

# Project 3: Benchmarking openGauss Against PostgreSQL: A Comparative Analysis

## Criteria评价指标

### 基础评价

对于insert、update等基础操作，实现相对简单，对于造成两者差异的原因没有太多探究乐趣，故本文档略过，转而测试 JOIN 多表连接。

### JOIN多表连接

多表连接的执行计划比较复杂，在多表联合查询的时候，如果我们查看它的执行计划，就会发现里面有多表之间的连接方式。多表之间的连接有三种方式：Nested Loops，Hash Join 和 Sort Merge Join。我将对于表的行数非常多的情况，通过分析个数据库查询计划的差异，阐述差异产生的原因。

### 压力测试

#### 点查QPS性能测试

### Specialties

postgres作为性能最佳的开源关系数据库，其声誉是其他开源数据库难以企及的。然而，这就意味着postgres的实现一定是最优的吗？

在postgres本就优秀的点查、OLTP等方面继续优化固然是一种成功，但若是能补齐postgres的不足之处，也许更有价值。于是，我收集了国内外网站对postgres的吐槽，并发现了如下问题：

- 1. Delete和Update带来的死元组问题
- 2. 堆表问题

我对比了postgres和openGauss在这几方面的差异，希望找出openGauss是否真正实现了对postgres的补强。

### Environment测试环境

规格	值
设备名称	SKADI
CPU	13th Gen Intel(R) Core(TM) i9-13900H 2.60 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
设备 ID	332C2236-0451-4568-AC1F-ED9B2D981332
产品 ID	00342-30952-61196-AAOEM
系统类型	64 位操作系统, 基于 x64 的处理器

# 基础评价

## JOIN多表连接

### 准备

因此在开始之前，我们需要调整两者的配置，以达到“fair play”的目的。

我自己第一次测试的时候，没有指定opengauss的**并行度**，所以两者的结果相差悬殊：


以下图片来自于第一次测试：

```
HashAggregate (cost=7350620263.09..7350620265.09 rows=200 width=12) (actual time=58213.411..58213.421 rows=100 loops=1)
  Group By Key: t1.c1
    -> Hash Join (cost=360414.30..5089146518.04 rows=452294749010 width=4) (actual time=16377.123..56571.473 rows=10000000 loops=1)
      Hash Cond: (t1.id = t2.id)
      -> Seq Scan on tb_join t1 (cost=0.00..139357.91 rows=9510991 width=8) (actual time=0.405..-4032.103 rows=10000000 loops=1)
      -> Hash (cost=139357.91..139357.91 rows=9510991 width=4) (actual time=16343.766..16343.766 rows=10000000 loops=1)
        Buckets: 2097152 Batches: 8 Memory Usage: 439779kB
        -> Seq Scan on tb_join t2 (cost=0.00..139357.91 rows=9510991 width=4) (actual time=0.012..10991.595 rows=10000000 loops=1)
Total runtime: 58213.751 ms
```

在没有进行并行度配置之前，opengauss的成绩是**58213.751 ms**。

而在配置过query\_dop之后，`explain analysis` 的结果会出现如下字样

```
Streaming(type: LOCAL REDISTRIBUTE dop: 16/16) (cost=200199.77..200202.42
rows=100 width=12)
```

 **Caution**

需要特别注意的是，opengauss的配置有些tricky，原因是其配置名称与PostgreSQL中有所不同。

在 **openGauss** 中，`query_dop` (**Query Degree of Parallelism**) 是控制查询并行度的一个参数。这个参数决定了一个查询在执行时，能够使用的最大并行工作线程数。

而在**postgres**中，对应的参数叫`max_parallel_workers_per_gather`，

### Baseline

#### opengauss

config_name	config_value
shared_buffers	128MB
work_mem	64MB
maintenance_work_mem	16MB
effective_cache_size	128MB
enable_nestloop	on
enable_hashjoin	on
enable_mergejoin	on

config_name	config_value
random_page_cost	4
seq_page_cost	1
temp_buffers	8MB
query_dop	16

## PostgreSQL

config_name	config_value
shared_buffers	128MB
work_mem	64MB
maintenance_work_mem	64MB
effective_cache_size	128GB
enable_nestloop	on
enable_hashjoin	on
enable_mergejoin	on
random_page_cost	4
seq_page_cost	1
temp_buffers	8MB
max_parallel_workers_per_gather	16

## 实验验证

运行脚本： `join.sql`

在这个脚本中，我使用了一个随机生成的1000万（10e7）行的表，进行自关联。

```
CREATE TABLE tb_join (id int, c1 int);
INSERT INTO tb_join select generate_series(1,10000000), random()*99;
```

我没有找到openGauss中有关JIT（**Just-in-Time Compilation**）即时编译的设置，但是openGauss的文档中有提到MOT查询原生编译（JIT），所以我选择使用 `prepare` 关键词显示地预编译查询语句

```
PREPARE test_join AS
SELECT t1.c1, count(*) FROM tb_join t1 JOIN tb_join t2 USING (id) GROUP BY t1.c1;
```

(此处不可避免的会涉及到JIT的差异，但是预编译时间相比执行整个查询操作来说实在微不足道，影响可以忽略不计。此处是因为我不清楚openGauss是否会自动调用JIT，故显式地调用了JIT)

