Multi-version concurrency control

Definition

DBMS maintains multiple physical versions of a single **logical** object in the db.

- A txn writes to an object, then the DBMS creates a new version of that object.
- A txn reads an object, it reads the newest version that existed when the txn started.

Compared to 2PL, MVCC has the following properties:

Writers do not block readers.

Readers do not block writers.

Read-only txns can read a consistent **snapshot** without acquiring locks or txn ids.

As long as you do not do garbarge collection, you can even visit historic version of data, aka `time -travel` queries.

SNAPSHOT ISOLATION

When a transaction starts, it will see a **consistent snapshot** of the db since the txn started.

- No torn writes from active txns.(Eg: txn A aims to update 10 tuples, when finishing updating 5 tuples, txn B starts, but it will not see the five updated tuples.)
- If two txns update the same object, the first writer wins.(Based on the time they write)

中文:

- 每一个数据都多个版本,读写能够并发进行
- 每个事务相当于看到数据的一个快照
- 写写不能并发,写写操作时需要加上行锁
- 谁加行锁,谁可以顺利执行(采用了"first win"的原则),后面的写事务要么abort,要么等待前面的事务执行完成后再执行(以Oracle 和 SQL Server 、MySQL等为代表)。

Attention! Snapshot isolation is not serializable. 'Write Skew Analomy'

Write Skew Definition:

from wiki:

In a write skew anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other. Were the system serializable, such an anomaly would be impossible, as either T1 or T2 would have to occur "first", and be visible to the other. In contrast, snapshot isolation permits write skew anomalies.

简单理解为两个并发进行的事务，由于更新的时候无法看到对方的更新，可能会导致产生违反一致性的更新结果。

> 数据库约束：A1+A2>0
>
> A1，A2 实际值都为100 事务T1：If（read（A1）+read（A2）>= 200）{ Set A1 = A1 – 200 } 事务T2：If（read（A1）+read（A2）>= 200）{ Set A2 = A2 – 200 }
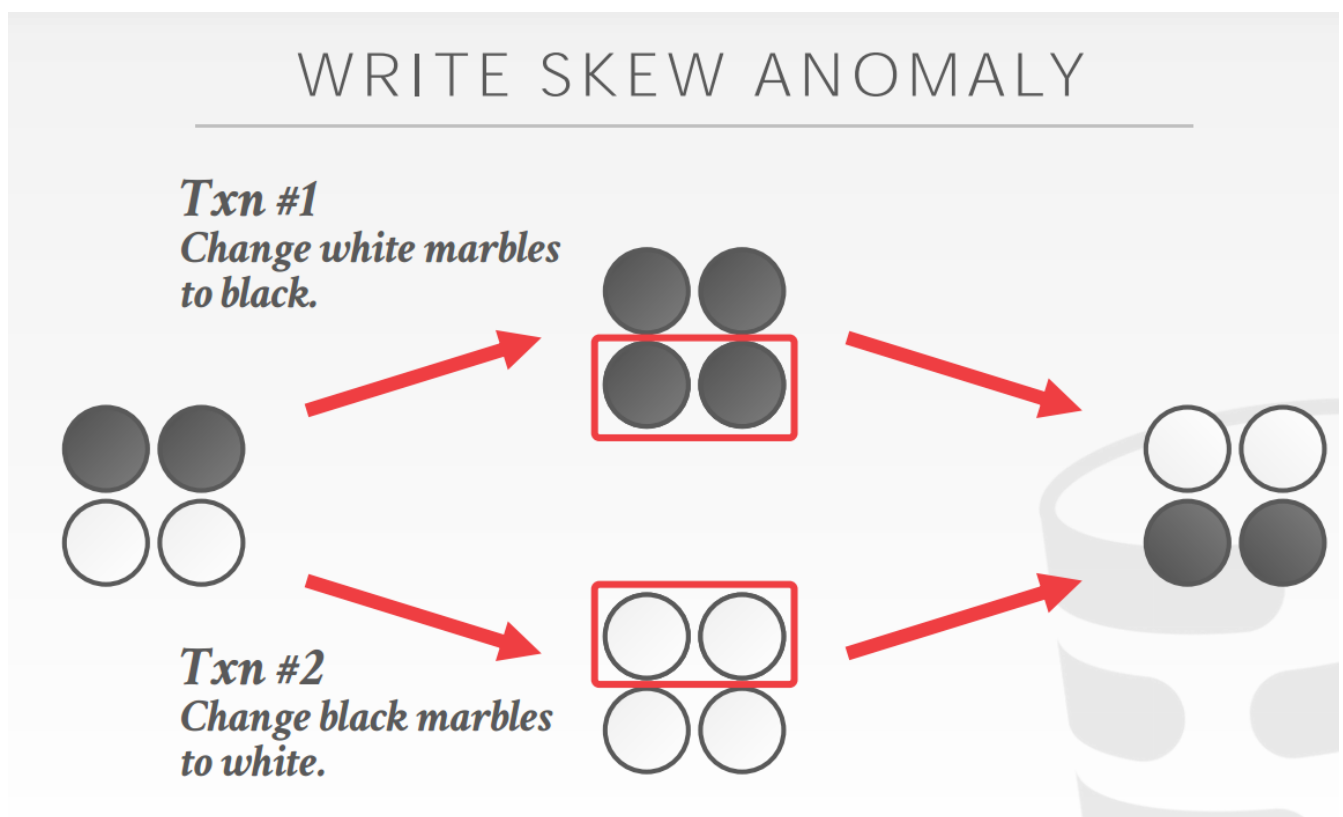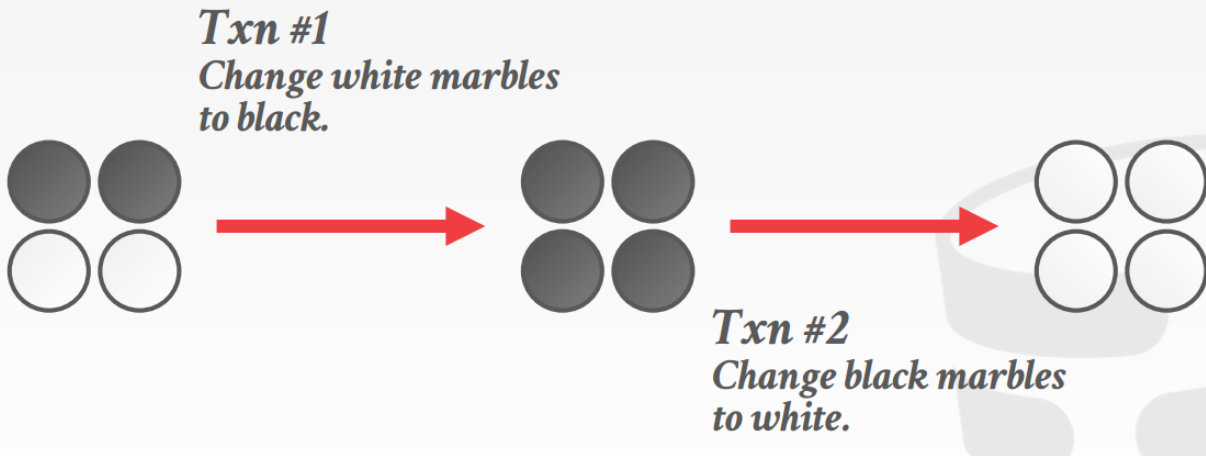
假设现在有2个事务:

Txn1:把所有白球变成黑球

Txn2:把所有黑球变成白球

两个事务开始的时候，分别看到的都是初始的两黑两白consistent snapshot的状态

而提交之后又是两黑两白，这样的情况我们不能说是可串行化的。

下面这种情况我们才可以说是可串行化。两个事务严格按照顺序进行，不能并发进行。

WRITE SKEW ANOMALY

Txn #1
Change white marbles to black.

Txn #2
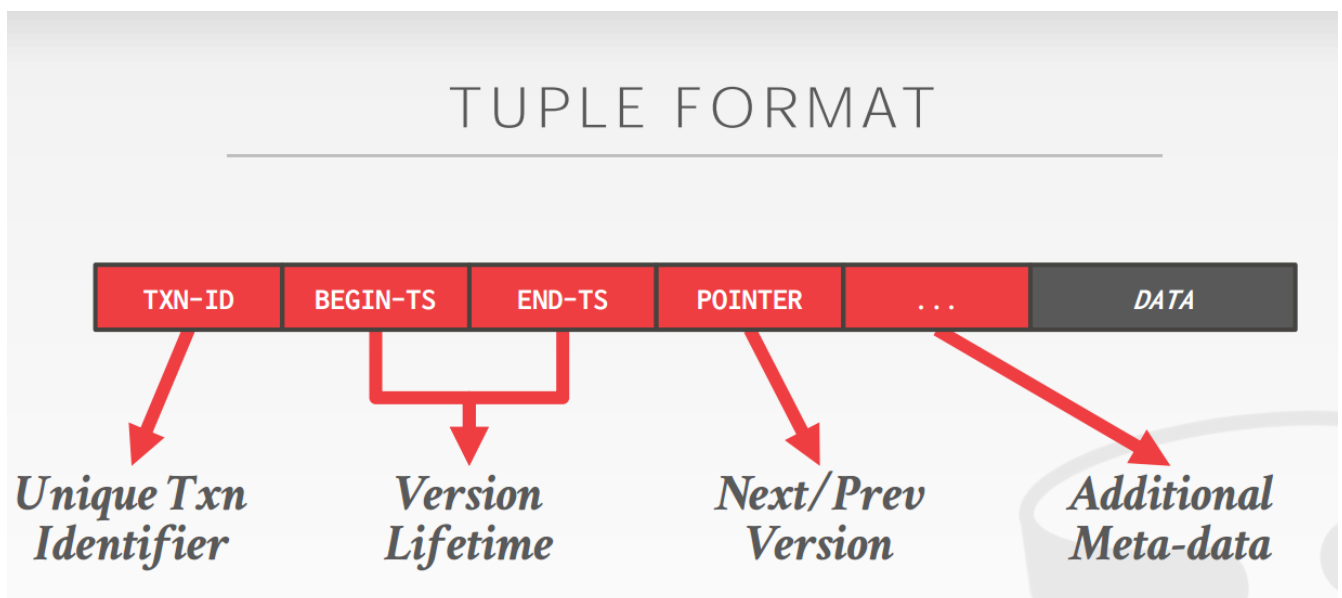**Change black marbles to white.**

下面是《An Empirical Evaluation of In-Memory MVCC》笔记,实际上是在CMU 15721第三课的课堂笔记

MVCC DESIGN DECISIONS

- Concurrency Control Protocol

In in-memory DBMS, it will store the metadata **within the tuple's header**. No need to read seperate memory locations to retrieve info about the tuple.

