



cedrusdata

Техническая архитектура CedrusData

ООО «Кверифай Лабс»

ОГРН 1217800163790

ИНН 7811766769

КПП 781101001

Введение	3
1. Узлы	3
2. Подключение к CedrusData	5
3. Подключение к источникам данных	5
4. Выполнение SQL-запросов	7
5. Технологический стек	10

Введение

CedrusData это высокопроизводительная распределенная платформа для сквозного анализа всех данных предприятия в облаке и on-premise через единую точку доступа с SQL интерфейсом.

CedrusData основана на распределенном SQL-движке Trino (<https://trino.io/>) и включает дополнительный функционал управления и мониторинга (в том числе в облачных инфраструктурах), улучшения производительности, профессиональную документацию и поддержку.

Данный документ рассматривает ключевые аспекты архитектуры CedrusData.

1. Узлы

CedrusData является распределенной системой, в которой несколько процессов работают над решением общих задач. Каждый отдельный процесс называется узлом. Все узлы CedrusData обеспечивают коммуникацию друг с другом посредством HTTP/HTTPS протокола и сообщений в текстовом формате JSON и бинарном формате Thrift.

В CedrusData существует три типа узлов - worker, coordinator и resource manager.

1.1. Coordinator

Узел **coordinator** отвечает за получение SQL-запросов от пользователей, создание плана выполнения SQL-запроса, запуск и координацию выполнения SQL-запроса на worker узлах, выдачу результатов выполнения SQL-запроса клиенту.

Для создания плана выполнения запроса coordinator получает из источников данных метаинформацию, такую как список таблиц и их атрибутов, а так же информацию о распределении данных в источнике.

Для распределения задач по worker узлам, coordinator собирает и обрабатывает информацию о доступных на worker узлах ресурсах, такую как количество процессорных ядер и количество доступной оперативной памяти.

Для выполнения SQL-запросов кластер должен содержать как минимум один узел coordinator.

1.2. Worker

Узел **worker** отвечает за выполнение отдельных частей SQL-запросов. В процессе старта worker оповещает coordinator о своей работоспособности, что позволяет coordinator узлу сформировать список доступных worker узлов.

В процессе выполнения SQL-запроса, worker может получить от coordinator задачу на выполнение отдельной части SQL-запроса. В процессе выполнения задачи worker может:

- Подключаться к источникам для получения необходимых для выполнения запроса данных.
- Пересылать результаты промежуточных вычислений другим worker узлам.
- Пересылать результаты выполнения части запроса на узел coordinator.

Для выполнения SQL-запросов кластер должен содержать как минимум один узел worker. Узел coordinator может опционально выполнять роль worker узла.

1.3. Resource Manager

При одновременном выполнении большого числа SQL-запросов узел coordinator может стать узким местом системы. CedrusData допускает наличие нескольких coordinator узлов в кластере, что позволяет распределить нагрузку между ними.

При использовании нескольких coordinator узлов CedrusData требует выделение сервиса по планированию и учету ресурсов в отдельный узел, называемый **resource manager**. Данный узел агрегирует информацию о ресурсах, доступных на worker узлах для выполнения SQL-запросов, таких как количество процессорных ядер и количество оперативной памяти. Coordinator периодически запрашивает данную информацию у resource manager и использует ее для планирования SQL-запросов.

1.4. Топология кластера

Для работы кластера необходим как минимум один coordinator и как минимум один узел, который может выполнять запросы (worker или coordinator). Возможности разных типов узлов приведены в таблице ниже.

Таблица 1: Типы узлов CedrusData.

Тип узла	Выполняет запросы	Управляет запросами	Управляет ресурсами
Coordinator	Да, если установлен специальный флаг	Да	Да, если в топологии нет узлов типа resource manager
Worker	Да	Нет	Нет
Resource manager	Нет	Нет	Да

В CedrusData допустимы несколько топологий кластера:

- Топология 1: Один узел, который одновременно выполняет роль worker и coordinator. Данная топология подходит для тестов и изучения возможностей продукта.
- Топология 2: Несколько узлов, среди которых есть один или несколько worker, и один coordinator. Данная топология применяется при промышленном использовании, когда нагрузка на coordinator невелика.
- Топология 3: Несколько узлов, среди которых есть один или несколько worker, несколько coordinator и как минимум один resource manager. Данная топология применяется при промышленном использовании, когда ресурсов одного coordinator узла недостаточно для эффективного управления всеми поступающими запросами.

