从 MVCC 机制看 POSTGRESQL 的应用场景

PostgreSQL 和 MYSQL 是目前应用最为广泛的两个开源数据库，PostgreSQL 因为其伯克利授权方式而更受到一些想利用开源数据库系统开发自主可控的企业数据库平台的企业和用户所青睐。关于 PostgreSQL 和 MYSQL 到底哪个数据库性能好，以及 PostgreSQL 能否替代 Oracle 数据库，在企业级应用中使用，是大家所关心的问题。本文从 PostgreSQL 的多版本并发访问（MVCC）机制角度，分析 PostgreSQL 较为适合的应用场景。

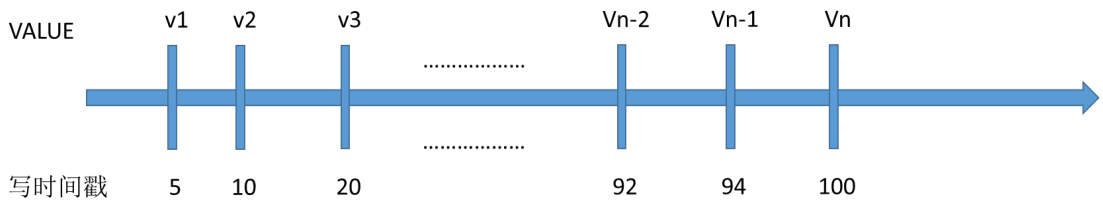
MVCC 是 Multi-Version Concurrency Control 的英文缩写，翻译成中文就是多版本并发控制。多版本并发控制的主要目的是为了提高在线事务处理系统(OLTP 系统）中的读操作的性能问题。

事务与事务隔离是现代关系型数据库的重要基础，通过所需要的事务隔离级别，来确保应用系统读取到的数据是符合业务逻辑的。事务隔离级别包含 read uncommitted(level 0，脏读)、read committed(level 1，提交读)、repeatable read(level 2，可重复读)、serializable(level 3，串行化)。其中脏读可以读取任何脏数据，因此不需要任何锁或者其他并发控制机制支持,并发性最好,串行化强制事务串行执行,并发能力最弱。提交读-Read committed 也叫一致性读，是目前在线联机事务（OLTP）系统中最为常见的事务隔离级别。

传统的事务理论采用锁机制来实现并发控制，简单的说，写操作使用排它锁，排斥其他操作；读操作使用共享锁，排斥写操作，但是可以支持其他读操作。随着信息化系统的发展，并发访问量越来越大，这种读写互斥锁的机制对并发访问的性能造成了极大的影响。1981 年位于马萨诸塞州剑桥的美国计算机公司（Computer Corporation of America）的两位技术人员 PHILIP A. BERNSTEIN AND NATHAN GOODMAN 发表了一篇具有历史意义的论文“Concurrency Control in Distributed Database Systems”，这篇以分布式数据库中并发控制为主要议题的论文中，提出了一种全新的并发控制算法，基于时间戳顺序的多版本并发控制机制（SYNCHRONIZATION TECHNIQUES BASED ON TIMESTAMP ORDERING）。这个全新的并发控制算法为数据库厂商提高并发能力提供了一条心的途径。DEC 公司的 VAX/RDB 是第一个采用多版本并发控制机制的商用关系型数据库,随后 Oracle 4.0 也开始支持 MVCC,IBM DB2 一直采用其独有的并发控制机制 Generalized Search Tme (GiST)，直到 DB2 9.7 才全面支

持 MVCC。到目前为止，绝大多数商用和开源数据库都已经全面支持多版本并发控制机制，多版本并发控制机制也已经成为交易型关系型数据库的标准配置。

Bernstein 和 Goodman 创造性的提出了基于 Timestamp Ordering（T/O）的多版本并发控制理论，对每个写操作记录 Timestamp，如下图：

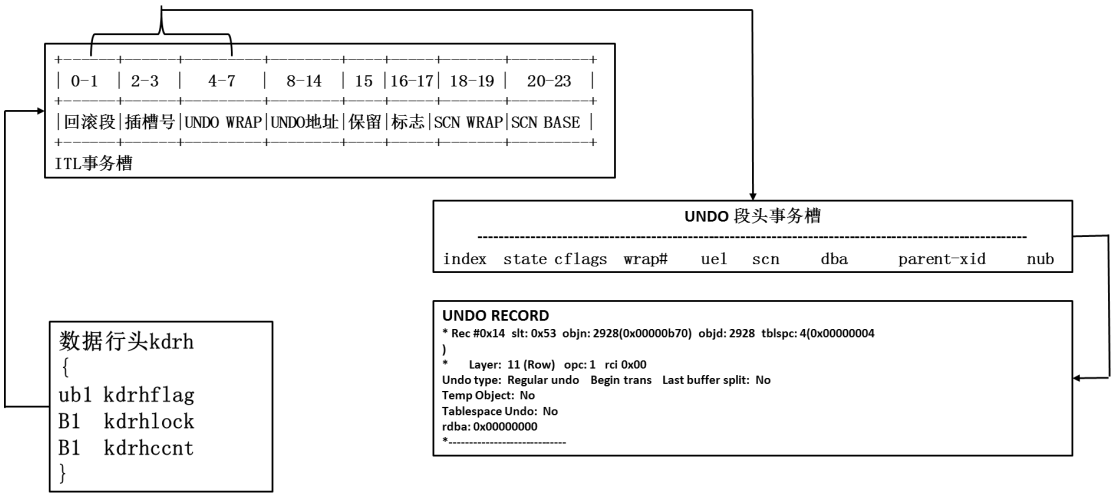


如果读操作 dm-read(95)需要读取 Timestamp 95 的数据，那么当前的最新写操作的 Timestamp 是 100，而小于 95 的最大 Timestamp 是 94，那么直接读取 dm-write(94)的值就可以了，也就是 Vn-1。

