Carnegie Mellon University

Multi-Version Concurrency Control

Lecture #20

Database Systems
15-445/15-645
Fall 2017

Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

ADMINISTRIVIA

Homework #4: Monday November 13th @ 11:59pm

Project #3: Wednesday November 15th @ 11:59am

CARNEGIE MELLON DATABASE GROUP

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.

MULTI-VERSION CONCURRENCY CONTROL

Protocol was first proposed in 1978 MIT PhD dissertation.

First implementation was InterBase (now Firebird).

→ Implemented by Jim Starkey, co-founder of NuoDB.

MVCC is now used in almost every new DBMS implemented in last 10 years.

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

Schedule

TS(T_1)=1

TS(T_2)=2

TIME

T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
R(A)	
COMMIT	
	COMMIT

Database

Version	Value	Begin	End
A_0	123	0	-

MVCC – EXAMPLE #1

$TS(T_1)=1$ $TS(T_2)=2$

Schedule

T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
R(A)	
COMMIT	
	COMMIT

TIME

Database

Version	Value	Begin	End
A_0	123	0	-

MVCC – EXAMPLE #1

Schedule

TS(T_1)=1 TS(T_2)=2

TIME

T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
R(A)	
COMMIT	
	C

T_2 creates version A_1 and sets A_0 End-TS.

Database

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

MVCC – EXAMPLE #1

TIME

$TS(T_1)=1$

$TS(T_2)=2$

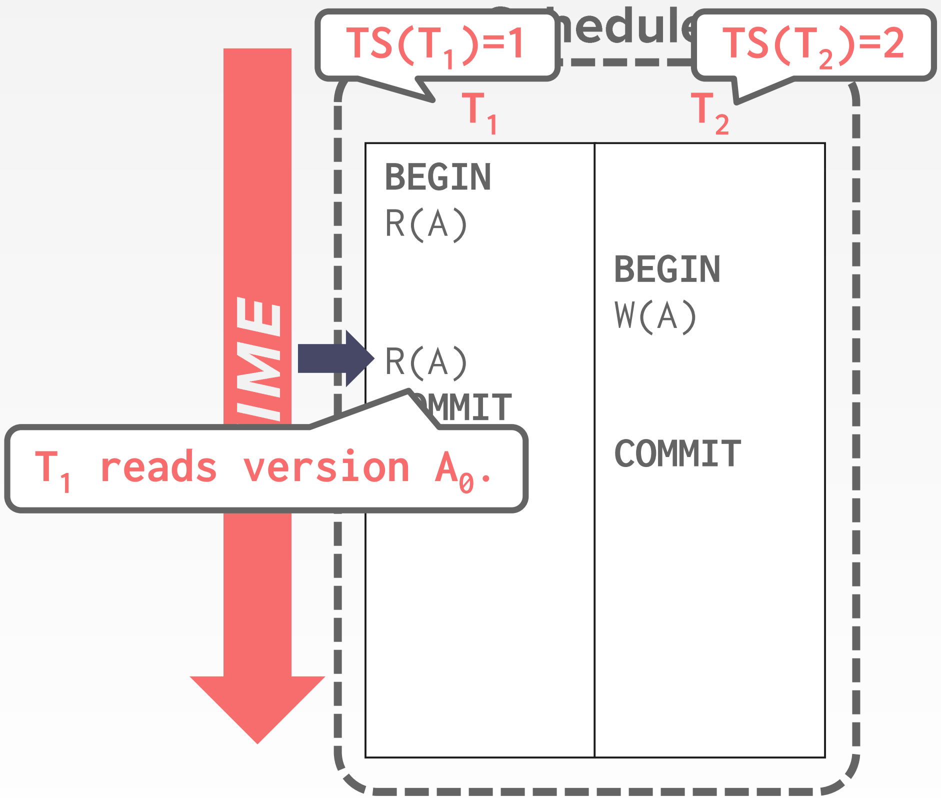
T_1	T_2
BEGIN	
R(A)	
	BEGIN
	W(A)
R(A)	
COMMIT	

T_2 creates version A_1 and sets A_0 End-TS.