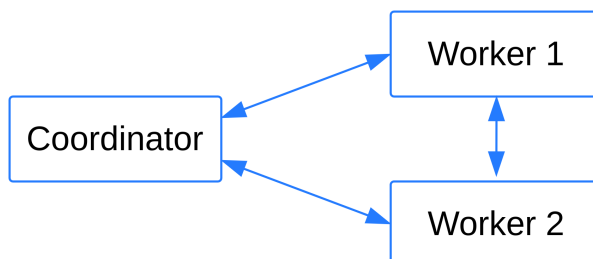


Рис. 1: Пример топологии с одним coordinator и двумя worker узлами.

2. Подключение к CedrusData

Для отправки SQL-запроса, пользователь должен подключиться к кластеру CedrusData одним из двух способов:

- Через JDBC драйвер.
- Через интерфейс командной строки.

В обоих случаях необходимо указать хост и порт coordinator узла. Коммуникация с coordinator осуществляется через протокол HTTP/HTTPS посредством отправки и получения JSON сообщений (REST).

После получения от клиента команды на запуск SQL-запроса, узел coordinator инициирует распределенное выполнение SQL-запроса. По мере появления результатов, coordinator отправляет их клиенту в ответных сообщениях. В случае возникновения ошибки при выполнении запроса, coordinator уведомляет об этом клиента.

3. Подключение к источникам данных

CedrusData позволяет пользователям выполнять SQL-запросы к одному или нескольким источникам данных. CedrusData поддерживает следующие популярные источники данных:

- Озера данных (data lakes) под управлением Hive Metastore и Apache Iceberg.
- Хранилища данных: Greenplum, ClickHouse, Apache Druid, Apache Pinot.
- Реляционные СУБД: Postgres, MySQL, Oracle, SQL Server, MariaDB.
- Нереляционные источники: Cassandra, MongoDB, Redis, Kafka.

Для подключения к источникам данных CedrusData использует подсистему плагинов и коннекторов.

3.1. Плагин

Плагин это дополнительный функционал, который может быть динамически подключен к узлу CedrusData. Плагин представляет собой набор скомпилированных Java-классов и иных ресурсов, необходимых для корректной работы целевого функционала. Среди скомпилированных Java-классов должен присутствовать класс, который реализует интерфейс `io.trino.spi.Plugin`.

Для подключения плагина к CedrusData, необходимо расположить его файлы в отдельной поддиректории внутри специальной директории `plugin/`. При старте узла происходит сканирование содержимого директории `plugin/`. Для каждой поддиректории, в которой найден Java-класс, реализующий `io.trino.spi.Plugin`, происходит создание экземпляра данного класса, через который происходит дальнейшая работа с плагином. Добавление и удаление плагинов возможно только при перезапуске узла.

3.2. Коннектор

Опциональной составной частью плагина является коннектор, который представляет собой Java-класс, реализующий интерфейс `io.trino.spi.connector.Connector`. Коннектор задает логику работы с конкретным типом источников данных, включая:

- Логика подключения к источнику.
- Логика получения метаданных источника.
- Логика чтения данных из источника.
- Логика записи данных в источник.

3.3. Каталог

Коннектор может быть использован для того, что бы подключиться к конкретному экземпляру источника данных. Такое подключение называется каталогом. CedrusData формирует список доступных каталогов в момент запуска узла на основе конфигурации, расположенной в директории `etc/catalogs`. Конфигурация каталога представляет собой файл с именем `<имя_каталога>.properties`. Данный файл содержит в себе название коннектора и специфичную для коннектора конфигурацию. Добавление и удаление каталогов возможно только при перезапуске узла.

Например, для работы с озерами данных в поставку CedrusData входит плагин `hive`, файлы которого расположены в директории `plugin/hive`. Точкой входа является Java-класс `io.trino.plugin.hive.HivePlugin`. Плагин предоставляет коннектор `io.trino.plugin.hive.HiveConnector`. Для подключения к конкретному озеру данных пользователь создает конфигурацию каталога, в которой указывает:

- Название коннектора: `hive`.
- Параметры для подключения к Hive Metastore. Например URL сервера Hive Metastore.

