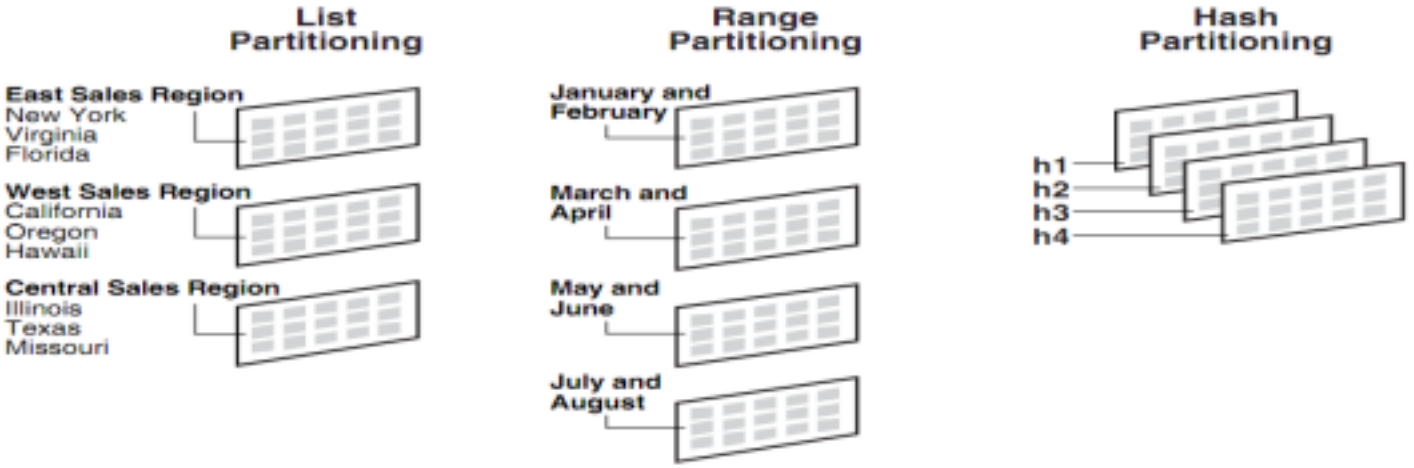
PostgreSQL 表分区实践分享

许中清 @Huawei
jonathan.shv@gmail.com

- **表分区是RDBMS最重要的特性之一**
- PG用继承表实现“表分区”
- PG实现原生表分区：三种方案
- 实践中的关键问题

表分区

分而治之：表分区就是把一张大表的数据根据一定策略拆分成若干个子表

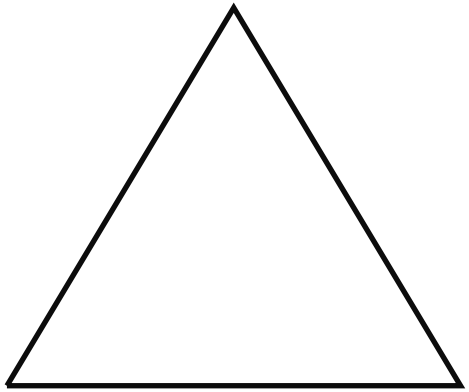



```
CREATE TABLE op_log(
        log_no          bigint,
        creation_time   date,
        created_by      text,
        content         text)
PARTITION  BY RANGE(creation_time)
(   PARTITION p1 VALUES LESS THAN (DATE'2014-07-01'),
    PARTITION p2 VALUES LESS THAN (DATE'2015-01-01'),
    PARTITION p3 VALUES LESS THAN (DATE'2015-07-01') ,
    PARTITION p4 VALUES LESS THAN (MAXVALUE)
);
```

