

Multi-Version Concurrency Control



Lecture #19



Database Systems
15-445/15-645
Fall 2018

AP

Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Homework #4: Monday Nov 12th @ 11:59pm

Project #3: Monday Nov 19th @ 11:59am



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.
- When a txn reads an object, it reads the newest version that existed when the txn started.



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

Writers don't block readers.

Readers don't block writers.

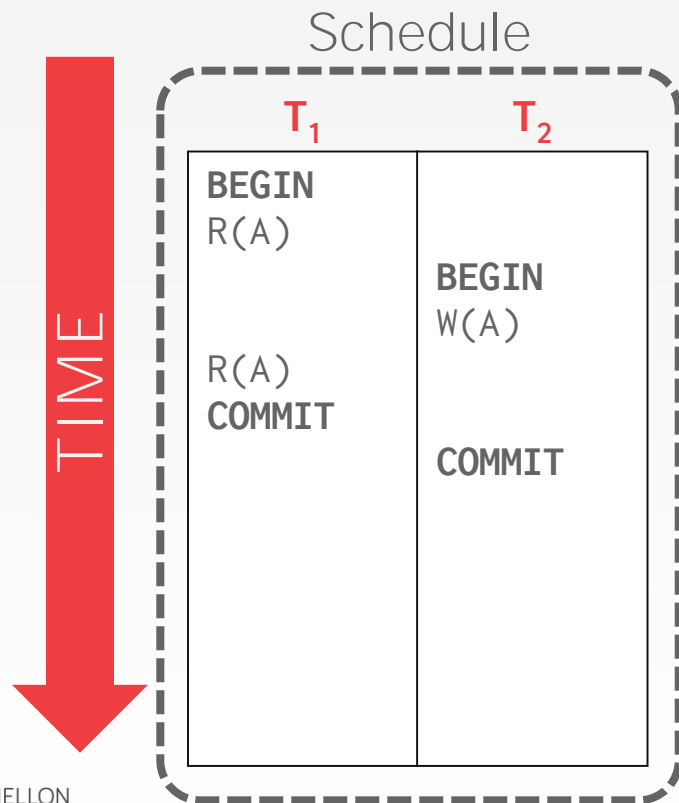
Read-only txns can read a consistent snapshot without acquiring locks.

→ Use timestamps to determine visibility.

Easily support time-travel queries.



