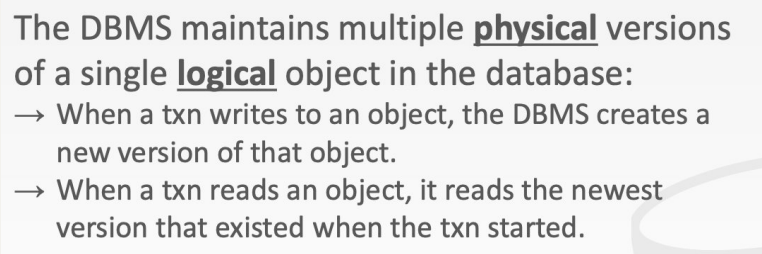
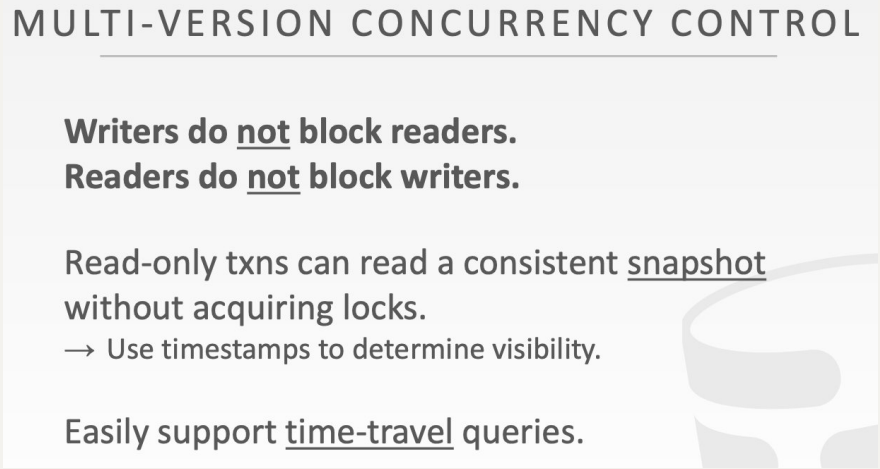
很少有DBMS单独拿MVCC来实现并发控制，MVCC更多地是和2PL，T/O，OCC这些结合起来使用，以达到增强的效果

MVCC中的Multi-Version，即多版本，说的是对于数据库中每一个对象/每一条数据，DBMS可以记录它的多个版本，就像git一样

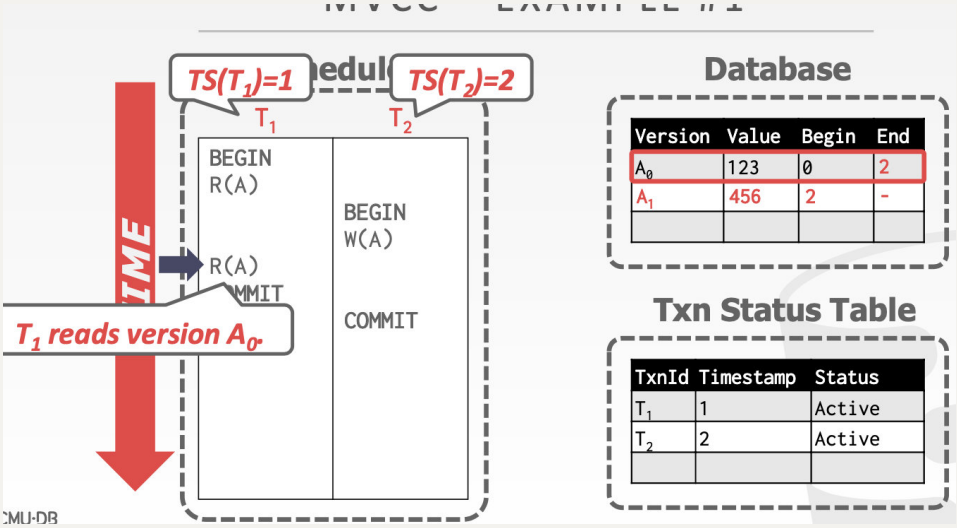


MVCC的宗旨与理念，或者说是它存在的原因是：2PL协议中，一个事务更新了一个对象之后，其他的事务就没有办法读这个对象了，直到这个事务提交；而MVCC的基础思想是，留下数据的历史版本，这样其他的事务可以读历史版本而不是被阻塞。MVCC给数据库中的对象都记录了它的多个版本后，只读的事务就可以在无锁的情况下读它所需要的那个版本的一致性快照，不受数据库动态变化的影响，而且DBMS一般使用事务的时间戳来决定版本号



