в котором приведены назначение и функции СУБД «Jatoba» и требования к среде функционирования СУБД;
- Раздел 2 содержит описание использования компонента;
- Раздел 3 содержит описание установки и настройки компонента;
- Раздел 4 описывает функциональные возможности компонента;
- Раздел 6 описывает обновление и удаление компонента;
- В Приложении 1 приведен пример установки СУБД «Jatoba» из локального репозитория для ОС Ubuntu 20.04;
- В Приложении 2 приведена инструкция по восстановлению данных при помощи компонента pg_ProBackup;
- В Приложении 3 приведена инструкция по настройке SSL соединения для ja_Hipe_Cluster.



Все примеры в данном документе приведены для СУБД «Jatoba» версии ядра 4.x, для других версий все шаги выполняются аналогично, разница состоит в именах директорий.

Например, СУБД «Jatoba» версии 6.x по умолчанию устанавливается в директорию:

- ОС Windows – «C:\Program Files\GIS\Jatoba\6\bin»;
- ОС Linux – «/usr/jatoba-6/bin».



Важная информация

Для сертифицированной версии СУБД «Jatoba» поддерживается работа только на ОС, указанных в формуляре на поставку!

Степени важности примечаний, применяемые в документе:



Важная информация – указания, требующие особого внимания



Дополнительная информация – указания, позволяющие упростить работу с изделием

СОДЕРЖАНИЕ

1. Назначение компонента.....	6
1.1. Функциональные возможности	6
1.2. Условия применения.....	7
1.3. Требования к среде функционирования	7
2. Использование	8
2.1. Общая схема работы	8
2.1.1. Узлы.....	9
2.1.2. Координатор и рабочие узлы	9
2.1.3. Распределенные данные.....	9
2.1.4. Структура таблицы метаданных pg_dist_shard.....	11
2.1.5. Размещение сегментов	12
2.1.6. Совместное размещение.....	13
2.1.7. Параллелизм	13
2.1.8. Выполнение запроса.....	13
2.1.9. Добавление координатора.....	15
2.1.10. Обработка аварийных ситуаций.....	15
3. Установка и настройка компонента	18
3.1. Установка компонента	18
3.2. Настройка сервера «Coordinator».....	19
3.2.1. Редактирование конфигурационного файла «postgresql.conf».....	19
3.2.2. Редактирование конфигурационного файла «pg_hba.conf»	20
3.2.3. Создание БД «citus_db».....	22
3.2.4. Установка расширения «citus»	22
3.3. Настройка сервера «Node1».....	23
3.4. Настройка сервера «Node2».....	24
3.5. Настройка кластера	25
3.5.1. Добавление информации об узле координатора.....	25
3.5.2. Добавление узлов кластера	27
3.5.3. Использование режима preferred для запросов только на чтение к узлам кластера.....	28
3.6. Установка кластера без SU	32
4. Функции компонента.....	34
4.1. Создание и модификация распределенных объектов	34
4.2. Справочные таблицы	35
4.3. Распределение данных координатора.....	36
4.4. Совместное размещение таблиц	37
4.4.1. Создание совместно размещенных таблиц.....	37

4.4.2. Выбор столбца распределения	39
4.5. Удаление таблиц	43
4.6. Изменение таблиц	43
4.7. Добавление / изменение столбцов	43
4.8. Добавление/удаление ограничений	44
4.9. Использование ограничений “NOT VALID”	47
4.10. Добавление / удаление индексов	48
4.11. Типы и функции	49
4.12. Служебные функции ja_Hipe_Cluster	51
4.12.1. Таблица и сегмент DDL.....	51
4.12.2. Метаданные / Информация о конфигурации.....	64
4.12.3. Функции управления кластером и восстановления.....	79
5. Чтение с реплик. Чтение с реплик с учетом геозоны.....	90
5.1. Описание функциональности	90
5.2. Проблемы и решения	91
5.2.1. Транзакция READ ONLY	91
5.2.2. Переадресация	92
5.2.3. Поведение при отказе реплики	92
5.2.4. Балансировка с учетом «зон доступности»	93
5.2.5. Проблема пула соединений.....	93
5.2.6. Балансировка на несколько узлов	93
5.2.7. Балансировка при локальном чтении.....	94
6. Обновление и удаление компонента	95
7. Сообщения об ошибках	96
7.1. Ошибка работы ja_hipe_cluster и pg_task на воркере.....	96
Приложение 1	97
Пример установки СУБД «Jatoba» из локального репозитория для ОС Ubuntu 20.04	97
Приложение 2	103
Восстановление данных при помощи компонента pg_ProBackup.....	103
Приложение 3	107
Настройка SSL соединения для ja_Hipe_Cluster	107
Термины и определения	111
Перечень сокращений.....	112

1. НАЗНАЧЕНИЕ КОМПОНЕНТА

Компонент `ja_Hipe_Cluster` — расширение СУБД «Jatoba», которое позволяет обычным серверам баз данных (называемым узлами) координировать свои действия друг с другом в архитектуре «ничего общего» («shared nothing»). Узлы образуют кластер, который позволяет СУБД хранить больше данных и использовать больше ядер центрального процессора, чем это было бы возможно на одном компьютере. Эта архитектура также позволяет масштабировать базу данных, просто добавляя дополнительные узлы в кластер. Данное расширение позволяет выполнять распределение таблиц и запросов по рабочим узлам, входящим в кластер.

Назначением данного компонента является создание многоцелевой базы данных, способной масштабировать как транзакционные рабочие нагрузки за счет маршрутизации и делегирования транзакций в кластере серверов СУБД «Jatoba», так и аналитические операции за счет их распараллеливания в кластере.

1.1. Функциональные возможности

Компонент обладает следующими функциональными возможностями:

- таблицы могут быть распределены между рабочими узлами кластера СУБД «Jatoba» для объединения их ресурсов: процессоров, памяти и хранилищ;
- справочные таблицы реплицируются на все узлы для соединений и внешних ключей из распределенных таблиц и максимальной производительности чтения;
- механизм распределенных запросов маршрутизирует и распараллеливает SELECT, DDL и другие операции над распределенными таблицами в кластере;
- справочные таблицы используются для хранения данных, к которым необходимо часто обращаться нескольким узлам кластера;
- колоночное хранилище сжимает данные, ускоряет сканирование и поддерживает быстрое отображение как в обычных, так и в распределенных таблицах;
- запрос с любого узла позволяет использовать все возможности кластера для распределенных запросов.

1.2. Условия применения

Компонент ja_Hipe_Cluster может использоваться совместно с СУБД «Jatoba», а также открытой СУБД PostgreSQL.

1.3. Требования к среде функционирования

Компонент функционирует под управлением ОС, указанных в таблице 1.1.

Таблица 1.1 – Поддерживаемые операционные системы

№	Наименование ОС	Сертификат ФСТЭК	
		№ серт.	Дата выдачи
1	Windows 10	—	—
2	Windows 11	—	—
3	Windows Server 2016	—	—
4	Windows Server 2019	—	—
5	Windows Server 2022	—	—
6	Astra Linux 1.7 Special Edition Смоленск (x86-64)	2557	30.01.2012
7	Astra Linux 1.8 (x86-64)	—	—
8	Astra Linux 2.12 Common Edition Орел (x86-64)	—	—
9	Debian 10	—	—
10	Debian 11	—	—
11	Debian 12	—	—
12	Альт 8 СП	3866	10.08.2018
13	Альт 10 СП	3866	10.08.2018
14	Альт 9.1 Server	—	—
15	Альт 10 Server	—	—
16	Ubuntu 20.04	—	—
17	Ubuntu 22.04	—	—
18	Ubuntu 24.04	—	—
19	ОСНОВА2	—	—
20	РЕД ОС 7.3 Муром	4060	12.01.2019
21	РЕД ОС 8	—	—
22	РОСА 7.9	—	—
23	РОСА 12.4	—	—
24	RedHat Enterprise Linux 8	—	—
25	Oracle Linux 8.4	—	—

2. ИСПОЛЬЗОВАНИЕ

2.1. Общая схема работы

ja_Hipe_Cluster превращается из одного узла СУБД «Jatoba» в кластер баз данных путем добавления рабочих узлов. В кластере исходный узел, к которому подключается приложение, называется координационным узлом. Координатор ja_Hipe_Cluster содержит как метаданные распределенных таблиц, так и справочные таблицы, а также обычные (локальные) таблицы, последовательности и другие объекты базы данных.

Данные в распределенных таблицах хранятся в «сегментах», которые на самом деле являются обычными таблицами СУБД «Jatoba» на рабочих узлах. При запросе распределенной таблицы на узле-координаторе ja_Hipe_Cluster будет отправлять регулярные SQL-запросы на рабочие узлы (рисунок 2.1). Таким образом, все обычные оптимизации и расширения СУБД «Jatoba» могут автоматически использоваться с ja_Hipe_Cluster.

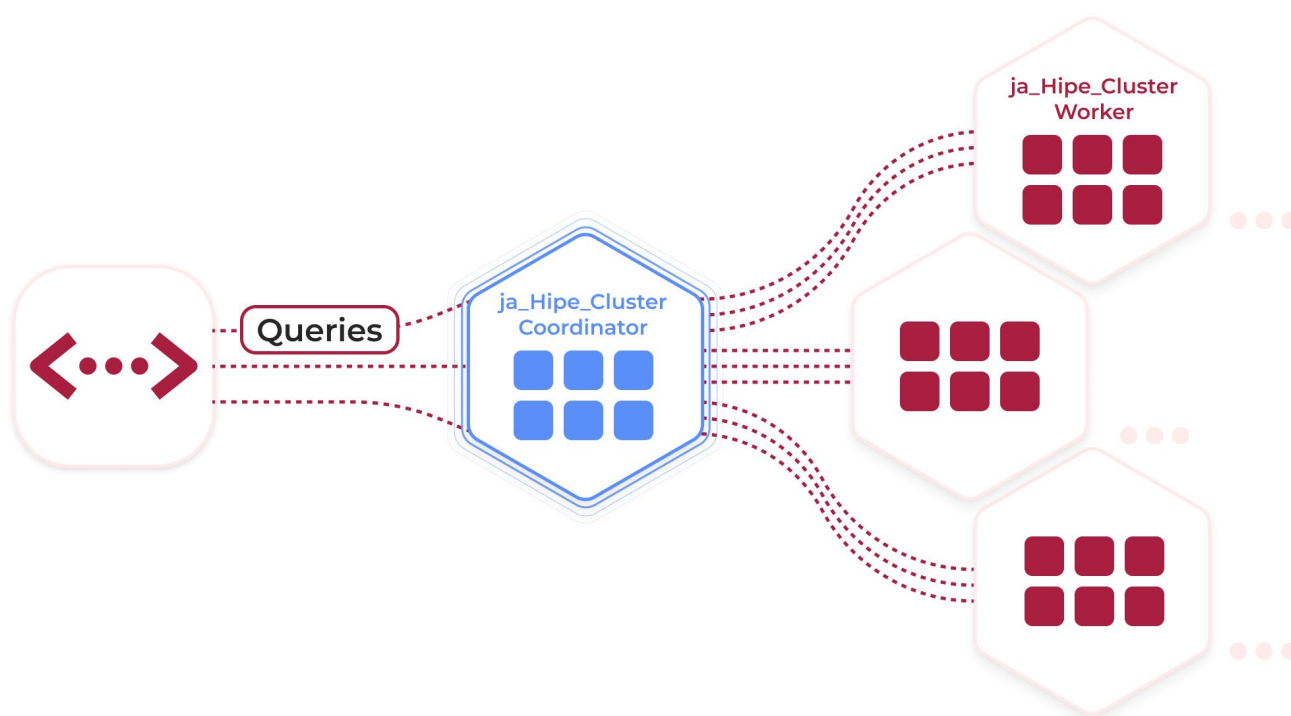


Рисунок 2.1 – Отправка запросов на рабочие узлы

При отправке запроса, в котором все (совместно расположенные) распределенные таблицы имеют одинаковый фильтр в столбце распределения, ja_Hipe_Cluster автоматически обнаружит это и отправит весь запрос на рабочий узел, который хранит данные. Таким образом, поддерживаются произвольно сложные запросы с минимальными

затратами на маршрутизацию, что особенно полезно для масштабирования транзакционных рабочих нагрузок. Если запросы не имеют определенного фильтра, каждый сегмент запрашивается параллельно, что особенно полезно в аналитических рабочих нагрузках. Распределенный исполнитель ja_Hipe_Cluster является адаптивным и предназначен для одновременной обработки обоих типов запросов в одной системе с высоким уровнем параллелизма, что позволяет выполнять крупномасштабные смешанные рабочие нагрузки.

Схема и метаданные распределенных и справочных таблиц автоматически синхронизируются со всеми узлами в кластере, поэтому можно подключиться к любому узлу для выполнения распределенных запросов. Изменения схемы и администрирование кластера должны проходить через координатора.

2.1.1. Узлы

ja_Hipe_Cluster – это расширение СУБД «Jatoba», которое позволяет серверам баз данных (называемыми узлами) координировать свои действия друг с другом. Узлы образуют кластер, который позволяет СУБД хранить больше данных и использовать больше процессорных ядер, чем это было бы возможно на одном компьютере. Эта архитектура также позволяет масштабировать базу данных, просто добавляя больше узлов в кластер.

2.1.2. Координатор и рабочие узлы

У каждого кластера есть один специальный узел, называемый координатором (остальные – рабочие узлы). Приложения отправляют свои запросы на узел координатора, который передает их соответствующим рабочим узлам и накапливает результаты.

Для каждого запроса координатор либо направляет его на один рабочий узел, либо распараллеливает его по нескольким в зависимости от того, находятся ли требуемые данные на одном узле или на нескольких. Для этой задачи на координаторе хранятся таблицы с метаданными. Эти таблицы, специфичные для ja_Hipe_Cluster, отслеживают DNS-имена и работоспособность рабочих узлов, а также распределение данных по узлам.

2.1.3. Распределенные данные

В кластере ja_Hipe_Cluster существует три типа таблиц, каждая из которых используется для разных целей.

– Тип 1: распределенные таблицы

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Распределенные таблицы выглядят как обычные таблицы для операторов SQL, но их данные распределены между рабочими узлами. Эти сегменты представляют собой обычные таблицы СУБД «Jatoba», которые могут иметь индексы, внешние ключи, ограничения, пользовательские типы (рисунок 2.2).

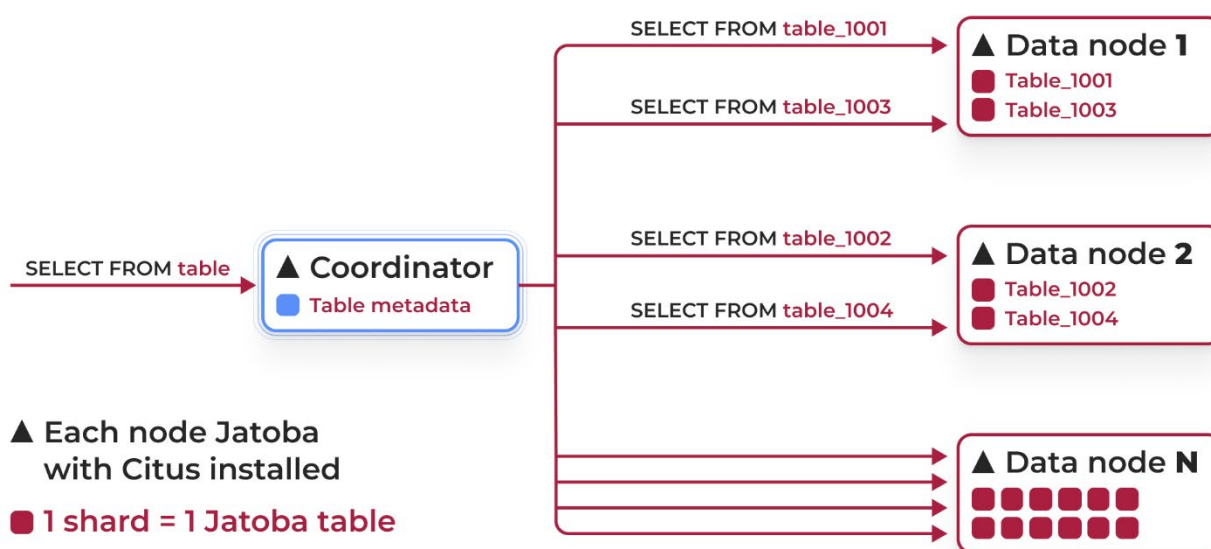


Рисунок 2.2 – Распределенные таблицы

Здесь строки table хранятся в таблицах table_1001 и table_1002 на рабочих узлах. Такие таблицы называются сегментами.

ja_Hire_Cluster выполняет не только функции SQL, но и функции DDL по всему кластеру, поэтому изменение схемы распределенной таблицы выполняется каскадно на всех узлах для обновления всех сегментов таблицы между рабочими узлами.

Создание распределенной таблицы описано в разделе 4.1.

ja_Hire_Cluster использует алгоритмическое сегментирование для назначения строк сегментам. Это означает, что назначение выполняется детерминированно – на основе значения определенного столбца таблицы, называемого столбцом распределения. Администратор кластера должен назначить этот столбец при распределении таблицы. Правильный выбор столбца распределения важен для производительности и функциональности.

– Тип 2: Справочные таблицы

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Справочная таблица – это тип распределенной таблицы, все содержимое которой сосредоточено в одном сегменте, который реплицируется на каждом рабочем узле. Запросы к любому рабочему узлу могут получить доступ к справочной информации локально, без затрат сети на запрос строк с другого узла. Справочные таблицы не имеют столбца распределения, поскольку нет необходимости определять сегмент строки.

Справочные таблицы обычно небольшие и используются для хранения данных, относящихся к запросам, выполняемым на любом рабочем узле.

При взаимодействии со справочной таблицей автоматически выполняются двухфазные коммиты (2PC) для транзакций. Это означает, что данные всегда находятся в согласованном состоянии, независимо от того, записываются они, изменяются или удаляются.

Создание справочных таблиц описано в разделе 4.2 документа.

– Тип 3: Локальные таблицы

При использовании ja_Hipe_Cluster узел координатора, к которому необходимо подключиться и взаимодействовать, представляет собой обычную базу данных СУБД «Jatoba» с установленным расширением ja_Hipe_Cluster. Таким образом, можно создавать стандартные таблицы и не разделять их. Это полезно для небольших административных таблиц, которые не участвуют в join-запросах.

Локальные таблицы СУБД «Jatoba» создаются по умолчанию при выполнении команды create table. Почти в каждом развертывании ja_Hipe_Cluster видно, что стандартные таблицы СУБД «Jatoba» используются вместе с распределенными и справочными таблицами. ja_Hipe_Cluster использует локальные таблицы для хранения метаданных кластера.

2.1.4. Структура таблицы метаданных pg_dist_shard

Таблица метаданных pg_dist_shard на координаторе содержит строку для каждого сегмента каждой распределенной таблицы в системе (рисунок 2.3). Строка соответствует идентификатору сегмента с диапазоном целых чисел в хэш-пространстве (shardminvalue, shardmaxvalue):

```
SELECT * from pg_dist_shard;
```

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Вывод SQL-команды представлен на рисунке 2.3.

logicalrelid	shardid	shardstorage	shardminvalue	shardmaxvalue
github_events	102026	t	268435456	402653183
github_events	102027	t	402653184	536870911
github_events	102028	t	536870912	671088639
github_events	102029	t	671088640	805306367

(4 строки)

Рисунок 2.3 – Вывод результата функции pg_dist_shard

При определении узлом-координатором в каком сегменте содержится определенная строка github_events, он хэширует значение столбца распределения в этой строке и проверяет, в каком диапазоне сегментов содержится хэшированное значение. Диапазоны определены таким образом, что образ хэш-функции является их непересекающимся объединением.

2.1.5. Размещение сегментов

Предположим, что сегмент 102027 связан с рассматриваемой строкой. Это означает, что строка должна быть прочитана или записана в таблицу github_events_102027, которая находится в одном из рабочих узлов. Этот узел определен в таблицах с метаданными. Сопоставление сегментов с рабочими узлами называется размещением сегментов.

Размещение таблиц можно определить с помощью join-запросов к таблице с метаданными. Их выполняет координатор для маршрутизации запросов. Координатор переписывает запросы во фрагменты, которые ссылаются на конкретные таблицы, и выполняет эти фрагменты на соответствующих рабочих узлах.

```
SELECT
    shardid,
    node.nodename,
    node.nodeport
FROM pg_dist_placementplacement
JOIN pg_dist_node node
    ON placement.groupid = node.groupid
    AND node.noderole = 'primary'::noderole
```

```
WHERE shardid = 102027;
```

Вывод SQL-команды представлен на рисунке 2.4.

shardid	nodename	nodeport
102027	localhost	5433

Рисунок 2.4 – Вывод результата функции

В примере `github_events` было четыре фрагмента (рисунок 2.3). Количество сегментов настраивается для каждой таблицы во время ее распространения по кластеру. Оптимальный выбор количества сегментов зависит от варианта использования.

2.1.6. Совместное размещение

Поскольку администратор СУБД может задать, на каких узлах будут храниться какие сегменты, имеет смысл разместить вместе сегменты, содержащие связанные строки связанных таблиц. Тогда запросы объединения между ними будут выполняться внутри одного узла, что поможет избежать отправки большого количества информации по сети.

2.1.7. Параллелизм

Распределение запросов по нескольким устройствам позволяет выполнять больше запросов одновременно. Можно также увеличить скорость обработки за счет добавления новых устройств в кластер. Разделение одного запроса на фрагменты увеличивает выделяемую для него вычислительную мощность. Это обеспечивает наибольший параллелизм через использование ядер процессора.

Запросы, читающие или пишущие в равномерно распределенные по узлам сегменты, могут выполняться со скоростью «реального времени».

Результаты запроса должны передаваться обратно через узел координатора, поэтому ускорение более заметно, когда результат компактен — например, при выполнении агрегирующих запросов.

