Выполнил: Бойко Илья Группа: ПИН-б-о-22-1

Лабораторная работа №9 (Часть 2): Миграция и анализ большого набора данных с помощью Spark SQL

1. Цель работы Освоить методы миграции больших объемов данных из реляционных баз данных в распределенные системы хранения и выполнения сложного анализа с использованием Spark SQL. Разработать ETL-пайплайн для преобразования и обогащения данных.

2. Используемый стек технологий

Платформы: Apache Spark, PostgreSQL, HDFS Языки программирования: Python (PySpark), SQL Библиотеки: pyspark, psycopg2, pandas ОС: Linux (Ubuntu) Инструменты: Jupyter Notebook, pgAdmin, Spark Web UI

3. Теоретические сведения

Архитектура решения:

1. **Extract:** Извлечение данных из PostgreSQL с использованием JDBC
2. **Transform:** Преобразование и обогащение данных с помощью Spark SQL
3. **Load:** Загрузка результатов в HDFS и аналитические таблицы
4. **Analysis:** Выполнение сложных аналитических запросов

Ключевые аспекты:

- Обработка datasets объемом 10+ GB
- Оптимизация производительности Spark Jobs
- Использование партиционирования и кэширования
- Анализ временных рядов и оконные функции

4. **Ход выполнения работы**

Этап 1: Подготовка данных и инфраструктуры

1.1 Настройка PostgreSQL с Docker

services: postgres: image: postgres:15 container_name: postgres-sales environment: POSTGRES_DB: sales_db POSTGRES_USER: admin POSTGRES_PASSWORD: admin123 PGDATA: /var/lib/postgresql/data/pgdata ports: - "5432:5432" volumes: - postgres_data:/var/lib/postgresql/data -

./init-scripts:/docker-entrypoint-initdb.d healthcheck: test: ["CMD-SHELL", "pg_isready -U admin"] interval: 10s timeout: 5s retries: 5 networks: - sales-network

pgadmin: image: dpage/pgadmin4 container_name: pgadmin environment: PGADMIN_DEFAULT_EMAIL: admin@example.com PGADMIN_DEFAULT_PASSWORD: admin123 ports: - "5050:80" depends_on: postgres: condition: service_healthy networks: - sales-network

volumes: postgres_data:

networks: sales-network: driver: bridge

Создайте директорию и SQL скрипт инициализации:

mkdir -p init-scripts

cat > init-scripts/01-init-schema.sql << 'EOF' -- Инициализация схемы данных для продаж SET statement_timeout = 0; SET lock_timeout = 0; SET idle_in_transaction_session_timeout = 0; SET client_encoding = 'UTF8'; SET standard_conforming_strings = on; SET check_function_bodies = false; SET client_min_messages = warning; SET row_security = off;

-- Создание таблицы продаж с оптимизациями CREATE TABLE IF NOT EXISTS sales (sale_id BIGSERIAL, product_id INTEGER NOT NULL, customer_id INTEGER NOT NULL, sale_date DATE NOT NULL, amount DECIMAL(10,2) NOT NULL, quantity INTEGER NOT NULL, region VARCHAR(50) NOT NULL, category VARCHAR(50) NOT NULL, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

```
-- Ограничения
CONSTRAINT pk_sales PRIMARY KEY (sale_id),
CONSTRAINT chk_amount_positive CHECK (amount >= 0),
CONSTRAINT chk_quantity_positive CHECK (quantity > 0)
```

);

-- Создание индексов для ускорения запросов CREATE INDEX IF NOT EXISTS idx_sales_sale_date ON sales(sale_date); CREATE INDEX IF NOT EXISTS idx_sales_product_id ON sales(product_id); CREATE INDEX IF NOT EXISTS idx_sales_customer_id ON sales(customer_id); CREATE INDEX IF NOT EXISTS idx_sales_region ON sales(region); CREATE INDEX IF NOT EXISTS idx_sales_category ON sales(category); CREATE INDEX IF NOT EXISTS idx_sales_date_region ON sales(sale_date, region); CREATE INDEX IF NOT EXISTS idx_sales_date_category ON sales(sale_date, category);

