

Lecture 16: Multi-Version Concurrency Control

Kb³
midterm exam
(Mar 28)

next Wed

Recap

Optimistic Concurrency Control

- The DBMS creates a private workspace for each txn.
 - ▶ Any object read is copied into workspace.
 - ▶ Modifications are applied to workspace.
- When a txn commits, the DBMS compares workspace write set to see whether it conflicts with other txns.
- If there are no conflicts, the write set is installed into the global database.



OCC Phases

- Phase 1 – Read:

- ▶ Track the read/write sets of txns and store their writes in a private workspace.

- Phase 2 – Validation:

- ▶ When a txn commits, check whether it conflicts with other txns.

- Phase 3 – Write:

- ▶ If validation succeeds, apply private changes to database. Otherwise abort and restart the txn.

Today's Agenda

- Multi-Version Concurrency Control

- Design Decisions

- ▶ Concurrency Control Protocol
- ▶ Version Storage
- ▶ Garbage Collection
- ▶ Index Management

Multi-Version Concurrency Control

Multi-Version Concurrency Control

- The DBMS maintains multiple physical versions of a single logical object in the database:
 - When a txn writes to an object, the DBMS creates a new version of that object (instead of private workspace in OCC)
 - When a txn reads an object, it reads the newest version that existed when the txn started.

snapshot isolation

MVTO
MV2PL

MVCC HISTORY

- Protocol was first proposed in 1978 MIT PhD dissertation.
- First implementations was Rdb/VMS and InterBase at DEC in early 1980s.
 - ▶ Both were by Jim Starkey, co-founder of NuoDB.
 - ▶ DEC Rdb/VMS is now "Oracle Rdb"
 - ▶ InterBase was open-sourced as Firebird.

Multi-Version Concurrency Control

MVCC

- Writers don't block readers. Readers don't block writers.
- Read-only txns can read a consistent snapshot without acquiring locks.
 - ▶ Use timestamps to determine visibility.
- Easily support time-travel queries.

