MVCC，全称是Multi-version Concurrency Control，即多版本并发控制。

常用的并发控制技术，一共三种：

第一种就是我们今天要着重介绍的--Multi-version Concurrency Control(MVCC),多版本并发控制

第二种是Strict Two-Phase Locking（S2PL），严格的两阶段锁

第三种是Optimistic Concurrency Control（OCC），乐观锁

目前市场上的数据库用的基本就是这三种并发控制技术。

在MVCC实现前，我们先了解几个概念：

事务和隔离

要了解多版本并发控制，我们需要了解两个概念：事务和隔离

事务：

第一个概念就是事务，事务的本质就是讲多个步骤捆绑称一个原子操作。比如先INSERT，然后UPDATE,在SELECT 三个执行语句，在事务中，我们会把这三个执行步骤捆绑称一个原子的步骤，并希望这一个原子的步骤或者全部成功，即所有的语句均能执行成功，或者全不成功。如果在中间出现了故障导致事务无法完成，则所有步骤都不会影响数据库，此时会回滚到初始化状态。

由于事务的这种特性，事务执行的中间状态是不会被别的事务看到的。大家想一想，如果这个事务的中间状态会被另外一个事务看到，因为已经对其他事务产生了影响，回滚将没有办法完成。这里就涉及到了另一个概念：隔离。

**隔离**

由于事务是具有原子性的，因此事务和事务之间，我们是不希望看到中间状态的，因此事务和事务之间要有隔离。SQL标准里把事务隔离级别分成了四级：

Read Uncommunitted(读未提交)

Read Committed(不可重复读)

Repeatable Read(可重复读)

Serializable (串行化)

1.Read Uncommited

第一个级别Read Uncommitted是可以脏读的。即不管事务进行到哪一步，该事务所做的改动对别的事务均可见。按理说，在一个严肃的数据库里，这似乎是不应该发生的事情，因为我们希望事务是一个原则性的操作，如果他失败了需要回滚，所以说他的中间状态一定是不能被别的事务看到的，可见这是一个非常低的级别，是一个通常情况下，很难被人们接受的一个级别。

2.Read Committed

第二个级别叫Read Committed。在这个级别下，是不支持脏读的，但一旦事务提交了，它所造成的影响就对别的事务可见了。基本满足了原子性的要求。

但是Read Committed存在一个问题。如果一个事务中有很多条语句，那么可能会出现，在不同的语句中看到的数据集不一样的情况。

例如，事务一和事务二并发进行，但是在事务二进行到第三个语句的时候，事务一提交了，接着，事务二回继续执行第四条语句。这种情况下，前三条语句和第四条语句中，事务二读取的内容便会不一样。

这种能够情况虽然不影响事务的原子性，但由于在执行计算的过程中，拿到的数据集是不稳定的，这会给数据库的使用者造成一些困扰。如果是否能让整个事务中每条语句中看到的数据集都一样呢？这就是下一个隔离级别--Repeaable Read

3.Repeatable Read

Repeatable Read很好的解决了Read Committed隔离级别中数据集不稳定的问题。在这种隔离级别下，不管读多少数据集，均能得到一致的结果。

4.Serial lizable

第四个隔离级别是Serialzable。Serializable的要求是，虽然事务在并发执行，但是需要其执行结果完成等价于顺序执行的事务，这显然是一个更高的隔离要求。Serializable和Repeatable Read的差别是非常微妙的。

Greenplum实现了中间的两个隔离级别：Read Committed和Reapeatable Read，其默认的隔离级别是Read Committed。

**什么是MVCC**

MVCC到底是什么呢？MVCC具体可以解决什么问题呢？

包括MVCC，严格两阶段锁，乐观锁在内的并发控制协议的存在，使我们能够正确并且高效的进行并发访问，包括读和写。

单独做到正确或者是高效非常简单。要单独实现正确的进行并发访问，我们只需要把他串行化，一个数据集在同一时间只让一个事务进行访问，这样的话，肯定是正确的，不顾正确，单独实现高效的进行并发访问同样也很简单，但兼顾高效与正确便很难了。MVCC存在的目的，就是在高效性和正确性之间找到一个平衡点。

并发访问中涉及的操作包括读操作和写操作。其中，由于同一份数据可以支持多人读取，因此读操作和读操作是不会冲突的。同一份数据的写操作和写操作则一定会发生冲突，此时便需要用锁来解决。读操作和写操作有时候也会出现冲突，MVCC可以很好的解决这一冲突，其优化目标是读操作不会阻塞写操作，写操作也不会阻塞读操作。

这个目的是如何实现的呢？在update一个数据时，将创建数据项目的新版本，同时保留旧版本，并坐上标记，即采用多版本的方法来解决读和写之间的冲突。由于写的时候旧版本数据不会被删除，读和写发生在了不同版本的数据上，读和写操作便不会发生冲突。这种方法允许Greenplum在读写同时进行的情况下，仍然能够提高并发特性。