分而治之：解决复杂问题的方法论

- 提高数据聚集度（按分区键）
- 利于多核能力，并行处理，提升分析类业务性能

性能



易管理

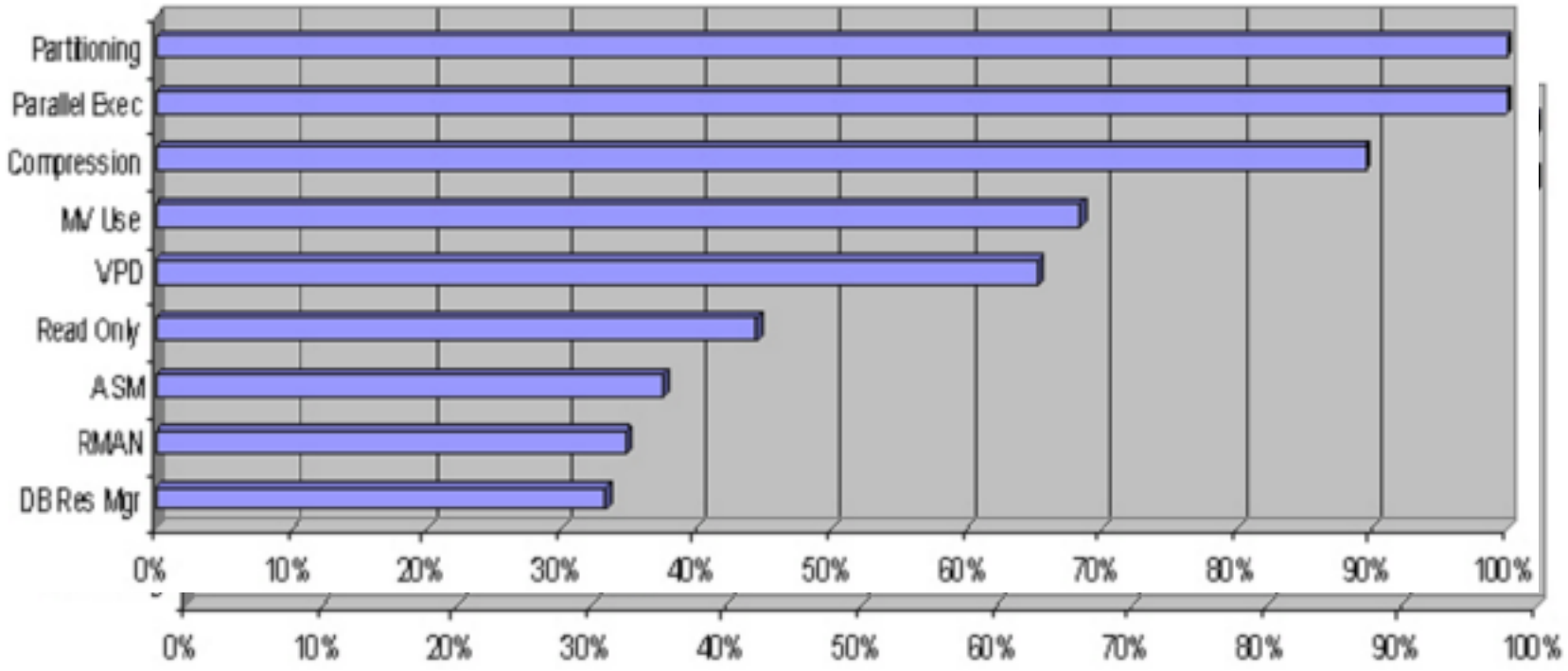
- 表的维护从操作一个大表变成操作多个子表
- 冷热数据差异化管理操作。

可用性

- 表的维护与读写产生冲突
- 表分区缩短维护时间窗，提供可用性

重要：表分区特性广泛应用

- 在数据仓库领域，表分区是最常使用的特性之一



- 表分区是RDBMS最重要的特性之一
- **PG用继承表实现“表分区”**
- PG实现原生表分区：三种方案
- 实践中的关键问题

PG如何处理继承表

```
CREATE TABLE join_table(c1 int,c2 int);
CREATE TABLE parent (c1 int,c2 int);
CREATE TABLE child1(c3 int) inherits (parent);
CREATE TABLE child2(c4 int) inherits (parent);
```

INSERT：子表不继承

INSERT INTO parent **VALUES**(10,10);

```
------------------------------------------------------
 Insert on parent  (cost=0.00..0.01 rows=1 width=0)
   ->  Result  (cost=0.00..0.01 rows=1 width=0)
```

SELECT：子表在底层展开

SELECT * **FROM** parent p, join_table j **WHERE** p.c2=j.c2;

```
------------------------------------------------------------------------
Merge Join  (cost=457.34..1125.13 rows=43806 width=16)
  Merge Cond: (j.c2 = p.c2)
  ->  Sort  (cost=149.78..155.13 rows=2140 width=8)
        Sort Key: j.c2
        ->  Seq Scan on join_table j  (cost=0.00..31.40 rows=2140 width=8)
  ->  Sort  (cost=307.57..317.80 rows=4094 width=8)
        Sort Key: p.c2
        ->  Append  (cost=0.00..61.94 rows=4094 width=8)
              ->  Seq Scan on parent p  (cost=0.00..3.14 rows=214 width=8)
              ->  Seq Scan on child1 p  (cost=0.00..29.40 rows=1940 width=8)
              ->  Seq Scan on child2 p  (cost=0.00..29.40 rows=1940 width=8)
```

PG如何处理继承表(2)

UPDATE/DELETE: 执行计划从顶层展开所有子表

UPDATE parent p **SET** c1=j.c1
FROM join_table j **WHERE** p.c2=j.c2;

```
Update on parent p  (cost=5.82..1334.02 rows=43806 width=24)
  ->  Hash Join  (cost=5.82..121.64 rows=2290 width=20)
        Hash Cond: (j.c2 = p.c2)
        ->  Seq Scan on join_table j  (cost=0.00..31.40 rows=2140 width=14)
        ->  Hash  (cost=3.14..3.14 rows=214 width=10)
              ->  Seq Scan on parent p  (cost=0.00..3.14 rows=214 width=10)
  ->  Merge Join  (cost=285.12..606.19 rows=20758 width=24)
        Merge Cond: (p.c2 = j.c2)
        ->  Sort  (cost=135.34..140.19 rows=1940 width=14)
              Sort Key: p.c2
              ->  Seq Scan on child1 p  (cost=0.00..29.40 rows=1940 width=14)
        ->  Sort  (cost=149.78..155.13 rows=2140 width=14)
              Sort Key: j.c2
              ->  Seq Scan on join_table j  (cost=0.00..31.40 rows=2140 width=14)
  ->  Merge Join  (cost=285.12..606.19 rows=20758 width=24)
        Merge Cond: (p.c2 = j.c2)
        ->  Sort  (cost=135.34..140.19 rows=1940 width=14)
              Sort Key: p.c2
              ->  Seq Scan on child2 p  (cost=0.00..29.40 rows=1940 width=14)
        ->  Sort  (cost=149.78..155.13 rows=2140 width=14)
              Sort Key: j.c2
              ->  Seq Scan on join_table j  (cost=0.00..31.40 rows=2140 width=14)
```

PG使用表继承可实现表分区

创建表分区四步法

op_log

log_no	creation_time	created_by	content

op_log_2012

log_no	creation_time	created_by	content

< DATE'2013-01-01'

op_log_2013

log_no	creation_time	created_by	content

DATE'2013-01-01' < DATE'2014-01-01'

op_log_2014

log_no	creation_time	created_by	content

DATE'2014-01-01' < DATE'2015-01-01'

⑪创建主表：**CREATE TABLE op_log(…);**

⑫创建子表：

```
CREATE TABLE op_log_2013()
                INHERITS(op_log);
```

⑬增加子表约束：

```
ALTER TABLE op_log_2013 ADD
CONSTRAINT log_range  CHECK
(creation_time>DATE'2013-01-01'  AND
creation_time<DATE'2014-01-01');
```