不同数据库实现多版本并发控制的方式是不同的，比如 Oracle 数据库的 T/O 使用的是一种类时间戳的机制 SCN（System Change Number），SCN 是每个事物提交的序号，是严格按照时间顺序增长的。PostgreSQL 和 Mysql 则采用了事物 ID（XID），事物 ID 也是严格按照时间顺序产生的，因此也满足 T/O 的基本要求。在 MVCC 的实现粒度方面，Oracle 采用的是页级并发控制，通过一致读数据页（Consistent Read Block，CR BLOCK）的机制实现多版本，而 PostgreSQL 和 Mysql 则采用行级多版本控制机制。

Oracle 的多版本并发控制机制一致是业界公认的较为优秀的，多版本并发控制机制，其特点是对读写操作的支持较为均衡，在高并发环境下有较好的性能。下面我们将通过分析 Oracle、PostGreSQL 和 Mysql 的多并发控制算法，分析这三种数据库在不同应用场景下的并发性能。

Oracle 的多版本并发控制是基于 Oracle UNDO/回滚段机制的，在回滚段中保存了某个数据被修改之前的前映像的数据。在每条记录的记录头（kdrh）中，kdrhlock 指向前一次修改该数据的事务槽（ITL）的位置，在 ITL 中记录了该次修改的 SCN 信息，以及回滚段的地址信息。其结构如下图：



当某个事务开始的时候，会在回滚段的段头 TRN TBL 中分配一个事务表记录，同时分配第一个 UNDO 记录，记下事务的一些信息。当事务修改某个数据的时候，在该数据的 DATA BLOCK 的 ITL 表中分配一个 ITL 记录，并锁定这个 ITL 记录，然后将数据行头中的 kdrhlock 指向这个 ITL 槽，然后再对数据进行修改。并把修改前的数据存储在回滚段的 UNDO RECORD 中。

如果有事务要读取相关的数据，首先对数据库的 DB CACHE 缓冲区进行搜索。在 Oracle 的 DB CACHE 中，同一个数据块可能存在多个版本，这些版本被称为 CR BLOCKS。如果在 DB CACHE 中已经找到了符合条件的 CR BLOCK（根据 SCN 来判断 CR BLOCK 是否符合查询条件），就可以直接使用，如果没有找到可用的 CR BLOCK，那么就需要通过该数据块的当前版本（CURRENTBLOCK）来生成所需要的 CR BLOCK。

在生成 CR BLOCK 的时候，可以根据该行数据的 kdrhlock 找到相关的 ITL 槽，通过比对 SCN 来判断要读取的数据是数据块中的数据还是修改前的数据。如果发现当前 ITL 槽中的 SCN 高于本事务所需要读取的 SCN，那么就会通过 ITL 槽找到该数据在 UNDO 中的前映像数据（PRE-IMAGE），通过前映像数据和当前数据生成一个一致性读块（CR BLOCK），然后通过访问这个 CR BLOCK 来找到所需要读的数据。实际环境中可能更为复杂，因为 ITL 槽可能会被覆盖，在这种情况下，Oracle 会把 ITL 信息写入 UNDO RECORD 中，形成一个链状结构，可以一层层的找到所需要的 UNDO RECORD，从而完成这种操作。

Oracle 的多版本并发控制机制使用了一个独立的 UNDO 表空间来存储 UNDO 数据，数据的前映像通过在 DB BUFFER 中的 CR BLOCK 来实现，因此数据无论修改多少次，都不会对存储数据的数据段产生负面的影响。而且一个 CR BLOCK 生成后，可以在缓冲区中较长时间内存在，供相关的事务使用。这个功能对于大并发的读操作来说，是十分有用的，可以大大提高相关操作的性能。

由于 Oracle UNDO 的空间容量有限，因此不可能永久保存回滚段的数据，Oracle 采用了 UNDO RETENTION 的机制来保护 UNDO 数据，可以设定一定的 UNDO 数据保存周期，当 UNDO 数据在保护期内，可以保证 UNDO 记录不被覆盖。这种机制很好的解决了 UNDO 数据生命周期管理的问题，同时确保了在一个大型查询中确保所需的 PRE-IMAGE 不会被覆盖失效。

PostgreSQL 早期的版本中是不支持多版本并发控制的，因此 PostgreSQL 的多版本并发控制不是通过类似 Oracle 的回滚段方式实现的，其实现手段是在数据表中保存某条数据的多个版本。比如说要对某条记录进行修改，并不是直接修改该数据，而是通过插入一条全新的数据，同时对老数据加以标识。而删除数据也不是直接删除该数据，而是在相应的数据

