**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc198740723)

[1 Исследование предметной области 5](#_Toc198740724)

[1.1 Исследование особенностей систем агрегирования данных 5](#_Toc198740725)

[1.1.1 Определение и назначение систем агрегирования данных 5](#_Toc198740726)

[1.1.2 Классификация систем агрегирования данных 6](#_Toc198740727)

[1.1.3 Архитектурные принципы систем агрегирования данных 7](#_Toc198740728)

[1.2 Исследование методов выполнения аналитических запросов 10](#_Toc198740729)

[1.2.1 Введение в аналитические запросы 10](#_Toc198740730)

[1.2.2 Основные методы выполнения аналитических запросов 10](#_Toc198740731)

[1.3 Исследование особенностей работы оперативной аналитической обработки 12](#_Toc198740732)

[1.3.1 Определение оперативной аналитической обработки (OLAP) 12](#_Toc198740733)

[1.3.2 Основные принципы OLAP 12](#_Toc198740734)

[1.3.3 Технологии оперативной аналитической обработки 13](#_Toc198740735)

[1.3.4 Преимущества и недостатки OLAP 13](#_Toc198740736)

[1.3.5 Системы, использующие OLAP 13](#_Toc198740737)

[1.3.6 Заключение 16](#_Toc198740738)

[2 Исследование подхода к OLAP технологии 17](#_Toc198740739)

[2.1 Анализ и сравнение существующих подходов 17](#_Toc198740740)

[2.1.1 ROLAP (Relational OLAP) 18](#_Toc198740741)

[2.1.2 MOLAP (Multidimensional OLAP) 18](#_Toc198740742)

[2.1.3 HOLAP (Hybrid OLAP) 19](#_Toc198740743)

[2.2 Разработка алгоритма для системы агрегирования данных 20](#_Toc198740744)

[2.2.1 Как формируются индексы 29](#_Toc198740745)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 39](#_Toc198740746)

**ВВЕДЕНИЕ**

В современном мире финансовая аналитика требует обработки больших объемов данных, поступающих из различных источников. Для эффективного анализа и принятия решений необходимо агрегировать данные, сводя их к структурированным наборам, удобным для последующей обработки. В условиях растущей цифровизации, ускорения бизнес-процессов и усложнения требований к отчетности, особенно в финансовом и корпоративном секторе, возникает необходимость в создании высокопроизводительных, масштабируемых и адаптивных систем агрегации и аналитики данных. Одним из наиболее эффективных подходов к решению этой задачи являются OLAP-технологии, обеспечивающие многомерный анализ данных в режиме реального времени. Цель данной магистерской работы — проектирование и реализация системы агрегирования данных с использованием OLAP-подхода на базе колоночной базы данных и стека современных технологий. Разрабатываемая система ориентирована на задачи оперативного расчета финансовых показателей, таких как премии, KPI и агрегированные метрики, с возможностью анализа в различных разрезах (временных, региональных, организационных). В ходе выполнения работы проведено исследование архитектурных принципов систем агрегации, выполнено сравнение подходов OLAP, реализован алгоритм построения иерархической многомерной модели данных, а также рассмотрена методика оптимизации агрегации и индексирования. Отдельное внимание уделено построению OLAP-куба, его описанию, формированию структуры узлов и рёбер, преобразованию данных и интеграции в аналитическую платформу.

1. **Исследование предметной области**

В современных финансовых организациях и крупных компаниях система мотивации сотрудников часто включает премирование на основе KPI, объемов продаж, выполнения планов и других показателей. Ручной расчет таких премий трудоемок, подвержен ошибкам и не позволяет оперативно анализировать данные в различных разрезах (по отделам, регионам, временным периодам).

На протяжении последних лет различные исследователи и компании предлагали решения в области агрегирования данных и аналитической обработки. Однако существующие подходы либо ориентированы на обработку данных в пакетном режиме, что замедляет расчет различных данных, том числе расчета премий, в реальном времени, либо требуют значительных вычислительных ресурсов. Исследование направлено на разработку системы, способной эффективно обрабатывать финансовые данные в режиме оперативной аналитики, сочетая высокую скорость обработки, точность расчетов и адаптивность к изменяющимся условиям.

* 1. **Исследование особенностей систем агрегирования данных**
     1. **Определение и назначение систем агрегирования данных**

Системы агрегирования данных представляют собой программные и аппаратные комплексы, предназначенные для сбора, обработки, обобщения и хранения информации из разнородных источников. Эти системы применяются для консолидации данных и их подготовки к аналитической обработке.

Основные задачи агрегирования данных:

* Объединение данных из различных источников (базы данных, API, файловые хранилища);
* Очистка и трансформация данных;
* Поддержка процессов ETL (Extract, Transform, Load);
* Обеспечение оперативного доступа к агрегированным данным;
* Подготовка данных для аналитических и BI-систем.
  + 1. **Классификация систем агрегирования данных**

По типу источников данных.

Реляционные базы данных (SQL-based) – традиционные базы данных, такие как PostgreSQL, MySQL, Oracle, которые хорошо подходят для структурированных данных и обеспечивают мощные средства запросов и агрегирования.

NoSQL-хранилища – базы данных вроде MongoDB, Cassandra, предназначенные для обработки неструктурированных и полуструктурированных данных, часто применяются для потоковой аналитики и масштабируемых решений.

Потоковые данные (Stream Data) – данные, поступающие в реальном времени из сенсоров, логов, API-интерфейсов (Kafka, Apache Flink). Используются для построения предсказательных моделей и аналитики в реальном времени.

По способу обработки данных.

Пакетная обработка (Batch Processing) – применяется, когда анализ проводится на больших объемах данных, но в режиме периодического обновления. Подходит для исторического анализа, но не всегда удовлетворяет требованиям оперативной аналитики.

Потоковая обработка (Stream Processing) – позволяет анализировать данные в реальном времени. Применяется для финансовой аналитики, где требуется оперативное реагирование.

Гибридные подходы – сочетают пакетную и потоковую обработку, позволяя обрабатывать данные как в реальном времени, так и ретроспективно. Этот вариант предпочтителен в задачах финансовой аналитики, так как он обеспечивает баланс между производительностью и точностью.

По архитектуре.

Централизованные системы – все данные собираются и обрабатываются в одном хранилище. Такой подход удобен для небольших объемов данных, но плохо масштабируется.

Распределенные системы – данные обрабатываются на множестве узлов, что повышает отказоустойчивость и масштабируемость. Такие системы, как Hadoop, Spark и ClickHouse, позволяют работать с большими объемами данных и обеспечивают высокую производительность.

Выбор конкретного класса системы агрегирования данных зависит от требований к обработке данных, скорости аналитики и потребностей бизнеса. В рамках исследования предпочтение отдается распределенной системе с гибридной моделью обработки, так как она позволяет анализировать финансовые данные как в реальном времени, так и с учетом исторических трендов.

* + 1. **Архитектурные принципы систем агрегирования данных**

Архитектура системы агрегирования данных определяет её способность эффективно справляться с большими объемами информации, обеспечивать высокую производительность, отказоустойчивость и масштабируемость. Рассмотрим основные архитектурные принципы и подходы.

Многоуровневая архитектура.

Большинство систем агрегирования строятся по многоуровневой архитектуре, включающей следующие уровни:

* **Уровень сбора данных** (ETL/ELT): отвечает за извлечение данных из различных источников, их преобразование и загрузку в хранилище.
* **Хранилище данных**: может быть реализовано в виде реляционной СУБД, колоночного хранилища или многомерного куба.
* **Уровень агрегации**: здесь выполняются операции группировки, фильтрации, вычисления метрик.
* **Уровень визуализации (BI)**: предоставляет интерфейс для аналитиков и пользователей для построения отчетов и дашбордов.

Подходы к масштабированию.

Современные системы должны обеспечивать горизонтальное масштабирование. Например, ClickHouse реализует шардирование и репликацию на уровне кластера, обеспечивая балансировку нагрузки и отказоустойчивость.

Таблица 1 – Сравнение масштабируемости

|  |  |  |
| --- | --- | --- |
| **Система** | **Масштабирование** | **Характеристика** |
| PostgreSQL | Вертикальное | Один узел, ограниченные ресурсы |
| ClickHouse | Горизонтальное | Кластеризация, репликация |
| Druid | Горизонтальное | Микросервисная архитектура |
| Hive | Пакетная обработка | Работает на Hadoop, медленная |

Типы хранилищ и их роль.

* Реляционные базы данных (RDBMS): хорошо подходят для OLTP и базовой аналитики, но ограничены по скорости при больших объемах данных.
* Колоночные базы данных: хранят данные по колонкам, что ускоряет агрегирующие операции.
* Многомерные кубы: предагрегированные структуры, быстро обрабатывающие аналитические запросы, но плохо масштабируются.

Таблица 2 – Сравнение хранилищ

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип хранилища** | **Пример** | **Преимущества** | **Недостатки** |
| Реляционное (RDBMS) | PostgreSQL | Простота, стабильность | Низкая скорость агрегации |
| Колоночное | ClickHouse | Высокая скорость, масштабируемость | Требует кластерной инфраструктуры |
| Многомерное (кубы) | SSAS, Essbase | Мгновенный отклик, Drill-Down | Статичность, сложная настройка |

Подходы к агрегации данных.

* Предварительная агрегация (pre-aggregation): используется для повышения производительности. Часто реализуется в виде материализованных представлений.
* On-the-fly агрегация: применяется в колоночных БД, где агрегаты вычисляются при выполнении запроса.
* Гибридный подход: часть метрик предагрегирована, часть рассчитывается динамически.

Обработка потоков данных.

Современные системы часто используют потоковую обработку данных. Например, Apache Kafka в связке с ClickHouse или Apache Flink позволяет агрегировать метрики в реальном времени.

Безопасность и управление доступом.

Системы должны обеспечивать разграничение прав, шифрование, аудит. ClickHouse поддерживает контроль пользователей, настройки доступа к базам и таблицам.

Микросервисный подход, обеспечивающий гибкость и масштабируемость.

Микросервисная архитектура предполагает разбиение системы на независимые сервисы, каждый из которых отвечает за свою бизнес-функцию (агрегация, хранение, визуализация, аутентификация и т.д.).

Преимущества микросервисного подхода:

* Высокая гибкость: можно быстро вносить изменения в отдельные сервисы без нарушения всей системы;
* Масштабируемость: отдельные компоненты масштабируются независимо;
* Устойчивость к сбоям: отказ одного сервиса не приводит к остановке всей системы;
* Независимость технологий: можно использовать разные языки программирования и СУБД.

Недостатки:

* Сложность в настройке, мониторинге и тестировании;
* Необходимость организовать надёжную систему взаимодействия между сервисами (API, очереди сообщений);
* Повышенные требования к DevOps-инфраструктуре.

Использование облачных технологий, позволяющее динамически изменять ресурсы под нагрузку.

Облачные платформы (AWS, Google Cloud, Azure) предоставляют возможность построения масштабируемых систем агрегирования с динамическим выделением ресурсов. Это особенно актуально для обработки больших объемов данных с переменной нагрузкой.

Преимущества облачного подхода:

* Эластичность: ресурсы увеличиваются и уменьшаются автоматически;
* Меньшие капитальные затраты на инфраструктуру;
* Быстрый запуск и развёртывание;
* Высокая доступность и отказоустойчивость.

Недостатки:

* Зависимость от провайдера (vendor lock-in);
* Возможные высокие расходы при неправильной настройке;
* Требования к безопасности и соблюдению законодательства (особенно в финансовом секторе).
  1. **Исследование методов выполнения аналитических запросов**
     1. **Введение в аналитические запросы**

Аналитические запросы предназначены для обработки больших объемов данных с целью выявления закономерностей, трендов и аномалий. Они широко используются в финансовой аналитике для расчета премий, оценки рисков и прогнозирования.

* + 1. **Основные методы выполнения аналитических запросов**

OLAP (Online Analytical Processing) [1].

OLAP-технология предназначена для многомерного анализа данных и позволяет выполнять сложные аналитические запросы [2,3].

Основные типы OLAP:

* MOLAP (Multidimensional OLAP) — хранение данных в многомерных кубах.
* ROLAP (Relational OLAP) — хранение данных в реляционных таблицах, обработка с помощью SQL-запросов.
* HOLAP (Hybrid OLAP) — гибридный подход, сочетающий MOLAP и ROLAP.

Таблица 3 – Сравнение MOLAP и ROLAP

|  |  |  |
| --- | --- | --- |
| **Параметр** | **MOLAP** | **ROLAP** |
| Хранилище данных | Специализированный многомерный куб | Реляционная БД (звезда/снежинка) |
| Агрегация | Предагрегированные данные | Выполняется «на лету» SQL |
| Скорость запросов | Очень высокая (за счёт pre-computation) | Зависит от мощности БД и индексации |
| Гибкость схемы | Низкая: перестройка куба при изменениях | Высокая: схему можно менять без долгих процессов |
| Объём данных | Эффективно до десятков–сотен ГБ, выше — рост куба | Практически неограничен (зависит от СУБД) |
| Накладные расходы | Время и ресурсы на строительство куба | Загрузка на DWH при выполнении запросов |
| Язык запросов | MDX или проприетарный | SQL |

MapReduce.

Метод MapReduce позволяет обрабатывать большие объемы данных параллельно на распределенных системах. Этот метод эффективен для работы с неструктурированными и полуструктурированными данными.

MapReduce не удовлетворяет требованиям оперативной, низкозадержанной аналитики, необходимой для расчета премий в реальном времени. Он может быть использован лишь для фоновых, ночных пакетных расчетов — но не как основа системы, ориентированной на актуальные бизнес-задачи.

SQL и NoSQL-подходы.

SQL-методы: Используются в традиционных реляционных БД (PostgreSQL, MySQL, Oracle) для аналитических запросов с агрегацией (SUM, AVG, COUNT, GROUP BY).

Листинг 1 – Пример простого запроса

SELECT Region, SUM(Sales) AS TotalSales

FROM sales\_data

WHERE SaleDate BETWEEN '2023-01-01' AND '2023-06-30'

GROUP BY Region;

NoSQL-методы: Используются в документоориентированных, графовых и других БД (MongoDB, Cassandra) для обработки данных в реальном времени.