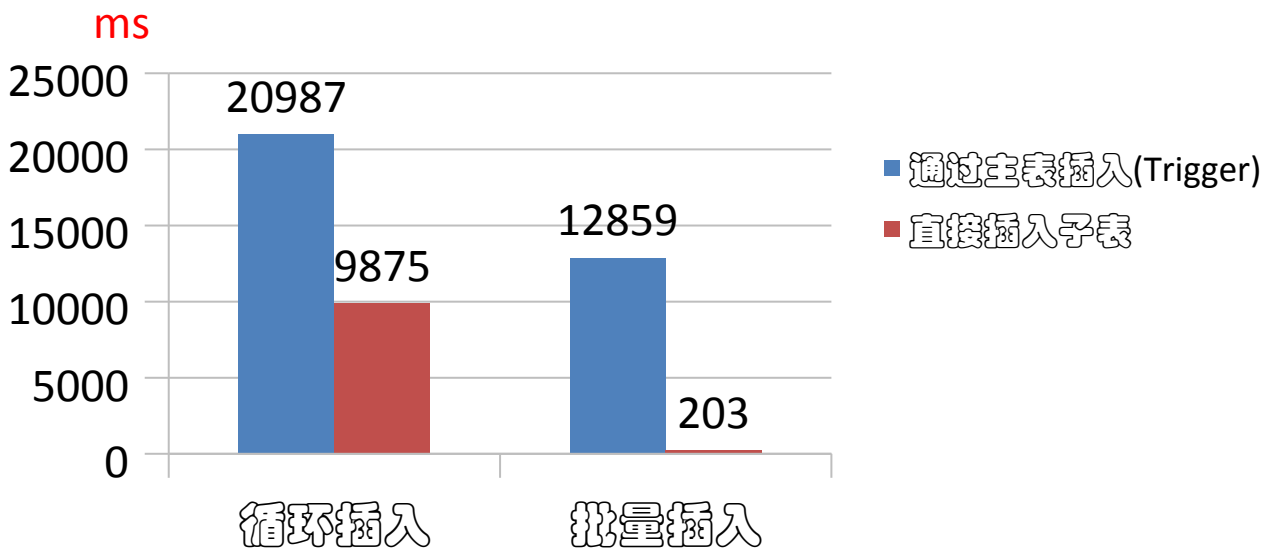
⑭指定路由机制：（Trigger/Rule）

```
CREATE TRIGGER  router_to_2013
BEFORE INSERT ON  log_range
FOR EACH ROW EXECUTE
PROCEDURE …;
```

partition_range.sql

PG“表分区”性能(INSERT)

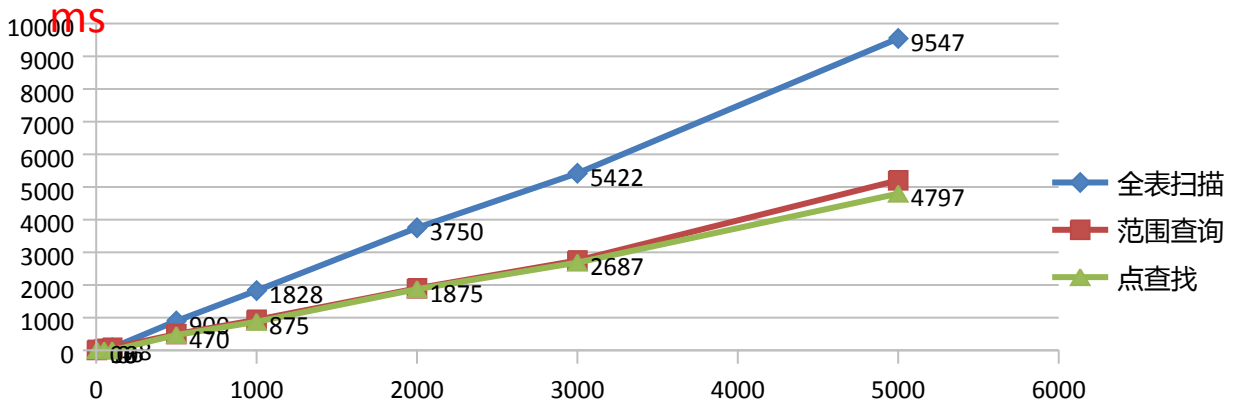
在批量插入的场景下，trigger对性能影响很大



测试环境：5个子表，插入10K条数据，插入的所有数据都在一个分区内
每个tuple的大小16 byte

PG “表分区”性能(SELECT)

SELECT生成执行计划的时间 与 子表数量成线性关系



子表数量	5	50	100	500	1k	2k	3k	5k
全表扫描	16	46	78	900	1,828	3,750	5,422	9,547
范围查询	16	47	78	500	937	1,900	2,750	5,203
点查询				470	875	1,875	2,687	4,797

