Benchmarking openGauss Against PostgreSQL: A Comparative Analysis

Zhou Zihan 12310814

目录

背景介绍:

本项目旨在通过测试一系列的功能去比较两个数据库（PostgreSQL和openGauss）在特定方面的优劣，具体会从以下几个方面进行探讨：

1.大量磁盘输入/输出

2.数据访问和更新的竞争

3.同时执行多种跨度复杂性的事务类型

4.由多种大小、属性和关系的表组成的数据库

数据来源

本项目的数据来源于**Java随机模型生成**的表格数据，以下附上源代码,这个代码的功能是可以输出对应行数的Insert语句用于生成具有大量数据的表格，主要采用了随机数模型去生成大量的数字和字符串，才用的表格是包含两个变量id和name的，分别是int和varchar(255)的数据类型。

```
package DBProject3;

/**
 * @author blueroom
 * @studentID 12310814
 * @school Sustech
 */
import java.util.Random;

public class SqlGenerator {
    private long m;  // 表格的总行数
    private long n;  // 表格的列数
    private String[] columnNames;  // 列字段名
    private String[] columnTypes;  // 列数据类型
```

```
    private Random random;

    public SqlGenerator(long m, long n, String[] columnNames, String[]
columnTypes) {
        this.m = m;
        this.n = n;
        this.columnNames = columnNames;
        this.columnTypes = columnTypes;
        this.random = new Random();
    }

    // 根据输入生成CREATE TABLE语句
    public String generateCreateTableSql(String tableName) {
        StringBuilder sql = new StringBuilder();
        sql.append("CREATE TABLE ").append(tableName).append(" (");

        for (int i = 0; i < n; i++) {
            sql.append(columnNames[i]).append(" ").append(columnTypes[i]);
            if (i < n - 1) {
                sql.append(", ");
            }
        }

        sql.append(");");
        return sql.toString();
    }

    // 根据输入生成批量INSERT语句，数据为随机数或随机字符串
    public String generateBatchInsertSql(String tableName, int batchSize) {
        StringBuilder sql = new StringBuilder();
        sql.append("INSERT INTO ").append(tableName).append(" (");

        // 构建字段部分
        for (int j = 0; j < n; j++) {
            sql.append(columnNames[j]);
            if (j < n - 1) {
                sql.append(", ");
            }
        }

        sql.append(") VALUES ");

        // 生成批量插入数据
        for (long i = 0; i < m; i++) {
            sql.append("(");

            // 根据数据类型生成随机数据
            for (int j = 0; j < n; j++) {
                sql.append(generateRandomData(columnTypes[j]));
                if (j < n - 1) {
                    sql.append(", ");
                }
            }

            sql.append(")");
```

```
            // 每生成完一个批次的INSERT数据，检查是否需要换行或分隔
            if ((i + 1) % batchSize == 0 || i == m - 1) {
                sql.append(";");
                if (i < m - 1) {
                    sql.append("\n");
                    sql.append("INSERT INTO ").append(tableName).append(" (");
                    for (int j = 0; j < n; j++) {
                        sql.append(columnNames[j]);
                        if (j < n - 1) {
                            sql.append(", ");
                        }
                    }
                    sql.append(") VALUES ");
                }
            } else {
                sql.append(", ");
            }
        }

        return sql.toString();
    }

    // 根据数据类型生成随机数据
    private String generateRandomData(String dataType) {
        switch (dataType.toLowerCase()) {
            case "int":
                return String.valueOf(random.nextInt(1000));  // 生成0-999之间的整
数
            case "long":
                return String.valueOf(random.nextLong());  // 生成一个随机long值
            case "varchar":
                return "'" + generateRandomString(10) + "'";  // 生成10个字符的随机
字符串
            default:
                return "NULL";  // 默认返回NULL（可以根据需求扩展）
        }
    }

    // 生成随机字符串
    private String generateRandomString(int length) {
        String characters =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
        StringBuilder randomString = new StringBuilder(length);
        for (int i = 0; i < length; i++) {

 randomString.append(characters.charAt(random.nextInt(characters.length())));
        }
        return randomString.toString();
    }

    public static void main(String[] args) {
        // 示例输入
        long m = 50;  // 总行数
        long n = 2;  // 列数
```

```
        String[] columnNames = {"id", "name", "age"};
        String[] columnTypes = {"int", "varchar", "long"};

        // 创建SqlGenerator对象
        SqlGenerator generator = new SqlGenerator(m, n, columnNames,
columnTypes);

        // 生成CREATE TABLE语句
        String createSql = generator.generateCreateTableSql("person");
        System.out.println(createSql);
        System.out.println();

        // 生成批量INSERT语句，设置每次插入批次为100条
        String insertSql = generator.generateBatchInsertSql("person", 100);
        System.out.println(insertSql);
    }
}
```