The metadata have the following 64-bits fields:

TUPLE FORMAT

| TXN-ID | BEGIN-TS | END-TS | POINTER | ... | DATA |

Unique Txn Identifier

Version Lifetime

Next/Prev Version

Additional Meta-data

Txn-Id: Unique Transaction Identifier. This is usually a timestamp. **充当EXCLUSIVE写锁**

Begin-TS / End-TS: A start/end range that specifies that version's lifetime. The DBMS uses this to determine for a transaction whether this particular physical version of the tuple exists in its consistent snapshot

Pointer: The memory address for the next/previous version in the tuple's version chain. The version chain is a singly linked-list of the physical versions for a logical object

Timestamp Ordering(MV-TO)

The DBMS adds an additional Read-TS field in the tuple header to keep track of the timestamp of the last transaction that read it. **No need to maintain a global data structure for txn to look up its previous transaction that it reads**

For reads, a transaction is allowed to read version if the lock is unset and its transaction id (Tid) is between Begin-TS and End-TS. Latches are not required for read operations.

For writes, a transaction creates a new version if no other transaction holds lock and Tid is greater than Read-TS. The write operation performs **CAS** on the Txn-Id field that provides a latch on this data tuple. After creating a new version, it updates the End-TS field with transaction timestamp.

eg:

时间戳为10的事务1，读A写B

对于读操作：检查TXN-ID字段是否被上锁以及时间戳是否在BEGIN-TS与END-TS之间。如果TXN-ID非零就spin

TIMESTAMP ORDERING (MVTO)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

|	TXN-ID	READ-TS	BEGIN-TS	END-TS
A₁	0	1	1	∞
B₁	0	0	1	∞

Use *read-ts* field in the header to keep track of the timestamp of the last txn that read it.

Txn can read version if the latch is unset and its T_{id} is between *begin-ts* and *end-ts*.

对于写操作：检查TXN-ID是否上锁，没有的话CAS"锁住"TXN-ID，同时创建一条新版本记录，其BEGIN-TS为当前事务时间戳。最后旧纪录的END-TS赋为当前事务时间戳。最后解锁。

TIMESTAMP ORDERING (MVTO)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A₁	0	10	1	∞
B₁ 🔒	10	0	1	10
B₂ 🔒	10	0	10	∞

Use *read-ts* field in the header to keep track of the timestamp of the last txn that read it.

Txn can read version if the latch is unset and its T_{id} is between *begin-ts* and *end-ts*.

Txn creates a new version if no other txn holds latch and T_{id} is greater than *read-ts*.

解锁之后，把TXN-ID复原为0
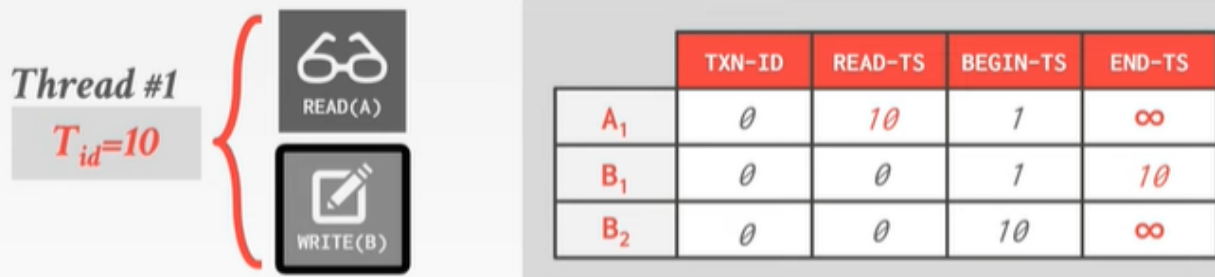
MVOCC(多版本乐观并发控制)

1.**read phase**

Search for valid version (Write lock is not acquired and Txn id between begin-ts and end-ts)

2.**validation phase**

DBMS assigns the txn another timestamp(T commit), and check whether the read set was updated by another committed txn.

3.**write phase**

install new version, set the begin-ts to T commit and end-ts to inf.

乐观并发控制对于冲突少的情况比较好，冲突一多了，读取过期版本的事务都会在validation phase被丢弃。

MV-2PL

The DBMS adds an additional **Read-Cnt** field to each tuple's header that acts as a **shared lock**. This is a 64-bit counter that tracks the number of transactions currently holding this lock. For reads, a transaction is allowed to hold the share lock if Txn-Id is zero. It then performs a CAS on Read-Cnt to increment the counter by one.