blockchain

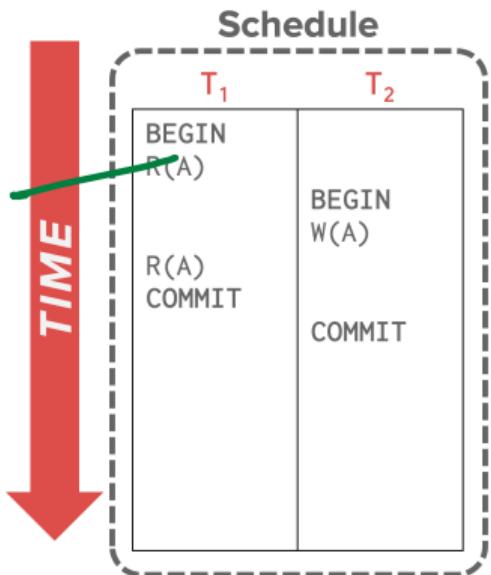
Architectural

Any

MVCC – Example 1

MV TO

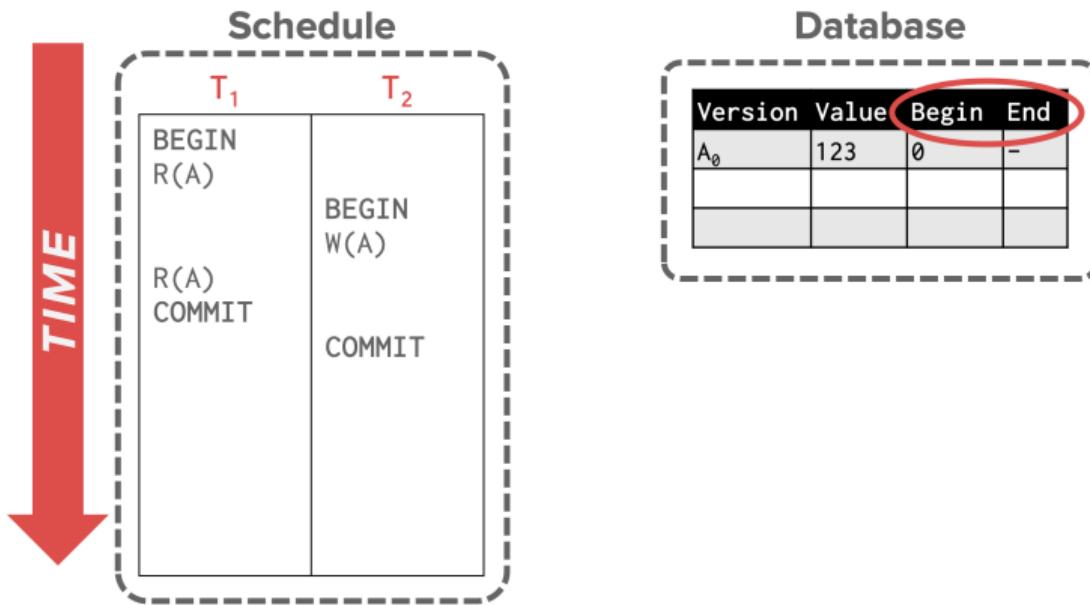
Timestamp



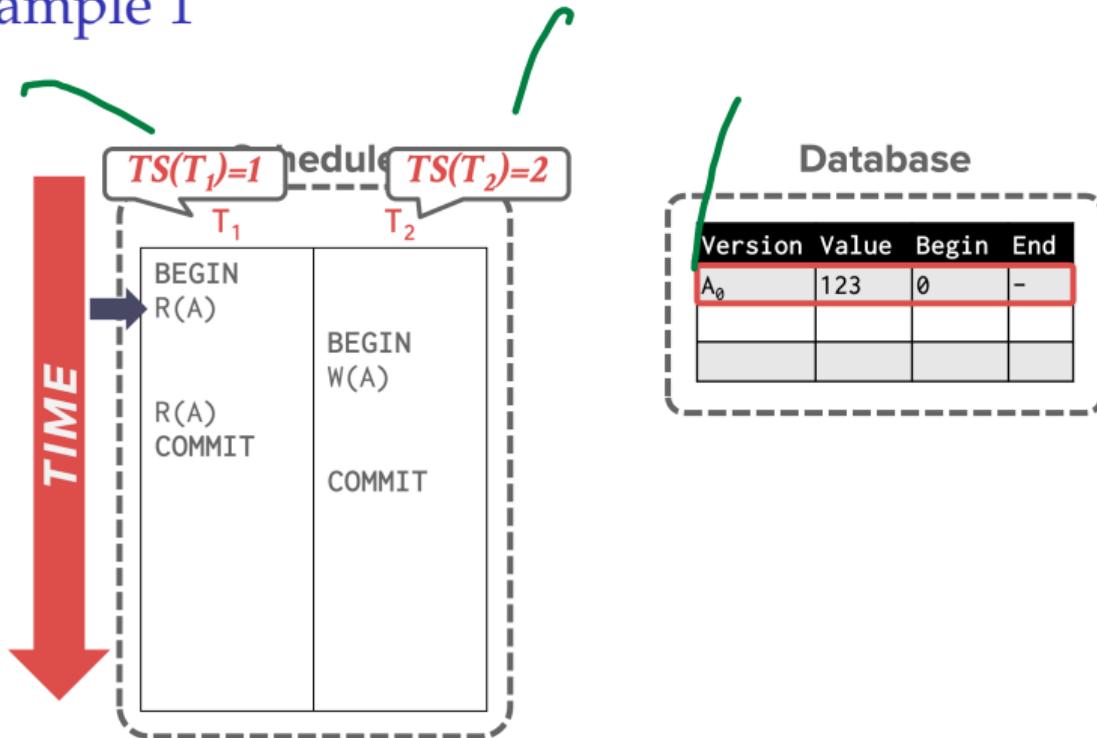
Database

Version	Value	Begin	End
A ₀	123	0	

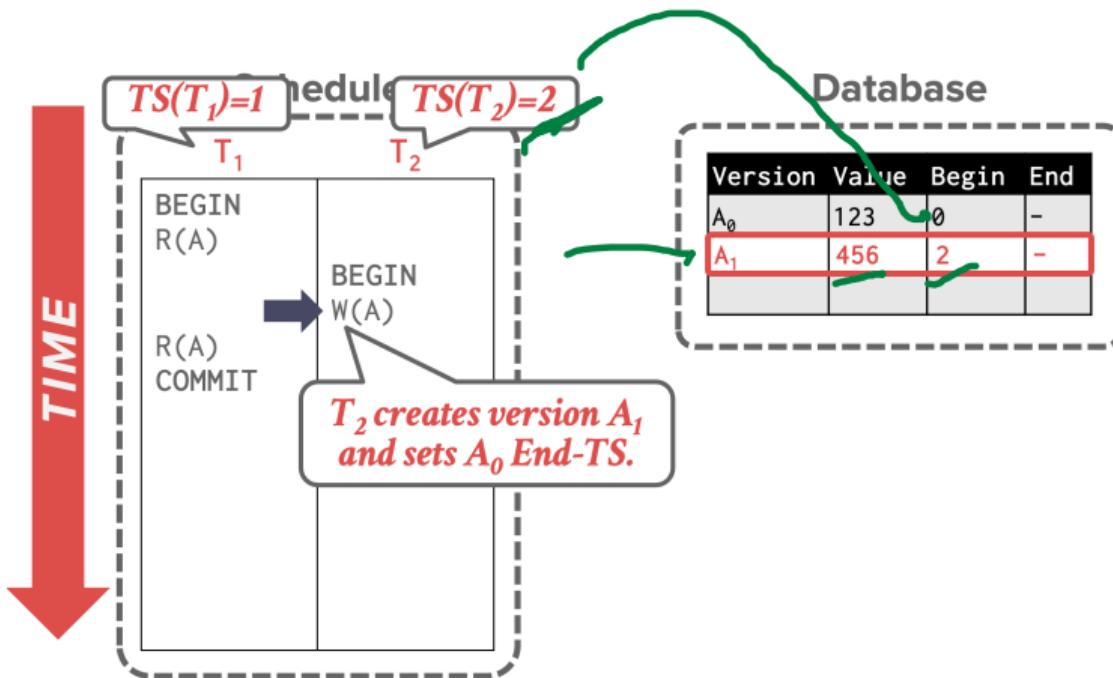
MVCC – Example 1



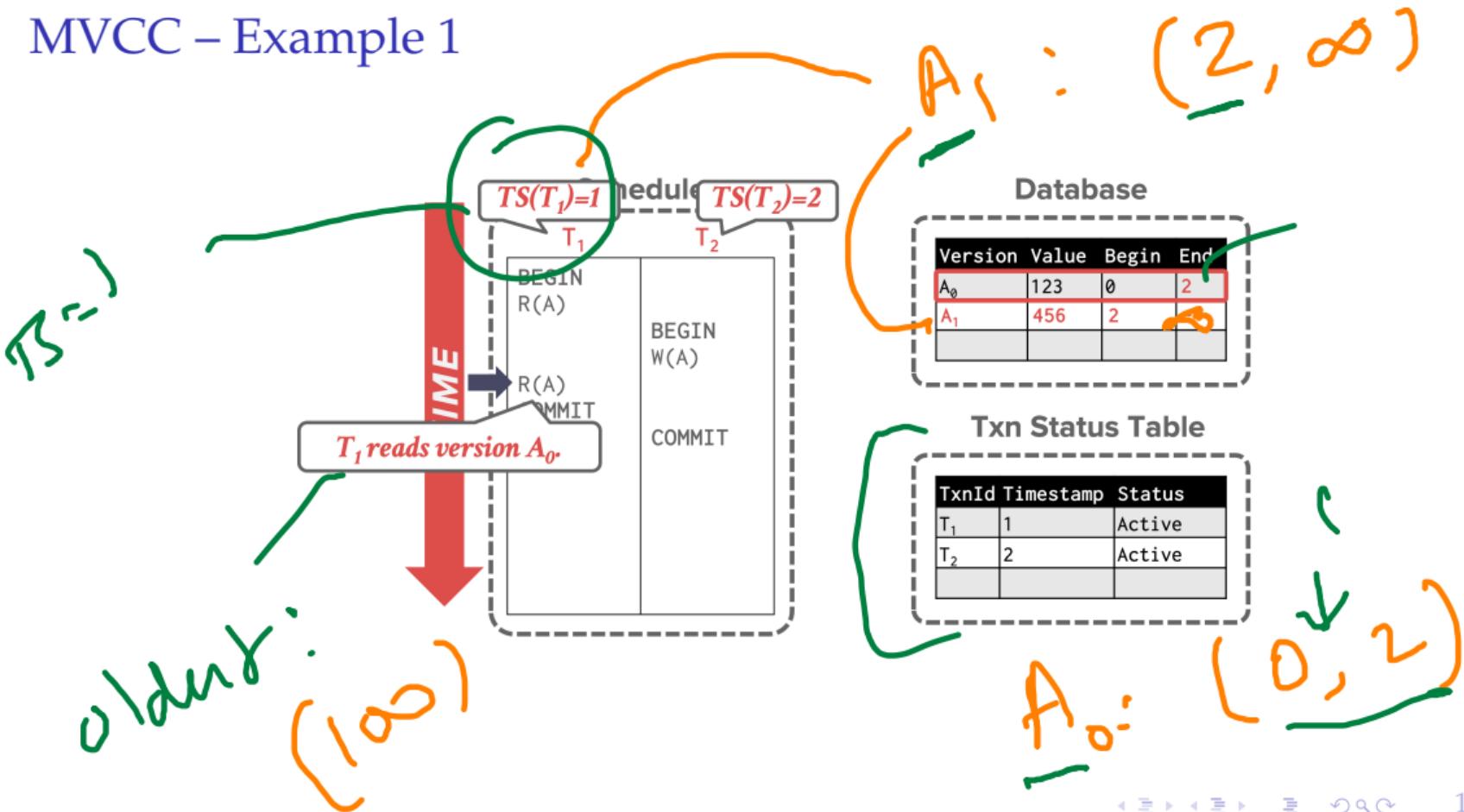
MVCC – Example 1



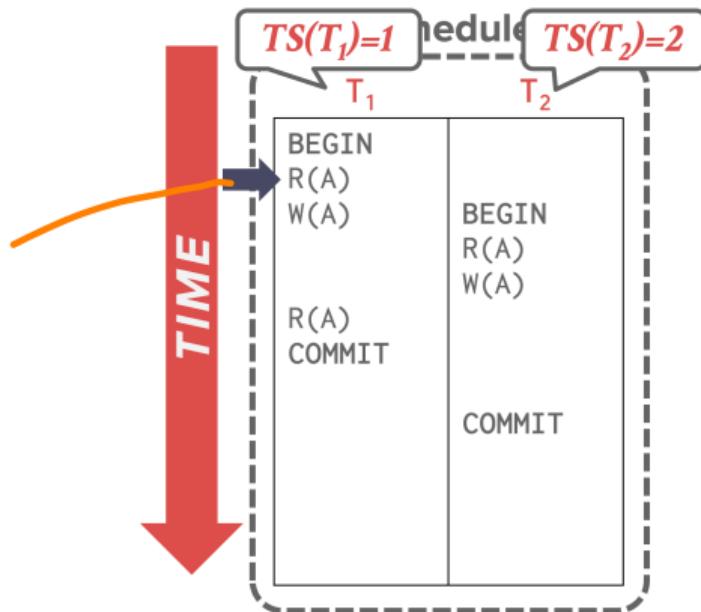
MVCC – Example 1



MVCC - Example 1



MVCC – Example 2



Globally

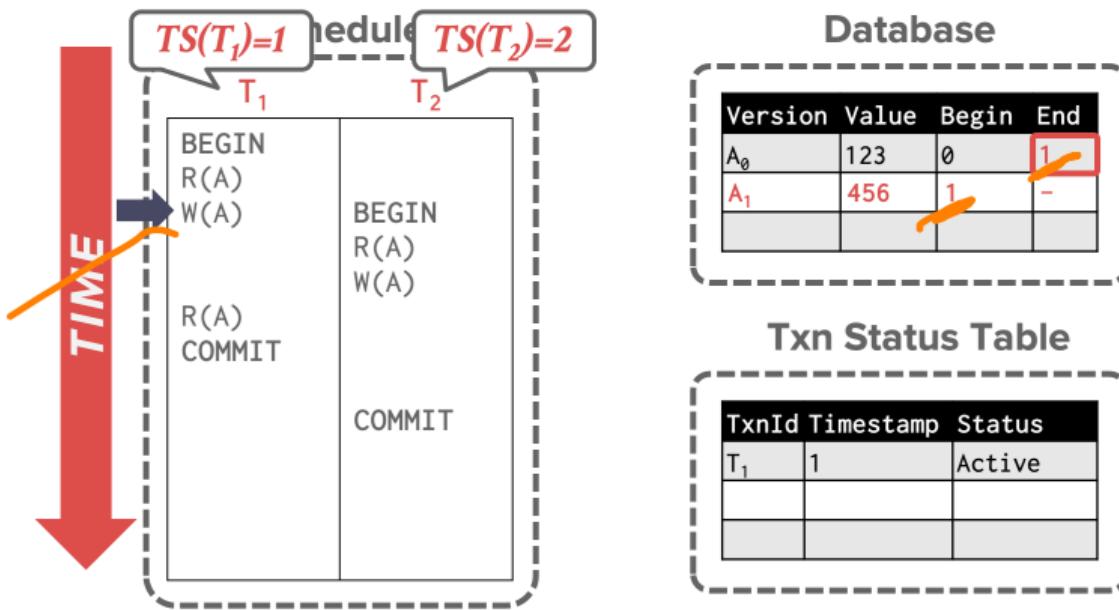
Database

Version	Value	Begin	End
A_0	123	0	

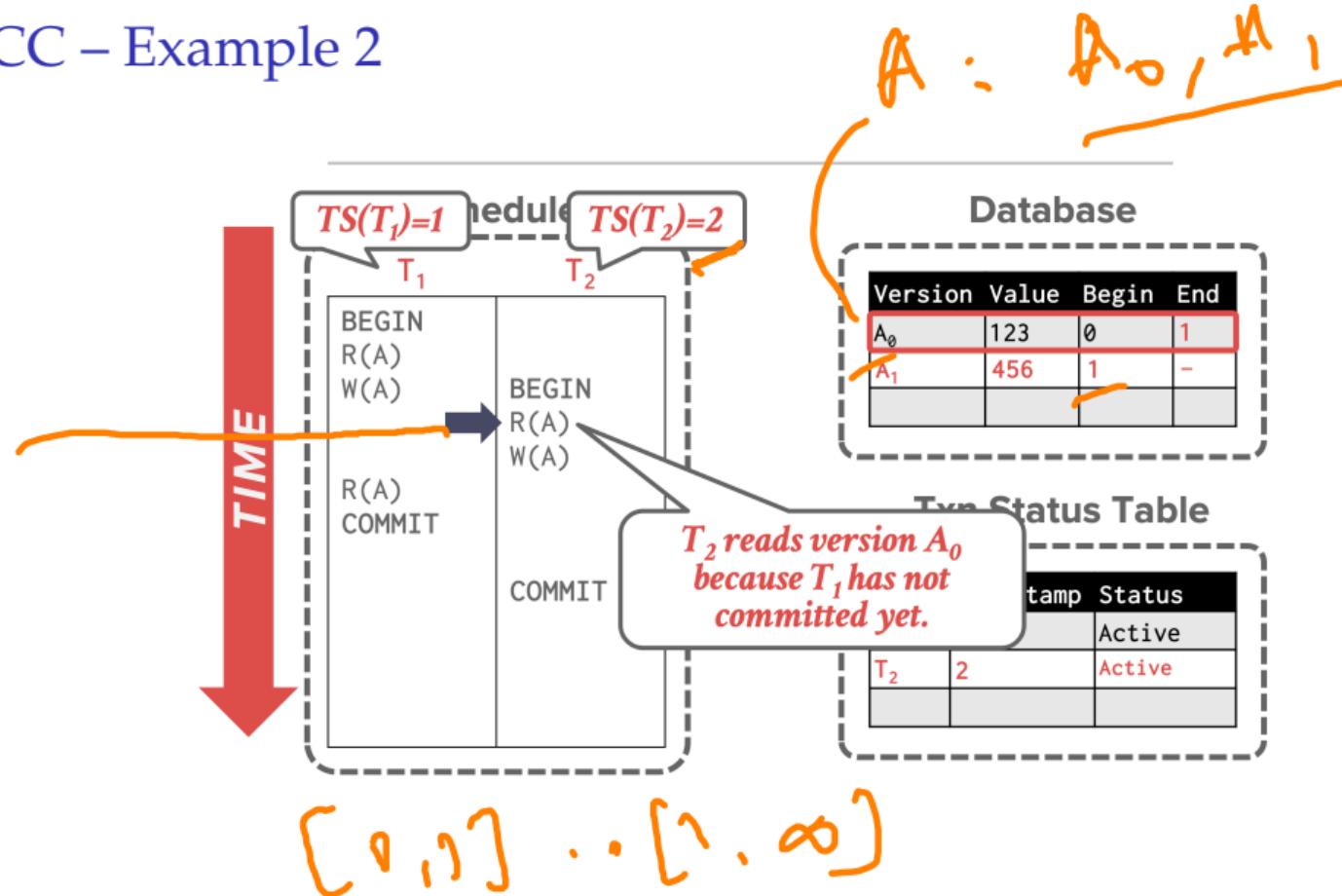
Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