本报告中所有表格数据的单位都是毫秒(ms)。

性能测试

1.大量磁盘输入/输出

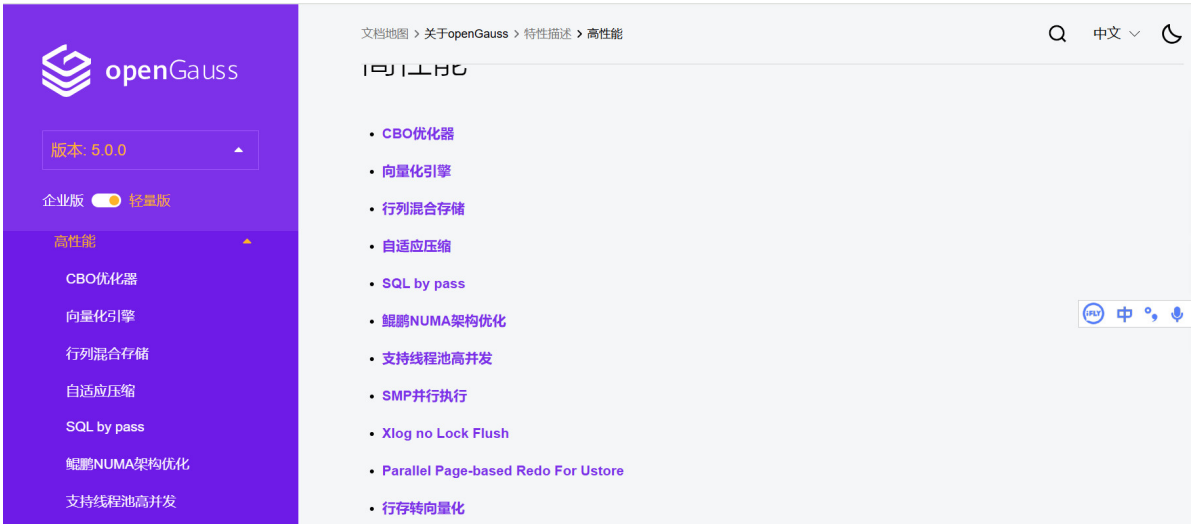
针对这项性能的测试，我将从Insert,Select,Update这三个逻辑出发，通过改变行数，得到两个数据库随着不同参数改变运行时间的改变，我将从行数50~50000来着重探讨执行相应逻辑所需的时间区别，

我将使用

```
EXPLAIN ANALYZE
INSERT INTO person (id, name) VALUES (943, 'ckDlfWi2wl'), (871, 'hJDjfizWmL');
```

这样的模式去对执行这些操作的时间与成本进行测量。

以下是通过调用EXPLAIN ANALYZE获得的三个时间变量，在这里我们着重考虑总体的运行时间Execution Time和每一个可检测的分步骤的时间，即Actual Time(表格中会注释Actual以示区别)。在我的测试中，我发现了一个小特点，就是OpenGuass的Analyse中并没有出现Planning Time这个时间参数，我进行了一些查阅和分析，得到了以下的结果：
在执行一些简单的DDL指令时，OpenGuass省略了查询计划的生成和优化过程，因为这个过程在一些简单的逻辑中显得没有必要；同时，正常情况下PostgreSQL可能会在Planning Time中进行评估是否需要加锁、是否需要并行执行等，而OpenGuass可能略去了这些步骤；还有OpenGuass可能会采用一些硬编码的方式来简化不必要的流程。



结合华为的OpenGuass官网对其高性能的分点解释来看，“没有Planning Time”这一点很有可能和SQL by Pass有关系。SQL bypass 通过简化查询优化过程，避免了复杂的计划生成和优化步骤，因此直接减少了 Planning Time。对于简单查询的加速效果尤为明显，这使得它成为 OpenGauss 减少 Planning Time 的关键特性。同时，也有可能和向量化引擎有关系，向量化引擎通过更高效的数据处理方式提高了查询的执行速度，特别是在涉及大规模数据处理时，使得查询优化器能够更快速地生成执行计划。因此，向量化引擎也是减少 Planning Time 的一个关键因素。

时间类型	相应含义
Planning Time	查询计划阶段的时间，即数据库优化器决定如何执行查询的时间。
Execution Time	查询实际执行阶段的时间，表示从查询开始执行到返回结果的时间。
Actual Time	查询执行中每个操作的实际花费时间。实际时间通常出现在每个操作的详细输出中。

所以我们将主要去探讨这两个数据库在执行同一个指令时的Actual Time的区别来探讨它们性能的优劣。

首先是关于Insert：

这是OpenGuass的表现：

	50	500	5000	50000
Insert(Actual)	0.085	0.875	6.294	86.218
Value(Actual)	0.015	0.213	1.412	16.065
Total	0.096	1.45	6.84	87.598
Cost	0.62	6.25	62.5	625

这是PostgreSQl的表现：

	50	500	5000	50000
Insert(Actual)	0.001	0	0.001	0
Value(Actual)	0.021	0.165	1.243	22.969

Value(Actual)	0.024	0.109	1.243	22.909
	50	**500**	**5000**	**50000**
Total	0.297	1.23	5.059	85.309
Cost	0.62	6.25	62.5	625

经过比对我们可以发现，**OpenGuass有一个很显著的特点，就是它并没有Planning time的表示**，并且在Insert这一个小步骤中所花时间远远高于几乎不花时间的PostgreSQL。同时，对于很小的数据量，OpenGuass显然处理起来很得心应手，可能就是与它对于查询计划的简化有关；但是对于较大数据量，OpenGuass创建Values虚拟表的时间就略高于后者，并且两个数据库对于较大数据量的总运行时间不相上下，PostgreSQL略快一些，结合上述结果进行分析，可以得到一些结论：

OpenGauss 和 PostgreSQL 对于 `INSERT` 操作的处理机制不同。PostgreSQL 在 INSERT 方面的表现更高效，尤其是在处理较小数据量时。

OpenGauss 的额外开销主要体现在事务和虚拟表处理上，这可能导致单条 INSERT 操作的时间较长，尤其是在处理 `VALUES` 子句时的额外开销。

对于较大数据量，OpenGauss 通过某些优化策略可能在批量插入时弥补了单条插入操作的开销，从而使得总体运行时间与 PostgreSQL 相近。虽然 PostgreSQL 略快，但差异不大。

另外，可以观察到两个数据库在进行同样的操作时Cost一样，可能是因为整体的算法结构几乎一样。

所以我们可以看到，在批量处理大量数据的时候，OpenGuass的优势并不是很明显。由于一般对于数据库的需求都是处理极其庞大的数据，OpenGuass可能在进行这种批量单次的逻辑时优化了流程，省去了一些细节，与上面图片中的SQL by Pass密切相关，**SQL Bypass** 作为 OpenGauss 的一个优化特性，可以直接减少小数据量插入时的额外开销，特别是在绕过事务和虚拟表处理时，使得插入操作能够更加高效。这与 **OpenGauss 在处理单条 `INSERT` 操作时的开销问题** 密切相关，特别是在没有复杂逻辑的简单插入操作中，可以显著提高性能并减少时间开销。