3.4 Схема

Каталог представляет собой конкретный источник данных, который может содержать в себе несколько логических групп объектов, называемых схемами. Схема может содержать следующие объекты: таблицы (`table`), представления (`view`), материализованные представления (`materialized view`), функции и процедуры.

Физическая реализация и содержимое схемы зависит от источника данных. Например, схемами каталога типа `postgres` являются соответствующие схемы конкретного экземпляра Postgres. Схемами каталога типа `tpch` являются сгруппированные по `scale factor` виртуальные таблицы, генерирующие тестовые TPC-H данные.

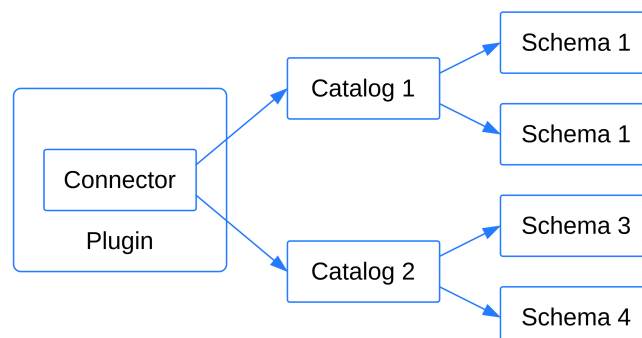


Рис. 2: Иерархия объектов *plugin*, *connector*, *catalog* и *schema*.

Для использования объекта схемы, необходимо обратиться к нему по имени `<имя_каталога>.<имя_схемы>.<имя_объекта>`. Например, если в CedrusData создан каталог с именем `my_pg`, который подключен к экземпляру Postgres, в котором существует схема `hr` с таблицей `employee`, то обратиться к данной таблице можно по имени `my_pg.hr.employee`.

4. Выполнение SQL-запросов

Выполнение SQL-запросов в CedrusData состоит из двух фаз: - Создание плана запроса на coordinator узле. - Выполнение плана на worker узлах.

4.1. Планирование

При получении SQL-запроса coordinator осуществляет его планирование, состоящее из следующих шагов:

- Синтаксический анализ запроса (парсинг). Результатом синтаксического анализа является синтаксическое дерево.
- Семантический анализ синтаксического дерева. На данном этапе происходит проверка семантической корректности запроса, а так же определение объектов, присутствующих в синтаксическом дереве (каталоги, схемы, таблицы, колонки, функции, и т.д.) на основе имеющихся каталогов и плагинов.
- Реляционная трансляция. Семантически корректное синтаксическое дерево преобразуется в дерево реляционных операторов, которое используется для последующей оптимизации плана выполнения запроса.
- Оптимизация. Дерево реляционных операторов преобразуется в более оптимальный формат. Ключевые оптимизации включают в себя: (1) перестановку операторов (например, `filter pushdown`); (2) определение условно-оптимального порядка выполнения операций `join` (`join order planning`); (3) определение условно-оптимальных этапов передачи данных между worker узлами (`exchange planning`), (4) перенос части вычислений в источники (`connector pushdown`).

Результатом планирования является дерево реляционных операторов, представляющее собой условно-оптимальный план выполнения запроса.

