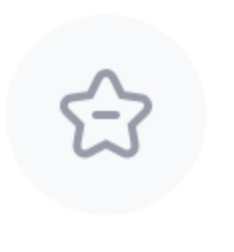
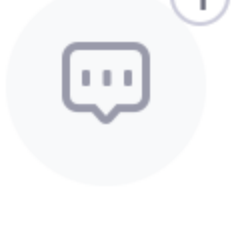

 5

 1

 1



PG vs MySQL mvcc机制实现的异同

原创 进击的CJR 2025-01-17

175

MVCC实现方法比较

MySQL
写新数据时，把旧数据写入回滚段中，其他人读数据时，从回滚段中把旧的数据读出来

PostgreSQL
写新数据时，旧数据不删除，直接插入新数据。

MVCC实现的原理

PG的MVCC实现原理

- 定义多版本的数据——使用元组头部信息的字段来标示元组的版本号
- 定义数据的有效性、可见性、可更新性——通过当前的事务快照和对应元组的版本号判断
- 实现不同的数据库隔离级别——通过在不同时机获取快照实现

PG的数据多版本实现

pg中元组由三部分组成——元组头结点、空值位图、用户数据。每一行元组，都有一个版本号。该版本由如下几个数据组成。

t_xmin：保存插入该元组的事务txid（该元组由哪个事务插入）
t_xmax：保存更新或删除该元组的事务txid。若该元组尚未被删除或更新，则t_xmax=0，即invalid
t_cid：保存命令标识（command id,cid），指在该事务中，执行当前命令之前还执行过几条sql命令（从0开始计算）
t_ctid：一个指针，保存指向自身或新元组的元组的标识符（tid）。

当更新该元组时，t_ctid会指向新版本元组。若元组被更新多次，则该元组会存在多个版本，各版本通过t_cid串联，形成

元组insert时版本号规则

```
postgres=# CREATE TABLE test (id int);
CREATE TABLE
postgres=# begin;
BEGIN
postgres=# SELECT txid_current();
txid_current
-----
778
(1 row)

postgres=# insert into test values(1);
INSERT 0 1
postgres=# SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid FROM heap_page_it
tuple | t_xmin | t_xmax | t_cid | t_ctid
-----+-----+-----+-----+-----
1 | 778 | 0 | 0 | (0,1)
(1 row)
```

- t_xmin 被设置为778，表示插入该元组的txid（当事务开始，事务管理器会为该事务分配一个txid（transaction id）作为唯一标识符。）
- t_xmax 被设置为0，因为该元组还未被更新或删除过
- t_cid 被设置为0，因为这是该事务的第一条命令
- t_ctid 指向自身，被设置为（0,1），表示该元组位于0号page的第1个位置上

 进击的CJR

104文章

176粉丝

610K+浏览量

获得了 453 次点赞

内容获得 148 次评论

获得了 477 次收藏

TA的专栏

 PG vs MySQL
收录 1 篇内容

 postgresql学习笔记
收录 7 篇内容

 MySQL8.0
收录 7 篇内容

热门文章

MySQL资源整合
2023-05-26 148824浏览


PostgreSQL的pg_basebackup备份恢复详解
2021-12-10 32608浏览

MySQL--SQL优化--隐式字符编码转换
2021-11-02 18191浏览

实战篇：如何查看mysql里面的锁
2021-11-13 16942浏览

MySQL高可用--MGR入门（4）异常恢复
2021-11-20 16781浏览

在线实训环境入口

 MySQL在线实训环境

[查看详情](#)

最新文章

PG vs MySQL 统计信息收集的异同
2025-02-05 84浏览

PG备份恢复--pg_dump
2024-12-25 40浏览

MySQL8.0后的double write有什么变化
2024-12-24 117浏览

PG的权限管理
2024-12-18 100浏览

pgbench的使用
2024-11-26 43浏览

目录

元组delete时版本号规则

pg的删除只是将目标元组在逻辑上标为删除（将t_xmax设为执行delete命令的事务txid），实际该元组依然存在于数据库的存储页面，直至该元组被清理进程清理掉。

```
postgres=# begin;
BEGIN
postgres=# SELECT txid_current();
 txid_current
-----
          779
(1 row)

postgres=# delete from test where id=1;
DELETE 1
postgres=# SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid FROM heap_page_it
 tuple | t_xmin | t_xmax | t_cid | t_ctid
-----+-----+-----+-----+-----
      1 |      778 |      779 |      0 | (0,1)
(1 row)
```