行上打上一个删除标识。为了更好的理解这种多版本并发控制机制，我们首先来看一个十分重要的数据结构 HeapTupleHeaderData 和 HeapTupleFields。HeapTupleHeaderData 是每一行数据的头结构，其定义如下：

```
typedef struct HeapTupleHeaderData
{
  union
    {
     HeapTupleFields t_heap;
     DataumTupleFields t_dataum;
    } t_choice;
  ItemPointerData t_ctid;
  uint16          t_informask;
  uint16          t_informask2;
  uint8           t_hoff;
  unit8           t_bits[1];
}  HeapTupleHeaderData;
```

HeapTupleFields 是用于多版本并发控制的核心数据结构，其定义如下：

```
typedef struct HeadTupleFields
{
    TransactionId  t_xmin;
    TransactionId  t_xmax;
    union
     {
      Commid t_cid;
      TransactionId txvac;
     } t_field3;
}  HeapTupleFields;
```

HeapTupleFields 结构是 PostgreSQL 实现 MVCC 的核心数据结构，xmin,xmax,cmin,cmax 和 xvac 是其中最为关键的字段。下面我们通过数据结构来看看 PostgreSQL 的 MVCC 工作机制。由于本文并不是分析 PostgreSQL 的 MVCC 机制原理的，因此仅仅对 PostgreSQL 的 MVCC 机制的基本原理进行分析，不会涉及其深成次的工作原理。

当一条记录被插入到数据库时，其 xmin 字段被存储为本事务的 XID，xmax 为 0,当事务提交后，所有的事务的 XID 大于等于 xmin 中存储的 XID 的事务，都可以看到这条记录。这完全符合 read commit 事务隔离级别的要求。

如果该记录被删除，在 PostgreSQL 中，暂时不会删除这条记录，而是会在这条记录上做一个标识。PostgreSQL 的做法是将该记录的 xmax 设置为删除这条记录的事务的 XID。这样，所有的该记录删除后的事务的 XID 都大于 xmax 的值，因此删除后发起的查询都无法读取到这条记录；而对于删除这条记录之前的启动的查询，由于 XID 小于 xmax，因此仍

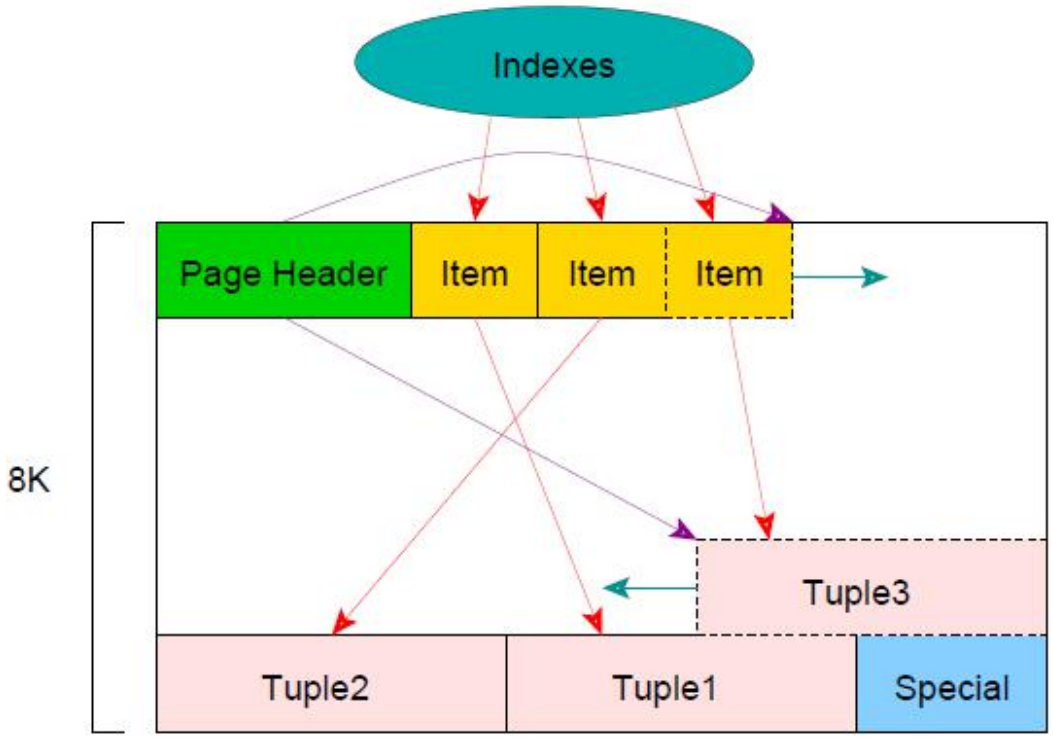
然可以读取到这条记录。这样就解决了 MVCC 的事务隔离和一致性读的问题。

如果该记录被修改（update）了，那么 PostgreSQL 不会直接修改原有的记录，而是会生成一条新的记录，新记录的 xmin 为 update 操作的 XID，xmax 为 0，同时会将老记录的 xmax 设置为当前操作的 XID，也就是说新记录的 xmin 和老记录的 xmax 相同。这样在同一张表中，同一条记录就会有存在多个副本。