Database

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

MVCC – EXAMPLE #1



Database

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

CARNEGIE MELLON DATABASE GROUP

MVCC – EXAMPLE #2

TIME

$TS(T_1)=1$ $TS(T_2)=2$

Schedule

T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

Database

Version	Value	Begin	End
A_0	123	0	

MVCC – EXAMPLE #2

TIME

Schedule

TS(T_1)=1 TS(T_2)=2

T_1	T_2
BEGIN	
R(A)	
W(A)	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

Database

Version	Value	Begin	End
A_0	123	0	
A_1	456	1	-

CARNEGIE MELLON DATABASE GROUP

MVCC – EXAMPLE #2

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

TIME

T_1	T_2
BEGIN	
R(A)	
W(A)	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

Database

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

MVCC – EXAMPLE #2

TIME

Schedule

$TS(T_1)=1$ $TS(T_2)=2$

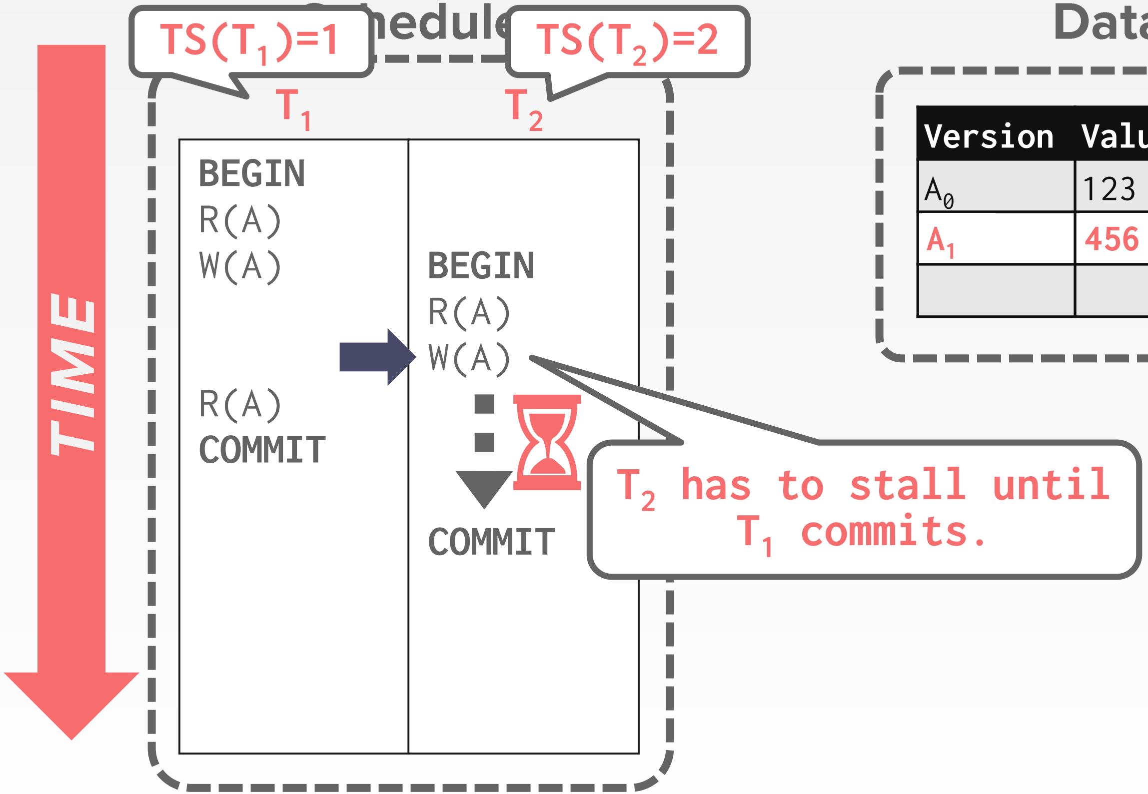
T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

T_2 reads version A_0 because T_1 has not committed yet.

Database

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

MVCC – EXAMPLE #2

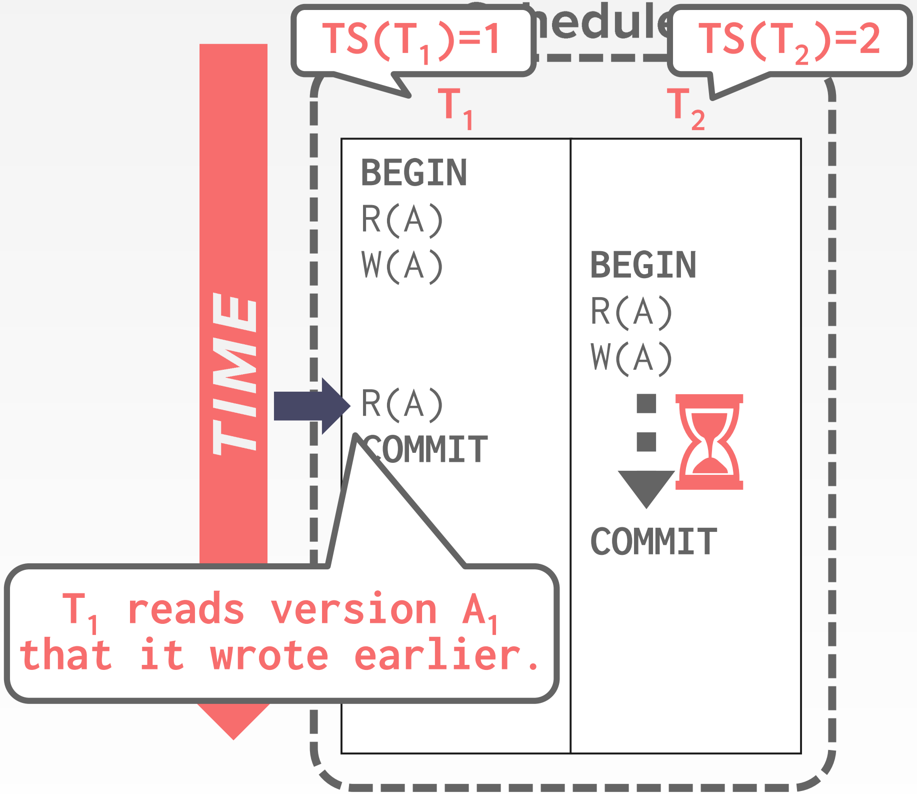


Database

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

CARNEGIE MELLON DATABASE GROUP

MVCC – EXAMPLE #2



Database

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

MVCC – EXAMPLE #2

Schedule

TS(T_1)=1

TS(T_2)=2

T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(A)
	W(A)
R(A)	
COMMIT	
	COMMIT

Now T_2 can create the new version.