Python [10]: часто интегрируется для гибкой аналитики и ETL. Например, с помощью pandas можно прочесть данные из DWH и сделать анализ:

Листинг 1 – Пример предобработки и вывода группировки

import pandas as pd  
# Загружаем данные через SQL-запрос  
df = pd.read\_sql('SELECT \* FROM sales\_data', connection)  
# Вычисляем новый столбец и сводим по региону  
df['RevenueUSD'] = df['Revenue'] \* df['Rate\_USD']  
print(df.groupby('Region')['RevenueUSD'].sum())

Для данной работы выбираем SQL-методы, так как для аналитики необходимы аналитические запросы с агрегацией.

Индексы и материализованные представления.

Для ускорения аналитических запросов применяются индексы (B-деревья, Bitmap-индексы) и материализованные представления, хранящие предварительно рассчитанные результаты запросов.

* 1. **Исследование особенностей работы оперативной аналитической обработки**
     1. **Определение оперативной аналитической обработки (OLAP)**

Оперативная аналитическая обработка [4,5] (OLAP) представляет собой технологию, обеспечивающую быстрый доступ к агрегированным данным в многомерных структурах. OLAP используется в системах финансовой аналитики для мгновенного расчета показателей, таких как премии, рентабельность и финансовые риски.

* + 1. **Основные принципы OLAP**

Многомерность данных: Данные организованы в виде кубов с различными измерениями (время, категория, география).

Агрегация: Данные сводятся к обобщенным показателям, что снижает объем вычислений.

Оптимизация хранения: Используются специальные структуры данных для быстрого доступа.

* + 1. **Технологии оперативной аналитической обработки**

Среди наиболее распространенных технологий OLAP можно выделить:

Apache Druid: Высокопроизводительная аналитическая база данных для работы с потоковыми и историческими данными.

ClickHouse: Колонночная СУБД с высокой скоростью выполнения аналитических запросов.

Microsoft Analysis Services: Инструмент для работы с OLAP-кубами в экосистеме Microsoft.

* + 1. **Преимущества и недостатки OLAP**

**Преимущества:**

* Высокая скорость выполнения запросов за счет предварительной агрегации данных.
* Возможность многомерного анализа данных.
* Поддержка сложных аналитических вычислений.

**Недостатки:**

* Высокие затраты на вычислительные ресурсы.
* Ограниченная гибкость по сравнению с транзакционными базами данных.
* Сложность настройки и поддержки.
  + 1. **Системы, использующие OLAP**

Oracle OLAP [6,7] (Рис. 1)– интегрирован в Oracle Database, обеспечивает мощные аналитические функции для обработки многомерных данных.

Минусы: высокая стоимость лицензирования, сложность настройки и администрирования, требует значительных вычислительных ресурсов.

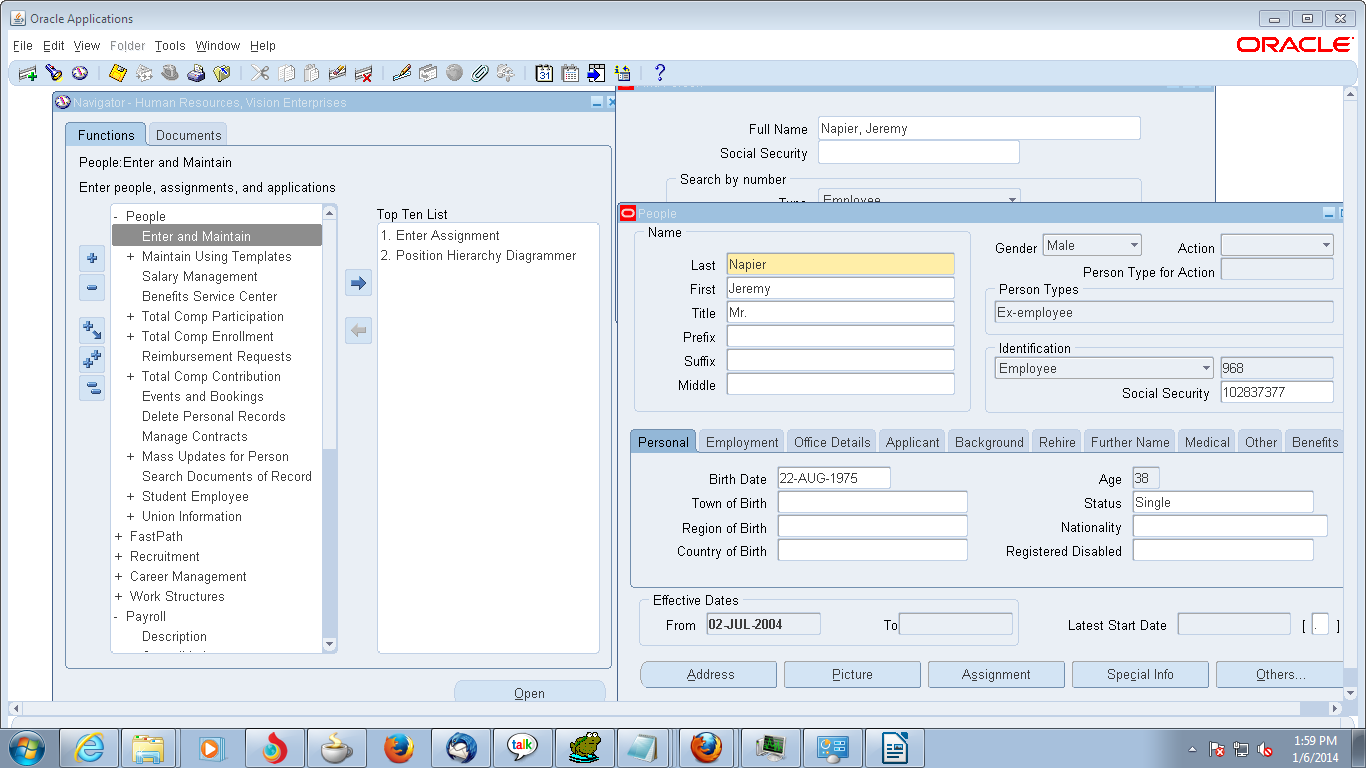


Рисунок 1 – Интерфейс Oracle OLAP

Microsoft Analysis Services [8] (Рис. 2) – мощный инструмент для построения OLAP-кубов и выполнения аналитических запросов.

Минусы: ограниченная поддержка неструктурированных данных, зависимость от экосистемы Microsoft, требует профессиональной настройки.

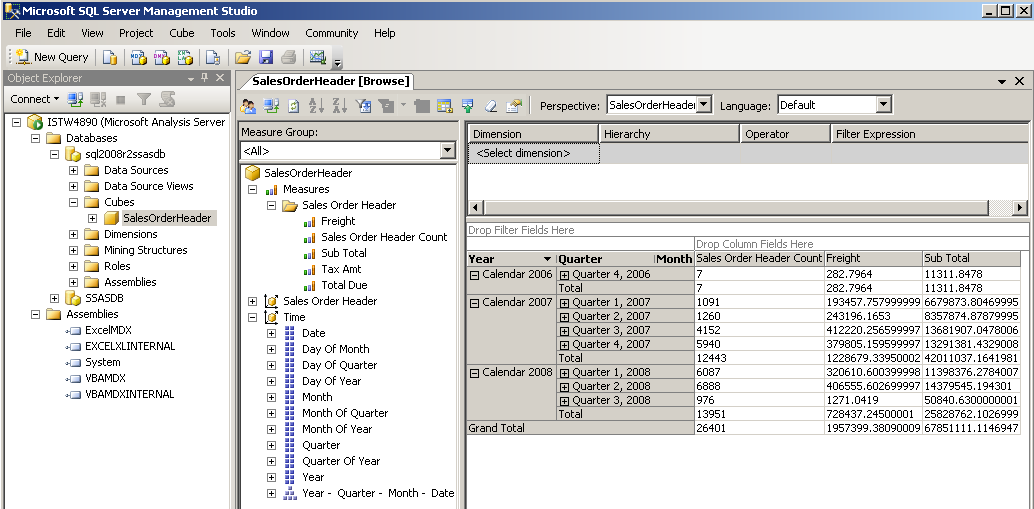


Рисунок 2 – Интерфейс Microsoft Analysis Services

IBM Cognos TM1 (Рис. 3) – высокопроизводительная платформа для финансового моделирования и планирования.

Минусы: высокая стоимость, сложность интеграции с внешними системами, требует квалифицированных специалистов.

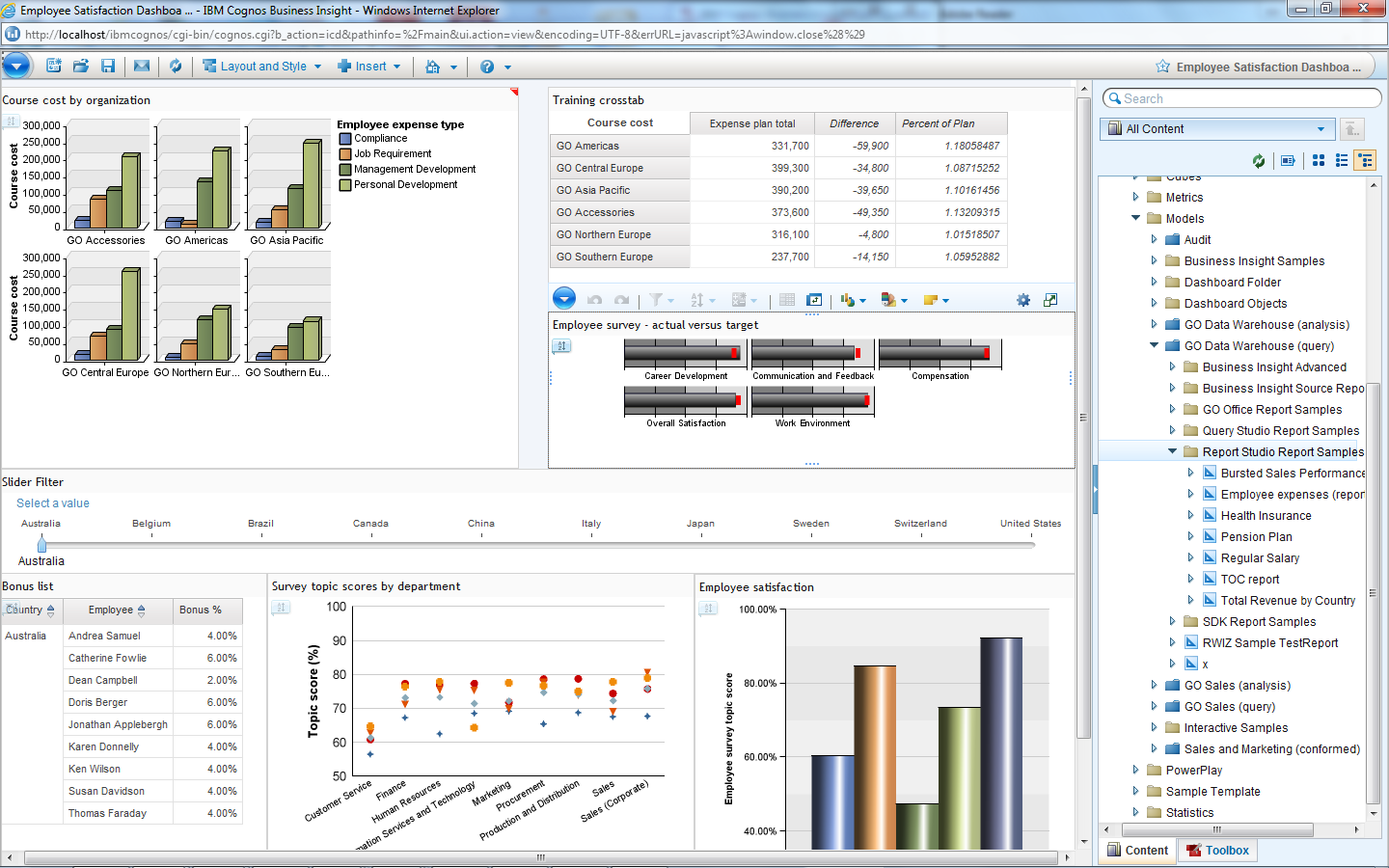


Рисунок 3 – Интерфейс IBM Cognos TM1

SAP BW/4HANA [9] (Рис. 4) – корпоративное хранилище данных с OLAP-функциями, работает в in-memory режиме.

Минусы: высокая стоимость лицензий и инфраструктуры, требует больших объемов оперативной памяти, сложность администрирования.