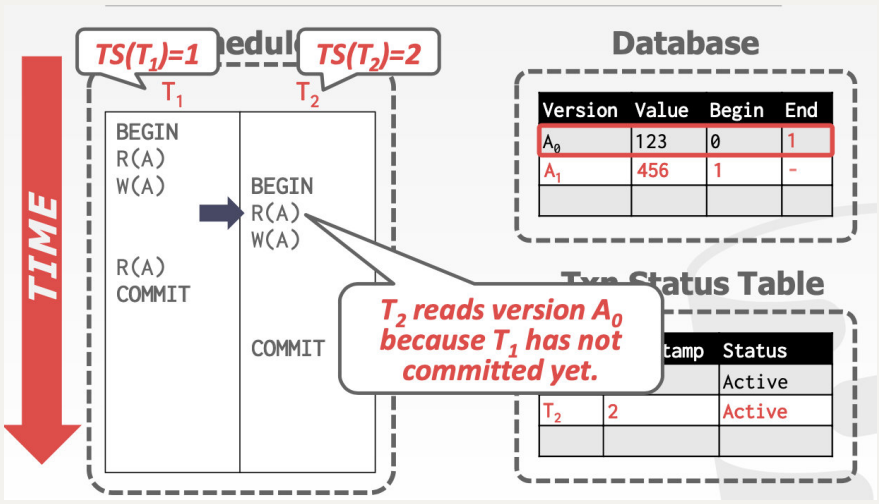
结合如下的例子理解：

MVCC的实现里也会维护一个全局的事务状态表，便于事务之间互相查询对方的状态（下图Txn Status Table）



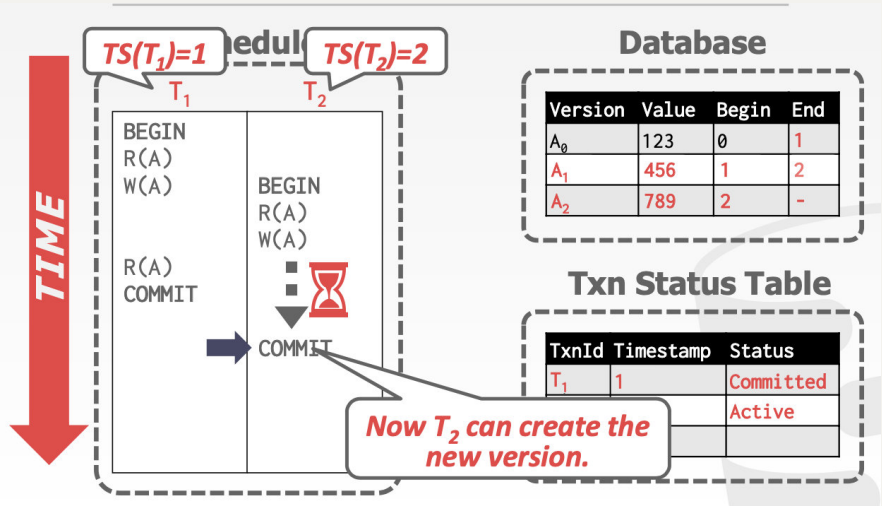
事务T1第二次执行R(A)时，根据它的时间戳1去数据库里面找[Begin, End]区间里面包含1的版本，这属于是读历史快照。因为MVCC记录了历史版本，所以上图的两个事务得以无锁地进行下来

再看一个例子：



T2执行R(A)时不会去读`Begin=1`这个最新版本的数据，因为此时T1的状态是active，还没有提交，所以读到的可能是脏数据，如果T1后面abort了，那就会导致级联回滚或者是执行调度的不可恢复

T2之后需要等到T1的状态变为committed再执行W(A)，因为存在未提交的新版本。T1第二次执行R(A)时可以去数据库中读最新的版本，因为这是它自己完成的更新。



这个执行调度并不是serializable的，因为T2并没有在T1提交更新的基础上进行操作，因此只依靠MVCC是不能达到调度的串行化的

MVCC不只是一个并发控制的手段/协议，它可以说是完全改变了DBMS管理事务的方式，对传统的并发控制理论实现有极大的增强

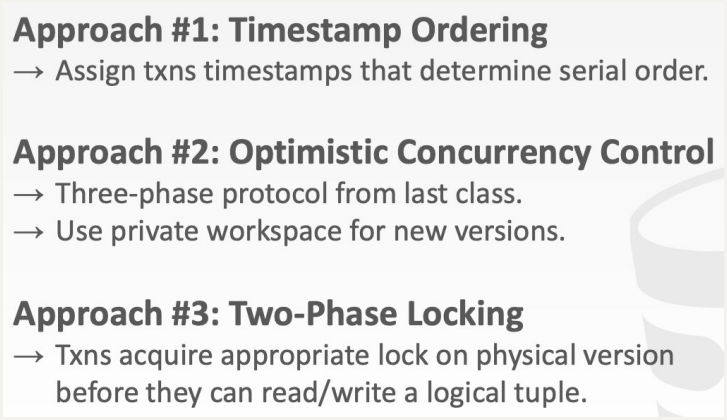
> MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.

本Lecture接下来将研究MVCC的几个设计重点

Concurrency Control Protocol

并发控制手段/协议

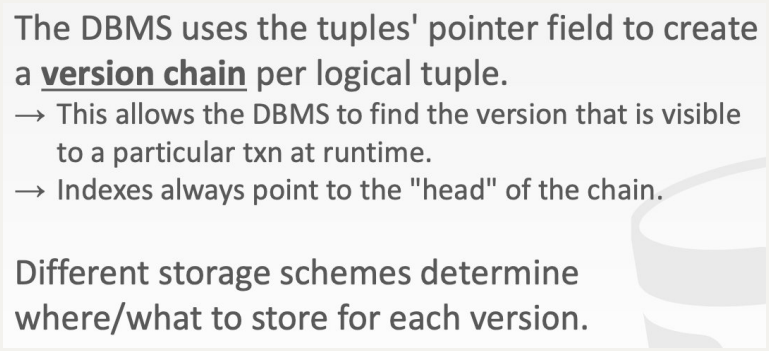
只使用MVCC无法做到serializable的隔离级别，因此会和如下的其他一些协议结合在一起：