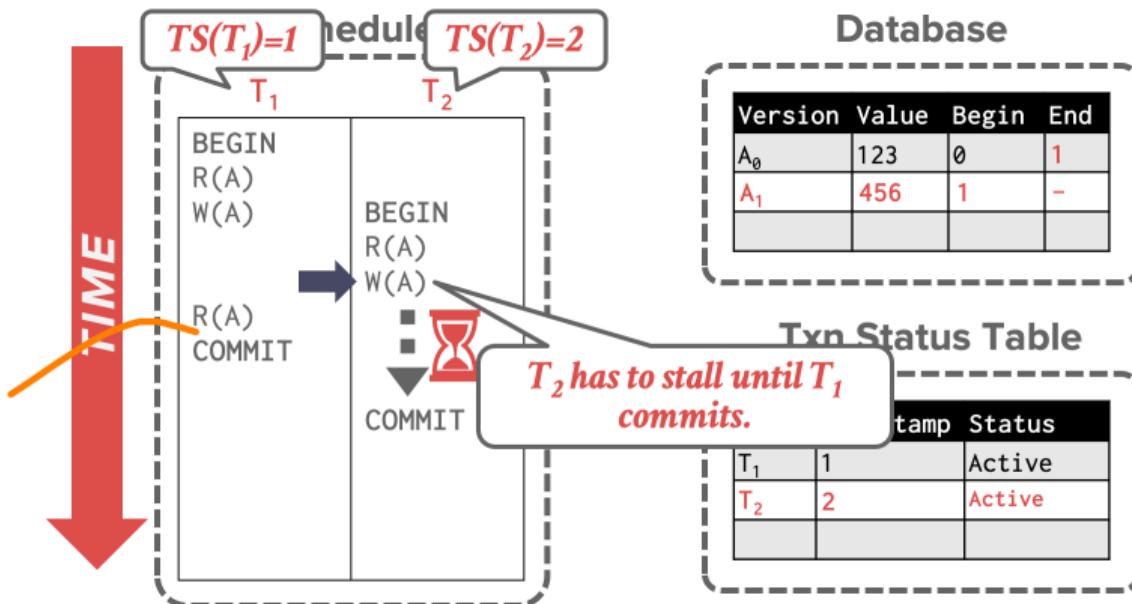
MVCC – Example 2



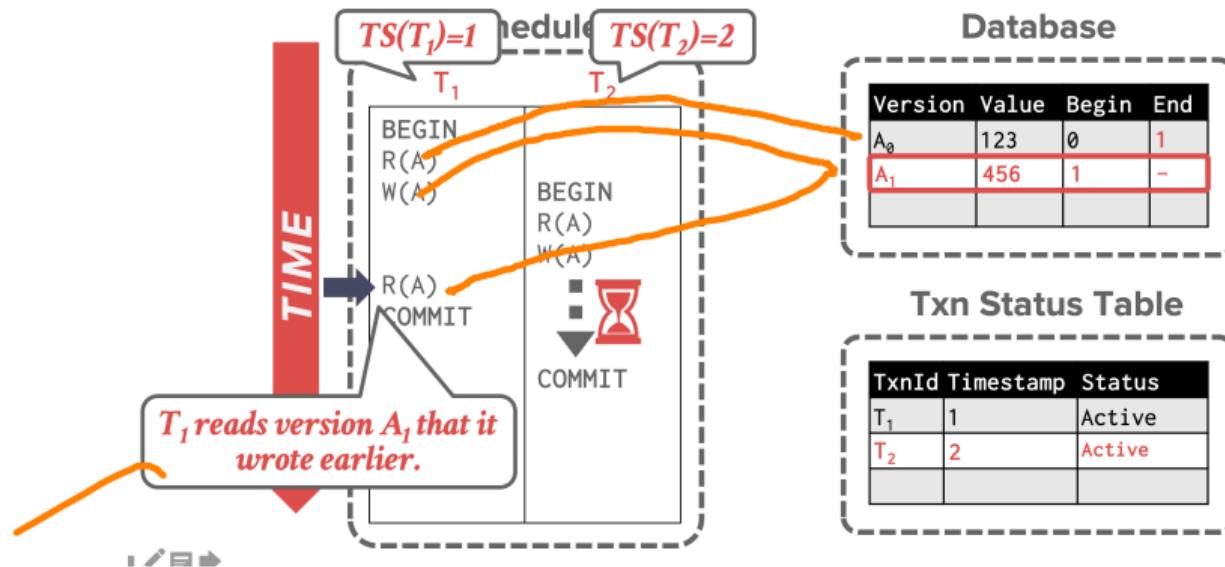
MVCC – Example 2



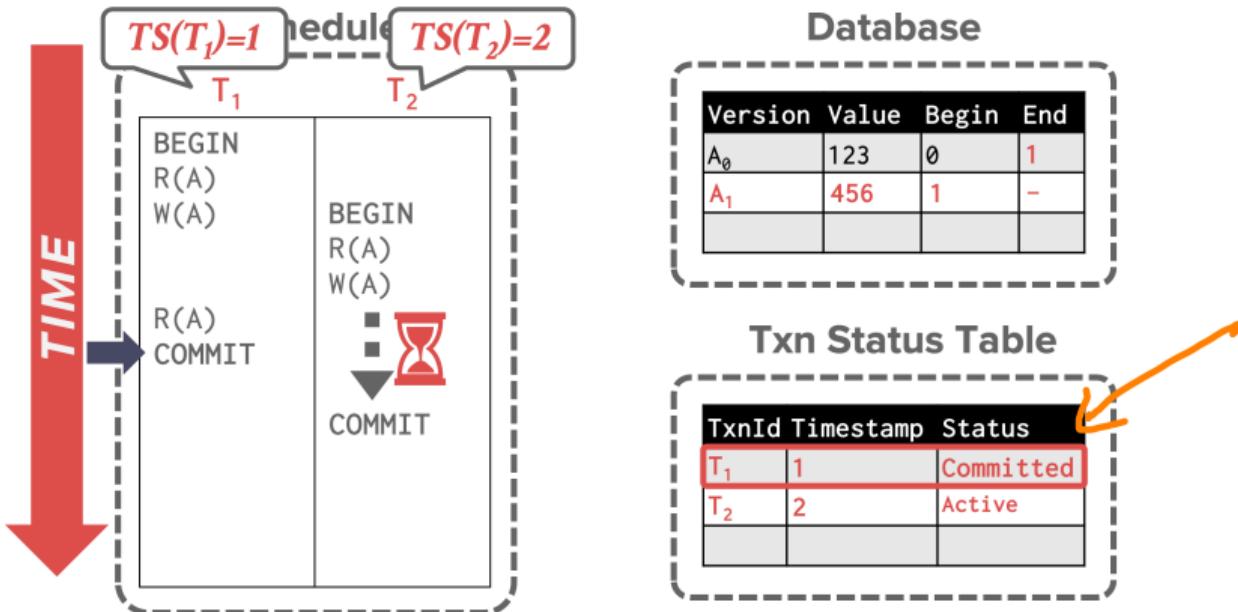
MVCC – Example 2



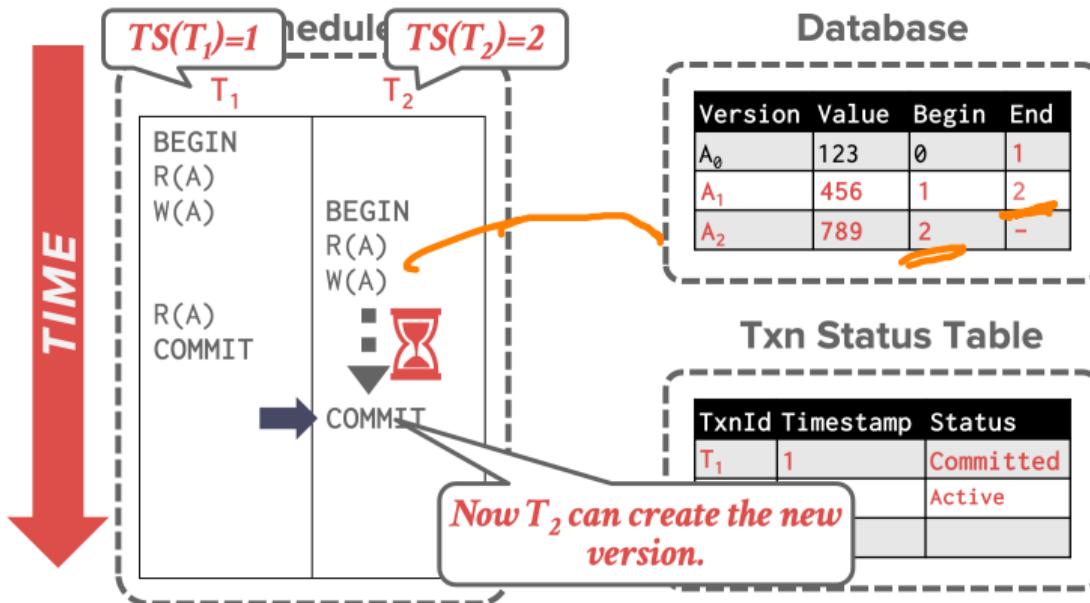
MVCC – Example 2



MVCC – Example 2

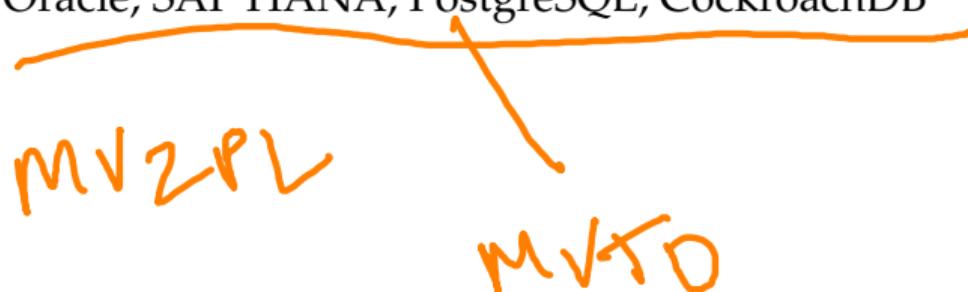


MVCC – Example 2



Multi-Version Concurrency Control

- MVCC is more than just a Concurrency Control protocol.
- It completely affects how the DBMS manages transactions and the database.
- Examples: Oracle, SAP HANA, PostgreSQL, CockroachDB



MVCC Design Decisions

- Concurrency Control Protocol
- Version Storage
- Garbage Collection
- Index Management

Key: value

physical tuple point
logical triple id

Concurrency Control Protocol

Concurrency Control Protocol

- Approach 1: Timestamp Ordering

- Assign txns timestamps that determine serial order.