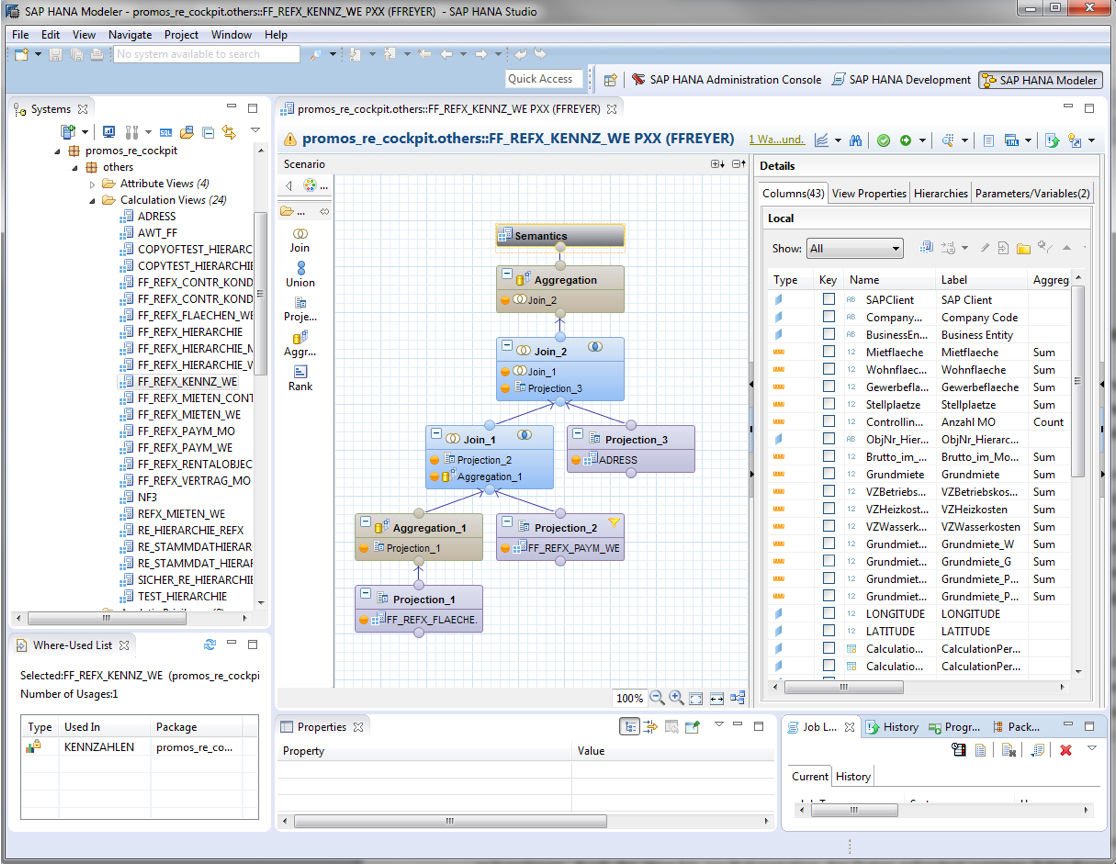


Рисунок 4 – Интерфейс SAP BW/4HANA

Amazon Redshift – облачное хранилище данных с поддержкой OLAP-запросов и масштабируемой аналитикой.

Минусы: высокая стоимость при больших объемах данных, зависимость от AWS-инфраструктуры, задержки при обработке сложных многомерных запросов.

Каждая из этих систем обладает своими характеристиками и подходами. Например, Oracle OLAP интегрирован в реляционную СУБД и поддерживает сложные аналитические запросы, а Amazon Redshift предлагает горизонтальное масштабирование и облачную доступность. Microsoft SSAS ориентирован на удобство работы с Excel и Power BI, а SAP BW/4HANA обеспечивает высокую скорость обработки за счет технологии in-memory.

* + 1. **Заключение**

Для реализации системы агрегирования данных, ориентированной на оперативную финансовую аналитику и расчет премий, наиболее рациональным выбором является использование системы управления базами данных ClickHouse. Эта технология представляет собой колоночную СУБД, оптимизированную для высокоскоростной аналитической обработки больших объемов данных. В отличие от традиционных строковых СУБД, ClickHouse обеспечивает быструю агрегацию и фильтрацию по нужным измерениям, что критично для построения OLAP-запросов и финансовой отчетности. Среди основных преимуществ системы — поддержка многомерного анализа (OLAP), высокая масштабируемость, сжатие данных и возможность обрабатывать миллиарды строк в секунды. Кроме того, ClickHouse легко интегрируется с BI-средствами и имеет низкие требования к инфраструктуре по сравнению с корпоративными решениями вроде SAP BW или Oracle OLAP. Таким образом, с учетом требований к скорости, надежности и эффективности при работе с финансовыми данными в реальном времени, ClickHouse представляет собой оптимальную технологическую основу для разработки данной системы.

1. **Исследование подхода к OLAP технологии**

Каждая из этих систем обладает своими характеристиками и подходами. Например, Oracle OLAP интегрирован в реляционную СУБД и поддерживает сложные аналитические запросы, а Amazon Redshift предлагает горизонтальное масштабирование и облачную доступность. Microsoft SSAS ориентирован на удобство работы с Excel и Power BI, а SAP BW/4HANA обеспечивает высокую скорость обработки за счет технологии in-memory.

* 1. **Анализ и сравнение существующих подходов**

OLAP (Online Analytical Processing) — это технология многомерной аналитики, предназначенная для оперативного анализа больших объемов данных по различным измерениям (время, регионы, клиенты, продукты и т.д.). Она позволяет выполнять сложные аналитические запросы с минимальной задержкой и широко используется в бизнес-аналитике, финансовом моделировании, логистике, телекоммуникациях и других отраслях.

* + 1. **ROLAP (Relational OLAP)**

ROLAP работает поверх реляционных баз данных и использует SQL-запросы для выполнения аналитических операций. Данные хранятся в обычных таблицах, но дополнительно создаются агрегаты и представления (views), оптимизированные под OLAP-запросы.

Агрегация происходит во время выполнения запроса (on-the-fly). Используются индексы, материализованные представления, партиционирование. Возможна работа с большими объемами исходных данных без предварительной агрегации.

**Преимущества:**

* Гибкость.
* Масштабируемость.
* Использование стандартного SQL.

Можно легко изменять структуру запросов и измерения. Хорошо работает с современными колоночными СУБД. Облегчает интеграцию и обучение персонала

**Недостатки:**

* Производительность зависит от оптимизации запросов.
* Высокая нагрузка.

Зависит от оптимизации запросов. Высокая нагрузка на СУБД при больших объемах данных.

Используется в таких СУБД: ClickHouse, PostgreSQL (с расширениями), Amazon Redshift, Google BigQuery.

Применяется в финансовых платформах «Тинькофф Аналитика», BI-системах Tableau и Metabase.

* + 1. **MOLAP (Multidimensional OLAP)**

MOLAP использует специально подготовленные многомерные структуры (кубы), в которых данные агрегируются и хранятся заранее. Эти кубы представляют собой нативное OLAP-хранилище, не зависящее от реляционной модели.

Данные проходят процесс ETL и загружаются в многомерные кубы. Кубы агрегируют значения по измерениям заранее (pre-aggregation). Запросы к кубам выполняются почти мгновенно, так как обращаются к уже рассчитанным данным.

**Преимущества:**

* Очень высокая скорость отклика.
* Эффективность для предсказуемых сценариев анализа.

**Недостатки:**

* Ограниченная гибкость: изменение структуры куба требует перестроения.
* Трудности масштабирования при больших объемах данных.

Используется в таких система: Microsoft SQL Server Analysis Services (SSAS), IBM Cognos, Oracle Essbase.

Используется в крупных корпорациях с фиксированными аналитическими сценариями (например, отчетность по кварталам).

* + 1. **HOLAP (Hybrid OLAP)**

HOLAP — это гибридный подход, который совмещает в себе преимущества ROLAP и MOLAP. Детальные данные хранятся в реляционной базе (ROLAP), а агрегированные — в кубах (MOLAP).

Кубы содержат агрегаты для часто используемых запросов. Подробные значения запрашиваются из реляционной базы. Система определяет, откуда извлекать данные в зависимости от запроса.

**Преимущества:**

* Компромисс между гибкостью и производительностью.
* Возможность обработки больших объемов без потери скорости.

**Недостатки:**

* Сложность реализации и поддержки.
* Необходимость синхронизации кубов и таблиц.

Используется в: SAP BW, Microsoft SSAS (в гибридном режиме), IBM Cognos TM1. Применяется в крупных банках, ритейле, логистике.

Таблица 4 – Сравнение MOLAP, ROLAP и HOLAP

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Подход** | **Хранение данных** | **Скорость агрегаций** | **Гибкость** | **Масштабируемость** | **Недостатки** |
| ROLAP | Реляционные таблицы | Средняя | Высокая | Высокая | Медленная агрегация, нагрузка на СУБД |
| MOLAP | Многомерные кубы | Очень высокая | Низкая | Средняя | Требует подготовки, ограничен по объему |
| HOLAP | Комбинированное | Высокая | Средняя | Средняя | Сложная реализация, дублирование логики |

После детального анализа архитектур OLAP-технологий видно, что ROLAP обеспечивает необходимую гибкость и масштабируемость, особенно в условиях постоянно меняющейся финансовой отчетности. Несмотря на то, что MOLAP превосходит по скорости при фиксированных сценариях, его ограниченность в динамической бизнес-среде делает его менее подходящим. HOLAP представляет интерес как компромисс, однако его сложность и ресурсоемкость внедрения не оправданы в рамках данной задачи. Поэтому в данной работе выбран **MOLAP-подход** с реализацией на базе **ClickHouse** — колоночной СУБД, предоставляющей быстрый отклик, нативную поддержку аналитических функций и совместимость с SQL-запросами. Для улучшения MOLAP-подхода изменим способ хранения уже агрегированных данных для того чтобы увеличить скорость выдачи данных и уменьшить объем хранимых агрегированных данных.

* 1. **Разработка алгоритма для системы агрегирования данных**

Описание входных данных.

Для формирования куба, необходимо его описание, какие у него есть грани и их уровень вложенности. Для этого составим его описание.

Листинг 2 – Описание структуры куба

{

"name": "Тестовое создание куба",

"id": 120,

"analytics": [

{

"name": "X",

"id\_length": 2,

"indexing": null

},

{

"name": "Y",

"id\_length": 2,

"indexing": null

}

],

"access\_analytics": null,

"analytics\_columns": {

"X": [

"x\_1",

"x\_2",

"x\_3"

],

"Y": [

"y\_1",

"y\_2",

"y\_3"

]

},

"analytics\_merge": [],

"data\_columns": [

{

"name": "Значения",

"visible": true,

"aggregation": [

{

"analytics": "X",

"aggregation\_id": 0

},

{

"analytics": "Y",

"aggregation\_id": 0

}

]

}

],

"data\_columns\_order": [

{

"name": "Значения",

"mock": false,

"order": 0,

"id": 1744567412161

}

]

}

Разберем данное описание:

* name – название куба
* id – уникальный идентификатор куба
* analytics – список граней куба
* name – название столбца аналитики
* id\_length – длина индекс (количество уникальных значений)
* indexing – индексация
* access\_analytics – доступ к аналитике
* analytics\_columns – уровни граней куба
* analytics\_merge – если аналитика является общей
* data\_columns – описание данных
* name – название столбца с данными
* visible – видимость столбца с данными
* aggregation – грани для расчета данных
* analytics – название грани куба для формирования данного столбца с данными
* aggregation\_id – если является отдельным расчетным значением
* data\_columns\_order – описание столбца данных

Сформируем данные:

Таблица 5 – Пример стартовых данных

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x\_3 | x\_2 | x\_1 | y\_3 | y\_2 | y\_1 | Данные |
| x31 | x21 | x11 | y31 | y21 | y11 | 100 |
| x31 | x21 | x12 | y31 | y21 | y12 | 200 |
| x31 | x22 | x13 | y31 | y22 | y13 | 300 |
| x31 | x22 | x14 | y31 | y22 | y14 | 400 |

Количество комбинаций возможной выборки для аналитики составляет 84, в нее входят как листья, так и корневые данные.

Обработка входных данных.

Листинг 3 – Функция по обработке загружаемых данных в куб

def upload(df, cube\_id, engine, user\_id=None, current\_user=None, dtype\_dist = None, new\_data = True, file\_name = 'no filename passed'):

if current\_user is not None:

user\_id = current\_user.id

db\_prefix = 'cube'

query = f''' SELECT \* FROM cubes WHERE id = '{cube\_id}' LIMIT 1 '''

cube\_in\_cubes = galileo.db.query(query, engine=engine, return\_df = True)

cube\_name = cube\_in\_cubes['name'].squeeze()

if len(cube\_in\_cubes) == 0:

return {'code': 0, 'message': f"Куб с id {cube\_id} не найден", "data": None }

if new\_data: table\_name = f'{db\_prefix}\_{cube\_id}\_newdata'

else: table\_name = f'{db\_prefix}\_{cube\_id}\_data'

query = f''' SELECT \* FROM {table\_name} LIMIT 1'''

new\_table\_flag = False

try:

galileo.db.query(query, engine=engine, return\_df = False)

except:

new\_table\_flag = True

## проверка наличия необходимых столбцов

query = f''' SELECT \* from cube\_{cube\_id}\_info ORDER BY primary\_key'''

structure\_response = galileo.cubes.get\_structure(cube\_id=cube\_id, engine=engine, convert=True, old\_format = False, user\_id = None)

if structure\_response['code'] == 0:

return { 'code': 0, 'message': 'Оишбка получения структуры куба', 'data': None }

structure = structure\_response['data']['structure']

mapping = structure\_response['data']['mapping']

analitic\_columns = structure\_response['data']['analitic\_columns']

not\_mapped\_analitic\_columns = structure\_response['data']['not\_mapped\_analitic\_columns']

rewrite\_group\_names = structure\_response['data']['rewrite\_group\_names']

data\_columns = structure\_response['data']['data\_columns']

group\_names = structure\_response['data']['group\_names']

rewrite\_columns = structure\_response['data']['rewrite\_columns']

merge\_analytics= structure\_response['data']['structure']['analytics\_merge']

missing\_columns = []

for col in not\_mapped\_analitic\_columns:

if col not in list(df.columns):

missing\_columns.append(col)

df[col] = np.nan

else:

df[col] = df[col].astype('string')

if len(missing\_columns)>0:

return {'code': 0, 'message': f'''не хватает столбцов для аналитиков: {", ".join([f"'{col}'" for col in missing\_columns])} ''', "data": None }

missing\_columns = []

not\_float\_data\_columns = []

for col in data\_columns:

if col not in list(df.columns):

df[col] = 0

else:

try:

df[col] = df[col].astype('float')

except:

not\_float\_data\_columns.append(col)

df[col] = df[col].fillna(0)

if len(missing\_columns)>0:

return {'code': 0, 'message': f'''Не хватает столбцов с данными: {", ".join([f"'{col}'" for col in missing\_columns])} ''', "data": None }

if len(not\_float\_data\_columns)>0:

return {'code': 0, 'message': f'''Не удалась конвертация в число в столбцах с данными: {", ".join([f"'{col}'" for col in not\_float\_data\_columns])}, проверьте, нет ли ячеек со значениями отличными от числа (или пустых) ''', "data": None }

cols = not\_mapped\_analitic\_columns + data\_columns

df = df[cols + list(df.columns.difference(cols))]

for analitic\_column in not\_mapped\_analitic\_columns:

# print(df[analitic\_column])

df[analitic\_column] = df[analitic\_column].map(lambda x: str(x).replace("'", '"'))

df[analitic\_column] = df[analitic\_column].map(lambda x: re.sub('''[^0-9а-яА-Яa-zA-Z!?\_,.()#№:/&<>[]{}\*-" ]+''', '\*', str(x)))

for column\_name in df.columns.difference(data\_columns):

df[column\_name] = df[column\_name].map(lambda x: str(x)[:128].strip())

month\_columns = []

for column in not\_mapped\_analitic\_columns:

if all([str(row\_value).lower().strip() in galileo.months\_names for row\_value in list(df[column].unique())]):

month\_columns.append(column)

df[column].apply(lambda x: x.lower().strip())

df[column] = df[column].replace(galileo.month\_dict)

df[column] = df[column].astype('int')

df = df.sort\_values(by=list(df.columns))

for column in month\_columns:

df[column] = df[column].replace(galileo.month\_dict\_reverse)

df = df.reset\_index(drop = True)

non\_boolean\_columns = list(df.dtypes[df.dtypes != 'boolean'].index)

df[non\_boolean\_columns] = df[non\_boolean\_columns].replace({'None': None})

df[non\_boolean\_columns] = df[non\_boolean\_columns].replace({'nan': None})

df[non\_boolean\_columns] = df[non\_boolean\_columns].replace({'<NA>': np.nan})

df[non\_boolean\_columns] = df[non\_boolean\_columns].replace({'\*NA\*': np.nan})

dtype\_dist = dict()

for analitic\_column in not\_mapped\_analitic\_columns:

dtype\_dist[analitic\_column] = String

for data\_column in data\_columns:

dtype\_dist[data\_column] = Float

for other\_column in list(df.columns.difference(not\_mapped\_analitic\_columns + data\_columns)):

dtype\_dist[other\_column] = String

if new\_table\_flag: ## Скорее всего никгда не будет исполняться т.к. таблицы создаются при define куба

if dtype\_dist is not None:

dtype\_dist = dict()

for col\_name in not\_mapped\_analitic\_columns:

dtype\_dist[col\_name] = String

for col\_name in data\_columns:

dtype\_dist[col\_name] = Float

upload\_response = galileo.db.upload\_df\_to\_postgres(df=df, table\_name=table\_name, engine=engine, chunksize = 10000000,

append = False, cols = list(df.columns), log = False, string\_default\_length = 128,

dtype\_dist = dtype\_dist, col\_name\_max\_length = None, primary\_key = None, skip\_check\_col = False)

if upload\_response['code'] == 1:

log\_response = galileo.db.cubes.log( cube\_id = cube\_id, action = 'upload', value = file\_name,

user\_id = user\_id, engine = engine )

if log\_response['code'] != 1:

log\_message = ' (ошибка логирования)'

else:

log\_message = ''

return {'code': 1, 'message': f'''Создана очередь на сборку для куба "{cube\_name}" {log\_message}''', "data": None}

else:

return {'code': 0, 'message': upload\_response['message'], 'error\_message': upload\_response.get('error\_message', None), "data": None }

else:

constraint\_columns =merge\_analytics #rewrite\_columns

if len(constraint\_columns) > 0:

update\_response = galileo.db.update\_df\_in\_postgres(df=df, constraint\_cols=constraint\_columns, table\_name=table\_name, engine = engine,

cols = list(df.columns), constraint\_values = None, col\_name\_max\_length = 30, dtype\_dist = dtype\_dist,

check\_for\_int = False, skip\_check\_col = False, )

else:

update\_response = galileo.db.upload(df=df, table\_name=table\_name, engine = engine, append = True,

log = False, cols = list(df.columns), col\_name\_max\_length = 30, string\_default\_length = 128,

dtype\_dist = dtype\_dist, primary\_key = None, skip\_check\_col = False, )

if update\_response['code'] == 1:

return {'code': 1, 'message': f'''Обновлена очередь на сборку куба "{cube\_name}": {update\_response['message']}''', 'error\_message': update\_response.get('error\_message', None), 'data': update\_response['data']}

else:

return {'code': 0, 'message': update\_response['message'], 'error\_message': update\_response.get('error\_message', None), 'data': update\_response['data'] }

Данный код реализует проверку загружаемых данных, с тем как была описана структура куба.

Функция является универсальным загрузчиком данных в OLAP-кубы, включающим, валидацию данных и типов, поддержку обновлений и новых загрузок, обработку русскоязычных значений (в т.ч. месяцев). интеграцию с системой логирования и аудита.

Аргументы:

df — входной DataFrame с загружаемыми данными;

cube\_id — идентификатор куба;

engine — SQLAlchemy engine для подключения к БД;

user\_id, current\_user — идентификаторы пользователя;

dtype\_dist — словарь с типами данных колонок;

new\_data — флаг, указывающий, загружаются ли новые данные или обновляются старые;

file\_name — имя файла, из которого были загружены данные (для логирования).

Так как описанная структура куба, лежит в базе данных, необходимо её получить при помощи запроса SQL, далее распаршиваем структуру на отдельные части, проверяем в загруженном массиве данных (далее DataFrame), что перечисленные аналитики и грани куба указаны, что заданные столбцы с данными так же присутствуют. Проверка столбцов данных, на числовые и строковые значения, в столбцах данных, не должны присутствовать строковые значения, так как данные в этих столбцах должны выполнять различные арифметические операции, что невозможно сделать со строковыми значениями.

Проверка наличия нужных колонок. Проверка, что в df есть все необходимые аналитические колонки. Добавление недостающих с NaN. Преобразование data\_columns в float и обработка исключений.

Очистка и нормализация данных.

Очистка текстов от недопустимых символов (через re.sub). Ограничение длины строк (до 128 символов). Преобразование названий месяцев в числовой формат и обратно. Удаление лишних значений типа 'None', 'nan', '<NA>'.

Создание таблицы (если не существует). Если таблица не существует, она создается — с учётом типов и первичных ключей. Возвращается сообщение об успешной загрузке и логируется операция.

Обновление или добавление данных. Если данные в кубе уже присутствуют, загружаемые данные являются либо дополнительными для досборки куба, либо новыми, в зависимости от типа загрузки.

Основной идеей формирование куба, заключается в том, чтобы формировать каждый последующий разворот куба мы будет на основе предыдущего, так идя он начальной таблице, мы постепенно будет ее увеличивать, добавляя сгруппированные значения и из полученной таблице будем рекурсивно группировать её по каждой иерархии аналитика (рис. 1).

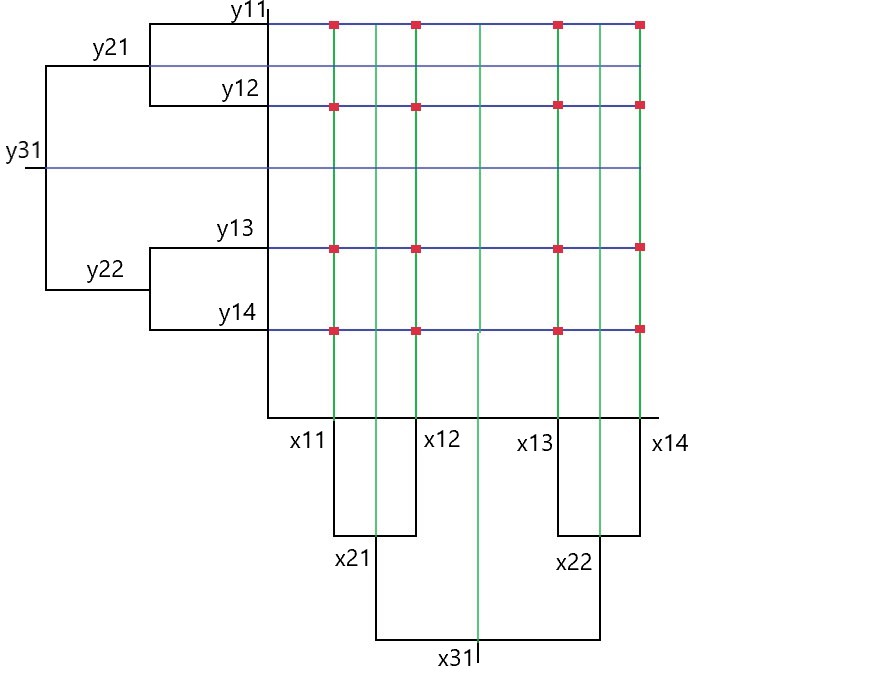


Рисунок 5 – Агрегация в двумерной плоскости

На рис. 5 представлена первая агрегация, при которой формируются значения на листьях. Добавляя полученные значения к основным, мы можем делать группировку следующего уровня иерархии аналитика. Нам не приходится каждый раз рассчитывать куб до нужного уровня, поскольку при каждой агрегации уровня иерархии аналитика, сформированные данные уже будут, но используя больше памяти при каждой последующей агрегации над кубом. Данную проблему можно решить тем, что каждый столбец данных считать отдельно, последовательно. Таким образом если у нас в изначальных столбцах данных находятся 100 столбцов, мы соберем куб 100 раз для каждого столбца, следовательно, мы уменьшим объем потребляемой памяти при формировании куба, но скорость формировании всего куба увеличиться.

* + 1. **Как формируются индексы**

С точки зрения математики для того чтобы хранить плоскость на прямой необходимо каждой точки дать свой уникальный индекс (номер), таким образов если у нас многомерная плоскость, то для каждой из плоскостей нужно присвоить свой индекс. Сформированный куб это и есть многомерная пространство плоскостей. Каждый индекс будет состоять из индексов аналитика.

Для того чтобы сформировать индекс аналитика, нужно подготовить таблицу, в которой они будут храниться. Данная таблица будет состоять из уникальных значений каждого уровня иерархии аналитика, в которой в столбце «Имя» будут находится имена всех уникальных значений аналитики, а в столбце «Индекс» число для каждого уникального значения в столбце «Имя».

Перед подстановкой индексов, нужно отредактировать с агрегированные данные. Для этого нужно создать столбы с названиями всех аналитиков и подставить первое не пустое значение из названий столбцов иерархии аналитика. Таки образов в итоговой таблицу окажется количество столбцов равное количеству аналитиков.

Далее необходимо заменить значения в столбцах на значение в таблице с индексами и соединить каждую строку в одну ячейку. Таким образом получаться индексы для каждого значения OLAP-куба.

Преобразование данных для расчета.

Понадобиться таблицы в которой хранятся индексы всех уникальных элементов, которые используются при аналитике.

Таблица 6 – Уникальные индексы для данных

|  |  |  |  |
| --- | --- | --- | --- |
| level | index | axis | name |
| 0 | 0 | 0 |  |
| 1 | 9 | 0 | x1 |
| 2 | 4 | 0 | x21 |
| 2 | 8 | 0 | x22 |
| 3 | 1 | 0 | x31 |
| 3 | 2 | 0 | x32 |
| 3 | 3 | 0 | x33 |
| 3 | 5 | 0 | x34 |
| 3 | 6 | 0 | x35 |
| 3 | 7 | 0 | x36 |
| 3 | 10 | 0 | x33 |
| 0 | 0 | 1 |  |
| 1 | 9 | 1 | y1 |
| 2 | 4 | 1 | y21 |
| 2 | 8 | 1 | y22 |
| 3 | 1 | 1 | y31 |
| 3 | 2 | 1 | y32 |
| 3 | 3 | 1 | y33 |
| 3 | 5 | 1 | y34 |
| 3 | 6 | 1 | y35 |
| 3 | 7 | 1 | y36 |
| 3 | 10 | 1 | y33 |
| 0 | 0 | 2 |  |
| 1 | 1 | 2 | Значения |

Листинг 4 – Часть кода по составлению уникальных индексов

for axis in range(len(group\_names)):

# максимальный индекс аналитика +1

new\_index = all\_nodes\_df[all\_nodes\_df['axis'] == axis]['index'].max() + 1

## Полная таблица сущесвтующего налитика куба

existed\_axis\_nodes\_df = all\_nodes\_df.copy()[all\_nodes\_df['axis'] == axis]

existed\_axis\_edges\_df = all\_edges\_df.copy()[all\_edges\_df['axis'] == axis]

original\_existed\_axis\_nodes\_df = pd.merge( existed\_axis\_nodes\_df, existed\_axis\_edges\_df, how='left', left\_on=['index', 'axis'], right\_on=['to', 'axis'] ).rename( { 'from': 'parent\_id', }, axis=1 ).drop('to', axis = 1)

for level in range(0, len(group\_columns[axis])):

if level == 0:

existed\_axis\_nodes\_df = original\_existed\_axis\_nodes\_df[original\_existed\_axis\_nodes\_df['level'] == 0]

if level != 0:

level\_df = level\_df.rename( { 'name': group\_columns[axis][level-1] }, axis = 1 )

level\_df = original\_existed\_axis\_nodes\_df[original\_existed\_axis\_nodes\_df['level'] == level+1]

level\_df = level\_df[['index', 'parent\_id', 'axis', 'name', 'level']]

level\_df = level\_df.rename( { 'name': group\_columns[axis][level] }, axis = 1 )

level\_df['level'] = level\_df['level']-1

new\_existed\_axis\_nodes\_df = pd.merge( existed\_axis\_nodes\_df, level\_df, how='right', left\_on=['index', 'axis', 'level'], right\_on=['parent\_id', 'axis', 'level'], suffixes=('', '\_parent') )

new\_existed\_axis\_nodes\_df=new\_existed\_axis\_nodes\_df.drop(['index', 'parent\_id\_parent'], axis=1)

new\_existed\_axis\_nodes\_df=new\_existed\_axis\_nodes\_df.rename({'index\_parent': 'index'}, axis=1)

new\_existed\_axis\_nodes\_df['level'] = new\_existed\_axis\_nodes\_df['level']+1

new\_existed\_axis\_nodes\_df['name'] = new\_existed\_axis\_nodes\_df[group\_columns[axis][level]]

existed\_axis\_nodes\_df = pd.concat( [ existed\_axis\_nodes\_df,new\_existed\_axis\_nodes\_df ] )

existed\_axis\_nodes\_df = existed\_axis\_nodes\_df.drop('parent\_id', axis = 1)