2.1.8. Выполнение запроса

При выполнении запросов с несколькими сегментами `ja_Hipe_Cluster` должен сбалансировать выгоды от параллелизма с накладными расходами на подключения к базе

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

данных (задержка в сети и использование ресурсов рабочего узла). Настройка выполнения запросов `ja_Hipe_Cluster` помогает понять, как компонент управляет и сохраняет соединения с базой данных между координационным узлом и рабочими узлами.

`ja_Hipe_Cluster` преобразует каждый входящий сеанс запроса с несколькими сегментами в запросы для каждого сегмента, называемые задачами. Он ставит задачи в очередь и запускает их, как только сможет получить соединения с соответствующими рабочими узлами. Для запросов к распределенным таблицам `foo` и `bar` на рисунке 2.5 представлена схема управления соединениями.

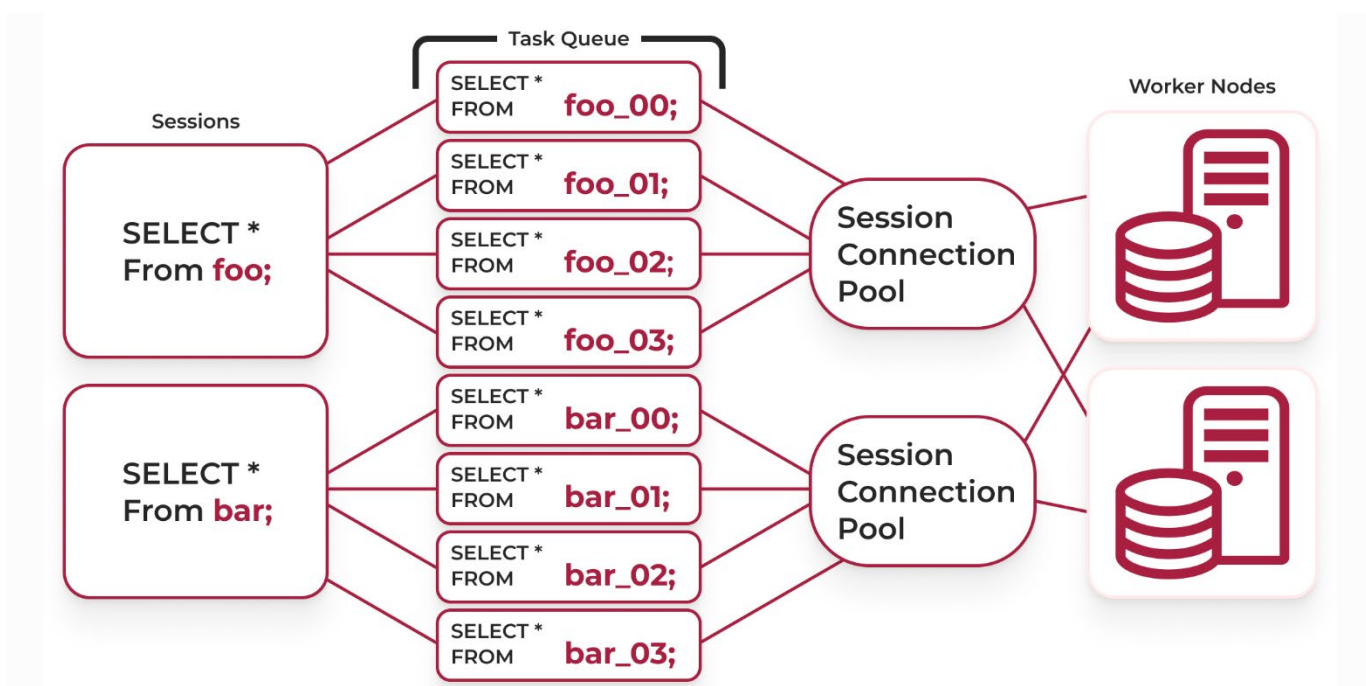


Рисунок 2.5 – Схема управления соединениями

Узел координатора имеет пул подключений для каждого сеанса. Каждый запрос (например, `SELECT * FROM foo`) ограничен открытием не более `citus.max_adaptive_executor_pool_size` (целое число) одновременных подключений для своих задач на один рабочий узел. Этот параметр настраивается на уровне сеанса для управления приоритетами.

Выполнять короткие задачи последовательно через одно и то же соединение быстрее, чем устанавливать для них новые соединения параллельно. С другой стороны, длительные задачи имеют преимущество перед непосредственным параллелизмом.

Для балансировки потребностей коротких и длительных задач `ja_Hipe_Cluster` использует `citus.executor_slow_start_interval` (целое число). Этот параметр определяет

задержку между попытками подключения для задач в запросе к нескольким сегментам. Первые задачи в очереди используют только одно соединение. В конце каждого интервала, если есть ожидающие подключения, ja_Hipe_Cluster увеличивает количество одновременных подключений, которые он откроет. Поведение медленного запуска можно полностью отключить, установив для параметра значение 0.

Когда задача перестает использовать соединение, пул сеансов сохраняет соединение открытым для последующего использования. Кэширование соединения позволяет избежать накладных расходов на восстановление соединения между координатором и рабочим узлом. Каждый пул будет содержать не более citus.max_cached_conns_per_worker (целое число) незанятых соединений, открывающихся одновременно, для ограничения использования ресурсов незанятых соединений в рабочем узле.

Параметр citus.max_shared_pool_size (целое число) действует как отказоустойчивый. Это ограничивает общее количество подключений на один рабочий узел между всеми задачами.

2.1.9. Добавление координатора

Координатор ja_Hipe_Cluster хранит только метаданные о сегментах таблицы и не хранит никаких данных. Все вычисления передаются рабочим узлам, а координатор выполняет только окончательные агрегации результатов. Повысить производительность координатора можно при помощи переключения на более мощное устройство.

В некоторых случаях, связанных с интенсивным использованием, пользователи могут добавить другого координатора. Поскольку таблицы метаданных небольшие (обычно размером в несколько мегабайт), можно скопировать метаданные на другой узел и регулярно синхронизировать их. После этого пользователи смогут отправлять свои запросы любому координатору и увеличивать производительность.

2.1.10. Обработка аварийных ситуаций

2.1.10.1 Варианты обработки аварийных ситуаций

Координатор ja_Hipe_Cluster поддерживает таблицу pg_dist_node с метаданными для отслеживания всех узлов кластера и местоположений сегментов базы данных на этих узлах. Таблицы метаданных небольшие (обычно размером в несколько мегабайт) и меняются не часто. Если на узле произойдет сбой, они могут быть быстро восстановлены.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Справиться со сбоями узла с ролью «Coordinator» можно следующими способами:

1. Использовать потоковую репликацию СУБД «Jatoba». Если основной узел координатора выходит из строя, резервный узел может быть автоматически переведен в основной для обслуживания запросов к кластеру.
2. Использовать средства резервного копирования. Пользователи могут использовать средства резервного копирования СУБД «Jatoba» для резервного копирования метаданных. После чего скопировать эти метаданные на новые узлы для возобновления работы.
3. Использовать резервирование узлов (как координаторов, так и реплик) высокопроизводительного кластера с использованием компонента «jaDog», входящего в СУБД «Jatoba». Взаимодействие компонентов «ja_Hipe_Cluster» и «jaDog» позволяет обеспечить отказоустойчивость за счет использования резервирования вычислительных ресурсов и обработки возникающих отказов, предоставляет возможность производить действия по изменению профиля отказоустойчивости высокопроизводительного кластера компонента «ja_Hipe_Cluster» и его подкластеров, получать текущее состояние кластера как в целом, так и его составных частей.

С данной целью создается общий кластер (так называемый bundle). Кластер компонента «ja_Hipe_Cluster», вместе с узлом «Coordinator» и рабочими узлами, присоединяются к общему кластеру (bundle), после чего к нему возможно присоединять дополнительные резервирующие кластеры при помощи компонента «jaDog».



«Новый» узел продолжит работать в качестве узла высокопроизводительного кластера либо до ручного переключения (switchover) в компоненте «jaDog» и последующим обновлением списка реплик в таблице pg_dist_node компонента «ja_Hipe_Cluster», либо до возникновения следующей аварийной ситуации, когда переключение вновь будет выполнено автоматически.

Описание создания и управления отказоустойчивым кластером компонента «jaDog» приведен в разделе 8 «Настройка общего кластера (bundle) с компонентом «jaDog» в ручном режиме» второй части документа «Компонент jaDog. Управление режимом работы узлов кластера» 643.72410666.00067-07 98 02-02.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

2.1.10.2 Сбой координатора и/или рабочего узла

При выполнении настройки высокопроизводительного кластера на основе «ja_Hipe_Cluster» администратор использует IP-адреса узлов. Это позволяет компоненту «jaDog», использовать функцию citus_update_node «ja_Hipe_Cluster» (см. п.п. 4.12.2.2).

В случае возникновения аварийной ситуации, затрагивающей узел с ролью «Coordinator» и/или рабочий узел (реплику), автоматизировано выполняются следующие действия:

- Компонент «jaDog» выполняет переключение аварийного узла на один из резервных;
- Компонент «jaDog», используя функцию citus_update_node, выполняет запрос по обновлению списка узлов кластера «ja_Hipe_Cluster»:

```
SELECT citus_update_node(nodeid, 'new-address', nodeport) FROM  
pg_dist_node WHERE nodename = 'old-address';
```

Где nodeid – идентификатор узла из таблицы pg_dist_node, new-address – IP-адрес узла, на который выполнено переключение, nodeport – номер сетевого порта узла с идентификатором nodeid из таблицы pg_dist_node, old-address – IP-адрес узла, который был аварийно выведен из эксплуатации.

Обновления таблицы pg_dist_node обеспечивает компонент «ja_Hipe_Cluster» актуальным списком доступных узлов.

- Компонент «ja_Hipe_Cluster» на основе обновленной таблицы pg_dist_node выполняет перенаправление обращений клиентов на «новый» узел.

3. УСТАНОВКА И НАСТРОЙКА КОМПОНЕНТА

В разделе приведен пример настройки кластера с парольной аутентификацией. В Приложении 3 приведена инструкция по настройке SSL соединения для ja_Hipe_Cluster.

3.1. Установка компонента

В качестве примера рассматривается три сервера с СУБД «Jatoba». Параметры серверов приведены в таблице 3.1.

Таблица 3.1 – Сетевая адресация серверов стенда

№	ОС	Имя сервера	DNS-имя узла	IP-адрес	Маска подсети	Роль
1	Ubuntu 22.04	Citus	coord	10.116.102.61	255.255.255.0	Coordinator
2	Ubuntu 22.04	Node1	worker1	10.116.102.62	255.255.255.0	Node1
3	Ubuntu 22.04	Node2	worker2	10.116.102.63	255.255.255.0	Node2

Схема стенда представлена на рисунке 3.1.

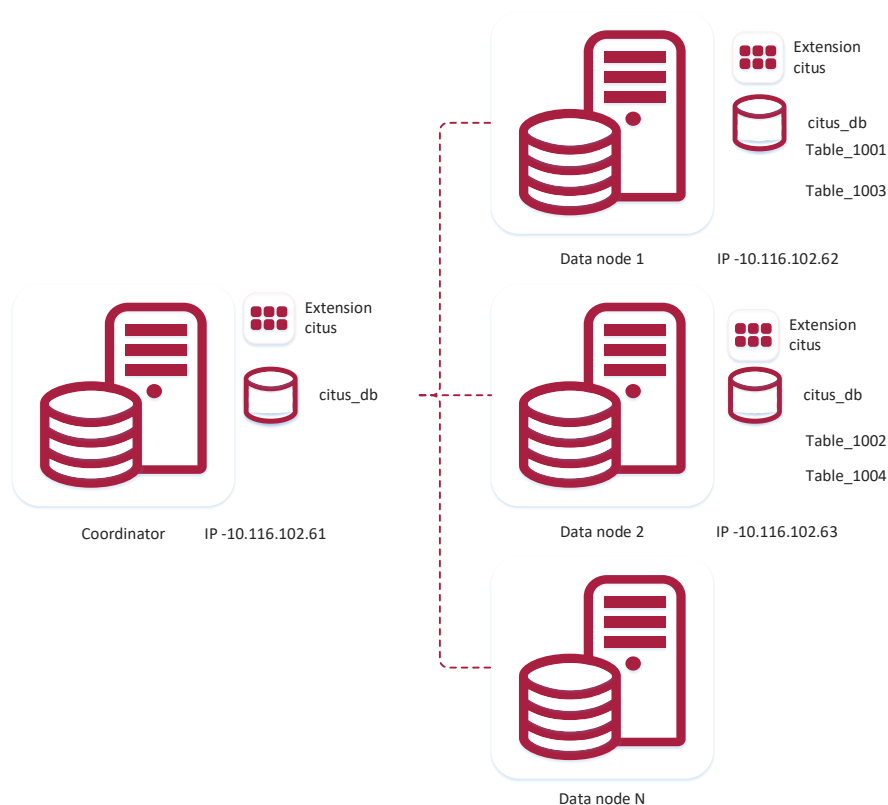


Рисунок 3.1 – Схема установки компонента

Компонент ja_Hipe_Cluster позволяет использовать при настройке узлов кластера вместо IP-адресов записи с именем узла с использованием DNS. Для этого используемые сервера DNS должны быть соответствующим образом настроены.

В случае отсутствия DNS-сервера, обрабатывающего имена узлов кластера, необходимо на каждом узле создать и заполнить файл /etc/hosts с IP-адресами, например:

```
10.116.102.61 coord
10.116.102.62 worker1
10.116.102.63 worker2
```

Перед началом настройки необходимо убедиться, что при установке СУБД «Jatoba» был установлен компонент ja_Hipe_Cluster. Если компонент отсутствует, необходимо его установить с помощью команды:

```
apt install jatoba6-ja-hipe-cluster
```

Полностью установка СУБД «Jatoba» вместе с компонентом ja_Hipe_Cluster приведена в Приложении 1 документа.

3.2. Настройка сервера «Coordinator»

3.2.1. Редактирование конфигурационного файла «postgresql.conf»

Перед запуском БД необходимо изменить права доступа к ней. По умолчанию сервер БД пропускает только клиентов на localhost. В части данного шага следует пропускать все IP-интерфейсы, а затем настроить файл аутентификации клиента для разрешения всех входящих подключений из локальной сети.

Для чего открыть для редактирования конфигурационный файл «postgresql.conf»:

```
cd /var/lib/jatoba/6/data/
nano postgresql.conf
```

В разделе «Connections and authentication» установить параметр:

```
listen_addresses = '*'
```

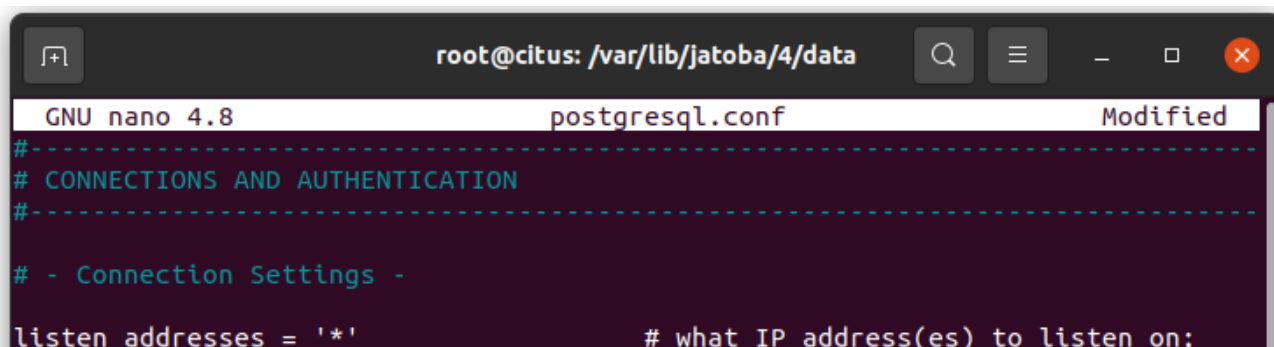


Рисунок 3.2 – Конфигурационный файл «postgresql.conf»

В разделе «Version and Platform Compatibility» убедиться в том, что следующий параметр закомментирован:

```
#standard_conforming_strings = off
```

В разделе «Shared Library Preloading» для последующей загрузки разделяемой библиотеки расширения установить параметр:

```
shared_preload_libraries = 'citus'
```

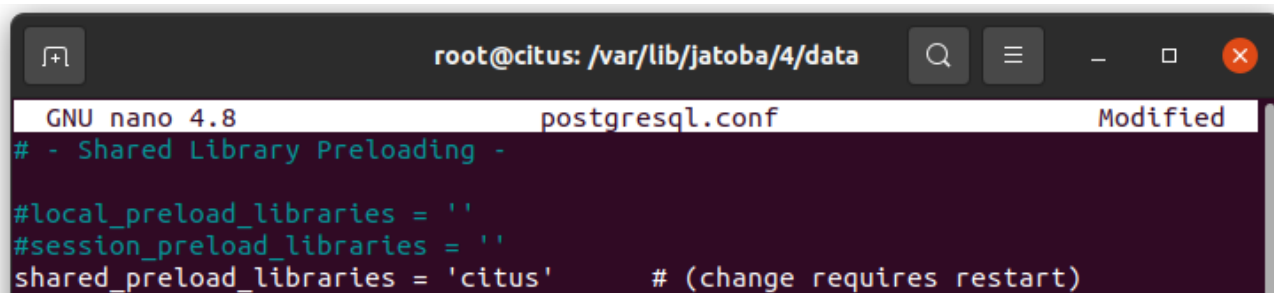


Рисунок 3.3 – Раздел «Shared Library Preloading»

Сохранить внесенные изменения.

3.2.2. Редактирование конфигурационного файла «pg_hba.conf»

Установить метод аутентификации «MD5» для соединений СУБД, для этого открыть для редактирования конфигурационный файл «pg_hba.conf»:

```
nano pg_hba.conf
```

и установить следующие параметры:

TYPE	DATABASE	USER	ADDRESS	METHOD
# Require password access and a ssl/tls connection to nodes in the local network				
hostssl	all	all	10.116.102.0/24	md5

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Require passwords and ssl/tls connections when the host connects to itself as well.

hostssl	all	all	127.0.0.1/32	md5
hostssl	all	all	::1/128	md5

Произойдет изменение установленного по умолчанию метода аутентификации на «MD5» и добавится строка с указанием подсети 10.116.102.0/24, в которой находятся узлы кластера.

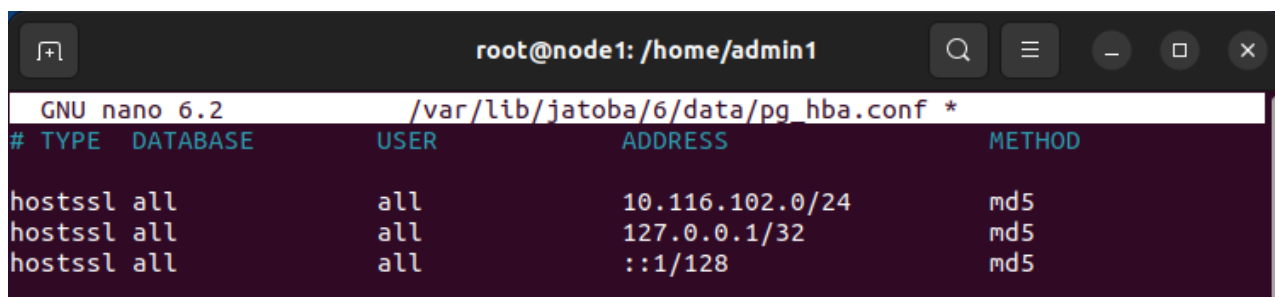


Рисунок 3.4 – Конфигурационный файл «pg_hba.conf»

Необходимо создать файл, содержащий информацию о паролях к узлам. Для этого создать и открыть для редактирования файл «.pgpass»:

```
nano /var/lib/jatoba/.pgpass
```

Записать информацию о каждом узле в кластере в формате:

```
hostname:port:database:username:password
```

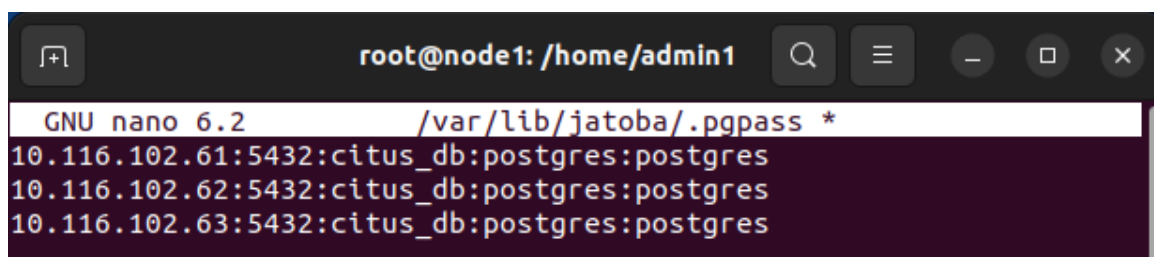


Рисунок 3.5 – Файл «.pgpass»

Изменить права доступа на файл «.pgpass»:

```
chmod 600 /var/lib/jatoba/.pgpass
```

Изменить владельца файла «.pgpass» на пользователя «postgres»:

```
chown postgres:postgres /var/lib/jatoba/.pgpass
```

Сохранить внесенные изменения и перезапустить демон СУБД «jatoba-6» командами:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
systemctl stop jatoba-6
systemctl start jatoba-6
systemctl status jatoba-6
```

3.2.3. Создание БД «citus_db»

Войти в СУБД от имени и с правами привилегированного пользователя:

```
cd /usr/jatoba-6/bin/
psql -h localhost -d postgres -U postgres
```

Создать БД «citus_db» и просмотреть список БД в СУБД SQL-командой;