运行命令：

```
explain analyse execute test_join
```

## openGauss的计划如下

```
"Streaming(type: LOCAL GATHER dop: 1/16) (cost=200202.45..200202.71 rows=100
width=12) (actual time=[17231.476,17231.785]..[17231.476,17231.785], rows=100)"
"  -> HashAggregate (cost=200202.45..200202.51 rows=100 width=12) (actual time=
[17210.606,17210.607]..[17221.609,17221.611], rows=100)"
      Group By Key: t1.c1
"      -> Streaming(type: LOCAL REDISTRIBUTE dop: 16/16)
      (cost=200199.77..200202.42 rows=100 width=12) (actual time=
[17107.532,17210.246]..[17118.665,17221.439], rows=1600)"
"      -> HashAggregate (cost=200199.77..200199.83 rows=100 width=12)
      (actual time=[17122.979,17122.998]..[17227.694,17227.719], rows=1600)"
            Group By Key: t1.c1
"            -> Hash Join (cost=91432.81..197074.75 rows=10000048
width=4) (actual time=[4118.325,16962.350]..[4346.175,24613.910], rows=10000000)"
                  Hash Cond: (t1.id = t2.id)
"                  -> Streaming(type: LOCAL REDISTRIBUTE dop: 16/16)
                  (cost=0.00..76906.34 rows=10000048 width=8) (actual time=[0.070,3145.846]..
[212.893,3349.521], rows=10000000)"
"                  -> Seq Scan on tb_join t1 (cost=0.00..24015.53
rows=10000048 width=8) (actual time=[0.005,2540.156]..[0.008,2648.286],
rows=10000000)"
"                  -> Hash (cost=76906.34..76906.34 rows=10000048
width=4) (actual time=[4099.909,4099.909]..[4115.587,4115.587], rows=10000000)"
                        Max Buckets: 131072 Max Batches: 16 Max Memory
Usage: 1369kB
                        Min Buckets: 131072 Min Batches: 16 Min Memory
Usage: 1369kB
"                        -> Streaming(type: LOCAL REDISTRIBUTE dop:
16/16) (cost=0.00..76906.34 rows=10000048 width=4) (actual time=
[0.077,2558.128]..[179.766,2732.366], rows=10000000)"
"                        -> Seq Scan on tb_join t2
                        (cost=0.00..24015.53 rows=10000048 width=4) (actual time=[0.134,2014.841]..
[1.152,2109.835], rows=10000000)"
Total runtime: 17284.136 ms
```

实际用时：17284.136 ms

## PostgreSQL的执行如下

```
Finalize GroupAggregate (cost=224732445.98..224732873.88 rows=200 width=12)
(actual time=23038.091..28648.779 rows=100 loops=1)
  Group Key: t1.c1
  -> Gather Merge (cost=224732445.98..224732855.88 rows=3200 width=12) (actual
time=23038.055..28648.693 rows=800 loops=1)
    workers Planned: 16
```

```

Workers Launched: 7
-> Sort (cost=224731445.63..224731446.13 rows=200 width=12) (actual
time=22852.345..22852.375 rows=100 loops=8)
    Sort Key: t1.c1
    Sort Method: quicksort  Memory: 29kB
    Worker 0:  Sort Method: quicksort  Memory: 29kB
    Worker 1:  Sort Method: quicksort  Memory: 29kB
    Worker 2:  Sort Method: quicksort  Memory: 29kB
    Worker 3:  Sort Method: quicksort  Memory: 29kB
    Worker 4:  Sort Method: quicksort  Memory: 29kB
    Worker 5:  Sort Method: quicksort  Memory: 29kB
    Worker 6:  Sort Method: quicksort  Memory: 29kB
-> Partial HashAggregate (cost=224731435.98..224731437.98
rows=200 width=12) (actual time=22852.274..22852.310 rows=100 loops=8)
    Group Key: t1.c1
    Batches: 1  Memory Usage: 48kB
    Worker 0:  Batches: 1  Memory Usage: 48kB
    Worker 1:  Batches: 1  Memory Usage: 48kB
    Worker 2:  Batches: 1  Memory Usage: 48kB
    Worker 3:  Batches: 1  Memory Usage: 48kB
    Worker 4:  Batches: 1  Memory Usage: 48kB
    Worker 5:  Batches: 1  Memory Usage: 48kB
    Worker 6:  Batches: 1  Memory Usage: 48kB
-> Parallel Hash Join (cost=60752.57..68479935.98
rows=31250300001 width=4) (actual time=18698.276..22742.421 rows=1250000 loops=8)
    Hash Cond: (t1.id = t2.id)
-> Parallel Seq Scan on tb_join t1
(cost=0.00..50498.03 rows=625003 width=8) (actual time=1.113..1251.532
rows=1250000 loops=8)
-> Parallel Hash (cost=50498.03..50498.03 rows=625003
width=4) (actual time=9063.024..9063.024 rows=1250000 loops=8)
    Buckets: 131072  Batches: 256  Memory Usage:
2656kB
-> Parallel Seq Scan on tb_join t2
(cost=0.00..50498.03 rows=625003 width=4) (actual time=86.099..1537.025
rows=1250000 loops=8)
Planning Time: 2.918 ms
JIT:
    Functions: 115
" Options: Inlining true, Optimization true, Expressions true, Deforming true"
" Timing: Generation 16.294 ms, Inlining 278.393 ms, Optimization 229.166 ms,
Emission 170.155 ms, Total 694.008 ms"
Execution Time: 28663.835 ms