**MVCC的实现**

Heap表页面布局

我们再来了解一下Heap表的页面布局。由于Greenplum中的每一个表都是文件，文件的底层数据布局由一些相等大小，约32kb的Page组成。

将Page展开后，大家可以看到，Page中包括表头的一些信息，表头信息后是由Item组成的数组。Item代表了下面的Tuple。

HeapPage中，前面是一些定长的指针，指针里面装的offset（偏移量），而偏移量指向真正储存数据的Tuple（元组）。指针从前向后增长，turple从后向前增长，中间是空闲空间。

turple的内部结构由两部分组成，后面一部分是用户数据。前面这一小部分是可见性信息，存了xmin，xmax，Cid,Ctid等信息。其中xmax和xmin是用来跟踪事务之间的可见性；而Cid是用来记录事务内部的可见性。

xmin：创建Tuple的事务ID

xmax：删除Tuple的事务ID，有时用于行锁

Cid：事务内的查询命令编号，用户跟踪事务内部的可见性

Ctid：指向下一个版本Tuple的指针，由两个成员blocknumber、offset组成

快照理论上是某个时间点正在运行的事务列表。Greenplum使用快照判断一个事务是否已提交，即快照存在的目的是在某个时间点跟踪所有事务的运行状态，用来控制元组是否对于当前查询可见。快照和turple里的xmin、xmax一起来决定这个turple的可行性。

在Read Committed隔离级别，每个查询开始时生成快照，在REPEATABLE READ隔离级别，在每个事务开始时生成快照。

xmin：所有小于xmin的事务都已提交

Running：正在执行的事务列表

xmax： 所有大于等于xmax的事务都未提交

**可见性**

为什么要有可见性呢？主要是由于两个原因，第一个是因为前面提到的隔离性。例如Read Committed隔离级别下，只有事务提交了，其他事务才能看见其所对应的数据，于是我们需要知道数据库中的事务状态，即那些事务是在运行当中，那些事务已经提交了，因此我们就需要快照来判断turple的可见性。

第二个原因是，由于MVCC这种多版本的机制，update的操作会导致同一个数据在数据库里村很多版本。

那么问题来了，这些版本里只有一个版本可见，即已经提交的最新版本，而还没有提交的，或已经被删除的版本是不可见的，因此我们会通过一种方法来判断可见性，从而实现多版本。

可见性是如何判断的呢？我们来看一个比较简单的例子，下图的三条数据中，xmin都是有值的，xmax都是没值的，这意味着着三个数据被插入进来后，还未被删除。和快照中的xmin，xmax和Running进行对比，30小于快照中的xmin，已经提交了，因此这条数据时可见的。50在Running列表中，因此它还在执行，110大于xmax，还未提交。

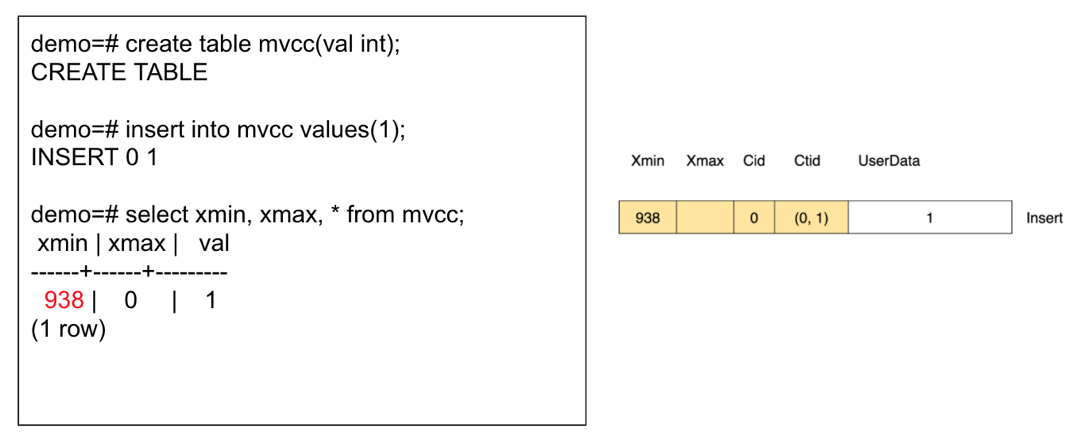
上图下面的数据中，xmin和xmax都有值，这意味着数据在被插入后，已经被某些事物标记为删除了。判断Tuple是否可见的规则就是两条，xmin已提交，xmax未提交，便是可见的。xmin对应的事务已提交，也就是说创建它的事务已提交，创建这个动作被认为删除这个动作还没发生，因此它是可见的。

和快照中的数据做对比，xmin都是30，小于快照中的xmin40.xmax80，小于快照中的xmax100，Running Array没有，因此，这个数据便已经被删除了。第二行中的75，正在Running，因此它是可见的，120大于快照中的xmax100，此事务未被提交。