```
create database citus_db;
\list
```

```
root@citus: /usr/jatoba-4/bin
postgres=# create database citus_db;
CREATE DATABASE
postgres=# \list
               List of databases
  Name      | Owner   | Encoding | Collate | Ctype  | Access privileges
-----+-----+-----+-----+-----+-----
citus_db   | postgres | SQL_ASCII | C       | C      |
postgres   | postgres | SQL_ASCII | C       | C      |
template0  | postgres | SQL_ASCII | C       | C      | =c/postgres      +
           |          |          |          |          | postgres=CTc/postgres
template1  | postgres | SQL_ASCII | C       | C      | =c/postgres      +
           |          |          |          |          | postgres=CTc/postgres
(4 rows)
postgres=#
```

Рисунок 3.6 – Создание БД

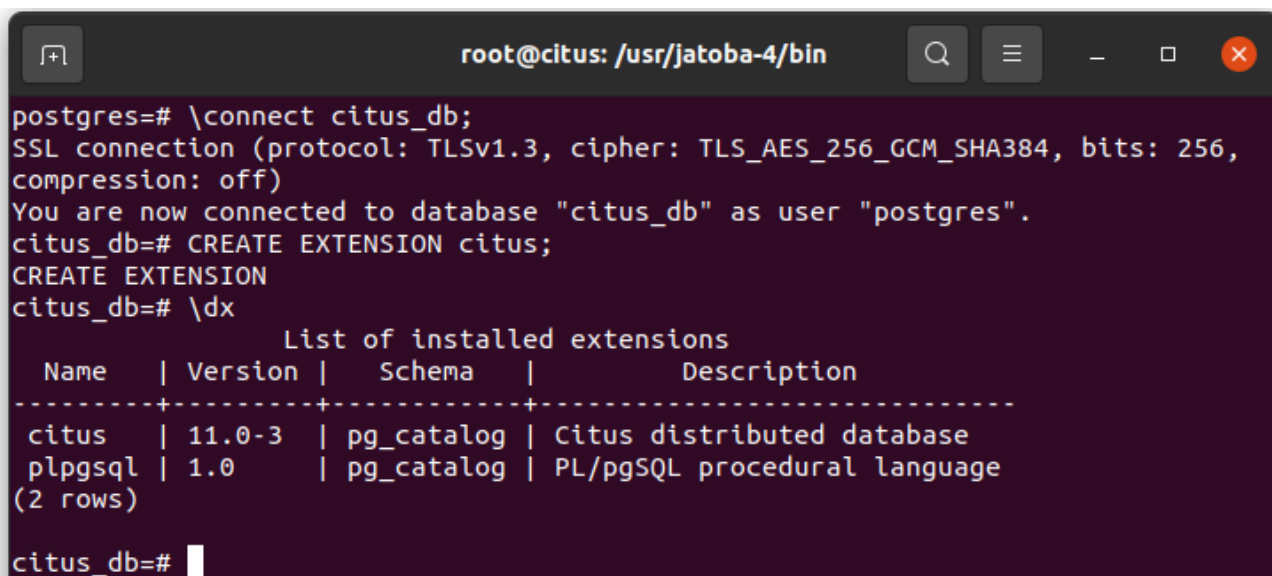
3.2.4. Установка расширения «citus»

Расширение «citus» устанавливается в БД, которую планируется сделать распределенной. Для чего потребуется установить соединение с БД «citus_db». Затем выполнить установку расширения «citus» SQL-командой;

```
\connect citus_db;
CREATE EXTENSION citus;
```

Просмотр установленных расширений выполняется SQL-командой:

\dx



```
root@citus: /usr/jatoba-4/bin
postgres=# \connect citus_db;
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
compression: off)
You are now connected to database "citus_db" as user "postgres".
citus_db=# CREATE EXTENSION citus;
CREATE EXTENSION
citus_db=# \dx

               List of installed extensions
  Name  | Version | Schema  | Description
-----+-----+-----+-----
 citus  | 11.0-3  | pg_catalog | Citus distributed database
 plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
(2 rows)

citus_db=#
```

Рисунок 3.7 – Установка расширения

На данном шаге первоначальная настройка компонента завершена и можно переходить к конфигурированию узлов кластера.

3.3. Настройка сервера «Node1»

Действия по настройке сервера СУБД с ролью первого узла аналогичны описанным в следующих пунктах документа:

- 3.2.1 «Редактирование конфигурационного файла «postgresql.conf»;
- 3.2.2 «Редактирование конфигурационного файла «pg_hba.conf»;
- 3.2.3 «Создание БД «citus_db»;

```

root@node1: /usr/jatoba-4/bin
postgres=# create database citus_db;
CREATE DATABASE
postgres=# \list

```

Name	Owner	Encoding	Collate	Ctype	Access privileges
citus_db	postgres	SQL_ASCII	C	C	
postgres	postgres	SQL_ASCII	C	C	
template0	postgres	SQL_ASCII	C	C	=c/postgres postgres=CTc/postgres
template1	postgres	SQL_ASCII	C	C	=c/postgres postgres=CTc/postgres

```

(4 rows)
postgres=#

```

Рисунок 3.8 – Создание БД «citus_db» на сервере СУБД «Node1»

- 3.2.4 «Установка расширения «citus»;

```

root@node1: /usr/jatoba-4/bin
postgres=# \connect citus_db;
You are now connected to database "citus_db" as user "postgres".
citus_db=# CREATE EXTENSION citus;
CREATE EXTENSION
citus_db=# \dx

```

Name	Version	Schema	Description
citus	11.0-3	pg_catalog	Citus distributed database
plpgsql	1.0	pg_catalog	PL/pgSQL procedural language

```

(2 rows)
citus_db=#

```

Рисунок 3.9 – Установка расширения «citus» на сервере СУБД «Node1»

3.4. Настройка сервера «Node2»

Действия по настройке сервера СУБД с ролью второго узла аналогичны описанным в следующих пунктах документа:

- 3.2.1 «Редактирование конфигурационного файла «postgresql.conf»;
- 3.2.2 «Редактирование конфигурационного файла «pg_hba.conf»;
- 3.2.3 «Создание БД «citus_db»;


```

root@node2: /usr/jatoba-4/bin
postgres=# create database citus_db;
CREATE DATABASE
postgres=# \list

```

Name	Owner	Encoding	Collate	Ctype	Access privileges
citus_db	postgres	SQL_ASCII	C	C	
postgres	postgres	SQL_ASCII	C	C	
template0	postgres	SQL_ASCII	C	C	=c/postgres +
template1	postgres	SQL_ASCII	C	C	postgres=CTc/postgres +

```

(4 rows)
postgres=#

```

Рисунок 3.10 – Создание БД «citus_db» на сервере СУБД «Node2»

- 3.2.4 «Установка расширения «citus»;

```

root@node2: /usr/jatoba-4/bin
postgres=# \connect citus_db;
You are now connected to database "citus_db" as user "postgres".
citus_db=# CREATE EXTENSION citus;
CREATE EXTENSION
citus_db=# \dx

```

Name	Version	Schema	Description
citus	11.0-3	pg_catalog	Citus distributed database
plpgsql	1.0	pg_catalog	PL/pgSQL procedural language

```

(2 rows)
citus_db=#

```

Рисунок 3.5 – Установка расширения «citus» на сервере СУБД «Node2»

3.5. Настройка кластера

3.5.1. Добавление информации об узле координатора

Узлы кластера (распределенной БД) должны знать об узле выполняющего функции координатора. Для этого требуется:

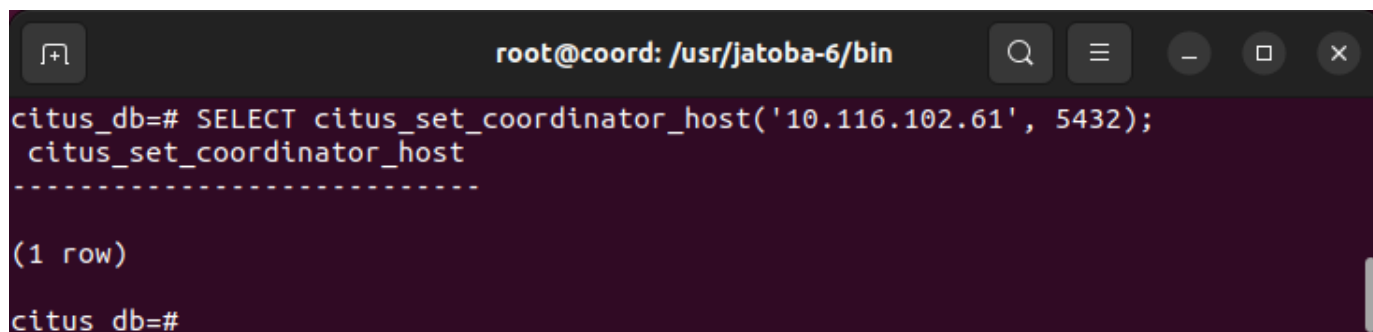
- установить роль координатора на хосте;
- зарегистрировать имя хоста координатора на рабочих узлах кластера.

Для этого на сервере СУБД «Citus» выполните SQL-команду, которая определит его роль как координатора:

```
SELECT citus_set_coordinator_host('10.116.102.61', 5432);
```

Или с использованием DNS-имени узла:

```
SELECT citus_set_coordinator_host('coord', 5432);
```



```
root@coord: /usr/jatoba-6/bin
citus_db=# SELECT citus_set_coordinator_host('10.116.102.61', 5432);
citus_set_coordinator_host
-----
(1 row)
citus_db=#
```

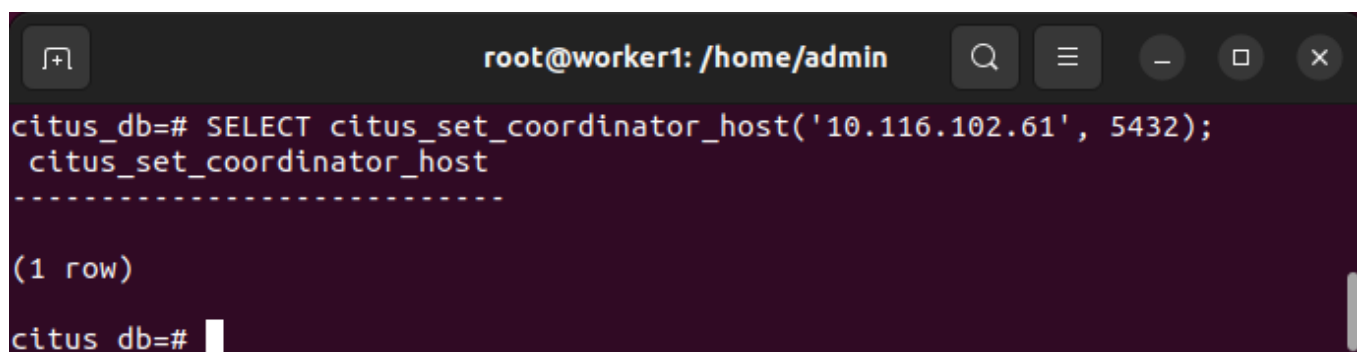
Рисунок 3.12 – Назначение узлу роли координатора

Затем на сервере СУБД «worker1» выполните SQL-команду, которая зарегистрирует имя хоста, к которому он будет обращаться, как к координатору:

```
SELECT citus_set_coordinator_host('10.116.102.61', 5432);
```

Или с использованием DNS-имени узла:

```
SELECT citus_set_coordinator_host('coord', 5432);
```



```
root@worker1: /home/admin
citus_db=# SELECT citus_set_coordinator_host('10.116.102.61', 5432);
citus_set_coordinator_host
-----
(1 row)
citus_db=#
```

Рисунок 3.63 – Регистрация имени хоста координатора на узле СУБД «worker1»

Аналогичное действие выполняется на сервере СУБД «worker2». Выполняется SQL-команда:

```
SELECT citus_set_coordinator_host('10.116.102.61', 5432);
```

```

root@node2: /usr/jatoba-4/bin
citus_db=# SELECT citus_set_coordinator_host('10.96.1.80', 5432);
citus_set_coordinator_host
-----
(1 row)
citus_db=#
  
```

Рисунок 3.14 – Регистрация имени хоста координатора на узле СУБД «worker2»

3.5.2. Добавление узлов кластера

После того, как для всех узлов кластера определена и зарегистрирована роль координатора, требуется зарегистрировать на узле координатора рабочие узлы кластера.

Добавьте узлы кластера SQL-командой на сервере СУБД «Citus»:

```
SELECT citus_add_node('10.116.102.62', 5432);
```

Или с использованием DNS-имени узла:

```
SELECT citus_add_node('worker1', 5432);
```

```

root@coord: /usr/jatoba-6/bin
citus_db=# select citus_add_node('10.116.102.62', 5432);
citus_add_node
-----
17
(1 row)
citus_db=#
  
```

Рисунок 3.15 – Добавление первого узла (сервер СУБД «Node1»)

```
SELECT citus_add_node('10.116.102.63', 5432);
```

Или с использованием DNS-имени узла:

```
SELECT citus_add_node('worker2', 5432);
```

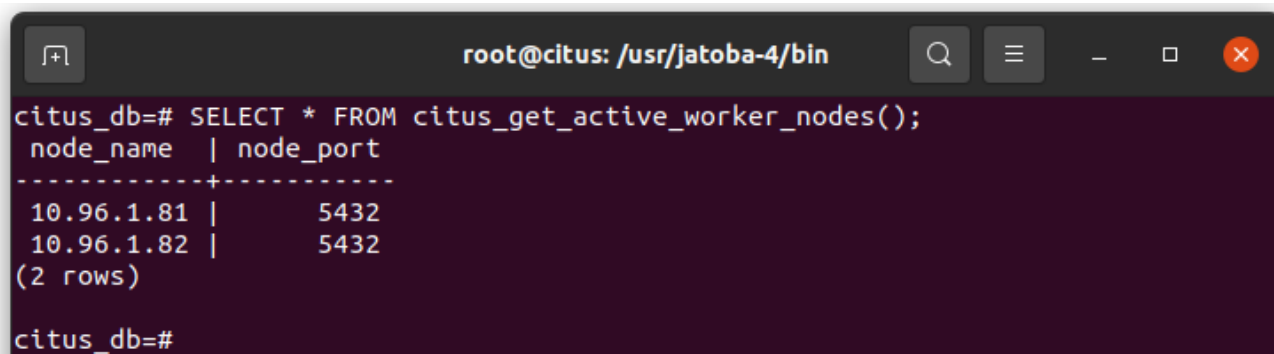
При успешной установке узел координатора выбирает желаемую рабочую конфигурацию.

Информация об узлах кластера добавляется в таблицу pg_dist_node.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Для вывода информации об узлах кластера выполните SQL-команду на сервер СУБД выполняющего роль координатора:

```
sudo -i -u postgres psql -c "SELECT * FROM
citus_get_active_worker_nodes();" "
```



```
root@citus: /usr/jatoba-4/bin
citus_db=# SELECT * FROM citus_get_active_worker_nodes();
 node_name | node_port
-----+-----
 10.96.1.81 |      5432
 10.96.1.82 |      5432
(2 rows)
citus_db=#
```

Рисунок 3.7 – Список узлов кластера

3.5.3. Использование режима **preferred** для запросов только на чтение к узлам кластера

В компоненте `ja_Hipe_Cluster` применяется параметр `citus.use_secondary_nodes`, который принимает значения «always», «never» и «noderack».

При значении «never» запросы на чтение данных никогда не транслируются на вторичные узлы (релики).

При значении «always» все запросы переадресуются на вторичные узлы, а возможность выполнения транзакций чтения-записи отключена.

При значении «noderack» все запросы направляются к узлам, привязанным к определенному региону или ЦОД. Подробно параметр описан в разделе 5, настоящего документа.

Для выполнения запросов только на чтение данных (READ ONLY) с вторичных узлов в компоненте `ja_Hipe_Cluster` может применяться специальная обработка поведения в планировщике запросов к узлам через дополнительный режим «preferred», который обеспечивает следующее поведение кластера:

- все транзакции чтения-записи данных выполняются штатно, как в режиме «never»;

– транзакции только для чтения данных (READ ONLY) могут быть переадресованы на вторичные узлы, как в режиме «always».

Для обеспечения доступа к узлам в режиме preferred кластер рекомендуется создавать из следующих трех узлов:

- координатор;
- основной (worker) узел primary (см. п.п. 4.12.2.1);
- вторичный (replica) узел secondary (см. п.п. 4.12.2.7).

Для объяснения механизма необходимо привести пример: кластер, состоящий из двух основных узлов (worker1 и worker2) и их вторичных узлов (replica1 и replica2). На основном узле worker1 запущена транзакция, которая требует чтения сегментов данных, расположенных на основных узлах worker1 и worker2. В этом случае обеспечивается следующее поведение:

- Для чтения данных с worker2, узел worker1 будет обращаться к replica1;
- Для чтения данных с worker1, узел worker1 не должен обращаться к replica1, а выполняет это чтение локально.

Стоит отметить, что при добавлении узла в кластер по умолчанию он считается основным.

Включение режима preferred для запросов только на чтение

В СУБД «Jatoba» режим preferred активируется при помощи специальной опции citus_use_secondary = preferred. Для чего необходимо открыть для редактирования конфигурационный файл «postgresql.conf»:

```
cd /var/lib/jatoba/6/data/  
nano postgresql.conf
```

В разделе «CUSTOMIZED OPTIONS» установить параметр:

```
citus_use_secondary = preferred
```

Сохранить изменения в файле postgresql.conf.

Другим вариантом установки опции `citus_use_secondary = preferred` является использование функции СУБД `alter system set`. Для этого необходимо подключиться к СУБД и выполнить команду:

```
alter system set citus.use_secondary_nodes = 'preferred';
```

После внесения изменений в параметры СУБД необходимо перезапустить и проверить статус службы «jatoba-6» при помощи команд:

```
systemctl stop jatoba-6  
systemctl start jatoba-6  
systemctl status jatoba-6
```

Подключиться на узле координатор к СУБД и выполнить команду:

```
show citus.use_secondary_nodes;
```

Если параметры установлены верно будет отображено сообщение «preferred».

Проверка режима preferred запросов на чтение с вторичных узлов

Для проверки запросов на чтение с вторичного узла (secondary) необходимо на узле координаторе подключиться к СУБД и выполнить следующий запрос на чтение одной строки:

```
begin read only;  
select *, pg_is_in_recovery() from <table_name> where  
<condition> = <value>;  
commit;
```

Где `table_name`, `condition` и `value` – параметры проверяемой таблицы СУБД.

Пример:

```
begin read only;  
select *, pg_is_in_recovery() from tbl-test where id = 123;  
commit;
```

После выполнения данного запроса будет возвращен результат выполнения функции `pg_is_in_recovery`. Если выведенное значение `t` – запрос на чтение данных выполнялся на вторичном узле (`secondary`), если выведенное значение `f` – запрос на чтение данных выполнялся на основном узле (`primary`).

Если же вторичный узел недоступен, то при выполнении запроса будет выведен текст с ошибкой.

Если кластер состоит из нескольких вторичных узлов, то запрос на чтение данных будет применяться для первого такого вторичного узла из таблицы `pg_dist_node`.

Проверка размера распределенной таблицы

Для проверки размера распределенной таблицы необходимо на вторичном узле подключиться к СУБД и выполнить запрос:

```
select pg_size_pretty(citus_total_relation_size('table_name'));
```

Где `table_name` – название распределенной таблицы.

Размер распределенной таблицы на основном и вторичном узлах должен совпадать.

Отключение режима `preferred` для запросов только на чтение

Для отключения режима `preferred` для запросов только на чтение к узлам кластера необходимо подключиться к СУБД на узле координатор и выполнить запрос:

```
alter system reset citus.use_secondary_nodes;
```

Другим вариантом отключения является удаление или комментирование в конфигурационном файле `postgresql.conf` строки:

```
#citus_use_secondary = preferred
```

После внесения изменений в параметры СУБД необходимо перезапустить и проверить статус службы «jatoba-6» при помощи команд:

```
systemctl stop jatoba-6  
systemctl start jatoba-6
```

```
systemctl status jatoba-6
```

Для проверки корректности отключения режима preferred необходимо подключиться на узле координатор к СУБД и выполнить команду:

```
show citus.use_secondary_nodes;
```

Если режим preferred успешно отключен будет отображено сообщение «never».

3.6. Установка кластера без SU

В компоненте есть интерконнект (соединение между узлами), для всех пользовательских соединений, например, обращений к распределённым таблицам он происходит под именем пользователя, который исполнил запрос. Но есть ещё и техническое взаимодействие между узлами кластера от имени пользователя, который создал расширение.

Пользователь-владелец расширения должен обладать атрибутом суперпользователя (SU) и когда компоненту требуется роль суперпользователя, он обращается к пользователю-владельцу расширения, в том числе для интерконнекта.

Реализована функциональная возможность через GUC указать имя пользователя, обладающего атрибутом суперпользователя. В этом случае предполагается, что у пользователя-владельца расширения можно забрать роль суперпользователя, а когда компоненту потребуется роль суперпользователя, он будет обращаться от имени пользователя, указанного в GUC'е.

Чтобы задать имя пользователя, обладающего ролью суперпользователя, необходимо использовать GUC citus.superuser

Если citus.superuser не задан, компонент должен работать без каких-либо изменений. В этом случае, пользователь-владелец расширения должен обладать атрибутом суперпользователя, как того требует стандартное поведение компонента.

Если citus.superuser задан, то в этом случае у пользователя-владельца расширения можно отобрать атрибут суперпользователя. При этом указанный в GUC пользователь обязан иметь атрибут суперпользователя (SU).

Порядок настройки на каждом узле следующий:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

- 1) Создать пользователя со всеми правами, в том числе с атрибутом суперпользователя;
- 2) Создать пользователя с ролью суперпользователя без права логина;
- 3) Создать БД под пользователем, созданным в пункте (1). Создать под ним расширение «citrus». Отобрать атрибут суперпользователя у этого пользователя. Остановить узел
- 4) В конфигурационном файле «pg_hba.conf» добавить настройку citrus.superuser с указанным именем пользователя, созданным в пункте (2)
- 5) Запустить узел, работать под пользователем, созданным в пункте (1)

В текущей реализации поддерживается только работа пользователя в той БД, в которой он является владельцем. То есть пользователь-владелец расширения должен быть владельцем БД, в которой он работает.

4. ФУНКЦИИ КОМПОНЕНТА

4.1. Создание и модификация распределенных объектов

Перед созданием распределенной таблицы необходимо определить ее схему. Создать таблицу можно с помощью функции CREATE TABLE как в случае с обычной таблицей СУБД «Jatoba».

```
CREATE TABLE github_events
(
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
    actor jsonb,
    org jsonb,
    created_at timestamp
);
```

Для указания столбца распределения таблицы и создания рабочих сегментов используется функция create_distributed_table().