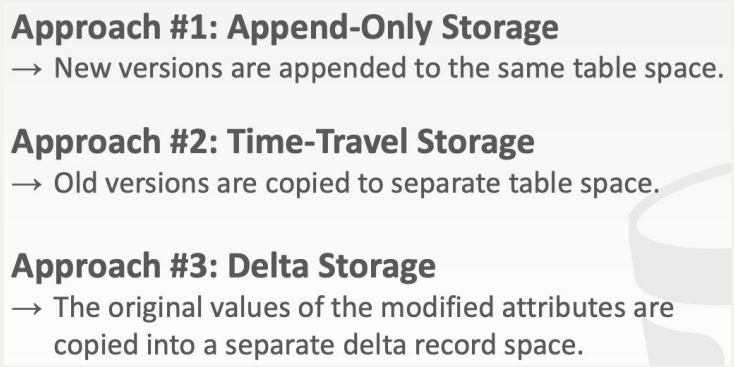
Version Storage

版本存储

DBMS可以用tuple对象所包含的指针，建立一个包含各个版本的链表，因此DBMS可以在这个链表上去寻找各个版本，如果对表构建了B+树之类的索引，那么索引最后指向的是版本链表的头节点

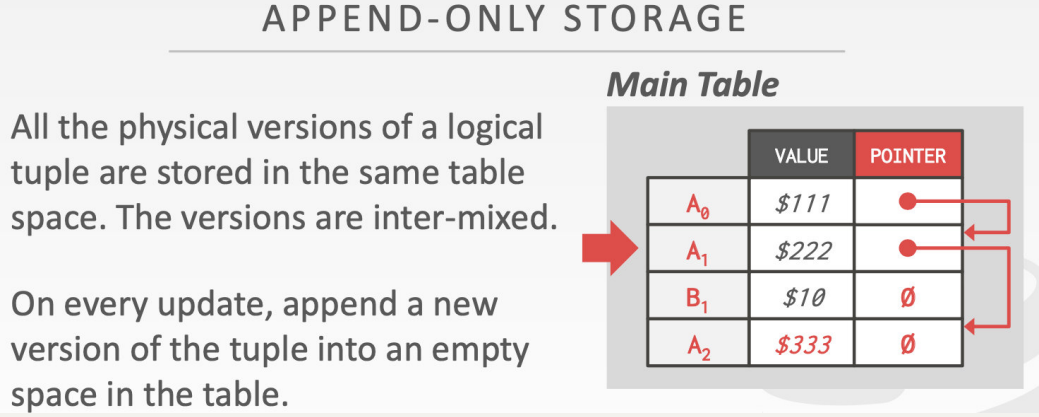


有如下三个方法去具体实现版本存储；

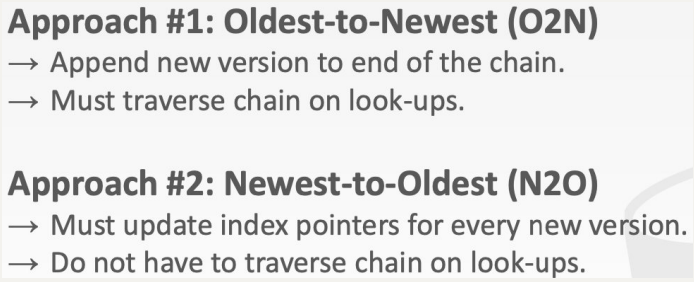


- 简单追加（Append-Only Storage）

新版本的tuple被追加到表中，一个logical tuple的多个版本通过链表被连起来



实现简单追加有两种思路：把新的版本放在链表末尾/把新的版本放在链表开头

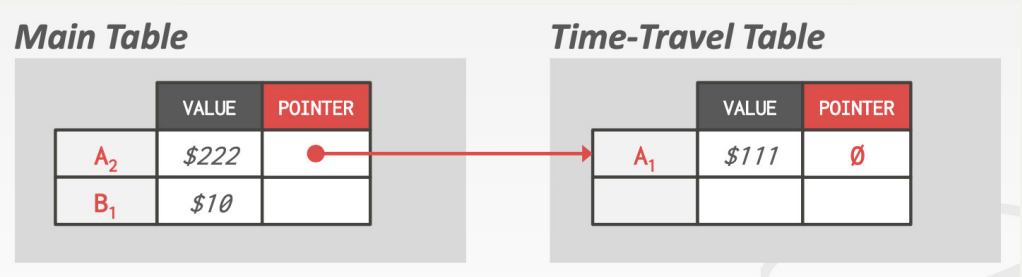


如果把新的版本放在链表的末尾，那么经过索引查到的是最老的版本，需要经过多次的链表内跳跃才能得到最新的版本。也就是说，这么做的话，追加新版本的时候开销小，但想要读取快照，即执行查询时搜索对应版本的时候开销会大

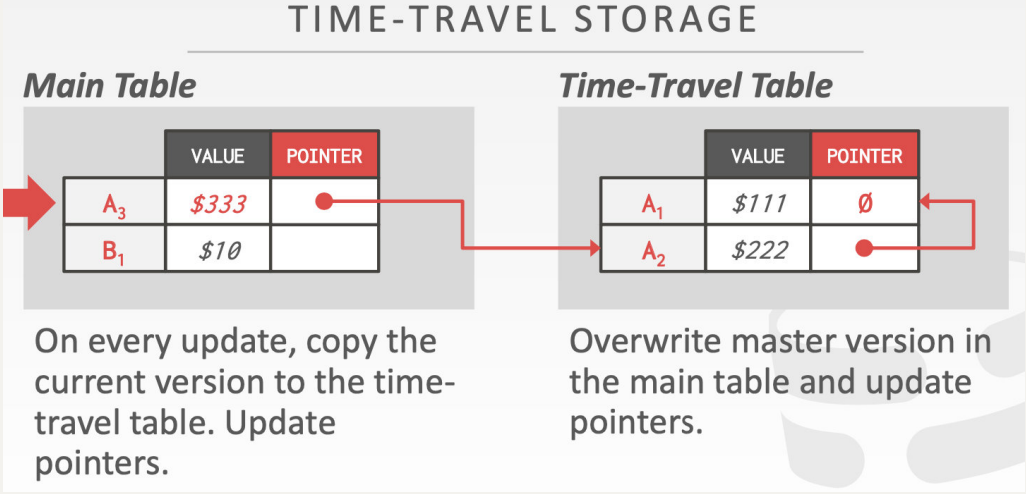
反之，如果把新的版本放在链表的头部，那么执行查询，搜索相应版本的时候效率会高，但追加新版本时维护链表的工作量大，因为要更新索引结构中的指针