For writes, a transaction is allowed to hold the **exclusive lock if both Txn-Id and Read-Cnt are zero**. The DBMS uses Txn-Id and Read-Cnt together as exclusive lock. On commit, Read-Cnt and Txn-Id are reset to zero.

This design is good for deadlock prevention, but needs extra global data structures for deadlock detection.
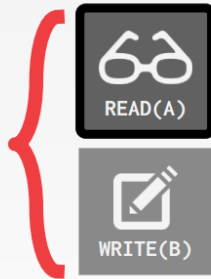
读操作：

检查txn-id和read-cnt是否为0，是的话CAS更新READ-CNT

TWO-PHASE LOCKING (MV2PL)

Thread #1
$T_{id}=10$

READ(A)

WRITE(B)

🔒

	TXN-ID	READ-CNT	BEGIN-TS	END-TS
A_1	0	1	1	∞
B_1	0	0	1	∞

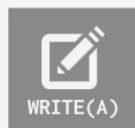Txns use the tuple's *read-cnt* field as SHARED lock. Use *txn-id* and *read-cnt* together as EXCLUSIVE lock.

If *txn-id* is zero, then the txn acquires the SHARED lock by incrementing the *read-cnt* field.

Wrap-around issue

假设TXN-ID为2^31 -1，那么会创建A2这么一个新版本记录

Thread #1
$T_{id}=2^{31}-1$

WRITE(A) ➔

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	$2^{31}-1$	–	99999	$2^{31}-1$
A_2	$2^{31}-1$	–	$2^{31}-1$	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

这时候下一个事务ID回到1,生成A3，并且把A2的END-TS设为1

Thread #1
$T_{id}=2^{31}-1$

Thread #2
$T_{id}=1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	–	99999	$2^{31}-1$
A_2	0	–	$2^{31}-1$	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

这就会导致下图的问题：

A3记录物理上版本是更新的，但是根据BEGIN-TS和END-TS，其版本比A2还要旧，需要另外的机制解决这个问题。

Thread #1
$T_{id}=2^{31}-1$

Thread #2
$T_{id}=1$

WRITE(A)

	TXN-ID	READ-TS	BEGIN-TS	END-TS
A_1	0	–	99999	$2^{31}-1$
A_2	1	–	$2^{31}-1$	1
A_3	1	–	1	∞

If the DBMS reaches the max value for its timestamps, it will have to wrap around and restart at one. This will make all previous versions be in the "future" from new transactions.

Postgres做法：增加一个frozen字段，任何记录版本都会比frozen的记录要更新。

Set a flag in each tuple header that says that it is "frozen" in the past. Any new txn id will always be newer than a frozen version.

Runs the vacuum before the system gets close to this upper limit.

Otherwise it must stop accepting new commands when the system gets close to the max txn id.

- Version Storage

The DBMS uses the tuple's pointer field to create a **latch-free version chain per logical tuple**.

1.This allows the DBMS to find the version that is visible to a particular transaction at runtime

2.Indexes always point to "head" of the chain.

Append-Only Storage

APPEND-ONLY STORAGE

Main Table

All the physical versions of a logical tuple are stored in the same table space. The versions are mixed together.

On every update, append a new version of the tuple into an empty space in the table.

	VALUE	POINTER
A_0	$111	●
A_1	$222	●
B_1	$10	Ø
A_2	$333	Ø

1.Oldest-To-Newest (O2N)

-> Append every new version to end of the chain.

-> Must traverse chain on look-ups.(Must begin at A0 if you want to visit A2)

2.Newest-To-Oldest(N2O)

-> Must update index pointers for every new version.

-> Dont't have to traverse chain on look ups.

The benefit of method 2: time complexity is slower, but you need to update the indexes.

Time-Travel Storage

Instead of storing all the tuples versions in a single table, the DBMS splits a single logical table into two sub-tables:
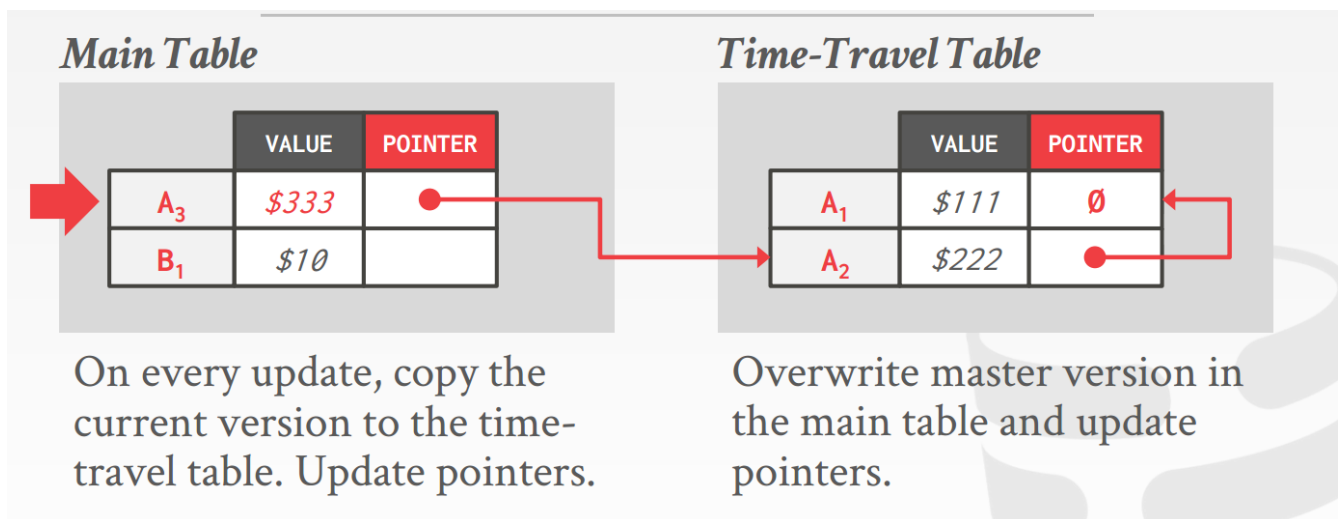
(1) main table