2.数据访问和更新的竞争

1. 查询

首先我们使用SELECT对于一个50000行数据的表进行数据的访问操作，记录访问出现相同次数数据的平均时间：

	OpenGuass	**PostgreSQL**
Seq Scan(Actual)	12.93	5.915
Rows Removed	100112	49957
Cost	1794.5	896
Total	13.858	8.682

可以发现OpenGuass在扫描行以及过滤行数的成本与时间都几乎是PostgreSQL的两倍，可见在大量数据查找方面OpenGuass的性能低于PostgreSQL，在经过相关查询和分析，可以得到以下结论：

OpenGauss 的查询优化可能还不如 **PostgreSQL**，经过我的分析和猜测，我觉得OpenGuass可能是需要检查是否有正确的索引，或是否采用了顺序扫描（`Seq Scan`）的方式，而且还与上面图片中提到的行列混合存储有关。假设查询需要扫描大量行，并且只使用少数几个字段。在这种情况下，列存储可能无法提供预期的性能提升，我推测可能**列存储的优势主要在于处理单列查询或大规模数据聚合**（如`SUM()`、`COUNT()`等）。如果查询需要获取多列的数据，使用列存储可能会增加不必要的列加载，导致性能降低。

Sum() and Count() Test

```
-- 计算age列的总和
SELECT SUM(id) AS total_age FROM person;

-- 计算age列的行数
SELECT COUNT(id) AS total_count FROM person;
```

```
-- 计算age列的总和，并过滤age > 30的记录
SELECT SUM(id) AS total_age
FROM person
WHERE id > 30;

-- 计算age列的行数，并过滤age > 30的记录
SELECT COUNT(id) AS total_count
FROM person
WHERE id > 30;
```

(500000 rows)	OpenGuass	Postgresql
Sum(No limitations)	2080	2260
Count(No limitations)	1510	1633
Sum(With limitations)	1708	1629
Count(With limitations)	5880	1307

可以发现在Sum()和Count()这样的测试中，OpenGuass的表现实在是不尽人意，可能还是与顺序扫描（`Seq Scan`）的方式等逻辑有关，看来前面的推测"**列存储的优势主要在于处理单列查询或大规模数据聚合"不一定完全正确**，只能说OpenGuass在这方面的性能差强人意，并不具有相比PostgreSql的明显优势。

2.UPDATE 操作中的竞争

(1)行级锁对比：

我们将用以下代码来检测两个数据库对于UPDATE操作中的竞争的处理：

```
-- 执行第一个事务的操作
BEGIN;
UPDATE person SET name = 'Temp1' WHERE id = 1;

-- 在另一个会话中执行此查询
SELECT * FROM pg_locks WHERE relation = (SELECT oid FROM pg_class WHERE relname
= 'person');

-- 执行第二个事务的操作
BEGIN;
UPDATE person SET name = 'Temp2' WHERE id = 1;`
```

检测代码：

```
SELECT * FROM pg_locks WHERE relation = (SELECT oid FROM pg_class WHERE relname
= 'person');
```

OpenGuass显示以下lock的类型：

locktype	database	relation	page	tuple	bucket	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	sessionid	mode	granted	fastpath
relation	14560	12010									12/4156	1.4E+14	1.4E+14	AccessShareLock	TRUE	TRUE
virtualxid						12/4156					12/4156	1.4E+14	1.4E+14	ExclusiveLock	TRUE	TRUE
virtualxid						8/4331					8/4331	1.4E+14	1.4E+14	ExclusiveLock	TRUE	TRUE

PostgreSQL显示以下lock的类型：

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
relation	13761	12290								3/10995	851	AccessShareLock	TRUE	TRUE	
virtualxid					3/10995					3/10995	851	ExclusiveLock	TRUE	TRUE	

在 **PostgreSQL** 和 **OpenGauss** 中，针对并发 UPDATE 操作，**行级锁**将被用来确保每个事务对数据的独占访问，避免多个事务修改同一行数据。

两者的 virtualxid 锁虽然在功能上相似（都用于标识并发事务），但 **OpenGauss** 显示了更多的标识符和状态信息，可能意味着它在内部对事务的跟踪和锁的管理有更多的细节。

(2)表级锁对比：

```
-- 表级锁的检测
BEGIN;
ALTER TABLE person ADD COLUMN address VARCHAR;
COMMIT;
-- 在另一个会话中执行此查询
BEGIN;
SELECT * FROM person;
COMMIT;
```

检测代码：

```
SELECT * FROM pg_stat_activity;  -- 查看当前活动的会话
SELECT * FROM pg_locks;          -- 查看当前锁的信息
```

```
[YY003] ERROR: Lock wait timeout: thread 140269959182080 on node gaussdb waiting for AccessShareLock on relation 24629 of database 14560 after 1199922.988 ms 详细：blocked by hold lock thread 140269913634560, statement <SHOW TRANSACTION ISOLATION LEVEL>, hold lockmode AccessExclusiveLock.
```

OpenGuass:

locktype	database	relation	page	tuple	bucket	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	sessionid	mode	granted	fastpath	locktag	global_sessionid	
virtualxid						8/5801					8/5801	1.4E+14	1.4E+14	ExclusiveLock	TRUE	TRUE	8:16a9:0:0:0:7	0:0#0	
transactionid						18761				11/13523	1.4E+14	1.4E+14	ExclusiveLock	TRUE	FALSE	4949:0:0:0:6	0:0#0	
relation	14560	24629									11/13523	1.4E+14	1.4E+14	RowExclusiveLock	TRUE	FALSE	38e0:6035:0:0:0	0:0#0	
relation	14560	24629									11/13523	1.4E+14	1.4E+14	AccessExclusiveLock	TRUE	FALSE	38e0:6035:0:0:0	0:0#0	