- Time-Travel Storage

  单独建立一个额外的表用于存储历史数据，主表（Main Table）存储的就是当前最新的数据，历史数据在单独的Time-Travel Table（历史表）里。当事务对数据进行更新时，也就是产生新版本时，DBMS会把旧版本的对象拷贝至历史表中，并且在历史表中维护好串联起多个旧版本的链表，之后在主表里写入新版本的数据，覆盖旧版本，然后再修改主表中相应的tuple的指针，令其指向历史表里最新的历史版本



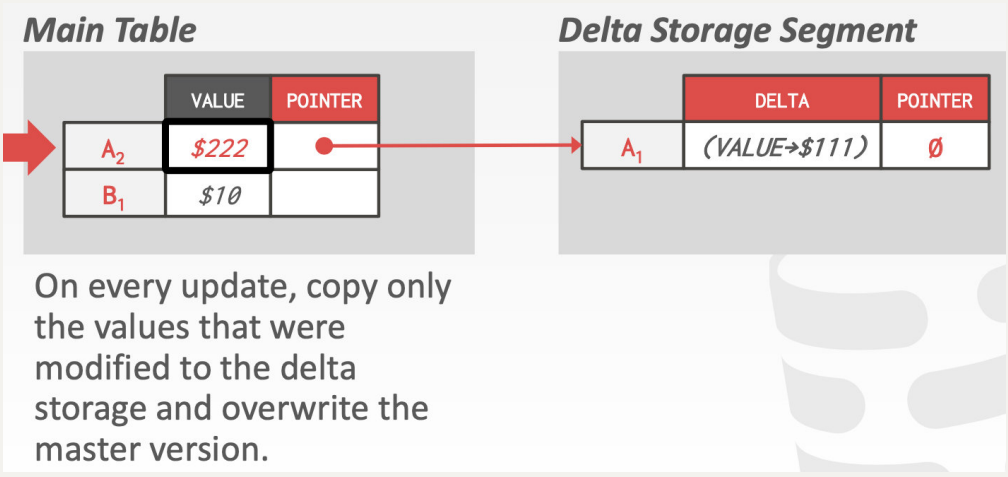
->next->...->（详见slides）



- 存储增量（Delta Storage）

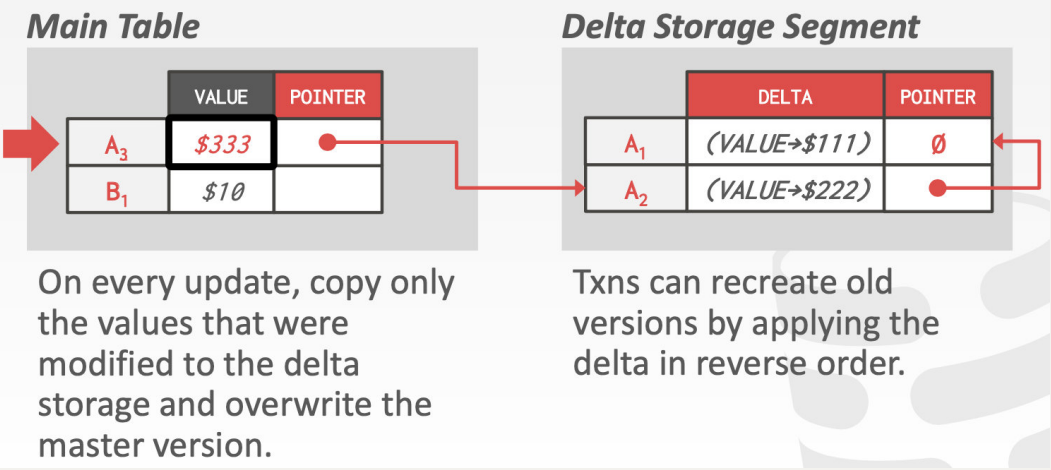
  总体上和Time-Travel Storage差不多，但做了一点优化

  实际的数据库的表中，每个tuple不止有一个attribute，而事务对tuple的一次更新可能只是修改了众多attribute里的一个，因此在历史表里每次都追加一个完整的tuple，这未免有些浪费存储空间。因此就有了只存储每次数据更新时的增量的idea，如下所示



  按照Time-Travel Storage的思想，更新了A1那行的tuple之后应该把A1版本的tuple这一整行都追加到历史表里，但根据delta storage的思想，只需要把A1版本相较于当前版本的变化拷贝至历史表里（delta stroage策略下，这个表叫Delta Storage Segment，即增量存储段）。如果后面有访问A1这个历史版本的需求，就可以通过读取增量存储段中所记录的信息恢复到A1版本

  如果后续又有更新，那么就需要继续往增量存储段里追加增量（笔者认为这和日志机制挺像的）



这样做就可以减少存储历史版本数据所需要的空间，相应的坏处就是，如果想查找历史版本的话，要通过增量存储段做恢复。这是一个用时间换空间的策略，MySQL采取的也是这个方法。