(2) time-travel table.

The main table keeps the latest version of tuples. When a transaction updates a tuple, the DBMS copies current version from the main table to the time-travel table.

It then overwrites the master version in main table and updates its pointer to the recently copied entry in the time-travel table

Garbage collection is fast with this approach since the DBMS can just drop entries from the time-travel entry without scanning the main table. **Sequential scans** are faster easy since the DBMS can just scan the main table **without checking version information**.

Main Table

	VALUE	POINTER
A₃	$333	●
B₁	$10	

On every update, copy the current version to the time-travel table. Update pointers.

Time-Travel Table

	VALUE	POINTER
A₁	$111	Ø
A₂	$222	●

Overwrite master version in the main table and update pointers.

HANA,SQL SERVER都是采用这种做法

这样做还有的好处是可以对Main Table和Time-Travel Table用不同的存储方式来存储。

前者适用于经常发生更新操作的行式存储
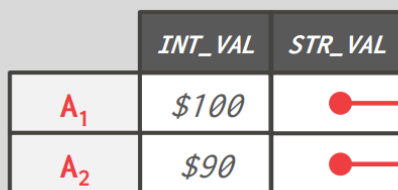
后者适用于经常发生读和遍历操作的列式存储

Delta Storage

这么做的好处是假如记录有非常多column，假设1000+column，你不需要像上一种方法一样把1000多列的数据都copy到memory另一个地方。你只需要记录变更的列的记录即可。

Non-inline Attributes

以下问题常见于Append-Only Storage和Time-Travel Storage.

Variable-length data can be stored in separate space and be referenced by a pointer in the data tuple. Direct duplication of these data is wasteful, so the DBMS can reuse pointers to variable-length pool for values that do not change between versions. One option is to use reference counters to know when it is safe to free from memory. As a result the DBMS would not be able to relocate memory easily, thus no existing system implements this optimization.

就例如A2相比于A1来说，每次更新都没有改动STR_VAL这个字段。那么每次都需要复制STR_VAL所指的字符串示例，如果都指向一个的话，GC之后可能就会指向一个空对象。

解决方法是利用垃圾回收的引用计数算法:

- Garbage Collection


Tuple-level

With this approach, transactions do not maintain additional meta-data about old versions. Thus, the DBMS has to scan tables to find old versions.

• Background Vacuuming:

后台线程进行GC

Separate threads **periodically scan the table and look for reclaimable versions**. Works with any version storage technique. To avoid repeatedly scanning through unmodified data, the DBMS can use a **dirty block bitmap** to keep track of what blocks of data have been modified since the last scan.

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
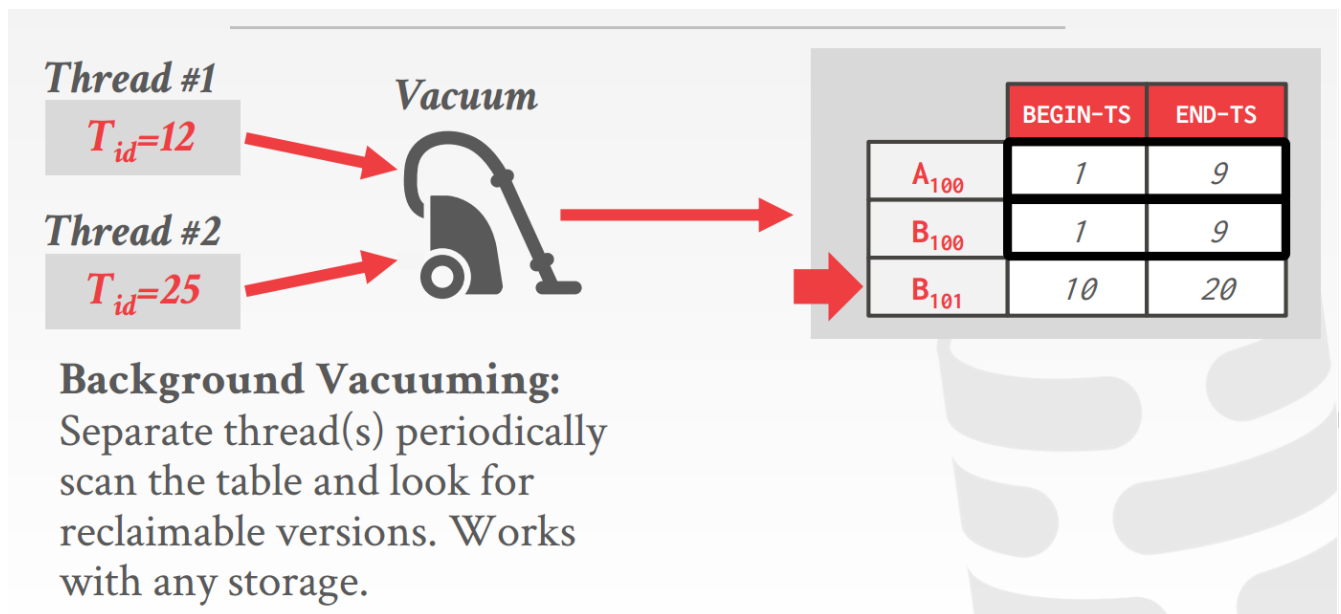
把间隔不包括扫描线程时间戳的记录删除，但是这样性能消耗其实比较大。