df\_columns = list(DB.query( f"SELECT \* FROM cube\_{cube\_id}\_data\_assemble LIMIT 1; " ,engine=engine, return\_df=True))

col\_list = df\_columns

for col\_name in data\_columns:

if col\_name in col\_list:

col\_list.remove(col\_name)

if 'Столбец данных' in group\_columns[axis]:

new\_df\_rewrite\_values= pd.DataFrame({group\_columns[axis][0]: data\_columns})

else:

new\_df\_rewrite\_values = DB.query( query= f''' SELECT DISTINCT ON ( {' , '.join([ f'"{str(x)}"' for x in group\_columns[axis] ])}) {' , '.join([ f'"{str(x)}"' for x in group\_columns[axis] ])} FROM cube\_{cube\_id}\_data\_assemble ''' , return\_df=True, engine=engine)

axis\_nodes\_df = pd.DataFrame()

for level in range(len(group\_columns[axis]), 0, -1):

col = group\_columns[axis][level-1]

level\_df = new\_df\_rewrite\_values.dropna(subset=[col])

level\_df.iloc[:,level:] = np.nan

level\_df['level'] = level

level\_df['axis'] = axis

level\_df['name'] = level\_df.iloc[:,level-1]

axis\_nodes\_df = pd.concat([axis\_nodes\_df, level\_df])

def convert\_month\_names(val):

if str(val).lower().strip() in galileo.months\_names:

return galileo.month\_dict\_reverse[galileo.month\_dict[str(val).lower().strip()]]

return val

axis\_nodes\_df['name'] = axis\_nodes\_df['name'].apply(lambda x: convert\_month\_names(x))

axis\_nodes\_df = axis\_nodes\_df.drop\_duplicates()

axis\_nodes\_df = pd.merge( existed\_axis\_nodes\_df, axis\_nodes\_df, how='outer', on=['axis', 'level'] + group\_columns[axis], suffixes=('', '\_y') )

axis\_nodes\_df.loc[axis\_nodes\_df['name'].isna(),'name'] = axis\_nodes\_df.loc[axis\_nodes\_df['name'].isna(),'name\_y']

axis\_nodes\_df = axis\_nodes\_df.drop('name\_y', axis = 1)

new\_axis\_df = axis\_nodes\_df[axis\_nodes\_df['index'].isna()]

new\_axis\_df = new\_axis\_df.sort\_values(by = group\_columns[axis])

new\_axis\_df.loc[:,'index'] = list(range(new\_index, new\_index + len(new\_axis\_df)))

axis\_nodes\_df = pd.concat( [ axis\_nodes\_df[~axis\_nodes\_df['index'].isna()], new\_axis\_df ] )

axis\_nodes\_df = axis\_nodes\_df[ ['index', 'axis', 'level', 'name'] + group\_columns[axis][::-1] ]

new\_axis\_df = new\_axis\_df[ ['index', 'axis', 'level', 'name'] + group\_columns[axis][::-1] ]

axis\_nodes\_df['index'] = axis\_nodes\_df['index'].astype('int')

axis\_nodes\_df['level'] = axis\_nodes\_df['level'].astype('int')

axis\_nodes\_df['axis'] = axis\_nodes\_df['axis'].astype('int')

axis\_nodes\_df = axis\_nodes\_df.fillna(np.nan).replace({np.nan: None})

new\_axis\_df['index'] = new\_axis\_df['index'].astype('int')

new\_axis\_df['level'] = new\_axis\_df['level'].astype('int')

new\_axis\_df['axis'] = new\_axis\_df['axis'].astype('int')

new\_axis\_df = new\_axis\_df.fillna(np.nan).replace({np.nan: None})

nodes5\_df\_list[axis] = axis\_nodes\_df

new\_nodes\_df = pd.concat([new\_nodes\_df, new\_axis\_df])

new\_nodes.extend( list( new\_axis\_df.T.to\_dict().values() ) )

new\_nodes\_full.extend( list( new\_axis\_df.T.to\_dict().values() ) )

## Рассчёт новых рёбер аналитика

levels\_df\_list = []

for level in range(len(group\_columns[axis]), 0, -1):

level\_children\_df = axis\_nodes\_df[axis\_nodes\_df['level'] == level]

level\_parent\_df = axis\_nodes\_df[axis\_nodes\_df['level'] == level-1]

level\_parent\_df = level\_parent\_df[['index'] +group\_columns[axis][:level-1]]

level\_parent\_df = level\_parent\_df.rename( {'index': 'parent\_index'}, axis = 1 )

if level != 1:

level\_children\_df = pd.merge( level\_children\_df, level\_parent\_df, how='left', on=group\_columns[axis][:level-1], suffixes=['', '\_y'] )

else:

level\_children\_df['parent\_index'] = level\_parent\_df['parent\_index'].iloc[0]

levels\_df\_list.append(level\_children\_df)

levels\_df\_list.append(axis\_nodes\_df[axis\_nodes\_df['level'] == 0])

axis\_edges\_df = pd.concat(levels\_df\_list).sort\_values(by='index')[['index', 'axis', 'parent\_index']]

axis\_edges\_df = axis\_edges\_df.rename( {'index': 'to', 'parent\_index': 'from'}, axis = 1 )

axis\_edges\_df = axis\_edges\_df[~axis\_edges\_df['from'].isna()]

axis\_edges\_df = axis\_edges\_df[~axis\_edges\_df['to'].isna()]

axis\_edges\_df = axis\_edges\_df.astype('int')

existed\_axis\_edges\_df = all\_edges\_df[all\_edges\_df['axis'] == axis]

existed\_axis\_edges\_df['is\_existed'] = True

all\_axis\_edges\_df = pd.merge( axis\_edges\_df, existed\_axis\_edges\_df, how='outer', on=['from', 'to', 'axis'] )

new\_axis\_edges\_df = all\_axis\_edges\_df[all\_axis\_edges\_df['is\_existed'].isna()]

new\_axis\_edges\_df = new\_axis\_edges\_df[['from', 'axis', 'to']]

all\_axis\_edges\_df = all\_axis\_edges\_df[['from', 'axis', 'to']]

edges\_df\_list[axis] = all\_axis\_edges\_df

new\_edges\_df = pd.concat([new\_edges\_df, new\_axis\_edges\_df])

for axis\_n in range(len(edges\_df\_list)):

edges\_df\_list[axis\_n] = edges\_df\_list[axis\_n].drop\_duplicates(subset = ['axis', 'from', 'to'])

for axis\_n in range(len(nodes\_df\_list)):

nodes\_df\_list[axis\_n] = nodes\_df\_list[axis\_n].drop\_duplicates(subset = ['axis', 'index'])

Данный код реализует построение и обновление иерархической модели данных на Python с использованием библиотеки pandas, SQL-запросов и структуры «узлы–рёбра» для аналитических осей. Рассмотрим подробнее данный код.

Этот код — ядро OLAP-сборщика, который:

* Строит дерево иерархий по каждой оси (X, Y, Данные);
* Обеспечивает возможность многомерной агрегации (по уровням и категориям);
* Поддерживает динамическое добавление новых значений;
* Готовит данные к визуализации или агрегации.

Обход по каждой оси (X, Y, Данные)

Листинг 5 – Обход по каждой оси (X, Y, Данные)

for axis in range(len(group\_names)):

Основной цикл по осям, где происходит: подгрузка текущих узлов и рёбер (existed\_axis\_nodes\_df), построение полной иерархии на основе уровней (0–3), расчёт новых узлов и обновление атрибута name, генерация недостающих индексов (ID).

Листинг 6 – Расширение иерархии для каждой оси

for level in range(0, len(group\_columns[axis])):

На каждой итерации: происходит объединение с родительскими уровнями (merge по parent\_id), уровень сдвигается на 1, формируется полная таблица новых и старых узлов (axis\_nodes\_df).

Листинг 7 – Загрузка значений из базы данных (cube\_{cube\_id}\_assemble)

DB.query(f"SELECT \* FROM cube\_{cube\_id}\_data\_assemble LIMIT 1;")

Используется объект DB (обёртка над SQL) для извлечения колонок, необходимых для построения узлов.

Преобразование новых данных: строятся DataFrame-ы с новыми значениями, формируется таблица axis\_nodes\_df для всех уровней, имена месяцев стандартизируются с помощью galileo.month\_dict.

Листинг 8 – Расчёт новых узлов

new\_axis\_df.loc[:,'index'] = list(range(new\_index, new\_index + len(new\_axis\_df)))

Все новые узлы получают уникальные index, узлы объединяются с существующими (axis\_nodes\_df), в итог добавляются в new\_nodes, new\_nodes\_full.

Листинг 9 – Построение рёбер

axis\_edges\_df = axis\_edges\_df.rename({'index': 'to', 'parent\_index': 'from'}, axis=1)

Каждому узлу ищется родитель по иерархии, формируется таблица рёбер (edges\_df\_list), добавляются только новые связи (new\_edges\_df).

Листинг 10 – Очистка от дубликатов

edges\_df\_list[axis\_n] = edges\_df\_list[axis\_n].drop\_duplicates(subset=['axis', 'from', 'to'])

nodes\_df\_list[axis\_n] = nodes\_df\_list[axis\_n].drop\_duplicates(subset=['axis', 'index'])

Удаляются повторяющиеся узлы и рёбра для чистоты структуры.

Так же для построения куба понадобится таблица с составленными узлов вершин (Таблица 7).

Таблица 7 – Узлы вершин

|  |  |  |
| --- | --- | --- |
| from | to | axis |
| 4 | 1 | 0 |
| 4 | 2 | 0 |
| 4 | 3 | 0 |
| 9 | 4 | 0 |
| 8 | 5 | 0 |
| 8 | 6 | 0 |
| 8 | 7 | 0 |
| 9 | 8 | 0 |
| 0 | 9 | 0 |
| 8 | 10 | 0 |
| 4 | 1 | 1 |
| 4 | 2 | 1 |
| 4 | 3 | 1 |
| 9 | 4 | 1 |
| 8 | 5 | 1 |
| 8 | 6 | 1 |
| 8 | 7 | 1 |
| 9 | 8 | 1 |
| 0 | 9 | 1 |
| 8 | 10 | 1 |
| 0 | 1 | 2 |

Все таблицы необходимо загрузить в базу данных для последующего использования при сборке.

На основе составленных таблиц с индексами, необходимо заменить первоначально загруженные данные на эти индексы, этот шаг необходим для создания составного индекса координаты после агрегаций.

Используя язык программирования Python и составленной структуры куба, составим SQL запрос для замены значений на индексы.

Листинг 11 – SQL-запрос для замены значений на индексы

CREATE UNLOGGED TABLE IF NOT EXISTS cube\_120\_data\_assemble\_index AS

(

SELECT distinct on(

cube\_120\_data\_assemble."x\_1", cube\_120\_data\_assemble."x\_2", cube\_120\_data\_assemble."x\_3",

cube\_120\_data\_assemble."y\_1", cube\_120\_data\_assemble."y\_2", cube\_120\_data\_assemble."y\_3"

)

cube\_120\_data\_assemble."Значения",

n0.index AS "x\_1",

n1.index AS "x\_2",

n2.index AS "x\_3",

n3.index AS "y\_1",

n4.index AS "y\_2",

n5.index AS "y\_3"

FROM cube\_120\_data\_assemble

LEFT OUTER JOIN cube\_120\_nodes\_axis\_0 n2 USING( "x\_1", "x\_2", "x\_3" )

LEFT OUTER JOIN cube\_120\_nodes\_axis\_0 n1 USING( "x\_1", "x\_2" )

LEFT OUTER JOIN cube\_120\_nodes\_axis\_0 n0 USING( "x\_1" )

LEFT OUTER JOIN cube\_120\_nodes\_axis\_1 n5 USING( "y\_1", "y\_2", "y\_3" )

LEFT OUTER JOIN cube\_120\_nodes\_axis\_1 n4 USING( "y\_1", "y\_2" )

LEFT OUTER JOIN cube\_120\_nodes\_axis\_1 n3 USING( "y\_1" )

WHERE ( cube\_120\_data\_assemble."x\_1"= n0."name" and n0."x\_2" is NULL and n0."x\_3" is NULL ) and

( cube\_120\_data\_assemble."x\_2"= n1."name" and n1."x\_3" is NULL ) and

( cube\_120\_data\_assemble."x\_3"= n2."name" ) and

( cube\_120\_data\_assemble."y\_1"= n3."name" and n3."y\_2" is NULL and n3."y\_3" is NULL ) and

( cube\_120\_data\_assemble."y\_2"= n4."name" and n4."y\_3" is NULL ) and

( cube\_120\_data\_assemble."y\_3"= n5."name" )

);

Главные таблицы при составление данного SQL-запроса являются cube\_120\_nodes\_axis\_0 и cube\_120\_nodes\_axis\_1, в них хранятся все комбинации каждой из граней куба. Эти данный сформировали на этапе составления уникальных индексов, перенеся в базу данных.

Разберем основную часть скрипта.

Листинг 12 – Происходит последовательное соединение с таблицами узлов

LEFT OUTER JOIN cube\_120\_nodes\_axis\_0 n2 USING("x\_1", "x\_2", "x\_3")

LEFT OUTER JOIN cube\_120\_nodes\_axis\_0 n1 USING("x\_1", "x\_2")

LEFT OUTER JOIN cube\_120\_nodes\_axis\_0 n0 USING("x\_1")

...

n0, n1, n2 — соответствуют уровням иерархии X: x\_1, x\_2, x\_3;

n3, n4, n5 — соответствуют уровням иерархии Y: y\_1, y\_2, y\_3.

Листинг 13 – Условия на точное соответствие уровней

WHERE

cube\_120\_data\_assemble."x\_1" = n0."name" AND n0."x\_2" IS NULL AND n0."x\_3" IS NULL

AND cube\_120\_data\_assemble."x\_2" = n1."name" AND n1."x\_3" IS NULL

AND cube\_120\_data\_assemble."x\_3" = n2."name"

...

Здесь происходит жёсткая фильтрация:

Чтобы избежать совпадений с одноименными значениями из других уровней.

Убедиться, что n0 — действительно уровень 1 (в нём нет x\_2, x\_3);

n1 — уровень 2 (x\_3 is NULL);

n2 — уровень 3.