Garbage Collection

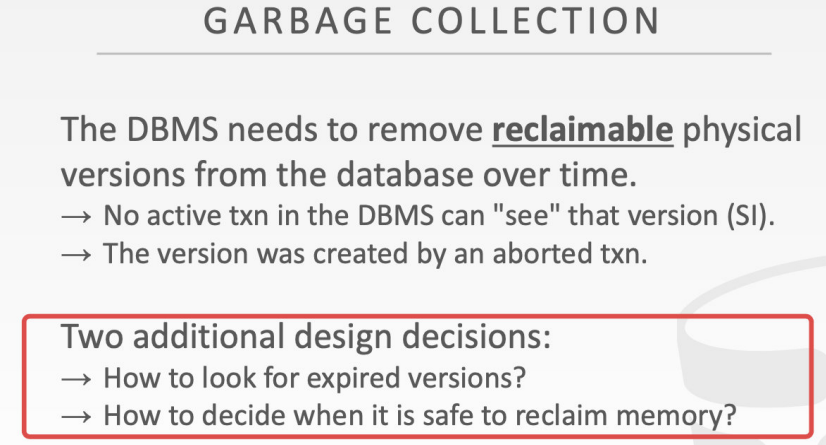
垃圾回收

数据库不可能无限地存储各个数据的历史版本，否则就会导致数据库爆炸，存不下那么多的数据，因此DBMS需要把那些已经没有用了，作废了的历史版本删除，这样就可以保证数据库文件不会太大。这就是MVCC里的垃圾回收，它的两个宗旨如下：

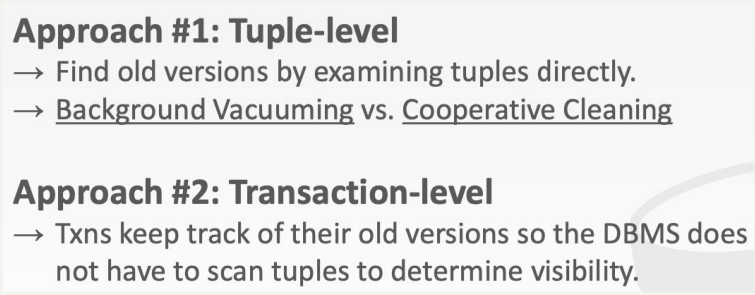
- 如果任何active状态的事务都看不到某个历史版本（比如说现在的事务的时间戳都是10以上的，但这个历史版本的时间戳/版本号是1），那么这个古老的历史版本就再也用不上了，就可以把它删除
- 如果某个事务创建了某个历史版本，但这个事务后来回滚了，那么这个历史版本也是无用的，可以删除

在实现垃圾回收时，要解决如下两个问题：

- 如何发现无用的历史版本?
- 什么时候去删除无用的历史版本?