```
SELECT create_distributed_table('github_events', 'repo_id');
```

Эта функция сообщает ja_Hipe_Cluster, что таблица github_events должна быть распределена по столбцу repo_id (путем хэширования значения столбца). Функция также создает сегменты на рабочих узлах с использованием citus.shard_count.

В данном примере будет создано общее количество citus.shard_count сегментов, где каждому сегменту принадлежит часть пространства хэш-токенов. После создания сегментов эта функция сохраняет все распределенные метаданные на координаторе.

Каждому созданному сегменту присваивается уникальный идентификатор сегмента. Каждый сегмент представлен на рабочем узле в виде обычной таблицы СУБД «Jatoba» с

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

именем «tablename_shardid», где tablename – имя распределенной таблицы, а shardid – уникальный идентификатор, присвоенный этому сегменту.

После проделанных выше действий можно вставлять данные в распределенную таблицу и отправлять к ней запросы.

4.2. Справочные таблицы

Вышеупомянутый метод распределяет таблицы по нескольким горизонтальным сегментам, но другой возможностью расширения является распределение таблиц в один сегмент и репликация сегмента на каждый рабочий узел. Таблицы, распределенные таким образом, называются справочные таблицы. Они используются для хранения данных, к которым необходимо часто обращаться нескольким узлам кластера.

Виды справочных таблиц:

- таблицы меньшего размера, которые необходимо объединить с большими распределенными таблицами;
- таблицы в многопользовательских приложениях, в которых отсутствует столбец идентификатора клиента или которые не связаны с клиентом;
- таблицы, которые требуют уникальных ограничений для нескольких столбцов и достаточно малы.

Создание справочной таблицы осуществляется следующей командой:

```
CREATE TABLE device_types (device_type_id int primary key,  
device_type_name text not null unique);
```

В дополнение к распространению таблицы в виде единого реплицированного фрагмента, create_reference_table UDF помечает его как справочную таблицу в таблицах метаданных ja_Hipe_Cluster. Компонент автоматически выполняет двухфазные коммиты (2PC) для изменений в таблицах, помеченных таким образом, что обеспечивает надежные гарантии согласованности.

```
SELECT create_reference_table('device_types');
```

4.3. Распределение данных координатора

Если существующая база данных СУБД «Jatoba» преобразуется в узел координатора для кластера ja_Hipe_Cluster, данные в ее таблицах могут быть распределены с минимальным прерыванием работы приложения.

Функция `create_distributed_table` работает как с пустыми, так и с непустыми таблицами. Если таблица содержит данные, функция автоматически распределяет строки таблицы по всему кластеру. Определить, делает ли функция распределение строк таблицы, можно по сообщению «NOTICE: Copying data from local table...»:

```
CREATE TABLE series AS SELECT i FROM generate_series(1,1000000)
i;
SELECT create_distributed_table('series', 'i');
NOTICE: Copying data from local table...
NOTICE: copying the data has completed
DETAIL: The local data in the table is no longer visible, but
is still on disk.
HINT: To remove the local data, run: SELECT
truncate_local_data_after_distributing_table($$public.series$$)
```

Вывод SQL-команды представлен на рисунке 4.1.

```
create_distributed_table
-----
(1 row)
```

Рисунок 4.1 – Вывод результата функции `create_distributed_table`

Записи в таблице блокируются на время переноса данных, а ожидающие записи обрабатываются как распределенные запросы после успешного завершения работы функции (если функция завершается с ошибкой, запросы снова становятся локальными).

При распределении таблиц А и В, где А имеет внешний ключ для В, сначала необходимо распределить таблицу назначения ключа В. Выполнение этого в неправильном порядке приведет к ошибке:

```
ERROR: cannot create foreign key constraint
DETAIL: Referenced table must be a distributed table or a
reference table.
```

Если невозможно распределить в правильном порядке, нужно удалить внешние ключи, распределить таблицы и заново создать внешние ключи.

После распределения таблиц используется функция `truncate_local_data_after_distributing_table` для удаления локальных данных. Оставшиеся локальные данные в распределенных таблицах недоступны для запросов `ja_Hipe_Cluster` и могут привести к неожиданным нарушениям ограничений на координаторе.

При переносе данных из внешней базы данных в управляемую службу, необходимо сначала создать распределенные таблицы `ja_Hipe_Cluster` с помощью `create_distributed_table`, затем скопировать данные в таблицу. Копирование в распределенные таблицы позволяет избежать нехватки места на узле координатора.

4.4. Совместное размещение таблиц

4.4.1. Создание совместно размещенных таблиц

Совместное размещение – это тактическое распределение данных, хранение связанной информации на одних и тех же компьютерах для более эффективного выполнения реляционных операций. При этом используется горизонтальная масштабируемость для всего набора данных.

Таблицы расположены совместно в группах. Для ручного управления назначения группы совместного размещения таблицы используется необязательный параметр `colocate_with` функции `create_distributed_table`. По умолчанию используется значение «default», которое группирует таблицу с любой другой таблицей совместного размещения по умолчанию, имеющей тот же тип столбца распределения и количество сегментов. Для отмены или изменения этого неявного размещения используется `update_distributed_table_colocation()`.

```
SELECT create_distributed_table('A', 'some_int_col');
SELECT create_distributed_table('B', 'other_int_col');
```

Если новая таблица не связана с другими в предполагаемой неявной группе совместного размещения, требуется указать `colocated_with => 'none'`.

```
SELECT create_distributed_table('A', 'foo', colocate_with =>
'none');
```

Разделение несвязанных таблиц на их собственные группы совместного размещения улучшит производительность перебалансировки сегментов, поскольку сегменты в одной группе должны перемещаться вместе.

Если таблицы связаны между собой (например, если над ними будут выполняться операции `join`), возможно имеет смысл явно разместить их вместе.

Для явного совместного размещения нескольких таблиц требуется распределить одну, а затем поместить остальные в ее группу совместного размещения.

К примеру:

```
CREATE TABLE stores (
  id UUID,
  owner_email VARCHAR(255),
  owner_password VARCHAR(255),
  name VARCHAR(255),
  url VARCHAR(255),
  last_login_at TIMESTAMPTZ,
  created_at TIMESTAMPTZ
)
```

```
CREATE TABLE products (  
    id UUID,  
    name VARCHAR(255),  
    description TEXT,  
    price INTEGER,  
    quantity INTEGER,  
    store_id UUID,  
    created_at TIMESTAMPTZ,  
    updated_at TIMESTAMPTZ  
)
```

```
SELECT create_distributed_table('stores', 'store_id');  
  
SELECT create_distributed_table('products', 'store_id',  
    colocate_with => 'stores');
```

Информация о группах совместного размещения хранится в таблице pg_dist_colocation.

pg_dist_partition показывает, какие таблицы назначены каким группам.

4.4.2. Выбор столбца распределения

Чтобы распределить строки распределенных таблиц по сегментам, ja_Hipe_Cluster использует столбец распределения. Важно правильно выбрать столбец распределения, потому что это влияет на то, как данные распределены по узлам.

Если столбец распределения выбран верно, связанные между собой данные хранятся вместе на одном узле. Тогда запросы выполняются быстро и доступны все возможности SQL. Если же столбец выбран неправильно, то система будет обрабатывать запросы долго и не все запросы SQL будут доступны.

К примеру, есть сервис, собирающий аналитику с разных приложений (tenant). В СУБД «Jatoba» хранятся таблицы event и page:

```
CREATE TABLE event (  
    tenant_id int,  
    event_id bigint,  
    page_id int,  
    payload jsonb,  
    primary key (tenant_id, event_id)  
);  
  
CREATE TABLE page (  
    tenant_id int,  
    page_id int,  
    path text,  
    primary key (tenant_id, page_id)  
);
```

Сейчас эти таблицы локальные (хранятся на одном узле).

Сервис выводит для пользователя страницу, на которой показана статистика посещений его приложения (например, №6). Чтобы найти число посещений страниц в /blog за последнюю неделю приложения №6, необходимо выполнить запрос:


```
SELECT page_id, count(event_id)
FROM
    page
LEFT JOIN (
    SELECT * FROM event
    WHERE (payload->>'time')::timestamptz >= now() - interval '1
week'
) recent
USING (tenant_id, page_id)
WHERE tenant_id = 6 AND path LIKE '/blog%'
GROUP BY page_id;
```

При увеличении количества данных в таблицах может потребоваться сделать их распределенными:

```
SELECT create_distributed_table('event', 'event_id');
SELECT create_distributed_table('page', 'page_id');
```

Сейчас сегменты распределены по разным узлам, поэтому вместо предыдущего запроса необходимо выполнить два запроса.

Для таблицы page:

```
SELECT page_id FROM page WHERE path LIKE '/blog%' AND tenant_id
= 6;
```

Для таблицы event:

```
SELECT page_id, count(*) AS count
FROM event
WHERE page_id IN (/*...page IDs from first query...*/)
    AND tenant_id = 6
    AND (payload->>'time')::date >= now() - interval '1 week'
```

```
GROUP BY page_id ORDER BY count DESC LIMIT 10;
```

Данные, необходимые для выполнения запроса, разбросаны по сегментам на разных узлах, и на каждый из этих сегментов необходимо отправить запрос.

Такое распределение данных по узлам создает проблемы:

- необходимо выполнить много запросов к разным сегментам;
- первый запрос возвращает много ненужных данных;
- второй запрос получается очень объемным;
- на стороне клиента необходимо обработать больше информации (вставить ответ на первый запрос во второй).

Если еще раз взглянуть на запрос, можно заметить, что во всех строках ответа одно и то же значение `tenant_id`. Использование этого сервиса предполагает, что страница статистики выводит информацию только для одного приложения (только для одного `tenant_id`).

В `ja_Hipe_Cluster` строки с одним и тем же значением столбца распределения гарантированно попадают на один рабочий узел. Каждый сегмент в распределенной таблице расположен вместе с сегментами из других распределенных таблиц, у которых указаны те же значения столбца распределения. В данном случае можно указать `tenant_id` в качестве столбца распределения:

```
SELECT create_distributed_table('event', 'tenant_id');  
SELECT create_distributed_table('page', 'tenant_id',  
    colocate_with => 'event');
```

Теперь можно использовать тот же запрос, который выполнялся на таблицах, когда они были локальными. Поскольку `tenant_id` используется в `filter` и `join`, `ja_Hipe_Cluster` знает, что запрос использует информацию от совместно размещенных сегментов таблиц, содержащих информацию для этого приложения, и СУБД «Jatoba» может выполнить этот запрос за один шаг.

4.5. Удаление таблиц

Для удаления распределенных таблиц можно использовать стандартную команду DROP TABLE. Как и в случае с обычными таблицами, DROP TABLE удаляет все индексы, правила, триггеры и ограничения, которые существуют для целевой таблицы. Кроме того, она также удаляет сегменты на рабочих узлах и очищает их метаданные.

```
DROP TABLE github_events;
```

4.6. Изменение таблиц

ja_Hire_Cluster автоматически распространяет множество типов DDL, это означает, что изменение распределенной таблицы на узле координатора также обновит сегменты на рабочих элементах. Другие функции DDL требуют распространения вручную, а некоторые другие запрещены, например, те, которые могут изменять столбец распределения. Попытка запустить DDL, который не подходит для автоматического распространения, вызовет ошибку и оставит таблицы на узле координатора неизменными.

Автоматическое распространение может быть включено или отключено с помощью параметра конфигурации.

4.7. Добавление / изменение столбцов

ja_Hire_Cluster автоматически распространяет большинство команд ALTER TABLE. Добавление столбцов или изменение их значений по умолчанию работают так же, как и в базе данных СУБД «Jatoba».

Ниже создается таблица и назначается столбец распределения. На примере этой таблицы будет продемонстрировано добавление и изменение столбцов.

```
CREATE TABLE products (  
    store_id bigint,  
    product_id bigint,  
    name text,  
    price money,  
    CONSTRAINT products_pkey PRIMARY KEY (store_id, product_id)  
);
```

```
SELECT create_distributed_table('products', 'store_id');
```

– Добавление столбца:

```
ALTER TABLE products ADD COLUMN;
```

– Изменение значения по умолчанию:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Значительные изменения в существующем столбце, такие как его переименование или изменение типа данных возможны, но тип данных столбца распределения изменить нельзя. Этот столбец определяет, как данные таблицы распределяются по кластеру `ja_Hipe_Cluster`, и изменение его типа данных потребует перемещения данных.

Попытка сделать это приводит к ошибке:

```
ALTER TABLE products
ALTER COLUMN store_id TYPE text;

/*
ERROR: cannot execute ALTER TABLE command involving partition
column
*/
```

В качестве обходного пути можно рассмотреть возможность изменения столбца распределения, его обновления и повторного изменения.

4.8. Добавление/удаление ограничений

Использование `ja_Hipe_Cluster` позволяет продолжать пользоваться настройками безопасности реляционной базы данных, включая создание ограничений. Из-за свойств распределенной системы `ja_Hipe_Cluster` не будет устанавливаться перекрестные ссылки на ограничения уникальности или ссылочную целостность между рабочими узлами.

При создании внешнего ключа между совместно размещенными распределенными таблицами, необходимо включить в него столбец распределения. При необходимости внешний ключ может быть составным.

Внешние ключи могут быть созданы в следующих ситуациях:

- между двумя локальными (нераспределенными) таблицами;
- между двумя справочными таблицами;
- между справочными таблицами и локальными таблицами (включено по умолчанию через параметр `citus.enable_local_reference_table_foreign_keys` (логическое значение));
- между двумя совместно размещенными распределенными таблицами, если в ключе есть столбец распределения;
- распределенная таблица может ссылаться на справочную таблицу.

Внешние ключи от справочных таблиц к распределенным не поддерживаются.

`ja_Hipe_Cluster` поддерживает все ссылочные действия с внешними ключами от локальных к справочным таблицам, но не поддерживает `ON DELETE/UPDATE CASCADE` в обратном направлении (ссылку от справочных таблиц на локальные).



Первичные ключи и ограничения уникальности должны включать столбец распределения. Если добавить их без использования столбца распределения, возникнет ошибка.

Ниже показано создание таблиц, в которых дальше будут созданы первичные и внешние ключи:

```
CREATE TABLE accounts (  
  id bigserial PRIMARY KEY,  
  name text NOT NULL,  
  image_url text  
);
```

```
CREATE TABLE ads (  
    id bigserial PRIMARY KEY,  
    account_id bigint,  
    name text NOT NULL,  
    image_url text  
);
```

```
CREATE TABLE clicks (  
    id bigserial PRIMARY KEY,  
    account_id bigint,  
    ad_id bigint UNIQUE,  
    clicked_at timestamp without time zone NOT NULL,  
    site_url text NOT NULL  
);
```

- изменение первичных ключей:

```
ALTER TABLE accounts ADD PRIMARY KEY (id);  
ALTER TABLE ads ADD PRIMARY KEY (account_id, id);  
ALTER TABLE clicks ADD PRIMARY KEY (account_id, id);
```

- распределение таблиц:

```
SELECT create_distributed_table('accounts', 'id');  
SELECT create_distributed_table('ads', 'account_id');  
SELECT create_distributed_table('clicks', 'account_id');
```

- добавление внешних ключей:



Внешние ключи можно добавить и до, и после распределения. Но необходимо создать ограничения уникальности на используемых столбцах. Это можно сделать только перед распределением.

```
ALTER TABLE ads ADD CONSTRAINT ads_account_fk  
    FOREIGN KEY (account_id) REFERENCES accounts (id);  
ALTER TABLE clicks ADD CONSTRAINT clicks_ad_fk  
    FOREIGN KEY (account_id, ad_id) REFERENCES ads (account_id,  
id);
```

При создании ограничений уникальности также необходимо использовать столбец распределения:

```
ALTER TABLE ads ADD CONSTRAINT ads_unique_image  
    UNIQUE (account_id, image_url);
```

Ограничения `not null` могут быть применены к любому столбцу (распределенному или нет), поскольку они не требуют проверок между рабочими узлами.

```
ALTER TABLE ads ALTER COLUMN image_url SET NOT NULL;
```

4.9. Использование ограничений “NOT VALID”

В некоторых ситуациях может быть полезно создавать ограничения для новых строк, позволяя при этом существующим несоответствующим строкам оставаться неизменными. `ja_Hipe_Cluster` поддерживает эту функцию для проверки ограничений и внешних ключей.

В качестве примера рассмотрим приложение, которое хранит профили пользователей в справочной таблице.

```
CREATE TABLE users (email text PRIMARY KEY);  
SELECT create_reference_table('users');
```

Со временем в таблицу записывается больше адресов:

```
INSERT INTO users (email) VALUES ('foo@example.com'),  
('hacker12@aol.com'), ('lol');
```

Начать проверять корректность адресов можно через использование ограничения `check`, однако СУБД «Jatoba» не позволит его добавить, если в таблице хранятся строки, которые не соответствуют ограничению. Но можно добавить ограничение, если пометить его как `not valid`:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
ALTER TABLE users
ADD CONSTRAINT syntactic_email
CHECK (email ~
      '^[a-zA-Z0-9.!#$%&'"+/=?^_`{|}~-]+@[a-zA-Z0-9]
      (?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9]
      (?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$'
) NOT VALID;
```

Теперь при создании новых строк будет срабатывать ограничение:

```
INSERT INTO users VALUES ('fake');

/*
ERROR:  new row for relation "users_102010" violates
        check constraint "syntactic_email_102010"
DETAIL:  Failing row contains (fake).
*/
```

Администратор базы данных может попытаться исправить некорректные строки, хранящиеся в таблице, и повторно проверить ограничение:

```
ALTER TABLE users
VALIDATE CONSTRAINT syntactic_email;
```

4.10. Добавление / удаление индексов

ja_Hipe_Cluster поддерживает добавление и удаление индексов.

Добавление индекса происходит с помощью команды:

```
CREATE INDEX clicked_at_idx ON clicks USING BRIN (clicked_at);
```

Удаление индекса происходит с помощью команды:

```
DROP INDEX clicked_at_idx;
```


Создание индекса может помешать нормальной работе базы данных. СУБД «Jatoba» блокирует таблицу для индексации от записи и выполняет всю сборку индекса с помощью одного сканирования таблицы. Другие транзакции могут читать таблицу, но, если они попытаются вставить, обновить или удалить строки в таблице, они будут блокироваться до завершения построения индекса. Это может иметь серьезные последствия, если система представляет собой рабочую базу данных. Индексирование очень больших таблиц может занять много часов.

СУБД «Jatoba» поддерживает построение индексов без блокировки записи. Этот метод вызывается путем указания `CONCURRENTLY` параметра `CREATE INDEX`. При использовании этой опции, СУБД должна выполнить два сканирования таблицы и ожидать завершения всех существующих транзакций, которые потенциально могут изменять или использовать индекс. Это требует больше операций, чем стандартная сборка индекса, и занимает значительно больше времени. Поскольку данный метод позволяет продолжать обычные операции во время построения индекса, метод полезен для добавления новых индексов в продуктовой среде. Дополнительная нагрузка на процессор и ввод-вывод, вызванная созданием индекса, может замедлить другие операции.

```
CREATE INDEX CONCURRENTLY clicked_at_idx ON clicks USING BRIN  
(clicked_at);
```

4.11. Типы и функции

Для распределенных таблиц можно создать новые типы SQL и функции. Однако у создания таких объектов через распределенные операции есть некоторые недостатки.

`ja_Hipe_Cluster` распараллеливает операции над сегментами, используя несколько подключений для каждого рабочего узла. При создании объекта базы данных `ja_Hipe_Cluster` распространяет его на рабочие узлы, используя одно соединение для каждого рабочего узла. Объединение двух операций в одной транзакции может вызвать проблемы, поскольку параллельные соединения не увидят объект, который был создан через одно соединение и еще не зафиксирован.

Ниже представлен блок транзакции, который создает тип, таблицу, загружает данные и распределяет таблицу по узлам:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
BEGIN;

-- создание типов через одно соединение:
CREATE TYPE coordinates AS (x int, y int);
CREATE TABLE positions (object_id text primary key, position
coordinates);

-- загрузка данных через одно соединение:
SELECT create_distributed_table('positions', 'object_id');
\COPY positions FROM 'positions.csv'

COMMIT;
```

Если распространение объекта происходит после параллельной команды в той же транзакции, транзакция больше не может быть завершена, о чем свидетельствует ошибка в блоке кода ниже:

```
BEGIN;

CREATE TABLE items (key text, value text);

-- параллельная загрузка данных:
SELECT create_distributed_table('items', 'key');
\COPY items FROM 'items.csv'
CREATE TYPE coordinates AS (x int, y int);

ERROR:  cannot run type command because there was a parallel
operation on a distributed table in the transaction
```

Есть два способа решения данной проблемы:

– использовать `citus.create_object_propagation` для возвращения к старому поведению распространения объектов. Может возникнуть некоторое несоответствие между тем, какие объекты базы данных существуют на разных узлах;

– использовать `citus.multi_shard_modify_mode` для отключения параллелизма для каждого узла. Загрузка данных в одной транзакции может быть медленнее.

4.12. Служебные функции `ja_Hipe_Cluster`

В данном разделе содержится справочная информация для пользовательских функций, предоставляемых `ja_Hipe_Cluster`. Функции помогают в предоставлении дополнительной функциональности `ja_Hipe_Cluster`, отличной от стандартных команд SQL.

4.12.1. Таблица и сегмент DDL

4.12.1.1 `create_distributed_table`

Функция `create_distributed_table()` используется для определения распределенной таблицы и создания ее сегментов, если это таблица с распределением по хэш-сумме. Функция принимает имя таблицы, столбец распространения и необязательный метод распространения и вставляет соответствующие метаданные, чтобы пометить таблицу как распределенную. Функция по умолчанию использует распределение по хэш-сумме, если не указан другой метод распределения. Если таблица распределена по хэш-сумме, функция также создает рабочие сегменты на основе значения количества сегментов, указанного в конфигурации. Если таблица содержит какие-либо строки, они автоматически распределяются по рабочим узлам.

Аргументы:

`table_name`: имя таблицы, которую необходимо распространить.

`distribution_column`: столбец, по которому должна быть распространена таблица.

`colocate_with`: (необязательно) включение текущей таблицы в группу совместного размещения другой таблицы. По умолчанию таблицы располагаются совместно, когда они распределены по столбцам одного типа с одинаковым количеством сегментов. Для дальнейшего изменения размещения использовать `update_distributed_table_colocation`. Возможные значения `colocate_with` это `default`, `none` (для создания новой группы совместного размещения) или имя другой таблицы для совместного размещения с этой таблицей.

Значение по умолчанию `colocate_with` делает неявное совместное расположение. При не связанном, но случайно использующем один и тот же тип данных для своих столбцов распределения, случайное совместное размещение может снизить производительность при

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

перевесу сегментов. Сегменты таблицы будут без необходимости перемещаться вместе в «cascade». При нарушении этого неявного размещения используется `update_distributed_table_colocation`.

Если новая распределенная таблица не связана с другими таблицами, лучше указать `colocate_with => 'none'`.

shard_count: (необязательно) количество сегментов, которые необходимо создать для новой распределенной таблицы. При указании `shard_count` нельзя указать значение `colocate_with` отличное от `none`. Для изменения количества сегментов существующей таблицы или группы взаимодействий, используется функция `alter_distributed_table`.

Возможные значения: `shard_count` = от 1 до 64000.

Возвращаемое значение: N/A

Пример:

Пример ниже информирует базу данных о том, что таблица `github_events` должна быть распределена по хэш-сумме в столбце `repo_id`.