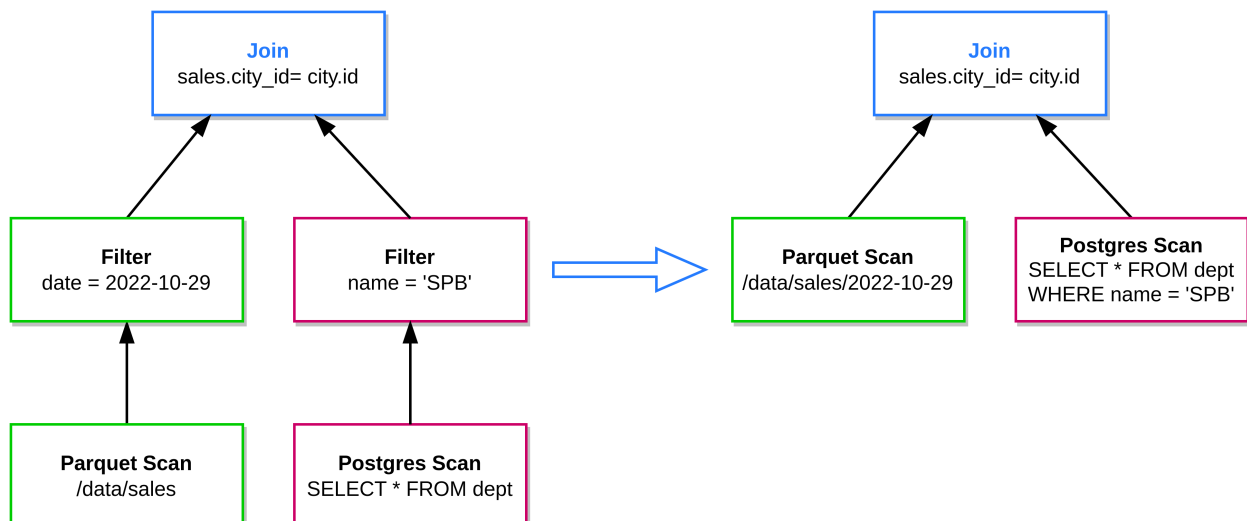


Рис. 3: Пример оптимизации *connector pushdown*, в которой один из фильтров использован для уменьшения количества читаемых *Parquet* файлов из озера данных, а второй фильтр использован для создания более специализированного запроса к СУБД *Postgres*.

Особым типом реляционных операторов является оператор **Exchange**, который описывает обмен данными между независимыми частями распределенного SQL-запроса. Перераспределение может происходить как в рамках одного worker узла, так и между разными worker узлами.

4.2. Stage

Оптимизированное реляционное дерево разбивается по границам операторов **Exchange** на независимые части, называемые **stages**. **Stage** это поддерево операторов, которое может быть выполнено на одном или нескольких worker узлах независимо.

Каждый stage имеет как минимум один вход и строго один выход. Входами stage могут быть объекты каталогов (**Scan**), или результаты выполнения других stage. Выходом stage является поток данных, который либо передается в другой stage, либо поступает на *coordinator*, где проходит опциональную постобработку (например, расчет агрегатов), после чего отправляется пользователю.

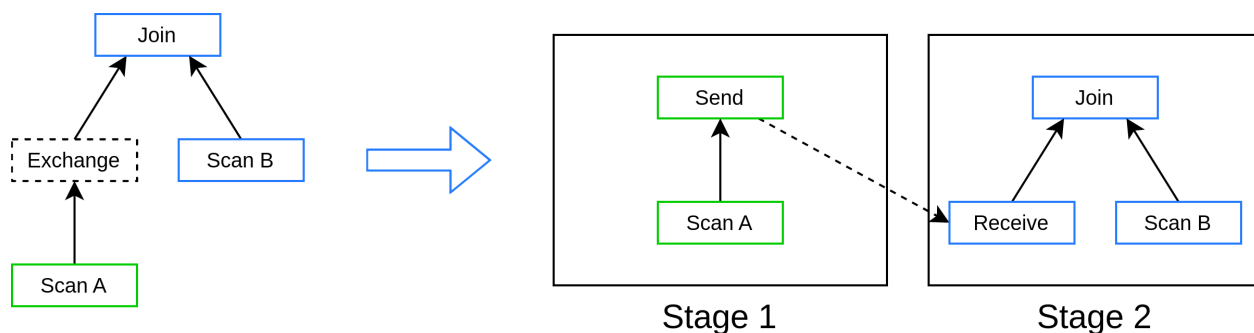


Рис. 4: Разбиение плана запроса с одним **Exchange** оператором на два stage.

4.3. Split

Coordinator определяет для каждого stage набор **Scan** операторов, из которых данных stage будет получать данные. Для каждого **Scan** *coordinator* обращается к соответствующему коннектору и получает один или несколько объектов **split**. **Split** это описание части данных, которая могут быть обработана независимо.

Например, таблица в *Hive Metastore* может быть представлена набором файлов в распределенной файловой системе. Каждый файл может быть прочитан и обработан независимо от остальных.

4.4. Task

По мере получения информации о **split**-ах для текущей stage, *coordinator* начинает распределять обработку отдельных **split** между *worker* узлами. Для каждого *worker* узла, участвующего в выполнении запроса, происходит создание объекта **Task**, который содержит:

- Описание stage (т.е. набор операторов, которые необходимо выполнить).