接下来看几个例子。

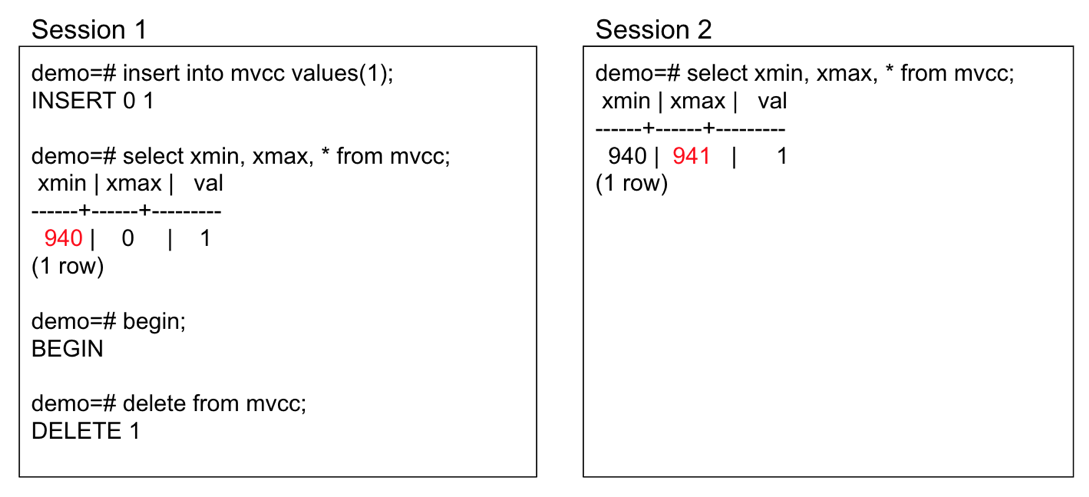
1：Insert

首先我们创建了一个Table，Select后我们得到了xmin，xmax的值，938和0.这表示该Insert操作时被938这个事务创建的。

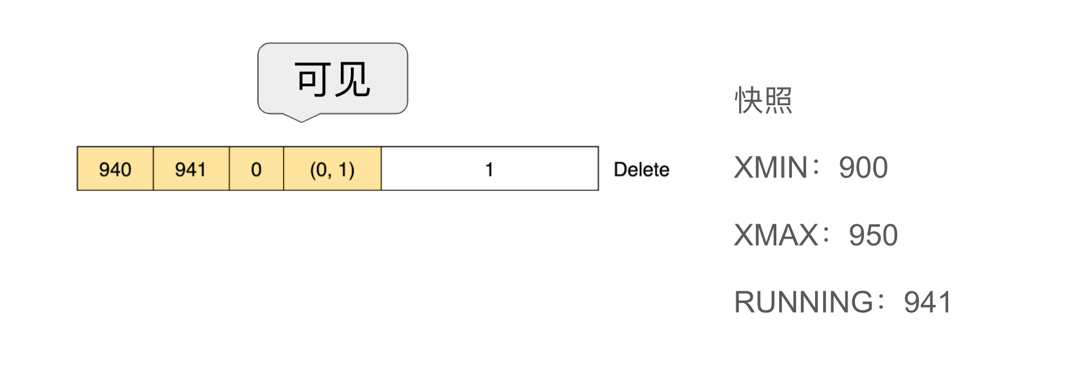


2.Delete

我们再看delete。首先我们给MVCC这个表Insert了一条数据，在将其删除，不提交，如Session1所示。然后再起一个Session2，select后发现，我们是可以看到这条数据的，并且xmax值为941.删除这个事务未提交，也就意味着他对于其他事务是不可见的。

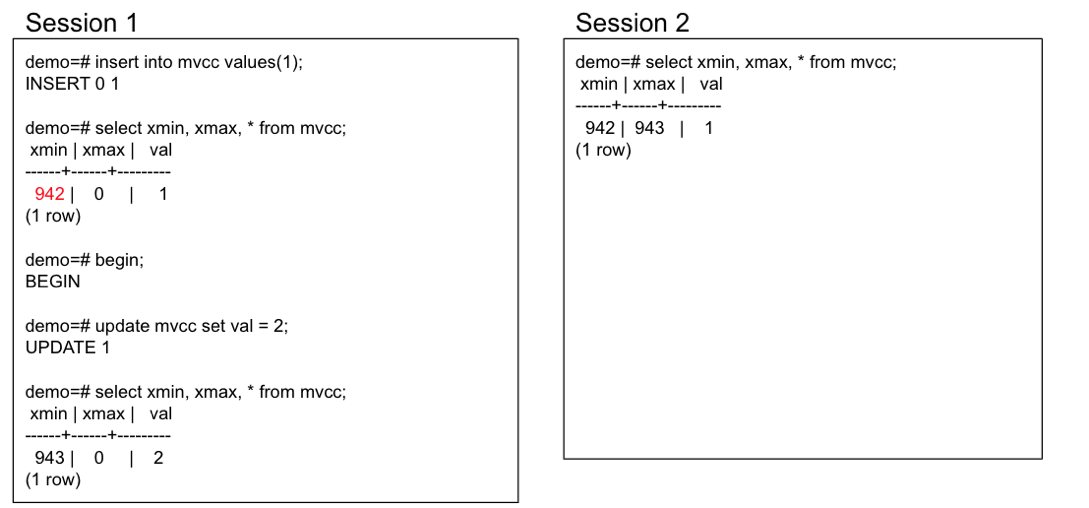


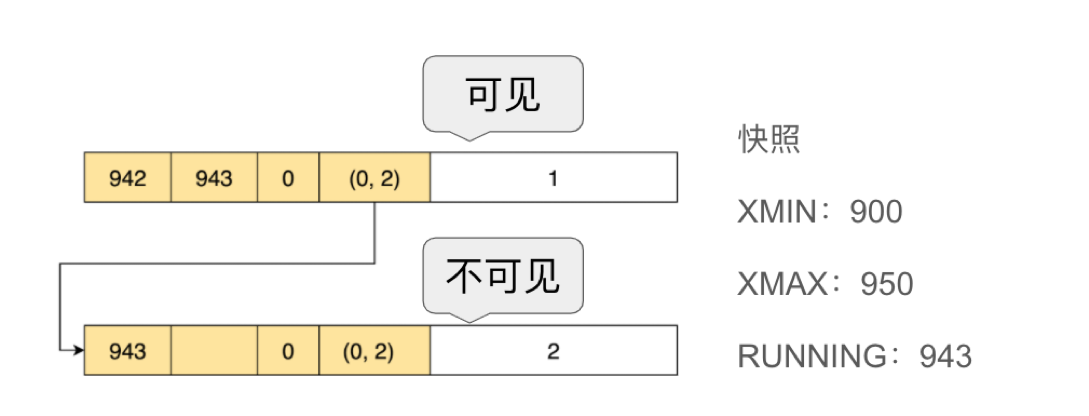
下图更好的反应了事务的状态



3.update

我们再看一下update的情况，我们再Session1中首先Insert一个数据1，然后将其update成2，并不提交。在Session2中，我们发现数值仍然是1.这是因为update的事务还没有提交，所以update这个操作对于别的事务来说不可见，因此值扔为1。此时，我们看到的是旧版本的数据，其xmin的值为942，xmax为943，而Session1中的新版本的数据，xmin的值为943，xmax为0.update的时候，并没有把旧版本的数据删除，而只是写了一个新版本的数据，同时把这条数据的事务ID的写到旧版本的xmax和新版本的xmin上，并指向新版本。