原因：

对主表的查询会展开成N+1条类似的查询语句。(N为主表的子表数量)
相当于要把同一条SQL语句(主表替换成子表)优化N+1遍

PG“表分区”：管理不易

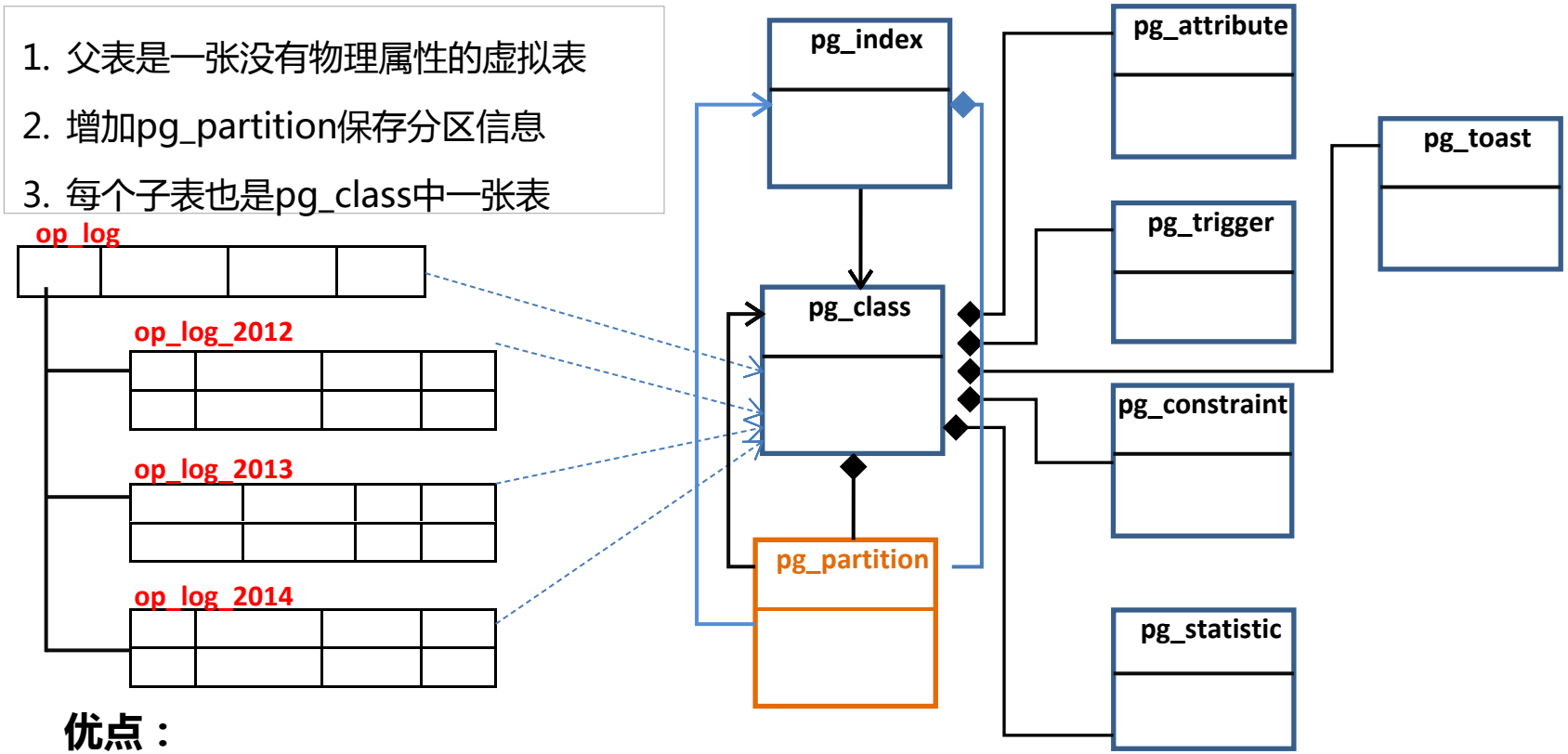
1. **创建分区表：（四步法）**
    1. 步骤复杂(全手工)
    2. 易出错
2. **创建索引：**
    3. 创建N+1个索引
3. **分区表的管理：**
    4. 增加子表，三步。并需要修改主表路由机制