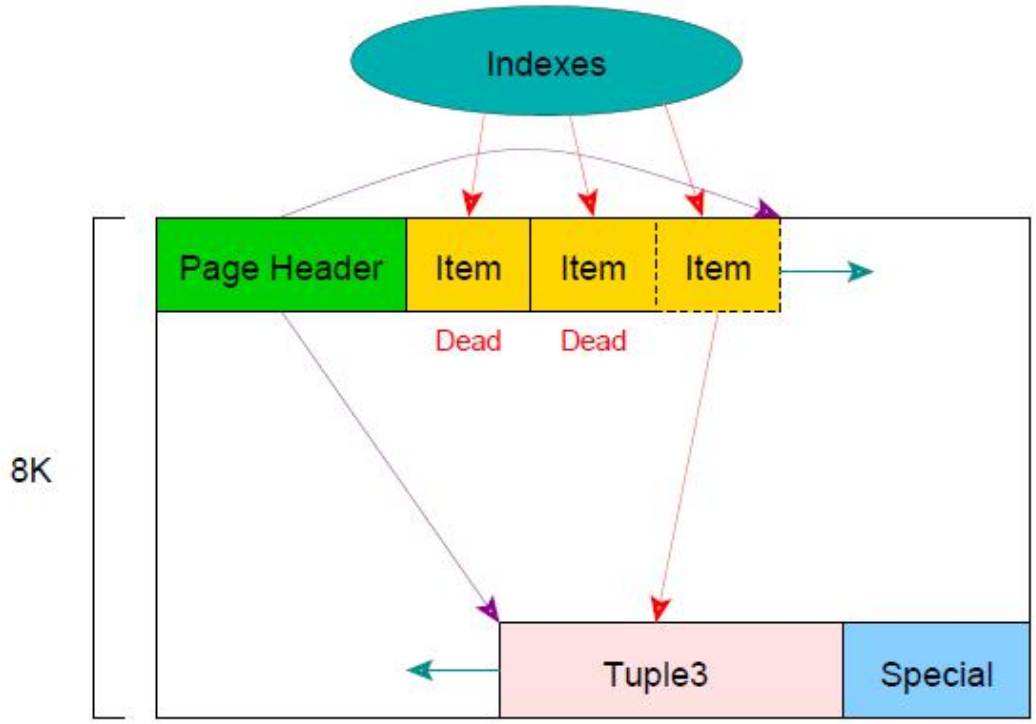
由于数据多副本的存在，在索引中也会产生类似的多副本情况，因此当 DML 操作产生时，索引页会做相关的调整。

从 PostgreSQL 的 MVCC 机制工作原理来看，INSERT 操作并没有太多的问题，PostgreSQL 的 INSERT 操作和其他数据库的工作原理十分类似，只是 PostgreSQL 的行头大小为 20 字节，远远大于 Oracle 的 3 字节，因此 PostgreSQL 的存储额外开销要略大于 Oracle。从 UPDATE 操作来看，无论 UPDATE 多少个字段，PostgreSQL 都需要插入一条新的记录，这样会造成 SEGMENT 高水位的增长，如果某张表的数据插入后，需要多次 UPDATE，那么这张表的高水位会出现暴涨。为了解决这个问题，PostgreSQL 使用了一个版本回收机制----VACUUM。通过 VACUUM，PostgreSQL 可以回收旧版本，从而避免多版本带来的性能问题。下面通过 Bruce Monjian 的一组示意图来回顾一下 PostgreSQL 的 MVCC 机制以及 VACUUM。

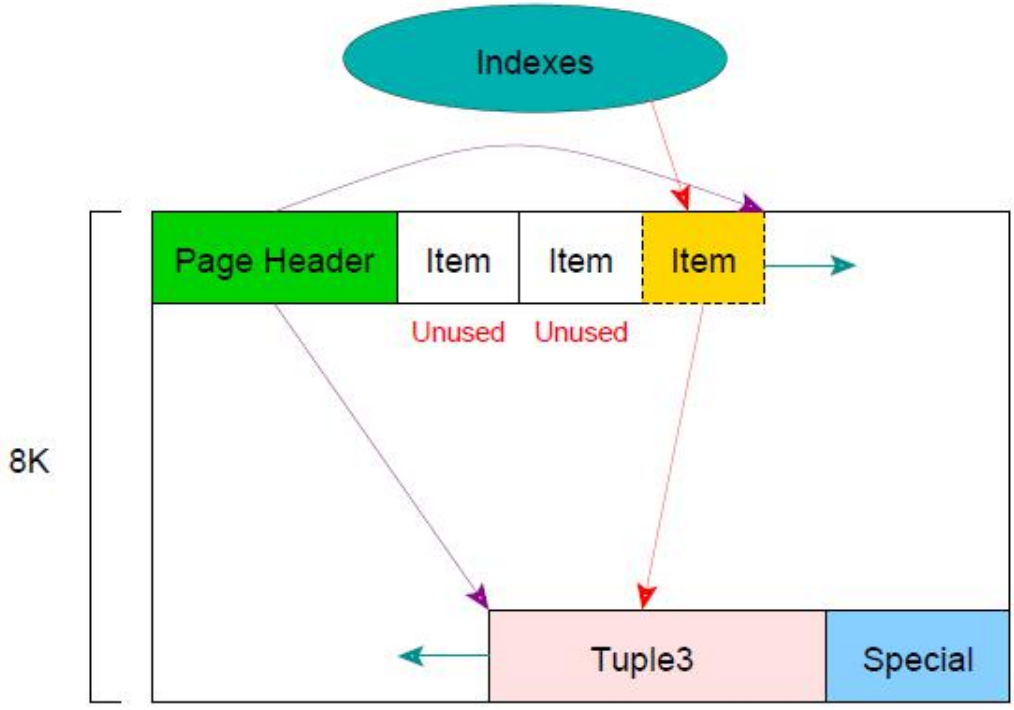
我们先来看一个简单的场景，在一个数据块中有 3 条记录，如下图：



当 Tuple1 和 Tuple2 被删除后，数据块中的数据如下图：



这时候索引指向的块头中的 Tuple Item 还没被清理，因此记录索引表的内容还不能复用。必须经过 VACUUM 才能完成这个工作，VACUUM 后，块内的的情况如下图：