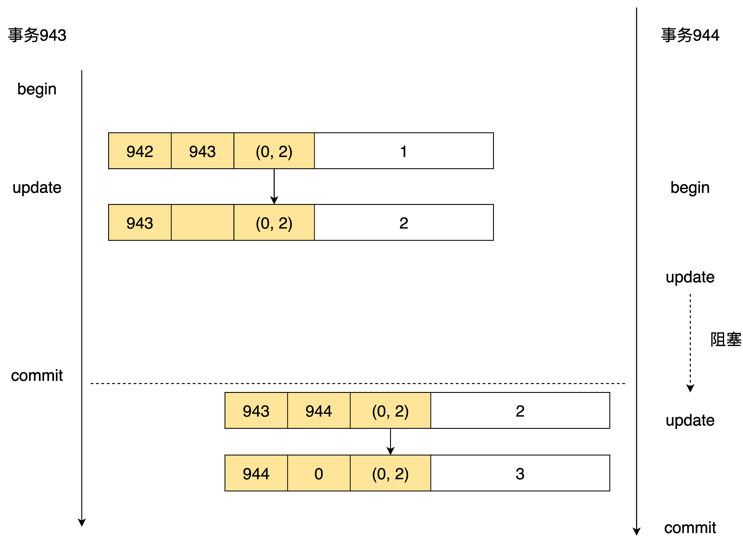


**并发的update**

并发的update是指向两个事务都要对同一个tuple做update。在前面的内容中我们提到过，读和读一定是不会冲突的，二是写和写一定是冲突的。因此并发的update就一定会发生冲突。

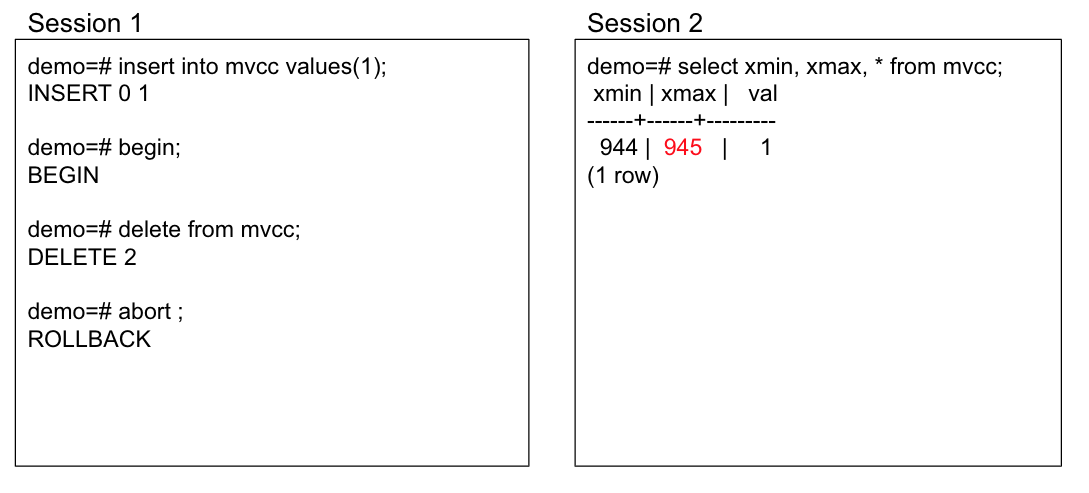
在发生这种冲突时，Greenplum是如何处理的呢？如果发生了并发的update，那么先进入的事务，会成功的update这条数据，后进入的事务，会在需要update这个数据上阻塞住，直到前面的事务提交或回滚。

大家请看下面的例子，首先事务943把这条数据从1update成了2，没有提交。此时，事务944也试图update这条数据。因为事务944开始执行时，会拿到快照，由于943还未提交，因此它只能看到旧版本的数据。它会发现xmax被标记，也就是说这条数据要么被更新，要么被删除了，并且xmax对应的事务还没有提交。由于事务943接下来可能会提交也可能会回滚，因此，数据库的update将无法被完成。此时只有等事务943 commit后，事务才能继续执行。一旦commit，就可以顺着ctid指针找到这条新的数据，再将新的数据update。