- 表分区是RDBMS最重要的特性之一
- PG用继承表实现“表分区”
- **PG实现原生表分区：三种方案**
- 实践中的关键问题

改造PG：原生支持表分区

改造PG-方案1：在上包一层

1. 父表是一张没有物理属性的虚拟表
2. 增加pg_partition保存分区信息
3. 每个子表也是pg_class中一张表



优点：

1. 普通表、父表、子表id在同一个id空间。元数据缓存空间、存储引擎接口无需改动。
2. 改动小，实现相对容易

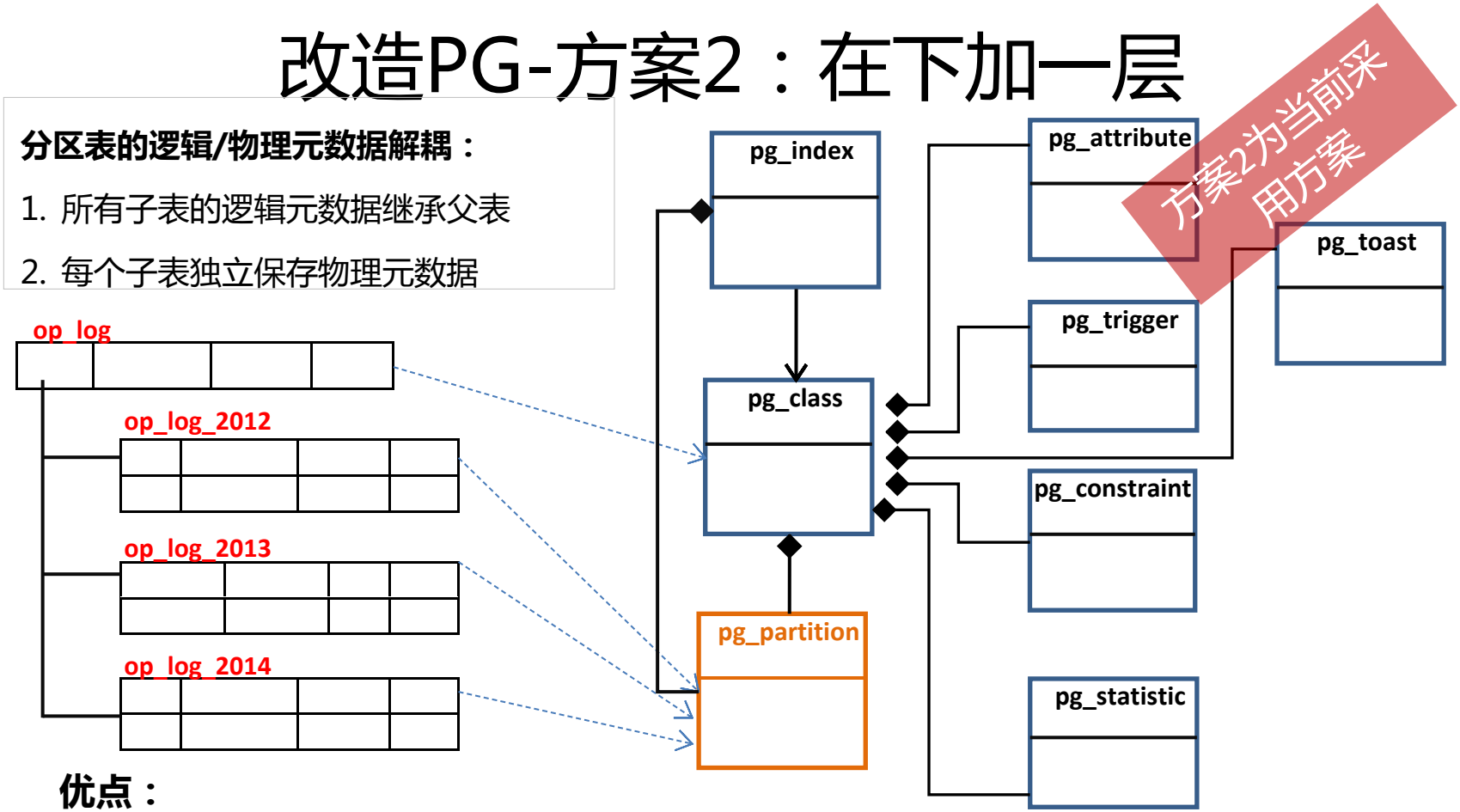
缺点：

3. 每一个子表都有一份元数据，冗余度高，系统资源占用高
4. 表模式修改(DDL)需要更新所有子表元数据，失效消息数量剧增
5. 因为以上原因导致对分区数的限制

改造PG-方案2：在下加一层

分区表的逻辑/物理元数据解耦：

1. 所有子表的逻辑元数据继承父表
2. 每个子表独立保存物理元数据



优点：

1. 元数据缓存大大减小：所有子表的逻辑元数据只需要存一份
2. DDL语句修改更高效
3. 子表数量支持的更多

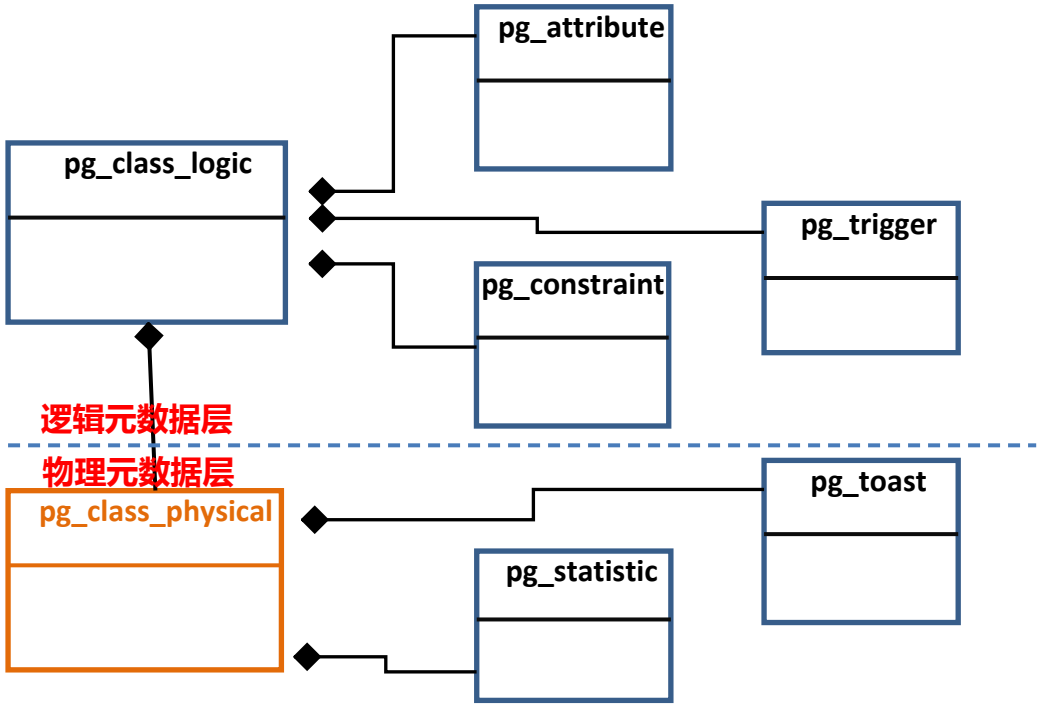
缺点：（复杂度、工作量、内核侵入程度较高）

4. 普通表与子表的id不在一个id空间
5. 元数据缓存机制复杂：子表需要单独的缓存空间
6. 存储引擎接口必须要区分普通表与分区表，并新增处理分支

改造PG-方案3：向下拉一层

所有表的逻辑/物理元数据解耦：

1. 所有表的元数据分逻辑/物理两层，对应将pg_class拆分成pg_class_logic和pg_class_physical两层
2. 普通表是只有一个子表的特殊分区表