```

**实际用时: 28663.835 ms**

## 结果分析

从结果上来看, openGuass的成绩非常亮眼, 相较于postgres整整提升了约**11ms**。

两者都使用了Hash join, 执行计划也相差不多, 先并行扫描, 然后 Parallel Hash, Parallel Hash Join, Partial HashAggregate, 接着排序, 合并, 并最终聚合成结果。

openGauss与postgres最大的不同在于，在 **Hash Join** 中，openGauss的哈希表是通过**并行重分布**来构建的，这些表的**大小、哈希桶数量和内存使用是动态分配的**。

openGauss在这次比拼中最大的亮点在于 **Streaming 操作**，在执行计划中，数据是**分布式**和**并行化**处理的，查询通过“局部重分布”和“本地收集”来分发和聚合数据。openGauss的文档中提到，其采用了SMP并行技术。

openGauss的SMP并行技术是一种利用计算机多核CPU架构来实现多线程并行计算，以充分利用CPU资源来提高查询性能的技术。在复杂查询场景中，单个查询的执行较长，系统并发度低，通过SMP并行执行技术实现算子级的并行，能够有效减少查询执行时间，提升查询性能及资源利用率。SMP并行技术的整体实现思想是对于能够并行的查询算子，将数据分片，启动若干个工作线程分别计算，最后将结果汇总，返回前端。SMP并行执行增加数据交互算子（Stream），实现多个工作线程之间的数据交互，确保查询的正确性，完成整体的查询。

#### ⚠ Caution

SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，包括CPU、内存、I/O等资源的消耗都会出现明显的增长，而且随着并行度的增大，资源消耗也随之增大。

## 使用限制

想要利用SMP提升查询性能需要满足以下条件：

系统的CPU、内存、I/O和网络带宽等资源充足。SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，当上述资源成为瓶颈的情况下，SMP无法提升性能，反而可能导致性能的劣化。在出现资源瓶颈的情况下，建议关闭SMP。

所以说，在我们的“**玩具数据库**”中，这样的操作是被允许的，但是在实际的生产活动中，必须考虑“**数据倾斜**”和“**系统并发度**”造成资源波动对SMP的影响。

**结论是**，在低并发、低I/O等资源充足的情境下，openGauss可以通过SMP架构极大地提高Hash Join的性能，但在出现资源瓶颈的情况下，应该关闭SMP。

这一局，**OpenGauss**胜出。

## 压力测试

## QPS点查性能

### 背景

**QPS**（Queries Per Second）是衡量信息检索系统（例如搜索引擎或数据库）在一秒钟内接收到的搜索流量的一种常见度量。该术语在任何请求-响应系统中都得到更广泛的使用，更正确地称为每秒请求数（RPS：Request Per Second）。

**RT**（Response-time）响应时间，表示执行一个请求从开始到最后收到响应数据所花费的总体时间，即从客户端发起请求到收到服务器响应结果的时间。该请求可以是任何东西，从内存获取，磁盘IO，复杂的数据库查询或加载完整的网页。

暂时忽略传输时间，响应时间是处理时间和等待时间的总和。处理时间是完成请求要求的工作所需的时间，等待时间是请求在被处理之前必须在队列中等待的时间。响应时间是一个系统最重要的指标之一，它的数值大小直接反应了系统的快慢。

总而言之，RT越小，QPS越大，数据库的点查性能越好。

# Baseline测试基准

运行sql脚本： QPS\_config.sql

## openGauss

参数名称	值
shared_buffers	2MB
work_mem	16MB
maintenance_work_mem	128MB
effective_cache_size	8GB
random_page_cost	4
seq_page_cost	1
temp_buffers	8MB
query_dop	4

## postgres:

参数名称	值
shared_buffers	2MB
work_mem	16MB
maintenance_work_mem	128MB
effective_cache_size	8GB
random_page_cost	4
seq_page_cost	1
temp_buffers	8MB
max_parallel_workers	4