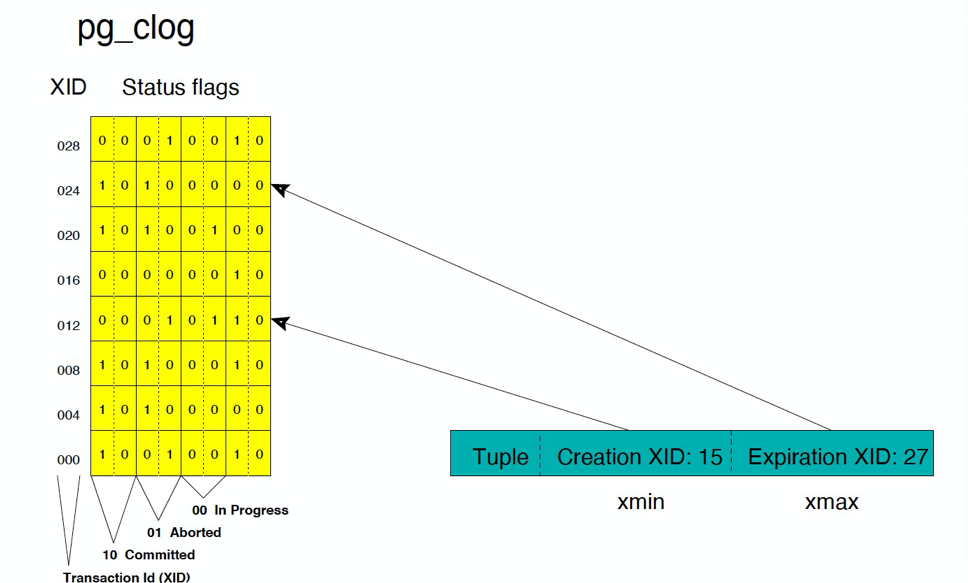
**Abort**

我们刚讨论了事务的两种状态：InProgress和Committed，Abort是事务的第三种状态。当事务失败时，就需要回滚。



下面的例子中，在MVCC表里添加一条数据，接着启动一个事务，将这条数据delete掉，接着Abort。此时，从Session 2去查这个表时，我们发现这条数据还在，因为我们是在一个事务里先delete，再abort。也就是说delete这个事务的影响不该发生。但是由于在delete时，是将事务id写在xmax里来进行标记这个事务删除了这一条turple，在下图中也确实看到xmax被写入了。

那么Abort发生时，是否应该将xmax改回去了呢？答案是在Greenplum里，不需要把它改回去。那么如果不改回去，是否会出现问题呢？



首先我们回忆一下快照。快照只能处理两种状态，一种是inpress或者叫running状态，另外一种committed的状态。也就是说快照只能判断是否是running状态，并不能跟踪具体是Committed的状态还是abort状态。此时，便需要通过查pg\_clog来实现跟踪。

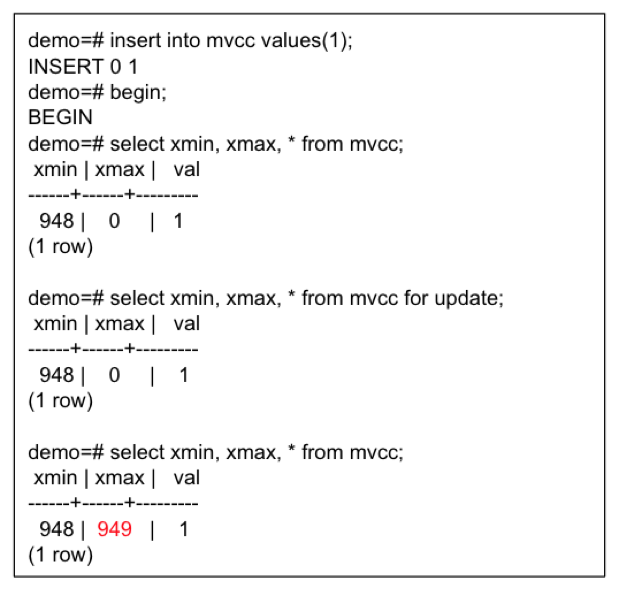
pg\_clog是纪录事务状态的一种log。它本质上一个数组，数组的每一项只有两个bit，通过0、1的组合,标志了3种状态，00是in progress，01是aborted，10是Committed。通过查询pg\_clog，便可以知道这个数是不是回滚，从而补充了快照。如果一旦是abort的状态，便认为这个事务所造成的一切影响是不存在的。

每扫描一个turple，我们都需要判断其可见性，但如果每次都需要先通过快照来判断是否in progress，再通过commit log再判断一下是否Aborted。虽然他们都存在内存里，但是快照和commit log是两个很大的数组，会导致查询速度不够快。

这里Greenplum引入了一个优化项，叫标记。t\_infomask有8个字节，分别代表了8个标记，xmin Committed代表了xmin所代表的事务已提交，xmin Invalid代表xmin所在事务已回滚；xmax Committed和xmax Invalid同理。

有了这些标记，一旦访问到这个turple，并且查询了它的可见性，这些标志就会立刻被更新。因此这个tuple只会访问一次快照，或者commit log，就会完成这些标记位的填充，后面就不在需要访问快照或commit log，仅需要判断标记位，这是Greenplum针对可见性速度的优化。

xmax除了标记删除的作用，还有行锁的作用。例如下图中的select for update，需要把select的数据锁住，再去做其他的操作。



当把xmax作为行锁，我们需要记一个标志位在t\_infomask里。标志位HEAP\_xmax\_EXCL\_LOCK用来表示xmax是用来做行锁而不是表示删除。

|  |
| --- |
| MVCC实现总结  Xmin：创建元组的事务ID  Xmax：使元组过期的事务ID，由update或delete语句设置。在某些情况下，用作行锁  Cmin/cmax：用作标志事务内的可见性。如果在一个事务内先创建再过期，则使用combo Command id |