TIME

Database

Version	Value	Begin	End
A_0	123	0	1
A_1	456	1	2
A_2	789	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.

Couchbase
Clustrix
PostgreSQL
RAVENDB
NUODB
Peloton
SAP HANA
IMDB
HyPer
ORACLE
FOUNDATIONDB
CUBRID
ArangoDB
Microsoft SQL Server Hekaton
MySQL
MariaDB
MEMSQL
OrientDB
APACHE HBASE
RethinkDB
Cockroach LABS
HYRISE

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

Main Table

	KEY	VALUE	POINTER
A_x	XXX	$111	●
A_{x+1}	XXX	$222	Ø
B_x	YYY	$10	Ø

All of the physical versions of a logical tuple are stored in the same table space

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

APPEND-ONLY STORAGE

Main Table

	KEY	VALUE	POINTER
A_x	XXX	\$111	●
A_{x+1}	XXX	\$222	Ø
B_x	YYY	\$10	Ø
A_{x+2}	XXX	\$333	Ø

All of the physical versions of a logical tuple are stored in the same table space

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

APPEND-ONLY STORAGE

Main Table

	KEY	*VALUE*	POINTER
A_x	*XXX*	*$111*	●
A_{x+1}	*XXX*	*$222*	●
B_x	*YYY*	*$10*	Ø
A_{x+2}	*XXX*	*$333*	Ø

All of the physical versions of a logical tuple are stored in the same table space

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

CARNEGIE MELLON DATABASE GROUP

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

	KEY	VALUE	POINTER
A_2	XXX	\$222	•
B_1	YYY	\$10	

Time-Travel Table

	KEY	VALUE	POINTER
A_1	XXX	\$111	Ø

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

CARNEGIE MELLON DATABASE GROUP

TIME-TRAVEL STORAGE

Main Table

	KEY	VALUE	POINTER
A_2	XXX	$222	•
B_1	YYY	$10	

Time-Travel Table

	KEY	VALUE	POINTER
A_1	XXX	$111	Ø
A_2	XXX	$222	•

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

TIME-TRAVEL STORAGE

Main Table

	KEY	VALUE	POINTER
A_3	XXX	\$333	
B_1	YYY	\$10	

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

Time-Travel Table

	KEY	VALUE	POINTER
A_1	XXX	\$111	Ø
A_2	XXX	\$222	

Overwrite master version in the main table. Update pointers.

CARNEGIE MELLON DATABASE GROUP

DELTA STORAGE

Main Table

	KEY	*VALUE*	POINTER
A_1	*XXX*	*$111*	
B_1	*YYY*	*$10*	

Delta Storage Segment

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

CARNEGIE MELLON
DATABASE GROUP

DELTA STORAGE

Main Table

	KEY	VALUE	POINTER
A_1	XXX	$\boxed{\$111}$	
B_1	YYY	$10	

Delta Storage Segment

	DELTA	POINTER
A_1	(VALUE→$111)	Ø

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

DELTA STORAGE

Main Table

	KEY	*VALUE*	POINTER
A_2	*XXX*	*$222*	
B_1	*YYY*	*$10*	

Delta Storage Segment

	DELTA	POINTER
A_1	*(VALUE→$111)*	Ø
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

	KEY	VALUE	POINTER
A_3	XXX	$333	●
B_1	YYY	$10	

Delta Storage Segment

	DELTA	POINTER
A_1	(VALUE→$111)	Ø
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?

CARNEGIE MELLON
DATABASE GROUP

GARBAGE COLLECTION

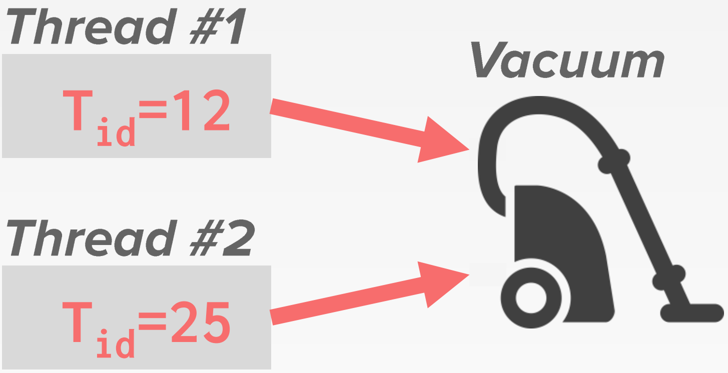
Approach #1: Tuple-level

→ 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.