MVTO

- Approach 2: Optimistic Concurrency Control

MVOCC

- Three-phase protocol from last class.

- Use private workspace for new versions.

- Approach 3: Two-Phase Locking

MV2PL

- Txns acquire appropriate lock on physical version before they can read/write a logical tuple.

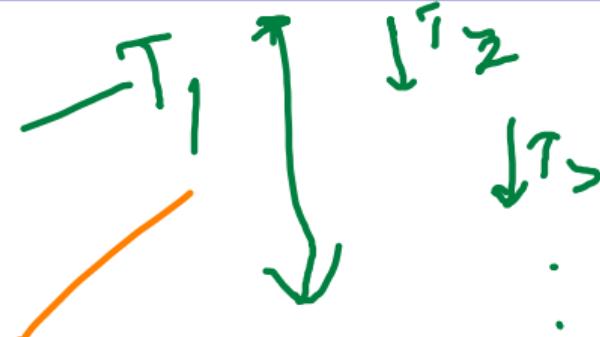
Read A

A₀
A₁
A₂

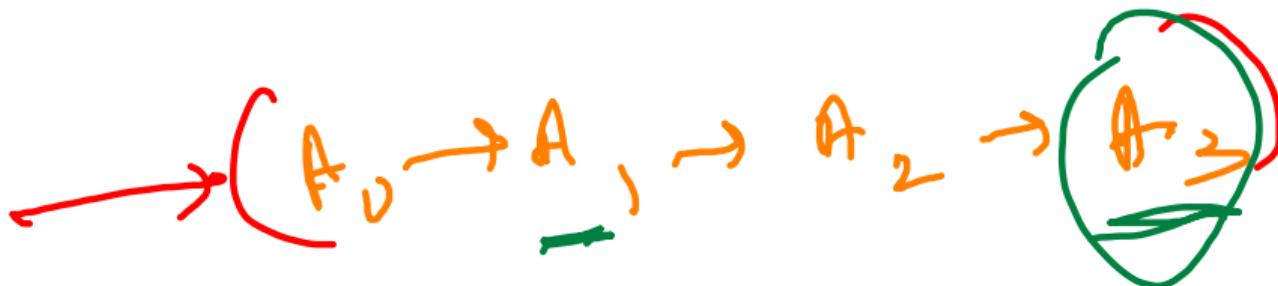
Version Storage

Version Storage

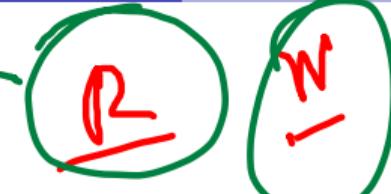
Advanced methods



- The DBMS uses the tuples' pointer field to create a version chain per logical tuple.
 - ▶ This allows the DBMS to find the version that is visible to a particular txn at runtime.
 - ▶ Indexes always point to the head of the chain.
- Different storage schemes determine where/what to store for each version.



Version Storage



storage
fashion

- Approach 1: Append-Only Storage

- New versions are appended to the same table space.

- Approach 2: Time-Travel Storage

- Old versions are copied to separate table space.

- Approach 3: Delta Storage

- The original values of the modified attributes are copied into a separate delta record space.



Append-Only Storage

- All of 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.

Main Table

A diagram illustrating an Append-Only Storage system. A large blue arrow points from the left towards a table labeled "Main Table". Above the table, a green curved line spans across the first two columns of the header. The table has three rows, each representing a different version of a tuple. The columns are labeled "VERSION", "VALUE", and "POINTER". The first row contains "A₀", "\$111", and a red circle pointing to the second row. The second row contains "A₁", "\$222", and an empty set symbol. The third row contains "B₁", "\$10", and another empty set symbol. The last two rows are empty.

VERSION	VALUE	POINTER
A ₀	\$111	•
A ₁	\$222	Ø
B ₁	\$10	Ø

Version
chain

Append-Only Storage

- All of 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.

Main Table

The diagram illustrates an Append-Only Storage Main Table. It consists of a table with three columns: VERSION, VALUE, and POINTER. The table contains four rows of data:

VERSION	VALUE	POINTER
A ₀	\$111	∅
A ₁	\$222	∅
B ₁	\$10	∅
A ₂	\$333	∅

A large blue arrow on the left points towards the table, indicating an insertion operation. A red box highlights the last row, A₂, which has just been appended. Green arrows point from the bottom of the previous rows to the first cell of the new row, illustrating how the table grows by appending new rows at the bottom. Red arrows also point from the last cell of each row to the first cell of the next row, showing the linked list structure where each row's pointer points to the next row's starting position.

Append-Only Storage

- All of 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.

Main Table

The diagram illustrates an append-only storage mechanism. A large grey arrow points from the left towards a table labeled "Main Table". The table has three columns: "VERSION", "VALUE", and "POINTER". It contains four rows, each representing a version of a tuple:

VERSION	VALUE	POINTER
A_0	\$111	○ →
A_1	\$222	○ ←
B_1	\$10	∅
A_2	\$333	∅

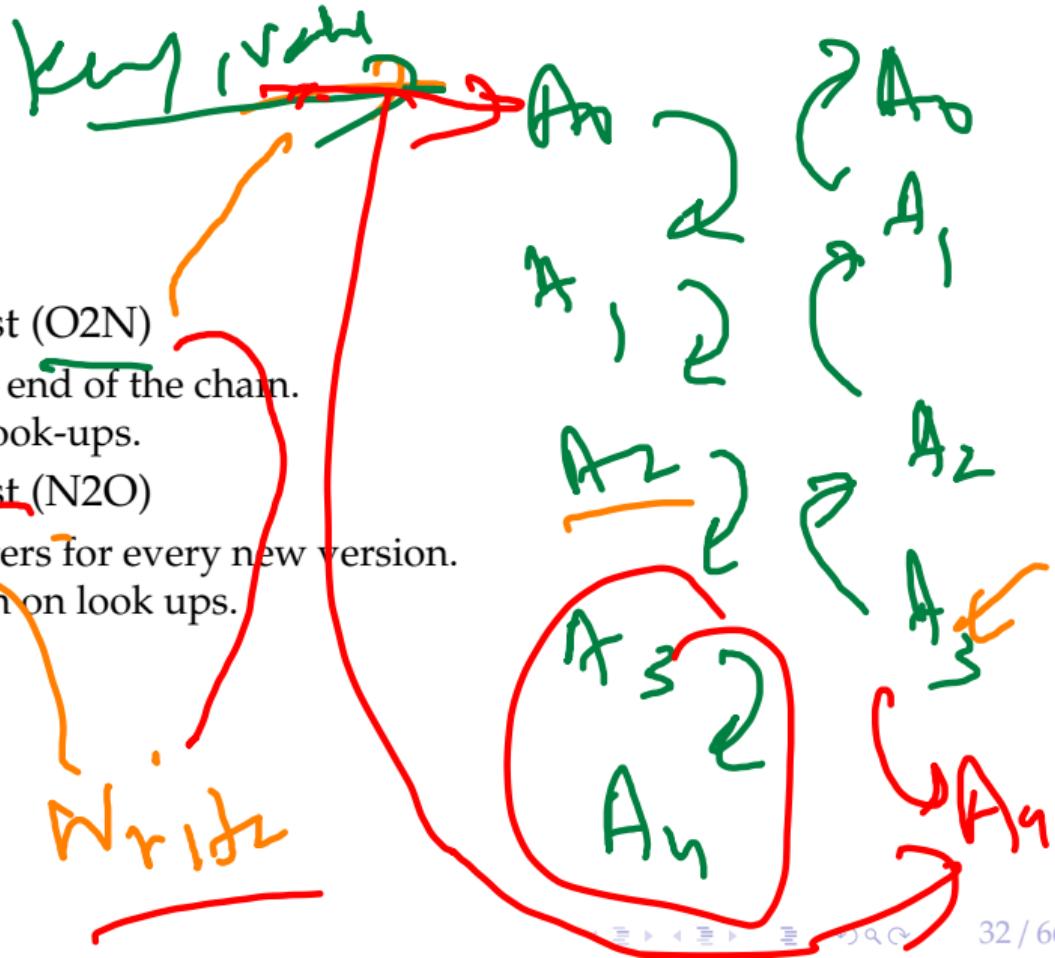
Red arrows point from the "POINTER" column back to the "VERSION" column, indicating that the current version's pointer is to its previous version. The last two rows, B_1 and A_2 , have empty pointers, indicating they are the first versions in the sequence.

Version Chain Ordering

- **Approach 1: Oldest-to-Newest (O2N)**
 - ▶ Just append new version to end of the chain.
 - ▶ Have to traverse chain on look-ups.
- **Approach 2: Newest-to-Oldest (N2O)**
 - ▶ Have to update index pointers for every new version.
 - ▶ Don't have to traverse chain on look ups.