Optimization：

用一个位图来标记已经修改过的纪录,那么线程扫描的时候skip掉那些没有被修改过的记录。

TUPLE-LEVEL GC

Background Vacuuming:
Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.
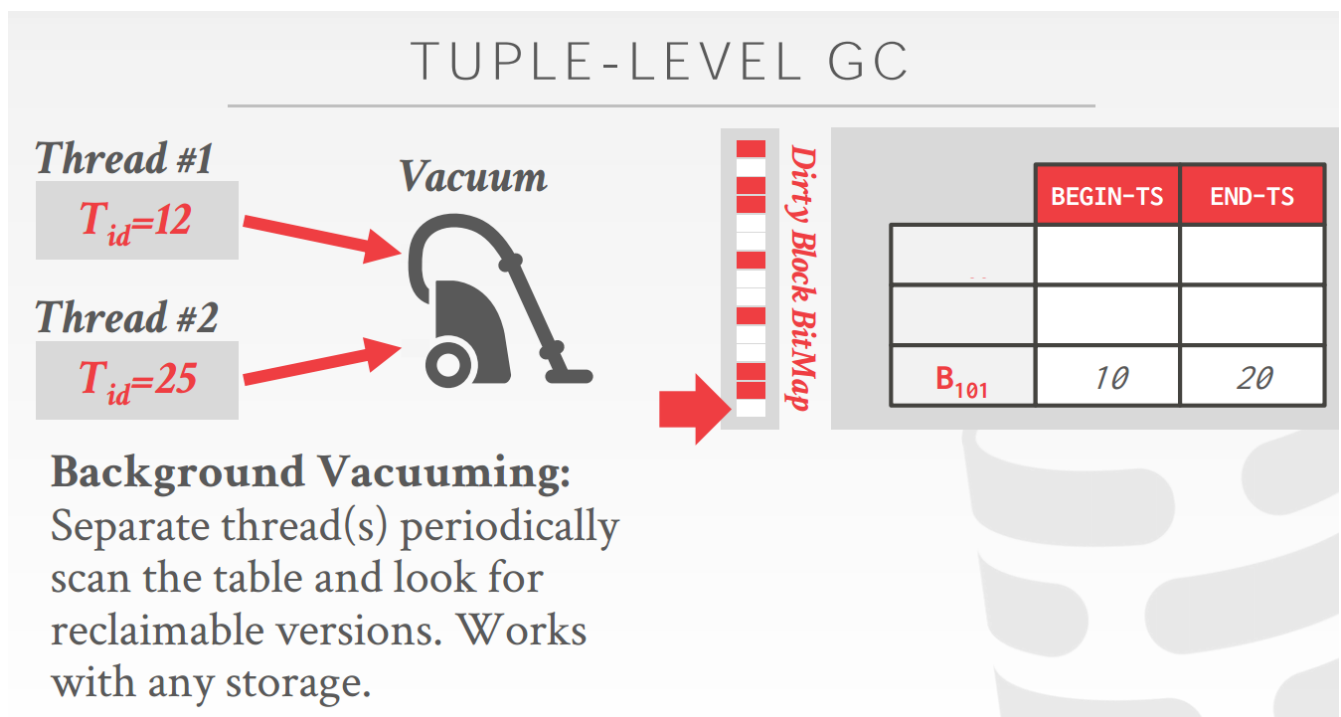
• Cooperative Cleaning:

使用事务处理本身的线程，协同处理GC

Worker threads identify reclaimable versions as they traverse version change. Only works with O2N version chains. A problem with this is that if there are never any queries that access tuples with reclaimable versions, then these versions will not get cleaned up (i.e., "dusty corners"). Thus, the DBMS still has to periodically scan the table to find old versions.