缺点：

1. 对PG的程序架构和代码要伤筋动骨的修改

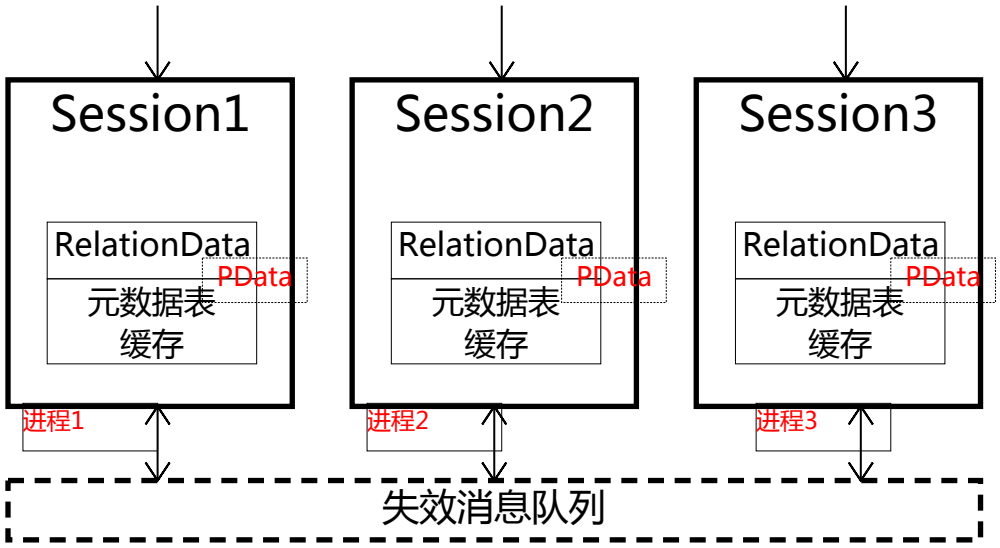
优点：

1. 从逻辑上统一了分区表和普通表。所有的表id都在同一个空间
2. 元数据缓存和失效机制不需要再区分分区表和普通表
3. 存储引擎接口不需要区分分区表和普通表
4. 垃圾回收/统计采样等等都不再需要区分分区表和普通表

- 表分区是RDBMS最重要的特性之一
- PG用继承表实现“表分区”
- PG实现原生表分区：三种方案
- **实践中的关键问题**

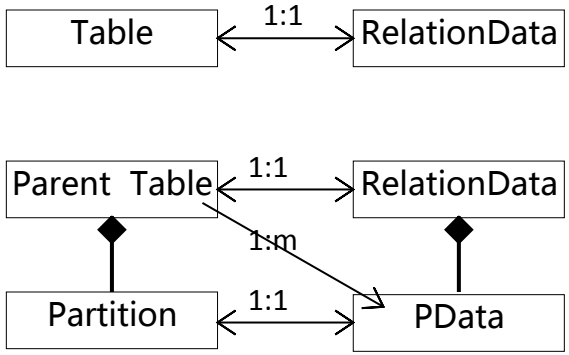
关键问题1：元数据与缓存失效机制

PostgreSQL元数据缓存同步策略



1. 一个RelationData包含一张表的所有表模式信息
2. 一个Session中同一张表的RelationData实例最多只有一个
3. RelationData是从优化器、执行器、到存储引擎等共用的数据结构
4. 收到实效消息后RelationData会重建

分区表的关键问题



1. 子表的数据结构(PData)如何作为参数在系统中传递(改动最小)？难道要改所有的函数接口吗？
2. 接收到表模式失效消息时，如何重建主表和子表的信息？

关键问题2：分区表的统计信息

PostgreSQL统计信息：

```
create table sta_test(c1 int,c2 int);

insert into sta_test values(generate_series(1,10),generate_series(100,200));
insert into sta_test values(generate_series(30,40),generate_series(300,400));
insert into sta_test values(generate_series(60,70),generate_series(600,700));
```

特征值统计信息（c1）：MCV(Most Common Values)

relname name	attname name	stanumbers1 real[]	stavalues1 anyarray
sta_test	c1	{0.03125,0.03125,0.03125,0.0312	{1,2,3,4,5,6,7,8,9,10,30,31,32,33,34,35,3

直方图统计信息（c2）：

relname name	attname name	stavalues1 anyarray
sta_test	c2	{100,103,106,109,112,116,119,122,125,129,132,135,138,142,14

分区表统计信息的关键问题：

1. 采集时，生成主表的统计信息还是每个子表的统计信息？
2. 如果使用主表的统计信息，那么Pruning之后的，据此计算的cost偏差很大(特别是分区键上的cost)
3. 如果使用子表的统计信息，那么如何累加成主表的统计信息？子表太多时，优化器效率如何保证？

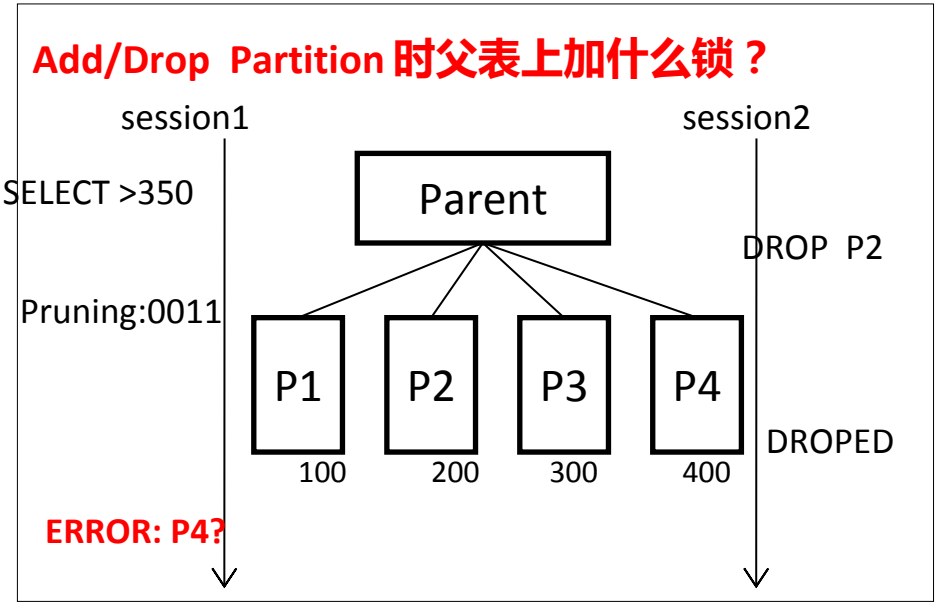
关键问题3：表级锁与并发

PG表级锁冲突矩阵(简化)

Requested Lock Mode	Current Lock Mode: ACCESS SHARE (SELECT)	ROW EXCLUSIVE (I/U/D/)	ACCESS EXCLUSIVE (DDL)
ACCESS SHARE (SELECT)			X
ROW EXCLUSIVE (I/U/D/)			X
ACCESS EXCLUSIVE (DDL)	X	X	X