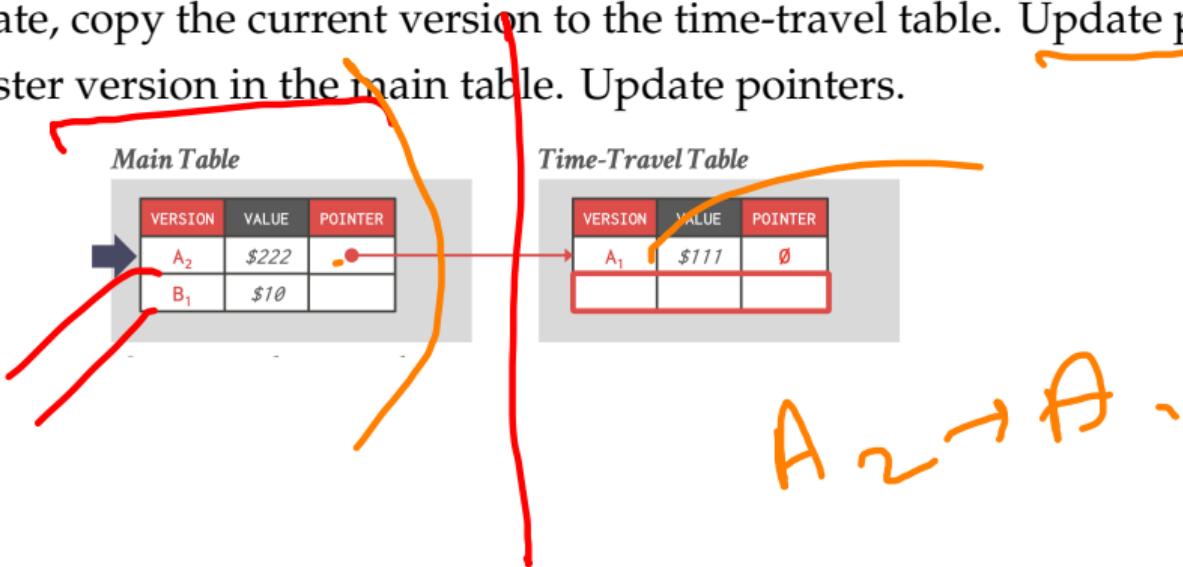
Read

Write



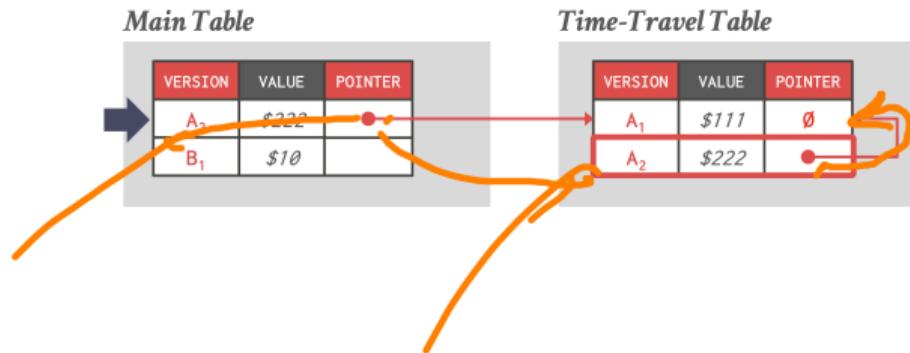
Time-Travel Storage

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



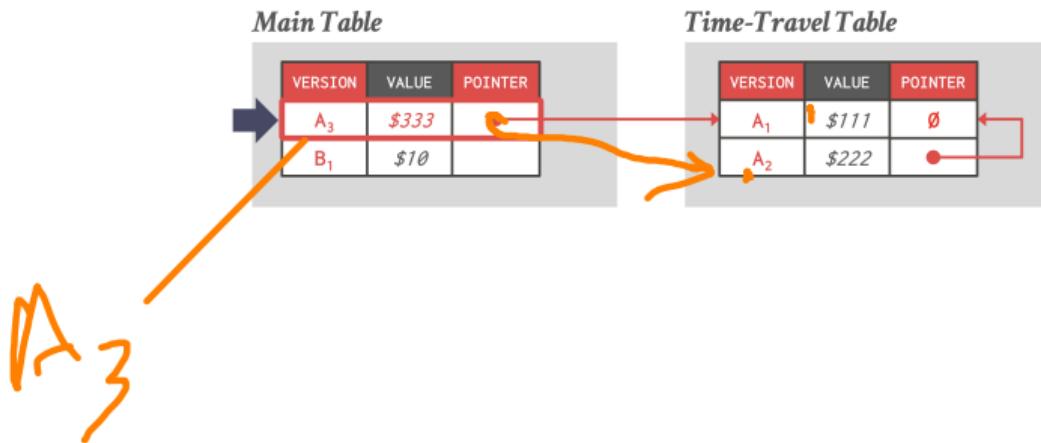
Time-Travel Storage

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



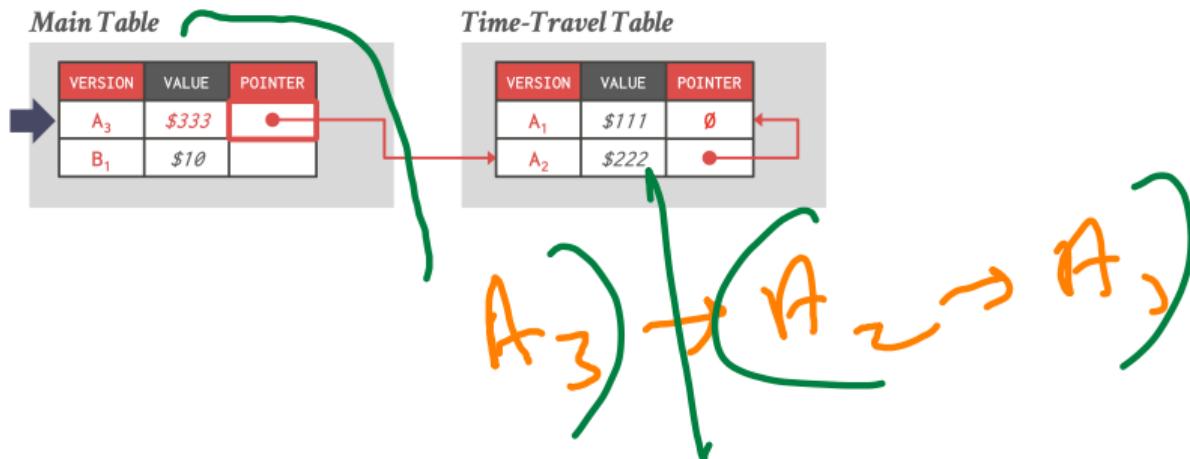
Time-Travel Storage

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



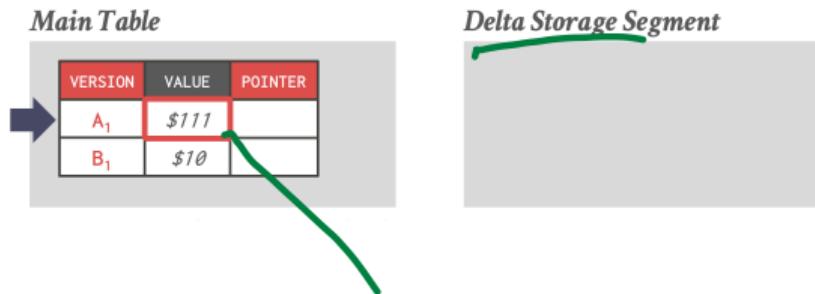
Time-Travel Storage

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



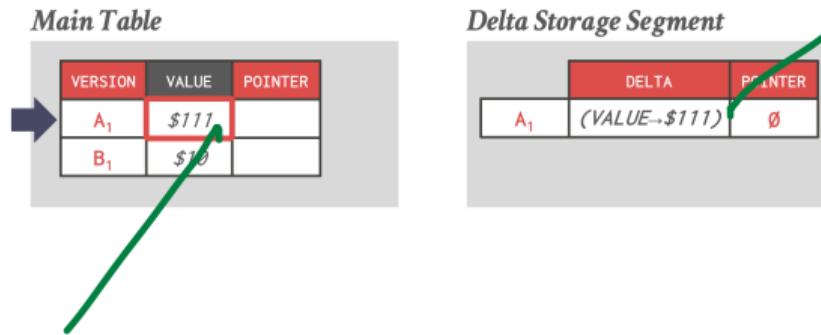
Delta Storage

- On every update, copy only the values that were modified to the delta storage and overwrite the master version.
- Txns can recreate old versions by applying the delta in reverse order.



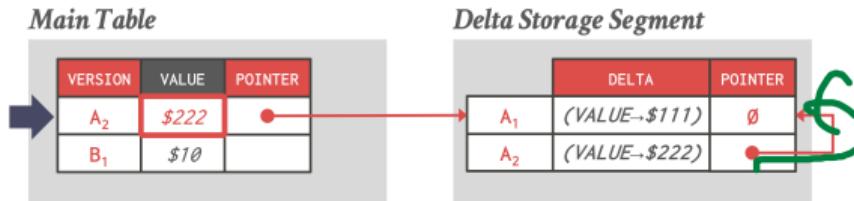
Delta Storage

- On every update, copy only the values that were modified to the delta storage and overwrite the master version.
- Txns can recreate old versions by applying the delta in reverse order.

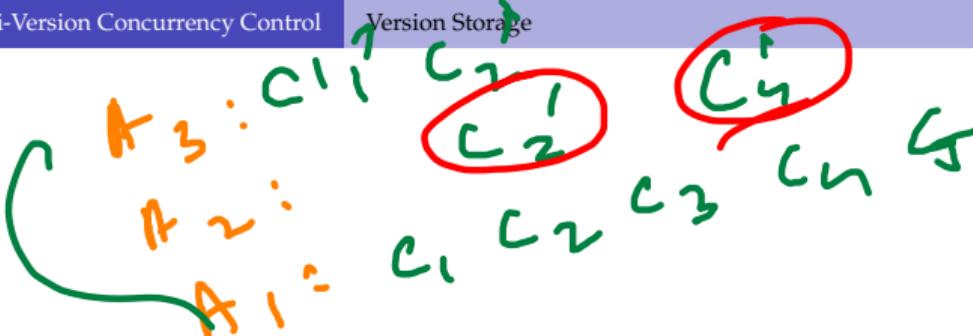


Delta Storage

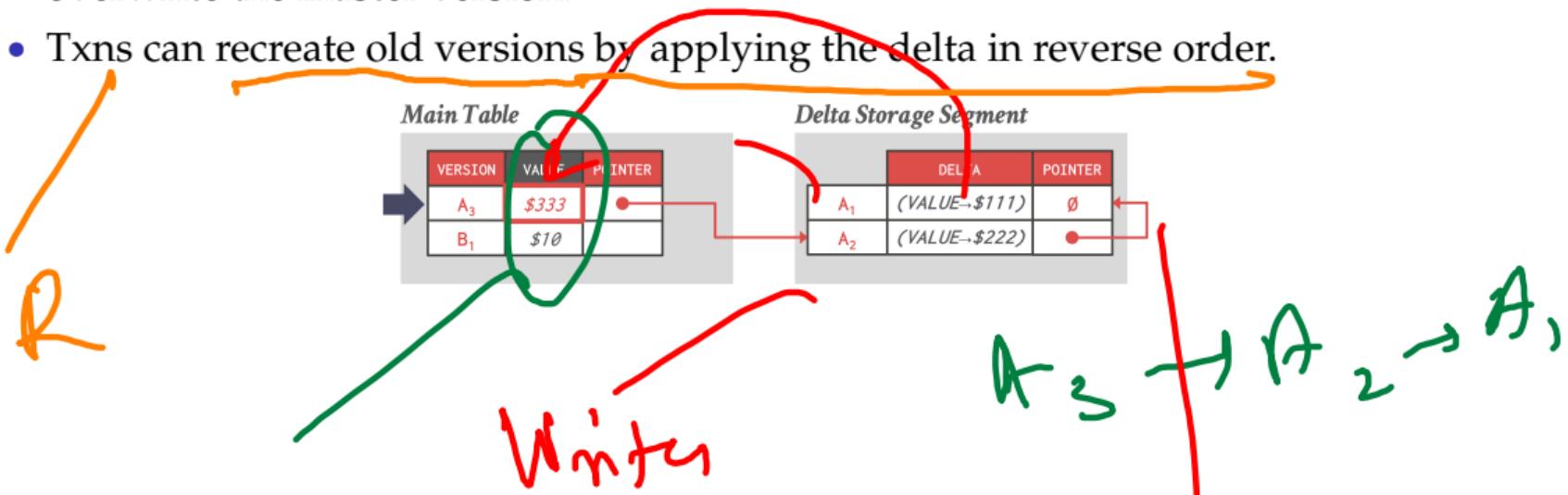
- On every update, copy only the values that were modified to the delta storage and overwrite the master version.
- Txns can recreate old versions by applying the delta in reverse order.