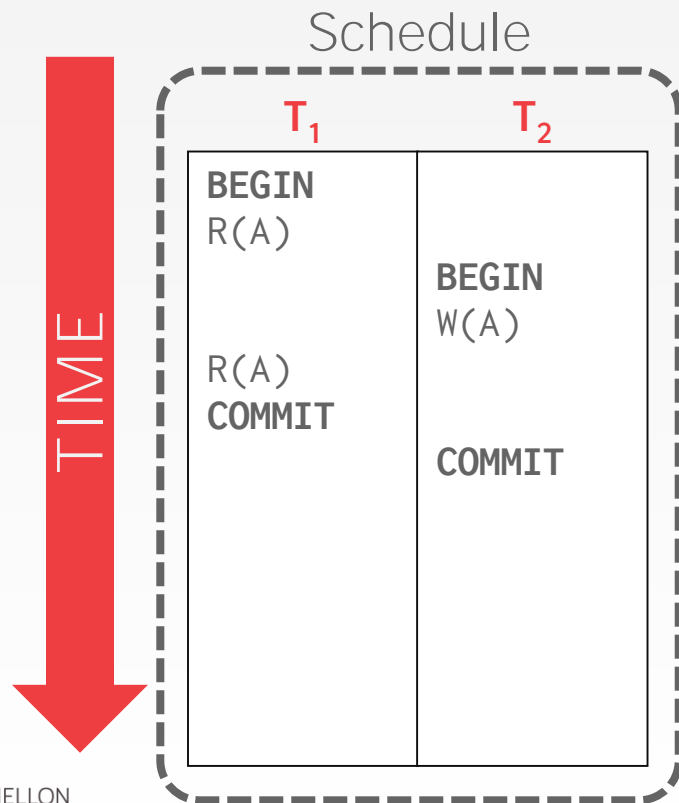
MVCC – EXAMPLE #1



Database

Version	Value	Begin	End
A_0	123	0	-

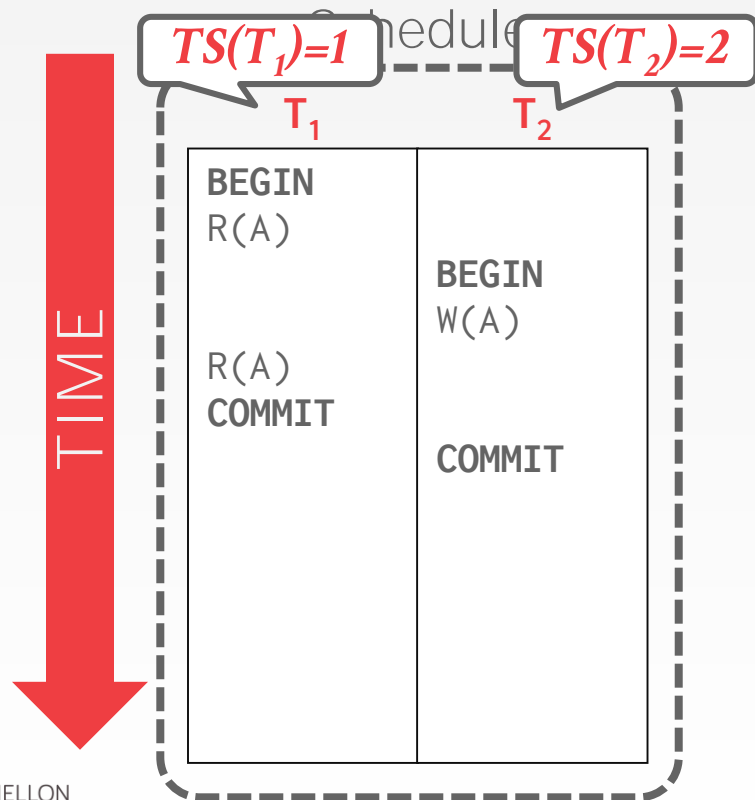
MVCC – EXAMPLE #1



Database

Version	Value	Begin	End
A_0	123	0	-

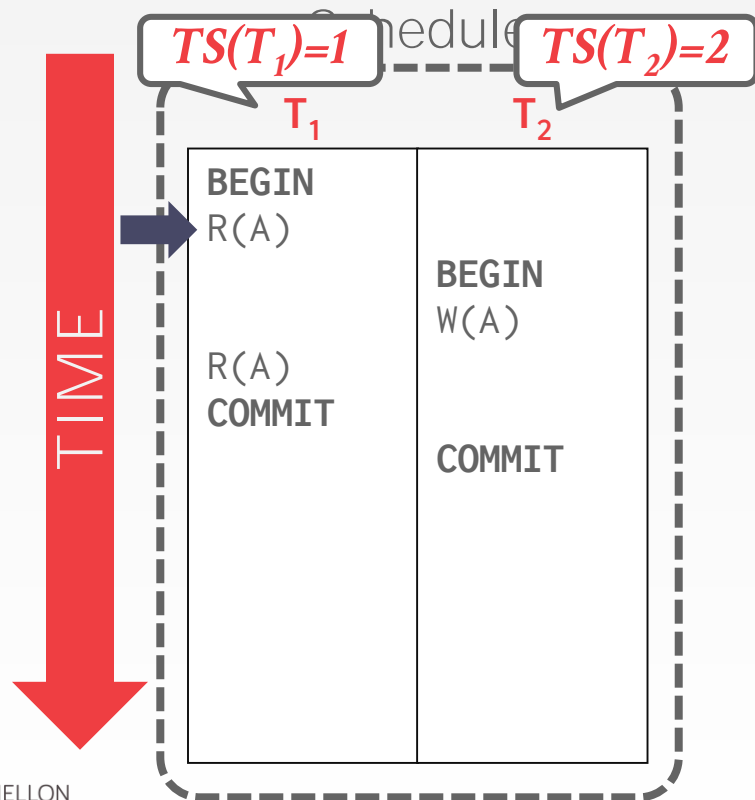
MVCC – EXAMPLE #1



Database

Version	Value	Begin	End
A_0	123	0	-

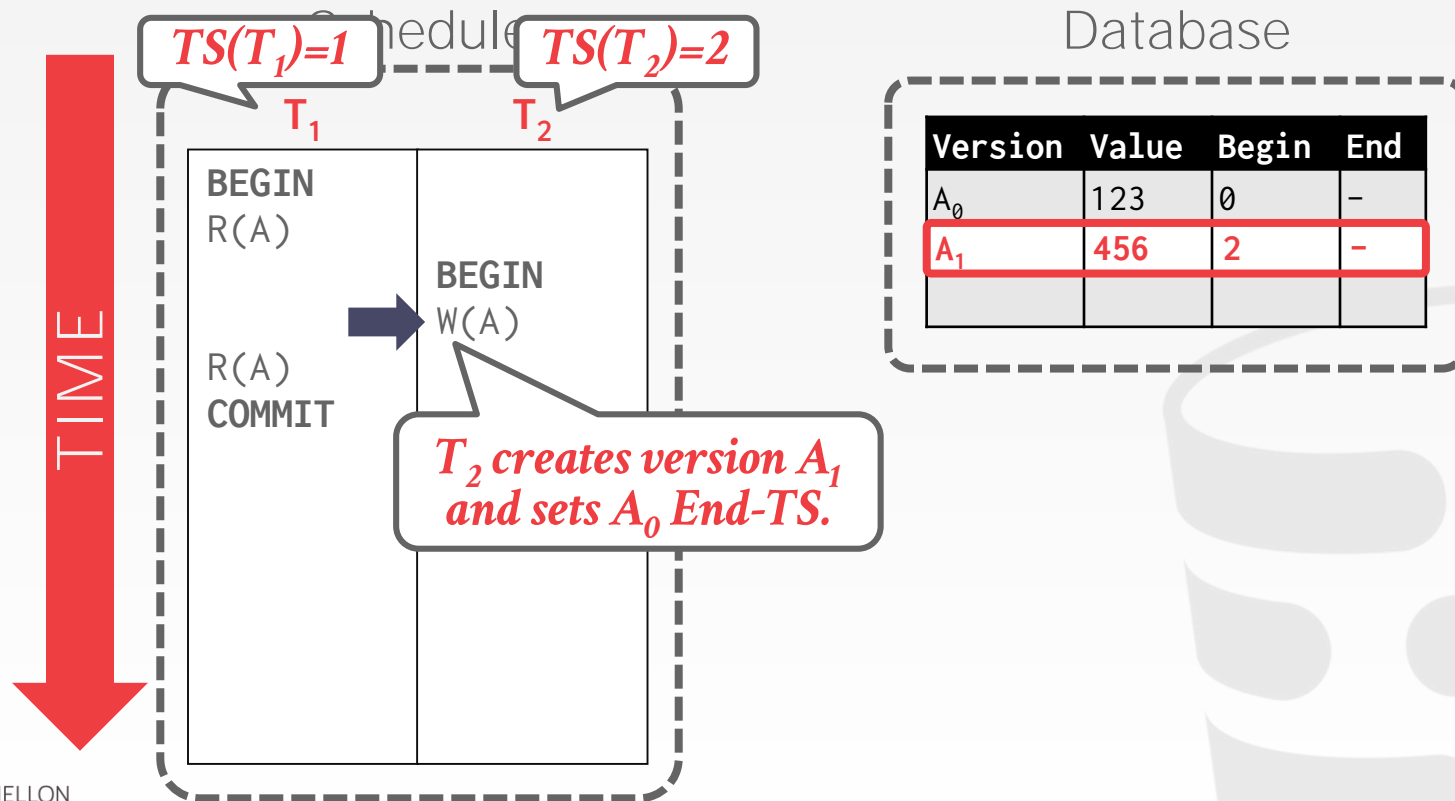
MVCC – EXAMPLE #1



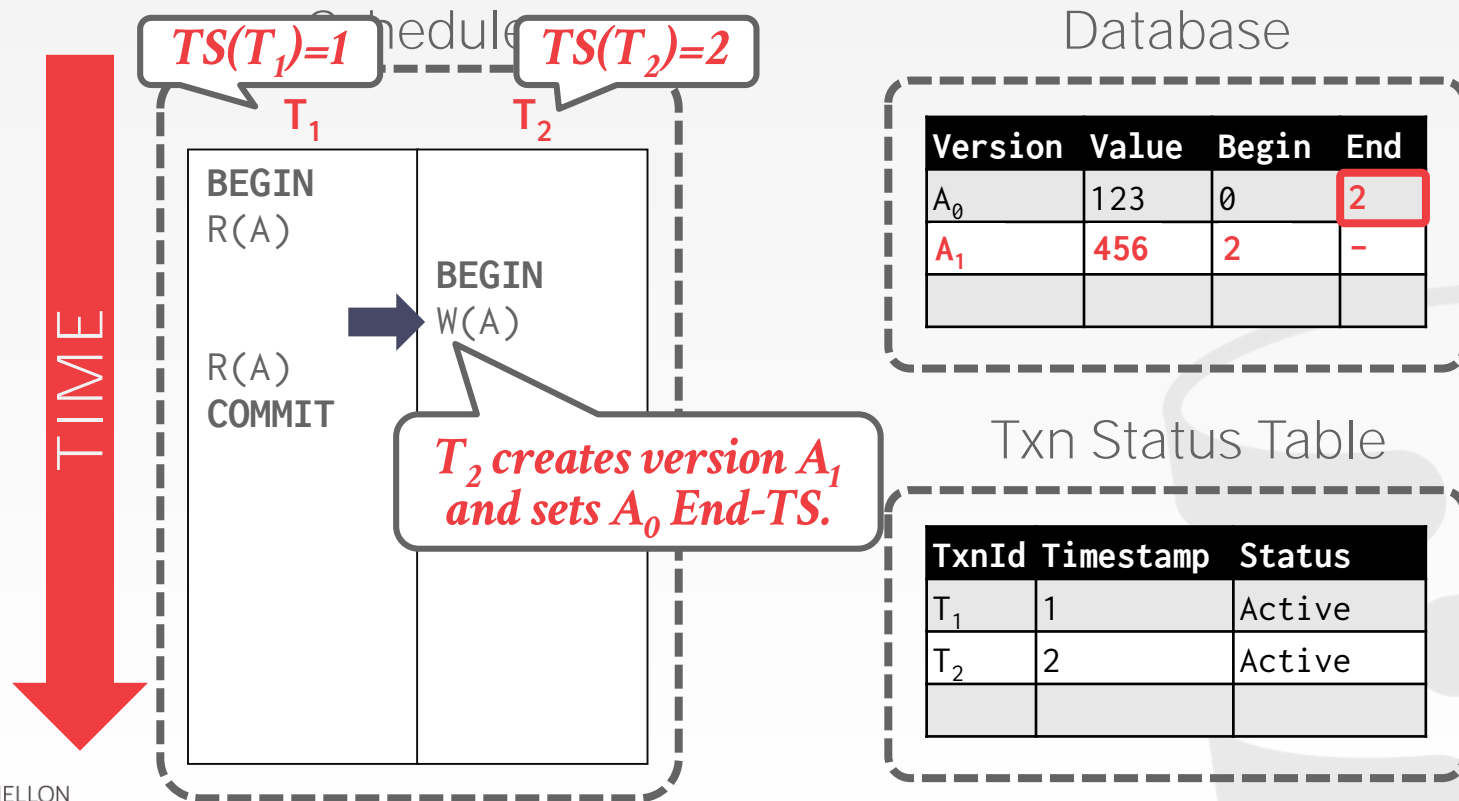
Database

Version	Value	Begin	End
A_0	123	0	-

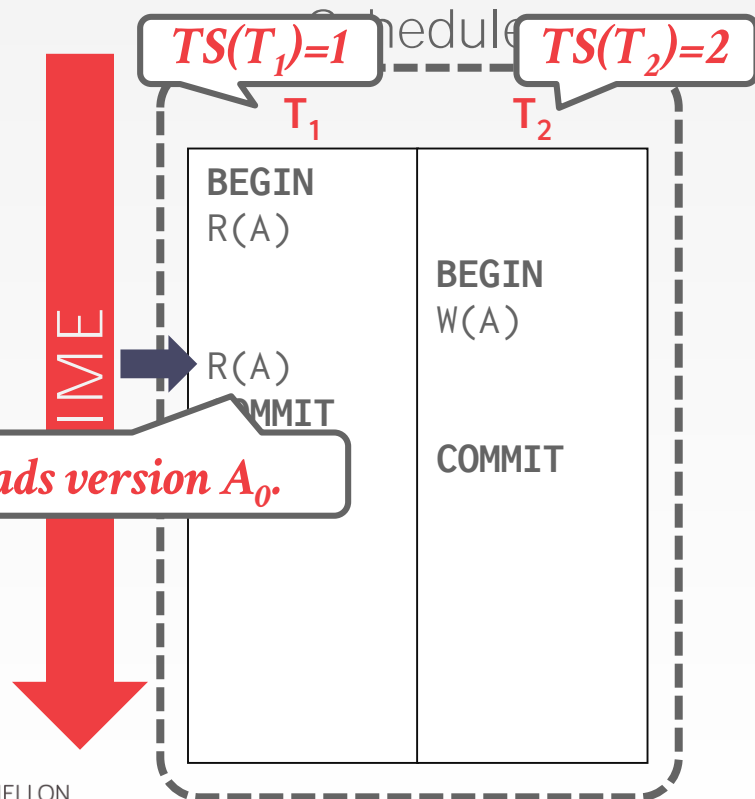
MVCC – EXAMPLE #1



MVCC – EXAMPLE #1



MVCC – EXAMPLE #1



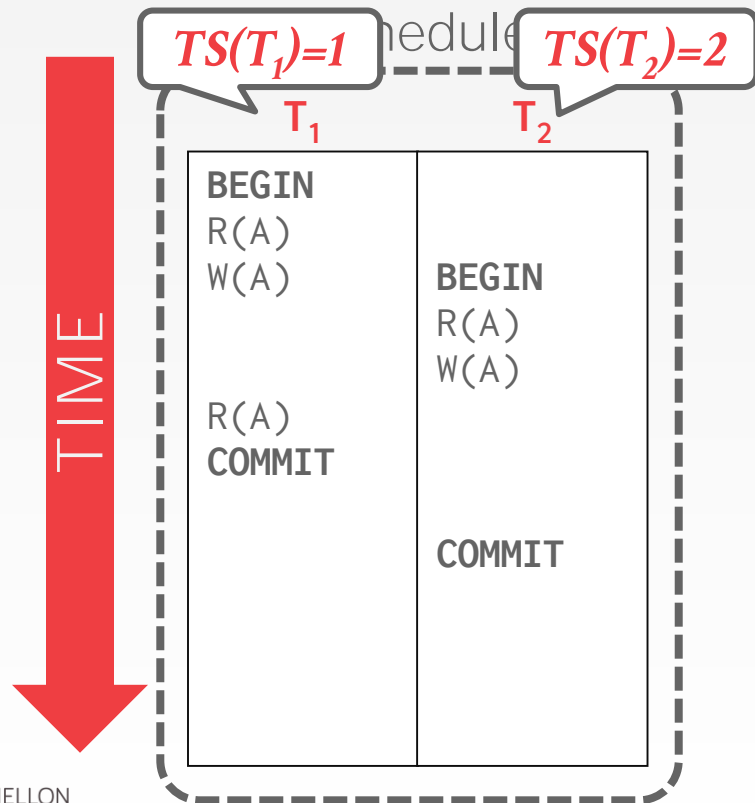