**MVCC的清理需求**

前文中，我们提到过MVCC的优点在于它是有很高的并发访问性能，但同时它还有一些缺点，即MVCC在以下情况会产生很多的垃圾，还需要清理。

在更新元组时会创建一个新的元组，所以旧的元组需要清理

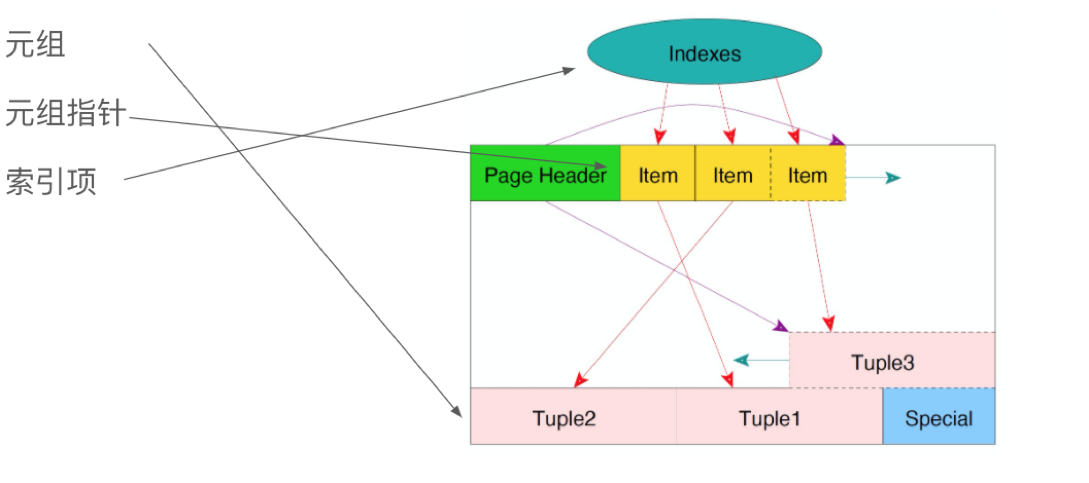
在删除元组时，只会标记xmax，不会立即删除元组

失败的事务所创建的元组

那么何时需要执行清理呢？一共两个时机

时机1：在查询操作访问到某个页面时，会清理这个页面

时机2：通过vacuum操作来清理

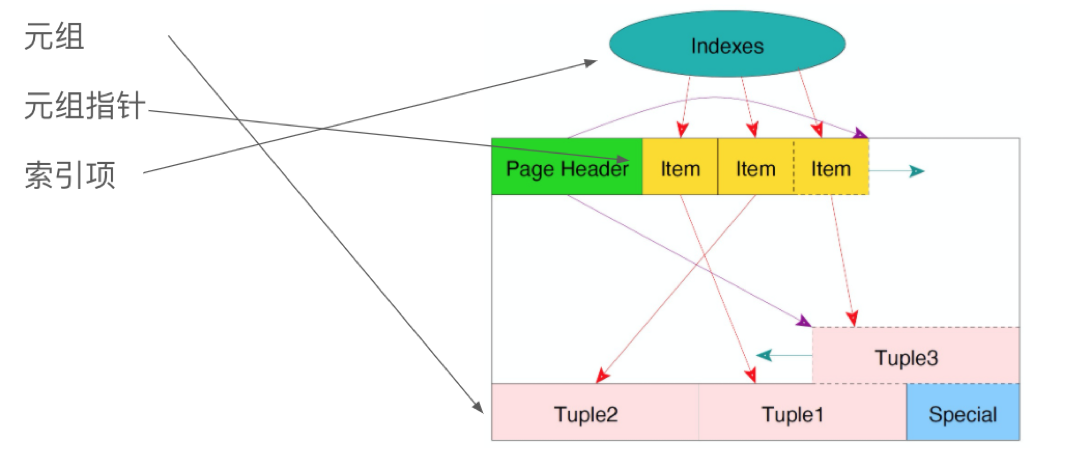


前文提到，Heap表页面布局中提到Heap是由一系列的大小相等的page组成的。清理时可能会面临另一个问题，它之上可能会存在index，需要清理的项共有3项

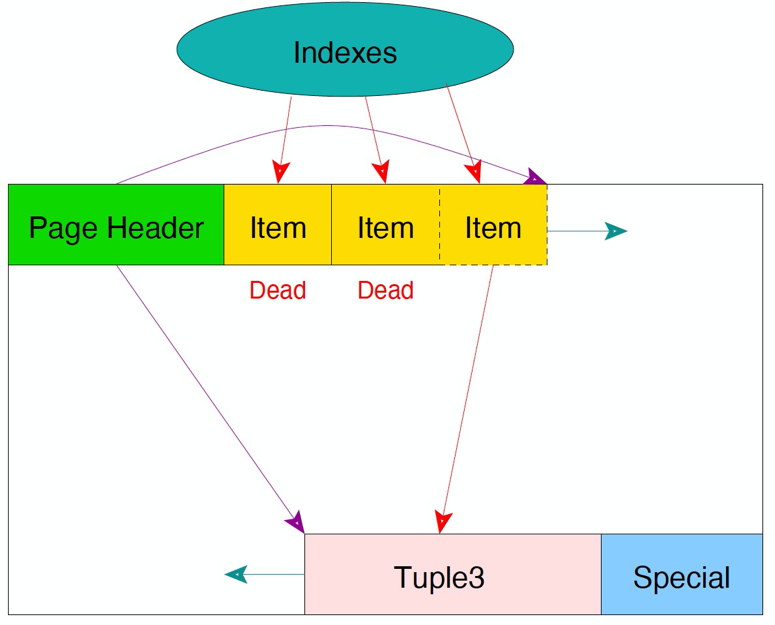
1：被标记删除的tuple

2：指向这些tuple的Item

3：Index的某些项



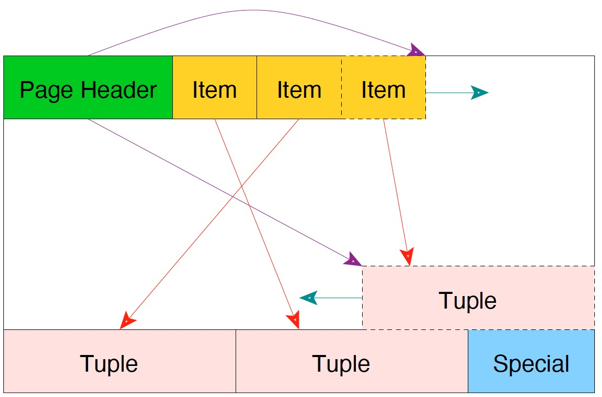