MVCC中的垃圾回收有如下两个实现思路



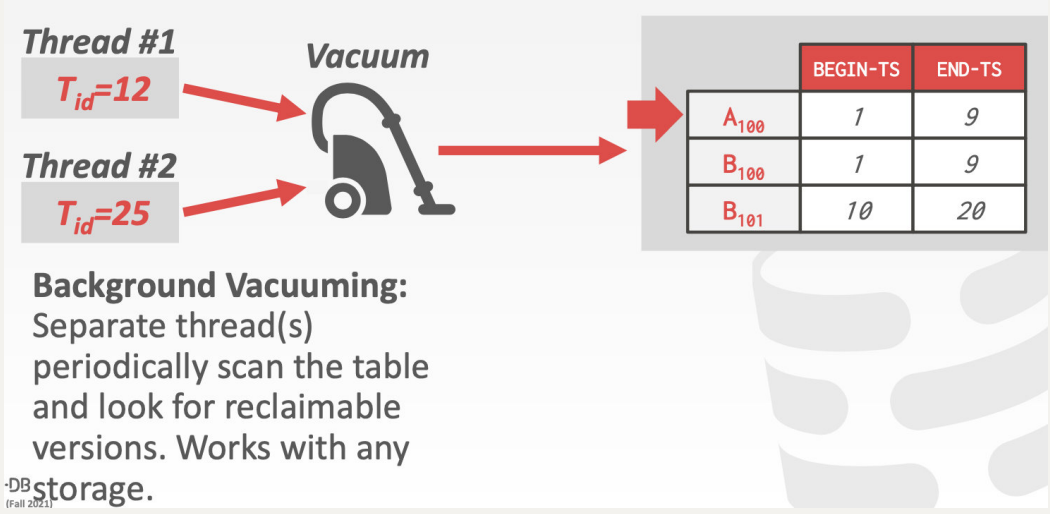
- 以行记录为单位（Tuple-Level）的垃圾回收

  就是直接去检查tuple，发现无用的tuple历史版本。

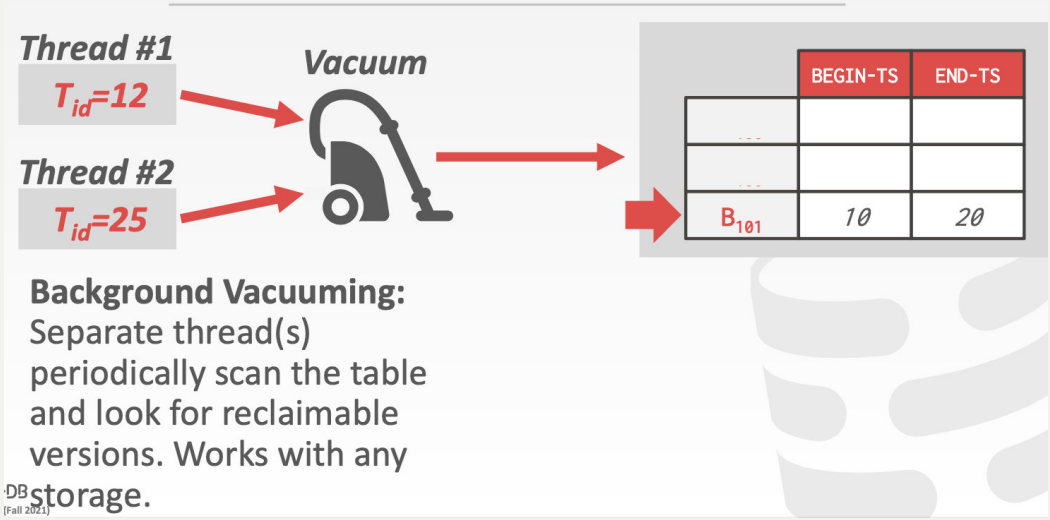
  有两个实现方案：

  - 后台清理

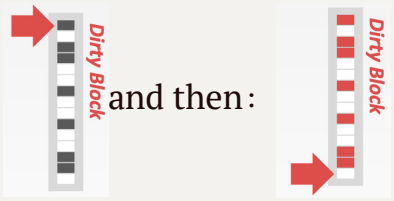
    后台会有一个单独的线程每隔一段时间就扫描历史表（或同类的其他表），然后结合当前active的事务的时间戳去分析表中哪一个版本是无用的



->next->



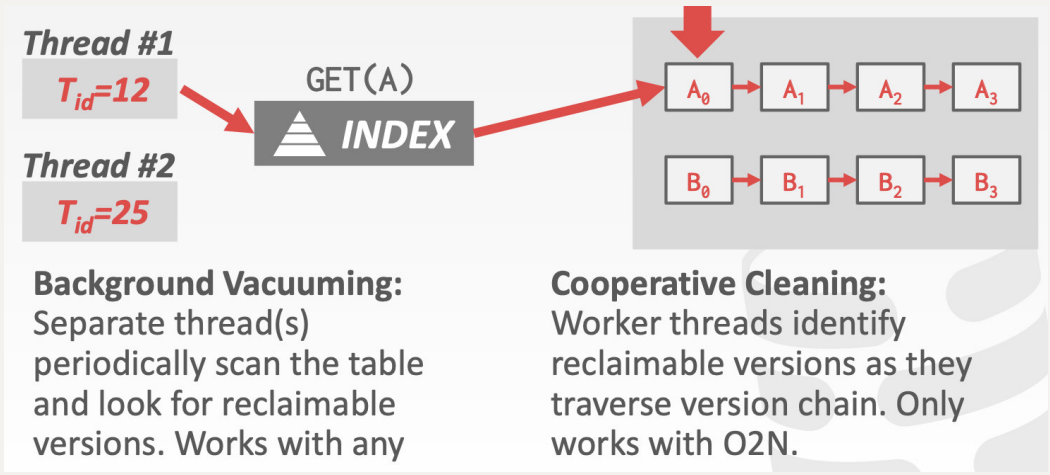
后台清理也有一个优化手段：后台的GC线程没必要扫描整张表，对每一个历史版本进行判断。DBMS可以维护一个dirty block bitmap，去跟踪自从上次垃圾回收至今表里有哪些页被事务修改过，因此下一次垃圾回收的时候，只需要扫描自从上次GC以来被修改过的页面



Postgres中就是使用了这样的优化策略

- 合作清理

  事务执行SQL查询语句时需要通过索引来访问相应的tuple及其历史版本，在这个过程中可以顺便扫描tuple，把无用的历史版本删除。这样的话就不需要额外的GC线程来完成垃圾回收，而是各个事务在执行的时候都会做一点垃圾回收的工作。

