Оптимизация DataLakeHouse

Чтобы я делал в первую очередь:

1. Партиционирование - представьте файловую структуру хранения, пусть данные сгруппированы по каталогам, названия которых это значения из некоторого столбца таблицы, а файлы внутри каталога - это данные соответствующие этому значению в таблице

Когда таблица не партиционирована, то чтобы сделать поиск по значению, или сгруппировать данные, или выполнить соединение двух наборов данных, необходимо полностью прочитать таблицу (то есть прочитать все файлы) и перемешать по соответствующим полям, использующимся для агрегации - это эквивалентно передаче данных между узлами кластера (по сети), что является дорогой операцией, куда быстрее сразу выделить только те наборы данных, которые используются в агрегациях, поэтому таблицу надо партиционировать

Какое поле выбрать для партиционирования?

- Поле часто используется для условий фильтрации, например where
- Поле часто используется для агрегации - group by, или для соединений - join
- Поле должно удовлетворять условиям кардинальности (кол-во уникальных значений для поля):
  - Слишком много уникальных значений -> много маленьких файлов -> плохо
  - Слишком мало уникальных значений -> зависит от запросов к таблице - как правило не даст значительного прироста в эффективности, может быть обоснованно, когда выделяется явный бизнес смысл и поле участвует во всех запросах в качестве фильтрации (например, у нас в X5 для одной таблицы есть партиционирование по поле format_id - {ts5, tsx} - ts5 относится к пятерочкам, tsx к перекресткам, ты не считаешь сквозную статистику, а всегда делаешь запросы к одной из частей данных, поэтому тут это оправдано)
  - Идеальный вариант: 10-100 партиций
- Данные должны быть равномерно распределены по партициям - то есть не должно быть в одной патриции 90% данных, а в другой - 10%, это приведет к перекосу данных и замедлит выполнение работы

Как выбрать поле?

Смотрим какие поля чаще всего используется, в where, group by, join on -> смотрим на кол-во уникальных значений (в идеале 10-100) -> данные распределены равномерно (100mb - 1gb на партицию)

Еще хорошие практики - иерархическое партиционирование

-- Хорошая практика:
PARTITIONED BY (year INT, month INT, day INT)
-- Запрос за конкретный месяц будет читать только 30-31 партицию
WHERE year = 2024 AND month = 1

Дополнительная кластеризация вместе с партиционированием - то есть близкие по значению данные лежат рядом друг с другом -> быстрее поиск и фильтрация

-- Партиционирование по дате + кластеризация внутри партиции
PARTITIONED BY (event_date DATE)
CLUSTERED BY (user_id, timestamp) или CLUSTERED BY (user_id, timestamp) INTO 32 BUCKETS

-- Данные до кластеризации (случайный порядок):
user_id	event_type	timestamp
100	click	2024-01-01
200	view	2024-01-02
100	view	2024-01-01
300	click	2024-01-03

-- После кластеризации по (user_id, timestamp):
100 | click | 2024-01-01
100 | view | 2024-01-01
200 | view | 2024-01-02
300 | click | 2024-01-03

Как выбрать кол-во бакетов?

-- Ориентировочная формула:
target_bucket_size = 100MB - 1GB (идеальный диапазон)
num_buckets = CEIL(total_data_size / target_bucket_size)

При этом малые объемы данных (< 10 GB) - 16-64 бакета
(10Gb - 1Tb) - 64-256 бакетов
(> 1 Tb) - 256-1024+ бакетов

Слишком много бакетов не надо - нагрузка на метастор
Слишком мало бакетов - тоже толку ноль

Лучше использовать степени двойки - более равномерное распределение

2. Денормализация

Денормализация — это намеренное дублирование данных и нарушение нормальных форм БД для повышения производительности запросов.

Когда применять?

Частые join различных таблиц - легче объединить все в одну большую таблицу и работать только с ней

Денормализация, когда запрос содержат 3 и более join на больших таблицах

Примеры:

```
-- БЫЛО: Медленный запрос с JOIN
SELECT
    date_trunc('month', o.order_date) as month,
    c.segment,
    p.category,
    COUNT(*) as order_count,
    SUM(o.amount) as total_revenue
FROM orders o
JOIN customers c ON o.customer_id = c.id
JOIN products p ON o.product_id = p.id
WHERE o.order_date >= '2024-01-01'
GROUP BY 1, 2, 3;

-- СТАЛО: Быстрый запрос к денормализованной таблице
SELECT
    date_trunc('month', order_date) as month,
    customer_segment as segment,
    product_category as category,
    COUNT(*) as order_count,
    SUM(amount) as total_revenue
FROM denormalized_orders
WHERE order_date >= '2024-01-01'
GROUP BY 1, 2, 3;
```

1. Создание материализованных представлений

Материализованное представление (Materialized View) — это предварительно вычисленные результаты запроса, сохраненные как физическая таблица. В отличие от обычного представления (view), которое вычисляется на лету, материализованное представление хранит данные, что значительно ускоряет выполнение запросов.