Database

Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2



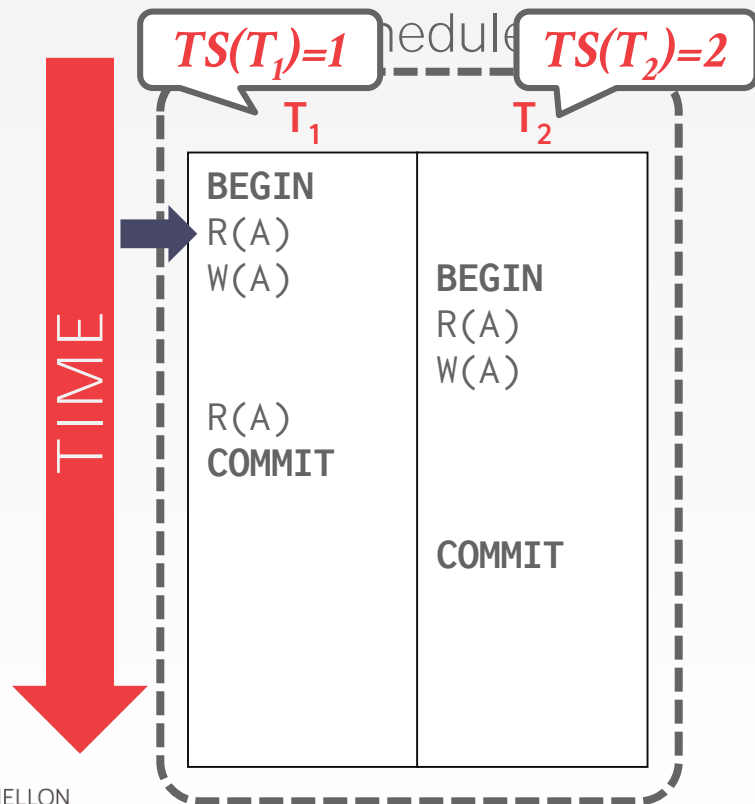
Database

Version	Value	Begin	End
A_0	123	0	

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



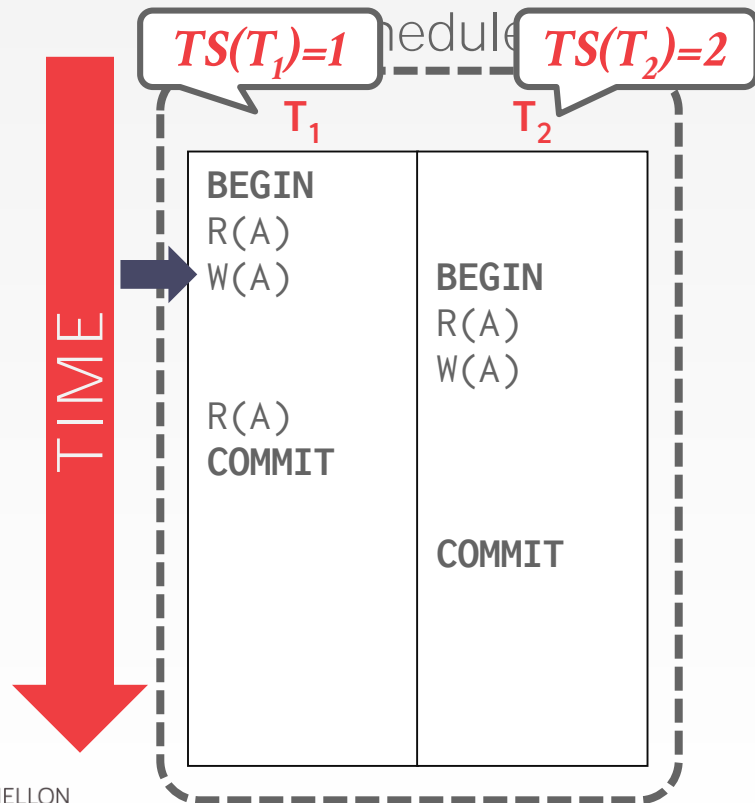
Database

Version	Value	Begin	End
A_0	123	0	

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



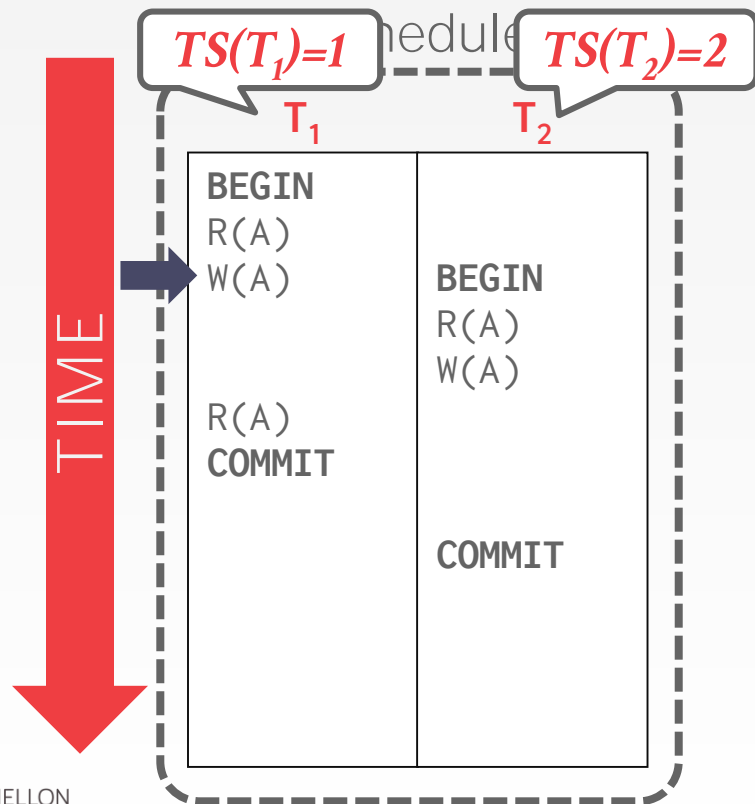
Database

Version	Value	Begin	End
A_0	123	0	
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

MVCC – EXAMPLE #2



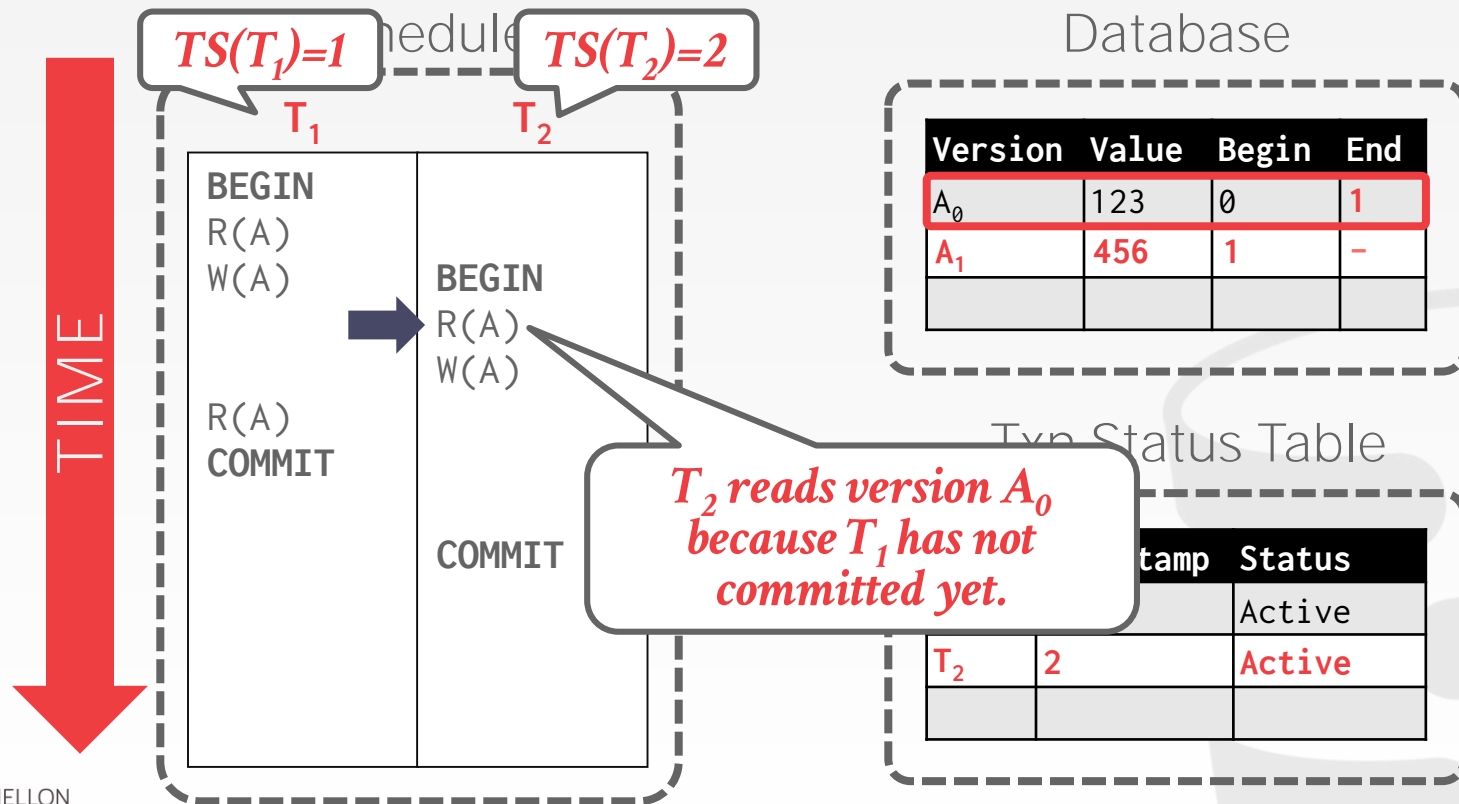
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