PostgreSQL:

locktype	database	relation	page	tuple	virtualxid	transactionid	classid	objid	objsubid	virtualtransaction	pid	mode	granted	fastpath	waitstart
virtualxid					3/11017					3/11017	851	ExclusiveLock	TRUE	TRUE	
relation	13761	24731								4/1989	981	AccessExclusiveLock	TRUE	FALSE	
transactionid					8081				4/1989	981	ExclusiveLock	TRUE	FALSE		

PostgreSQL 和 **OpenGauss** 在这方面的锁机制相似，都使用AccessExclusiveLock来保护ALTER TABLE 操作，防止其他事务在表结构变更时进行读取或写入。因此，当两个会话中的操作发生锁竞争时，都会遇到类似的 **锁等待超时** 错误。

经过对两种lock类型的检验与探索，可以得出一些有关安全性的结论，

行级锁的管理：

- 在处理行级锁时，PostgreSQL 和 OpenGauss 都具备较强的事务隔离能力。两者都会确保 **行级锁** 不会造成数据竞争，在不同事务之间保持数据一致性和完整性。
- 在并发处理多个事务时，行级锁是确保事务隔离性（如REPEATABLE READ 或SERIALIZABLE）的一种机制，两者都能有效避免行级数据的并发修改导致的问题。

表级锁的处理：

- 两个数据库在执行ALTER TABLE这样的表结构修改操作时，都采用了 AccessExclusiveLock 锁，这是一种非常强的锁，会完全阻止其他事务访问同一张表。这确保了在表结构变更过程中不会发生数据损坏或者一致性问题。
- 由于 AccessExclusiveLock锁的强制性，两者在表结构变更时的处理方式较为一致，都要求其他事务等待，直到表变更完成。

结合OpenGuass宣传的高性能特点，我们可以分析一下OpenGuass对于锁的改进在于：

OpenGauss 引入了 **Xlog 无锁刷新**（Xlog no Lock Flush）机制，在事务日志的刷新过程中避免了锁的使用。传统的数据库系统在刷新事务日志时会锁定一定的资源，影响性能。OpenGauss 的这一优化使得日志写入的过程更加高效，避免了由于日志写入时的锁竞争造成的性能瓶颈。同时，我还查阅了一下资料，发现了还有一个特点，OpenGauss 在存储引擎的日志重做操作中，采用了 **基于页的并行重做（Parallel Page-based Redo）**，使得数据在恢复时能够并行处理，提升了大规模数据写入操作时的性能。通过减少重做操作期间对锁的需求，OpenGauss 在进行大规模事务操作时表现得更加高效。

3.同时执行多种跨度复杂性的事务类型

在这一部分中，我们将用50,500,5000,50000组transactions来测试两个数据库的性能，以下是用来生成指令的java文件;

```
package DBProject3;

/**
 * @author blueroom
 * @studentID 12310814
 * @school Sustech
 */
import java.util.Scanner;

public class SqlSetGenerator{

    // 生成一个事务的SQL操作，包含插入、更新和删除操作
    public static String generateTransactionSql(int transactionId) {
        StringBuilder sql = new StringBuilder();
        sql.append("BEGIN;\n");

        // 插入新数据
        sql.append("INSERT INTO person2 (id, name) VALUES (")
                .append(transactionId)
                .append(", 'Name")
                .append(transactionId)
                .append("');\n");

        // 更新数据
        sql.append("UPDATE person2 SET name = 'Name Updated ")
                .append(transactionId)
                .append("' WHERE id = ")
                .append(transactionId)
                .append(";\n");

        // 删除数据
        sql.append("DELETE FROM person2 WHERE id = ")
                .append(transactionId)
                .append(";\n");

        // 提交事务
        sql.append("COMMIT;\n");

        return sql.toString();
    }

    // 生成指定次数的事务SQL操作
    public static String generateMultipleTransactions(int times) {
        StringBuilder allSql = new StringBuilder();

        // 循环生成每个事务
        for (int i = 1; i <= times; i++) {
            allSql.append(generateTransactionSql(i)).append("\n");
        }

        return allSql.toString();
    }
```

```
    public static void main(String[] args) {
        // 创建Scanner对象以接收用户输入
        Scanner scanner = new Scanner(System.in);

        // 提示用户输入事务数量
        System.out.print("请输入事务次数：");
        int times = scanner.nextInt();

        // 生成并输出SQL指令集合
        String transactionsSql = generateMultipleTransactions(times);
        System.out.println("生成的SQL指令集合：\n");
        System.out.println(transactionsSql);

        scanner.close();
    }
}
```

一下分别是两个数据库对于不同组（由小到大）Transactions进行的时长比较(单位：ms)：

	50	500	5000
OpenGuass	4836	71923	934999
PostgreSQL	4486	132613	3447938

可以看到OpenGuass在小数据量测试的时候与PostgreSQL具有相近的运行时间，并不具有很明显的优势，但是在较大与很大的数据量规模上具有非常明显的优势，运行时间几乎是PostgreSQL运行时间的一半，下面请由我结合上述图片中提到的OpenGuass的高性能点来分析一下出现这种情况的可能原因：

支持线程池高并发

OpenGauss 引入了线程池机制，用于管理数据库的连接和任务执行。相比 PostgreSQL 的基于进程的连接处理，线程池能够减少操作系统在**创建、销毁和切换进程**上的开销，特别是在高并发事务场景下更加高效。每个事务在 OpenGauss 中由线程池动态调度执行，而 PostgreSQL 会为每个连接创建单独的进程，导致系统资源（CPU、内存）消耗更大。