- t_xmin 不变，表示插入该元组的txid
- t_xmax 被设置为779，即删除该元组的txid
- t_cid 被设置为0，因为这是该事务的第一条命令
- t_ctid 指向自身，被设置为 （0,1），表示该元组位于0号page的第1个位置上

当txid=779的事务提交时，tuple_1就不再需要了，称为dead tuple。但是这个tuple依然残留在页面上，随着数据库的运行，这种死元组越来越多，它们会在VACUUM时最终被清理掉。

元组update时版本号规则

pg不会直接修改数据，而是将目标元组标记为删除，并插入一条新元组，同时修改t_ctid执行新版本元组。

```
postgres=# begin;
BEGIN
postgres=# SELECT txid_current();
 txid_current
-----
          783
(1 row)

postgres=# SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid FROM heap_page_it
 tuple | t_xmin | t_xmax | t_cid | t_ctid
-----+-----+-----+-----+-----
      1 |      778 |      779 |      0 | (0,1)
      2 |      781 |        0 |      0 | (0,2)
      3 |      782 |        0 |      0 | (0,3)
(3 rows)

postgres=# update test set id = 8;
UPDATE 1
postgres=# SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid FROM heap_page_it
 tuple | t_xmin | t_xmax | t_cid | t_ctid
-----+-----+-----+-----+-----
      1 |      778 |      779 |      0 | (0,1)
      2 |      781 |        0 |      0 | (0,2)
      3 |      782 |      783 |      0 | (0,4)
      4 |      783 |        0 |      0 | (0,4)
(4 rows)
```

Tuple_3

- t_xmin 不变，表示插入该元组的txid
- t_xmax 被设置为783，即删除该元组的txid
- t_cid 被设置为0，因为这是该事务的第一条命令
- t_ctid 指向新版本元组，被设置为 （0,4），表示新元组位于0号page的第4个位置上

Tuple_4

- t_xmin 被设置为783，表示插入该元组的txid
- t_xmax 被设置为0，因为该元组还未被更新或删除过
- t_cid 被设置为1，因为这是该事务的第一条命令
- t_ctid 指向自身，被设置为 （0,4），表示该元组位于0号page的第4个位置上

PG的事务快照实现

事务状态

pg定义了四种事务状态——IN_PROGRESS, COMMITTED, ABORTED和SUB_COMMITTED。

- MVCC实现方法比较

- MVCC实现的原理

- [PG的MVCC实现原理](#)

- PG的数据多版本实现

- PG的事务快照实现

- PG的隔离级别实现

- MySQL的MVCC实现原理

- MySQL的数据多版本实现

- MySQL的事务快照实现

- MySQL的隔离级别实现

- PG vs MySQL

事务快照

事务快照就是当一个事务执行期间，那些事务active、那些非active。即这个事务要么在执行中，要么还没开始。

```
postgres=# SELECT txid_current_snapshot();
 txid_current_snapshot
-----
 796:796:
(1 row)
```

快照由这样一个序列构成 xmin:xmax:xip_list

- xmin：最早的active的 tid，所有小于该值的事务状态为visible(commit)或dead(abort)
- xmax: 第一个还未分配的xid，大于等于该值的事务在快照生成时都不可见
- xip_list 快照生成时所有active事务的txid

事务快照是用来存储数据库的事务运行情况。一个事务快照的创建过程可以概括为：

查看当前所有的未提交并活跃的事务，存储在数组中
选取未提交并活跃的事务中最小的XID，记录在快照的xmin中
选取所有已提交事务中最大的XID，加1后记录在xmax中
根据不同的情况，赋值不同的satisfies，创建不同的事务快照