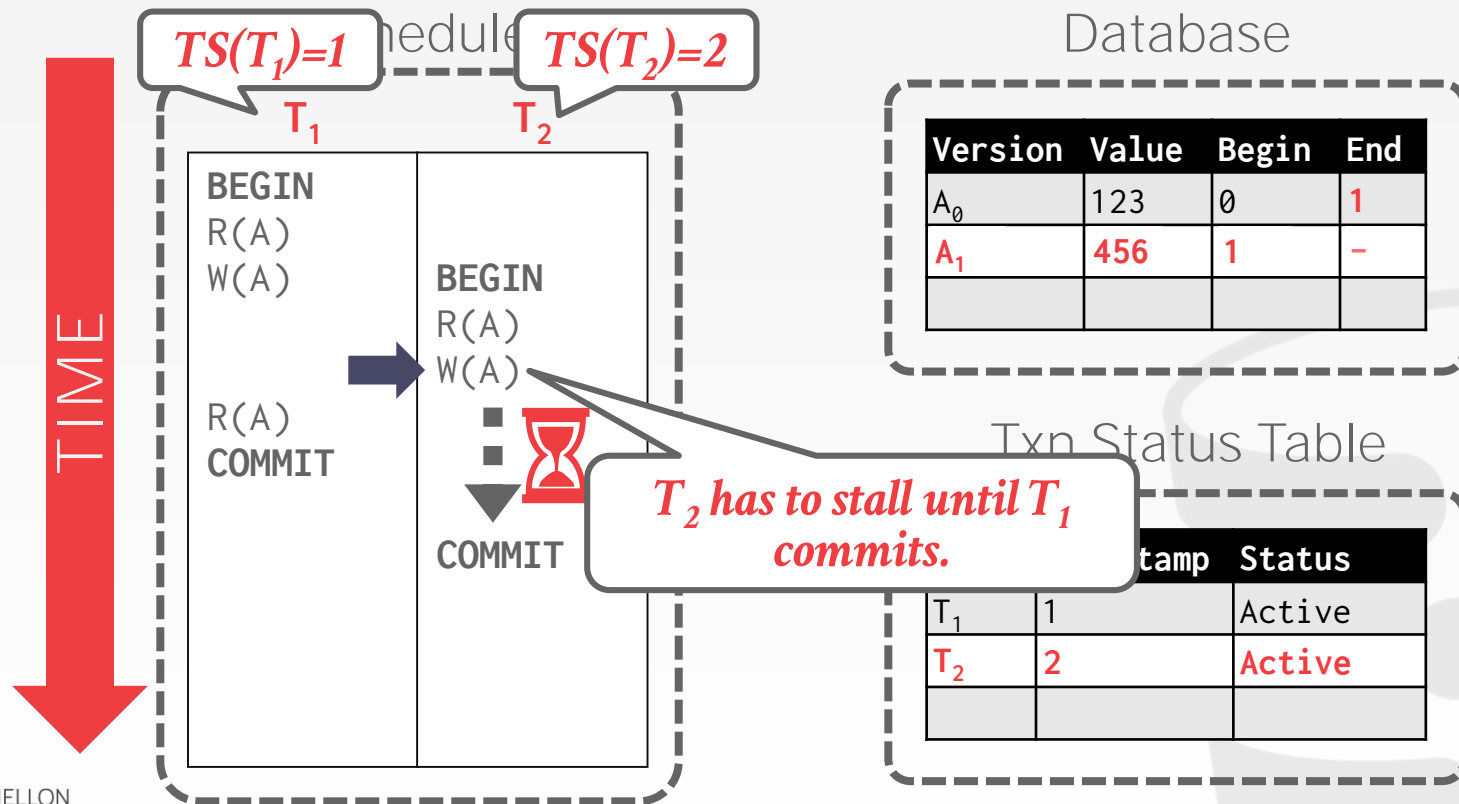
Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active

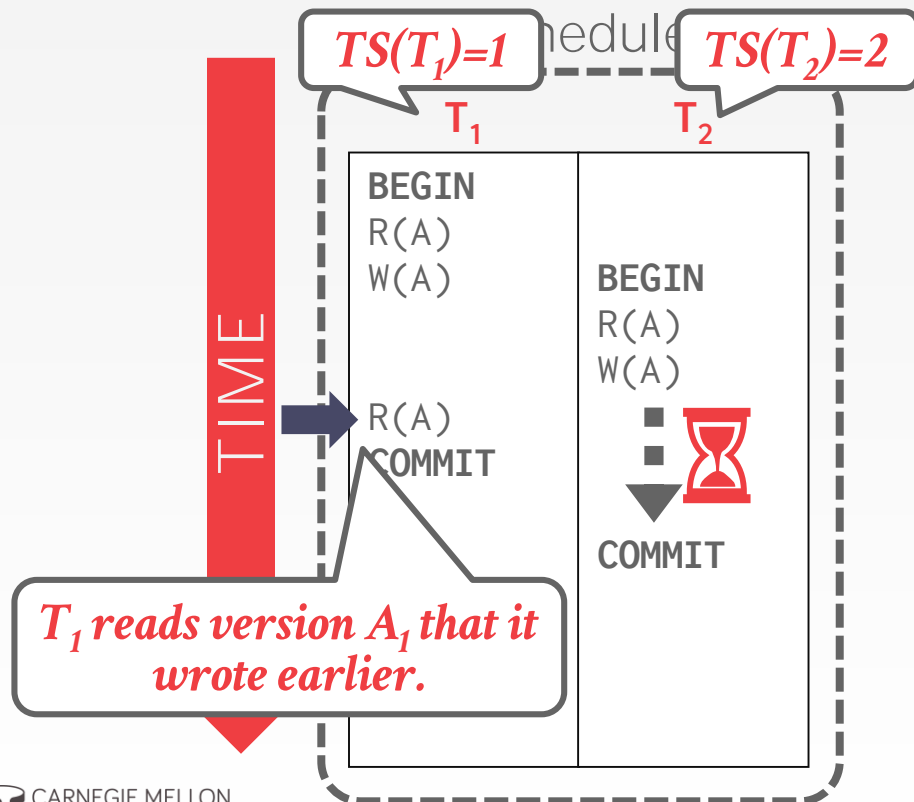
MVCC – EXAMPLE #2



MVCC – EXAMPLE #2



MVCC – EXAMPLE #2



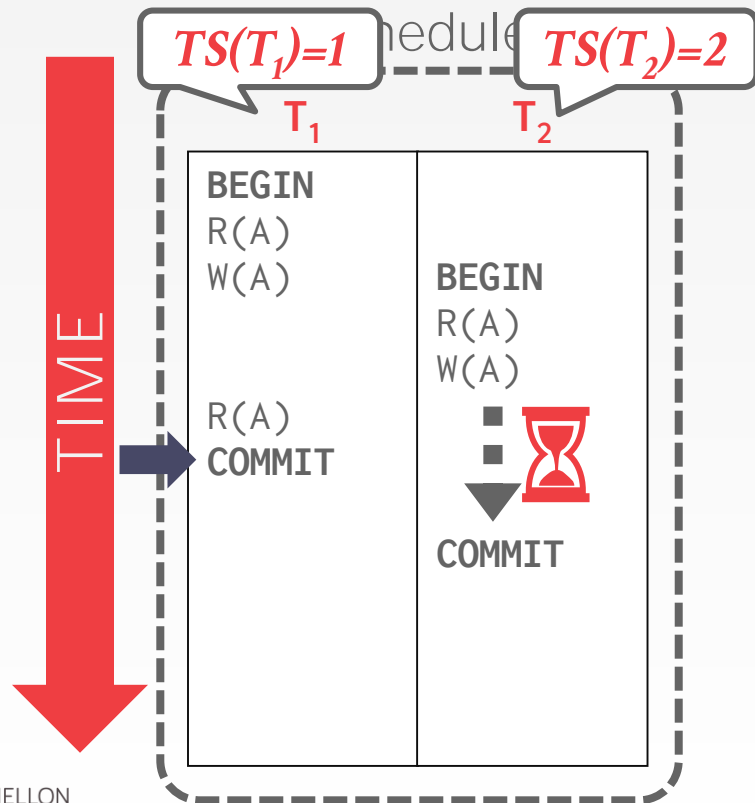
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2



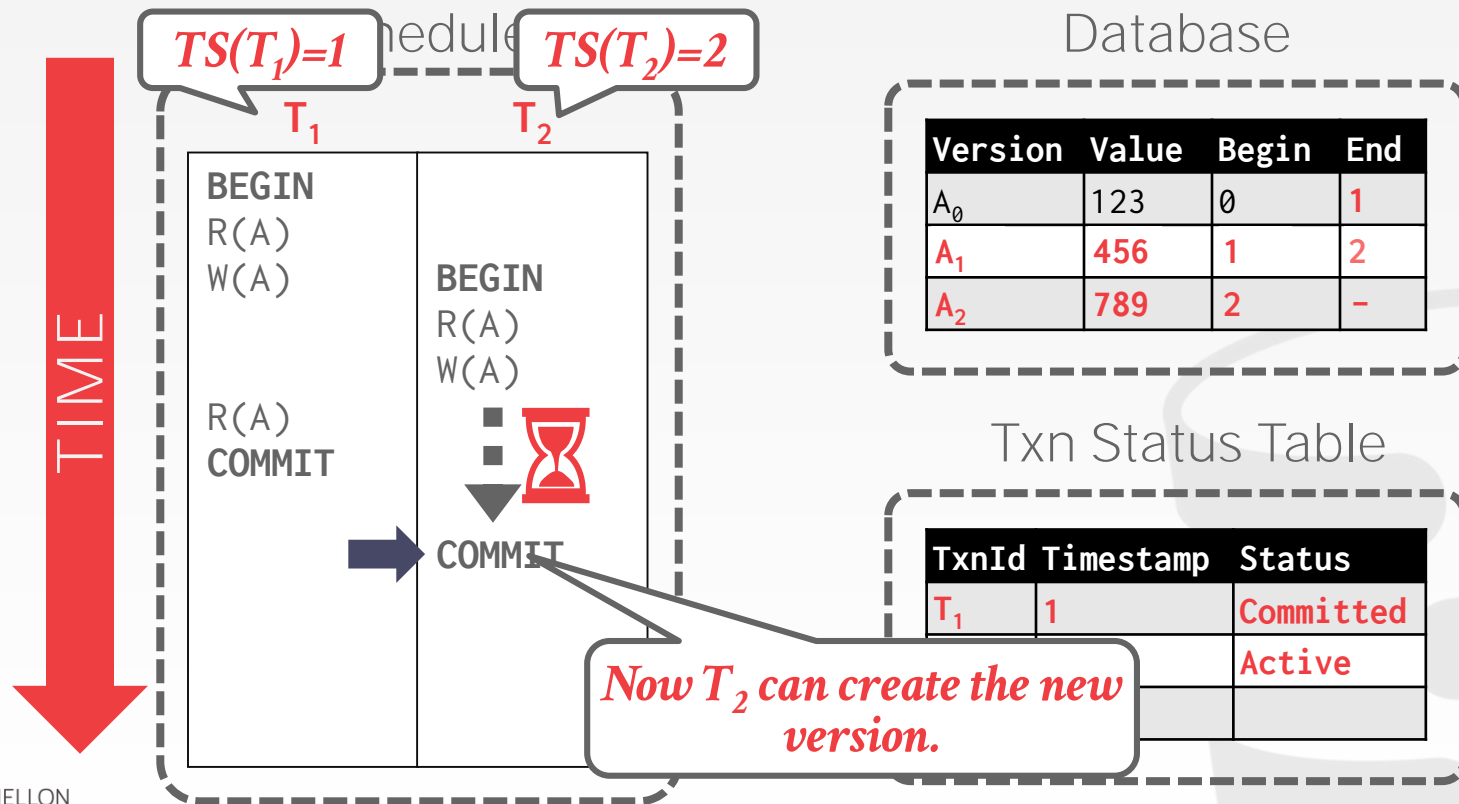
Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Committed
T_2	2	Active