Delta Storage



- On every update, copy only the values that were modified to the delta storage and overwrite the master version.
 - Txns can recreate old versions by applying the delta in reverse order.



Garbage Collection

Garbage Collection

- The DBMS needs to remove reclaimable physical versions from the database over time.
 - ▶ No active txn in the DBMS can see that version (SI).
 - ▶ The version was created by an aborted txn.
- Two additional design decisions:
 - ▶ How to look for expired versions?
 - ▶ How to decide when it is safe to reclaim memory?

Snapshot
Isolation

Garbage Collection

VACUUM

- Approach 1: Tuple-level

- ▶ Find old versions by examining tuples directly.
- ▶ Background Vacuuming vs. Cooperative Cleaning

- Approach 2: Transaction-level

- ▶ Txns keep track of their old versions so the DBMS does not have to scan tuples to determine visibility.

OU

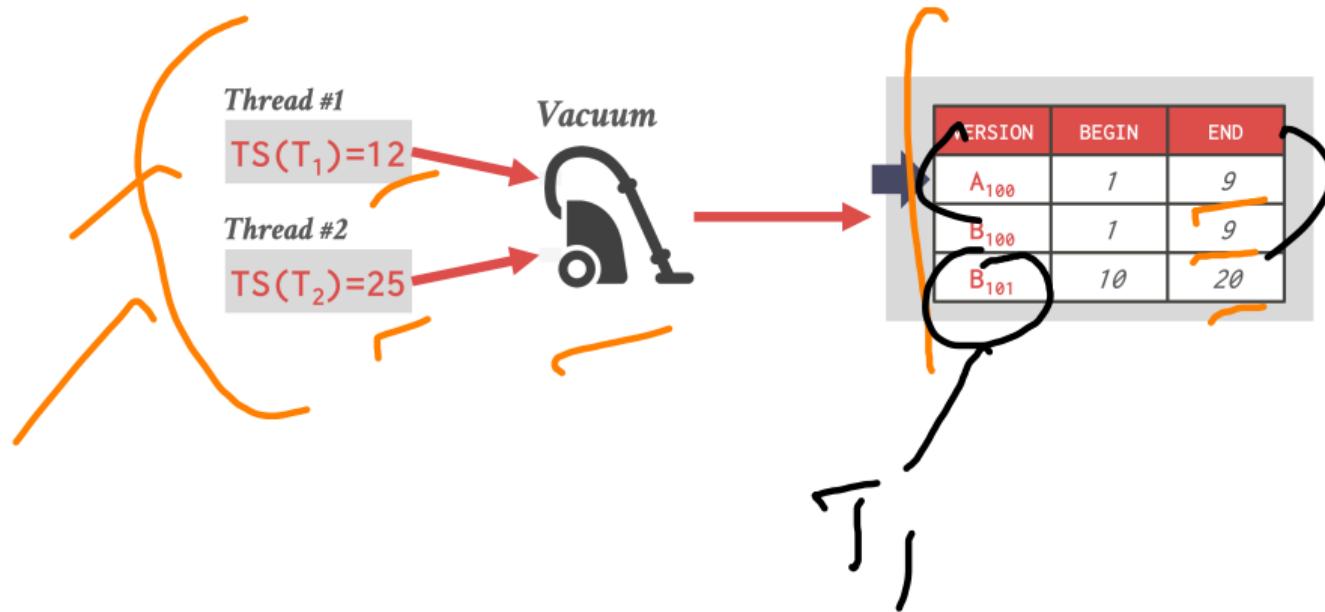
Tuple-level GC

- **Background Vacuuming:**

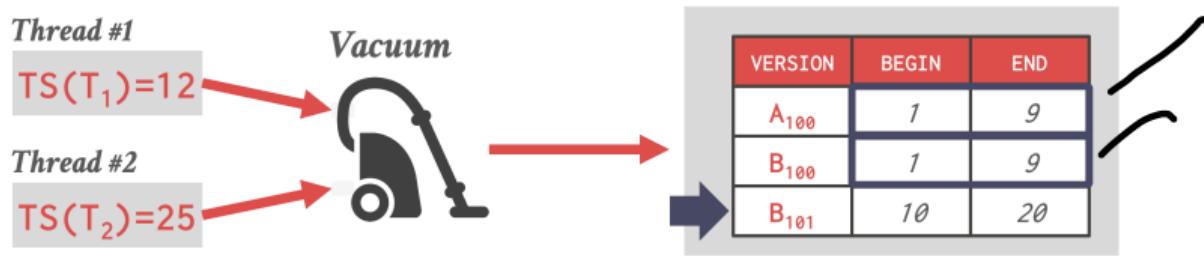
- Separate thread(s) periodically scan the table and look for reclaimable versions.
- Works with any storage.