## 实验验证

数据源：<https://github.com/dr5hn/countries-states-cities-database>

测试文件：

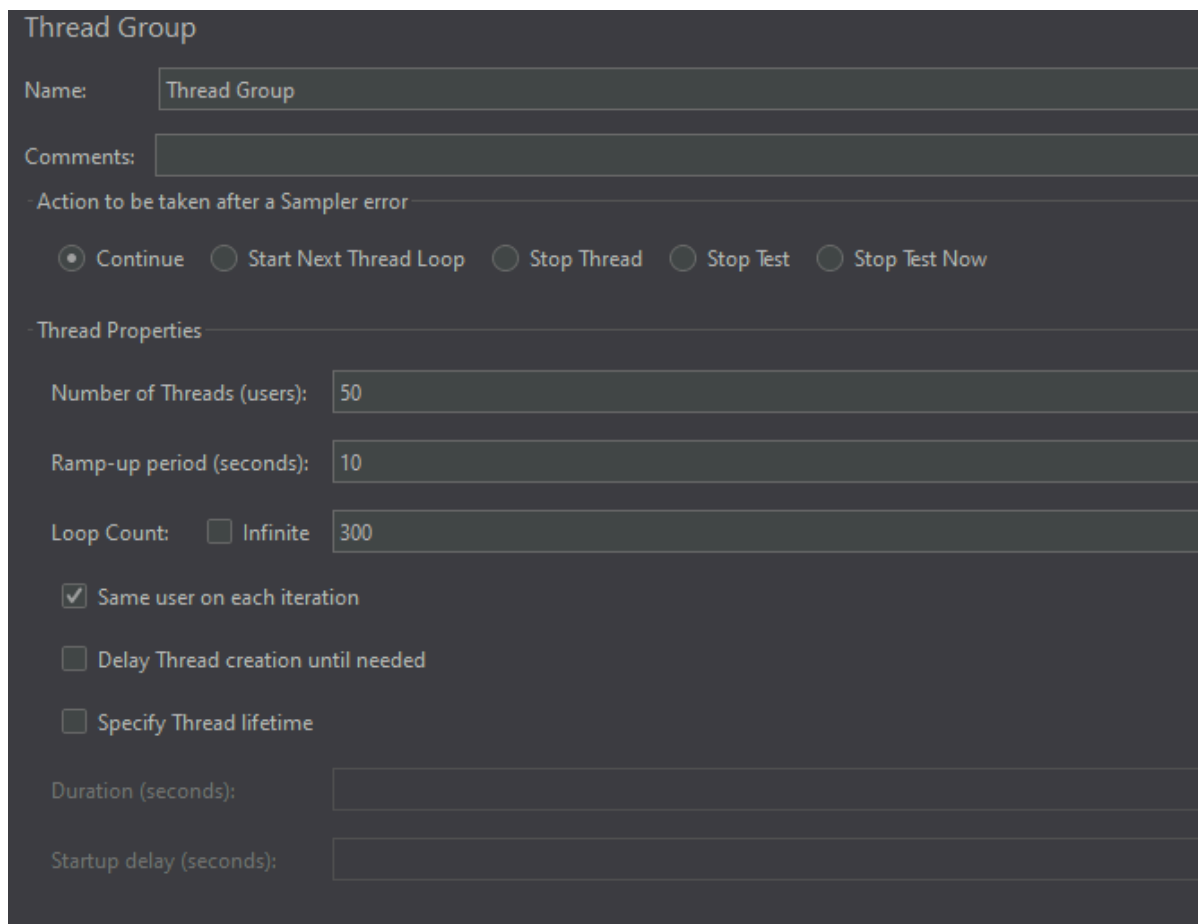
QPS\_test\_postgres.jmx

QPS\_test\_postgres.jmx

数据源是一个150000行的表。

QPS点查需要在**多线程、高并发**的场景下才有意义，在这个实验中，我使用了**JMeter**进行QPS性能测试。

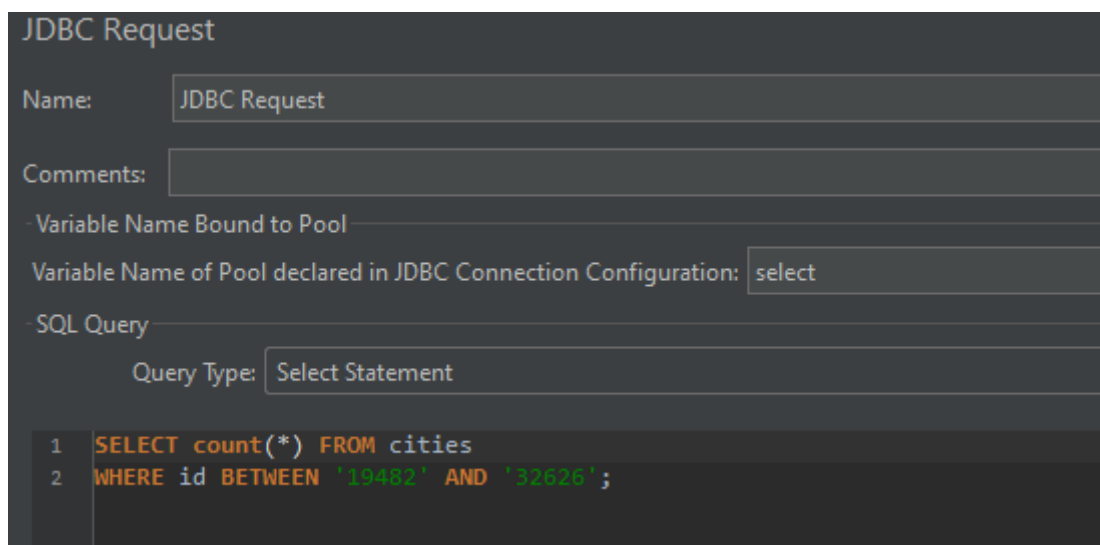
首先配置线程组



The screenshot shows the 'Thread Group' configuration window in JMeter. The 'Name' field is set to 'Thread Group'. The 'Comments' field is empty. Under 'Action to be taken after a Sampler error', the 'Continue' radio button is selected. Under 'Thread Properties', 'Number of Threads (users)' is set to 50, 'Ramp-up period (seconds)' is set to 10, and 'Loop Count' is set to 300 with the 'Infinite' checkbox unchecked. The 'Same user on each iteration' checkbox is checked, while 'Delay Thread creation until needed' and 'Specify Thread lifetime' are unchecked. The 'Duration (seconds)' and 'Startup delay (seconds)' fields are empty.