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.



MVCC DESIGN DECISIONS

Concurrency Control Protocol

Version Storage

Garbage Collection

Index Management



CONCURRENCY CONTROL PROTOCOL

Approach #1: Timestamp Ordering

→ Assign txns timestamps that determine serial order.

Approach #2: Optimistic Concurrency Control

- Three-phase protocol from last class.
- Use private workspace for new versions.

Approach #3: Two-Phase Locking

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

VERSION STORAGE

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

Approach #1: Append-Only Storage

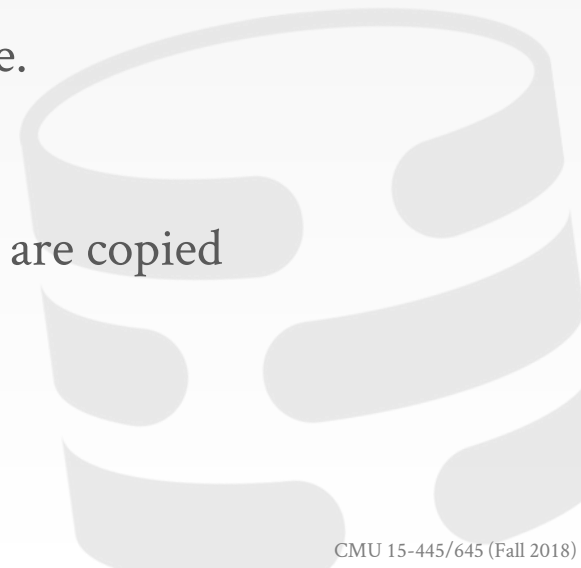
→ New versions are appended to the same table space.

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.

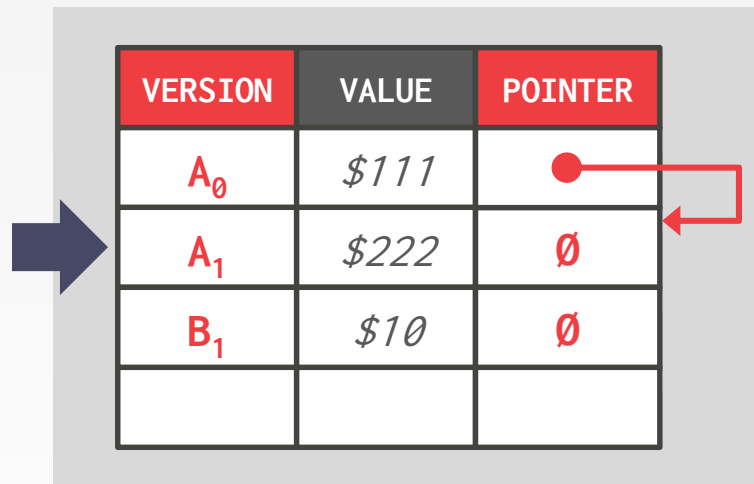


APPEND-ONLY STORAGE

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



The diagram shows a table with three columns: VERSION, VALUE, and POINTER. The first three rows contain data: A₀ with value \$111 and a pointer to the next row, A₁ with value \$222 and a null pointer, and B₁ with value \$10 and a null pointer. A blue arrow points to the first row, and a red arrow points from the pointer cell of the first row to the first row of the next page.

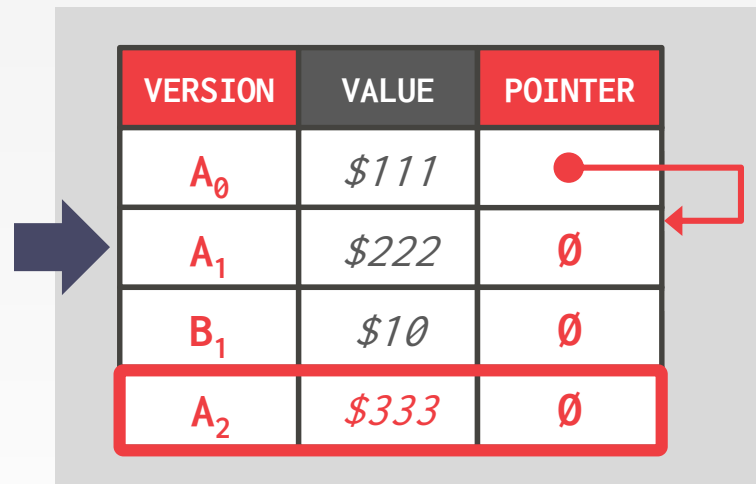
VERSION	VALUE	POINTER
A ₀	\$111	●
A ₁	\$222	∅
B ₁	\$10	∅

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 the Main Table structure. It is a table with three columns: VERSION, VALUE, and POINTER. The rows represent different versions of tuples. A blue arrow points to the table from the left. A red box highlights the row for version A₂. A red dot in the POINTER column of the row for A₀ has a red arrow pointing to the row for A₁, indicating a pointer to the next version of the tuple.

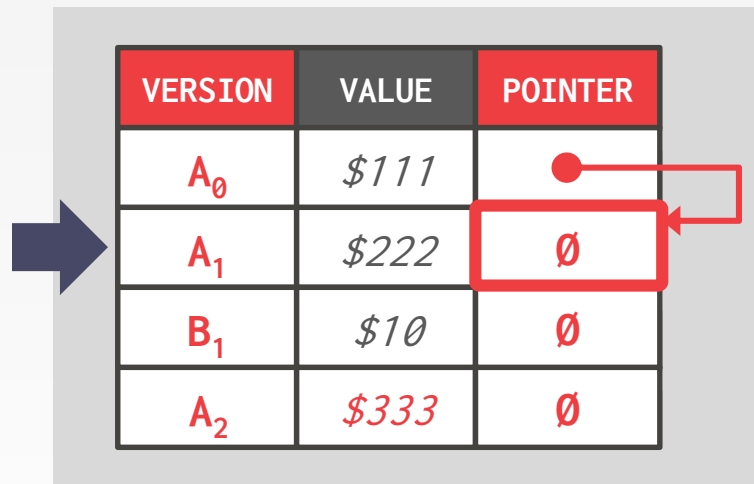
VERSION	VALUE	POINTER
A ₀	\$111	●
A ₁	\$222	∅
B ₁	\$10	∅
A ₂	\$333	∅

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 shows a table with three columns: VERSION, VALUE, and POINTER. The rows contain logical tuple versions A₀, A₁, B₁, and A₂. A red dot in the POINTER column of row A₀ has a red arrow pointing to the empty cell in the POINTER column of row A₁. A blue arrow points to the entire table structure.

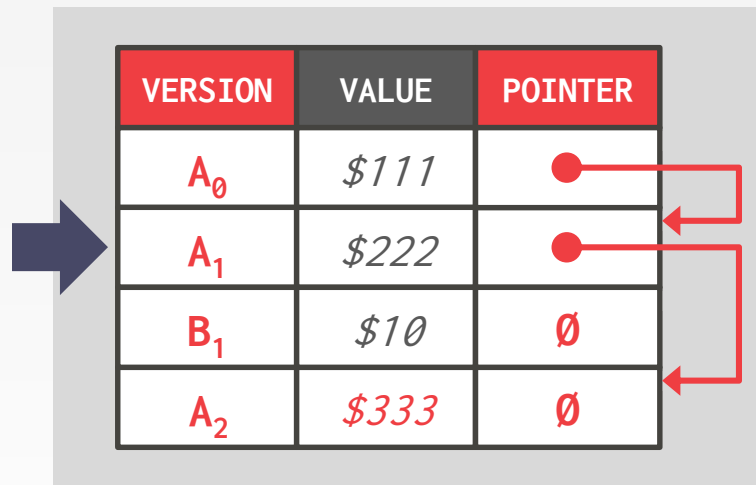
VERSION	VALUE	POINTER
A ₀	\$111	●
A ₁	\$222	∅
B ₁	\$10	∅
A ₂	\$333	∅

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 the Main Table structure. A large blue arrow points from the text on the left towards a table. The table has three columns: VERSION, VALUE, and POINTER. The first two columns are white with black text, and the third column is white with red text. The rows are: A₀ (\$111), A₁ (\$222), B₁ (\$10), and A₂ (\$333). Red dots in the POINTER column indicate pointers to the next version of the tuple. Red arrows show the sequence of updates: A₀ points to A₁, A₁ points to A₂, and B₁ points to A₂.