从上图，我们可以看出经过 VACUUM,PostgreSQL 可以有效的进行空间的回收，不过普通的 VACUUM 仅仅能够回收数据块内的空间，无法降低高水位。要降低高水位，必须进行一种称为 VACUUM FULL 的操作，这个操作类似于 Oracle 的 SHRINK 操作，可以对 PostgreSQL 的表进行空间回收和降低高水位的操作。不过 VACUUM FULL 操作和 Oracle 的 Shrink Table 操作不完全类似，VACUUM FULL 操作必须排他方式锁定表，直到操作完成。VACUUM FULL 操

作如此高的成本，使 7*24 系统中经常性进行 VACUUM FULL 操作的可能极低，那么不经常性进行 VACUUM FULL 操作，会对系统带来什么影响呢？

从 PostgreSQL 的 MVCC 机制上来看，对于经常有 DML 操作的表来说，其高水位会有较大的变化。甚至可能达到正常水平的数倍、数十倍。这样会影响全表扫描的性能。由于表本身的碎片，也会引起表上索引出现不正常增长。这种增长也会影响范围扫描和索引唯一扫描的性能。那么在实际应用环境中，这种碎片对 PostgreSQL 的访问性能会产生什么样的影响呢？下面我们通过一组实验来分析由于 DML 语句产生的数据碎片对性能有什么用的影响。实验通过对一张表进行初始化后分别进行删除、修改等操作，然后针对 POSTGRESQL 数据库的各种访问行为，进行比对分析。

对于 PostgreSQL 数据库，测试用表 p_sms_send_attr_1 的结构如下表：

字段名	字段类型	索引
sms_send_id	numeric(24,0)	index_sms_send_id btree (sms_send_id)
cust_no	character(32)	
policy_id	character(32)	
remind_plan_id	character(32)	
emp_no	character(32)	
handle_time	date	
send_emp	character(32)	
resend_times	character(32)	
cons_name	character(32)	
line_id	numeric(32,0)	
send_type	character(32)	
busi_type	character(32)	
org_no	character(32)	
app_no	character(32)	
outage_source	character(50)	
accident_new_id	character(120)	

初始化的测试表 p_sms_send_attr_1 的大小为 4G 单表，索引大小 1102 MB。

测试用例 1：UPDATE 表的一列值，看表的变化

	测试语句
PostgreSQL	test=> update p_sms_send_attr_1 set emp_no='aaa'; UPDATE 52319600 test=> update p_sms_send_attr_1 set line_id=100; UPDATE 52319600
MySQL	mysql> update p_sms_send_attr_1 set emp_no='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'; Query OK, 3934000 rows affected, 65535 warnings (4 min 42.35 sec) Rows matched: 3934000 Changed: 3934000 Warnings: 3934000

测试结果：

	PostgreSQL	MySQL

	原来	vacuum full p_sms_send_attr_1 表后	原来	多版本数据的清理
表大小	35G	11G	2020M	984M
索引大小	5613 MB	1403 MB		
Page	4629721	1413904		

备注：PostgreSQL 采用 vacuum full p_sms_send_attr_1 后，表的数据量由原来的 35G 减少为 11G，索引由原来的 5.6G 直接下降到了 1.4G，PAGES 由原来的 4629721 变成 1413904 说明着 UPDATE 表的数据后，存在着表的多个版本数据，对表和索引的大小都存在着影响。

测试用例 2：DELETE 表记录，验证 MVCC 对表和索引影响

	测试语句

PostgreSQL	test=> delete from p_sms_send_attr_1 where sms_send_id>=19880000 and sms_send_id<=20000000;
	DELETE 35807
	test=> delete from p_sms_send_attr_1 where sms_send_id>=19880000 and sms_send_id<=50000000;
	DELETE 9226844
	test=> delete from p_sms_send_attr_1 where sms_send_id>=50000000 and sms_send_id<=90000000;
	DELETE 9206992

MySQL	mysql> delete from p_sms_send_attr_1 where sms_send_id<=200000000; Query OK, 1152190 rows affected (8.20 sec)

测试结果：

	PostgreSQL		MySQL	
	原来	vacuum full p_sms_send_attr_1 表后	原来	多版本数据的清理
表大小	11G	7146 MB	984M	696M
索引大小	1.4G	1G		

备注：PostgreSQL 采用 vacuum (verbose,full,analyze) 进行表分析后，表在进行 DELETE 操作后，表的大小由原来的 11G 变为 7.1G,索引由原来的 1.4G 下降到 1G。