TUPLE-LEVEL GC

Thread #1
$T_{id}=12$

Thread #2
$T_{id}=25$

GET(A)

INDEX

$A_0 \rightarrow A_1 \rightarrow A_2 \rightarrow A_3$

$B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_3$

Background Vacuuming: Separate thread(s) periodically scan the table and look for reclaimable versions. Works with any storage.

Cooperative Cleaning: Worker threads identify reclaimable versions as they traverse version chain. Only works with O2N.

TUPLE-LEVEL GC

Thread #1
$T_{id}=12$

Thread #2
$T_{id}=25$

GET(A)

INDEX

$[X] \rightarrow X \rightarrow A_2 \rightarrow A_3$

$B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_3$

TUPLE-LEVEL GC

Thread #1
$T_{id}=12$

Thread #2
$T_{id}=25$

GET(A)

INDEX

$A_2 \rightarrow A_3$

$B_0 \rightarrow B_1 \rightarrow B_2 \rightarrow B_3$

Pros and cons

1.You don't have to maintain a background thread

2.Queries might get slower since you need to traverse a rather long version chain list.

Transaction-level

With this approach, **transactions keep track of their old version** so the DBMS does not have to scan tuples to determine visibility. The DBMS determines when all versions created by a finishing transaction are no longer visible.

- Index Management

PKey indexes always point to version chain head.
→ How often the DBMS must update the pkey index depends on whether the system creates new versions when a tuple is updated.
→ If a txn updates a tuple's pkey attribute(s), then this is treated as a **DELETE** followed by an **INSERT**.
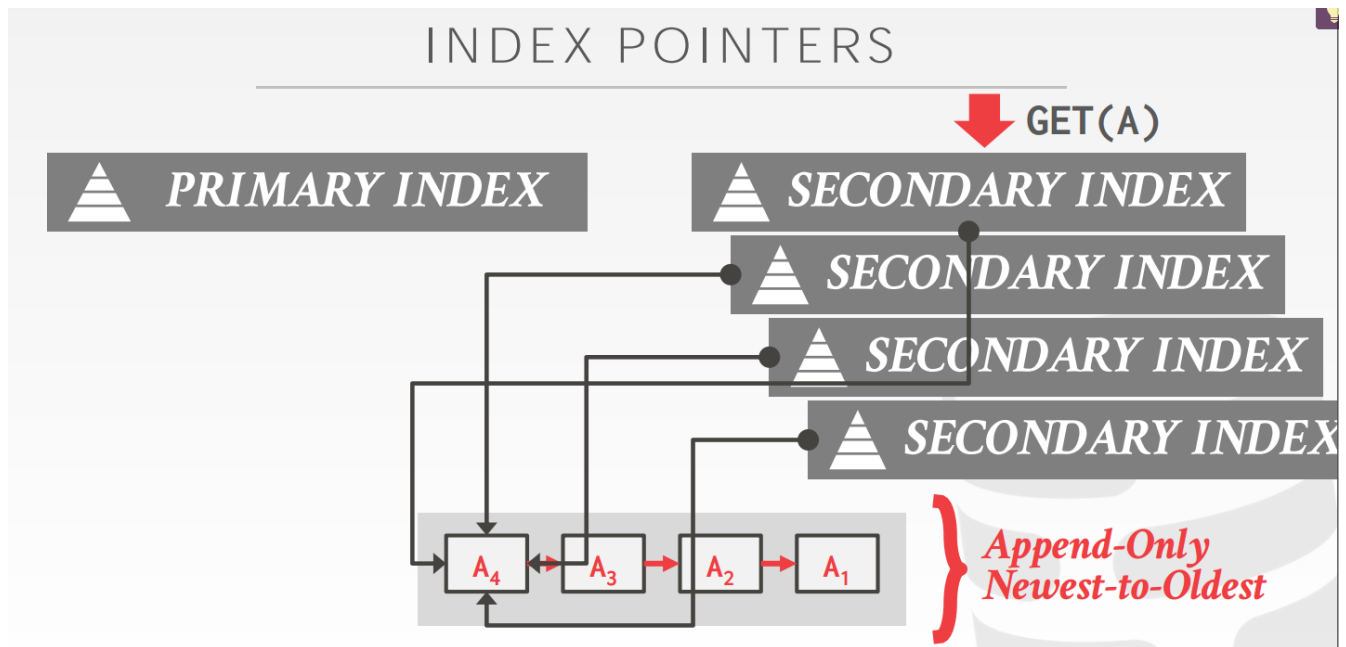
- Secondary Indexes

Managing secondary indexes is more complicated than primary key indexes. There are two approaches to storing values that represent the location of a tuple's version chain.