```
SELECT create_distributed_table('github_events', 'repo_id');

-- альтернативный метод:
SELECT create_distributed_table('github_events', 'repo_id',
colocate_with => 'github_repo');
```

4.12.1.2 truncate_local_data_after_distributing_table

Очистка методом `truncate` всех локальных строк после распределения таблицы и предотвращение сбоев ограничений из-за устаревших локальных записей. `Truncate` распространяется каскадом до таблиц, имеющих внешний ключ к указанной таблице. Если ссылочные таблицы сами по себе не распределены, то операция `truncate` запрещена до тех пор, пока они не будут распределены для защиты ссылочной целостности:

```
ERROR: cannot truncate a table referenced in a foreign key
constraint by a local table
```

Очистка методом truncate данных таблицы локального узла координатора безопасно для распределенных таблиц, поскольку их строки, если они есть, копируются на рабочие узлы во время распространения.

Аргументы:

table_name: имя распределенной таблицы, локальный аналог которой на узле координатора должен быть очищен методом truncate.

Возвращаемое значение: N/A

Пример:

```
-- требуется, чтобы аргумент был распределенной таблицей
SELECT
truncate_local_data_after_distributing_table('public.github_events');
```

4.12.1.3 undistribute_table

Функция undistribute_table() отменяет действие create_distributed_table или create_reference_table. При этом все данные из сегментов перемещаются обратно в локальную таблицу на узле координатора (при условии, что данные могут поместиться), а затем сегменты удаляются.

ja_Hipe_Cluster не будет распределять таблицы, которые имеют или на которые ссылаются внешние ключи, если для аргумента cascade_via_foreign_keys не установлено значение true. Если этот аргумент равен false (или опущен), то нужно вручную удалить нарушающие ограничения внешнего ключа перед отменой распространения.

Аргументы:

table_name: имя распространяемой или справочной таблицы для распределения.

cascade_via_foreign_keys: (необязательно) при значении «true» параметр undistribute_table отменяет распределение всех таблиц, связанных с table_name через внешние ключи. Данный параметр потенциально может повлиять на многие таблицы.

Возвращаемое значение: N/A

Пример:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

В примере ниже таблица `github_events` распределяется, а затем не распределяется.

```
-- распределение таблицы:  
SELECT create_distributed_table('github_events', 'repo_id');  
  
-- отменить распределение и снова распределить:  
SELECT undistribute_table('github_events');
```

4.12.1.4 alter_distributed_table

Функция `alter_distributed_table()` может использоваться для изменения свойств столбца распределения, количества сегментов или взаимодействия распределенной таблицы.

Аргументы:

table_name: имя распределенной таблицы, которая будет изменена.

distribution_column: (необязательно) имя нового столбца рассылки.

shard_count: (необязательно) новое количество сегментов.

colocate_with: (необязательно) таблица, с которой будет размещена текущая распределенная таблица. Возможные значения default: none для начала новой группы размещения или имя другой таблицы, с которой нужно разместить.

cascade_to_colocated: (необязательно) при значении «true» изменения `shard_count` и `colocate_with` также будут применены ко всем таблицам, которые ранее были размещены вместе с таблицей, и взаимодействие будет сохранено. Если значение равно «false», текущее расположение этой таблицы будет нарушено.

Возвращаемое значение: N/A

Пример:

```
-- изменение столбца распределения:  
SELECT alter_distributed_table('github_events',  
distribution_column:='event_id');
```

```
-- изменение количества сегментов всех таблиц в группе
взаимодействия:

SELECT alter_distributed_table('github_events', shard_count:=6,
cascade_to_colocated:=true);

-- изменение взаимодействия:

SELECT alter_distributed_table('github_events',
colocate_with:='another_table');
```

4.12.1.5 alter_table_set_access_method

Функция alter_table_set_access_method() изменяет метод доступа к таблице.

Аргументы:

table_name: имя таблицы, метод доступа к которой будет изменен.

access_method: имя нового метода доступа.

Возвращаемое значение: N/A

Пример:

```
SELECT alter_table_set_access_method('github_events',
'columnar');
```

4.12.1.6 remove_local_tables_from_metadata

Параметр remove_local_tables_from_metadata() удаляет локальные таблицы из метаданных ja_Hipe_Cluster, которые больше не должны там находиться.

Наличие внешних ключей между таблицей и справочной таблицей является причиной нахождения локальной таблицы в метаданных ja_Hipe_Cluster. При отключенном параметре enable_local_reference_foreign_keys ja_Hipe_Cluster не будет управлять метаданными в этой ситуации, а ненужные метаданные могут сохраняться до тех пор, пока не будут очищены вручную.

Аргументы: N/A

Возвращаемое значение: N/A

4.12.1.7 create_reference_table

Функция `create_reference_table()` используется для определения небольшой справочной таблицы или таблицы измерений. Эта функция принимает имя таблицы и создает распределенную таблицу только с одним сегментом, реплицируемую на каждый рабочий узел.

Аргументы:

table_name: Название небольшого измерения или справочной таблицы, которую необходимо распространить.

Возвращаемое значение: N/A

Пример:

Этот пример информирует базу данных о том, что таблица `nation` должна быть определена как справочная таблица:

```
SELECT create_reference_table('nation');
```

4.12.1.8 citus_add_local_table_to_metadata

Функция `citus_add_local_table_to_metadata()` добавляет локальную таблицу СУБД «Jatoba» в метаданные `ja_Hipe_Cluster`. Основным вариантом использования этой функции – сделать локальные таблицы на координаторе доступными с любого узла в кластере. Это полезно при выполнении запросов с других узлов. Данные, связанные с локальной таблицей, остаются на координаторе – рабочим узлам передаются только его схема и метаданные.

Добавление локальных таблиц к метаданным требует незначительных затрат. При добавлении таблицы `ja_Hipe_Cluster` должен отслеживать ее в таблице разделов. Локальные таблицы, добавляемые в метаданные, наследуют те же ограничения, что и справочные таблицы.

При отмене `undistribute_table ja_Hipe_Cluster` автоматически удалит результирующие локальные таблицы из метаданных, что устраняет такие ограничения для этих таблиц.

Аргументы:

table_name: имя таблицы на координаторе, которое будет добавлено в метаданные `ja_Hipe_Cluster`.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

cascade_via_foreign_keys: (необязательно) при значении «true» параметра `citus_add_local_table_to_metadata` в метаданные автоматически добавляются другие таблицы, которые связаны внешним ключом с данной таблицей. Данный параметр потенциально может повлиять на многие таблицы.

Возвращаемое значение: N/A

Пример:

Пример ниже информирует базу данных о том, что таблица `nation` должна быть определена как локальная таблица координатора, доступная с любого узла:

```
SELECT citus_add_local_table_to_metadata('nation');
```

4.12.1.9 mark_tables_colocated

Функция `mark_tables_colocated()` помещает список (цели) в ту же группу совместного размещения, что и распределенная таблица (источник). Если источник еще не включен в группу, эта функция создает ее и назначает ей источник и целевые объекты.

Совместное размещение таблиц должно выполняться во время распространения таблицы с помощью `colocate_with` параметра `create_distributed_table`.

Для прерывания взаимодействия таблиц можно использовать `update_distributed_table_colocation`.

Аргументы:

source_table_name: имя распределенной таблицы, группе совместного размещения которой будут назначены целевые объекты.

target_table_names: массив имен распределенных целевых таблиц, должен быть непустым. Эти распределенные таблицы должны соответствовать исходной таблице в:

- способе распространения;
- типе столбца распределения;
- количестве сегментов.

В противном случае `ja_Hipe_Cluster` выдаст сообщение об ошибке.

Возвращаемое значение: N/A

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Пример:

В примере ниже продукты и элементы line_items помещаются в ту же группу совместного размещения, что и хранилище. Предполагается, что все эти таблицы распределены по столбцу с соответствующим типом «store id».

```
SELECT mark_tables_colocated('stores', ARRAY['products',  
'line_items']);
```

4.12.1.10 update_distributed_table_colocation

Функция `update_distributed_table_colocation()` используется для обновления взаимодействия распределенной таблицы. Эта функция также может использоваться для прерывания взаимодействия распределенной таблицы. `ja_Hipe_Cluster` неявно разместит две таблицы, если столбец распределения имеет один и тот же тип. Это может быть полезно, если таблицы связаны и над ними будет выполняться операция `join`. Если таблицы А и В размещены совместно и таблица А будет перебалансирована, таблица В также будет перебалансирована. Если таблица В не имеет идентификатора реплики, перебалансировка завершится ошибкой. Функция может быть полезной в случае нарушения неявного размещения.

Данная функция не перемещает какие-либо данные.

Аргументы:

table_name: имя таблицы, расположение которой будет обновлено.

colocate_with: таблица, с которой должна быть размещена таблица.

При изменении расположения таблицы указать `colocate_with => 'none'`.

Возвращаемое значение: N/A

Пример:

В примере показано, что расположение таблицы А обновляется как расположение таблицы В.

```
SELECT update_distributed_table_colocation('A', colocate_with  
=> 'B');
```

Предположим, что таблица А и таблица В размещены совместно (возможно, неявно). Для того, чтобы разорвать размещение:

```
SELECT update_distributed_table_colocation('A', colocate_with  
=> 'none');
```

Предположим, что таблица А, таблица В, таблица С и таблица D расположены совместно. Для размещения таблиц А и В вместе, и таблиц С и D вместе ввести команду:

```
SELECT update_distributed_table_colocation('C', colocate_with  
=> 'none');  
  
SELECT update_distributed_table_colocation('D', colocate_with  
=> 'C');
```

Если имеется распределенная хэш-таблица с именем none, можно обновить ее расположение командой:

```
SELECT update_distributed_table_colocation('"none"',  
colocate_with => 'some_other_hash_distributed_table');
```

4.12.1.11 create_distributed_function

Передаёт функцию от узла-координатора рабочим узлам и помечает ее для распределенного выполнения. ja_Hipe_Cluster использует значение «distribution argument», когда на координаторе вызывается распределенная функция, чтобы выбрать рабочий узел для запуска функции. Выполнение функции на рабочих элементах увеличивает параллелизм и может приблизить код к данным в сегментах для снижения задержек.

Путь поиска СУБД не передается от координатора к рабочим узлам во время выполнения распределенной функции, поэтому код распределенной функции должен полностью определять имена объектов базы данных.

Уведомления, выдаваемые функциями, не будут отображаться пользователю.

Аргументы:

function_name: имя функции, которая будет распространяться. Имя должно включать типы параметров функции в круглых скобках, поскольку несколько функций могут иметь одно и то же имя в СУБД «Jatoba». Например, 'foo(int)' отличается от 'foo(int, text)'.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

distribution_arg_name: (необязательно) имя аргумента, с помощью которого будет распространяться. Для удобства (или если аргументы функции не имеют имен) допускается использование позиционного заполнителя, например, '\$1'. Если этот параметр не указан, то функция с именем function_name создается для рабочих узлов. Если рабочие узлы будут добавлены в будущем, функция также будет автоматически создана там.

colocate_with: (необязательно) распределенная функция считывает или записывает в распределенную таблицу (или таблицы совместного размещения), обязательно указать имя этой таблицы, используя параметр colocate_with. Каждый вызов функции будет выполняться на рабочем узле, содержащем соответствующие сегменты.

Возвращаемое значение: N/A

Пример:

```
-- таблица event_responses распределяется по event_id
CREATE OR REPLACE FUNCTION
    register_for_event(p_event_id int, p_user_id int)
RETURNS void LANGUAGE plpgsql AS $fn$
BEGIN
    INSERT INTO event_responses VALUES ($1, $2, 'yes')
    ON CONFLICT (event_id, user_id)
    DO UPDATE SET response = EXCLUDED.response;
END;
$fn$;

-- распределение функции среди рабочих узлов с использованием
аргумента p_event_id для определения на какой сегмент влияет
каждый вызов и совместное использование с event_responses,
которые функция обновляет
SELECT create_distributed_function(
    'register_for_event(int, int)', 'p_event_id',
    colocate_with := 'event_responses'
);
```

4.12.1.12 alter_columnar_table_set

Функция `alter_columnar_table_set()` изменяет настройки в колоночной таблице. Вызов этой функции для таблицы, не состоящей из столбцов, выдаст ошибку. Все аргументы, кроме имени таблицы, являются необязательными.

Для просмотра текущих параметров для всех колоночных таблиц следует обратиться к таблице:

```
SELECT * FROM columnar.options;
```

Значения по умолчанию параметров столбцов для вновь созданных таблиц могут быть переопределены с помощью параметров:

- `columnar.compression;`
- `columnar.compression_level;`
- `columnar.stripe_row_count;`
- `columnar.chunk_row_count.`

Аргументы:

table_name: имя колоночной таблицы.

chunk_row_count: (необязательно) максимальное количество строк в блоке для вновь вставленных данных. Существующие фрагменты данных не будут изменены и могут содержать больше строк, чем это максимальное значение. Значение по умолчанию равно 10000.

stripe_row_count: (необязательно) максимальное количество строк на полосу для вновь вставленных данных. Существующие полосы данных не будут изменены и могут содержать больше строк, чем это максимальное значение. Значение по умолчанию равно 150000.

compression: (необязательно) `[none|pglz|zstd|lz4|lz4hc]` тип сжатия для вновь вставленных данных. Существующие данные не будут повторно сжаты или распакованы. Значение по умолчанию – `zstd` (если поддержка была скомпилирована).

compression_level: (необязательно) допустимые настройки от 1 до 19. Если метод сжатия не поддерживает выбранный уровень, вместо него будет выбран ближайший уровень.

Возвращаемое значение: N/A

Пример:

```
SELECT alter_columnar_table_set(  
    'my_columnar_table',  
    compression => 'none',  
    stripe_row_count => 10000);
```

4.12.1.13 create_time_partitions

Функция `create_time_partitions()` создает разделы заданного интервала для покрытия заданного диапазона времени.

Аргументы:

table_name: (regclass) таблица, для которой нужно создавать новые разделы. Таблица должна быть разделена на один столбец типа `date`, `timestamp` или `timestampz`.

partition_interval: интервал времени, например, `'2 hours'`, или `'1 month'`, для использования при настройке диапазонов для новых разделов.

end_at: (`timestampz`) создание разделов до этого времени. Последний раздел будет содержать точку `end_at` и последующие разделы не будут созданы.

start_from: (`timestampz`, необязательно) выбрать первый раздел так, чтобы он содержал точку `start_from`. Значение по умолчанию = `now()`.

Возвращаемое значение:

`True` – требуется создать новые разделы;

`False` – разделы существуют.

Пример:

```
-- создание ежемесячных разделов на год вперед в таблице foo,  
начиная с текущего времени
```

```
SELECT create_time_partitions(  
    table_name          := 'foo',  
    partition_interval := '1 month',  
    end_at              := now() + '12 months'  
);
```

4.12.1.14 drop_old_time_partitions

Функция `drop_old_time_partitions()` удаляет все разделы, интервалы между которыми находятся перед заданной меткой времени.

В дополнение к использованию данной функции можно рассмотреть `alter_old_partitions_set_access_method` для сжатия старых разделов с помощью колоночного хранилища.

Аргументы:

table_name: таблица (regclass), для которой нужно удалить разделы. Таблица должна быть разделена на один столбец типа `date`, `timestamp` или `timestampz`.

older_than: (timestampz) удаление разделов, верхний диапазон которых меньше или равен `older_than`.

Возвращаемое значение: N/A

Пример:

```
-- удаление разделов, которым больше одного года  
CALL drop_old_time_partitions('foo', now() - interval '12  
months');
```

4.12.1.15 alter_old_partitions_set_access_method

В случае использования данных временных рядов таблицы разделяются по времени, а старые разделы сжимаются в колоночное хранилище, доступное только для чтения.

Аргументы:

parent_table_name: таблица (regclass), для которой нужно изменить разделы. Таблица должна быть разделена на один столбец типа `date`, `timestamp` или `timestampz`.

older_than: (timestampz) изменение разделов, верхний диапазон которых меньше или равен older_than.

new_access_method: (имя) либо «heap» для хранения на основе строк, либо «columnar» для колоночного хранения.

Возвращаемое значение: N/A

Пример:

```
CALL alter_old_partitions_set_access_method(  
    'foo', now() - interval '6 months',  
    'columnar'  
);
```

4.12.2. Метаданные / Информация о конфигурации

4.12.2.1 citus_add_node



Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция citus_add_node() регистрирует добавление нового узла в кластер в таблице метаданных ja_Hipe_Cluster pg_dist_node. Функция также копирует справочные таблицы на новый узел.

При запуске citus_add_node в кластере с одним узлом требуется сначала запустить citus_set_coordinator_host.

Аргументы:

nodename: DNS-имя или IP-адрес нового добавляемого узла.

nodeport: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

groupid: группа из одного основного сервера и его вторичных серверов, необходима только для потоковой репликации. Требуется установить groupid больше нуля, поскольку ноль зарезервирован для узла координатора. Значение по умолчанию = -1.

noderole: является ли он «основным» или «вторичным». По умолчанию «основной»

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

nodecluster: имя кластера. По умолчанию «default»

Возвращаемое значение: столбец nodeid из недавно вставленной строки в pg_dist_node.

Пример:

```
select * from citus_add_node('new-node', 12345);
```

Или с использованием IP-адреса узла:

```
select * from citus_add_node('ip-address', 12345);
```

Пример:

```
select * from citus_add_node('10.96.1.83', 12345);
```

Вывод SQL-команды представлен на рисунке 4.2.

```

citus_add_node
-----
              7
(1 row)
```

Рисунок 4.2 – Вывод результата функции citus_add_node

4.12.2.2 citus_update_node



Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция citus_update_node() изменяет имя хоста и порт для узла, зарегистрированного в таблице метаданных ja_Hipe_Cluster pg_dist_node.

Аргументы:

node_id: идентификатор из таблицы pg_dist_node.

node_name: обновленное DNS-имя или IP-адрес для узла.

node_port: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

Возвращаемое значение: N/A

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Синтаксис команды:

```
select * from citus_update_node(123, 'ip-address', 5432);
```

Пример:

```
select * from citus_update_node(123, '10.116.102.63', 5432);
```

Или с использованием DNS-имени узла:

```
select * from citus_update_node(123, 'worker1', 5432);
```

4.12.2.3 citus_set_node_property

Функция `citus_set_node_property()` изменяет свойства в таблице метаданных `ja_Hipe_Cluster pg_dist_node`. В настоящее время может изменять только свойство `shouldhaveshard`s.

Аргументы:

node_name: DNS-имя или IP-адрес для узла.

node_port: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

property: поддерживается только столбец для изменения.

value: новое значение для столбца.

Возвращаемое значение: N/A

Пример:

```
SELECT * FROM citus_set_node_property('localhost', 5433,  
'shouldhaveshard
```

s', false);

Или с использованием IP-адреса узла:

```
SELECT * FROM citus_set_node_property('127.0.0.1', 5433,  
'shouldhaveshard
```

s', false);

4.12.2.4 citus_add_inactive_node



Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция `citus_add_inactive_node`, аналогичная `citus_add_node`, регистрирует новый узел в `pg_dist_node`. Функция помечает новый узел как неактивный, то есть там не будут размещены сегменты. Также не копируются справочные таблицы в новый узел.

Аргументы:

nodename: DNS-имя или IP-адрес нового добавляемого узла.

nodeport: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

groupid: группа из одного основного сервера и нуля или более дополнительных серверов, соответствующая только для потоковой репликации. По умолчанию = -1

node role: является ли он «основным» или «вторичным». По умолчанию «основной»

nodecluster: имя кластера. По умолчанию «default»

Возвращаемое значение: столбец `nodeid` из недавно вставленной строки в `pg_dist_node`.

Синтаксис:

```
select * from citus_add_inactive_node('node-name', 12345);
```

Пример:

```
select * from citus_add_inactive_node('worker3', 12345);
```

Или с использованием IP-адреса узла:

```
select * from citus_add_inactive_node('ip-address', 12345);
```

Пример:

```
select * from citus_add_inactive_node('10.96.1.83', 12345);
```

Вывод SQL-команды представлен на рисунке 4.3.

```
citus_add_inactive_node
-----
7
(1 row)
```

Рисунок 4.3 – Вывод результата функции citus_add_inactive_node

4.12.2.5 citus_activate_node



Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция citus_activate_node помечает узел как активный в таблице метаданных ja_Hipe_Cluster pg_dist_node и копирует справочные таблицы в узел. Полезно для узлов, добавленных через citus_add_inactive_node.

Аргументы:

nodename: DNS-имя или IP-адрес нового добавляемого узла.

nodeport: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

Возвращаемое значение: столбец nodeid из недавно вставленной строки в pg_dist_node.

Синтаксис:

```
select * from citus_activate_node('node-name', 12345);
```

Пример:

```
select * from citus_activate_node('worker2', 12345);
```

Или с использованием IP-адреса узла:

```
select * from citus_activate_node('ip-address', 12345);
```

Пример:


```
select * from citus_activate_node('10.116.102.63', 12345);
```

Вывод SQL-команды представлен на рисунке 4.4.