-- Создание таблицы продуктов CREATE TABLE IF NOT EXISTS products (product_id SERIAL PRIMARY KEY, product_name VARCHAR(255) NOT NULL, price DECIMAL(10,2) NOT NULL, cost DECIMAL(10,2) NOT NULL, category VARCHAR(50) NOT NULL, supplier_id INTEGER, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP);

-- Создание таблицы клиентов CREATE TABLE IF NOT EXISTS customers (customer_id SERIAL PRIMARY KEY, customer_name VARCHAR(255) NOT NULL, email VARCHAR(255), phone

VARCHAR(50), registration_date DATE DEFAULT CURRENT_DATE, region VARCHAR(50), segment VARCHAR(50), created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP);

-- Создание материализованного представления для агрегированных данных CREATE MATERIALIZED VIEW IF NOT EXISTS daily_sales_summary AS SELECT sale_date, region, category, COUNT(*) as total_sales, SUM(amount) as total_revenue, SUM(quantity) as total_quantity, AVG(amount) as avg_sale_amount FROM sales GROUP BY sale_date, region, category WITH DATA;

-- Создание индекса для материализованного представления CREATE UNIQUE INDEX IF NOT EXISTS idx_daily_summary ON daily_sales_summary (sale_date, region, category);

-- Функция для обновления updated_at CREATE OR REPLACE FUNCTION update_updated_at_column() RETURNS TRIGGER AS $$ BEGIN NEW.updated_at = CURRENT_TIMESTAMP; RETURN NEW; END; $$ language 'plpgsql';

-- Триггер для автоматического обновления updated_at CREATE TRIGGER update_sales_updated_at BEFORE UPDATE ON sales FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

-- Создание пользователя для приложения CREATE USER sales_app WITH PASSWORD 'sales_app_password'; GRANT CONNECT ON DATABASE sales_db TO sales_app; GRANT USAGE ON SCHEMA public TO sales_app; GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO sales_app; GRANT USAGE ON ALL SEQUENCES IN SCHEMA public TO sales_app;

-- Разрешение на чтение материализованного представления GRANT SELECT ON daily_sales_summary TO sales_app;

EOF

echo " SQL скрипт инициализации создан"

Запуск PostgreSQL

docker-compose -f docker-compose-postgres.yml up -d

docker ps | grep postgres

docker logs postgres-sales --tail 10

docker exec -it postgres-sales psql -U admin -d sales_db

1.2 Генерация тестовых данных

Создайте Python скрипт для генерации данных:

import psycopg2 from psycopg2.extras import execute_batch import random from datetime import datetime, timedelta import time import logging from typing import List, Tuple import multiprocessing as mp from dataclasses import dataclass import argparse

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s') logger = logging.getLogger(**name**)

@dataclass class SalesRecord: """Модель записи продажи""" sale_id: int product_id: int customer_id: int sale_date: str amount: float quantity: int region: str category: str