测试用例 3：全表扫描测试

首先针对 PostgreSQL 进行测试：

```
test=> \dt+p_sms_send_attr_1;
                          List of relations
 Schema |       Name        | Type  | Owner |    Size    | Description
--------+-------------------+-------+-------+------------+-------------
 test   | p_sms_send_attr_1 | table | test  | 7146 MB    |

test=> select pg_size_pretty(pg_relation_size('index_sms_send_id'));
 pg_size_pretty
----------------
 1008 MB
test=>  explain ( analyze on, buffers on ) select count(*) from p_sms_send_attr_1;
```

```
QUERY PLAN
------------------------------------------------------------------------------------------
 Aggregate          (cost=1337849.00..1337849.01     rows=1     width=0)     (actual
time=8791.895..8791.895 rows=1 loops=1)
   Buffers: shared hit=192 read=914532
   ->  Seq Scan on p_sms_send_attr_1  (cost=0.00..1253224.00 rows=33850000 width=0)
(actual time=0.005..5568.922 rows=33849957 loops=1)
         Buffers: shared hit=192 read=914532
 Total runtime: 8791.928 ms
```

在 UPDATE 前，表大小为 7146MB，索引 1008 MB，数据量 33849957 行，全表扫描时间为 8.7 s，扫描了 914724 个 PAGE，也就是扫描的 Buffers 量。

```
test=> update p_sms_send_attr_1 set policy_id='bbb';
UPDATE 33849957
test=> \dt+p_sms_send_attr_1;
                          List of relations
 Schema |       Name        | Type  | Owner |    Size    | Description
--------+-------------------+-------+-------+------------+-------------
test    | p_sms_send_attr_1 | table | test  | 15 GB      |
test=>  select pg_size_pretty(pg_relation_size('index_sms_send_id'));
 pg_size_pretty
----------------
 2829 MB
test=>  explain ( analyze on, buffers on ) select count(*) from p_sms_send_attr_1;
                                                                  QUERY PLAN
------------------------------------------------------------------------------------------
Aggregate          (cost=2884706.74..2884706.75     rows=1     width=0)     (actual
time=25180.208..25180.208 rows=1 loops=1)
   Buffers: shared hit=74939 read=1897414 written=1003625
   ->  Seq Scan on p_sms_send_attr_1  (cost=0.00..2702235.99 rows=72988299 width=0)
(actual time=695.087..21853.726 rows=33849957 loops=1)
         Buffers: shared hit=74939 read=1897414 written=1003625
 Total runtime: 25180.247 ms
```

更新表的一列值，33849957 行数，在没有进行 vacuum 的情况下，当前表大小为 15G，索引大小为 2829MB，全表扫描时间为 25s（UPDATE 前为 8.7 秒），扫描了 1972353 个 PAGE，也就是扫描的 Buffers 量。说明 UPDATE 操作对全表扫描的性能产生了较大的影响。下面对标进行 VACUUM，然后再次分析全表扫描的性能。

```
test=>    vacuum (verbose,analyze) p_sms_send_attr_1;
test=> explain ( analyze on, buffers on ) select count(*) from p_sms_send_attr_1;

QUERY PLAN
```

```
---------------------------------------------------------------------------------------------------------------
Aggregate  (cost=2041032.56..2041032.57 rows=1 width=0) (actual time=9488.286..9488.286 rows=1 loops=1)
   Buffers: shared hit=1587747 read=339597
   ->  Index Only Scan using index_sms_send_id on p_sms_send_attr_1 (cost=0.56..1956411.99 rows=33848228 width=0) (actual ti
me=0.037..6256.993 rows=33849957 loops=1)
         Heap Fetches: 0
         Buffers: shared hit=1587747 read=339597
 Total runtime: 9488.326 ms
(6 rows)
```

对表进行的默认的 VACUUM 操作，表的大小和索引的大小都没有变，但是全表扫描的时间由原来的 25s 下降到了 9S。这说明 VACUUM 可以较为有效的优化全表扫描操作，不过普通 VACUUM 后表的访问性能还是略低于原有性能。

下面针对 Mysql Innodb 引擎进行相同的测试，在修改表之前：

```
mysql> select count(*) from p_sms_send_attr_1;
+----------+
| count(*) |
+----------+
|  2781810 |
+----------+
1 row in set (0.84 sec)

mysql> update p_sms_send_attr_1 set outage_source='lasdafasdfsdf3tdczasdfst5tx';
Query OK, 2781810 rows affected (1 min 54.45 sec)
Rows matched: 2781810  Changed: 2781810  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (2.18 sec)

mysql> select count(*) from p_sms_send_attr_1;
+----------+
| count(*) |
+----------+
|  2781810 |
+----------+
1 row in set (0.90 sec)
```

查看表的大小

```
[mysql@test6 tpcc]$ ls -l p_sms_send_attr_1.*
-rw-rw---- 1 mysql mysql          9186 Aug 11 07:21 p_sms_send_attr_1.frm
-rw-rw---- 1 mysql mysql 1564475392 Aug 11 07:33 p_sms_send_attr_1.ibd
```

更新后表大小变为 1492M，，全表扫描时间从 0.84s 变为 0.9s。可以看出对于 Mysql，修改操作影响甚微。

测试用例 4：MVCC 对索引范围扫描的影响

首先测试 PostgreSQL

```
test=> \dt+p_sms_send_attr_1;
                          List of relations
 Schema |       Name        | Type  | Owner |    Size    | Description
--------+-------------------+-------+-------+------------+-------------
test    | p_sms_send_attr_1 | table | test  | 8263 MB    |
test=> select pg_size_pretty(pg_relation_size('index_sms_send_id'));
 pg_size_pretty
----------------
 1008 MB
```