(a) Logical Pointers　　　(b) Physical Pointers
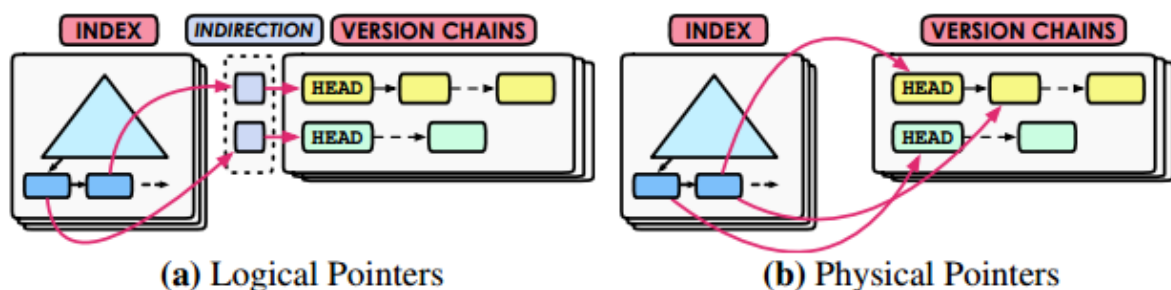
Postgres:

Approach #1: Physical Pointer • Use physical address to the version chain head. • If a databases has many secondary indexes, then updates can become expensive because the DBMS has update all the indexes to the new location (e.g., each update to a N2O version chain requires the DBMS to update every index with the memory address of the new version chain head).

直接指向存储tuple的物理地址，只适用于Append-Only的Version Storage(当修改一个Tuple是，会将新版本直接插入二级索引)，适合于读多的场景
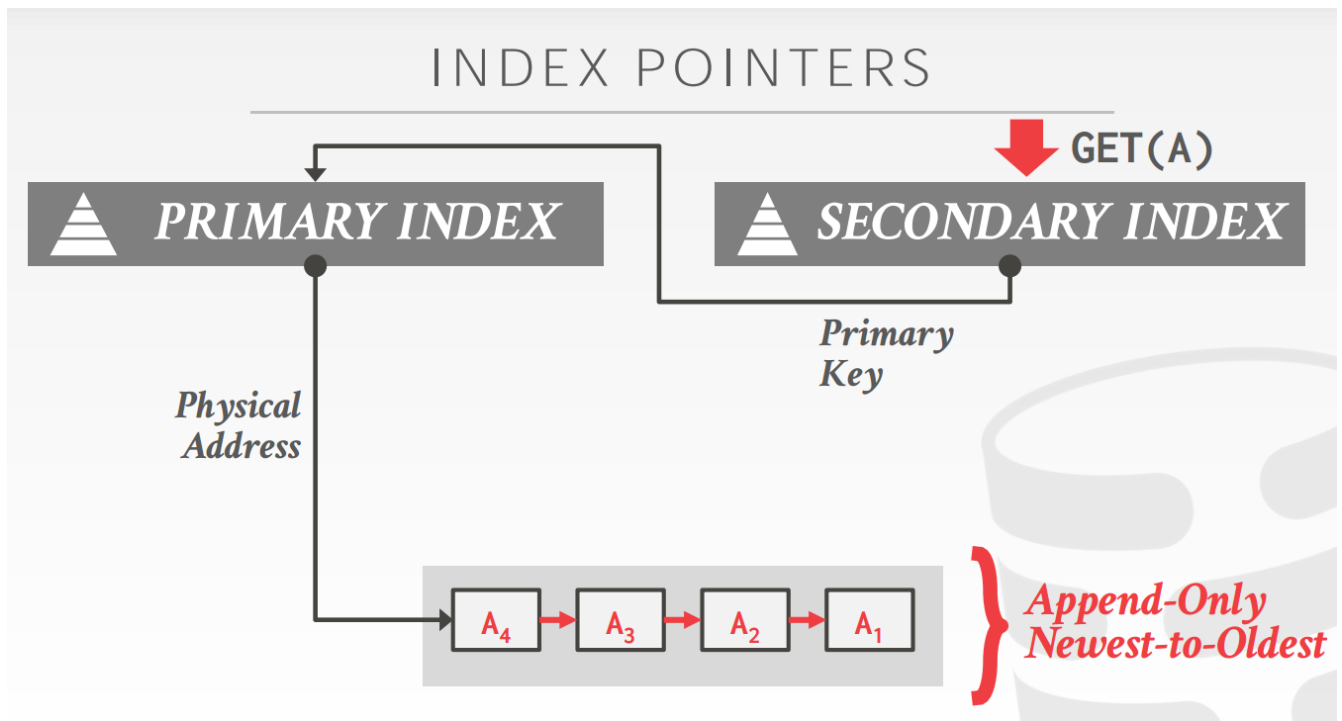

MySQL:

Approach #2: Logical Pointer • Primary Key: Store the tuple's primary key as the value for a secondary index, which will then redirect to the physical address. This approach has high store overhead if the size of the primary key is large.

通过增加一次间接层，解决Tuple物理地址的变更导致要修改索引的指向问题。索引的value的选择可以是主键，或者是tupleid隐式主键。隐式主键的好处是大小可控。适合于写多的场景.

INDEX POINTERS

一旦Primary key的大小变大，会对数据库的存储造成一定压力，并且secondary index是拥有其完整的copy.并且DBMS还要维护保证主键有序的数据结构。可以采用以下策略：

• Tuple ID: Use a fixed identifier per tuple that does not change. This approach requires an extra indirection layer(A hashmap) to map the id to a physical address（HEAD of the tuple's version chain）. For example, this could be a hash table that maps Tuple IDs to physical addresses

ref

http://www.nosqlnotes.com/technotes/mvcc-snapshot-isolation/