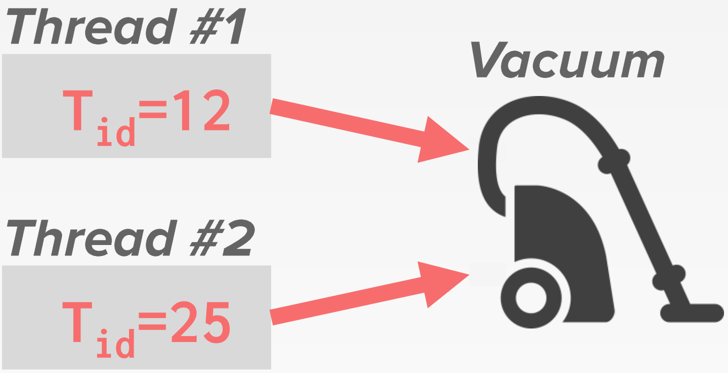
TUPLE-LEVEL GC



	TXN-ID	BEGIN	END
A_x	0	1	9
B_x	0	1	9
B_{x+1}	0	10	20

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

TUPLE-LEVEL GC



	TXN-ID	BEGIN	END
A_x	0	1	9
B_x	0	1	9
B_{x+1}	0	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

T_{id}=12

Thread #2

T_{id}=25

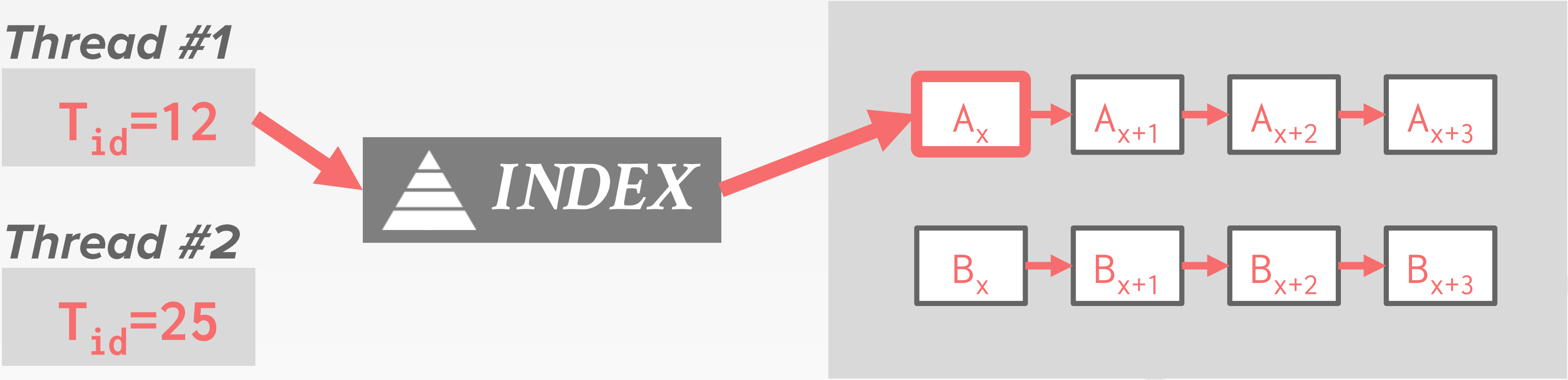
Vacuum

Dirty?	TXN-ID	BEGIN	END
B_{x+1}	0	10	20

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

CARNEGIE MELLON DATABASE GROUP

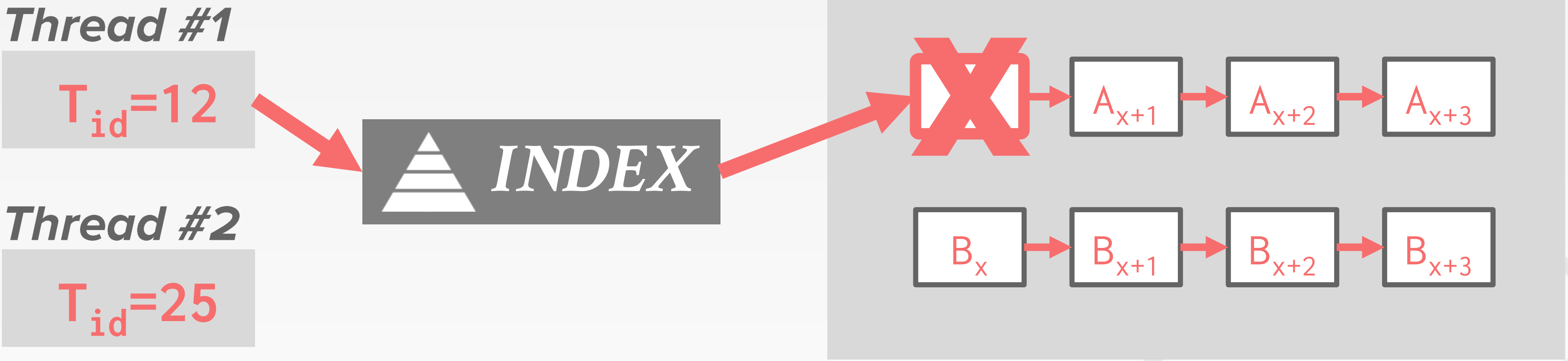
TUPLE-LEVEL GC



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



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

PKey indexes always 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

PKey in
chain he
→ How c
index
new v
→ If a txn
this is
INSER

Second
complic

UBER Engineering

JOIN THE TEAM

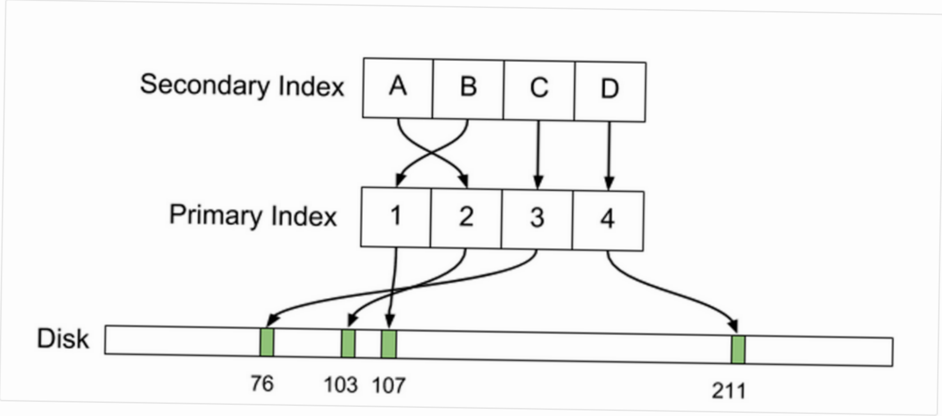
MEET THE PEOPLE

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE



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

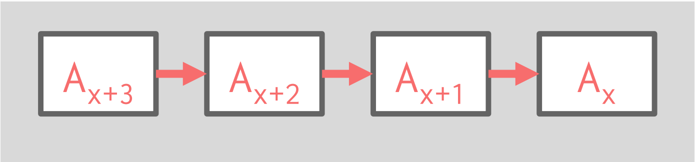
Approach #2: Physical Pointers

→ Use the physical address to the version chain head.