注:pg有8个级别的表级锁，常见的3个

分区表的关键问题



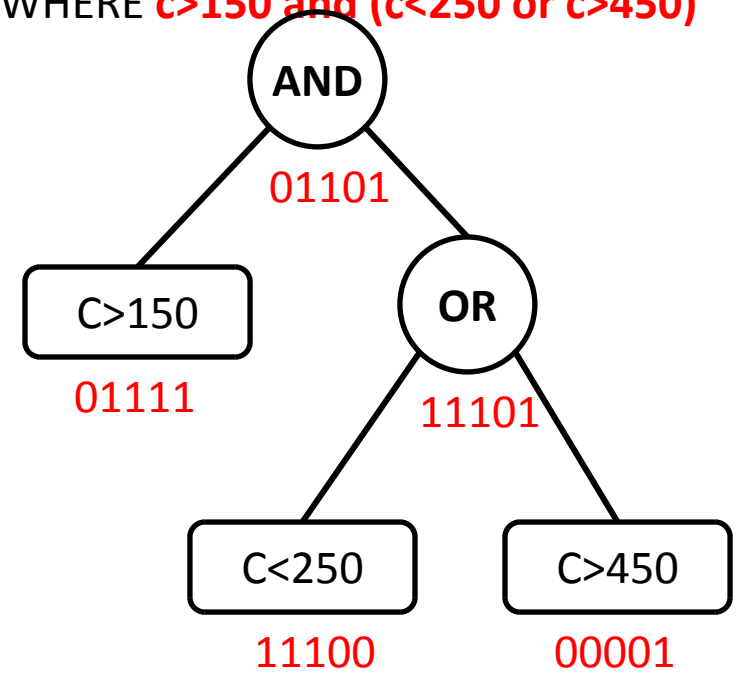
INSERT INTO Interval分区表如何避免为同一个子表创建两个物理文件？

（Interval分区表：向某个子表插入第一条记录时才会为这个子表创建物理文件）

关键问题4：子表剪枝算法

分区键是单列

分区
表：p_table(100,200,300,400,500)
查询: SELECT * FROM ptable
WHERE **c>150 and (c<250 or c>450)**



分区键是多列

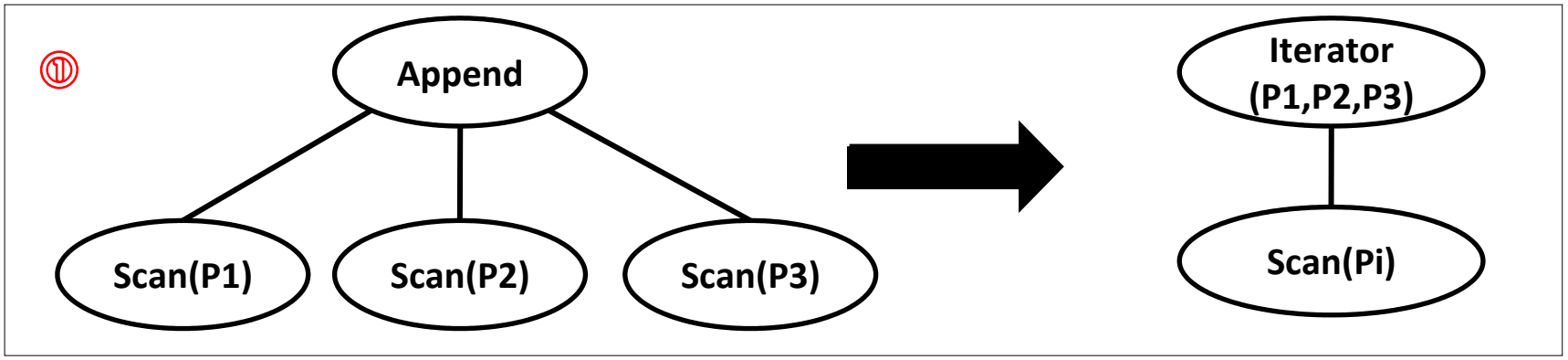
分区键: (c1,c2) **值域**
P1: (100, 100) **{(-∞,100], (-∞,+∞)}**
P2: (100, 200) **{100, [100,200)}**
P3: (100, 300) **{100, [200,300)}**
P4: (200, 200) **{[100,200], (-∞,+∞)}**
P5: (200, 300) **{200, [200,300)}**
P6: (200, 500) **{200, [300,500)}**
P7: (300, 200) **{[200,300], (-∞,+∞)}**
P8: (300, 400) **{[300, [200,400]}**

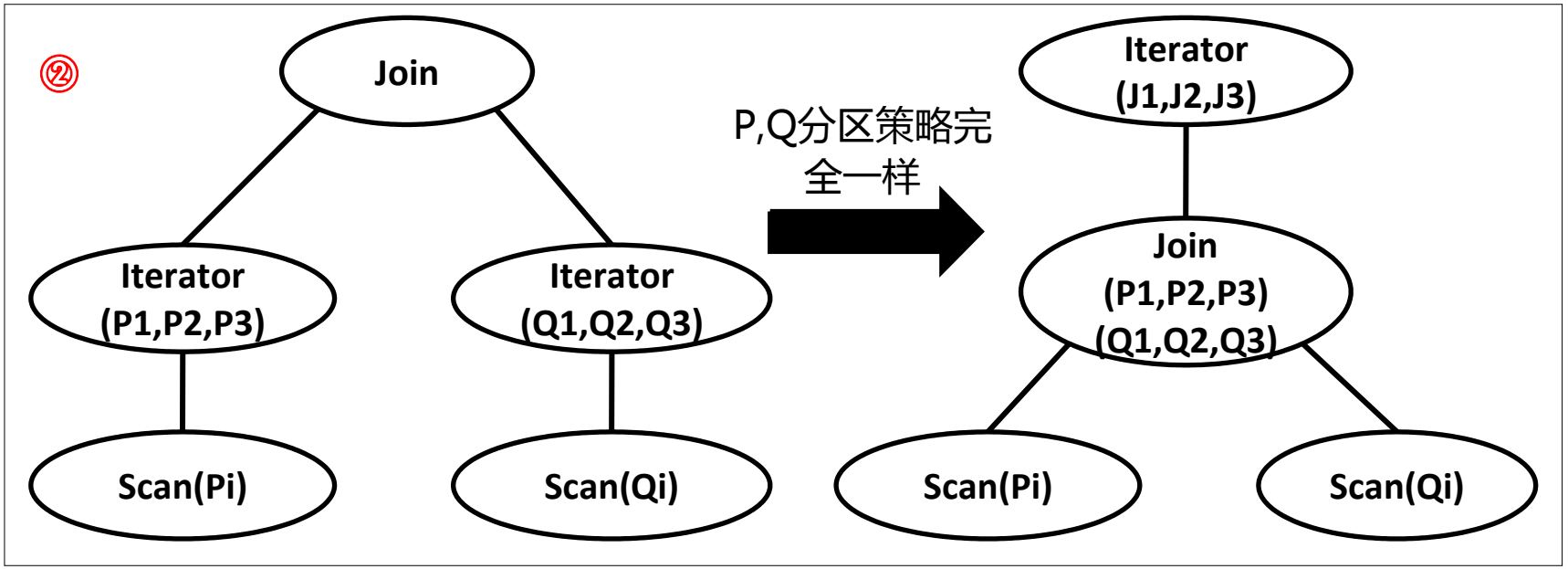
查询: SELECT * FROM ptable
WHERE **c2>350**