```
citus_activate_node
-----
7
(1 row)
```

Рисунок 4.4 – Вывод результата функции citus_activate_node

4.12.2.6 citus_disable_node

 Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция citus_disable_node противоположна citus_activate_node. Функция помечает узел как неактивный в таблице метаданных pg_dist_node, временно удаляя его из кластера. Функция также удаляет все размещения справочной таблицы с отключенного узла. Чтобы повторно активировать узел нужно запустить citus_activate_node его снова.

Аргументы:

nodename: DNS-имя или IP-адрес узла, который должен быть отключен.

nodeport: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

Возвращаемое значение: N/A

Синтаксис:

```
select * from citus_disable_node('node-name', 12345);
```

Пример:

```
select * from citus_disable_node('worker2', 12345);
```

Или с использованием IP-адреса узла:

```
select * from citus_disable_node('ip-address', 12345);
```

Пример:

```
select * from citus_disable_node('10.116.102.63', 12345);
```

4.12.2.7 citus_add_secondary_node



Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция `citus_add_secondary_node()` регистрирует новый вторичный узел в кластере для существующего основного узла. Функция обновляет таблицу метаданных `pg_dist_node`.

Аргументы:

nodename: DNS-имя или IP-адрес нового добавляемого узла.

nodeport: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

primaryname: DNS-имя или IP-адрес основного узла для вторичного.

primaryport: порт, на котором СУБД «Jatoba» прослушивает основной узел.

nodecluster: имя кластера. По умолчанию «default»

Возвращаемое значение: столбец `nodeid` для вторичного узла, вставленная строка в `pg_dist_node`.

Пример:

```
select * from citus_add_secondary_node('new-node', 12345,  
'primary-node', 12345);
```

Или с использованием IP-адреса узла:

```
select * from citus_add_secondary_node('ip-address-secnode',  
12345, 'ip-address-primary-node', 12345);
```

Вывод SQL-команды представлен на рисунке 4.5.

```
citus_add_secondary_node  
-----  
7  
(1 row)
```

Рисунок 4.5 – Вывод результата функции `citus_add_secondary_node`

4.12.2.8 citus_remove_node



Для запуска этой функции требуется доступ суперпользователя к базе данных.

Функция `citus_remove_node()` удаляет указанный узел из таблицы метаданных `pg_dist_node`. Эта функция выдаст ошибку, если на этом узле существуют места размещения сегментов. Перед использованием функции сегменты необходимо переместить с этого узла.

Аргументы:

nodename: DNS-имя удаляемого узла.

nodeport: порт, на котором СУБД «Jatoba» прослушивает рабочий узел.

Возвращаемое значение: N/A

Пример:

```
select citus_remove_node('node-name', 12345);
```

Или с использованием IP-адреса узла:

```
select citus_remove_node('ip-address', 12345);
```

Вывод SQL-команды представлен на рисунке 4.6.

```
citus_remove_node
-----
(1 row)
```

Рисунок 4.6 – Вывод результата функции `citus_remove_node`

4.12.2.9 citus_get_active_worker_nodes

Функция `citus_get_active_worker_nodes()` возвращает список имен активных рабочих узлов и номеров портов.

Аргументы: N/A

Возвращаемое значение: список записей, где каждая запись содержит следующую информацию:

node_name: DNS-имя рабочего узла

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

node_port: порт на рабочем узле, на котором прослушивается сервер базы данных

Пример:

```
SELECT * from citus_get_active_worker_nodes();
```

Вывод SQL-команды представлен на рисунке 4.7.

node_name	node_port
localhost	9700
localhost	9702
localhost	9701

(3 rows)

Рисунок 4.7 – Вывод результата функции citus_get_active_worker_nodes

4.12.2.10 citus_backend_gpid

Функция citus_backend_gpid() возвращает глобальный идентификатор процесса (global process identifier – GPID) для серверной части СУБД «Jatoba», обслуживающей текущий сеанс. Идентификатор GPID кодирует как узел в кластере ja_Hipe_Cluster, так и идентификатор процесса операционной системы Jatoba на этом узле.

ja_Hipe_Cluster расширяет сигнальные функции сервера pg_cancel_backend() СУБД «Jatoba» и pg_terminate_backend() позволяет им принимать идентификаторы GPID. В ja_Hipe_Cluster вызов этих функций на одном узле может повлиять на серверную часть, работающую на другом узле.

Аргументы: N/A

Возвращаемое значение: целочисленный GPID вида (NodeID * 10 000 000 000) + ProcessId.

Пример:

```
SELECT citus_backend_gpid();
```

Вывод SQL-команды представлен на рисунке 4.8.


```
citus_backend_gpid  
-----  
10000002055
```

Рисунок 4.8 – Вывод результата функции citus_backend_gpid

4.12.2.11 citus_check_cluster_node_health

Проверка подключений между всеми узлами. Если имеется N узлов, эта функция проверяет все N^2 соединения между ними.

Аргументы: N/A

Возвращаемое значение: список записей, где каждая запись содержит следующую информацию:

from_nodename: DNS-имя исходного рабочего узла

from_nodeport: порт на исходном рабочем узле, на котором прослушивается сервер базы данных

to_nodename: DNS-имя рабочего узла назначения

to_nodeport: порт на рабочем узле назначения, на котором прослушивается сервер базы данных

result: можно ли установить соединение

Пример:

```
SELECT * FROM citus_check_cluster_node_health();
```

Вывод SQL-команды представлен на рисунке 4.9.

from_nodename	from_nodeport	to_nodename	to_nodeport	result
localhost	1400	localhost	1400	t
localhost	1400	localhost	1401	t
localhost	1400	localhost	1402	t
localhost	1401	localhost	1400	t
localhost	1401	localhost	1401	t
localhost	1401	localhost	1402	t
localhost	1402	localhost	1400	t
localhost	1402	localhost	1401	t
localhost	1402	localhost	1402	t

(9 rows)

Рисунок 4.9 – Вывод результата функции citus_check_cluster_node_health

4.12.2.12 citus_set_coordinator_host

Эта функция требуется при добавлении рабочих узлов в кластер ja_Hipe_Cluster, который изначально был создан как кластер с одним узлом. Когда координатор регистрирует новый рабочий узел, он добавляет имя хоста координатора из значения citus.local_hostname (текст), который используется по умолчанию localhost. Рабочий узел попытается подключиться к localhost, чтобы взаимодействовать с координатором, что неправильно.

Системный администратор должен выполнить вызов citus_set_coordinator_host перед вызовом citus_add_node в кластере с одним узлом.

Аргументы:

host: DNS-имя узла-координатора

port: (необязательно) порт, на котором координатор перечисляет соединения Jatoba. По умолчанию используется значение current_setting('port').

node_role: (необязательно) по умолчанию используется значение primary.

node_cluster: (необязательно) по умолчанию используется значение default.

Возвращаемое значение: N/A

Пример:

```
-- в одноузловом кластере определить, как рабочие узлы будут
подключаться

SELECT citus_set_coordinator_host('coord', 5432);

-- добавить рабочий узел

SELECT * FROM citus_add_node('worker1', 5432);
```

4.12.2.13 master_get_table_metadata

Функция `master_get_table_metadata()` может использоваться для возврата метаданных, связанных с распространением, для распределенной таблицы. Эти метаданные включают идентификатор отношения, тип хранилища, метод распространения, столбец распространения, количество репликаций (`deprecated`), максимальный размер сегментов и политику размещения сегментов для этой таблицы. Функция запрашивает таблицы метаданных `ja_Hipe_Cluster` для получения требуемой информации и объединяет ее в запись, прежде чем возвращать его пользователю.

Аргументы:

table_name: имя распределенной таблицы, для которой нужно получить метаданные.

Возвращаемое значение: запись, содержащая следующую информацию:

logical_relid: идентификатор распределенной таблицы. Это значение ссылается на столбец `relfilenode` в таблице системного каталога `pg_class`.

part_storage_type: тип хранилища, используемого для таблицы. Может быть 't' (стандартная таблица), 'f' (внешняя таблица) или 'c' (колоночная таблица).

part_method: метод распространения, используемый для таблицы. Должно быть 'h' (хэш).

part_key: столбец распределения для таблицы.

part_replica_count: (устаревший) текущий счетчик репликации сегментов.

part_max_size: текущий максимальный размер фрагмента в байтах.

part_placement_policy: политика размещения сегментов, используемая для размещения сегментов таблицы. Может быть 1 (сначала локальный узел) или 2 (циклический перебор).

Пример:

В приведенном ниже примере извлекаются и отображаются метаданные таблицы для `github_events`.

```
SELECT * from master_get_table_metadata('github_events');
```

Вывод SQL-команды представлен на рисунке 4.10.

logical_relid	part_storage_type	part_method	part_key	part_replica_count	part_max_size	part_placement_policy
24180	t	h	repo_id	1	1073741824	2

(1 row)

Рисунок 4.10 – Вывод результата функции `master_get_table_metadata`

4.12.2.14 get_shard_id_for_distribution_column

`ja_Hipe_Cluster` присваивает сегменту каждую строку распределенной таблицы на основе значения столбца распределения строки и метода распространения таблицы. В большинстве случаев точное сопоставление является низкоуровневой детализацией, которую администратор базы данных может игнорировать. Может быть полезно определить сегмент строки для задач ручного обслуживания базы данных. Функция `get_shard_id_for_distribution_column` предоставляет эту информацию для таблиц с распределением хэшей, а также справочных таблиц.

Аргументы:

table_name: распределенная таблица.

distribution_value: значение столбца распределения.

Возвращаемое значение: идентификатор сегмента `ja_Hipe_Cluster` ассоциируется со значением столбца распределения для данной таблицы.

Пример:

```
SELECT get_shard_id_for_distribution_column('my_table', 4);
```

Вывод SQL-команды представлен на рисунке 4.11.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```

get_shard_id_for_distribution_column
-----
540007
(1 row)

```

Рисунок 4.11 – Вывод результата функции `get_shard_id_for_distribution_column`

4.12.2.15 `column_to_column_name`

Преобразует partkey столбец `pg_dist_partition` в текстовое имя столбца. Это полезно для определения столбца распределения распределенной таблицы.

Аргументы:

table_name: распределенная таблица.

column_var_text: значение partkey в таблице `pg_dist_partition`.

Возвращаемое значение: имя `table_name` столбца распределения.

Пример:

```

-- получение имени столбца распределения для таблицы продуктов
SELECT column_to_column_name(logicalrelid, partkey) AS
dist_col_name
FROM pg_dist_partition
WHERE logicalrelid='products'::regclass;

```

Вывод SQL-команды представлен на рисунке 4.12.

dist_col_name
company_id

Рисунок 4.12 – Вывод результата функции `column_to_column_name`

4.12.2.16 `citus_relation_size`

Получение дискового пространства, используемое всеми сегментами указанной распределенной таблицы. Это включает в себя размер «main fork», но исключает карту видимости и карту свободного места для сегментов.

Аргументы:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

logicalrelid: имя распределенной таблицы.

Возвращаемое значение: размер в байтах как bigint.

Пример:

```
SELECT pg_size_pretty(citus_relation_size('github_events'));
```

Вывод SQL-команды представлен на рисунке 4.13.

```
pg_size_pretty
-----
23 MB
```

Рисунок 4.13 – Вывод результата функции citus_relation_size

4.12.2.17 citus_table_size

Получение дискового пространства, используемое всеми сегментами указанной распределенной таблицы, исключая индексы (но включая TOAST, карту свободного места и карту видимости).

Аргументы:

logicalrelid: имя распределенной таблицы.

Возвращаемое значение: размер в байтах как bigint.

Пример:

```
SELECT pg_size_pretty(citus_table_size('github_events'));
```

Вывод SQL-команды представлен на рисунке 4.14.

```
pg_size_pretty
-----
37 MB
```

Рисунок 4.14 – Вывод результата функции citus_table_size

4.12.2.18 citus_total_relation_size

Получение общего дискового пространства, используемое всеми сегментами указанной распределенной таблицы, включая все индексы и данные TOAST.

Аргументы:

logicalrelid: имя распределенной таблицы.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Возвращаемое значение: размер в байтах как bigint.

Пример:

```
SELECT  
pg_size_pretty(citus_total_relation_size('github_events'));
```

Вывод SQL-команды представлен на рисунке 4.15.

```
pg_size_pretty  
-----  
73 MB
```

Рисунок 4.15 – Вывод результата функции citus_total_relation_size

4.12.2.19 citus_stat_statements_reset

Удаляет все строки из citus_stat_statements.



Данная функция работает независимо от pg_stat_statements_reset().

Для сброса всей статистики необходимо вызвать обе функции.

Аргументы: N/A

Возвращаемое значение: Нет

4.12.3. Функции управления кластером и восстановления

4.12.3.1 citus_move_shard_placement

Функция перемещает данный сегмент (сегменты, расположенные рядом с ним) с одного узла на другой. Обычно они используются косвенно во время перебалансировки сегментов, а не вызываются напрямую администратором базы данных.

Существует два способа перемещения данных: блокирование или неблокирование. Подход блокировки означает, что во время перемещения все изменения в сегменте приостанавливаются. Второй способ, который позволяет избежать блокировки записи в сегмент, основан на логической репликации СУБД «Jatoba».

После успешной операции перемещения сегменты в исходном узле удаляются. Если перемещение завершается неудачно в какой-либо момент, эта функция выдает ошибку и оставляет исходные и целевые узлы неизменными.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Аргументы:

shard_id: идентификатор фрагмента, который нужно переместить.

source_node_name: DNS-имя узла, на котором присутствует исправное размещение сегментов («исходный» узел).

source_node_port: порт на исходном рабочем узле, на котором прослушивается сервер базы данных.

target_node_name: DNS-имя узла, на котором присутствует недопустимое размещение сегментов («целевой» узел).

target_node_port: порт на целевом рабочем узле, на котором прослушивается сервер базы данных.

shard_transfer_mode: (необязательно) указание метода репликации – использовать логическую репликацию СУБД «Jatoba» или команду копирования между рабочими узлами.
Возможные значения:

auto: требование идентификации реплики, если возможна логическая репликация, в противном случае – устаревшее поведение (например, для восстановления сегментов). Это значение по умолчанию.

force_logical: использование логической репликации, даже если таблица не имеет идентификатора реплики. Любые одновременные инструкции update/delete для таблицы завершатся ошибкой во время репликации.

block_writes: использование копирования (блокирующую запись) для таблиц, в которых отсутствует первичный ключ или идентификатор реплики.

Возвращаемое значение: N/A

Синтаксис:

```
SELECT citus_move_shard_placement(12345, 'from_host', 5432,  
'to_host', 5432);
```

Пример с использованием DNS-имен узлов:


```
SELECT citus_move_shard_placement(12345, 'worker1', 5432,  
'worker2', 5432);
```

4.12.3.2 rebalance_table_shards

Функция `rebalance_table_shards()` перемещает сегменты заданной таблицы, чтобы они были равномерно распределены между рабочими элементами. Функция сначала вычисляет список перемещений, которые ей необходимо выполнить, чтобы гарантировать, что кластер сбалансирован в пределах заданного порога. Затем он перемещает размещения сегментов одно за другим с исходного узла на узел назначения и обновляет соответствующие метаданные сегментов, чтобы отразить перемещение.

Каждому сегменту присваивается стоимость при определении того, являются ли сегменты «равномерно распределенными». По умолчанию каждый сегмент имеет одинаковую стоимость (значение 1), поэтому распределение для выравнивания затрат между рабочими узлами аналогично выравниванию количества сегментов для каждого. Стратегия постоянных затрат называется «`by_shard_count`» и является стратегией перебалансировки по умолчанию.

Стратегия по умолчанию подходит в этих обстоятельствах:

- сегменты имеют примерно одинаковый размер;
- сегменты получают примерно одинаковый объем трафика;
- все рабочие узлы имеют одинаковый размер / тип;
- сегменты не были привязаны к конкретным рабочим узлам.

Если какое-либо из этих предположений не выполняется, то перебалансировка по умолчанию может привести к неправильному плану. В этом случае настраивается стратегия с помощью параметра `rebalance_strategy`.

Рекомендуется вызвать `get_rebalance_table_shards_plan` перед запуском `rebalance_table_shards`, чтобы увидеть и проверить действия, которые необходимо выполнить.

Аргументы:

table_name: (необязательно) имя таблицы, сегменты которой необходимо перебалансировать. При значении равном NULL необходимо выполнить перебалансировку всех существующих групп размещения.

threshold: (необязательно) число с плавающей запятой между 0.0 и 1.0, которое указывает на максимальный коэффициент отличия использования узла от среднего использования.

Указание 0.1 приведет к тому, что балансировщик сегментов попытается сбалансировать все узлы так, чтобы они содержали одинаковое количество сегментов $\pm 10\%$. В частности, балансировщик сегментов попытается свести использование всех рабочих узлов к $(1 - \text{пороговому значению}) * \text{average_utilization} \dots (1 + \text{пороговое значение}) * \text{диапазон average_utilization}$.

max_shard_moves: (необязательно) максимальное количество сегментов для перемещения.

excluded_shard_list: (необязательно) идентификаторы сегментов, которые не следует перемещать во время операции перебалансировки.

shard_transfer_mode: (необязательно) указание метода репликации – использовать логическую репликацию Jatoba или команду копирования между рабочими узлами. Возможные значения:

auto: требование идентификации реплики, если возможна логическая репликация (например, для восстановления сегментов). Это значение по умолчанию.

force_logical: использование логической репликации, даже если таблица не имеет идентификатора реплики. Любые одновременные инструкции update / delete для таблицы завершатся ошибкой во время репликации.

block_writes: использование копирования (блокирующую запись) для таблиц, в которых отсутствует первичный ключ или идентификатор реплики.

drain_only: (необязательно) при значении true необходимо переместить сегменты с рабочих узлов, для которых shouldhaveshard в таблице рабочих узлов установлено значение false; другие сегменты не перемещать.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

rebalance_strategy: (необязательно) имя стратегии в таблице стратегий перебалансировки. Если этот аргумент опущен, функция выбирает стратегию по умолчанию, как указано в таблице.

Возвращаемое значение: N/A

Пример:

В приведенном ниже примере предпринята попытка перебалансировать сегменты таблицы `github_events` в пределах порогового значения по умолчанию:

```
SELECT rebalance_table_shards('github_events');
```

В этом примере предпринята попытка перебалансировать таблицу `github_events` без перемещения сегментов с идентификаторами 1 и 2.

```
SELECT rebalance_table_shards('github_events',
excluded_shard_list:='{1,2}');
```

4.12.3.3 get_rebalance_table_shards_plan

Вывод запланированных перемещений сегментов `rebalance_table_shards` без их выполнения. Функция `get_rebalance_table_shards_plan` может вывести план, немного отличающийся от того, что будет делать вызов `rebalance_table_shards` с теми же аргументами. Это может произойти из-за того, что они не выполняются одновременно, поэтому данные о кластере могут отличаться между вызовами.

Аргументы:

Аргументы, что и у `rebalance_table_shards`: `relation`, `threshold`, `max_shard_moves`, `excluded_shard_list`, and `drain_only`.

Возвращаемое значение: записи, содержащие столбцы:

table_name: таблица, сегменты которой будут перемещаться

shardid: рассматриваемый фрагмент

shard_size: размер в байтах

sourcename: имя хоста исходного узла

sourceport: порт исходного узла

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

targetname: имя хоста узла назначения

targetport: порт узла назначения

4.12.3.4 get_rebalance_progress

При начале перебалансировки сегментов, функция `get_rebalance_progress()` отображает ход выполнения каждого задействованного сегмента, отслеживает запланированные и выполненные действия `rebalance_table_shards()`.

Аргументы: N/A

Возвращаемое значение: записи, содержащие столбцы:

sessionid: Postgres PID монитора перебалансировки

table_name: таблица, сегменты которой перемещаются

shardid: рассматриваемый фрагмент

shard_size: размер фрагмента в байтах

sourcename: имя хоста исходного узла

sourceport: порт исходного узла

targetname: имя хоста узла назначения

targetport: порт узла назначения

progress: 0 = ожидание перемещения; 1 = перемещение; 2 = завершение

source_shard_size: размер сегмента на исходном узле в байтах

target_shard_size: размер сегмента на целевом узле в байтах

Пример:

```
SELECT * FROM get_rebalance_progress();
```

Вывод SQL-команды представлен на рисунке 4.16.

sessionid	table_name	shardid	shard_size	sourcename	sourceport	targetname	targetport	progress	source_shard_size	target_shard_size
7083	foo	102008	1204224	n1.foobar.com	5432	n4.foobar.com	5432	0	1204224	0
7083	foo	102009	1802240	n1.foobar.com	5432	n4.foobar.com	5432	0	1802240	0
7083	foo	102018	614400	n2.foobar.com	5432	n4.foobar.com	5432	1	614400	354400
7083	foo	102019	8192	n3.foobar.com	5432	n4.foobar.com	5432	2	0	8192

Рисунок 4.16 – Вывод результата функции `get_rebalance_progress`

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

4.12.3.5 citus_add_rebalance_strategy

Добавление строки к pg_dist_rebalance_strategy.

Аргументы:

name: идентификатор для новой стратегии

shard_cost_function: определяет функцию, используемую для определения «стоимости» каждого сегмента

node_capacity_function: определяет функцию для измерения пропускной способности узла

shard_allowed_on_node_function: определяет функцию, которая определяет, какие сегменты могут быть размещены на каких узлах

default_threshold: пороговое значение с плавающей запятой, которое определяет, насколько точно совокупная стоимость сегментов должна быть сбалансирована между узлами

minimum_threshold: (необязательно) столбец защиты, который содержит минимальное значение, допустимое для порогового аргумента rebalance_table_shards(). Его значение по умолчанию = 0

Возвращаемое значение: N/A

4.12.3.6 citus_set_default_rebalance_strategy

Обновление таблицы стратегий перебалансировки, изменив стратегию, указанную в ее аргументе, на выбранную по умолчанию при перебалансировке сегментов.