class SalesDataGenerator: """Генератор тестовых данных продаж"""

```
# Статические данные
REGIONS = ['North', 'South', 'East', 'West', 'Central', 'International']
CATEGORIES = ['Electronics', 'Clothing', 'Food', 'Books', 'Sports', 'Home', 'Automotive', 'Be

# Ценовые диапазоны по категориям
CATEGORY_PRICE_RANGES = {
    'Electronics': (50, 2000),
    'Clothing': (10, 500),
    'Food': (5, 200),
    'Books': (5, 100),
    'Sports': (20, 800),
    'Home': (15, 1500),
    'Automotive': (100, 5000),
    'Beauty': (5, 300)
}

def __init__(self, total_records: int = 10000000, batch_size: int = 50000):
    self.total_records = total_records
    self.batch_size = batch_size
    self.start_date = datetime(2020, 1, 1)
    self.end_date = datetime(2023, 12, 31)
    self.date_range = (self.end_date - self.start_date).days

    # Предварительно сгенерированные данные для консистентности
    self.products = self._generate_products(1000)
    self.customers = self._generate_customers(100000)

def _generate_products(self, count: int) -> List[Tuple[int, str, float, str]]:
    """Генерация списка продуктов"""
    products = []
    for i in range(1, count + 1):
        category = random.choice(self.CATEGORIES)
        price_range = self.CATEGORY_PRICE_RANGES[category]
        price = round(random.uniform(*price_range), 2)
        cost = round(price * random.uniform(0.3, 0.7), 2)
        products.append((
            i,
            f"Product_{i:06d}",
            price,
            cost,
            category
        ))
    return products
```

```
def _generate_customers(self, count: int) -> List[Tuple[int, str, str]]:
    """Генерация списка клиентов"""
    customers = []
    for i in range(1, count + 1):
        customers.append((
            i,
            f"Customer_{i:06d}",
            random.choice(self.REGIONS)
        ))
    return customers

def generate_batch(self, start_id: int, batch_size: int) -> List[SalesRecord]:
    """Генерация батча записей продаж"""
    records = []

    for i in range(batch_size):
        sale_id = start_id + i

        # Выбор случайного продукта
        product = random.choice(self.products)
        product_id, _, base_price, _, category = product

        # Выбор случайного клиента
        customer = random.choice(self.customers)
        customer_id, _, region = customer

        # Генерация даты
        random_days = random.randint(0, self.date_range)
        sale_date = self.start_date + timedelta(days=random_days)

        # Генерация количества и суммы
        quantity = random.randint(1, 10)

        # Добавление случайности к цене (+/- 20%)
        price_variation = random.uniform(0.8, 1.2)
        amount = round(base_price * quantity * price_variation, 2)

        # Сезонность для определенных категорий
        if category == 'Clothing' and sale_date.month in [11, 12]:
            # Повышенные продажи одежды зимой
            amount *= random.uniform(1.1, 1.5)
        elif category == 'Sports' and sale_date.month in [5, 6, 7]:
            # Повышенные продажи спортивных товаров летом
            amount *= random.uniform(1.1, 1.4)

        # Тренд роста со временем
        years_passed = (sale_date.year - self.start_date.year)
        trend_factor = 1 + (years_passed * 0.05)  # 5% рост в год

        # Выходные и праздничные дни
        if sale_date.weekday() >= 5:  # Суббота или воскресенье
            amount *= random.uniform(1.1, 1.3)
```

```
        record = SalesRecord(
            sale_id=sale_id,
            product_id=product_id,
            customer_id=customer_id,
            sale_date=sale_date.strftime('%Y-%m-%d'),
            amount=round(amount * trend_factor, 2),
            quantity=quantity,
            region=region,
            category=category
        )

        records.append(record)

    return records
```