Примеры

1) **Предварительные вычисления сложных агрегаций**

```
-- Исходный медленный запрос
SELECT
    date_trunc('month', order_date) as month,
    customer_segment,
    product_category,
    COUNT(*) as order_count,
    SUM(amount) as total_revenue,
    AVG(amount) as avg_order_value
FROM orders
WHERE order_date >= '2023-01-01'
GROUP BY 1, 2, 3;

-- Материализованное представление
CREATE MATERIALIZED VIEW monthly_sales_mv AS
SELECT
    date_trunc('month', order_date) as month,
    customer_segment,
    product_category,
    COUNT(*) as order_count,
    SUM(amount) as total_revenue,
    AVG(amount) as avg_order_value,
    MAX(order_date) as last_updated
FROM orders
GROUP BY 1, 2, 3;

-- Ускоренный запрос
SELECT * FROM monthly_sales_mv
WHERE month >= '2023-01-01';
```

2) **Исключение дорогостоящих JOIN**

```
-- Сложный запрос с множественными JOIN
SELECT
    o.order_id,
    o.order_date,
    c.customer_name,
    c.segment,
```

```
    p.product_name,
    p.category,
    s.store_name,
    s.region
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id
JOIN stores s ON o.store_id = s.store_id;

-- Материализованное представление объединяет все данные
CREATE MATERIALIZED VIEW order_details_mv AS
SELECT
    o.*,
    c.customer_name,
    c.segment,
    p.product_name,
    p.category,
    s.store_name,
    s.region
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN products p ON o.product_id = p.product_id
JOIN stores s ON o.store_id = s.store_id;

-- Мгновенный доступ к данным
SELECT * FROM order_details_mv
WHERE segment = 'VIP' AND region = 'North';
```

3) **Оптимизация для конкретных паттернов запросов**

```
-- Частые запросы по временным диапазонам и фильтрам
CREATE MATERIALIZED VIEW sales_dashboard_mv AS
SELECT
    order_date,
    customer_id,
    product_id,
    store_id,
    amount,
    -- Предварительно вычисленные метрики
    SUM(amount) OVER (PARTITION BY customer_id ORDER BY order_date) as
customer_lifetime_value,
    AVG(amount) OVER (PARTITION BY product_id, store_id) as
avg_product_store_sale
FROM orders
WHERE order_date >= current_date - INTERVAL '365' DAY;
```

Стратегии обновления материализованных представлений:

Полное обновление:
-- Простое, но ресурсоемкое
REFRESH MATERIALIZED VIEW sales_mv;

-- Используется когда:
-- • Данные изменяются значительно
-- • Нет механизма инкрементального обновления
-- • Нечастое обновление (раз в день/неделю)

Инкрементальное обновление:
-- Обновление только измененных данных
REFRESH MATERIALIZED VIEW sales_mv
WHERE order_date >= current_date - INTERVAL '1' DAY;

-- Требует механизма отслеживания изменений
-- Эффективно для больших объемов данных

Автоматическое обновление:
-- Настройка периодического обновления
CREATE MATERIALIZED VIEW sales_mv
AUTO REFRESH EVERY '1' HOUR
AS SELECT ...;

-- Или на основе триггеров/событий

Когда стоит выбрать?

-- Частые агрегации по большим наборам данных
CREATE MATERIALIZED VIEW daily_aggregates AS ...

-- Сложные JOIN с множеством таблиц
CREATE MATERIALIZED VIEW joined_data AS ...

-- Регулярные отчеты и дашборды
CREATE MATERIALIZED VIEW dashboard_data AS ...

-- Данные с низкой частотой обновления
CREATE MATERIALIZED VIEW reference_data AS ...

Когда не стоит?

-- Данные, требующие real-time актуальности
-- Часто изменяемые данные с point-обновлениями
-- Редко выполняемые запросы
-- Очень маленькие таблицы