选择10s之内生成50线程，模仿用户请求。同时，循环300次，以统计平均数据。

接着设置请求样式：



The screenshot shows the 'JDBC Request' configuration window in JMeter. The 'Name' field is set to 'JDBC Request'. The 'Comments' field is empty. Under 'Variable Name Bound to Pool', the 'Variable Name of Pool declared in JDBC Connection Configuration' is set to 'select'. Under 'SQL Query', the 'Query Type' is set to 'Select Statement'. The SQL query is entered as:  
1 SELECT count(\*) FROM cities  
2 WHERE id BETWEEN '19482' AND '32626';

这是一个**只读请求**，因为包含了**主键索引**，因此适合对点查进行测试。如果询问执行器，会获得回应：

postgres:



	QUERY PLAN
1	Aggregate (cost=1563.57..1563.58 rows=1 width=8) (actual time=40.381..40.382 rows=1 loops=1)
2	-> Bitmap Heap Scan on cities (cost=20.15..1561.69 rows=754 width=0) (actual time=5.875..39.946 rows=13017 loops=1)
3	Recheck Cond: ((id >= 19482) AND (id <= 32626))
4	Heap Blocks: exact=250
5	-> Bitmap Index Scan on cities_pkey (cost=0.00..19.96 rows=754 width=0) (actual time=5.464..5.464 rows=13017 loops=1)
6	Index Cond: ((id >= 19482) AND (id <= 32626))
7	Planning Time: 0.107 ms
8	Execution Time: 40.437 ms

opengauss:

	QUERY PLAN
1	Aggregate (cost=1564.35..1564.36 rows=1 width=8) (actual time=50.258..50.258 rows=1 loops=1)
2	-> Bitmap Heap Scan on cities (cost=19.98..1562.47 rows=754 width=0) (actual time=9.405..49.344 rows=13017 loops=1)
3	Recheck Cond: ((id >= 19482) AND (id <= 32626))
4	Heap Blocks: exact=251
5	-> Bitmap Index Scan on cities_pkey (cost=0.00..19.79 rows=754 width=0) (actual time=8.421..8.421 rows=13017 loops=1)
6	Index Cond: ((id >= 19482) AND (id <= 32626))
7	Total runtime: 50.473 ms

最后还需要添加一个用于模拟实际情况下的常数吞吐量计时器，这个定时器保证了吞吐量为预设的吞吐量，与之前不设置相比，可以保证并发更接近为设置的值，从而计算起QPS更加准确

Constant Timer

Name: Constant Timer

Comments:

Thread Delay (in milliseconds): 300

实验结果

点查性能测试										
opengauss	样本量	平均响应时间 (ms)	最小响应时间 (ms)	最大响应时间 (ms)	标准差	错误数	吞吐量 (per sec)	每195th 百分位响应时间	每秒发送地字	平均收到地信息 (kb)
JDBC Request	15000	19	9	139	6.079128674	0	934.5212136	10.95142047	0	12
平均QPS	52.63157895									
平均RT	19									
postgres	样本量	平均响应时间 (ms)	最小响应时间 (ms)	最大响应时间 (ms)	标准差	错误数	吞吐量 (per sec)	每195th 百分位响应时间	每秒发送地字	平均收到地信息 (kb)
JDBC Request	15000	44	8	206	8.375223935	0	128.3982743	1.504667277	0	12
平均QPS	22.72727273									
平均RT	44									

根据QPS的计算公式

QPS = 1000ms / RT

我们得到两者的QPS为

openGauss的平均QPS为**52.6316**。

平均RT为，postgres的平均QPS为**22.7273**。

## 结果分析

在该实验中，我们在10秒内生成50个线程执行同一条sql语句，平均每秒5个用户访问，我们循环执行300次，共产生了15000的访问量，得到了15000次访问的平均访问时间。

在样本量为**15000**的情况下，openGauss做到了高达**52.6316**的平均QPS，相较于postgres整整高出两倍。

openGauss胜出。

## Specialties

### Delete和Update

#### 死元组和膨胀表问题

进行删除之后，postgre的磁盘管理机制会带来**死元组问题**。

在Postgresql做delete操作时，数据集(也叫做元组 (tuples))是没有立即从数据文件中移除的，仅仅是通过在行头部设置xmax做一个**删除标记**。update操作也是一样的，在postgresql中可以看作是**先delete再insert**。

这是Postgresql **MVCC**的基本思想之一，因为它允许在不同进程之间只进行最小的锁定就可以实现更大的并发性。这个MVCC实现的缺点当然是它会留下被标记删除的元组（dead tuples），也就是我们所说的“**死元组问题**”，即使在这些版本的所有事务完成之后。

这样的实现有其自然的考量，譬如，我们可以轻松地回滚数据。但同时，这样会造成大量**磁盘空间的浪费**。

在postgre中有办法**彻底清除数据**吗？

有，使用 `vacuum` 命令就可以做到这一点，`VACUUM` 的会移除表和索引中的死元组并将该空间标记为可在未来重用。PostgreSQL有一个可选的特性 `autovacuum`，它的目的是自动执行 `VACUUM` 和 `ANALYZE` 命令。当它被启用时，自动清理会检查被大量插入、更新或删除元组的表。

然而普通的VACUUM不能解决表膨胀的问题。它将不会把这些清除掉的空间交还给操作系统，除非在特殊的情况中表尾部的一个或多个页面变成完全空闲并且能够很容易地得到一个排他表锁。因此，被删除过的这张表会留下很多“空洞”，并引发表膨胀的问题。

有什么东西可以解决**表膨胀**的问题吗？

【3】中提到了 `VACUUM FULL` 命令，它可以回收这些空间。它将旧表文件中的活元组复制到新表中，通过重写整张表的方式将表压实。这将最小化表的尺寸，但是要花较长的时间。它也需要额外的磁盘空间用于表的新副本，直到操作完成。

但在实际生产中，因为该操作会持有表上的 `AccessExclusiveLock`，阻塞业务正常访问，因此在不间断服务的情况下并不适用。在现实的生产环境中，这需要通过长时间的停机来实现。

也有一些第三方插件尝试解决这个问题，譬如pg\_repack能够在线上业务正常进行的同时进行无锁的 `VACUUM FULL`。但是postgres本身没有更好的实现。