单页清理不会清理index，只会清理tuple，因此index指向这些Item的指针都还在。而vacuum则会把这三项全部清理。



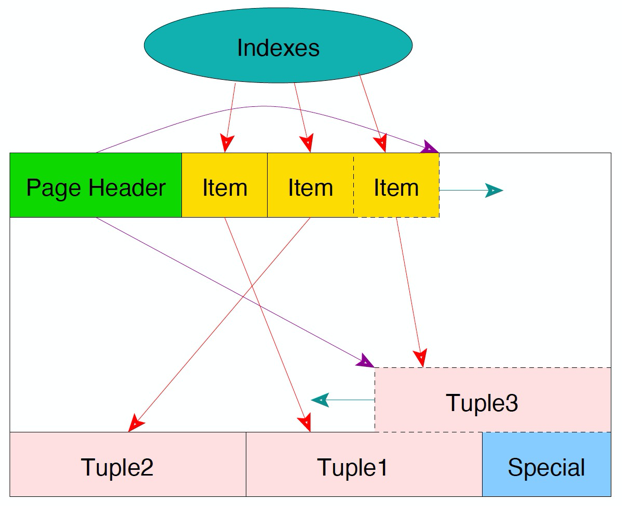
Update中产生的旧元组可以像被删除元组一样处理。有些元组的Item也可以被回收，使用Heap only tuple的话会甚至可以直接复用被删除的tuple。

如果每次更新都需要插入索引项，会导致索引的膨胀。我们使用链式更新机制（HOT）来避免插入新的索引项。使用HOT机制的条件是：更新不涉及索引项，且新旧元组在同一个页面内。

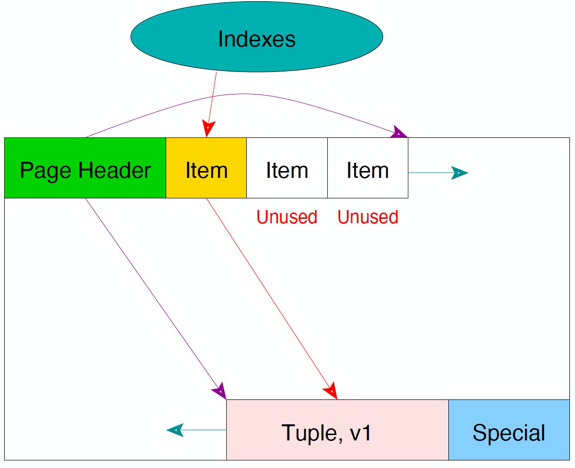
在下面的例子中，tuple2和tuple3被标记删除，对任何事物均不可见。



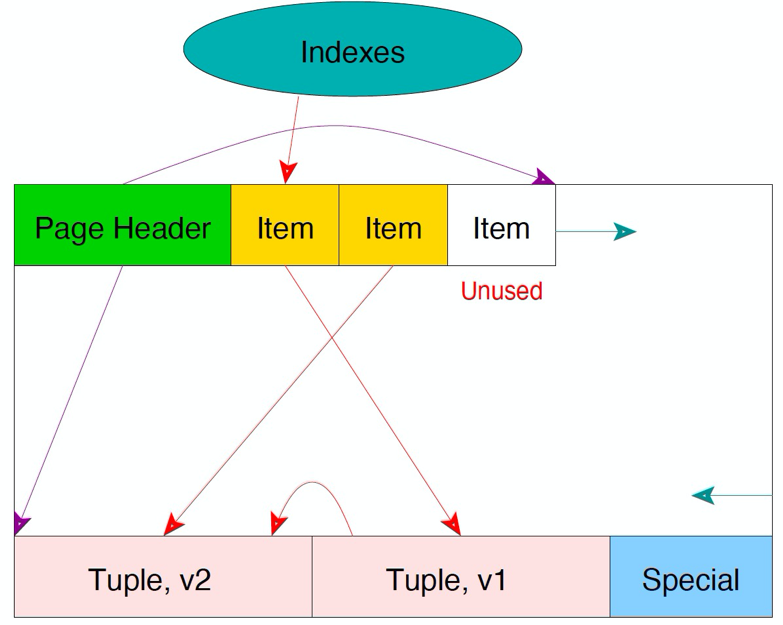
此时，当访问到该页面，就会发生单页面清理。其清理思路将tuple2和tuple都删除，由于空间区间要保持在中间，因此会把tuple3挪动到最后。