索引唯一扫描

```
test=> explain ( analyze on, buffers on )select sms_send_id from p_sms_send_attr_1
where sms_send_id=146079659;
                                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Index Only Scan using index_sms_send_id on p_sms_send_attr_1  (cost=0.56..8.58
rows=1 width=8) (actual time=0.076..0.077 row
s=1 loops=1)
   Index Cond: (sms_send_id = 146079659::numeric)
   Heap Fetches: 1
   Buffers: shared hit=1 read=4
 Total runtime: 0.095 ms
```

备注：当前表大小为 8263MB,索引大小为 1008MB，索引扫描查询耗时 0.095 ms

索引范围扫描

```
test=> explain ( analyze on, buffers on )select sms_send_id from p_sms_send_attr_1
where sms_send_id>=140000000 and sms_send_id<=200000000;
                                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on p_sms_send_attr_1  (cost=122291.96..1251884.97 rows=4797600
width=8) (actual time=1188.237..3701.899 row
s=4793570 loops=1)
   Recheck Cond: ((sms_send_id >= 140000000::numeric) AND (sms_send_id <=
200000000::numeric))
   Rows Removed by Index Recheck: 8605
```

```
   Buffers: shared hit=95 read=168388
   ->  Bitmap Index Scan on index_sms_send_id  (cost=0.00..121092.57 rows=4797600
width=0) (actual time=1185.720..1185.720 rows=4793570 loops=1)
         Index Cond: ((sms_send_id >= 140000000::numeric) AND (sms_send_id <=
200000000::numeric))
         Buffers: shared hit=3 read=18367
 Total runtime: 4043.839 ms
```

备注：当前表大小为 8263MB,索引大小为 1008MB，索引范围扫描查询耗时 4043.839 ms

场景：

创建一个函数，update 一条记录 100000 次

```
test=>CREATE or replace FUNCTION   fun_update_send_attr() RETURNS INTEGER   AS $$
     DECLARE
                i         INTEGER ;
     BEGIN
               for i in 1..100000loop
               update p_sms_send_attr_1   set policy_id='bbb'where sms_send_id=146079659;
               end loop;
               return 1;
     END;
$$ LANGUAGE 'plpgsql'; CREATE FUNCTION
test=> select fun_update_send_attr();
 fun_update_send_attr
----------------------
                    1
(1 row)
```

查看表的大小，索引的大小

```
test=> \dt+p_sms_send_attr_1;
                                    List of relations
 Schema |          Name          | Type  | Owner |      Size       | Description
--------+------------------------+-------+-------+-----------------+-------------
test    | p_sms_send_attr_1 | table | test   | 8326 MB       |

test=>   select pg_size_pretty(pg_relation_size('index_sms_send_id'));
 pg_size_pretty
----------------
 1008 MB
(1 row)
```

备注：表的大小为 8326MB，比原来表多了 100MB，索引大小没有变化。

索引唯一扫描：

```
test=>  explain ( analyze on, buffers on )select sms_send_id from p_sms_send_attr_1
where sms_send_id=146079659;
```

```
                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Only Scan using index_sms_send_id on p_sms_send_attr_1  (cost=0.56..8.58
rows=1 width=8) (actual time=0.041..1.775 row
s=1 loops=1)
   Index Cond: (sms_send_id = 146079659::numeric)
   Heap Fetches: 313
   Buffers: shared hit=324
 Total runtime: 1.796 ms
(5 rows)
```

可以看出，索引唯一性扫描的性能相比 UPDATE 前有了较大的下降。

索引范围扫描：

```
test=>  explain ( analyze on, buffers on )select sms_send_id from p_sms_send_attr_1
where sms_send_id>=140000000 and sms_send_id<=200000000;
                                                                    QUERY PLAN


-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on p_sms_send_attr_1  (cost=122658.68..1260583.49 rows=4832987
width=8) (actual time=1117.147..3219.184 rows=4793570 loops=1)
   Recheck Cond: ((sms_send_id >= 140000000::numeric) AND (sms_send_id <=
200000000::numeric))
   Rows Removed by Index Recheck: 8605
   Buffers: shared hit=168492
   ->  Bitmap Index Scan on index_sms_send_id  (cost=0.00..121450.43 rows=4832987
width=0) (actual time=1114.724..1114.724 ro
ws=4793570 loops=1)
         Index Cond: ((sms_send_id >= 140000000::numeric) AND (sms_send_id <=
200000000::numeric))
         Buffers: shared hit=18378
 Total runtime: 1532.491 ms
```

结论：在对表单条记录进行 1000000 次更新，表的大小会变大，索引变化较小，索引扫描的性能会有所下降。

```
test=> vacuum (verbose,analyze) p_sms_send_attr_1;
```

索引范围扫描：

```
test=> explain ( analyze on, buffers on )select sms_send_id from p_sms_send_attr_1 where
sms_send_id>=140000000 and sms_send_id<=200000000;
QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
Index Only Scan using index_sms_send_id on p_sms_send_attr_1  (cost=0.56..168688.23
rows=4786583 width=8) (actual time=0.060..1319.018 rows=4793570 loops=1)
   Index Cond: ((sms_send_id >= 140000000::numeric) AND (sms_send_id <=
200000000::numeric))
```

```
    Heap Fetches: 0
    Buffers: shared hit=535610
 Total runtime: 1622.731 ms
```