`VACUUM` 会产生大量I/O流量，这将导致其他活动会话性能变差。而 `VACUUM FULL` 会阻塞业务正常访问，总之没有非常好的实现。

## Baseline

与QPS性能测试中的Baseline相同

## 实验验证

脚本文件： `autovacuum.sql`

在这个实验中，我是用一个运行10min的脚本，并让两个数据库间隔1min就会进行一次 `autovacuum` 操作。在实际的生产中，这样的操作是不可取的。**该实验只是基于一个“微缩模型”，来对两个数据库的 `vacuum` 操作得到更深刻的认知。**

运行以下命令启用数据库的 `autovacuum` 功能，开启一个后台进程，定期清理死元组。

```
ALTER SYSTEM SET autovacuum TO on;
```

设置 `autovacuum` 的时间间隔为1 min，设置触发 `VACUUM` 或 `ANALYZE` 操作的死元组数量阈值为 50。当表中有 50 个或更多的死元组时，会触发 `autovacuum` 操作。

```
sql复制代码-- 设置 Autovacuum 检查间隔时间为 1 分钟
ALTER SYSTEM SET autovacuum_naptime TO '1min';

-- 设置触发 VACUUM 操作的死元组数量阈值
ALTER SYSTEM SET autovacuum_vacuum_threshold TO 50;
ALTER SYSTEM SET autovacuum_analyze_threshold TO 50;
```

然后执行此命令以重新加载数据库配置，使得新的配置立即生效。

```
sql复制代码-- 重新加载配置
SELECT pg_reload_conf();
```

接下来开始运行一个 10 min 的脚本。

这个部分的脚本通过批量更新 (`UPDATE`) 和删除 (`DELETE`) 操作，模拟高负载的工作负载。它运行 10 分钟，每次执行批量更新和删除操作后，会随机睡眠一段时间。通过这些操作，能够生成大量的死元组，从而触发 `Autovacuum` 进程来进行清理。

```
sql复制代码-- 设置脚本执行时间为 10 分钟
DO $$
DECLARE
    start_time TIMESTAMP := clock_timestamp(); -- 使用 clock_timestamp() 获取精确到微秒的时间
    end_time TIMESTAMP := start_time + INTERVAL '10 minutes'; -- 执行时间为 10 分钟
    op_start_time TIMESTAMP;
    op_end_time TIMESTAMP;
    op_duration INTERVAL;
    sleep_time INTERVAL;
BEGIN
    -- 循环直到脚本运行 10 分钟
    WHILE clock_timestamp() < end_time LOOP
```

```

-- 执行批量 UPDATE 操作
UPDATE cities
SET state_name = 'UpdatedState'
WHERE id IN (
    SELECT id FROM cities TABLESAMPLE SYSTEM (10) LIMIT 10000
);

-- 执行批量 DELETE 操作
DELETE FROM cities
WHERE id IN (
    SELECT id FROM cities TABLESAMPLE SYSTEM (10) LIMIT 4000
);

-- 随机睡眠一段时间后执行下一轮
sleep_time := (RANDOM() * 20 + 20) * INTERVAL '1 second';
PERFORM pg_sleep(EXTRACT(epoch FROM sleep_time));
END LOOP;
END $$;

```

## openGauss

	UPDATE Operation Duration (sec)	DELETE Operation Duration(sec)
	3.105316	0.129364
	3.551659	0.348598
	3.524617	0.412924
	3.927983	0.371847
	4.258932	0.265402
	4.202194	0.345595
	4.297078	0.471004
	3.94928	0.363444
	4.319723	0.330684
	4.322292	0.394895
	4.469608	0.649284
	4.177168	0.728595
	4.240142	0.855425
	5.155844	0.944257
	4.109183	0.817043
	5.048348	0.846472

	UPDATE Operation Duration (sec)	DELETE Operation Duration(sec)
平均	4.166210438	0.517177063

## postgres

	UPDATE Operation Duration(sec)	DELETE Operation Duration(sec)
	1.596563	0.06544
	2.14952	0.066574
	2.215008	0.065538
	2.277074	0.103702
	2.124946	0.097389
	2.49997	0.134271
	3.493015	0.148004
	2.870567	0.278882
	3.193673	0.219556
	2.70322	0.198941
	2.667977	0.432949
	2.566228	0.468688
	2.814681	0.487761
	3.013833	0.516016
	2.821225	0.57918
	2.840251	0.588937
	2.213131	0.665421
	3.040824	0.643148
	3.006824	0.640411
平均	2.637291053	0.336884632