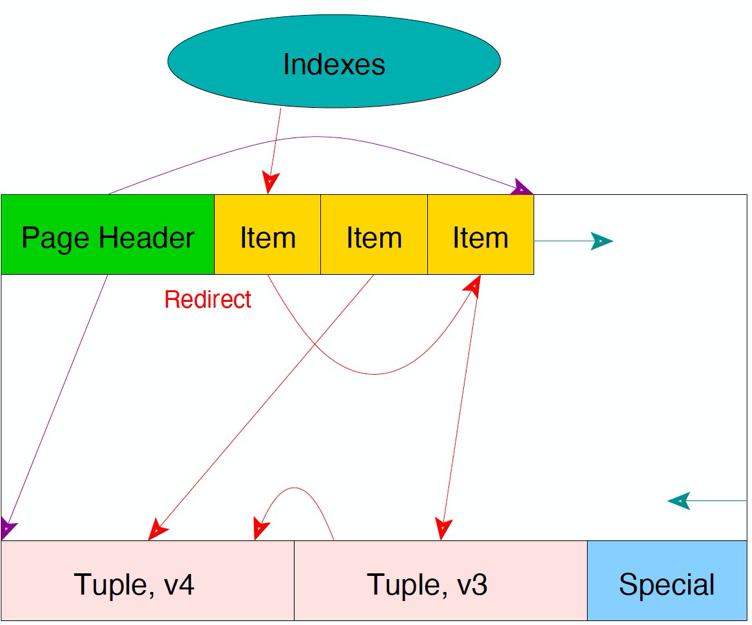
在单页清理中，不会清理索引项。所以在Heap中，只会清理tuple，而其Item会被标记为Dead。在vacuum操作中会清理索引项，并把对应的Item标记为Unused，这样Item就能被继续复用。



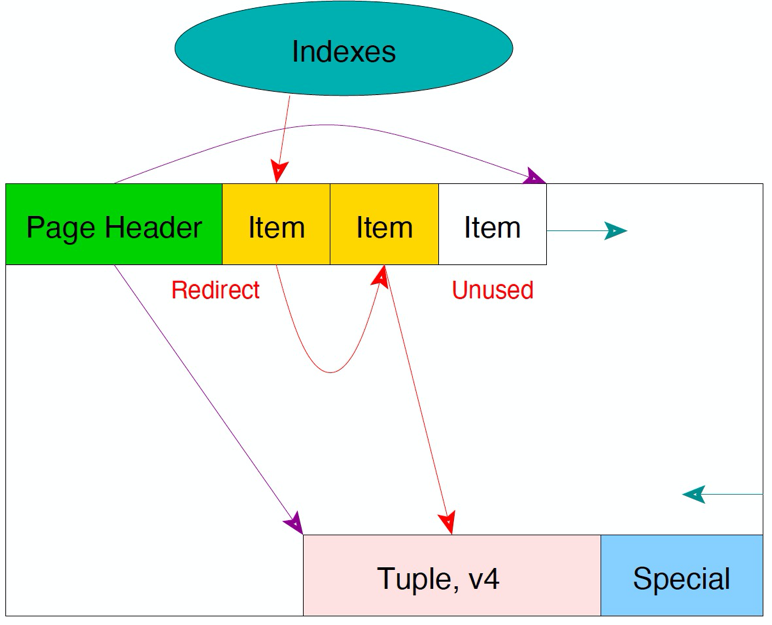
上图中，我们有个version1的turple，并在下图中更新到version2.这个过程不会新增index项，而是通过旧tuple的c\_tid指向新的tuple。



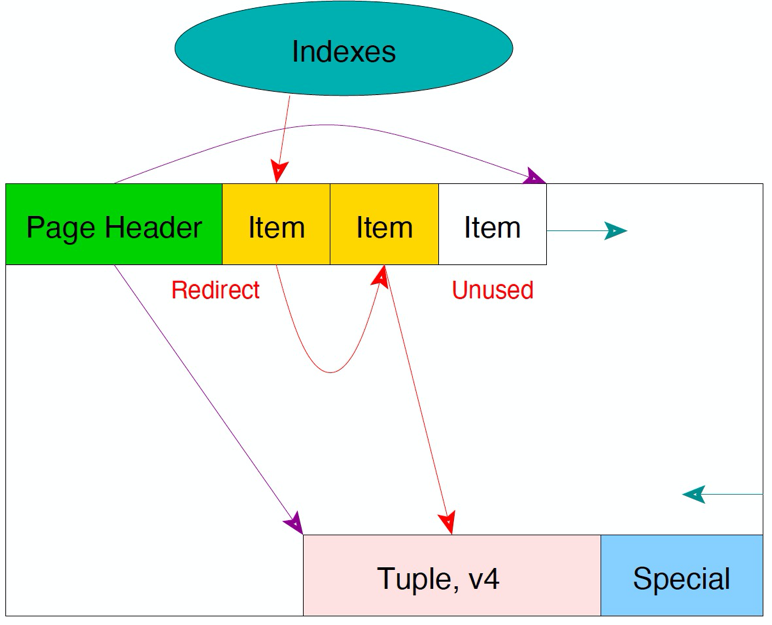
再update出现了version3，如果此时version1的turple不可见，version3的turple将取代version1.



再update出现version4的tuple。第一个tuple将指向version 3的item，version3的item指向version3，version3指向version 4。



HOT链的首尾节点保留，中间的所有节点都是可清理的。



需要注意的是：如果更新时更新了索引列，则不能使用HOT，仍然会引起索引膨胀。

|  |
| --- |
| 清理工作的总结  清理只发生在对于任何执行中的事务都不可见的tuple上  HOT发生在处于单个页面，并有相同的索引值的tuple上  多数的清理工作由单页清理操作完成，但是单页清理只涉及 Heap Page  Vacuum则会进行更加彻底的清理，包括tuple item index |