class PostgreSQLWriter: """Класс для записи данных в PostgreSQL"""

```
def __init__(self, connection_params: dict):
    self.connection_params = connection_params
    self.conn = None

def connect(self):
    """Установка соединения с PostgreSQL"""
    try:
        self.conn = psycopg2.connect(**self.connection_params)
        self.conn.autocommit = False
        logger.info(" Connected to PostgreSQL")
    except Exception as e:
        logger.error(f" Connection failed: {e}")
        raise

def create_tables(self):
    """Создание таблиц"""
    with self.conn.cursor() as cur:
        # Таблица продуктов
        cur.execute("""
            CREATE TABLE IF NOT EXISTS products (
                product_id INTEGER PRIMARY KEY,
                product_name VARCHAR(255) NOT NULL,
                price DECIMAL(10,2) NOT NULL,
                cost DECIMAL(10,2) NOT NULL,
                category VARCHAR(50) NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        """)

        # Таблица клиентов
        cur.execute("""
            CREATE TABLE IF NOT EXISTS customers (
                customer_id INTEGER PRIMARY KEY,
                customer_name VARCHAR(255) NOT NULL,
                region VARCHAR(50) NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
```

```
            )
        """)

        # Таблица продаж
        cur.execute("""
            CREATE TABLE IF NOT EXISTS sales (
                sale_id BIGINT PRIMARY KEY,
                product_id INTEGER NOT NULL,
                customer_id INTEGER NOT NULL,
                sale_date DATE NOT NULL,
                amount DECIMAL(10,2) NOT NULL,
                quantity INTEGER NOT NULL,
                region VARCHAR(50) NOT NULL,
                category VARCHAR(50) NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

                CONSTRAINT fk_product FOREIGN KEY (product_id)
                    REFERENCES products(product_id),
                CONSTRAINT fk_customer FOREIGN KEY (customer_id)
                    REFERENCES customers(customer_id)
            )
        """)

        # Создание индексов
        cur.execute("""
            CREATE INDEX IF NOT EXISTS idx_sales_sale_date
            ON sales(sale_date);

            CREATE INDEX IF NOT EXISTS idx_sales_region_category
            ON sales(region, category);

            CREATE INDEX IF NOT EXISTS idx_sales_product_id
            ON sales(product_id);

            CREATE INDEX IF NOT EXISTS idx_sales_customer_id
            ON sales(customer_id);
        """)

        self.conn.commit()
        logger.info(" Tables created successfully")

def insert_products(self, products: List[Tuple]):
    """Вставка данных продуктов"""
    query = """
        INSERT INTO products
        (product_id, product_name, price, cost, category)
        VALUES (%s, %s, %s, %s, %s)
        ON CONFLICT (product_id) DO NOTHING
    """

    with self.conn.cursor() as cur:
        execute_batch(cur, query, products)
        self.conn.commit()
```

```
        logger.info(f" Inserted {len(products)} products")

def insert_customers(self, customers: List[Tuple]):
    """Вставка данных клиентов"""
    query = """
        INSERT INTO customers (customer_id, customer_name, region)
        VALUES (%s, %s, %s)
        ON CONFLICT (customer_id) DO NOTHING
    """

    with self.conn.cursor() as cur:
        execute_batch(cur, query, customers)
        self.conn.commit()
        logger.info(f" Inserted {len(customers)} customers")

def insert_sales_batch(self, batch: List[SalesRecord]):
    """Вставка батча продаж"""
    query = """
        INSERT INTO sales
        (sale_id, product_id, customer_id, sale_date, amount, quantity, region, category)
        VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
    """

    data = [
        (
            record.sale_id,
            record.product_id,
            record.customer_id,
            record.sale_date,
            record.amount,
            record.quantity,
            record.region,
            record.category
        )
        for record in batch
    ]

    with self.conn.cursor() as cur:
        execute_batch(cur, query, data)
        self.conn.commit()

def close(self):
    """Закрытие соединения"""
    if self.conn:
        self.conn.close()
        logger.info(" Connection closed")
```

def worker_process(start_id: int, num_records: int, batch_size: int, worker_id: int): """Функция для параллельной генерации данных""" connection_params = { 'host': 'localhost', 'port': 5432, 'database': 'sales_db', 'user': 'admin', 'password': 'admin123' }

```
writer = PostgreSQLWriter(connection_params)
generator = SalesDataGenerator()

try:
    writer.connect()

    total_batches = num_records // batch_size
    for batch_num in range(total_batches):
        batch_start_id = start_id + (batch_num * batch_size)

        # Генерация батча
        batch = generator.generate_batch(batch_start_id, min(batch_size, num_records))

        # Вставка батча
        writer.insert_sales_batch(batch)

        # Логирование прогресса
        if batch_num % 10 == 0:
            logger.info(f"Worker {worker_id}: Processed {batch_num}/{total_batches} batches")

        # Небольшая пауза для избежания перегрузки
        time.sleep(0.01)

    logger.info(f" Worker {worker_id} completed successfully")

except Exception as e:
    logger.error(f" Worker {worker_id} failed: {e}")
finally:
    writer.close()
```

def main(): """Основная функция""" parser = argparse.ArgumentParser(description='Generate sales data for PostgreSQL') parser.add_argument('--total', type=int, default=10000000, help='Total records to generate') parser.add_argument('--batch-size', type=int, default=50000, help='Batch size for insertion') parser.add_argument('--workers', type=int, default=4, help='Number of parallel workers')

```
args = parser.parse_args()

logger.info(f" Starting data generation: {args.total:,} records")
logger.info(f" Batch size: {args.batch_size:,}, Workers: {args.workers}")

# Создаем базовые таблицы
writer = PostgreSQLWriter({
    'host': 'localhost',
    'port': 5432,
    'database': 'sales_db',
    'user': 'admin',
    'password': 'admin123'
})

try:
    writer.connect()
```

```
    # Создание таблиц
    writer.create_tables()

    # Генерация и вставка продуктов и клиентов
    generator = SalesDataGenerator()
    writer.insert_products(generator.products)
    writer.insert_customers(generator.customers)

    writer.close()

    # Параллельная генерация данных продаж
    records_per_worker = args.total // args.workers
    processes = []

    for i in range(args.workers):
        start_id = i * records_per_worker + 1
        p = mp.Process(
            target=worker_process,
            args=(start_id, records_per_worker, args.batch_size, i + 1)
        )
        processes.append(p)
        p.start()

    # Ожидание завершения всех процессов
    for p in processes:
        p.join()

    logger.info(" All data generation completed successfully")

except Exception as e:
    logger.error(f" Main process failed: {e}")
```