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

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.

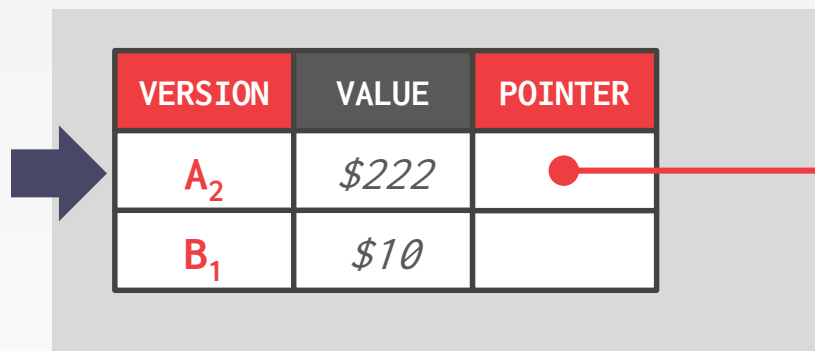
Approach #2: Newest-to-Oldest (N2O)

- Have to update index pointers for every new version.
- Don't have to traverse chain on look ups.



TIME-TRAVEL STORAGE

Main Table



VERSION	VALUE	POINTER
A_2	\$222	●
B_1	\$10	


Time-Travel Table

VERSION	VALUE	POINTER
A_1	\$111	\emptyset

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

TIME-TRAVEL STORAGE


Main Table



VERSION	VALUE	POINTER
A_2	\$222	●
B_1	\$10	

Time-Travel Table


VERSION	VALUE	POINTER
A_1	\$111	\emptyset
A_2	\$222	●



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

TIME-TRAVEL STORAGE

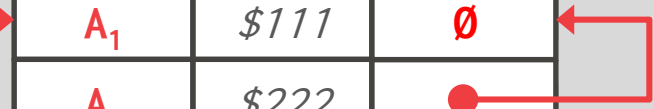
Main Table



VERSION	VALUE	POINTER
A_2	\$222	● →
B_1	\$10	

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

Time-Travel Table




VERSION	VALUE	POINTER
A_1	\$111	\emptyset
A_2	\$222	● →

Overwrite master version in the main table.
Update pointers.

TIME-TRAVEL STORAGE

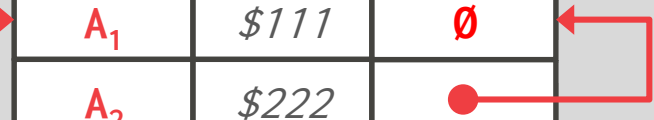
Main Table



VERSION	VALUE	POINTER
A ₃	\$333	● →
B ₁	\$10	

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

Time-Travel Table

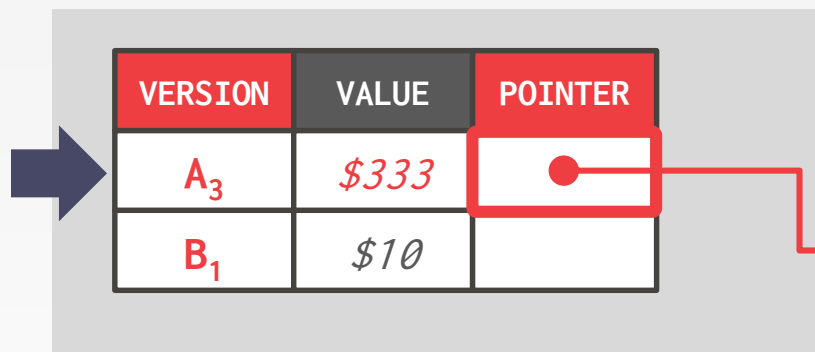


VERSION	VALUE	POINTER
A ₁	\$111	∅
A ₂	\$222	● →

Overwrite master version in the main table.
Update pointers.

TIME-TRAVEL STORAGE

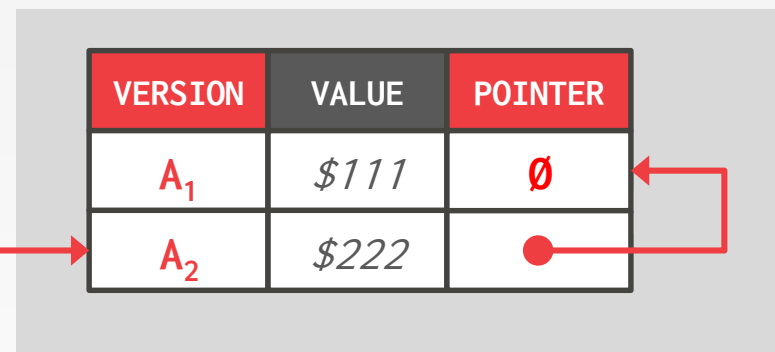
Main Table



VERSION	VALUE	POINTER
A ₃	\$333	● →
B ₁	\$10	

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

Time-Travel Table




VERSION	VALUE	POINTER
A ₁	\$111	∅ →
A ₂	\$222	● →

Overwrite master version in the main table.
Update pointers.

DELTA STORAGE

Main Table



VERSION	VALUE	POINTER
A_1	\$111	
B_1	\$10	


Delta Storage Segment



On every update, copy only the values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table



VERSION	VALUE	POINTER
A ₂	\$222	●
B ₁	\$10	


Delta Storage Segment

	DELTA	POINTER
A ₁	(VALUE→\$111)	∅

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

DELTA STORAGE

Main Table



VERSION	VALUE	POINTER
A ₂	\$222	● →
B ₁	\$10	

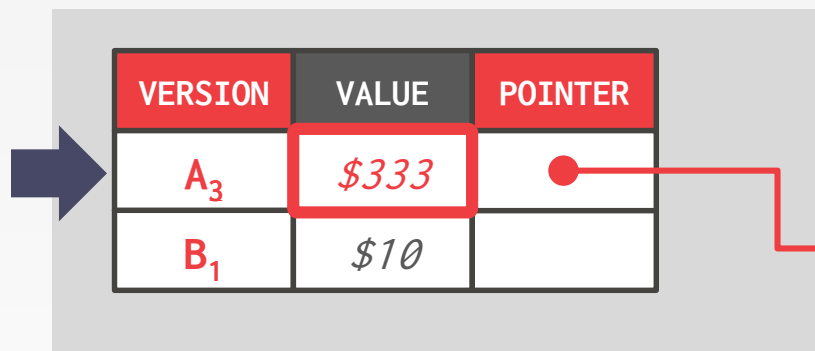
Delta Storage Segment

	DELTA	POINTER
A ₁	(VALUE→\$111)	∅
A ₂	(VALUE→\$222)	● ←

On every update, copy only the values that were modified to the delta storage and overwrite the master version.

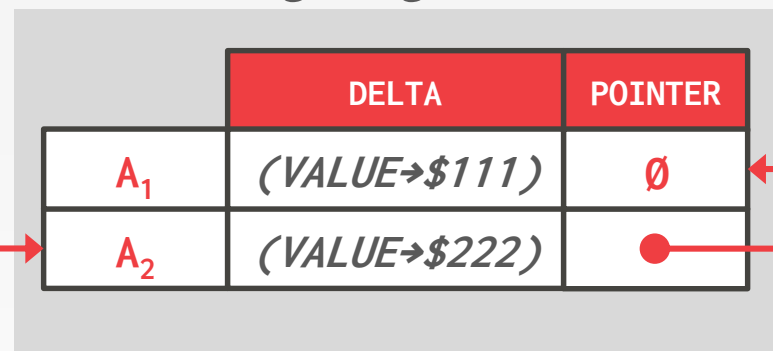
DELTA STORAGE

Main Table



VERSION	VALUE	POINTER
A ₃	\$333	●
B ₁	\$10	

Delta Storage Segment



	DELTA	POINTER
A ₁	(VALUE→\$111)	∅
A ₂	(VALUE→\$222)	●

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

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?

GARBAGE COLLECTION

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.

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:

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

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

Background Vacuuming:

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

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
A_{100}	1	9
B_{100}	1	9
B_{101}	10	20

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

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



VERSION	BEGIN	END
B_{101}	10	20

Background Vacuuming:

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

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



Dirty Page BitMap



VERSION	BEGIN	END
B_{101}	10	20

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

TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$

Vacuum



Dirty Page BitMap



VERSION	BEGIN	END
B_{101}	10	20

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

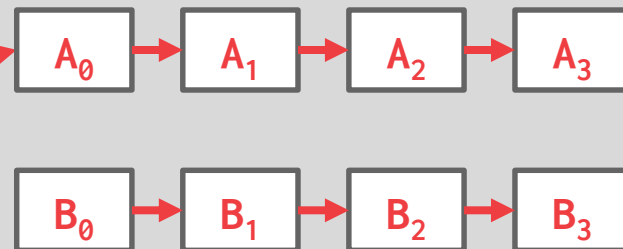
TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$



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.

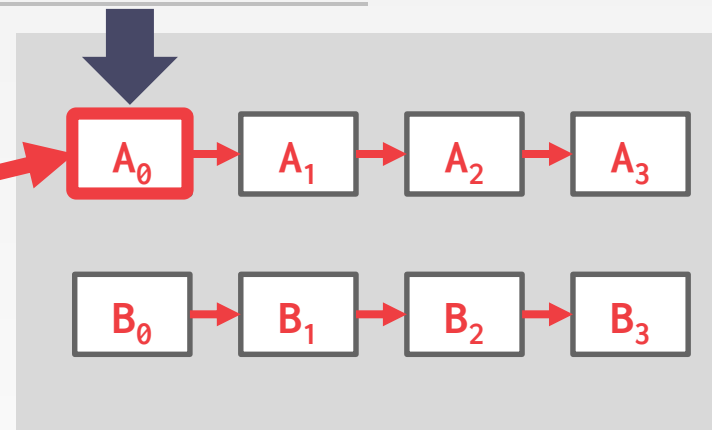
TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$



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.