0 2 4 N 2 2

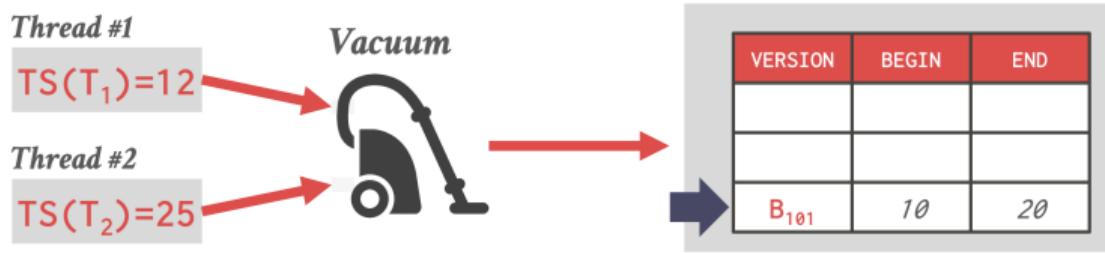
Tuple-level GC



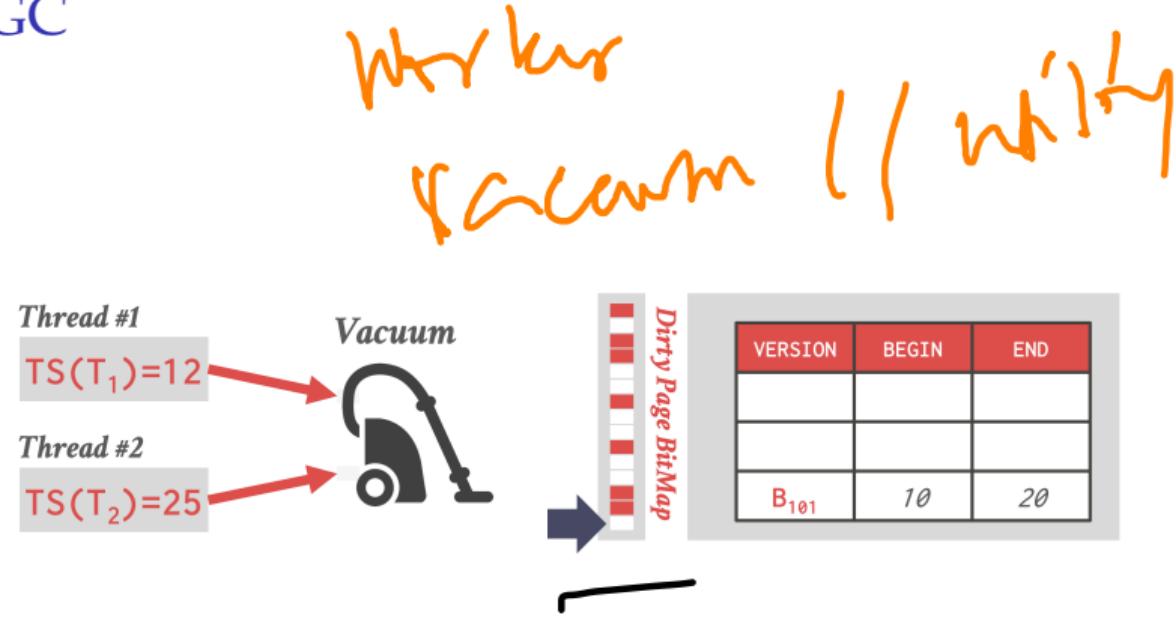
Tuple-level GC



Tuple-level GC



Tuple-level GC



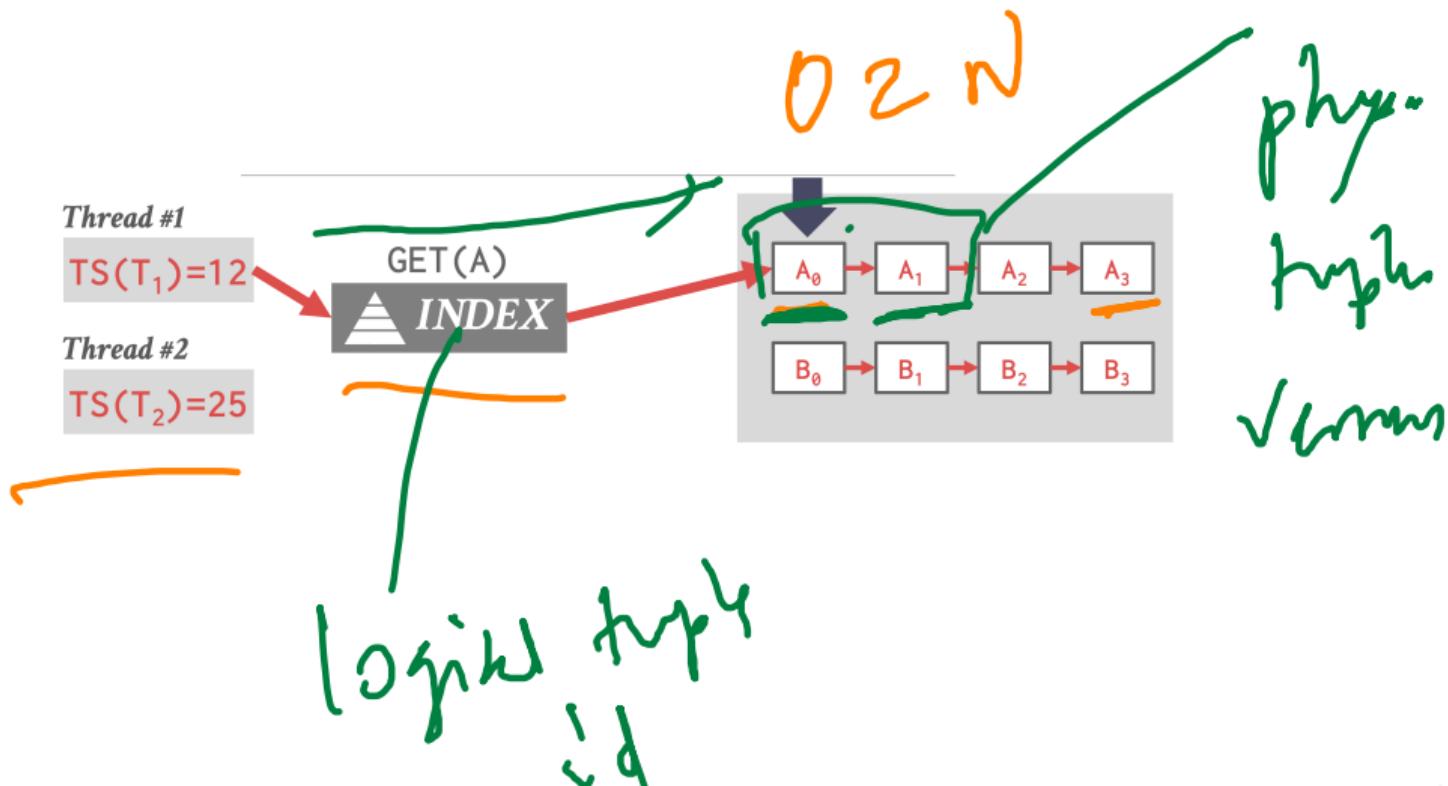
PostgreSQL 13

Tuple-level GC



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

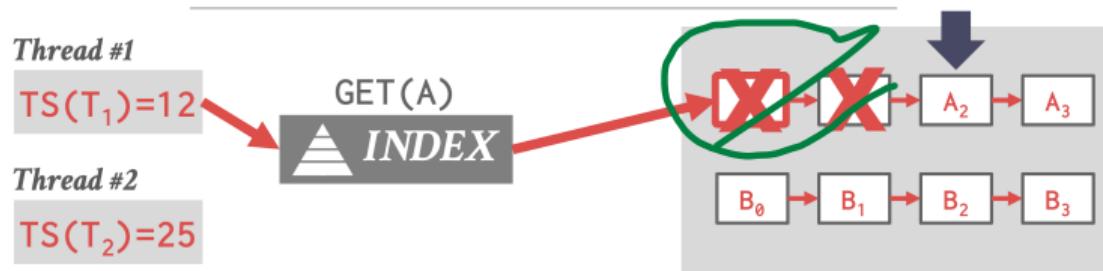
Tuple-level GC



Tuple-level GC

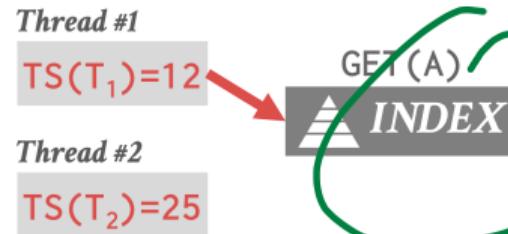


Tuple-level GC

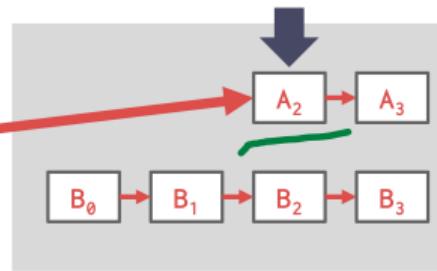


Tuple-level GC

Predictability



logged 2. 100 version



Vanillin

"dusty corners"

Transaction-level GC

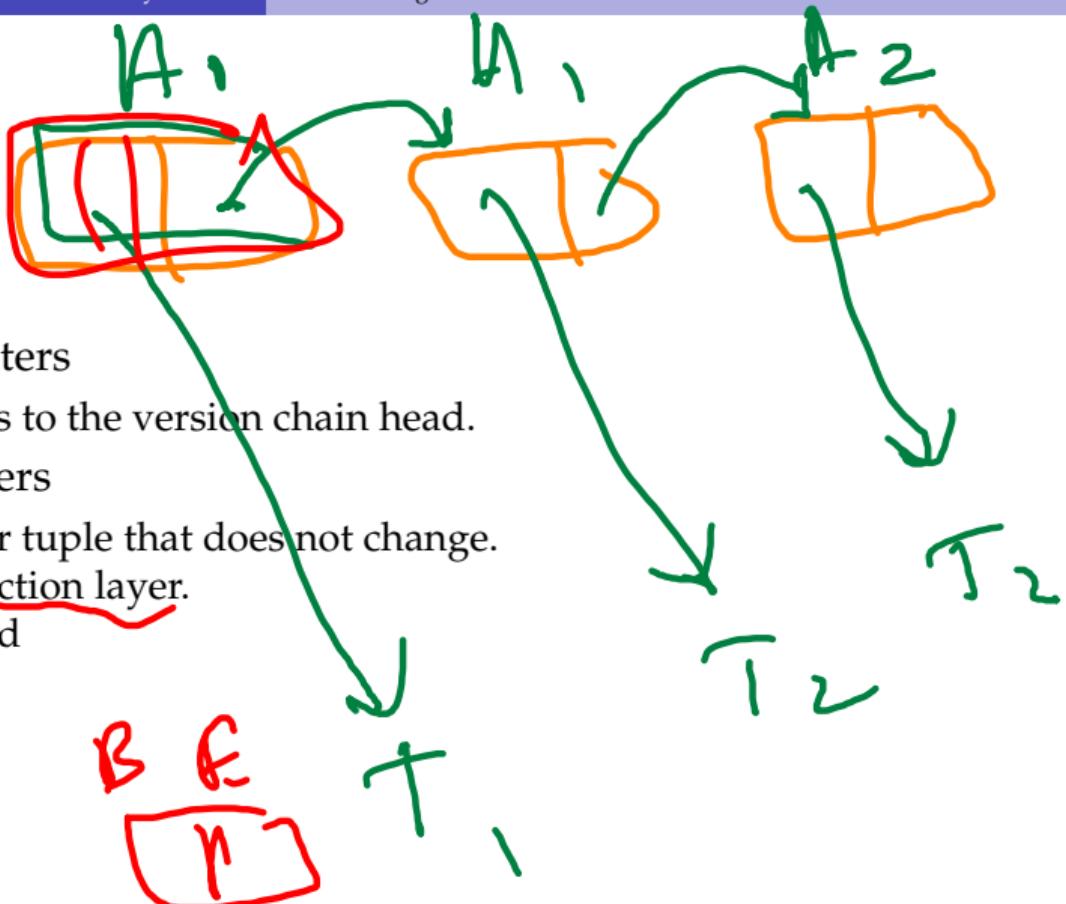
- Each txn keeps track of its read/write set.
 - The DBMS determines when all versions created by a finished txn are no longer visible.
- 

Index Management

Index Management

- Primary key indexes point to version chain head.
 - ▶ How often the DBMS has to 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 an **DELETE** followed by an **INSERT**.
 - Secondary indexes are more complicated...

Secondary Indexes



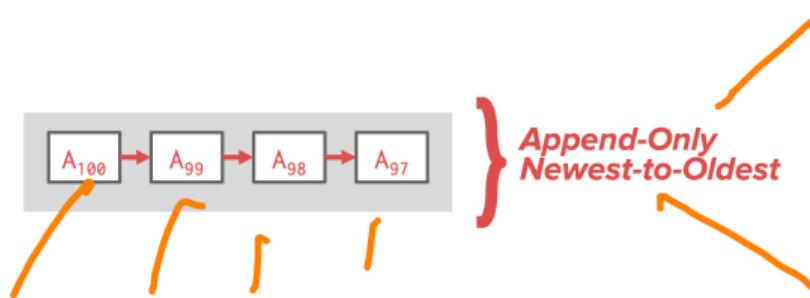
- **Approach 1: Physical Pointers**

- Use the physical address to the version chain head.

- **Approach 2: Logical Pointers**

- Use a fixed identifier per tuple that does not change.
- Requires an extra indirection layer.
- Primary Key vs. Tuple Id