可见性举例子

session 1：

```
postgres=# create table test(id int);
CREATE TABLE
postgres=# insert into test values(1);
INSERT 0 1
postgres=# begin;
BEGIN
postgres=# insert into test values(2);
INSERT 0 1
postgres=# select txid_current();
 txid_current
-----
          791
(1 row)

postgres=# select * from heap_page_items(get_raw_page('test',0));
 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid | t_infomask2 | t_ir
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
  1 |    8160 |         |    28 |     790 |      0 |         | (0,1) |           1 |
  2 |    8128 |         |    28 |     791 |      0 |         | (0,2) |           1 |
(2 rows)
```

session 2

```
postgres=# select txid_current_snapshot();
 txid_current_snapshot
-----
 791:791:
(1 row)

postgres=# select * from test;
 id
----
  1
(1 row)
```

session 1的事务791在session 2中并不可见，不仅因为txid>=xmax,还因为791的事务状态是

```
postgres=# select txid_status('791');
 txid_status
-----
 in progress
(1 row)
```

emp2
session 1


```
postgres=# begin;
BEGIN
postgres=# insert into test values(5);
INSERT 0 1
postgres=# select txid_current();
 txid_current
-----
          793
(1 row)

postgres=# rollback;
```

该事务回滚

在session2 中

```
postgres=# select * from test;
 id
----
  1
  2
  3
(3 rows)

postgres=# select txid_current_snapshot();
 txid_current_snapshot
-----
          794:794:
(1 row)

postgres=# select txid_status('793');
 txid_status
-----
 aborted
(1 row)
```

虽然txid<xmin 但是事务状态为aborted所以依然不可见。

PG的隔离级别实现

PostgreSQL中根据获取快照时机的不同实现了不同的数据库隔离级别

- 读未提交/读已提交：每个query都会获取最新的快照CurrentSnapshotData
- 重复读：所有的query 获取相同的快照都为第1个query获取的快照FirstXactSnapshot
- 串行化：使用锁系统来实现

比如说

session 1中

```
postgres=# truncate table test;
TRUNCATE TABLE
postgres=# insert into test values(1);
INSERT 0 1
postgres=# begin;
BEGIN
postgres=# insert into test values(2);
INSERT 0 1
postgres=# commit;
COMMIT
```

表test中插入两条数据，再插入第二条数据的时候开启了session 2，且隔离级别为RR，即使session 1提交了第二个事务，session 2 的快照依然没有变，也就没法读取到最新的数据。

```
postgres=# begin transaction isolation level repeatable read ;
BEGIN
postgres=# select * from test;
 id
----
  1
(1 row)

postgres=# select txid_current_snapshot();
 txid_current_snapshot
-----
          796:796:
(1 row)

postgres=# select * from test;
 id
----
  1
(1 row)
```


MySQL的MVCC实现原理

MySQL的数据多版本实现

区别于PG使用元组头部信息的字段来标示元组的版本号，MySQL 采用row trx_id来标示行数据的不同版本。同样，InnoDB 也会在事务开始的时候，申请一个顺序递增的事务 ID，叫作 transaction id。并且把 transaction id 赋值给这个数据版本的事务 ID，记为 row trx_id。

同时，旧的数据版本要保留到undo中，并且在新的数据版本中，能够有信息可以直接拿到它。也就是说，数据表中的一行记录，其实可能有多个版本 (row)，每个版本有自己的 row trx_id。

这里可以看出MySQL和PG标示不同的数据版本的差异，MySQL将旧数据写入到undo中，用row trx_id标识。而PG因为旧数据并没有删除，还在原堆表上，所以不能只用一个id标识，因此PG使用了t_xmin ，t_xmax等来多个id来和其他版本区分开。

MySQL的事务快照实现

在 MySQL 中，实际上每条记录在更新的时候都会同时记录一条回滚操作。记录上的最新值，通过回滚操作，都可以得到前一个状态的值。这个功能的实现依赖于UNDO。

InnoDB 为每个事务构造了一个数组，用来保存这个事务启动瞬间，当前正在“活跃”的所有事务 ID。“活跃”指的就是，启动了但还没提交。数组里面事务 ID 的最小值记为低水位，当前系统里面已经创建过的事务 ID 的最大值加 1 记为高水位。这个视图数组和高水位，就组成了当前事务的一致性视图（read-view）。也叫快照。