- Уникальный идентификатор целевого worker узла.
- Список split-ов, которые необходимо обработать в рамках данного task. Список может обновляться в режиме реального времени по мере получения coordinator узлом информации о новых объектах split для целевого источника.

Coordinator отправляет Task на целевой worker. Worker начинает выполнение дерева операторов. Если stage содержит операторы **Scan**, то worker использует соответствующий коннектор, что бы получить данные, описанные в объектах split из источника. Для обработки task, worker может использовать одно или несколько вычислительных ядер процессора.

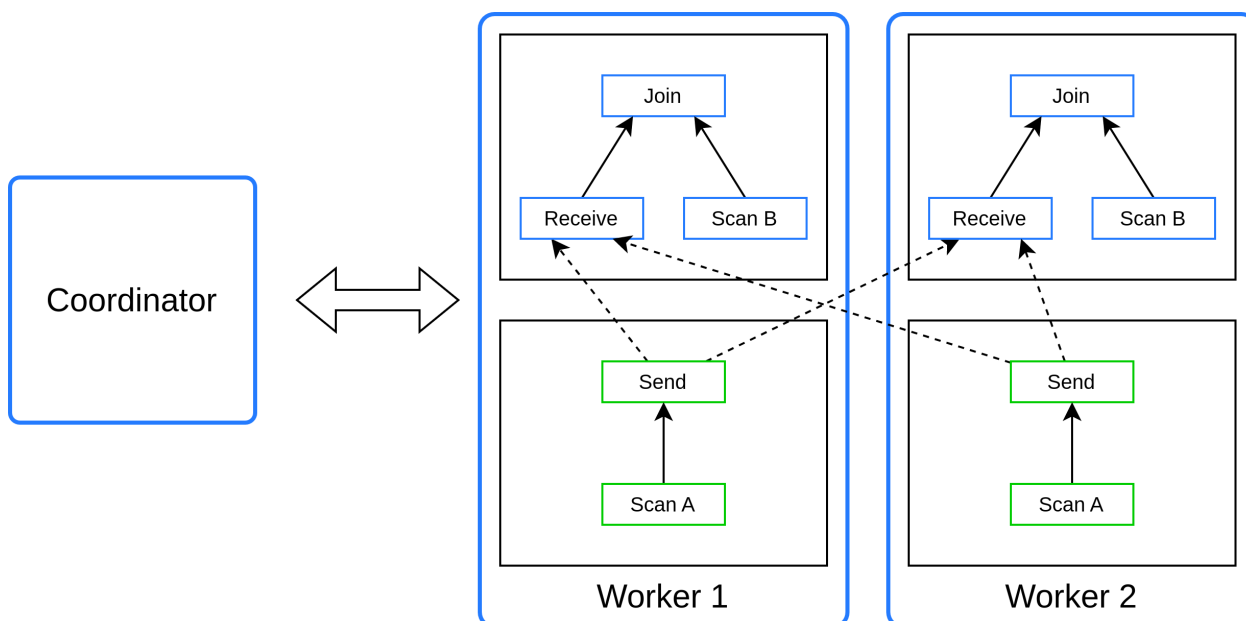


Рис. 5: Параллельное выполнение нескольких stage на двух worker узлах.

4.5. Завершение выполнения

Завершение выполнения SQL-запроса происходит после обработки всех задействованных объектов split, и передачи всех необходимых данных между объектами stage.

- Coordinator получает информацию о том, что все объекты split созданы и отправлены на обработку, и уведомляет об этом задействованные объекты task.
- Когда task на конкретном worker узле завершает обработку последнего split, он уведомляет об этом объект stage.
- Когда все объекты task для данного stage завершены, происходит завершение самого stage и уведомление зависимых stage.
- Когда все stage завершены, coordinator уведомляет клиент о завершении выполнения запроса.

Если в процессе исполнения возникает ошибка, то задействованные объекты получают уведомления в таком же порядке, как и при нормальном завершении. После этого клиент получает сообщение об ошибке.

5. Технологический стек

Платформа CedrusData основана на open-source продукте Trino <https://trino.io/>, который распространяется по лицензии Apache License 2.0.

Исходный код CedrusData и Trino написан на языках программирования Java, Python и C++.

Предварительными требованиями для запуска CedrusData являются:

- При запуске из архива - наличие на компьютере JDK 17 и Python версии 2.x или 3.x.
- При запуске в Docker-контейнере - наличие на компьютере Docker.