То же самое делается для y\_1, y\_2, y\_3.

1. **Разработка системы с использованием технологии OLAP** 
   1. **Выбор и обоснование средств для разработки**

Выбор и обоснование средств для разработки OLAP-системы агрегирования данных.

В рамках проектирования и реализации системы агрегирования данных для финансовой аналитики с использованием технологии OLAP требуется тщательный выбор технологического стека, соответствующего как функциональным, так и нефункциональным требованиям: высокой скорости обработки, масштабируемости, отказоустойчивости и удобства разработки.

Язык программирования: Python.

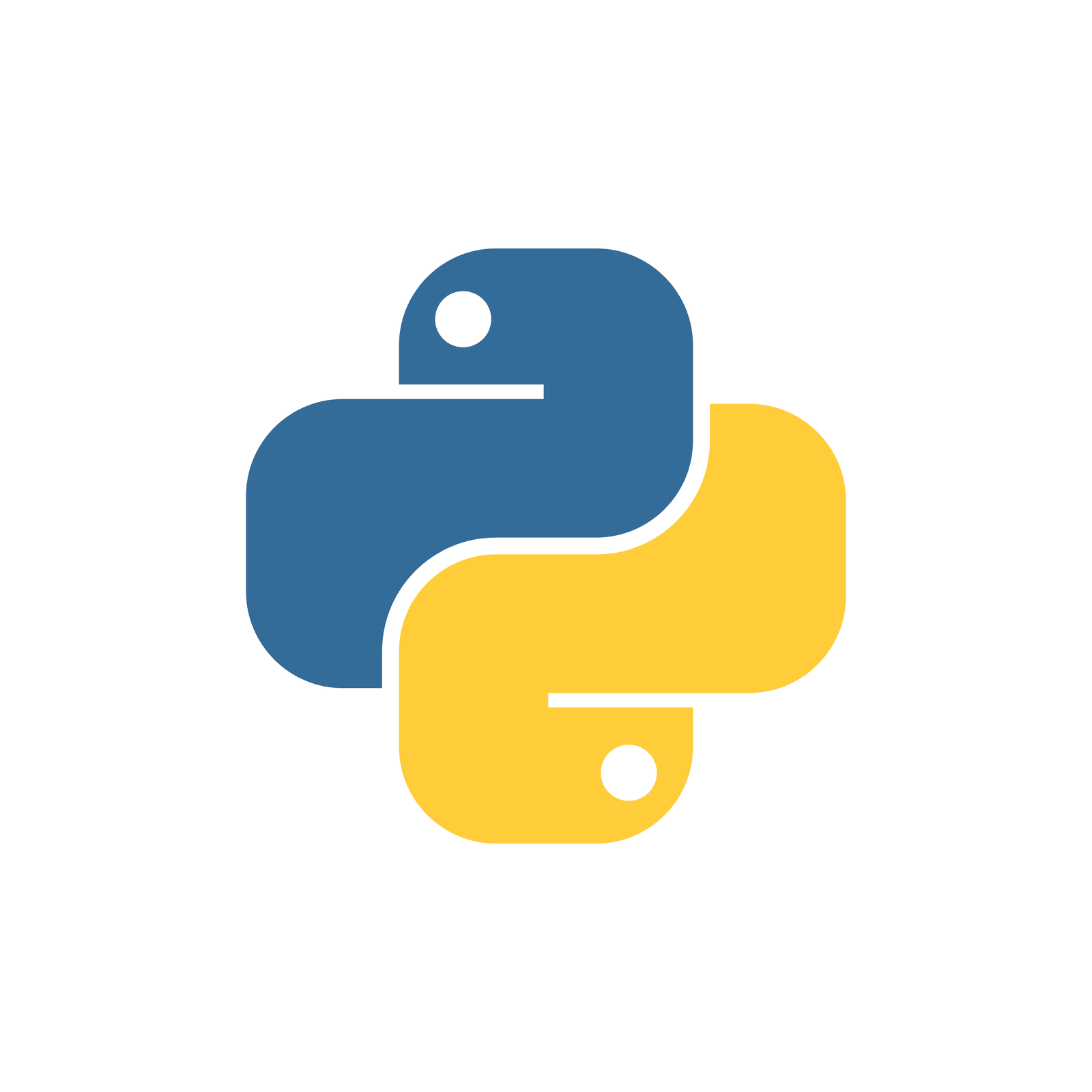


Рисунок 6 – Python

Основным языком разработки серверной части выбран Python, что обосновано следующими аргументами:

* Популярность и поддержка: Python является одним из наиболее популярных языков, активно используемым в аналитике данных, web-разработке и автоматизации.
* Богатая экосистема: наличие библиотек pandas, sqlalchemy, asyncpg, aiohttp, numpy, pydantic, uvicorn и других существенно ускоряет разработку.
* Асинхронность: поддержка асинхронного программирования (async/await) обеспечивает высокую производительность при обслуживании большого количества одновременных запросов.
* Совместимость с OLAP-хранилищами: Python имеет библиотеки и драйверы для работы с ClickHouse, PostgreSQL, Redis и др.

Таким образом, Python обеспечивает гибкость, скорость разработки и широкую интеграционную совместимость.

Серверный фреймворк: FastAPI.



Рисунок 7 – FastAPI

Для реализации API выбран фреймворк FastAPI. Его основные преимущества:

* Высокая производительность — FastAPI построен на Starlette и Pydantic, работает на uvicorn и обеспечивает производительность на уровне Node.js и Go.
* Поддержка OpenAPI и автоматической документации — Swagger UI и ReDoc доступны «из коробки».
* Валидация входных данных — строгая типизация через pydantic, которая гарантирует достоверность запроса на входе.
* Асинхронная архитектура — все маршруты можно писать с async def, эффективно обрабатывая множество параллельных соединений.

Листинг 14 – Пример реализации фреймворка FastAPI

from fastapi import FastAPI

from pydantic import BaseModel

app = FastAPI()

class QueryRequest(BaseModel):

axis: int

metric: str

@app.post("/aggregate")

async def aggregate\_data(query: QueryRequest):

result = await get\_olap\_result(query)

return result

FastAPI идеально сочетается с Python и ClickHouse благодаря легкости настройки, быстрому старту и возможности масштабирования через Gunicorn/Uvicorn.

База данных: ClickHouse.



Рисунок 8 – ClickHouse

В качестве основного хранилища данных для OLAP-аналитики выбран ClickHouse, так как он:

* Поддерживает колоночную структуру, что делает его идеально подходящим для агрегационных запросов;
* Реализует сжатие данных и индексирование, снижая объем хранения и ускоряя чтение;
* Масштабируется горизонтально, позволяя организовывать отказоустойчивые кластеры (репликация и шардинг);
* Интегрируется с Pandas и SQLAlchemy, имеет Python-драйвер clickhouse-connect;
* Поддерживает работу с миллиардами строк данных с откликом в миллисекунды.

Листинг 15 – Пример запроса на агрегирование данных

SELECT toStartOfMonth(date) as month, region, SUM(premium)

FROM financial\_data

GROUP BY month, region

ORDER BY month

ClickHouse позволяет реализовать ROLAP-архитектуру, не прибегая к построению кубов в физическом виде.

Клиентская часть (Рисунок 6): React.

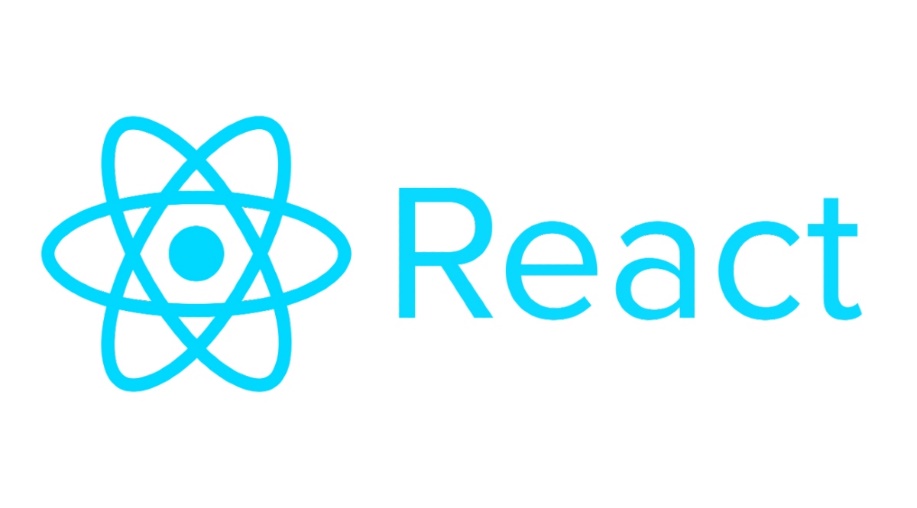


Рисунок 9 – React

Для создания пользовательского интерфейса (клиентской части) выбран React по следующим причинам:

* Компонентный подход — интерфейс строится из переиспользуемых блоков;
* Виртуальный DOM — высокая производительность при отрисовке и изменении данных;
* Широкая поддержка библиотек — для графиков (recharts, chart.js, highcharts), запросов (axios, swr) и маршрутизации (react-router-dom).
* Интеграция с REST/GraphQL API — простая и гибкая интеграция с FastAPI-бэкендом.

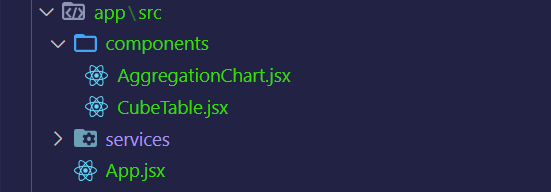


Рисунок 10 – Структура клиента

Интерфейс отображает OLAP-срезы, дашборды, поддерживает фильтрацию и экспорт данных.

Архитектура проекта (Рисунок 7)

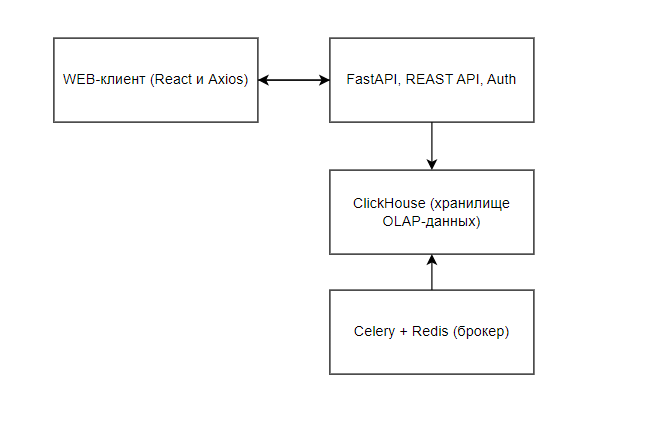


Рисунок 11 – Архитектура проекта

Резервное копирование и отказоустойчивость.

Для обеспечения сохранности данных и надежной работы системы применяются:

* ClickHouse BACKUP/RESTORE — нативные механизмы резервного копирования таблиц и баз данных.
* Ежедневный cron-скрипт, который запускает экспорт данных в S3 или локальное хранилище:
* clickhouse-backup create cube\_backup
* clickhouse-backup upload cube\_backup
* Репликация ClickHouse — используется в кластерах (ZooKeeper / ClickHouse Keeper) для обеспечения отказоустойчивости.

Брокер задач и кэширование.



Рисунок 12 – Celery

Для обработки тяжелых операций (например, пересборки куба, импорта данных, пересчёта агрегатов) используется Celery с брокером Redis:

* Celery — позволяет запускать фоновые задачи, независимо от запроса пользователя;
* Redis — используется как брокер и кэш для хранения промежуточных результатов, для кеширования популярных запросов (например, top N аналитика).

Листинг 15 – Пример задачи Celery

@celery.task

def aggregate\_metrics\_task(cube\_id):

result = run\_clickhouse\_query(cube\_id)

cache.set(f"cube\_{cube\_id}\_aggregate", result)

Выбранный стек технологий — Python + FastAPI + ClickHouse + React + Celery + Redis — обеспечивает:

* высокую производительность аналитики (OLAP);
* масштабируемую серверную архитектуру;
* удобный клиентский интерфейс;
* безопасность и резервирование.



Рисунок 13 – Redis

Это позволяет разрабатывать современную OLAP-систему, способную обслуживать как интерактивные запросы, так и сложные многомерные вычисления в условиях реального бизнеса.

* 1. **Проектирование и разработка архитектуры, структурной и функциональной схемы системы**

Система агрегирования данных для финансовой аналитики построена по трёхуровневой архитектуре.

1. Клиентский уровень (frontend) — отвечает за взаимодействие пользователя с системой через браузер.
2. Серверный уровень (backend) — реализует бизнес-логику, обработку запросов и агрегирование данных.
3. Уровень хранения и вычислений — база данных, OLAP-движок, брокер задач, кэш, и механизмы резервного копирования.

Компоненты архитектуры и их назначение.

React-приложение (Клиент).

Построено на фреймворке **React**. Состоит из модулей: таблица срезов (CubeTable), графики (AggregationChart), фильтры и формы (FiltersPanel). Обращается к API через Axios. Получает агрегации, отображает визуализации, передаёт параметры фильтрации.