INDEX POINTERS

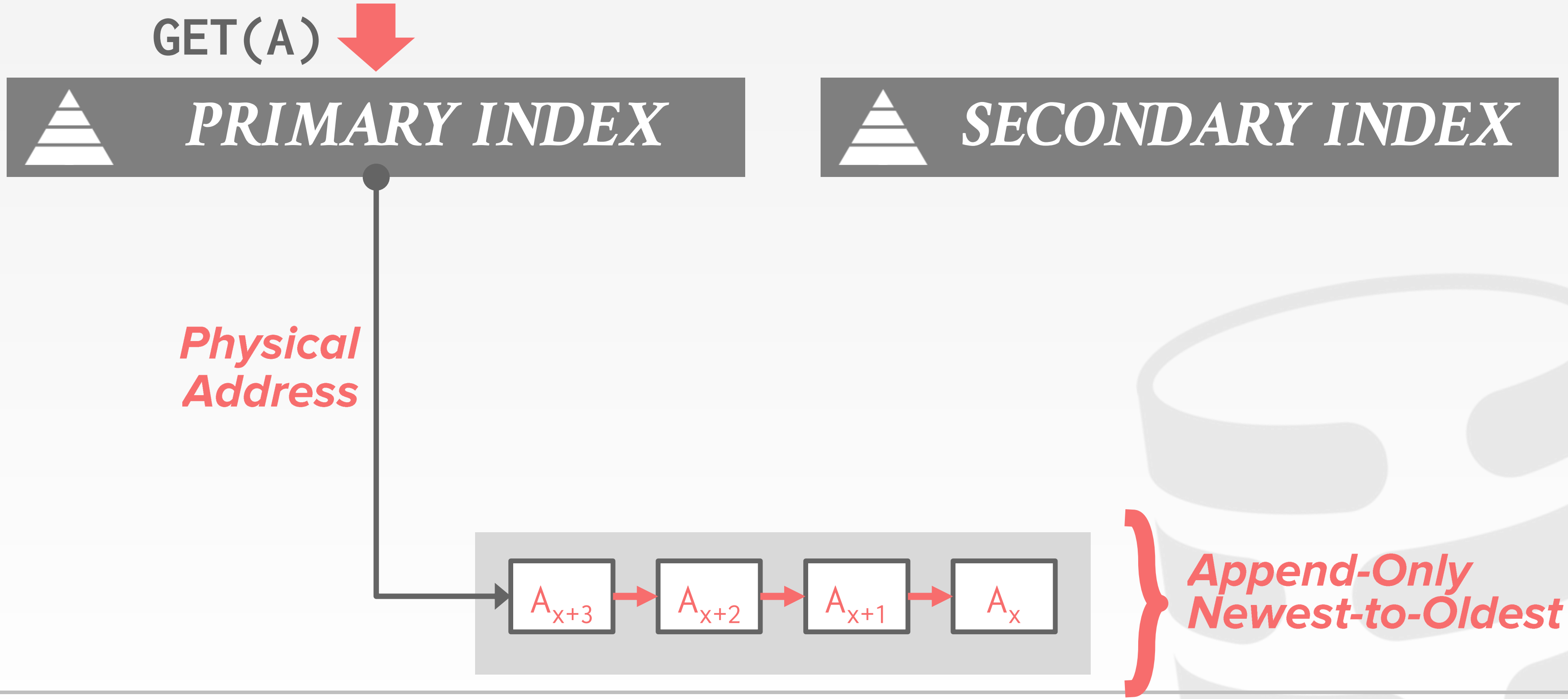
PRIMARY INDEX

SECONDARY INDEX

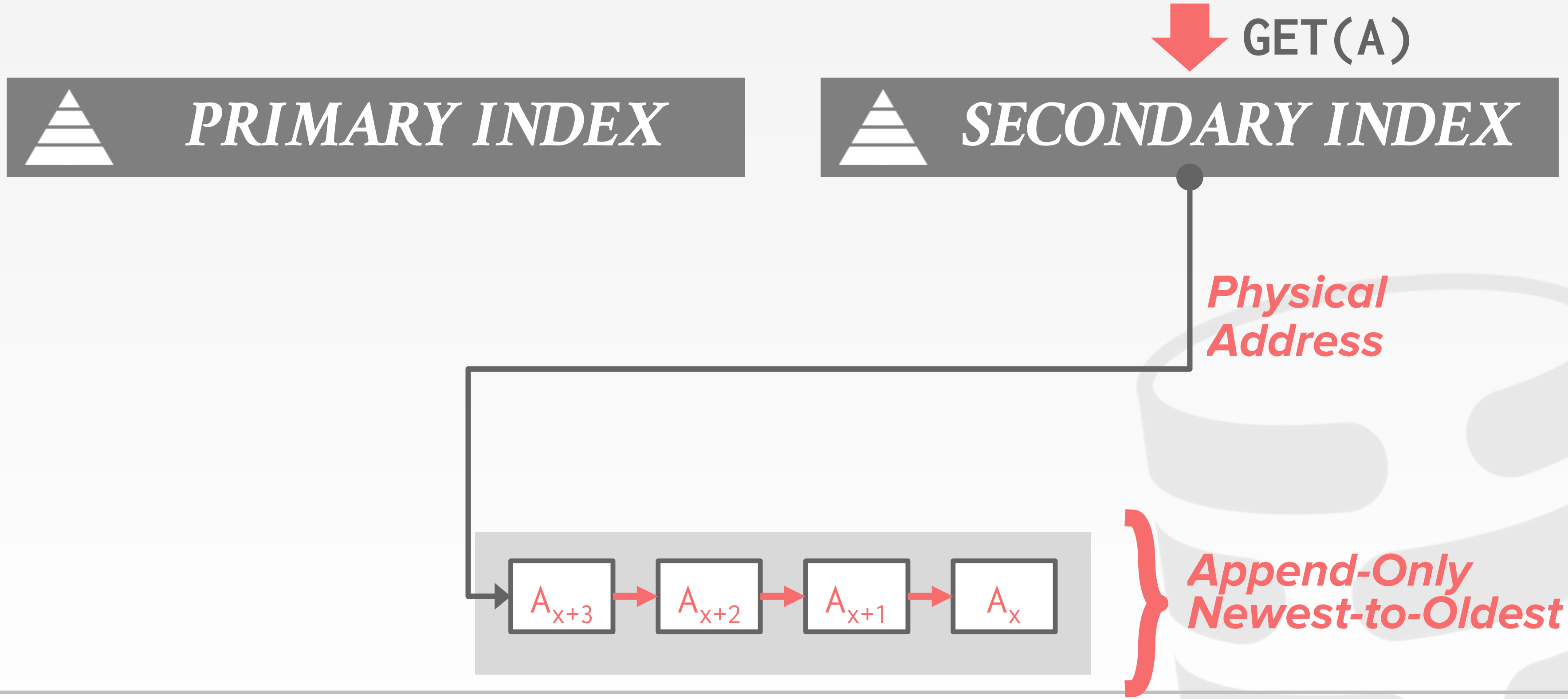