Аргументы:

name: имя стратегии в pg_dist_rebalance_strategy

Возвращаемое значение: N/A

Пример:

```
SELECT citus_set_default_rebalance_strategy('by_disk_size');
```

4.12.3.7 citus_remote_connection_stats

Функция `citus_remote_connection_stats()` показывает количество активных подключений к каждому удаленному узлу.

Аргументы: N/A

Пример:

```
SELECT * from citus_remote_connection_stats();
```

Вывод SQL-команды представлен на рисунке 4.17.

hostname	port	database_name	connection_count_to_node
-----+-----+-----+-----			
citus_worker_1	5432	postgres	3
(1 row)			

Рисунок 4.17 – Вывод результата функции `citus_remote_connection_stats`

4.12.3.8 citus_drain_node

Функция `citus_drain_node()` перемещает сегменты с назначенного узла на другие узлы, для которых `shouldhaveshard` установлено значение `true` в таблице рабочих узлов. Эта функция предназначена для вызова перед удалением узла из кластера, то есть отключением физического сервера узла.

Аргументы:

nodename: имя хоста, имя узла, который будет удален.

nodeport: номер порта узла, который будет удален.

shard_transfer_mode: (необязательно) указание метода репликации – использовать логическую репликацию СУБД «Jatoba» или команду копирования между рабочими узлами. Возможные значения:

auto: требование идентификации реплики, если возможна логическая репликация, в противном случае – устаревшее поведение (например, для восстановления сегментов). Это значение по умолчанию.

force_logical: использование логической репликации, даже если таблица не имеет идентификатора реплики. Любые одновременные инструкции update/delete для таблицы завершатся ошибкой во время репликации.

block_writes: использование копирования (блокирующую запись) для таблиц, в которых отсутствует первичный ключ или идентификатор реплики.

rebalance_strategy: (необязательно) имя стратегии в таблице стратегий перебалансировки. Если этот аргумент опущен, функция выбирает стратегию по умолчанию, как указано в таблице.

Возвращаемое значение: N/A

Пример:

Шаги по удалению одного узла (например, '10.0.0.1' на стандартном порту СУБД Jatoba):

1. Очистить узел:

```
SELECT * from citus_drain_node('10.0.0.1', 5432);
```

При использовании DNS-имени узла:

```
SELECT * from citus_drain_node('worker1', 5432);
```

2. Подождать, пока команда не завершится.
3. Удалить узел.

При очистке нескольких узлов рекомендуется использовать `rebalance_table_shards`. Это позволяет `ja_Hipe_Cluster` планировать заранее и перемещать сегменты минимальное количество раз.

1. Выполнить команду для каждого узла, который требуется удалить:

```
SELECT * FROM citus_set_node_property(node_hostname, node_port, 'shouldhaveshard', false);
```

2. Очистить все узлы сразу с помощью `rebalance_table_shards`:

```
SELECT * FROM rebalance_table_shards(drain_only := true);
```

3. Подождать, пока команда не завершится.

4. Удалить узлы.

4.12.3.9 isolate_tenant_to_new_shard

Эта функция создает новый сегмент для хранения строк с определенным единственным значением в столбце распределения. Это удобно для multi-tenant варианта использования компонента, где крупный tenant может быть размещен отдельно на своем собственном сегменте и на своем собственном физическом узле.

Аргументы:

table_name: имя таблицы для получения нового фрагмента.

tenant_id: значение столбца распределения, которое будет присвоено новому сегменту.

cascade_option: (необязательно) при установке значения «CASCADE» также изолирует сегмент от всех таблиц в таблицах совместного размещения текущей таблицы.

Возвращаемое значение:

shard_id: функция возвращает уникальный идентификатор, присвоенный вновь созданному сегменту.

Пример:

Создание нового сегмента для хранения элементов lineitems для клиента 135:

```
SELECT isolate_tenant_to_new_shard('lineitem', 135);
```

Вывод SQL-команды представлен на рисунке 4.18.

isolate_tenant_to_new_shard
102240

Рисунок 4.18 – Вывод результата функции isolate_tenant_to_new_shard

4.12.3.10 citus_create_restore_point

Временно блокирует запись в кластер и создает именованную точку восстановления на всех узлах. Эта функция похожа на `pg_create_restore_point`, но применяется ко всем узлам и обеспечивает согласованность точки восстановления между ними. Эта функция хорошо подходит для восстановления на определенный момент времени и разветвления кластера.

Аргументы:

name: имя точки восстановления для создания.

Возвращаемое значение:

coordinator_lsn: порядковый номер точки восстановления в WAL узла координатора.

Пример:

```
select citus_create_restore_point('foo');
```

Вывод SQL-команды представлен на рисунке 4.19.

<code>citus_create_restore_point</code>
<code>0/1EA2808</code>

Рисунок 4.19 – Вывод результата функции `citus_create_restore_point`

5. ЧТЕНИЕ С РЕПЛИК. ЧТЕНИЕ С РЕПЛИК С УЧЕТОМ ГЕОЗОНЫ

5.1. Описание функциональности

В компоненте есть параметр «citus.use_secondary_nodes», который принимает значения «always» и «never». При значении «never» запросы никогда не транслируются на реплики, при значении «always» - все запросы переадресуются на реплики, а возможность выполнения транзакций чтения-записи отключена.

Новый режим «preferred», устанавливающий следующее поведение:

- а) все транзакции чтения-записи выполняются штатно, как в режиме «never»;
- б) транзакции только для чтения могут быть переадресованы на реплики, как в режиме «always».

При этом под транзакцией «чтения-записи» и «только для чтения» подразумевается явно определенное соответствующее свойство «READ ONLY» транзакции. Кроме этого, чтобы предотвратить избыточное сетевое взаимодействие, на реплику не будут отправлены те операторы, которые могут быть выполнены локально на узле, вызвавшем транзакцию. И в завершении, чтобы обеспечить разумную балансировку в географически распределённой среде, требуется ограничить возможность переадресации между разными «зонами доступности» - запрос может быть передан лишь на реплику, которая расположена в той же «зоне доступности», что и вызвавший транзакцию узел. Таким образом, оператор будет отправлен для выполнения на реплику в случае выполнения следующих 4 условий:

- Значение citus.use_secondary_nodes="preferred";
- Транзакция определена как READ ONLY;
- Локальное выполнение невозможно. Оператор, который предлагается выполнить на реплике, не может быть выполнен на узле, который вызвал транзакцию;
- Существует реплика узла, которая расположена в той же зоне доступности, что и вызвавший транзакцию узел.

Кроме этого, при чтении с реплик существует проблема аномалии устаревшего чтения - ситуации, когда транзакция только для чтения может прочитать «устаревшее» состояние данных с реплики. Подробности этой проблемы рассмотрены в следующем разделе.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

5.2. Проблемы и решения

Под транзакциями только чтения мы подразумеваем не все транзакции, которые выполняют только операции чтения, а лишь те, для которых этот режим (READ ONLY) был определен явным образом. Транзакция в режиме "READ ONLY" может быть запущена одним из следующих способов, а именно:

- ## Пример транзакций «READ ONLY»

№ п/п	Имя, Фамилия, Отчество	Подпись студ. (инд.)	Дата подписания пом.
-------	------------------------	----------------------	----------------------

```
SET default_transaction_read_only TO true;
BEGIN;
SELECT * FROM table;
COMMIT;

-- Проблемный пример №1, требует отдельного
-- соглашения о "правильном" поведении
SET default_transaction_read_only TO false;
BEGIN;
SET default_transaction_read_only TO true;
SELECT * FROM table;
COMMIT;
```

Пример транзакций «READ WRITE»

```
SET default_transaction_read_only TO false;
BEGIN;
SELECT * FROM table;
COMMIT;

SET default_transaction_read_only TO true;
BEGIN READ WRITE;
SELECT * FROM table;
COMMIT;

-- Проблемный пример №1, требует отдельного
-- соглашения о "правильном" поведении
SET default_transaction_read_only TO true;
BEGIN;
SET default_transaction_read_only TO false;
SELECT * FROM table;
COMMIT;
```

5.2.2. Переадресация

В текущей реализации предполагается, что у каждого узла есть доступная реплика - иными словами условие существования реплики узла не проверяется.

5.2.3. Поведение при отказе реплики

В случае, когда, запрос удовлетворяет всем описанным условиям и происходит его переадресация на реплику. Если эта реплика оказывается фактически недоступной, то в этом случае поведение должно быть полностью эквивалентным ситуации недоступного узла - в зависимости от состояния соединения будет либо ожидание ответа по таймауту

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

statement_timeout (в случае, если соединение берется из пула), либо ошибка установки нового соединения (если открывается новое соединение).

5.2.4. Балансировка с учетом «зон доступности»

Будем считать, что переадресация оператора на географически удаленную реплику необходимо исключить. Для этого в метаданных вводится признак принадлежности каждого узла или реплики к некоторой «зоне доступности». Балансировка читающей нагрузки возможна только в пределах одной зоны доступности - вызвавший транзакцию узел может отправить запрос на чтение только реплику в пределах своей «зоны доступности».

5.2.5. Проблема пула соединений

Диагностирована проблема пула соединений. Компонент имеет очень большие издержки на обслуживание нескольких пулов соединений в случае, если их суммарный размер превышает оптимальное для узла предельное количество соединений (4 x vCPU).

Сейчас решение этой проблемы возможно только формальное - при включении режима чтения с реплик необходимо пропорционально уменьшить размер максимального пула соединений между узлами.

5.2.6. Балансировка на несколько узлов

Допустим, у узла имеется более чем одной реплики. При этом, эти реплики являются равнозначными - находятся в одной «зоне доступности» и имеют одинаковый тип репликации. В этом случае встает вопрос о том, как балансировать нагрузку между этими узлами.

Несмотря на наличие большого числа алгоритмов балансировки (циклический перебор, Least connection и их вариации) в первой реализации рекомендуется использовать простейший метод балансировки - выбор случайного узла из имеющихся (алгоритм «Random»).

Выбор и анализ эффекта от иных алгоритмов балансировки рекомендуется выполнить позже, при дальнейшем развитии указанной функциональности. А с учетом того, что уже диагностирована проблема обслуживания нескольких пулов соединений (по одному пулу к каждой реплике), то вероятно наилучшим алгоритмом балансировки нагрузки будет такой, который минимизирует количество независимых пулов соединений у узла (алгоритм "First").

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

5.2.7. Балансировка при локальном чтении

Допустим, у нас есть кластер, состоящий из двух узлов (W1 и W2) и их реплик (R1 и R2). На узле W1 начинается транзакция, которая требует чтения сегментов данных, расположенных на W1 и W2. В этом случае, необходимо обеспечить следующее поведение:

- Для чтения данных с W2, узел W1 должен обратиться на реплику R2;
- Для чтения данных с W1, узел W1 не должен обращаться к реплике R1, а должен выполнить это чтение локально.

6. ОБНОВЛЕНИЕ И УДАЛЕНИЕ КОМПОНЕНТА

Обновление компонента осуществляется командой:

```
ALTER EXTENSION citus UPDATE;
```

Удаление компонента происходит с помощью команды:

```
DROP EXTENSION citus;
```

7. СООБЩЕНИЯ ОБ ОШИБКАХ

7.1. Ошибка работы ja_hipe_cluster и pg_task на воркере

Предусловия

- Подготовить 2 узла;
- Выполнить установку СУБД на оба с компонентами ja_hipe_cluster, pg_task;
- Добавить координатор;
- Добавить воркер;
- Добавить pg_task в «shared_preload_libraries» конфигурационного файла;
- Выполнить перезагрузку СУБД на обоих узлах.

Сценарий

Подключиться к БД на координаторе и создать задачу на выполнение на координаторе и воркере:

```
INSERT INTO task (input) VALUES ('SELECT now()');
```

Вывести статус выполнения SQL-командой:

```
SELECT * FROM task;
```

В данном случае задача не работает, так как фоновый процесс запускается на локальной машине, то он и исполняется на локальной машине.

Решение проблемы

Написать SQL-запрос:

```
SELECT run_command_on_workers($cmd$ SHOW port; $cmd$);
```

Вставить в таблицу task запрос с указанием конкретного подключения на конкретный хост:

```
INSERT INTO task (input, remote) VALUES ('SELECT now()',  
'user=user host=10.10.104.131');
```


ПРИЛОЖЕНИЕ 1

Пример установки СУБД «Jatoba» из локального репозитория для ОС Ubuntu 20.04

Установка СУБД «Jatoba» из локального репозитория для ОС Ubuntu проводится в следующем порядке:

- 1) В терминале войти в режим суперпользователя, выполнив команду:

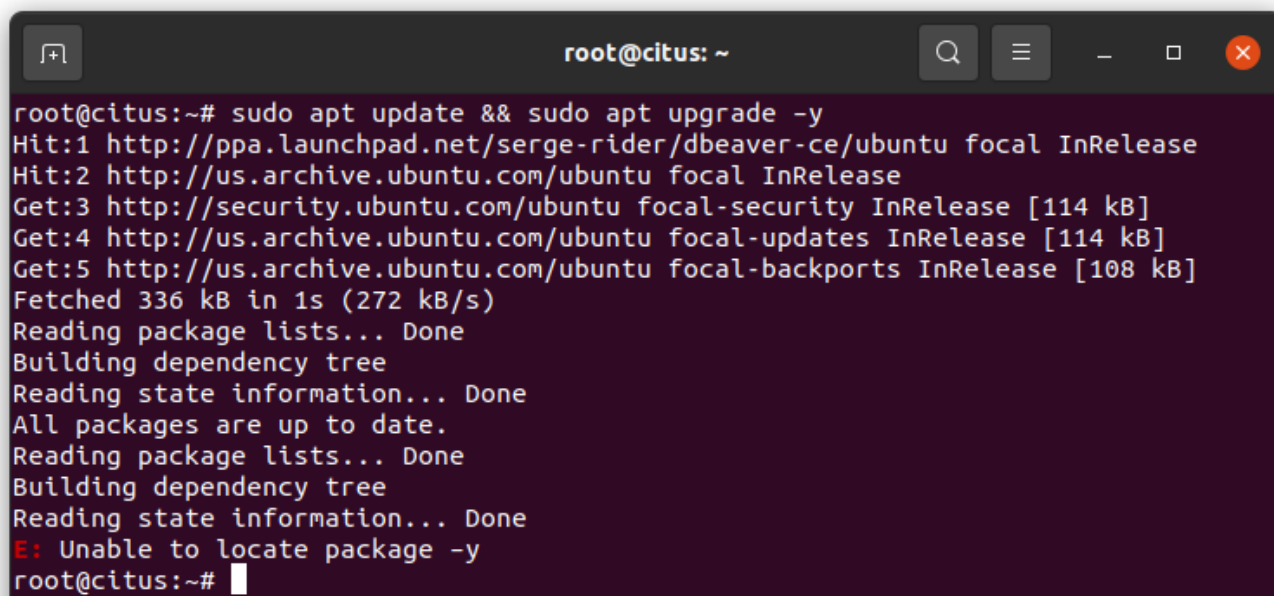
```
sudo su
```

- 2) Если команды sudo не существует – установить:

```
su -l  
apt-get install sudo -y
```

- 3) Выполнить обновление системы:

```
sudo apt update && sudo apt upgrade -y  
sudo apt -s dist-upgrade  
sudo apt dist-upgrade
```



The screenshot shows a terminal window titled 'root@citus: ~'. The user has entered the command 'sudo apt update && sudo apt upgrade -y'. The output shows several package lists being updated from various sources (launchpad.net, archive.ubuntu.com, security.ubuntu.com). It indicates that 336 kB were fetched in 1 second at a rate of 272 kB/s. After reading the package lists and building the dependency tree, it states 'All packages are up to date.' However, there is an error at the end: 'E: Unable to locate package -y'.

Рисунок 1.1 – Обновление системы

- 4) Создать папку localrepo в корневом каталоге:

```
mkdir /localrepo
```

- 5) В созданную папку скопировать:

- каталог <pool>
- каталог <dist>
- файл <DEB-GPG-KEY-Jatoba>

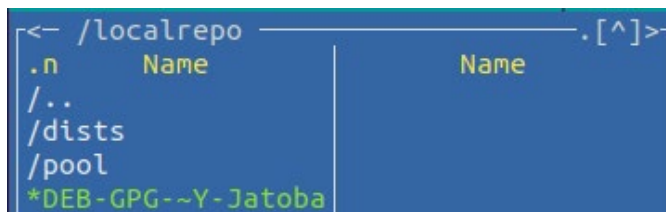


Рисунок 1.2 – Структура каталога «localrepo»

- 6) Установить открытый ключ репозитория:

```
apt-key add /localrepo/DEB-GPG-KEY-Jatoba
```

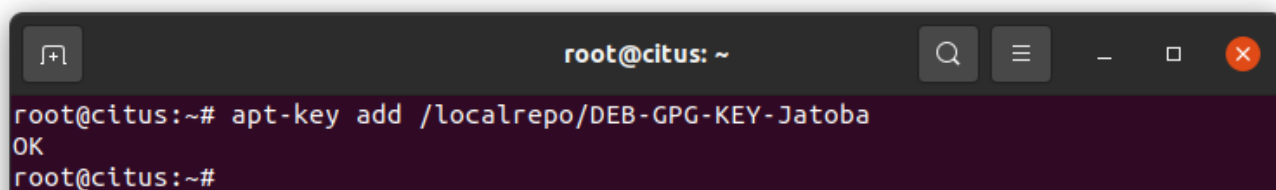


Рисунок 1.3 – Установка открытого ключа репозитория

- 7) Добавить описание локального репозитория в систему:

```
nano /etc/apt/sources.list.d/jatoba-6.list
```

- 8) Добавить в файл следующее содержимое и сохранить:

```
deb file:///localrepo stable non-free
```

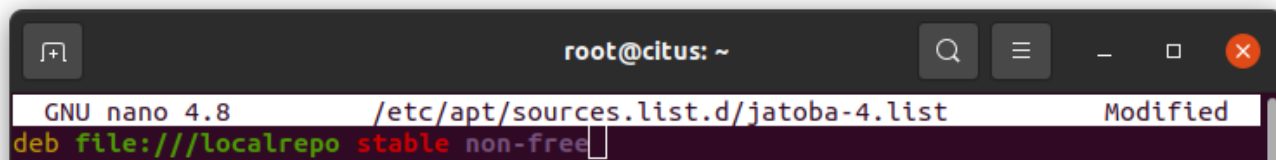
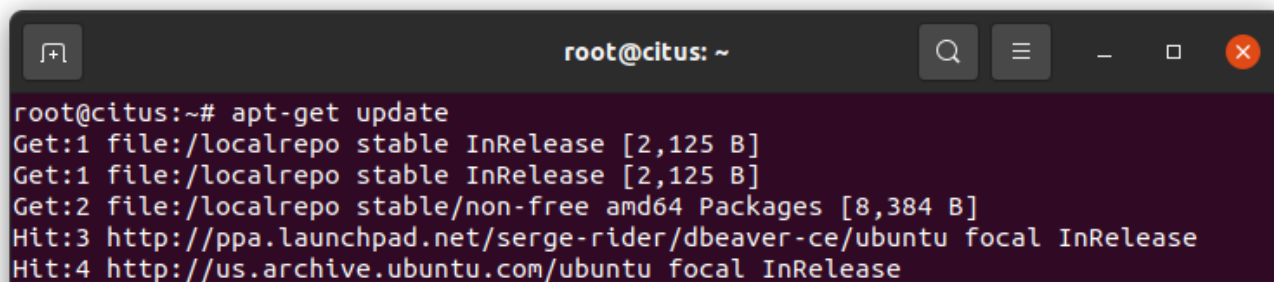


Рисунок 1.4 – Содержание файла «jatoba-6.list»

- 9) Проиндексировать обновленное состояние репозитория:

```
apt-get update
```



```
root@citus: ~
root@citus:~# apt-get update
Get:1 file:/localrepo stable InRelease [2,125 B]
Get:1 file:/localrepo stable InRelease [2,125 B]
Get:2 file:/localrepo stable/non-free amd64 Packages [8,384 B]
Hit:3 http://ppa.launchpad.net/serge-rider/dbeaver-ce/ubuntu focal InRelease
Hit:4 http://us.archive.ubuntu.com/ubuntu focal InRelease
```

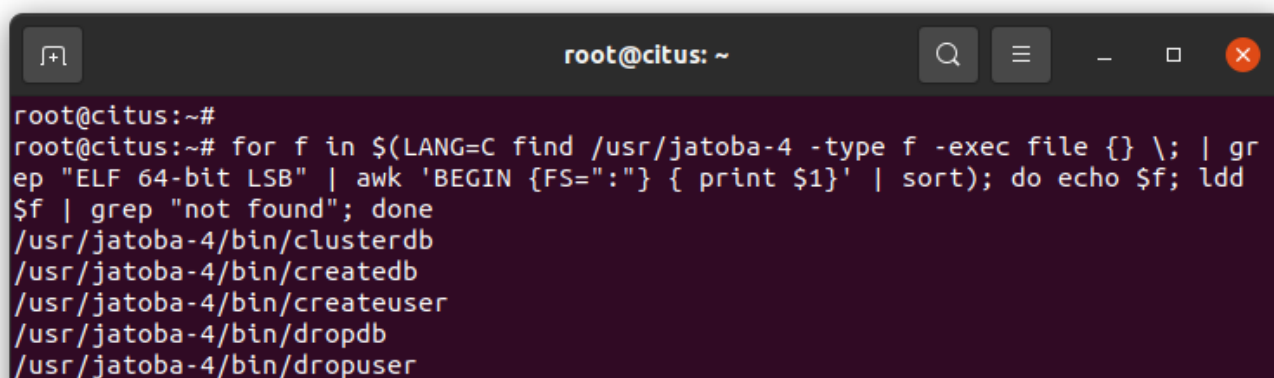
Рисунок 1.5 – Индексация репозитория

- 10) Установить СУБД Jatoba при помощи команды:

```
apt-get install jatoba6-client jatoba6-contrib jatoba6-libs
jatoba6-server jatoba6-ja-hipe-cluster
```