FastAPI (REST API-сервер)

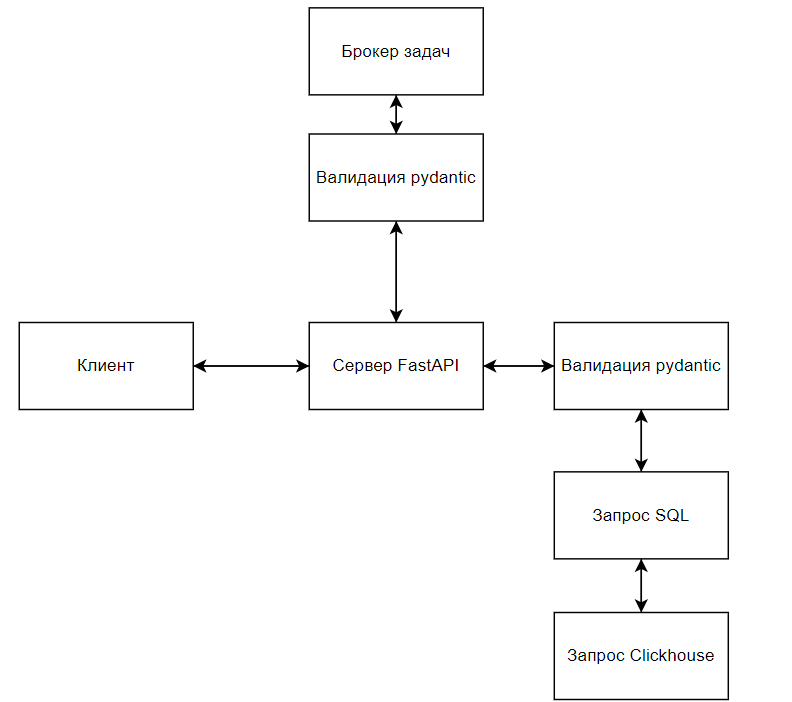


Рисунок 13 – Redis

Описание REST API-сервера:

1. Принимает запросы от клиента.
2. Валидирует параметры с помощью pydantic.
3. Формирует запросы к ClickHouse (через драйвер или SQL).
4. Инициирует отложенные задачи через Celery.
5. Возвращает результат клиенту.

Celery + Redis (Очереди задач и кэширование)

1. Celery отвечает за выполнение фоновых задач (агрегация куба, пересчёт метрик).
2. Redis используется в двух ролях: как брокер задач (Celery backend), как in-memory кэш популярных запросов и промежуточных результатов.

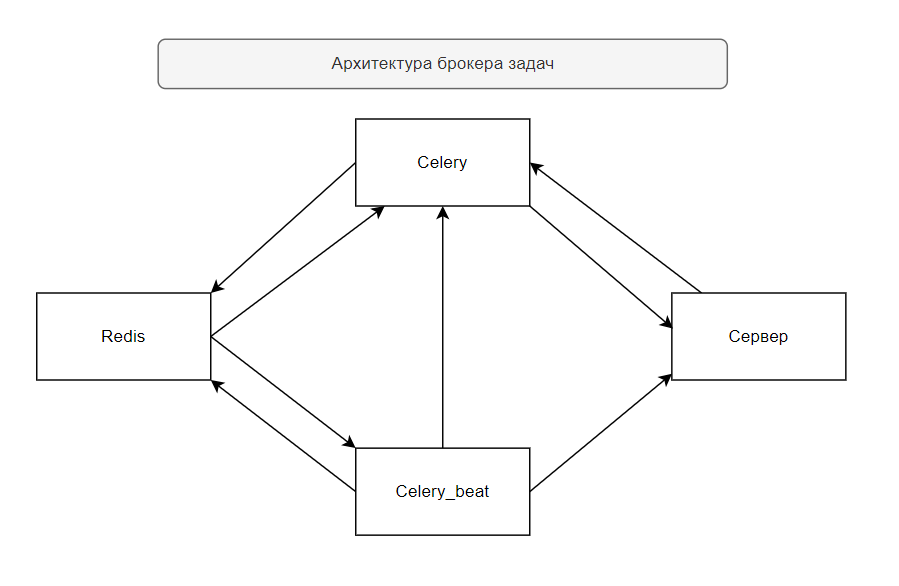


Рисунок 13 – Redis

Описание архитектуры брокера задач: Celery запускается в двух экземпляров, первый экземпляр сам Celery является основным, поскольку все задачи в фоновом режиме будет выполнять он, если серверу понадобиться поставить задачу в фоновый процесс, то он передаст задачу ему. Так же при инициализации Celery хранит в себе подключения в базе данных, Redis данные подключения хранить не умеет, поэтому celery всегда имеет активное подключение к базе данных. Celery\_beat необходим для регулярных фоновых задач, к примеру, если каждый ночью необходимо выполнять сборку куба или выгрузку данных из каких-либо ресурсов, то в назначенное время выполнять будет beat. Redis необходим для хранения информации переданной с сервера, если Celery выполняет какую либо задачу в фоновом процессе, то Redis будет хранить переданную информацию с сервера пока брокер не освободится.

Листинг 16 – Пример регулярной фоновой задачи

@celery.task(name='regular\_search\_from\_hh\_vacancy', bind=True, rate\_limit='1/m')

@after\_4\_schedule

@galileo\_task(name='regular\_search\_from\_hh\_vacancy', log=True, single=True)

def regular\_search\_from\_hh\_vacancy(self):

current\_user = galileo.admin.get\_user(user\_id=0, DB=DB)

search\_urls\_hh\_vacancy = galileo.hr.recruiting.get\_candidates\_from\_hh( DB=DB, current\_user=current\_user)

return search\_urls\_hh\_vacancy

Листинг 16 – Пример регулярных функций, запускающиеся каждые n-секунд

celery = Celery(

'background',

broker=config.message\_queue,

backend=config.message\_queue,

include=['background.tasks']

)

celery.conf.update(

task\_serializer='json',

result\_serializer='json',

result\_backend='redis',

)

celery.conf.beat\_schedule = {

'start\_tasks': {

'task': 'check',

'schedule': 60

},

regular\_search\_from\_hh\_vacancy: {

'task': 'regular\_search\_from\_hh\_vacancy',

'schedule': 60\*60\*20

}

}

Запуск Celery:

1. Запустить Redis (необходимо запустить файл формата.exe)
2. Запустить сам celery - celery.py

Создание задачи:

1. Создать функцию в папке с задачами следующего вида:

Листинг 16 – Создание регулярной задачи

@celery.task(name='название модуля', bind=True, rate\_limit='5/s')

@galileo\_task(name='название модуля', log=True, single=True)

def name\_function(self, ... use\_celery=False:

galileo.test.test( ..., use\_celery=False )

Параметр rate\_limit - частота вызова;

Параметр single в декораторе, «True» значит, что можно запустить только одну функцию, а все такие же будут вставать в очередь;

В функцию передаете все параметры, которые будут использоваться, кроме подключения к БД и экземпляров;

В init.py делаем импорт всего нашего ФАЙЛА;

Подключение к БД передаем внутрь функции celery и точно так же созданный экземпляр пользователя, созданы через декоратора «current\_user».

1. Переходим к редактированию функции которую ставим в очередь, нужно добавить аргумент «use\_celery =False» (при вызове, например, в роутах ставите «True»):

Внутри функции делаем следующую проверку.

Листинг 16 – Проверка функции внутри celery

if use\_celery:

from background import tasks

qwe =tasks.название функции.delay( ... , use\_celery=False)

return {

'message': 'Задача создана',

'recipients': [],

'is\_send': True,

'traceback\_str': None, 'qwe':qwe

}

Когда celery вызывет вашу функцию то аргумент use\_celery будет False

если задача может прерваться, при перезапуске celery, то ее нужно сбрасывать , иначе она встанет в бесконечную очередь.

нужно тогда добавить задачу в автоматический сброс, заходим в background\_run.py и в «tasks\_names =[....]». добавляем название вашей задачи.

ClickHouse (OLAP-хранилище)

1. Основная БД для хранения агрегируемых данных.
2. Использует колоночное хранение и индексирование.
3. Хранит: иерархии узлов (nodes), связи (edges), сырые данные (raw\_data), агрегаты (assembled).
4. Обрабатывает аналитические запросы по оси времени, метрикам, категориям.

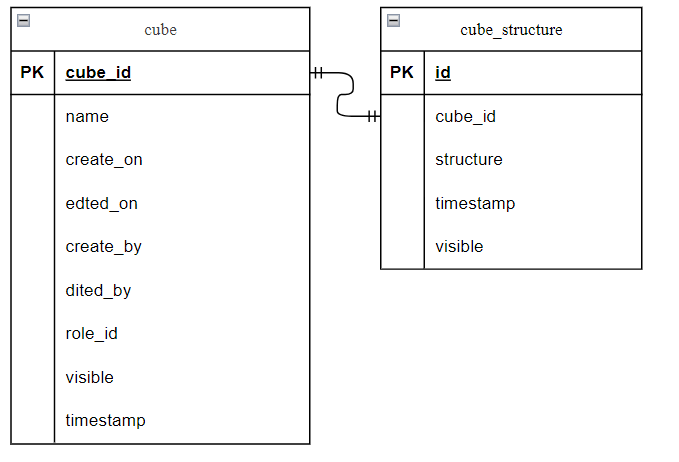


Рисунок 13 – Таблицы для описания куба

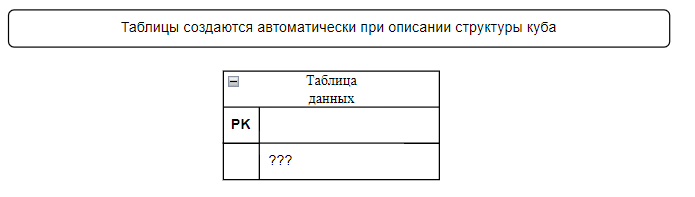


Рисунок 13 – Таблица с основными данными куба

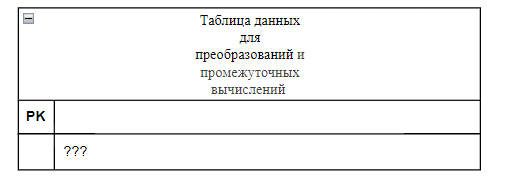


Рисунок 13 – Промежуточная таблица

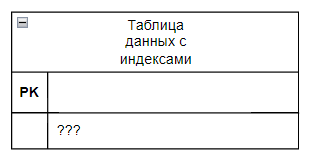


Рисунок 13 – Таблицы с индексами

Механизм резервного копирования:

1. Автоматически создаёт резервные копии по расписанию (через clickhouse-backup).
2. Сохраняет в облако (например, S3) или локальный NAS.
3. Позволяет восстановить OLAP-хранилище в случае отказа.

Клиентская зона системы.

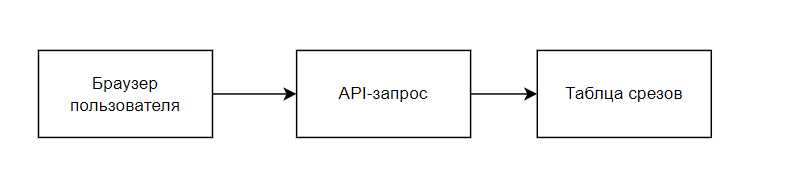


Рисунок 13 – Клиентская зона

1. Браузер пользователя.
2. API-запрос (Axios).
3. Таблица срезов.

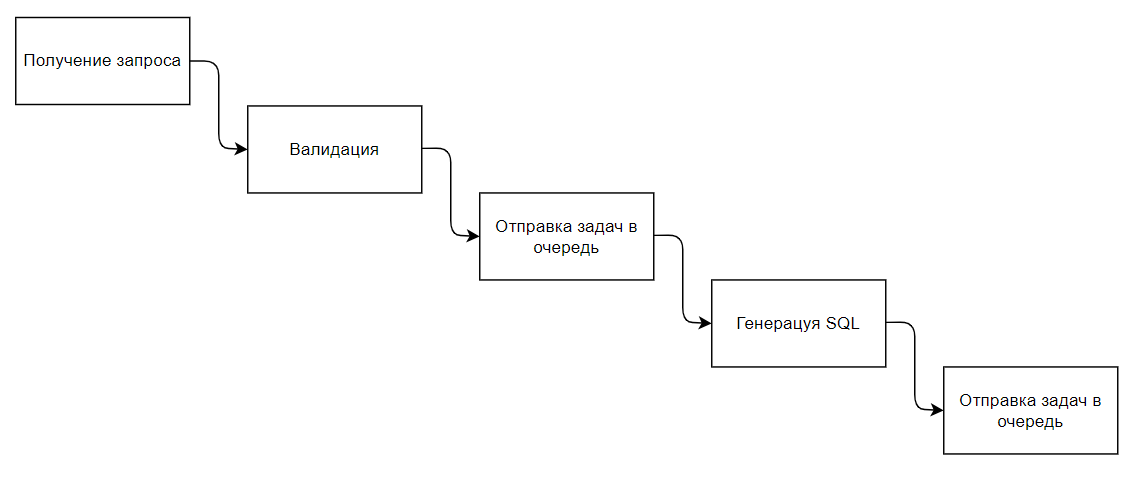


Рисунок 13 – Серверная зона

1. Получение запроса.
2. Валидация.
3. Генерация SQL.
4. Отправка задачи в очередь.
5. Ответ клиенту.

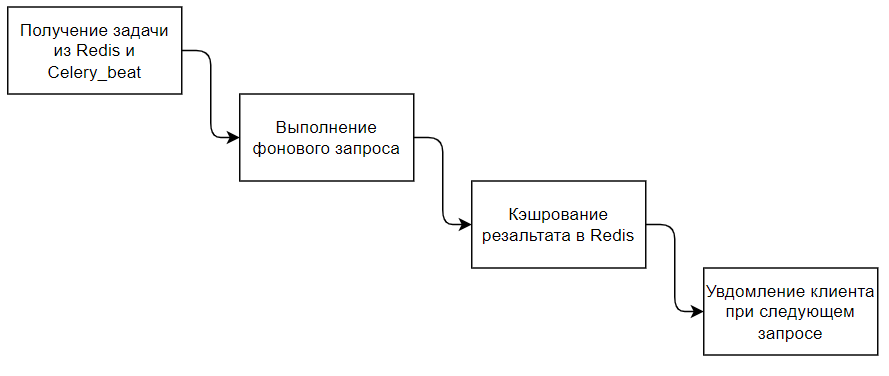


Рисунок 13 – Обработка задач

1. Получение задачи из Redis и Celery\_beat.
2. Выполнение фонового запроса.
3. Кэширование результата в Redis.
4. Уведомление клиента через WebSocket или при следующем запросе.