SMP并行执行

OpenGauss 的对称多处理（SMP）并行执行模式，支持对复杂查询、事务操作进行任务分解，将操作分发到多个 CPU 核心上并行完成。通过智能调度，OpenGauss 可动态调整并行度，尤其是在大量事务类型跨度较大的场景中，优化每个事务的执行时间。

行列混合存储

OpenGauss 支持行存和列存的混合存储模型，可根据事务类型动态调整存储模式：

- **行存**：适用于频繁写入和事务型操作（如 `INSERT`、`UPDATE`）。
- **列存**：适用于读密集型操作（如复杂的 `SELECT`）。

在执行大量事务时，OpenGauss 能够根据操作类型选择最优的存储方式，减少磁盘 I/O 和存储系统的延迟。

鲲鹏 NUMA 架构优化

OpenGauss 针对鲲鹏 CPU 进行了 NUMA（非一致性内存访问）架构优化，充分利用硬件特性，减少跨 NUMA 节点的内存访问开销。在处理大量事务时，OpenGauss 能够根据硬件架构动态优化任务调度和内存分配。

4.由多种大小、属性和关系的表组成的数据库

我们将使用以下代码创建一个复杂的数据库环境，其中包含多个表，并且这些表具有不同的属性、数据大小和关系。然后我们将进行一系列操作（数据量查询，关联操作等五个操作）来检测两个数据库在这里的区别。

```
-- 创建用户表
CREATE TABLE users (
    id SERIAL PRIMARY KEY,  -- 自动增长的用户ID
    name VARCHAR(255) NOT NULL,  -- 用户姓名
    email VARCHAR(255) UNIQUE NOT NULL,  -- 用户邮箱
    registered_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  -- 注册时间
);

-- 创建产品表
CREATE TABLE products (
    id SERIAL PRIMARY KEY,  -- 自动增长的产品ID
    name VARCHAR(255) NOT NULL,  -- 产品名称
    price DECIMAL(10, 2) NOT NULL,  -- 产品价格
    stock INT DEFAULT 0  -- 产品库存
);

-- 创建订单表
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,  -- 自动增长的订单ID
    user_id INT REFERENCES users(id),  -- 外键关联到用户表
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  -- 订单时间
    status VARCHAR(50) DEFAULT 'pending'  -- 订单状态
);

-- 创建订单-产品关联表 (多对多关系)
CREATE TABLE order_items (
    order_id INT REFERENCES orders(id),  -- 外键关联到订单表
    product_id INT REFERENCES products(id),  -- 外键关联到产品表
    quantity INT DEFAULT 1,  -- 产品数量
    PRIMARY KEY (order_id, product_id)  -- 复合主键，保证每个订单每个产品唯一
);

-- 示例数据插入 (多个表涉及)
-- 插入一些用户数据
INSERT INTO users (name, email) VALUES
    ('Alice', 'alice@example.com'),
    ('Bob', 'bob@example.com'),
    ('Charlie', 'charlie@example.com');

-- 插入一些产品数据
INSERT INTO products (name, price, stock) VALUES
```

```
    ('Product A', 10.00, 100),
    ('Product B', 20.00, 50),
    ('Product C', 30.00, 200);

-- 插入一些订单数据
INSERT INTO orders (user_id, order_date, status) VALUES
    (1, '2024-12-01', 'pending'),
    (2, '2024-12-02', 'shipped'),
    (3, '2024-12-03', 'delivered');

-- 插入订单-产品关联数据
INSERT INTO order_items (order_id, product_id, quantity) VALUES
    (1, 1, 2),  -- 订单1，购买产品A，2件
    (1, 2, 1),  -- 订单1，购买产品B，1件
    (2, 3, 3),  -- 订单2，购买产品C，3件
    (3, 1, 1);  -- 订单3，购买产品A，1件
```

（1）数据量测试：

- 你可以通过大量插入数据来测试数据库的表现。例如，可以将用户表插入数百万条记录，测试其响应时间。

```
INSERT INTO users (name, email)
SELECT 'user' || generate_series(1, 1000000), 'user' || generate_series(1,
1000000) || '@examp'
```

	OpenGuass	PostgreSQL
Running time	13162	7978

通过表格中的数据可以发现OpenGuass所花时间近乎于PostgreSQL的两倍，下面由我来分析一下可能出现这种情况的原因：

SMP并行执行

OpenGauss 在事务处理时使用了 SMP（对称多处理）并行执行技术，这对于一些复杂查询和长时间运行的事务有显著的加速作用。然而，在简单的批量插入操作中，过多的并行化可能会带来调度和资源竞争的开销，反而影响了插入性能。对于简单的 `INSERT` 操作，PostgreSQL 可能通过直接执行单线程插入或使用更高效的批量插入方式（如 `COPY` 命令）获得更好的性能。

额外测试：

```
COPY formal_user (name, email) FROM '.../DatagripSaves/Test1/tmp_users.csv' WITH
CSV;
```

当采用COPY直接导入现有CSV文件（与Insert的数据类型和数据内容相同）时，结果就变成了下表;

	OpenGuass	PostgreSQL
Running time	2246	3020

由此可见，PostgreSQL 确实可以通过直接执行单线程插入或使用更高效的批量插入方式（如 `COPY` 命令）获得更好的性能。也说明了**传统的插入数据的方法去生成表格是不适用于OpenGuass**的。

Xlog no Lock Flush 和日志写入

OpenGauss 采用了 **Xlog no Lock Flush** 技术，这减少了日志写入时的锁争用。然而，虽然这在事务提交时能提升性能，但在执行大规模插入时，**频繁的日志**写入仍然可能带来瓶颈。如果 OpenGauss 的日志管理机制没有针对批量数据插入进行优化，这可能会导致在数据插入过程中频繁写入日志，增加 I/O 开销。