可以看到，postgres的 UPDATE 平均用时比opengauss快1.5s左右，DELETE 快0.2s左右。

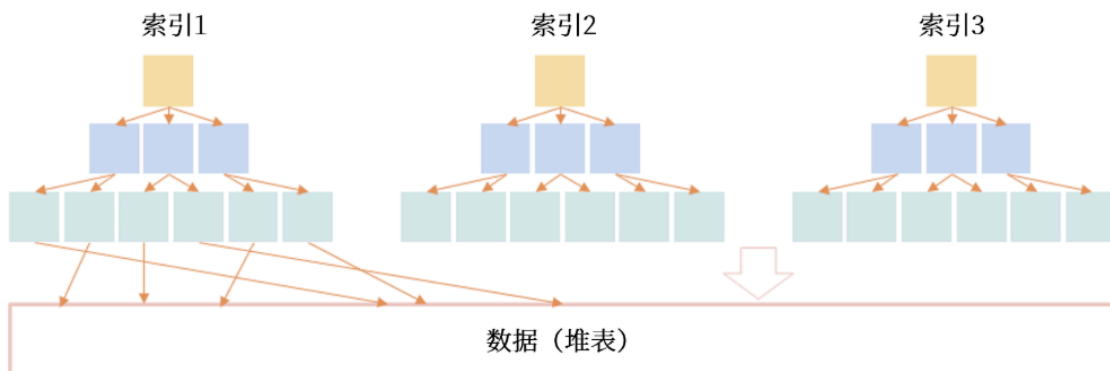
postgres胜出。

# 索引组织表问题

## 背景：什么是堆表和索引组织表？

在proj1中，我了解到，数据库不仅存储数据本身，还会存储数据的索引。所以，将索引和数据分别单独存放，还是将两者一起存放引出了两种数据的组织方式，即**堆表 (HOT)** 和**索引组织表(IOT)**。postgres使用堆表存储数据，索引和表数据是分开存储的，索引中存储了**数据行的指针**，当使用普通索引查找数据时，需要先**扫描索引树**，找到对应的**行指针**，再去表中找到相应的tuple。

**堆表中的数据无序存放**，数据的排序完全依赖于索引（Oracle、Microsoft SQL Server、PostgreSQL 早期默认支持的数据存储都是堆表结构）。



在讲**索引组织表**之前，先看下pg中index scan和index only scan：

**Index Scan**：也即普通索引扫描，对于给定的查询，我们先扫描一遍索引，从索引中找到符合要求的记录的位置(指针)，再定位到表中具体的Page去取。等于是两次I/O，先走索引，再取表记录。

**Index only scan**：建立index时，所包含的字段集合，囊括了我们查询的字段，这样就只需在索引中取数据，就不必访问表了。

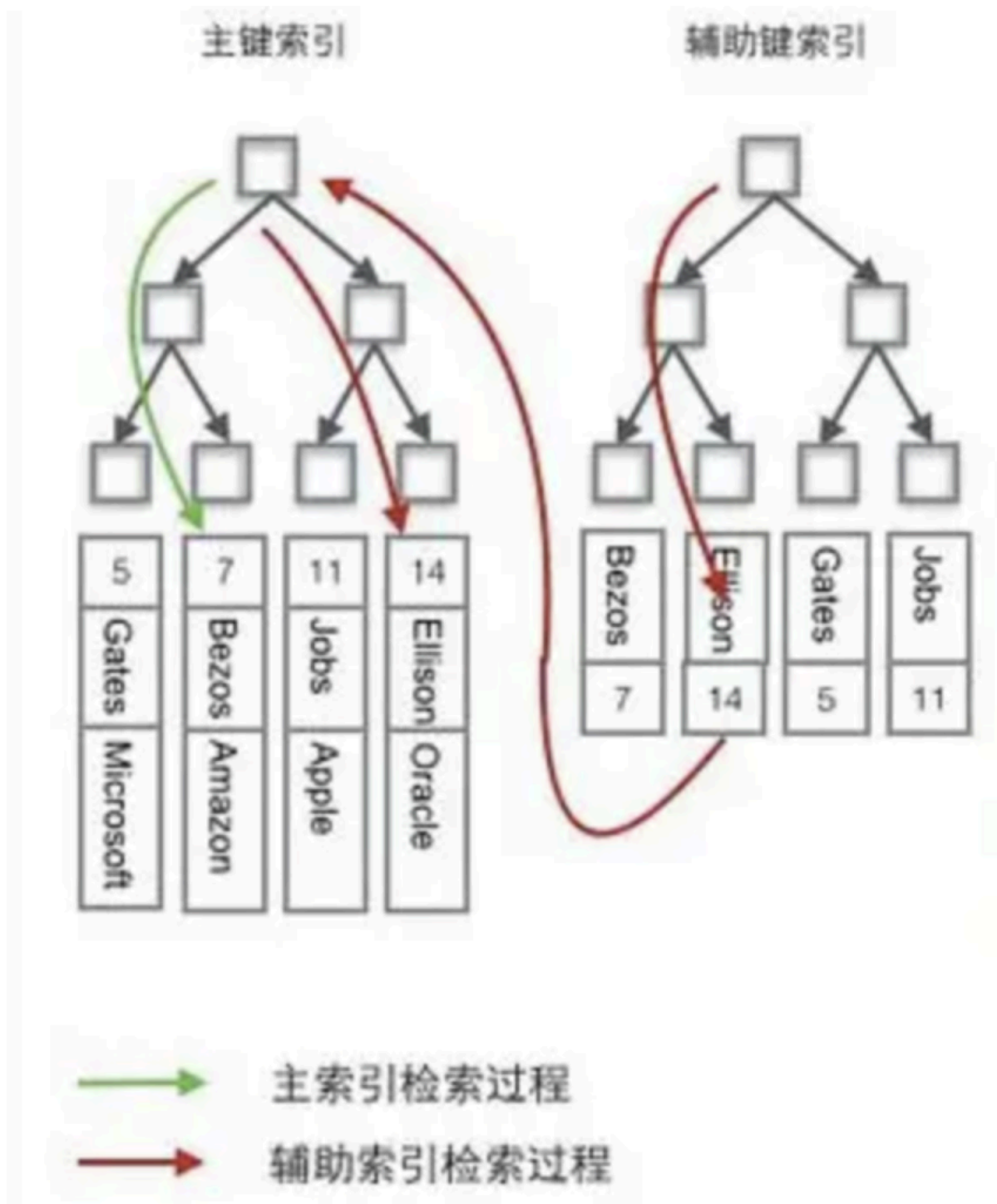
[4]