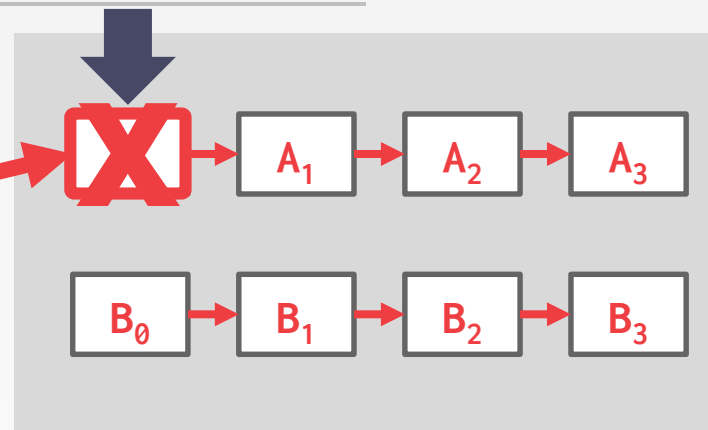
TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$



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.

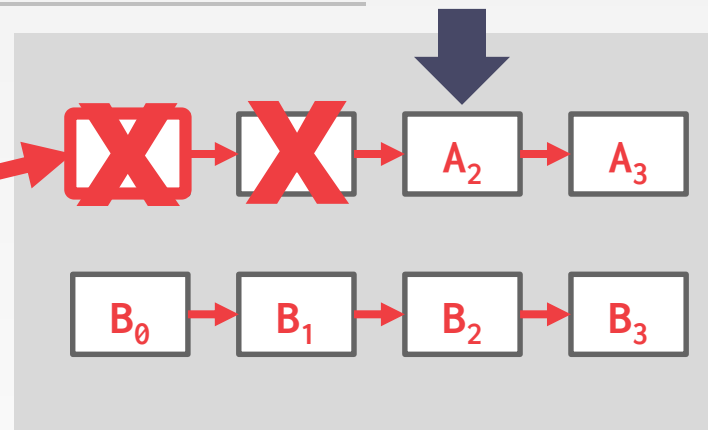
TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$



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

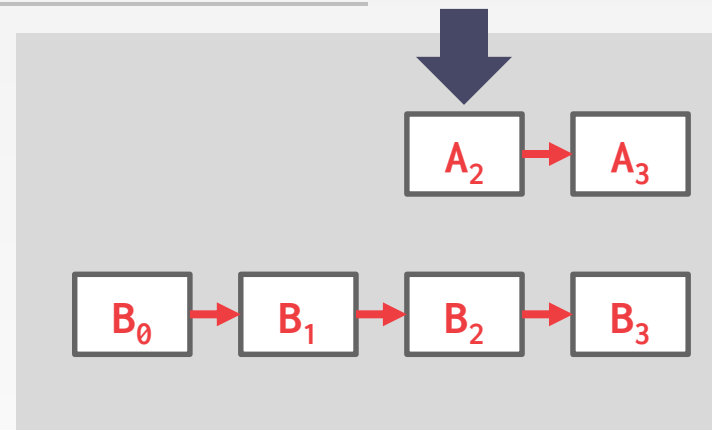
$TS(T_1)=12$

GET(A)



Thread #2

$TS(T_2)=25$



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.

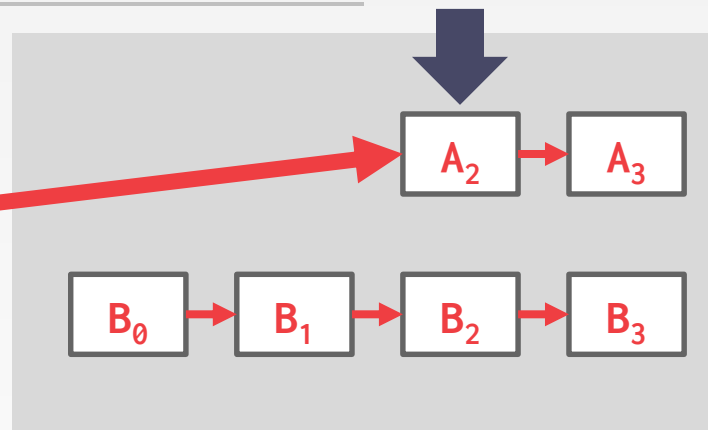
TUPLE-LEVEL GC

Thread #1

$TS(T_1)=12$

Thread #2

$TS(T_2)=25$



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.

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

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...

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...

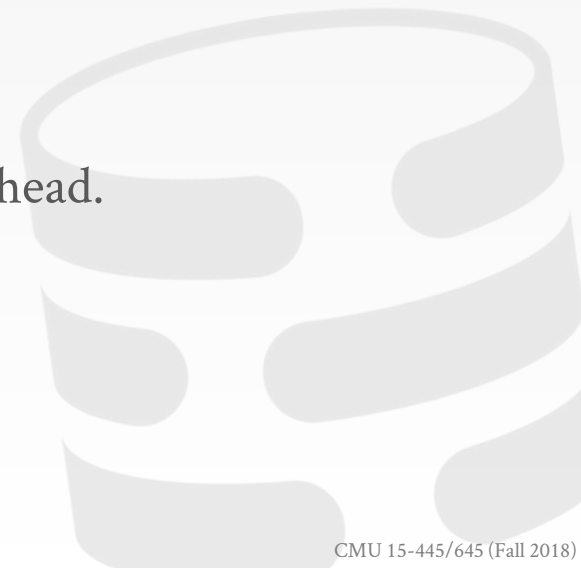
SECONDARY INDEXES

Approach #1: Logical Pointers

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

Approach #2: Physical Pointers

- Use the physical address to the version chain head.



INDEX POINTERS

GET(A)