(2)关联查询测试：

- 测试表之间的关联查询，例如查询每个订单和它所包含的产品：

```
SELECT o.id AS order_id, u.name AS user_name, p.name AS product_name,
oi.quantity
FROM orders o
JOIN users u ON o.user_id = u.id
JOIN order_items oi ON o.id = oi.order_id
JOIN products p ON oi.product_id = p.id
WHERE o.status = 'shipped';
```

OpenGuass:

QUERY PLAN
Nested Loop (cost=4.33..66.47 rows=14 width=534) (actual time=1.610..1.617 rows=1 loops=1)
-> Nested Loop (cost=4.33..60.68 rows=19 width=22) (actual time=1.321..1.325 rows=1 loops=1)
-> Nested Loop (cost=0.00..32.79 rows=2 width=14) (actual time=0.939..0.942 rows=1 loops=1)
-> Seq Scan on orders o (cost=0.00..16.23 rows=2 width=8) (actual time=0.224..0.226 rows=1 loops=1)
Filter: ((status)::text = 'shipped'::text)
Rows Removed by Filter: 2
-> Index Scan using users_pkey on users u (cost=0.00..8.27 rows=1 width=14) (actual time=0.709..0.710 rows=1 loops=1)
Index Cond: (id = o.user_id)
-> Bitmap Heap Scan on order_items oi (cost=4.33..13.85 rows=10 width=12) (actual time=0.375..0.376 rows=1 loops=1)
Recheck Cond: (order_id = o.id)
Heap Blocks: exact=1
-> Bitmap Index Scan on order_items_pkey (cost=0.00..4.33 rows=10 width=0) (actual time=0.212..0.212 rows=1 loops=1)
Index Cond: (order_id = o.id)
-> Index Scan using products_pkey on products p (cost=0.00..0.29 rows=1 width=520) (actual time=0.284..0.285 rows=1 loops=1)
Index Cond: (id = oi.product_id)
Total runtime: 1.783 ms

PostgreSQL:

QUERY PLAN
Nested Loop (cost=4.80..62.32 rows=8 width=534) (actual time=1.926..1.933 rows=1 loops=1)
-> Nested Loop (cost=4.66..60.83 rows=8 width=22) (actual time=1.531..1.538 rows=1 loops=1)
-> Nested Loop (cost=0.42..33.14 rows=2 width=14) (actual time=1.160..1.164 rows=1 loops=1)
-> Seq Scan on orders o (cost=0.00..16.25 rows=2 width=8) (actual time=0.395..0.398 rows=1 loops=1)
Filter: ((status)::text = 'shipped'::text)
Rows Removed by Filter: 2
-> Index Scan using users_pkey on users u (cost=0.42..8.44 rows=1 width=14) (actual time=0.758..0.758 rows=1 loops=1)
Index Cond: (id = o.user_id)
-> Bitmap Heap Scan on order_items oi (cost=4.23..13.75 rows=10 width=12) (actual time=0.362..0.365 rows=1 loops=1)
Recheck Cond: (order_id = o.id)
Heap Blocks: exact=1
-> Bitmap Index Scan on order_items_pkey (cost=0.00..4.23 rows=10 width=0) (actual time=0.165..0.165 rows=1 loops=1)
Index Cond: (order_id = o.id)
-> Index Scan using products_pkey on products p (cost=0.14..0.19 rows=1 width=520) (actual time=0.390..0.390 rows=1 loops=1)
Index Cond: (id = oi.product_id)
Planning Time: 2.528 ms
Execution Time: 2.027 ms

在对比 OpenGauss 和 PostgreSQL 的查询性能时，**OpenGauss 显示出更好的执行效率**，整体执行时间为 1.783 ms，略低于 PostgreSQL 的 2.027 ms。OpenGauss 在每个子操作（如 `Seq Scan` 和 `Index Scan`）上也表现更高效，尤其是在索引扫描和缓存利用方面，减少了 I/O 操作，表现出更优的性能。尽管 PostgreSQL 在查询规划阶段花费了更多时间（2.528 ms），但 OpenGauss 的查询优化器减少了规划时间，提升了执行效率。因此，在此测试场景下，OpenGauss 在性能上优于 PostgreSQL，特别适用于高效查询需求。

(3)更新测试：

- 测试更新数据的操作，观察数据库性能：

```
UPDATE products
SET stock = stock - 1
WHERE id IN (SELECT product_id FROM order_items WHERE order_id = 1);
```

OpenGuass:

QUERY PLAN
Update on products (cost=0.00..26.63 rows=10 width=552) (actual time=1.096..1.127 rows=2 loops=1)
-> Nested Loop Semi Join (cost=0.00..26.63 rows=10 width=552) (actual time=0.571..0.588 rows=2 loops=1)
-> Seq Scan on products (cost=0.00..11.43 rows=143 width=546) (actual time=0.008..0.011 rows=3 loops=1)
-> Index Scan using order_items_pkey on order_items (cost=0.00..0.61 rows=10 width=10) (actual time=0.556..0.556 rows=2 loops=3)
Index Cond: ((order_id = 1) AND (product_id = products.id))
Total runtime: 1.295 ms

PostgreSQL:

QUERY PLAN
Update on products (cost=14.91..26.71 rows=0 width=0) (actual time=2.373..2.375 rows=0 loops=1)
-> Hash Join (cost=14.91..26.71 rows=10 width=16) (actual time=0.991..0.999 rows=2 loops=1)
Hash Cond: (products.id = order_items.product_id)
-> Seq Scan on products (cost=0.00..11.40 rows=140 width=14) (actual time=0.057..0.060 rows=3 loops=1)
-> Hash (cost=14.79..14.79 rows=10 width=10) (actual time=0.794..0.795 rows=2 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 9kB
-> Bitmap Heap Scan on order_items (cost=4.23..14.79 rows=10 width=10) (actual time=0.153..0.156 rows=2 loops=1)
Recheck Cond: (order_id = 1)
Heap Blocks: exact=1
-> Bitmap Index Scan on order_items_pkey (cost=0.00..4.23 rows=10 width=0) (actual time=0.030..0.030 rows=2 loops=1)
Index Cond: (order_id = 1)
Planning Time: 5.348 ms
Execution Time: 2.860 ms