**index only scan** 可以极大的提高性能。因为不需要再去表中查找数据了。

**Index only scan** 依靠存储在索引中的**冗余数据**，消除了去访问堆表的操作。如果我们将这个概念进一步扩大，并将所有列放在索引中，我们还需要堆表吗？

这也就引出了索引组织表的概念，索引组织表的数据是按照主键顺序被存储到一个**B+树索引**中的，索引就是数据，数据就是索引，二者合二为一。当使用主键去查询一个索引组织表时，不需要再访问表，能从索引中获取到表的全部数据。这也是mysql中的**聚簇索引**的概念，数据行存储在索引的叶子节点中。在mysql中除了聚簇索引外，还有非聚簇索引(也叫二级索引)。非聚簇索引索引它的叶子节点存的是键值和主键值。

下图为MySQL使用的InnoDB引擎的索引组织表：



不难看出，因为堆表**无序存放**，所以 `INSERT` 比较快，而**索引组织表**有序存放，一节约了磁盘空间，二降低了IO，提高了查询的性能。尤其是当我们的数据几乎总是通过主键来进行搜索时，查询效率的提升将会很显著，但是当索引组织表上有二级索引并且频繁使用二级索引进行访问时，它的缺点也很明显：二级索引需要回表，它的效率要比堆表直接使用行指针访问数据的效率要低。

postgres中**不支持索引组织表，只支持堆表**。openGauss也不支持索引组织表，但是，postgres和opengauss中都提供了 `CLUSTER` 操作。

这个功能与索引组织表的主要功能类似：**数据按照索引顺序进行有序存储。**

使用 `CLUSTER` 操作可以根据表的索引对数据进行**聚簇排序**，将数据基于**索引**信息进行物理存储。但在openGauss和postgres中，这种聚簇排序操作是一次性的，即：当表被更新之后，更改的内容不会自动被聚簇排序后存储。也就是说，系统不会试图按照索引顺序对新的存储内容及更新记录进行重新聚簇排序存储。

通过网络检索，我发现：

CLUSTER背后的代码与VACUUM (FULL)相同，只是增加了一个排序。因此，CLUSTER存在和VACUUM (FULL)一样的问题：

1. CLUSTER以ACCESS EXCLUSIVE模式锁定表，锁定期间阻塞所有操作  
需要二倍于表的空间进行操作
2. 此外，CLUSTER建立的顺序不会持续保持:随后的insert和update不会遵守这个顺序，并且相关性会随着时间的推移而“失效”。因此，如果希望索引范围扫描保持快速，就必须定期对表进行cluster操作

DELETE不会破坏行顺序，但是也无法阻止insert打乱相关性。然而，PostgreSQL有一个特性可以防止UPDATE破坏表顺序:HOT更新。

HOT更新将新版本的行放在与旧版本相同的8kB块中，并且它不修改任何索引列，因此它极小干扰索引列的相关性。因此，在HOT更新之后，索引列上的索引范围扫描将保持高效!

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。  
上原文出处链接和本声明。

原文链接: <https://blog.csdn.net/dazuiba008/article/details/131422606>

## 实验探究

运行脚本: `cluster.sql`

在这个脚本中我进行了如下操作

```
-- 创建一个不记录日志的表 clu
DROP TABLE clu;
CREATE UNLOGGED TABLE clu (
    id bigint NOT NULL,
    key integer NOT NULL,
    val integer NOT NULL
) WITH (
    autovacuum_vacuum_cost_delay = 0,    -- 设置 autovacuum 的延迟
    fillfactor = 50                      -- 设置 fillfactor，控制表的填充度
);

-- 向 clu 表插入 1 到 100000 之间的数字，并计算 key 的值
INSERT INTO clu (id, key, val)
SELECT i, hashint4(i), 0
FROM generate_series(1, 100000) AS i;

-- 创建一个索引 clu_idx 在 key 列上
CREATE INDEX clu_idx ON clu(key);

-- 使用 clu_idx 对 clu 表进行聚簇操作
CLUSTER clu USING clu_idx;
```

其中 `fillfactor` 表示填充系数，范围10~100，控制表的填充度。这个参数会影响cluster操作的难度。

以下是不同参数下postgres和opengauss的表现



## opengauss

fillfactor	cluster 用时
100	5 s 109 ms
90	5 s 892 ms
70	6 s 849 ms
50	8 s 258 ms
30	8 s 551 ms
10	2 s 262 ms

## postgres

fillfactor	cluster 用时
100	4 s 849 ms
90	5 s 579 ms
70	6 s 344 ms
50	7 s 972 ms
30	8 s 659 ms
10	1 s 520 ms

需要注意的是，随着fillfactor增大，一方面表的紧凑增加，这回提高 CLUSTER 的速度，但同时数据量的增大，又会拖慢 CLUSTER 的速度。

## 写在最后

本文对两个数据库的比较基于本地环境，且笔者没有对两个数据库进行调优，只是保证两者在Baseline接近的情形下进行对比测试。文档的写作目的在于通过课程项目学习更多数据库的知识，了解更多技术内幕。

本文所使用的数据源、使用的sql脚本、得到的数据结果，都放置于GitHub仓库中：[https://github.com/OptimistiCompound/proj3\\_repo](https://github.com/OptimistiCompound/proj3_repo)

## Reference