这个其实和PG的实现是一样的低水位就相当于PG快照的xmin，高水位相当于PG快照的xmax。而活跃未提交的事务就相当于PG中的xip_list。

- 如果落在低水位之前的部分，表示这个版本是已提交的事务或者是当前事务自己生成的，这个数据是可见的；
- 如果落在高水位的，表示这个版本是由将来启动的事务生成的，是肯定不可见的；
- 如果落在低水位和高水位之间的部分，那就包括两种情况
 - a. 若 row trx_id 在数组中，表示这个版本是由还没提交的事务生成的，不可见；
 - b. 若 row trx_id 不在数组中，表示这个版本是已经提交了的事务生成的，可见。

MySQL的隔离级别实现

和PG的实现原理一致，和快照的创建时间有关。

- 在“可重复读”隔离级别下，这个视图是在事务启动时创建的，整个事务存在期间都用这个视图。
- 在“读提交”隔离级别下，这个视图是在每个 SQL 语句开始执行的时候创建的。
- 这里需要注意的是，“读未提交”隔离级别下直接返回记录上的最新值，没有视图概念；
- “串行化”隔离级别下直接用加锁的方式来避免并行访问。

PG vs MySQL

在MVCC实现上，PG和MySQL的原理类似，只是旧数据的处理上的差异。PG在工作负载频繁更新/删除的情况下，存储空间会过大。pg永远不用担心回滚段不够用的问题，他的rollback可以立刻执行，而对大表的DML操作MySQL回滚会很慢。同样pg会存在一些无用的垃圾数据，所以需要vacuum来定时清理。否则旧版本的数据可能会导致查询需要扫描的数据块增多，从而导致查询变慢。空间持续上涨，存储没有被有效利用的问题也需要考虑到。

参考

PgSQL·引擎特性·多版本并发控制介绍及实例分析

http://mysql.taobao.org/monthly/2019/08/01/

PgSQL·特性分析·MVCC机制浅析

http://mysql.taobao.org/monthly/2017/10/01/

事务到底是隔离的还是不隔离的？

https://time.geekbang.org/column/article/70562

🔗 墨力计划 mysql pg

「喜欢这篇文章，您的关注和赞赏是给作者最好的鼓励」

关注作者

赞赏

【版权声明】本文为墨天轮用户原创内容，转载时必须标注文章的来源（墨天轮），文章链接，文章作者等基本信息，否则作者和墨天轮有权追究责任。如果您发现墨天轮中有涉嫌抄袭或者侵权的内容，欢迎发送邮件至：contact@modb.pro进行举报，并提供相关证据，一经查实，墨天轮将立刻删除相关内容。

文章被以下合辑收录



PG vs MySQL （共1篇）
PG 和 MySQL 的对比

收藏合辑

评论

分享你的看法，一起交流吧~



wzf0072



MVCC实现方法比较
MySQL
写新数据时，把旧数据写入回滚段中，其他人读数据时，从回滚段中把旧的数据读出来

PostgreSQL
写新数据时，旧数据不删除，直接插入新数据。

MVCC实现的原理
PG的MVCC实现原理
定义多版本的数据——使用元组头部信息的字段来标示元组的

22天前 点赞 评论

相关阅读

【干货】2024年下半年墨天轮最受欢迎的50篇技术文章+文档

墨天轮编辑部 1549次阅读 2025-02-13 10:42:44

MySQL性能分析的“秘密武器”，深度剖析SQL问题

szrsu 680次阅读 2025-01-23 09:59:26

2025年1月“墨力原创作者计划”获奖名单公布

墨天轮编辑部 348次阅读 2025-02-13 15:07:02

MySQL 主从节点切换指导

CuiHulong 302次阅读 2025-01-23 11:50:29

[MYSQL] 忘记root密码时, 不需要重启也能强制修改了!

大大刺猬 286次阅读 2025-02-06 11:12:15

mysql 内存使用率高问题排查

蔡璐 266次阅读 2025-02-06 10:02:23

MySQL 9.2.0 中的更新（2025-01-21，创新版本）

通讯员 151次阅读 2025-01-22 09:54:21

MySQL基础高频面试题－划重点、敲难点

锁钥 145次阅读 2025-02-03 07:52:28

MySQL 底层数据&日志刷新策略解读

CuiHulong 133次阅读 2025-02-11 10:56:13

[MYSQL] mysql主从延迟案例(有索引但无主键)

大大刺猬 107次阅读 2025-01-21 13:53:21