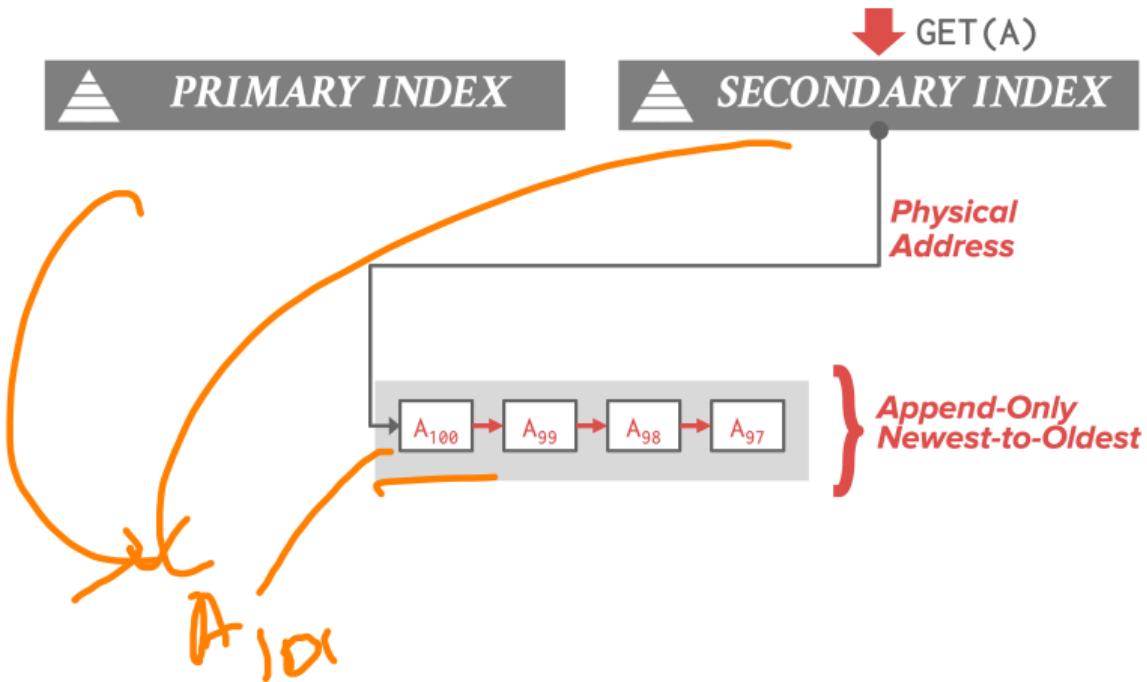
Physical Pointers



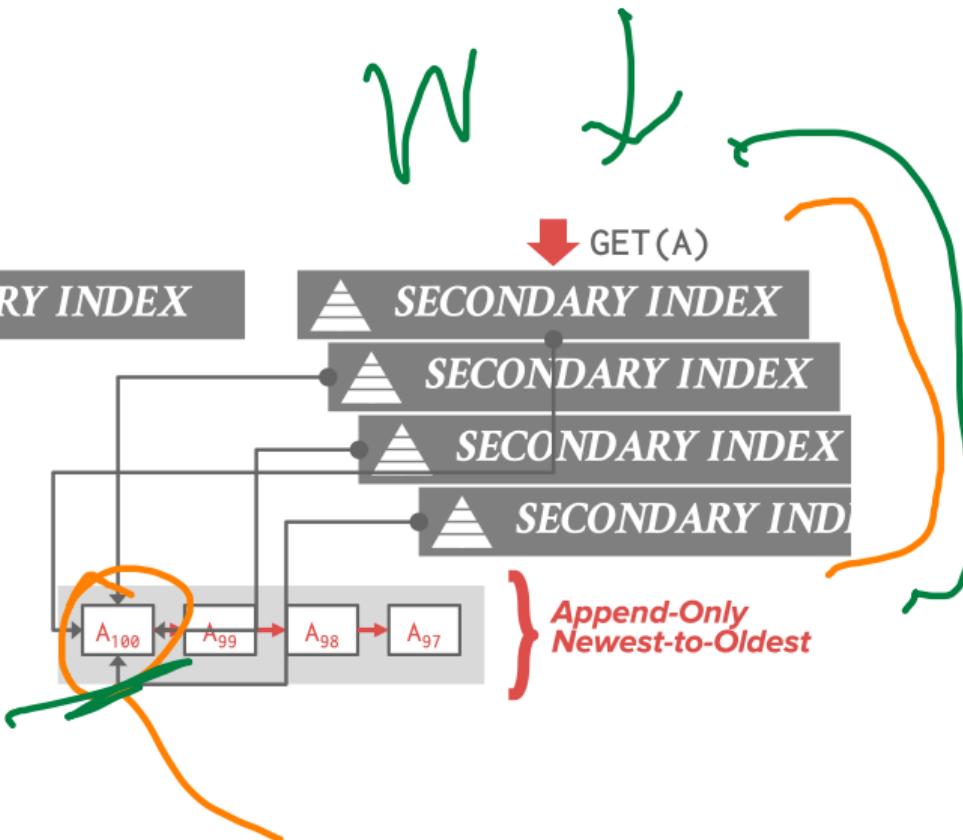
Physical Pointers



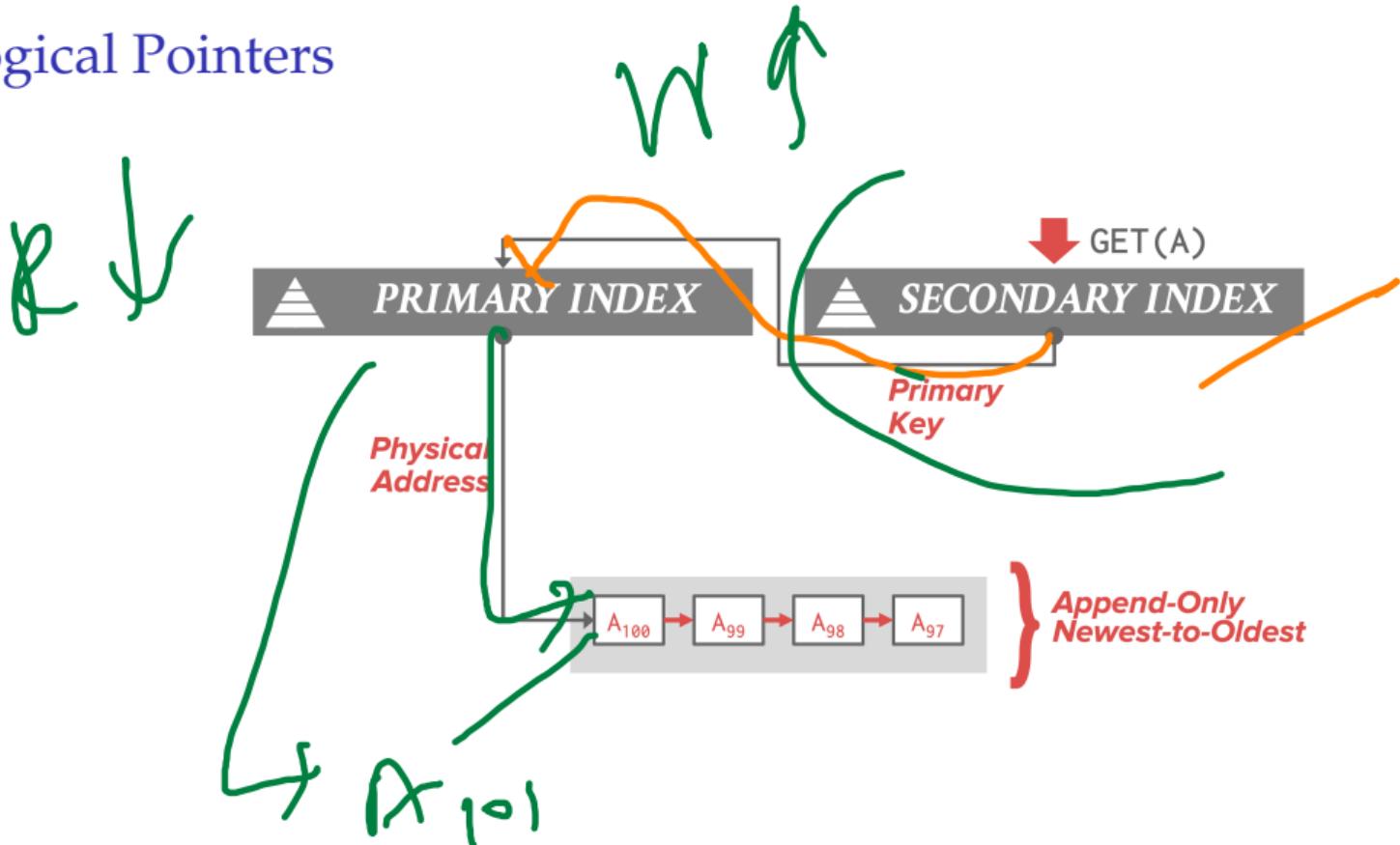
Physical Pointers



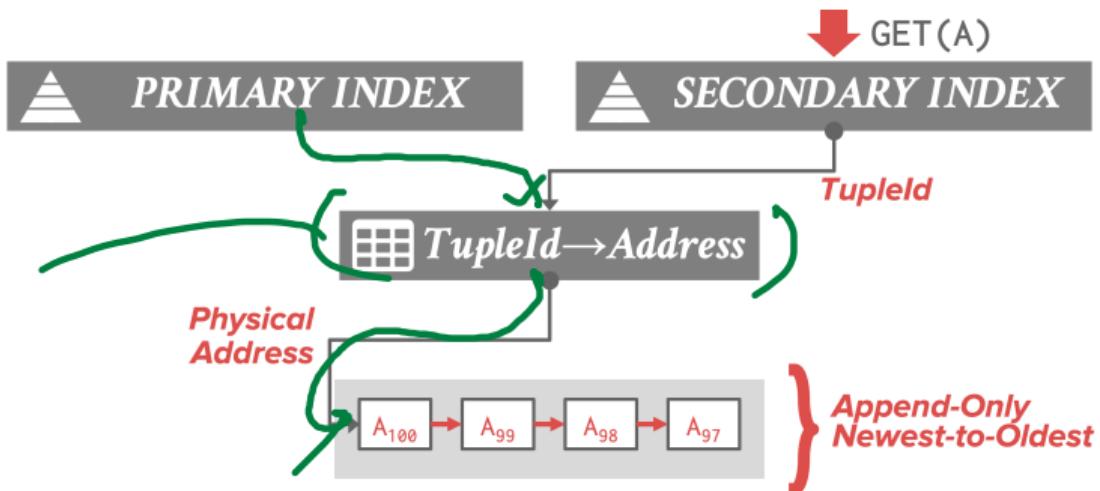
Physical Pointers



Logical Pointers



Logical Pointers

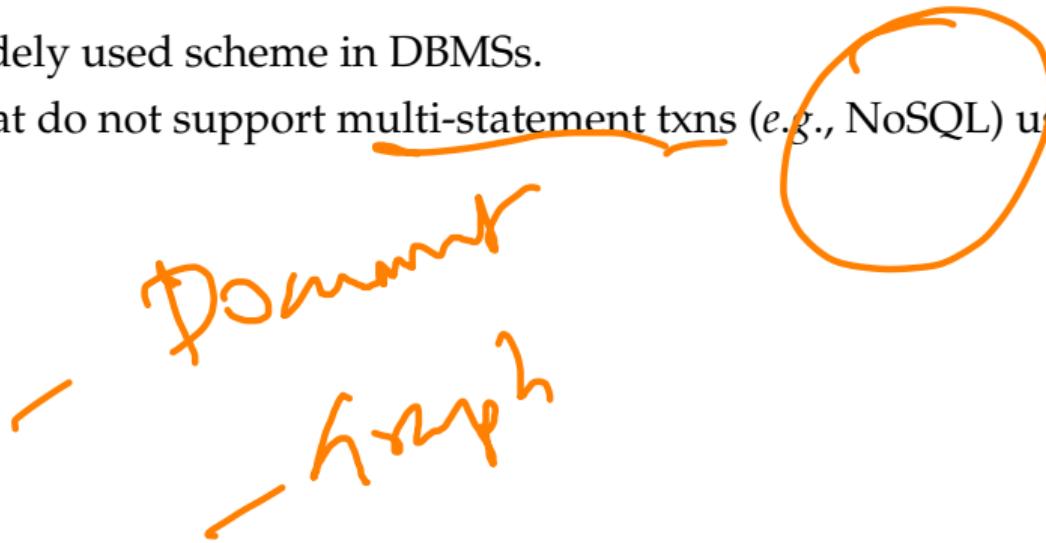


MVCC Implementations

DBMS	Protocol	Version Storage	Garbage Collection	Indexes
Oracle	MV2PL	Delta	Vacuum	Logical
Postgres	MV-2PL/MV-TO	Append-Only	Vacuum	Physical
MySQL-InnoDB	MV-2PL	Delta	Vacuum	Logical
HYRISE	MV-OCC	Append-Only	-	Physical
Hekaton	MV-OCC	Append-Only	Cooperative	Physical
MemSQL	MV-OCC	Append-Only	Vacuum	Physical
SAP HANA	MV-2PL	Time-travel	Hybrid	Logical
NuoDB	MV-2PL	Append-Only	Vacuum	Logical
HyPer	MV-OCC	Delta	Txn-level	Logical

Conclusion

- MVCC is the widely used scheme in DBMSs.
- Even systems that do not support multi-statement txns (e.g., NoSQL) use it.



Next Class

- Advanced topics in Concurrency Control