if **name** == "**main**": main()

Проверка сгенерированных данных

-- Общая статистика SELECT COUNT(*) as total_sales, MIN(sale_date) as earliest_sale, MAX(sale_date) as latest_sale, SUM(amount) as total_revenue, ROUND(AVG(amount), 2) as avg_sale_amount, SUM(quantity) as total_quantity FROM sales;

-- Распределение по регионам SELECT region, COUNT(*) as sales_count, SUM(amount) as revenue, ROUND(SUM(amount) * 100.0 / SUM(SUM(amount)) OVER (), 2) as revenue_percent, ROUND(AVG(amount), 2) as avg_sale_amount FROM sales GROUP BY region ORDER BY revenue DESC;

-- Распределение по категориям SELECT category, COUNT(*) as sales_count, SUM(amount) as revenue, SUM(quantity) as total_quantity, ROUND(AVG(amount), 2) as avg_sale_amount FROM sales GROUP BY category ORDER BY revenue DESC;

-- Продажи по месяцам SELECT DATE_TRUNC('month', sale_date) as month, COUNT(*) as sales_count, SUM(amount) as revenue, ROUND(AVG(amount), 2) as avg_sale_amount, ROUND((SUM(amount) - LAG(SUM(amount)) OVER (ORDER BY DATE_TRUNC('month', sale_date))) / LAG(SUM(amount)) OVER (ORDER BY DATE_TRUNC('month', sale_date)) * 100, 2) as growth_percent FROM sales GROUP BY DATE_TRUNC('month', sale_date) ORDER BY month LIMIT 12;

-- Топ 10 продуктов по выручке SELECT p.product_name, s.category, COUNT(*) as sales_count, SUM(s.amount) as total_revenue, SUM(s.quantity) as total_quantity, ROUND(AVG(s.amount), 2) as avg_sale_amount FROM sales s JOIN products p ON s.product_id = p.product_id GROUP BY p.product_name, s.category ORDER BY total_revenue DESC LIMIT 10;

-- Топ 10 клиентов по выручке SELECT c.customer_name, c.region, COUNT(*) as purchases_count, SUM(s.amount) as total_spent, ROUND(AVG(s.amount), 2) as avg_purchase_amount, MAX(s.sale_date) as last_purchase_date FROM sales s JOIN customers c ON s.customer_id = c.customer_id GROUP BY c.customer_name, c.region ORDER BY total_spent DESC LIMIT 10;

-- Ежедневная статистика за последние 30 дней SELECT sale_date, COUNT(*) as daily_sales, SUM(amount) as daily_revenue, ROUND(AVG(amount), 2) as avg_daily_sale, SUM(quantity) as daily_quantity FROM sales WHERE sale_date >= CURRENT_DATE - INTERVAL '30 days' GROUP BY sale_date ORDER BY sale_date DESC;

-- Продажи по дням недели SELECT EXTRACT(DOW FROM sale_date) as day_of_week, CASE EXTRACT(DOW FROM sale_date) WHEN 0 THEN 'Sunday' WHEN 1 THEN 'Monday' WHEN 2 THEN 'Tuesday' WHEN 3 THEN 'Wednesday' WHEN 4 THEN 'Thursday' WHEN 5 THEN 'Friday' WHEN 6 THEN 'Saturday' END as day_name, COUNT(*) as sales_count, SUM(amount) as revenue, ROUND(AVG(amount), 2) as avg_sale_amount FROM sales GROUP BY EXTRACT(DOW FROM sale_date), day_name ORDER BY day_of_week;