- 11) Убедиться, что отсутствуют ошибки зависимостей:

```
for f in $(LANG=C find /usr/jatoba-6 -type f -exec file {} \; |
grep "ELF 64-bit LSB" | awk 'BEGIN {FS=":"} { print $1}' |
sort); do echo $f; ldd $f | grep "not found"; done
```

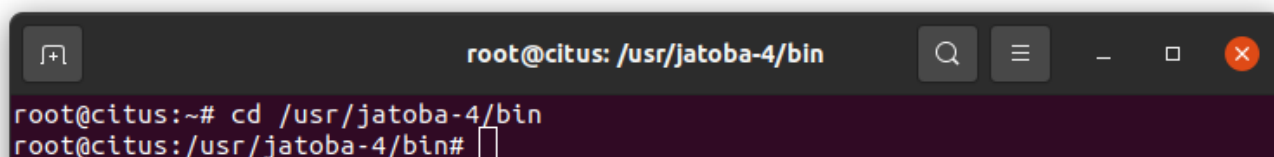


```
root@citus: ~
root@citus:~# for f in $(LANG=C find /usr/jatoba-4 -type f -exec file {} \; | gr
ep "ELF 64-bit LSB" | awk 'BEGIN {FS=":"} { print $1}' | sort); do echo $f; ldd
$f | grep "not found"; done
/usr/jatoba-4/bin/clusterdb
/usr/jatoba-4/bin/createdb
/usr/jatoba-4/bin/createuser
/usr/jatoba-4/bin/dropdb
/usr/jatoba-4/bin/dropuser
```

Рисунок 1.6 – Проверка отсутствия ошибок зависимостей

- 12) Перейти в директорию исполняемых файлов СУБД:

```
cd /usr/jatoba-6/bin
```



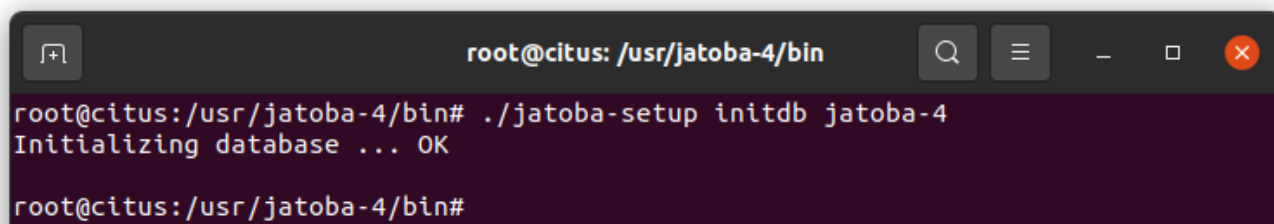
```
root@citus: /usr/jatoba-4/bin
root@citus:~# cd /usr/jatoba-4/bin
root@citus:/usr/jatoba-4/bin#
```

Рисунок 1.7 – Переход в каталог /usr/jatoba-4/bin

- 13) Инициализировать каталог данных СУБД при помощи команды:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
./jatoba-setup initdb jatoba-6
```



A terminal window with a dark background. The title bar shows 'root@citus: /usr/jatoba-4/bin'. The command prompt is 'root@citus:/usr/jatoba-4/bin#'. The user enters './jatoba-setup initdb jatoba-4'. The output is 'Initializing database ... OK'. The prompt returns to 'root@citus:/usr/jatoba-4/bin#'.

Рисунок 1.8 – Инициализация СУБД

- 14) Провести процедуру активации:

```
./jactivator
```

- 15) Открыть конфигурационный файл «postgresql.conf»:

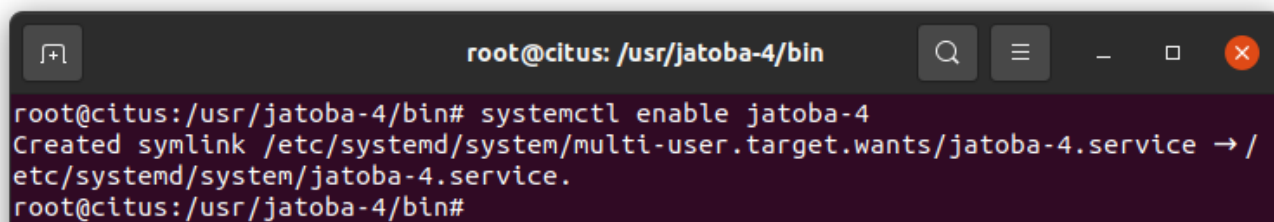
```
nano /var/lib/jatoba/6/data/postgresql.conf
```

- 16) Указать путь до файла лицензии и адрес сервера, затем сохранить изменения:

```
lic_product_name = 'Jatoba'  
lic_file_path = '/usr/jatoba-6/bin/jatoba.cer'  
lic_server_addr = 'https://license.gaz-is.ru'
```

- 17) Добавить сервис в список автозапуска:

```
systemctl enable jatoba-6
```



A terminal window with a dark background. The title bar shows 'root@citus: /usr/jatoba-4/bin'. The command prompt is 'root@citus:/usr/jatoba-4/bin#'. The user enters 'systemctl enable jatoba-4'. The output is 'Created symlink /etc/systemd/system/multi-user.target.wants/jatoba-4.service → /etc/systemd/system/jatoba-4.service.'. The prompt returns to 'root@citus:/usr/jatoba-4/bin#'.

Рисунок 1.9 – Добавление сервиса jatoba-4 а автозагрузку ОС

- 18) Запустить службу:

```
systemctl start jatoba-6
```

```
root@citus: /usr/jatoba-4/bin
root@citus:/usr/jatoba-4/bin# systemctl start jatoba-4
root@citus:/usr/jatoba-4/bin#
```

Рисунок 1.10 – Запуск службы jatoba-4

19) Проверить статус службы:

```
systemctl status jatoba-6
```

```
root@citus: /usr/jatoba-4/bin
● jatoba-4.service - Jatoba 4 database server
   Loaded: loaded (/etc/systemd/system/jatoba-4.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2022-12-21 02:31:12 PST; 12s ago
     Docs: https://www.gaz-is.ru/Jatoba/doc
   Process: 8465 ExecStartPre=/usr/jatoba-4/bin/jatoba-check-db-dir ${PGDATA} (code=exited, status=0/SUCCESS)
   Main PID: 8473 (postmaster)
    Tasks: 9 (limit: 8760)
   Memory: 18.9M
   CGroup: /system.slice/jatoba-4.service
           └─8473 /usr/jatoba-4/bin/postmaster -D /var/lib/jatoba/4/data/
             └─8474 postgres: logger
               └─8476 postgres: check licenser
                 └─8477 postgres: checkpointer
                   └─8478 postgres: background writer
                     └─8479 postgres: walwriter
                       └─8480 postgres: autovacuum launcher
                         └─8481 postgres: stats collector
                           └─8482 postgres: logical replication launcher

Dec 21 02:31:11 citus systemd[1]: Starting Jatoba 4 database server...
Dec 21 02:31:11 citus postmaster[8473]: 2022-12-21 02:31:11.917 PST [8473] LOG: starting PostgreSQL 14.0 (Debian 14.0-1) on x86_64, compiled by gcc (Debian 12.2.0-14) 64-bit
Dec 21 02:31:11 citus postmaster[8473]: 2022-12-21 02:31:11.917 PST [8473] HINT: logging on via Unix socket at "/var/lib/jatoba/4/data/.s.PGSQL.5432"
Dec 21 02:31:12 citus systemd[1]: Started Jatoba 4 database server.
lines 1-23
```

Рисунок 1.11 – Проверка статуса службы jatoba-6

20) Авторизоваться в psql:

```
su -l postgres
psql
```

21) Установить пароль для пользователя СУБД «postgres»:

```
\password
```

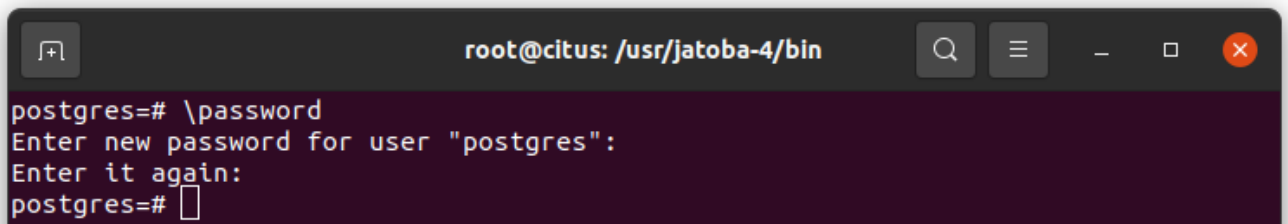


Рисунок 1.12 – Установка пароля для пользователя ОС

22) Установить пароль для системного пользователя ОС «postgres»:

```
sudo passwd postgres
```

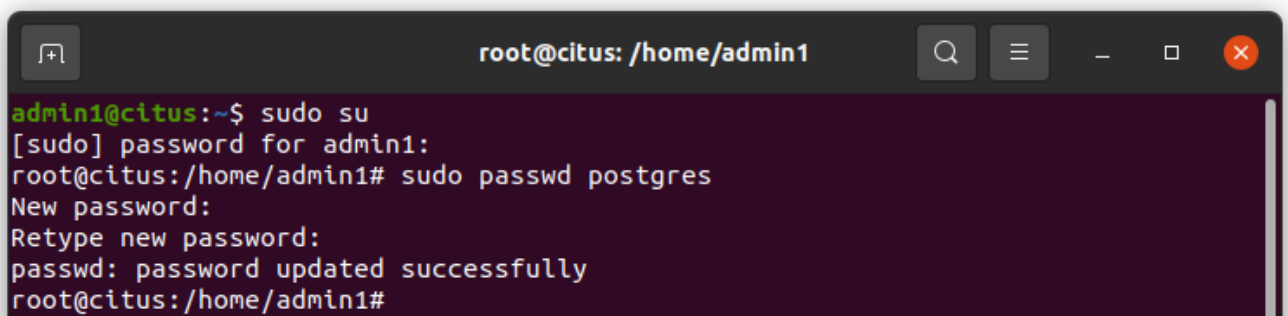


Рисунок 1.13 – Установка пароля для пользователя СУБД

На этом этапе установку СУБД с компонентом ja_Hipe_Cluster можно считать оконченной.

ПРИЛОЖЕНИЕ 2

Восстановление данных при помощи компонента pg_ProBackup

Компонент pg_ProBackup используется для управления резервным копированием и восстановлением баз данных СУБД «Jatoba». При использовании компонента ja_Hipe_Cluster также есть возможность сделать резервную копию кластера и восстановить кластер из нее.

Более подробная информация об использовании компонента pg_ProBackup находится в документе «СУБД «Jatoba». Руководство по настройке. Часть 4. Расширенное резервное копирование. Компонент pg_ProBackup».

Для подготовки кластера к созданию резервных копий необходимо выполнить следующие действия на каждом узле кластера:

1) Установить пакеты pg_ProBackup:

```
apt-get install jatoba6-pg-probackup jatoba4-ptrack
```

2) Создать каталог для хранения резервных копий:

```
mkdir /var/lib/jatoba/backup_pro
```

3) Проинициализировать созданный каталог компонентом pg_ProBackup:

```
/usr/jatoba-6/bin/pg_probackup init -B /var/lib/jatoba/backup_pro
```

4) Определить имя экземпляра СУБД:

```
/usr/jatoba-6/bin/pg_probackup add-instance -B  
/var/lib/jatoba/backup_pro -D /var/lib/jatoba/6/data --instance  
local_db
```

С помощью компонента pg_ProBackup можно создавать как полные, так и инкрементальные резервные копии, и восстанавливать кластер из них.

Полная резервная копия – копия, содержащая все файлы данных кластера, необходимые для его восстановления.

Инкрементальная резервная копия – копия, содержащая только изменения, сделанные после создания предыдущей полной или инкрементальной копии.

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

Пример создания и восстановления из копий.

1) На главном узле: создать таблицу «github_events»:

```
psql
CREATE TABLE github_events (
    event_id bigint,
    event_type text,
    event_public boolean,
    repo_id bigint,
    payload jsonb,
    repo jsonb,
    actor jsonb,
    org jsonb,
    created_at timestamp
);
```

2) На главном узле: сделать таблицу «github_events» распределенной:

```
psql
SELECT create_distributed_table('github_events', 'repo_id');
```

3) На главном узле: скачать тестовые данные для таблицы «github_events»:

```
wget
http://examples.citusdata.com/github\_archive/github\_events-2015-01-01-{0..5}.csv.gz
```

4) На главном узле: распаковать архив с тестовыми данными:

```
gzip -d github_events-2015-01-01-*.gz
```

5) На главном узле: заполнить таблицу «github_events» тестовыми данными:

```
psql
\COPY github_events FROM 'github_events-2015-01-01-0.csv.gz'
WITH (format CSV)
```

6) На главном узле: создать таблицу «test» для записи состояния кластера:

```
psql
CREATE TABLE test (dt timestamp default now(), state
varchar(100));
```


7) На всех узлах кластера: создать полную резервную копию:

```
/usr/jatoba-6/bin/pg_probackup          backup          -B  
/var/lib/jatoba/backup_pro --instance local_db -b FULL
```

8) На главном узле: заполнить таблицу «github_events» новыми данными:

```
psql  
\COPY github_events FROM 'github_events-2015-01-01-1.csv.gz'  
WITH (format CSV)
```

9) На главном узле: добавить в таблицу «test» информацию о количестве записей в таблице «github_events»:

```
psql  
INSERT INTO TEST (state) VALUES ('после полного бэкапа, в  
github_events - <количество> записей');
```

10) На всех узлах кластера: создать инкрементальную резервную копию:

```
/usr/jatoba-6/bin/pg_probackup          backup          -B  
/var/lib/jatoba/backup_pro --instance local_db -b DELTA
```

11) На главном узле: заполнить таблицу «github_events» новыми данными:

```
sql  
\COPY github_events FROM 'github_events-2015-01-01-2.csv.gz'  
WITH (format CSV)
```

12) На главном узле: добавить в таблицу «test» информацию о количестве записей в таблице «github_events»:

```
psql  
INSERT INTO TEST (state) VALUES ('после инкрементального  
бэкапа, в github_events - <количество> записей');
```

13) На главном узле: создать общую точку восстановления для кластера Citus:

```
psql  
SELECT citus_create_restore_point('citus_rp1');
```

14) На всех узлах кластера: остановить СУБД «Jatoba»:

```
systemctl stop jatoba-6
```

15) На всех узлах кластера: удалить каталог с данными:

```
rm -rf /var/lib/jatoba/6/data
```

16) На всех узлах кластера: восстановить СУБД до созданной точки восстановления:

```
/usr/jatoba-6/bin/pg_probackup          restore          -B  
/var/lib/jatoba/backup_pro --instance  local_db  --recovery-  
target-name="citus_rpl"
```

17) На всех узлах кластера: запустить СУБД «Jatoba»:

```
systemctl start jatoba-6
```

18) На главном узле: проверить, что резервная копия соответствует созданной точке восстановления:

```
psql  
SELECT count(*) FROM github_events;  
SELECT * FROM test;
```

ПРИЛОЖЕНИЕ 3

Настройка SSL соединения для ja_Hipe_Cluster

Данная инструкция описывает как настроить TLS соединение и доступ по сертификатам для внутренних соединений, которые узлы кластера устанавливают друг к другу.

В примере используются следующие SSL сущности:

- root.crt – корневой сертификат для всех остальных в системе;
- user1.crt и user1.key – пара пользовательских сертификат/приватный ключ для подключения пользователя к СУБД, CN = имя пользователя, в примере user1;
- server.crt и server.key - пара серверных сертификат/приватный ключ, CN = dns имя сервера;
- postgres.crt и postgres.key - пара серверных сертификат/приватный ключ для пользователя postgres используемого для межузловых соединений ja_Hipe_Cluster, CN = имя пользователя в примере postgresql. Если для межузлового соединения используется другое имя, то, оно должно иметь права SUPERUSER.

Получив вышеописанные сертификаты, выполните следующие шаги:

- Распределить необходимые сертификаты в соответствующие директории.

Сертификаты user1.crt и user1.key должны лежать в каталоге доступном для пользователя user1, в примере это домашний подкаталог /home/user1/.postgresql/.

Так как уровень проверки с sslmode=verify-full - то в каталоге /home/user1/.postgresql/ должен так же находиться и головной сертификат root.crt

Но возможны и более мягкие уровни защиты.

- Создать на всех серверах СУБД, куда этот пользователь должен иметь доступ, пользователя и назначить ему права:

```
# CREATE ROLE user1 LOGIN;  
# GRANT ALL PRIVILEGIS ON test TO user1;
```

- Создать директорию для отзывных сертификатов и назначить пользователя postgres на всех машинах кластера:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
# mkdir /usr/jatoba-6/etc/jatoba-6/crl
# chown postgres:postgres /usr/jatoba-6/etc/jatoba-6/crl
```

– Расположить сертификаты и ключи по соответствующим путям для всех машин кластера.

В представленном примере они находятся в `/usr/jatoba-6/etc/jatoba-6/`. Однако директория может быть изменена.

- Назначить права на все ключи 0400;
- Назначить права на все сертификаты 0600;
- Изменить конфигурационный файл `postgresql.conf` на всех машинах:

```
ssl = on
ssl_cert_file = '/usr/jatoba-6/etc/jatoba-6/server.crt'
ssl_key_file = '/usr/jatoba-6/etc/jatoba-6/server.key'
ssl_ca_file = '/usr/jatoba-6/etc/jatoba-6/root.crt'
ssl_crl_dir = '/usr/jatoba-6/etc/jatoba-6/crl'
citus.node_conninfo = 'sslcert=/usr/jatoba-6/etc/jatoba-6/postgres.crt sslkey=/usr/jatoba-6/etc/jatoba-6/postgres.key
sslrootcert=/usr/jatoba-6/etc/jatoba-6/root.crt sslmode=verify-full'
```

– После того, как в директорию `/usr/jatoba-6/etc/jatoba-6/crl` обновили список отозванных сертификатов, необходимо выполнить команду в терминале ОС:

```
openssl rehash
```

- Изменить конфигурационный файл `pg_hba.conf`:

```
hostssl all all <IP4 подсети>/24 cert clientcert=verify-full
hostssl all all <IP4 подсети>/24 cert clientcert=verify-full
```

Во втором столбце, целесообразно указать имя БД, куда должен иметь доступ пользователь, `all` - указывается для всех БД.

- Перегрузить сервер SQL-командой:

```
SELECT pg_reload_conf();
```

- Проверить, что есть SSL-соединение:

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------

```
psql -U user1 "host=db.datagile.ru sslcert=user1.crt  
sslkey=user1.key" -d test
```

– После настройки SSL на всех рабочих узлах, настраиваем ja_Hipe_Cluster, проверить соединение со всеми рабочими узлами:

```
# psql -U user1 -host=db2 "host=db.datagile.ru sslcert=user1.crt  
sslkey=user1.key" -d test  
  
# psql -U user1 -host=db3 "host=db.datagile.ru sslcert=user1.crt  
sslkey=user1.key" -d test
```

– На координаторе выполнить SQL-команду:

```
# ALTER SYSTEM SET citus.node_conninfo TO 'sslcert=/usr/jatoba-  
6/etc/jatoba-6/postgres.crt sslkey=/usr/jatoba-6/etc/jatoba-  
6/postgres.key sslmode=verify-full';
```

– Перегрузить конфигурацию сервера SQL-командой:

```
# SELECT pg_reload_conf();
```

– Создать рабочие узлы, указываем внешний IP или DNS на внешний IP:

```
# SELECT citus_set_coordinator_host('coordinator.dns',5432);  
# SELECT citus_add_node('node1.dns',5432);
```

– Настроить соединения между узлами ja_Hipe_Cluster для каждого узла, производим только на мастерах SQL-командой:

```
#INSERT INTO pg_dist_authinfo VALUES(1,'test',  
'sslcert=/usr/jatoba-6/etc/jatoba-6/user1.crt sslkey=  
/usr/jatoba-6/etc/jatoba-6/user1.key');  
  
#INSERT INTO pg_dist_authinfo VALUES(2,'test',  
'sslcert= /usr/jatoba-6/etc/jatoba-6/user1.crt  
sslkey= /usr/jatoba-6/etc/jatoba-6/user1.key');
```

– Проверить работоспособность кластера выполнив любую распределенную операцию SQL-командой:

```
# CREATE TABLE test (id int);  
# SELECT create_distributed_table('test', 'id', shard_count=>4);
```

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Рабочий узел (узел) — узел, который позволяет серверам базы данных координировать взаимодействие друг с другом.

Координационный узел (координатор) — исходный узел, который передает запросы приложения соответствующим рабочим узлам и получает результаты.

Распределенные таблицы — таблицы, которые горизонтально разделены между рабочими узлами.

Сегмент — рабочие таблицы компонентов.

Столбец распределения — столбец распределенной таблицы, который `ja_Hipe_Cluster` использует для создания сегментов.

Ссылочная таблица — тип распределенной таблицы, все содержимое которой сконцентрировано в одном сегменте.

Справочные таблицы — используются для хранения данных

Колоночное хранилище — служит для сжатия данных, ускорения сканирования и поддержки быстрого отображения в обычных и распределенных таблицах.

GUC (Grand Unified Configuration) — это система конфигурации PostgreSQL, которая управляет всеми параметрами настройки поведения базы данных. Она обеспечивает единый механизм управления конфигурацией.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

2PC	–	Two-phase commit
API	–	Application Programming Interface
DDL	–	Data Definition Language
DEB	–	сокращение от Debian
DNS	–	Domain Name System
GPIG	–	Global process identifier
GUC	–	Grand Unified Configuration
IP	–	Internet Protocol
N/A	–	Not available
PID	–	Process Identifier
SQL	–	Structured Query Language
TOAST	–	The Oversized-Attribute Storage Technique
UDF	–	User-Defined Function
WAL	–	Write-Ahead Log
БД	–	База данных
ОС	–	Операционная система
СУБД	–	Система управления базами данных

Лист регистрации изменений

№ изменения: _____	Подпись отв. лица: _____	Дата внесения изм: _____
--------------------	--------------------------	--------------------------