Append-Only
Newest-to-Oldest

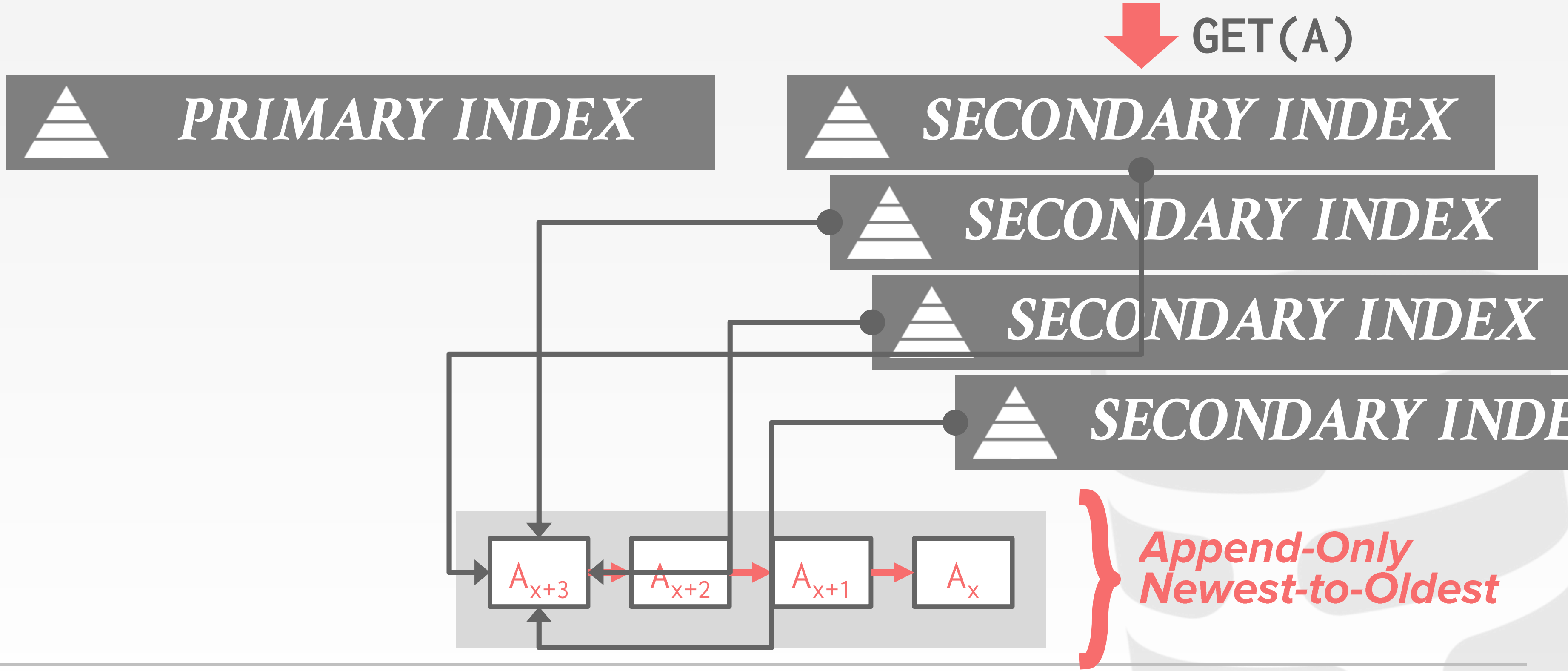
INDEX POINTERS



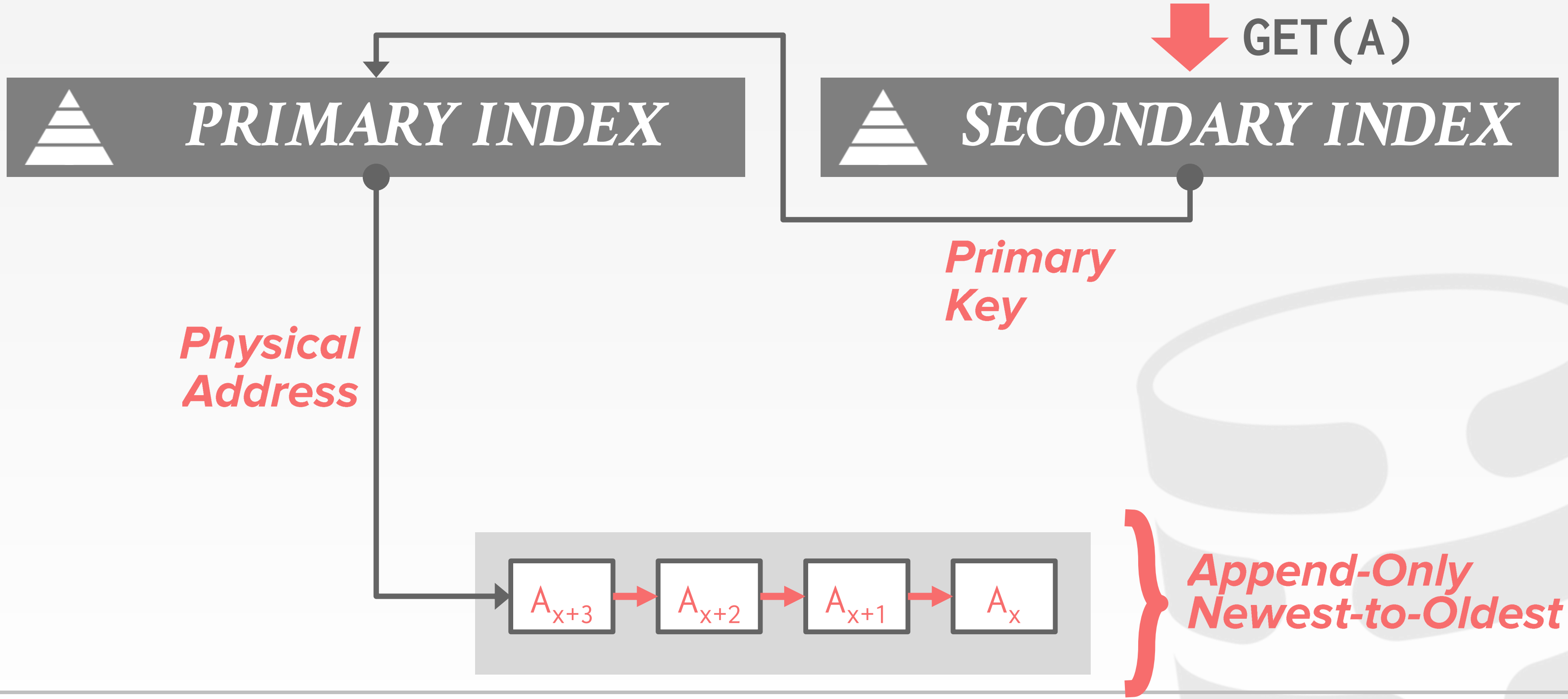