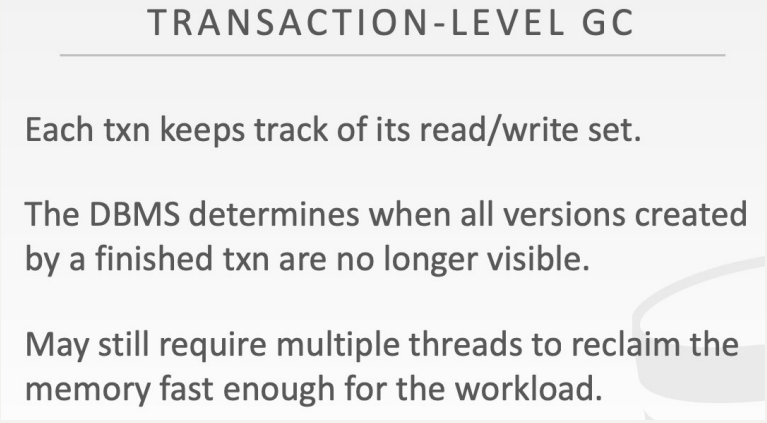


->next->

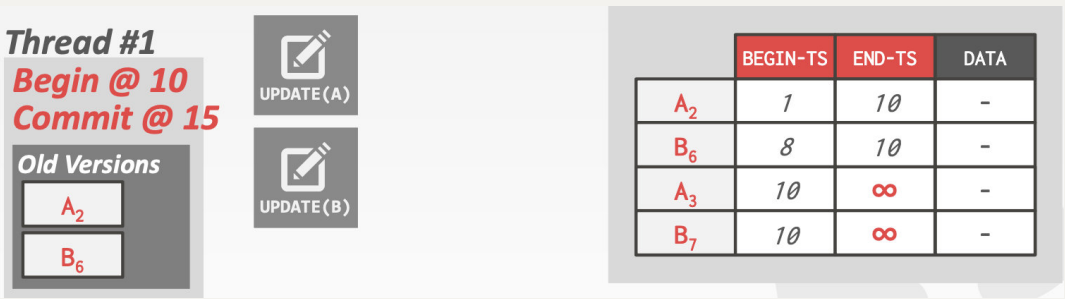
->next->

- 以事务为单位（Transaction-Level）的垃圾回收

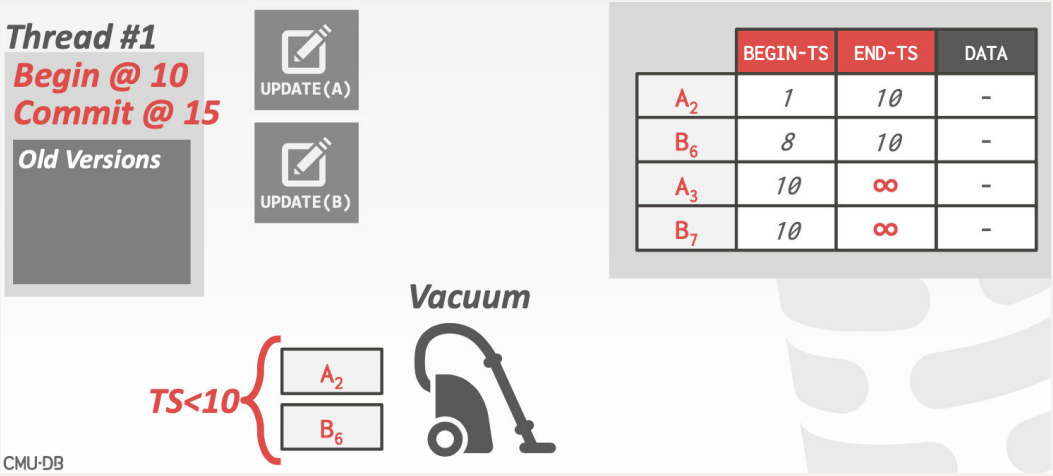
  DBMS对每个事务会记下来它读/写了哪些东西，除此之外还会记录因为这个事务所进行的更新所导致的历史版本（what will be the versions that invalidated by the update of this transaction），基于此，DBMS便可以在一个事务过期了（out of scope）之后把这些东西都删除



  结合例子理解



  and then:



在thread 1所代表的事务提交时，它顺便会把A2和B6这两个旧版本,以及它的Begin时间戳与Commit时间戳提交给DBMS的垃圾回收器

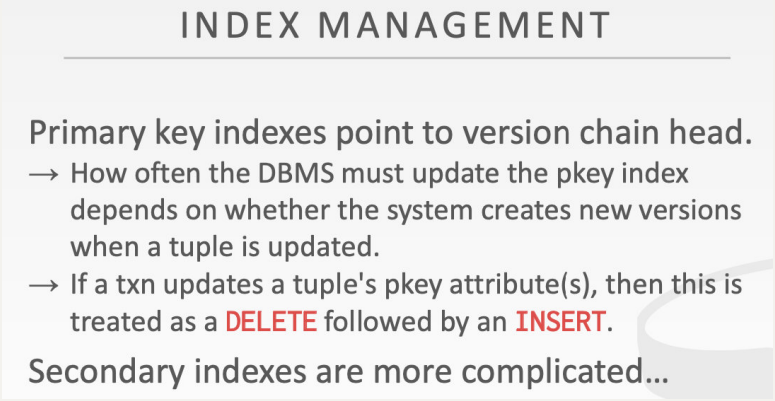
以事务为单位进行垃圾回收时，垃圾回收器不需要扫描一个个的tuple，而是只需要检查已经完成了的事务。在上图的例子中，如果DBMS发现所有正在执行的事务的时间戳/id都是大于15的，那么就可以判断出commit时间戳/id小于15的已提交的事务所创建的版本，以及它所导致的旧版本都可以被删除，因为不会再有事务来访问它们