PRIMARY INDEX



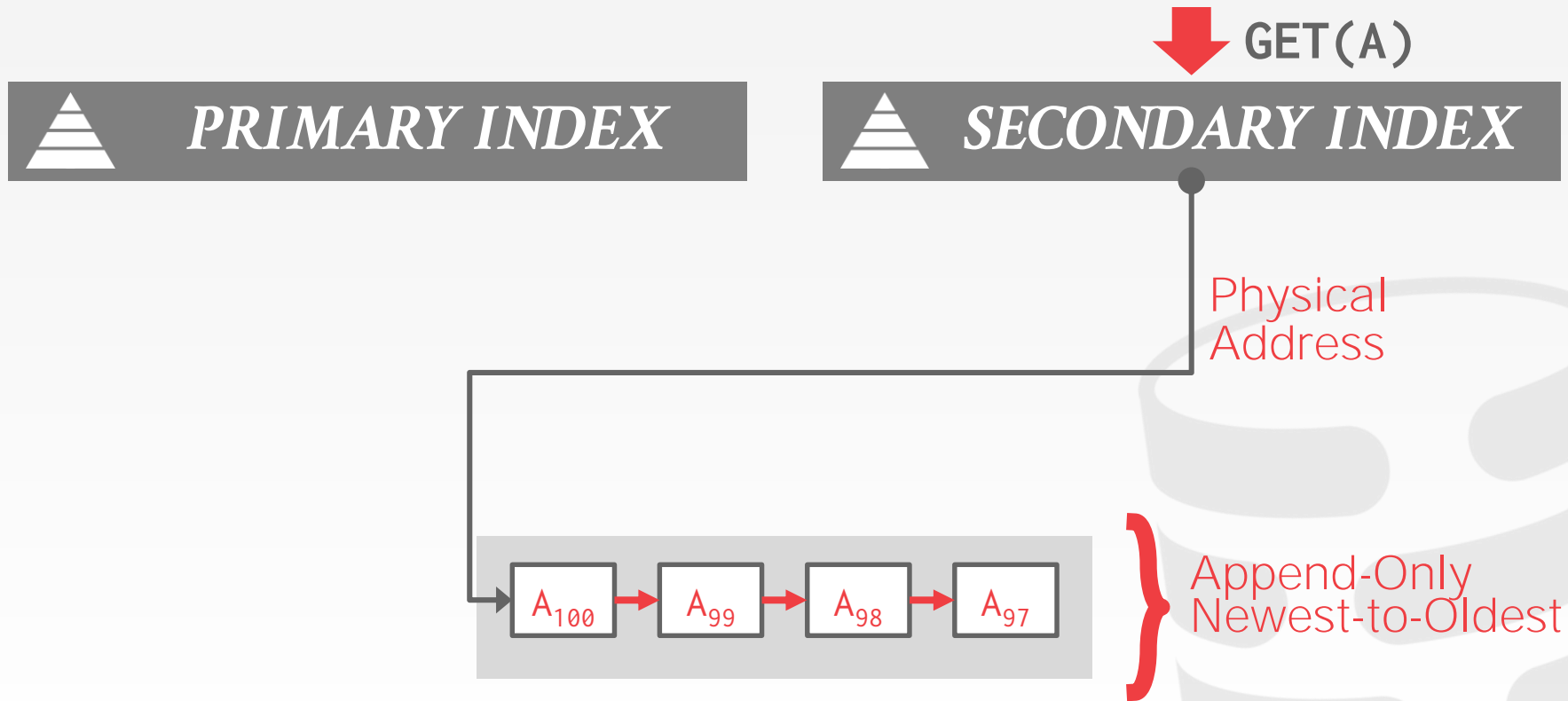
SECONDARY INDEX

Physical
Address

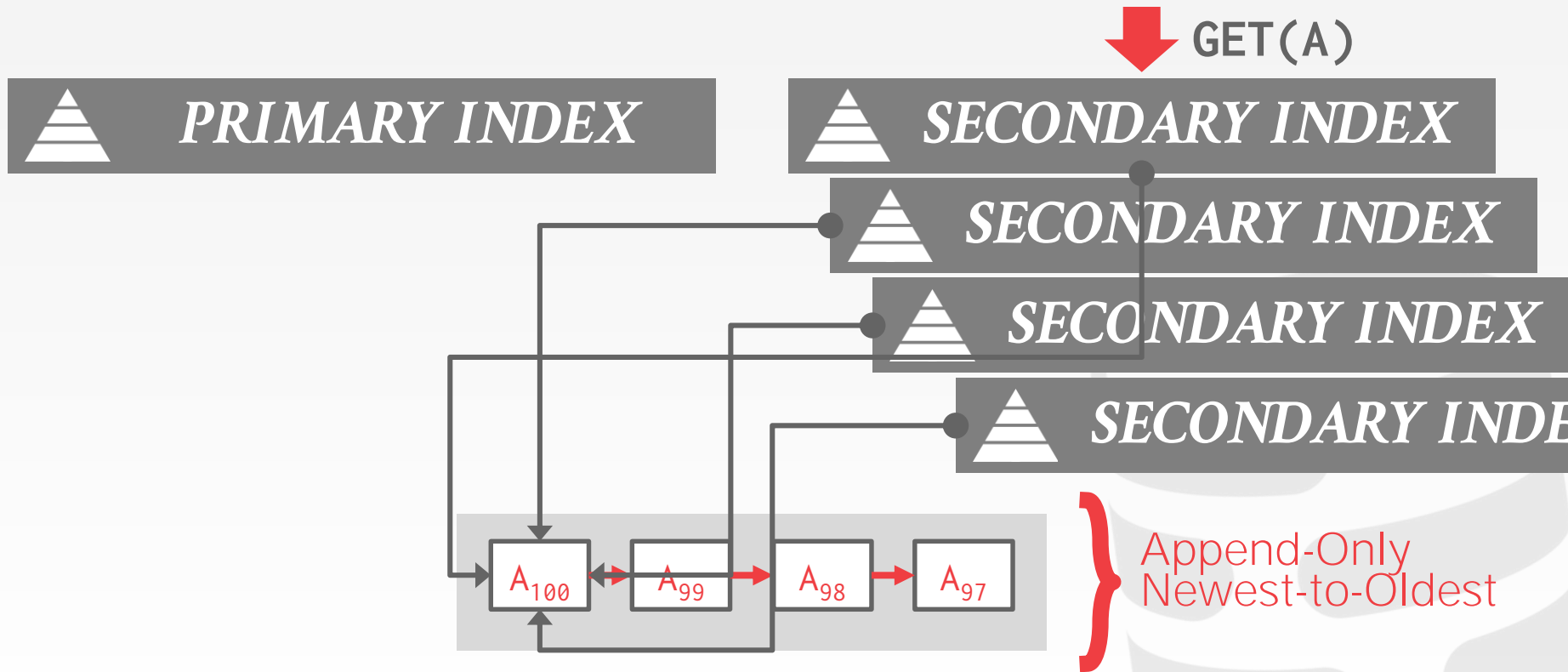


Append-Only
Newest-to-Oldest

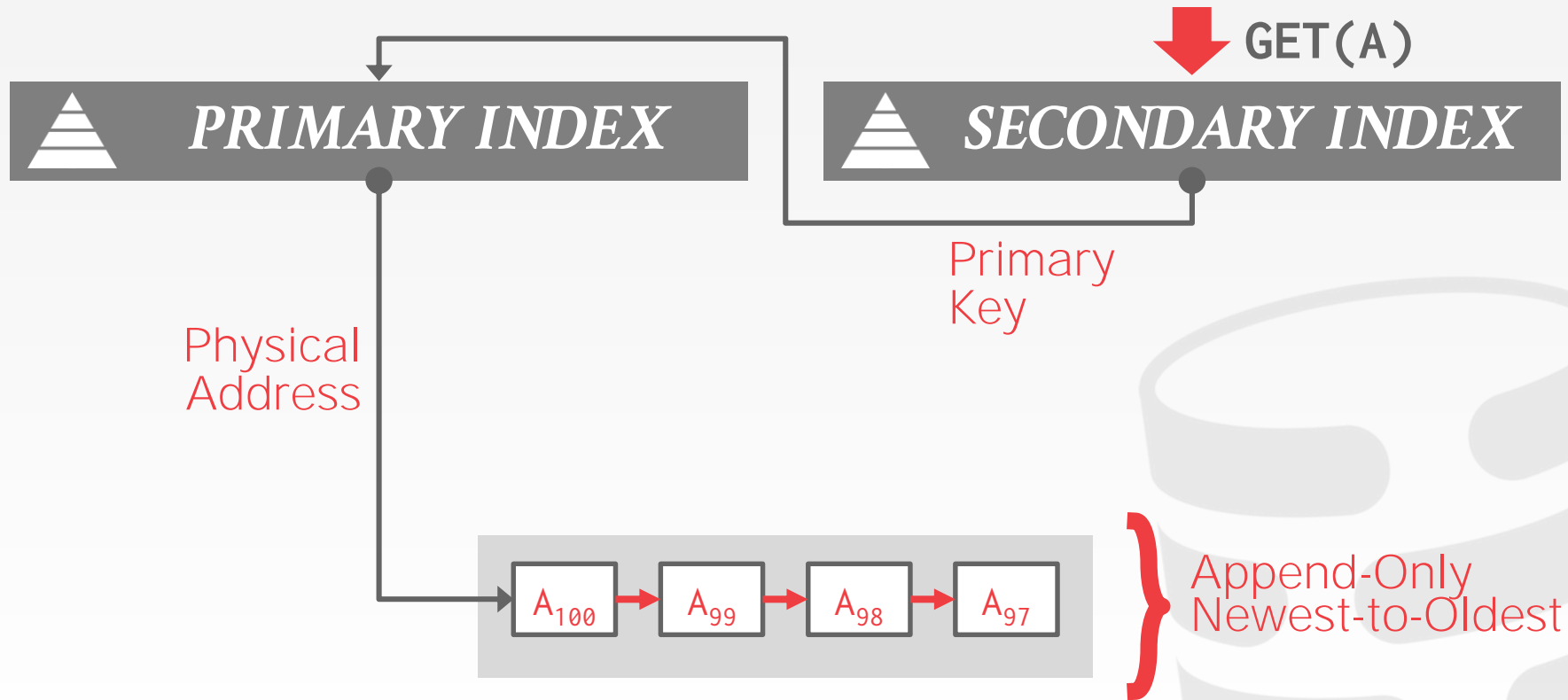
INDEX POINTERS



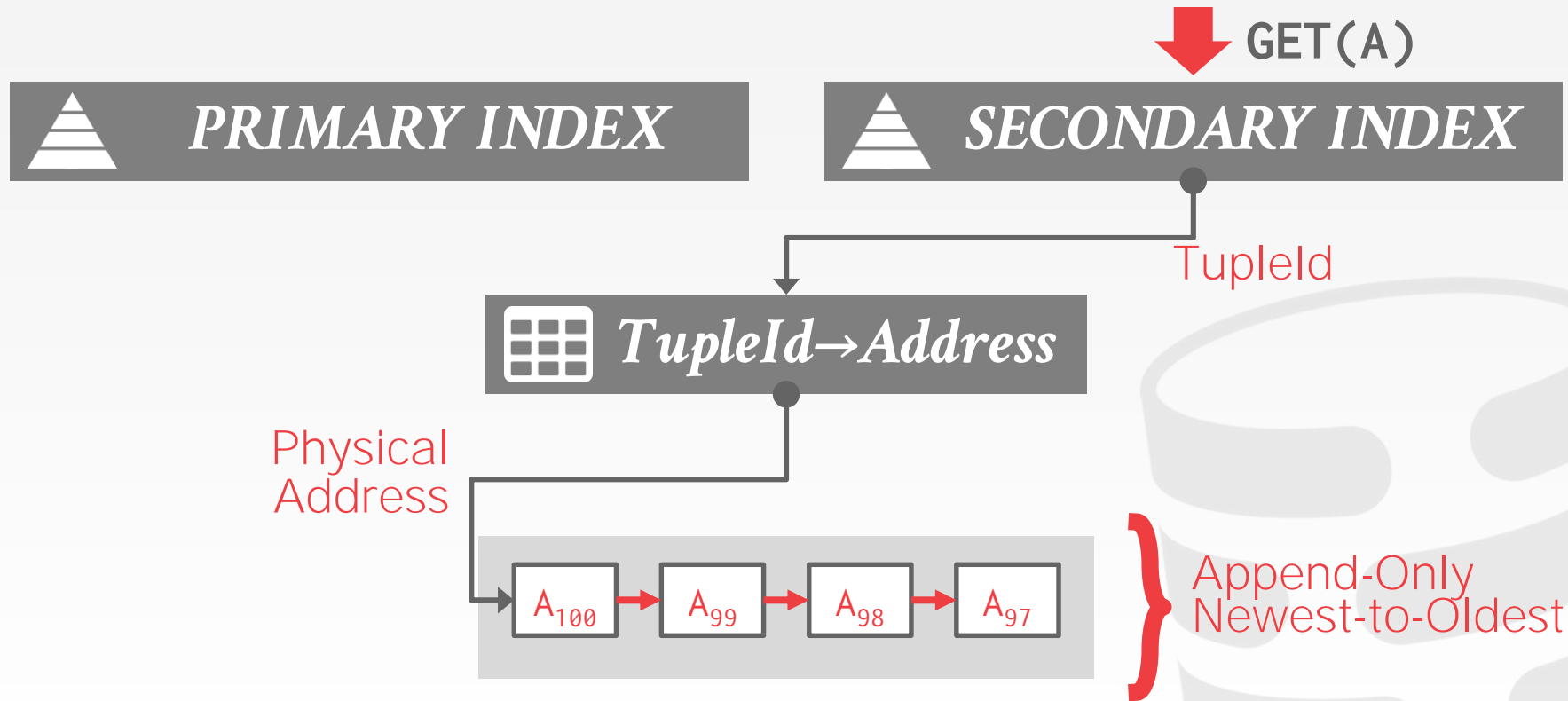
INDEX POINTERS



INDEX POINTERS



INDEX POINTERS



MVCC IMPLEMENTATIONS

	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

Logging & Recovery