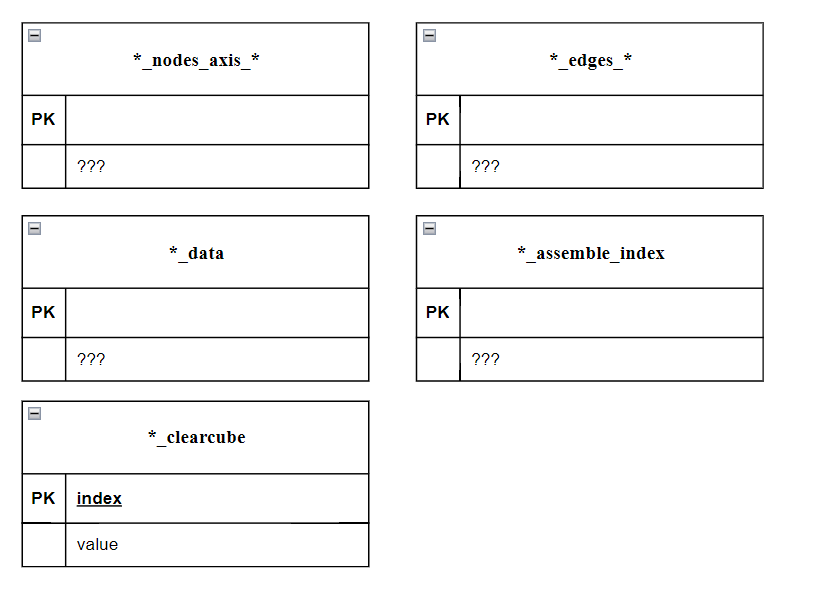


Рисунок 13 – Хранилище данных

Таблицы:

1. \*\_nodes\_axis\_\* — иерархии по измерениям
2. \*\_edges\_\* — связи между уровнями
3. \*\_data — исходные значения
4. \*\_assemble\_index — предагрегированные ключи
5. \*\_result — агрегированные метрики

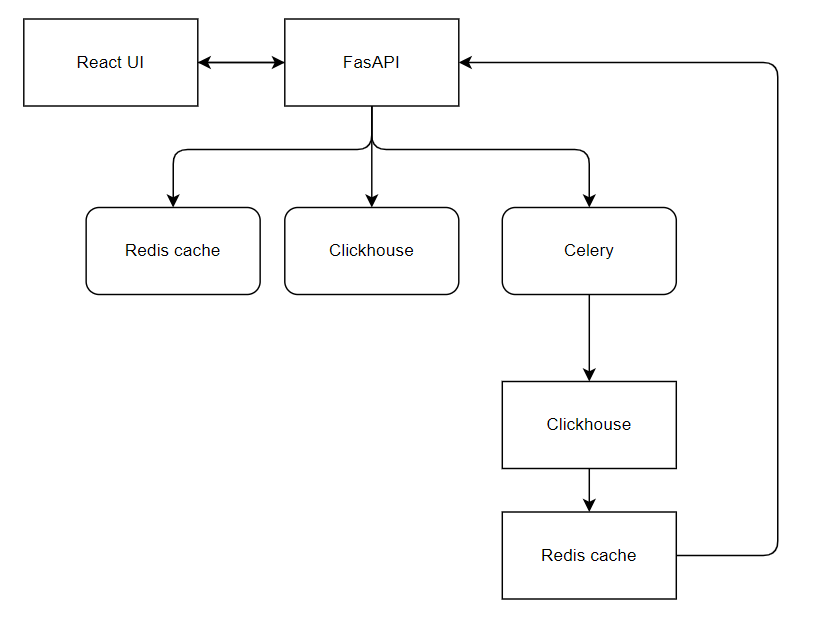


Рисунок 13 – Потоки данных

Разработанная архитектура обеспечивает:

* гибкость (все компоненты изолированы и масштабируемы);
* скорость (использование Redis и ClickHouse);
* надёжность (резервное копирование и фоновые задачи);
* удобство разработки (весь стек построен на современных open-source технологиях).
  1. **Разработка серверной части системы**

Серверная часть системы агрегирования данных реализована с использованием языка программирования Python, веб-фреймворка FastAPI, системы брокеров задач Celery, кэширующего сервиса Redis, аналитической СУБД ClickHouse, а также инструментов мониторинга и журналирования. Архитектура построена по микросервисному принципу, обеспечивая масштабируемость, отказоустойчивость и гибкость в поддержке и развитии системы.

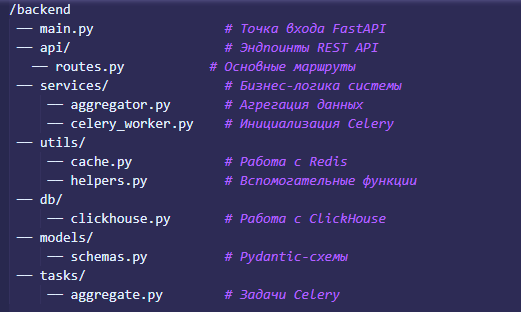


Рисунок 13 – Структура каталогов проекта

* + 1. **Работа Celery и Redis.**

Celery используется для выполнения задач в фоне, не блокируя основной поток API. Redis выполняет роль брокера задач и временного кэша.

Листинг 16 – Задача

# tasks/aggregate.py

@celery\_app.task(bind=True)

def recalculate\_aggregate(self, query\_dict):

query = QueryRequest(\*\*query\_dict)

result = aggregate\_data(query)

redis.set(query.cache\_key(), json.dumps(result), ex=86400)

return result

Листинг 16 – Задача

# celery\_worker.py

celery\_app = Celery(\_\_name\_\_, broker="redis://localhost:6379/0")

celery\_app.conf.task\_routes = {"tasks.\*": {"queue": "aggregation"}}

Листинг 16 – REST-маршрут

# Получение агрегированных данных

@router.post("/aggregate")

async def aggregate\_data(query: QueryRequest):

cached = redis.get(query.cache\_key())

if cached:

return JSONResponse(content=json.loads(cached))

result = aggregator.aggregate(query)

redis.set(query.cache\_key(), json.dumps(result), ex=3600)

return result

# Отправка задачи в Celery

@router.post("/aggregate/background")

async def run\_async\_aggregation(query: QueryRequest):

task = recalculate\_aggregate.delay(query.dict())

return {"task\_id": task.id, "status": "queued"}

* + 1. **Архитектура кэширования**

Кэширование осуществляется на двух уровнях, Redis — кэширует готовые агрегаты по ключу, формируемому из параметров запроса, Локальный словарь (LruCache) — используется для краткосрочного хранения промежуточных данных в процессе агрегации.

Листинг 16 – Код кэширования

def get\_cache\_key(query):

return f"agg:{query.axis}:{query.metric}:{hash(str(query.filters))}"

Особенности реализации, все ключи снабжены TTL (время жизни).

Кэш очищается при обновлении данных вручную или автоматически по расписанию, кэширование существенно снижает нагрузку на ClickHouse при повторных обращениях.

Данные подходы позволяют достичь высокой производительности системы, а модульная структура — облегчить сопровождение и внедрение новых функций. Эта архитектура лежит в основе всего серверного взаимодействия и является важнейшим элементом успешной реализации аналитической платформы.

Серверная часть системы агрегирования данных была разработана на языке **Python** с использованием фреймворка **FastAPI**. Это решение обеспечило асинхронную обработку запросов, автоматическую генерацию OpenAPI-спецификаций и быструю разработку REST API. Ниже приведены принципы архитектуры, структура проекта, примеры кода и схемы.

Архитектурная структура серверной части.

Серверный слой реализован по микросервисному принципу и включает в себя:

FastAPI REST API — основной сервер, принимающий HTTP-запросы;

ClickHouse Adapter — модуль работы с аналитической СУБД;

Redis Cache — хранение кэшированных ответов;

Celery Tasks — фоновая обработка задач агрегации и расчёта;

Мониторинг и логирование — Prometheus + Grafana и логгер на основе loguru.

* + 1. **Разработка класса для составления запросов**

Для создания основного класса (Рисунок !!!! ) по формированию SQL запросов для формирования куба, понадобятся следующие атрибуты.

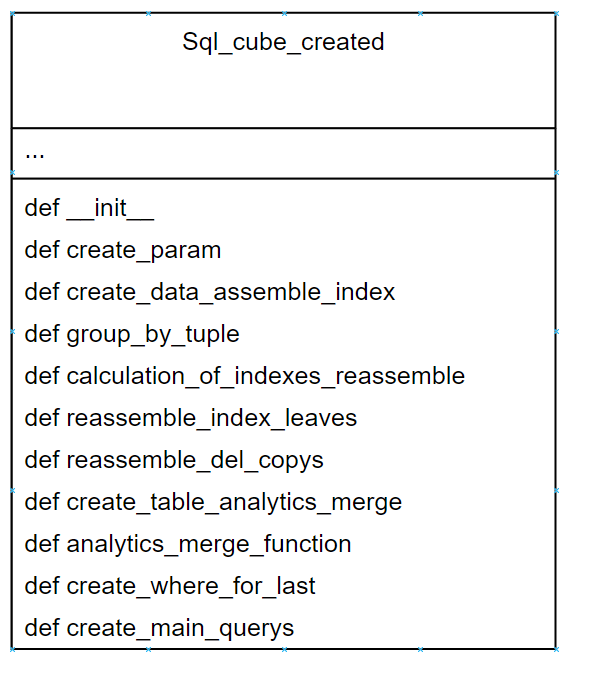


Рисунок 13 – Структура класса

Рассмотрим атрибуты:

engine – содержит в себе подключение к базе данных.

user\_id – id пользователя;

cube\_id – id куба;

group\_names – Список граней куба;

group\_columns – Список измерений куба.

Листинг 16 – Удаляем двойные разные кавычки

self.group\_names=[[j.replace('"','') for j in i] for i in self.group\_names]

Так как формат колонок может быть набран кириллицей, необходимо изменить кавычки на двойные, чтобы в SQL-запросе можно было вставлять столбцы.

Листинг 16 – Удаляем "Столбец данных"

self.group\_names=self.group\_names[:-1]

Листинг 16 – Удаляем двойные разные кавычки

self.group\_columns=[i.replace('"','') for i in self.group\_columns]

Листинг 16 – Список со всеми столбцами 2Данные»

self.group\_data\_columns = [ i['name'] for i in self.structure['data\_columns']+self.structure['data\_columns\_operations']+self.structure['data\_columns\_preoperations'] ]

Листинг 16 – Список всех столбцов куба

self.group\_names\_join\_nodes = [ 'n'+str(xx) for xx in range(len(self.group\_columns)) ]

Листинг 16 – Название аналитиков для join

self.group\_lenghts = [ int(ax['id\_length']) for ax in self.structure['axis'] ]

Листинг 16 – Все оригинальные столбцы с Данными

self.dict\_name\_col={ i['name']:i['name'] for i in galileo.cubes.get\_structure(engine=self.engine, сube\_id=self.cube\_id)['data']['data\_columns']}

Листинг 16 – Словарь «{'Столбец данных':2 }» для того чтобы сделать запрос в таблицу «cube\_{self.cube\_id}\_nodes\_2»

self.dict\_name\_col\_and\_index={ i['columns'][0]:i['id'] for i in galileo.cubes.get\_structure(engine=self.engine, cube\_id=self.cube\_id)['data']['axis'] if 'Столбец данных' in i['columns']}

Столбец данных – является зарезервированным словом в коде, его невозможно заполнить в структуре куба, так как он необходим для сборки и пользователю выдаст ошибку если он захочет ввести данное название в структуру.

Листинг 16 – Составление запроса

radical\_column= ''

for i in self.group\_data\_columns: radical\_column += f'COALESCE("{i}", 0 ) + '

Один из важнейших запросов. Функция COALESCE возвращает первое непустое значение из списка аргументов. Если все аргументы пустые или равны NULL, функция возвращает NULL. Она используется для замены пустых значений или NULL на другие значения, обычно по умолчанию.

Как работает COALESCE:

1. Поочередно оценивает аргументы, переданные в функцию.
2. Возвращает первое непустое значение (не NULL) среди аргументов.
3. Если все аргументы пустые или NULL, возвращает NULL.
   1. **Разработка клиентской части системы**

Клиентская часть реализована с использованием React и библиотеки визуализации Recharts. Её основное предназначение — предоставление интуитивно понятного пользовательского интерфейса для настройки и отображения агрегированных данных в OLAP-структуре.

* 1. **Пример использования**
  2. **Оценка эффективности**

1. **Введение, заключение, список источников, приложения**

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Вся правда о кубах данных OLAP: развенчиваем мифы: [Электронный ресурс] — URL: https://habr.com/ru/companies/vk/articles/703508/ (дата обращения: 15.01.2023).

2. OLAP-куб: [Электронный ресурс] — URL: https://ru.wikipedia.org/wiki/OLAP-%D0%BA%D1%83%D0%B1 (дата обращения: 15.01.2023).

3. Знакомство с OLAP: что это и как работает: [Электронный ресурс] — URL: https://skillbox.ru/media/code/znakomstvo-s-olap-chto-eto-i-kak-rabotaet/ (дата обращения: 15.01.2023).

4. Обзор кубов OLAP Service Manager для расширенной аналитики: [Электронный ресурс] — URL: https://learn.microsoft.com/ru-ru/system-center/scsm/olap-cubes-overview?view=sc-sm-2025 (дата обращения: 15.01.2023).

5. OLAP-кубы: вечная классика: [Электронный ресурс] — URL: https://bars-alpha.bi/olap-cubes (дата обращения: 15.01.2023).

6. Oracle Business Intelligence: [Электронный ресурс] — URL: https://www.oracle.com/business-analytics/business-intelligence/technologies/bi.html (дата обращения: 15.01.2023).

7. Oracle OLAP: [Электронный ресурс] — URL: https://download.oracle.com/opndocs/emea/Oracle\_11g\_OLAP.pdf (дата обращения: 15.01.2023).

8. Что такое службы анализа Analysis Services?: [Электронный ресурс] — URL: https://learn.microsoft.com/ru-ru/analysis-services/analysis-services-overview?view=asallproducts-allversions (дата обращения: 15.01.2023).

9. Развитие SAP Business Warehouse: [Электронный ресурс] — URL: https://www.sap.com/central-asia-caucasus/products/data-cloud/sap-bw-migration.html (дата обращения: 15.01.2023).

10. Python: [Электронный ресурс] — URL: https://www.python.org/ (дата обращения: 15.01.2023).