?

Pruning结果：P1,P4,P6,P7,P8

关键问题5：算子

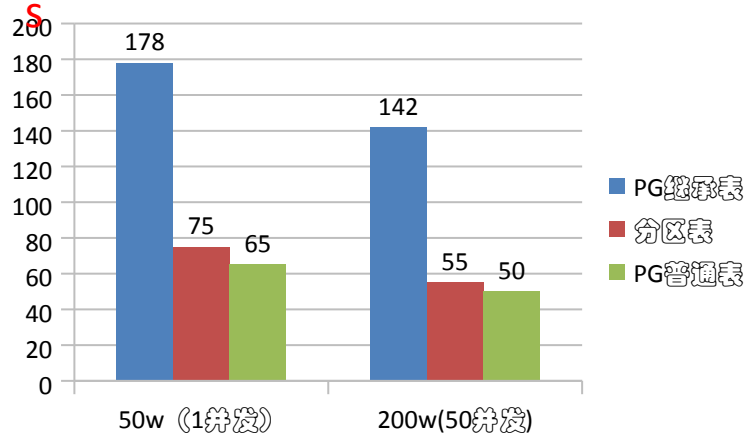

①
Append
Scan(P1)
Scan(P2)
Scan(P3)
Iterator
(P1,P2,P3)
Scan(Pi)


②
Join
Iterator
(P1,P2,P3)
Iterator
(Q1,Q2,Q3)
Scan(Pi)
Scan(Qi)
P,Q分区策略完
全一样
Iterator
(J1,J2,J3)
Join
(P1,P2,P3)
(Q1,Q2,Q3)
Scan(Pi)
Scan(Qi)

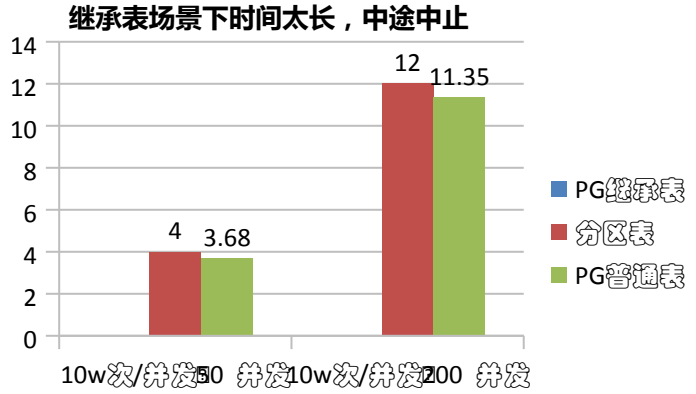
时延的代价

测试环境： **存储：**SAS(15000rps)*2 **计算：**2P12C
数据量 ： 100个子表，每个子表10w条数据，共1000w条数据
测试目的： 主要针对OLTP类场景，验证表分区之后的性能代价

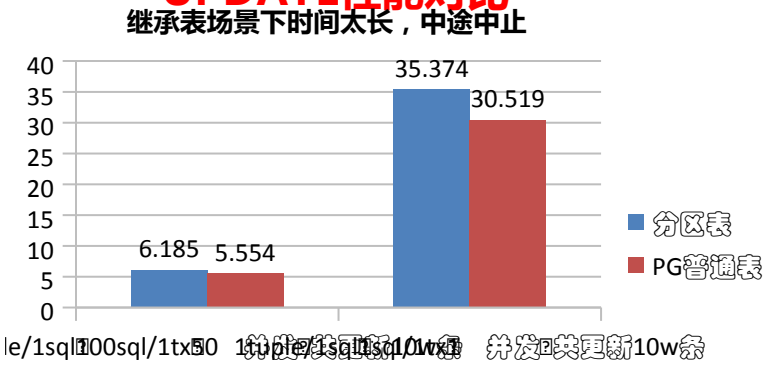
INSERT性能对比



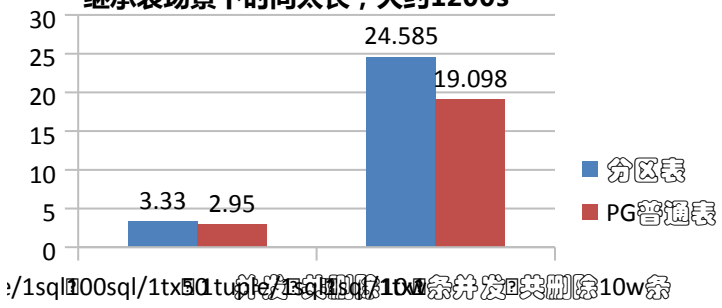
SELECT(Index)性能对比



UPDATE性能对比



DELETE性能对比



谢谢