将对表进行 vacuum 操作后，对表的索引唯一扫描和索引范围扫描正常。

相同的测试用例，在 Mysql Innodb 上进行同样的测试：

索引唯一扫描

```
mysql> select * from p_sms_send_attr_1 where sms_send_id=201142082;
1 row in set (0.01 sec)
```

备注：当前表大小为 1492M， 索引唯一扫描查询耗时 0.01s

索引范围扫描

```
mysql> select * from p_sms_send_attr_1 where sms_send_id>=200754160 and
sms_send_id<=201154160;

7648 rows in set (0.03 sec)
```

备注：当前表大小为 1492M，索引扫描查询耗时 0.03 s

场景：

创建一个存储过程，update 一条记录 100000 次

```
mysql> delimiter /
mysql> create procedure update100000()
    -> begin
    -> declare i int;
    -> set i=0;
    -> while i < 100000
    -> do
    -> update p_sms_send_attr_1 set policy_id='bbb' where sms_send_id=201142082;
    -> set i=i+1;
    -> end while;
    -> commit;
    -> end;
    -> /
Query OK, 0 rows affected (0.00 sec)
mysql> delimiter ;

mysql> call update100000;
Query OK, 0 rows affected (4.65 sec)
```

查看表的大小

```
[mysql@test6 tpcc]$ ls -l p_sms_send_attr_1.*
-rw-rw---- 1 mysql mysql          9186 Aug 11 07:21 p_sms_send_attr_1.frm
```

```
-rw-rw---- 1 mysql mysql 1564475392 Aug 11 08:11 p_sms_send_attr_1.ibd
```

表分配空间没有变化。

索引唯一扫描

```
mysql>    select * from p_sms_send_attr_1 where sms_send_id=201142082;
1 row in set (0.00 sec)
```

索引范围扫描

```
mysql> select * from p_sms_send_attr_1 where sms_send_id>=200754160 and
sms_send_id<=201154160;
7648 rows in set (0.03 sec)
```

结论：在对表单条记录进行 1000000 次更新，表的分配空间不变，索引唯一查询和索引范围扫描时间基本不变。

从上述测试可以看出，PostgreSQL 数据库在大量 DELETE、UPDATE 操作后，表和索引都会出现高水位大幅提升的问题。从而导致索引唯一性访问、索引范围扫描和全表扫描性能的下降，而且下降的幅度还是比较大的。经过 VACUUM 后，高水位无法恢复，不过索引访问和全表扫描的性能都可以大幅度恢复。而同样的测试场景，Mysql 数据库不存在类似的问题。

PostgreSQL 使用了一种十分简单的方法实现了多副本控制机制，并没有像 Oracle、MYSQL 一样引入专门的回滚段机制。这种 MVCC 机制实现十分简单，但是我们可以很快发现这种机制存在的问题，因为 update 操作会在表中产生大量的 MVCC 版本，从而导致表和索引碎片的大量产生，从而影响该表访问的性能。不过这种 MVCC 机制也不都是副作用，对于删除操作，由于 PostgreSQL 只需要简单的标识，因此大批量删除的场景，PostgreSQL 的性能优于 Mysql。

由于 MVCC 机制的缺陷，PostgreSQL 数据库的应用场景也受到了一定的限制。下表针对 PostgreSQL 的应用场景进行了详细的分析。

场景	是否适合
大量插入，少量修改的 OLAP 场景	是
并发量不大，大量 DML 的非 7*24 场景	是
并发量较大，大量 DML 的非 7*24 场景	是
并发量不大，大量 DML 的 7*24 场景	是*
并发量较大，大量 DML 的 7*24 场景	否
数据量较大，某些大表的 DML 较多，有较大并发量进行全表扫描的 7*24 场景	否

PostgreSQL 是一个十分优秀的关系型数据库，近年来的每个小版本的出现都会带来较大的性能提升。现在的 PostgreSQL 已经不仅仅在 OLAP 领域可以发挥很大的作用，在 OLTP 领域同样具有较高的性能。但是由于其多版本并发控制机制的特点，虽然在一些 DML 操作场景下性能与 Mysql 等数据库在伯仲之间，甚至高于 Mysql，但是在一些高并发的 DML 比例较大的应用场景下，由于多版本控制机制产生的表和索引碎片问题，会导致严重的性能问

题。因此在数据库选型时需要全面考虑。

参考文献：

1、"Concurrency Control in Distributed Database Systems",PHILIP A. BERNSTEIN AND NATHAN GOODMAN,Computer Corporation of America, Cambridge, Massachusetts 02139,1981,ACM Computing Surveys, Vol. 13, No. 2, June 1981

2、Mvcc Unmasked,BRUCE MOMJIAN,July,2014

3、Concurrency and Recovery in Generalized Search Trees , Marcel Kornacker (U. C. Berkeley) , C. Mohan (IBM Research Division) , Joseph M. Hellerstein (U. C. Berkeley),In Proceedings ACM SIGMOD International Conference 1997:62-72

4 、 Relationship among ITL, Transaction Table and Undo record , Doc ID: Note:120338.1 of Metalink