（以上三段话是对2021年CMU15-445的Lecturer-Lin.Ma的英文原话的翻译）

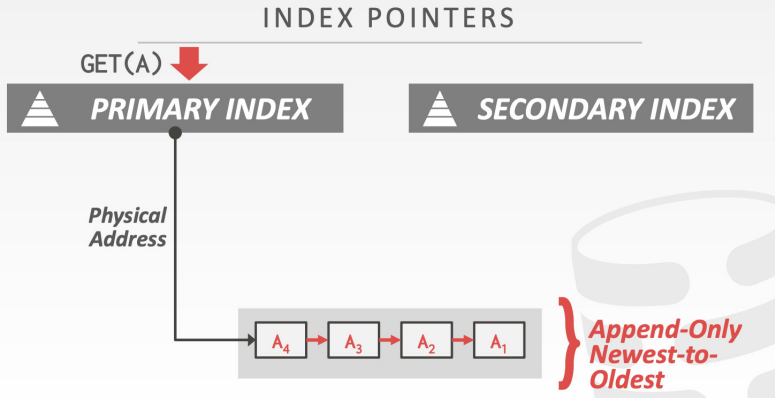
Index Management

索引管理

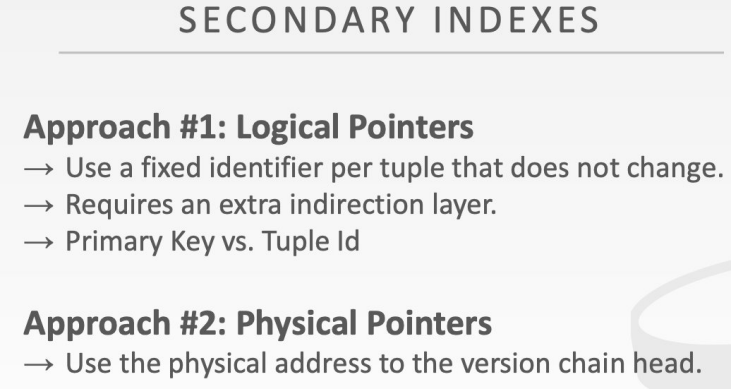
接下来要介绍的是，在多版本的情况下，索引该怎么管理



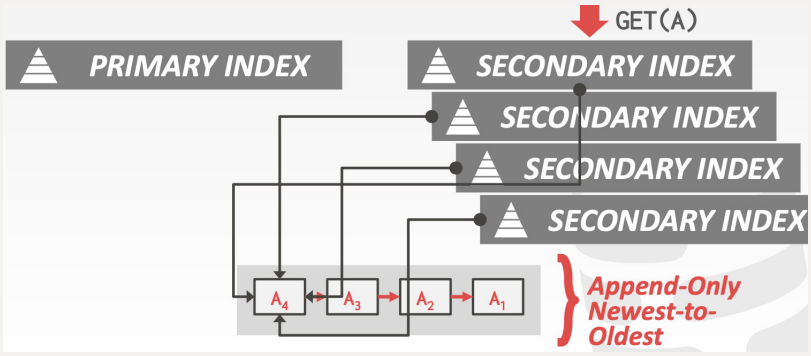
一般情况下，主键索引相对比较好处理，因为主键索引会指向版本链表的头部，而且是通过物理地址（在哪个页的哪个slot）来定位的（如下图）。在新的版本产生后，主键索引可能要更新它所指向的物理地址。如果想修改某个tuple的主键，那么就要完成一个先删除再插入的操作。



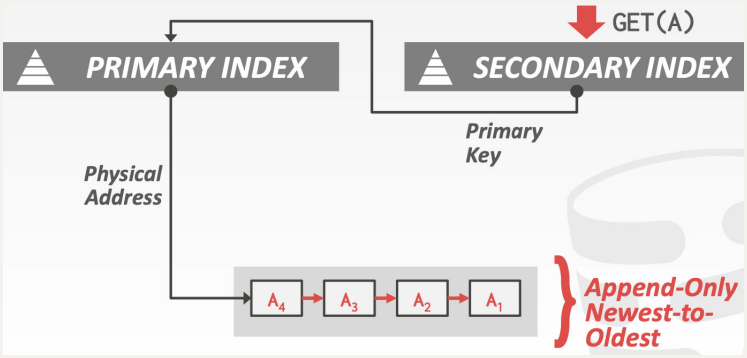
对辅助索引（secondary indexes）的处理要更麻烦一些



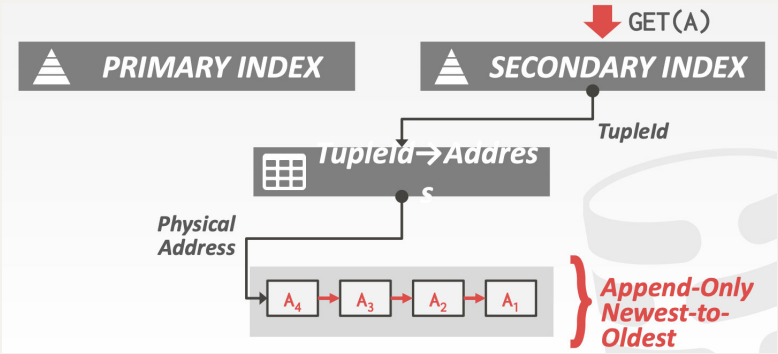
DBMS做辅助索引有两个流派，如上图所示。无论是哪个流派，索引的Key就是表里对应的那一列的attribute的数据，这两个流派的不同之处就是索引里的Value不同。Logical Pointers策略中索引的Value是逻辑地址，比如说主键/行id的值，这个策略的缺点是如果想拿到完整的tuple就需要再通过主索引找到相应的tuple；Physical Pointers策略就是对这一缺点进行的优化，它和主键索引一样，索引结构中最后记录的是物理地址，可以直接通过物理地址定位到完整的tuple，因此好处就是不用回表。但是这也有一定的问题：一个表通常有不止一个辅助索引，如果这些辅助索引都采用Physical Pointers策略，指向某个物理地址，那么在加入了新版本之后，不止是要修改/维护主键索引里的指针，还要修改所有的辅助索引里的指针。



与之相对的是，在Logical Pointers策略中，辅助索引最终只会指向相应的主键，然后再走主键索引，这样的话，在插入新版本之后只需要更新与维护主键索引里的指针，维护的开销大大降低



因此其实还有一个在这两个流派之间妥协的方案，维护一个tuple id->物理地址的中间表（跟TLB有一点像，都属于是拿空间换时间），版本更新的时候只需更新中间表就可以了



Ref/参考自：

https://www.bilibili.com/video/BV1Ja411z7mc/?spm_id_from=333.788

https://www.youtube.com/watch?v=QXWf1sgwvSs&list=PLSE8ODhjZXjZaHA6QcxDfJ0SIWBzQFKEG&index=18