INDEX POINTERS



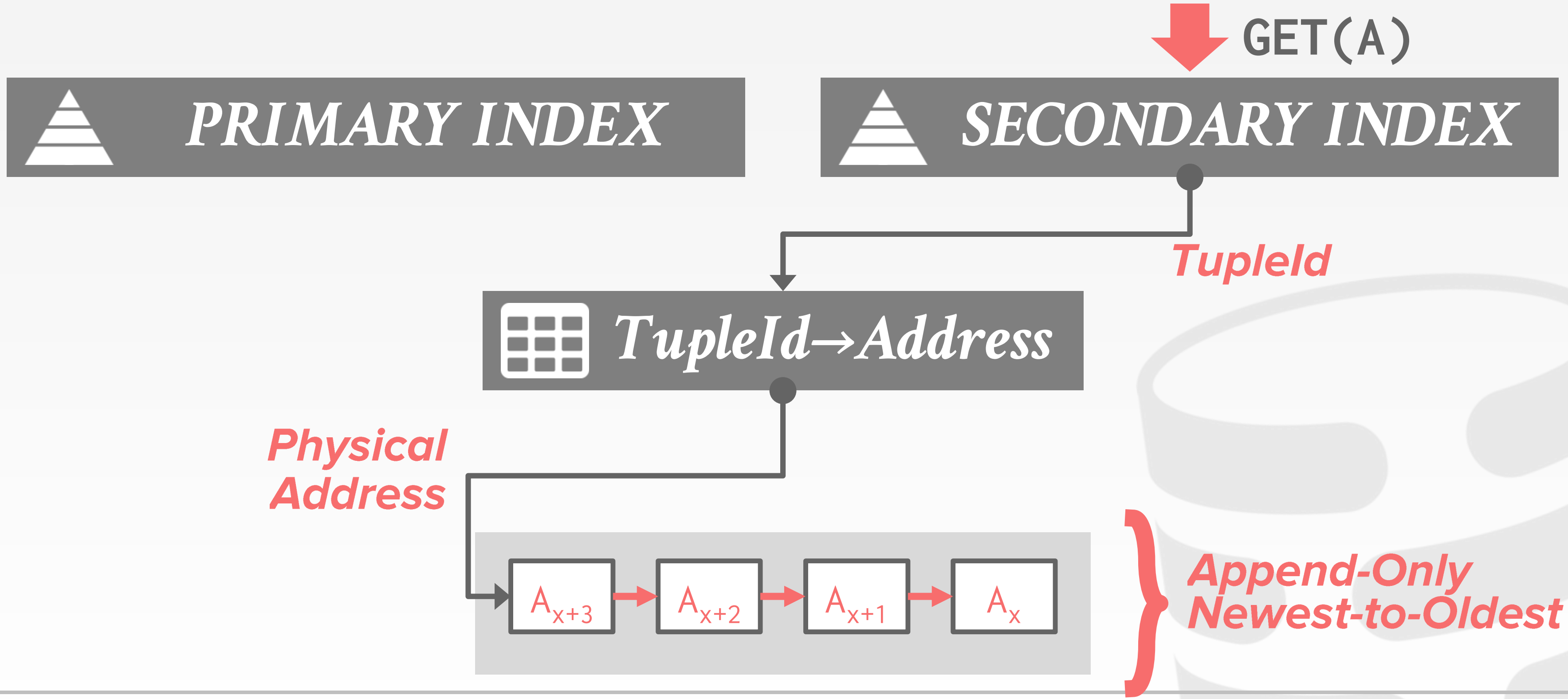
INDEX POINTERS



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 (NoSQL) systems that do not support multi-statement txns use it.

NEXT CLASS

Logging & Recovery