在测试 **OpenGauss** 和 **PostgreSQL** 的 **更新操作（UPDATE）** 性能时，发现两者在处理查询时的表现有显著差异。**OpenGauss** 使用了 **Nested Loop Semi Join** 和 **索引扫描**，执行时间为 1.295ms，查询计划较为简单，执行效率较高。它在处理较小或中等数据量时表现优异，执行时间较短，且内存消耗较少。

相比之下，**PostgreSQL** 使用了 **哈希连接** 和 **位图扫描**，执行时间为 2.860ms，并且在查询计划阶段花费了更多时间（**Planning Time: 5.348ms**）。哈希连接虽然适用于大数据集，但在此测试中并非最优，导致了较高的执行时间。此外，PostgreSQL 的内存使用也较高，显示了 9KB 的内存消耗，表明其需要更多的内存来处理查询。

(4)删除测试：

- 测试删除操作，特别是涉及外键约束时的表现：

```
DELETE FROM order_items WHERE order_id = 1;
```

OpenGauss:

QUERY PLAN
Delete on order_items (cost=4.33..14.88 rows=10 width=6) (actual time=0.597..0.614 rows=2 loops=1)
-> Bitmap Heap Scan on order_items (cost=4.33..14.88 rows=10 width=6) (actual time=0.129..0.131 rows=2 loops=1)
Recheck Cond: (order_id = 1)
Heap Blocks: exact=1
-> Bitmap Index Scan on order_items_pkey (cost=0.00..4.33 rows=10 width=0) (actual time=0.067..0.067 rows=2 loops=1)
Index Cond: (order_id = 1)
Total runtime: 0.982 ms

PostgreSQL:

QUERY PLAN
Delete on order_items (cost=4.23..14.79 rows=0 width=0) (actual time=0.299..0.299 rows=0 loops=1)
-> Bitmap Heap Scan on order_items (cost=4.23..14.79 rows=10 width=6) (actual time=0.088..0.089 rows=2 loops=1)
Recheck Cond: (order_id = 1)
Heap Blocks: exact=1
-> Bitmap Index Scan on order_items_pkey (cost=0.00..4.23 rows=10 width=0) (actual time=0.042..0.042 rows=2 loops=1)
Index Cond: (order_id = 1)
Planning Time: 0.576 ms
Execution Time: 0.573 ms

PostgreSQL 在 `DELETE` 操作的性能上明显优于 **OpenGauss**，表现为更短的执行时间（0.573 ms vs. 0.982 ms）。虽然两者的查询成本差异较小，但 PostgreSQL 的查询优化和执行过程中的资源利用更高效，尤其是在 `Bitmap Index Scan` 的执行上。OpenGauss 虽然在规划阶段减少了开销，但总体执行时间较长。此差异可能源于 PostgreSQL 在 I/O 操作、缓存管理和执行路径优化方面的优势，适合需要高效删除操作的场景。

(5)复杂查询测试：

- 测试涉及多个表、过滤条件、排序和聚合的复杂查询：

```
SELECT u.name, COUNT(o.id) AS order_count, SUM(oi.quantity) AS total_quantity
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
LEFT JOIN order_items oi ON o.id = oi.order_id
GROUP BY u.id
ORDER BY total_quantity DESC;
```

OpenGuass:

QUERY PLAN
Sort (cost=227322.08..229822.08 rows=1000003 width=38) (actual time=4742.692..4867.290 rows=1000003 loops=1)
Sort Key: (sum(oi.quantity)) DESC
Sort Method: external merge Disk: 37184kB
-> GroupAggregate (cost=183.65..76392.41 rows=1000003 width=38) (actual time=0.711..4352.696 rows=1000003 loops=1)
Group By Key: u.id
-> Merge Left Join (cost=183.65..37100.13 rows=3905634 width=22) (actual time=0.594..3748.298 rows=1000003 loops=1)
Merge Cond: (u.id = o.user_id)
-> Index Scan using users_pkey on users u (cost=0.00..34387.29 rows=1000003 width=14) (actual time=0.179..3549.565 rows=1000003 loops=1)
-> Sort (cost=183.65..188.51 rows=1945 width=12) (actual time=0.383..0.384 rows=3 loops=1)
Sort Key: o.user_id
Sort Method: quicksort Memory: 25kB
-> Hash Right Join (cost=21.21..77.40 rows=1945 width=12) (actual time=0.218..0.295 rows=3 loops=1)
Hash Cond: (oi.order_id = o.id)
-> Seq Scan on order_items oi (cost=0.00..29.45 rows=1945 width=8) (actual time=0.009..0.009 rows=2 loops=1)
-> Hash (cost=14.98..14.98 rows=498 width=8) (actual time=0.078..0.078 rows=3 loops=1)
Buckets: 32768 Batches: 1 Memory Usage: 1kB
-> Seq Scan on orders o (cost=0.00..14.98 rows=498 width=8) (actual time=0.050..0.073 rows=3 loops=1)
Total runtime: 4966.357 ms

PostgreSQL:

QUERY PLAN
Sort (cost=201201.55..203701.56 rows=1000003 width=30) (actual time=1364.614..1457.501 rows=1000003 loops=1)
Sort Key: (sum(oi.quantity)) DESC
Sort Method: external merge Disk: 37216kB
-> GroupAggregate (cost=169.62..77617.39 rows=1000003 width=30) (actual time=51.026..1126.079 rows=1000003 loops=1)
Group Key: u.id
-> Merge Left Join (cost=169.62..37017.27 rows=4080012 width=22) (actual time=50.826..811.042 rows=1000003 loops=1)
Merge Cond: (u.id = o.user_id)
-> Index Scan using users_pkey on users u (cost=0.42..34317.47 rows=1000003 width=14) (actual time=0.342..649.131 rows=1000003 loops=1)
-> Sort (cost=169.19..174.29 rows=2040 width=12) (actual time=0.458..0.463 rows=3 loops=1)
Sort Key: o.user_id
Sort Method: quicksort Memory: 25kB
-> Hash Right Join (cost=21.25..57.05 rows=2040 width=12) (actual time=0.268..0.281 rows=3 loops=1)
Hash Cond: (oi.order_id = o.id)
-> Seq Scan on order_items oi (cost=0.00..30.40 rows=2040 width=8) (actual time=0.013..0.015 rows=2 loops=1)
-> Hash (cost=15.00..15.00 rows=500 width=8) (actual time=0.188..0.190 rows=3 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 9kB
-> Seq Scan on orders o (cost=0.00..15.00 rows=500 width=8) (actual time=0.176..0.178 rows=3 loops=1)
Planning Time: 2.775 ms
JIT:
Functions: 21
Options: Inlining false, Optimization false, Expressions true, Deforming true
Timing: Generation 1.520 ms, Inlining 0.000 ms, Optimization 2.047 ms, Emission 47.914 ms, Total 51.481 ms
Execution Time: 1733.750 ms

在排序和聚合操作的性能比较中，PostgreSQL 显示出明显优于 OpenGauss 的表现。PostgreSQL 的总执行时间为 **1733.750 ms**，比 OpenGauss 的 **4966.357 ms** 快了约 **186%**，差距达到 **3232.607 ms**。尽管两者的排序磁盘使用量相近（PostgreSQL 为 **37216 kB**，OpenGauss 为 **37184 kB**），PostgreSQL 在查询优化方面做得更好，通过精细化的 **Hash Join** 和 **Merge Join** 优化，提高了数据连接和聚合的效率。此外，PostgreSQL 在执行过程中通过更高效的内存和磁盘管理，减少了不必要的磁盘 I/O 和中间计算，从而显著缩短了执行时间。相比之下，OpenGauss 的较高查询成本和较慢的连接聚合操作导致了更长的执行时间。尽管两者的磁盘使用量相似，但 PostgreSQL 的执行效率更高，能够更快速地处理大规模数据的排序和聚合。综合来看，PostgreSQL 在资源使用和查询优化方面表现更好，适合处理复杂的排序和聚合任务，特别是在大数据量场景下，具有更高的性能和性价比。

报告总结

经过上述的探讨，我可以得到OpenGuass相较于PostgreSQL的一些不同点。首先，在大量磁盘输入/输出方面，我得出的结论是在批量处理大量数据的时候，**OpenGuass的优势并不是很明显**，特别是在没有复杂逻辑的简单插入操作中，可以显著提高性能并减少时间开销，这可能与SQL By pass这个所谓的特性有关系。其次，在UPDATE 操作中的竞争这个测试中，我们由锁出发着重探讨了安全性的问题，最后得到的结论是在处理行级锁时PostgreSQL 和 OpenGauss **都具备较强的事务隔离能力**，两者都能有效避免行级数据的并发修改导致的问题。两个数据库在执行 ALTER TABLE这样的表结构修改操作时都采用了 AccessExclusiveLock 锁，这使得两者在表结构变更时的**处理方式较为一致**，都要求其他事务等待，直到表变更完成。Xlog 无锁刷新和基于页的并行重做（Parallel Page-based Redo）也在其中得到应用。然后关于同时执行多种跨度复杂性的事务类型，我得到的结论是OpenGuass在**小数据量测试的时候与PostgreSQL具有相近的运行时间**，并不具有很明显的优势，但是在较大与很**大的数据量规模上**具有**非常明显的优势**，这主要和高并发线程池机制，SMP并行执行和行列混合存储有关。最后，我们测试了由多种大小、属性和关系的表组成的数据库下的几种DDL基本操作的效率，得到的结果是**传统的插入数据的方法去生成表格**是不**适用于OpenGuass的**；**OpenGauss 显示出更好查询的执行效率**；更新操作中，**OpenGuass在处理较小或中等数据量时表现优异**，执行时间较短，且内存消耗较少;但是**PostgreSQL 在DELETE操作的性能上明显优于 OpenGauss**；在排序和聚合操作的性能比较中，**PostgreSQL 显示出明显优于 OpenGauss 的表现。**

参考文献

Sai. (2023). **如何理解OpenGauss的高并发性能优化**. 知乎. https://www.zhihu.com/question/637577102/answer/3346674968.

openGauss社区. (2023). **openGauss 高性能特点**. openGauss官方文档. https://docs.opengauss.org/zh/docs/5.0.0-lite/docs/AboutopenGauss/%E9%AB%98%E6%80%A7%E8%83%BD.html.

Github链接（包含报告文档和源代码）：https://github.com/blueroom18/Benchmarking-openGauss-Against-PostgreSQL-A-Comparative-Analysis.git

写在最后：非常感谢于教授一个学期以来对于数据库的详细与生动的讲解，我可能并没有完全学透彻，也曾在第二个Project的时候未能取得超过三十分的分数，但是我还是尽我最大的努力完成这一份报告，在今后的学习中我会继续去探索数据库的原理与逻辑，但由于作业的繁重与时间的紧迫，这篇报告可能不会很令人满意，希望教授能酌情给一个好点的分数，最后再次感谢于教授对于数据库原理的讲解。