Intro to Database Systems (15-445/645)

18 Multi-Version Concurrency Control

Carnegie Mellon University

SPRING 2023

Charlie Garrod

ADMINISTRIVIA

Project 3 ongoing
→ Due Sunday, April 9th at 11:59 p.m.

Homework 4 released today
→ Due Friday, April 7th at 11:59 p.m.

Final exam Monday, May 1st, 8:30 – 11:30 a.m.

LAST TIME: TIMESTAMP ORDERING

Basic timestamp ordering

Optimistic concurrency control

The phantom problem
→ Re-execute scans
→ Predicate locking
→ Index locking schemes
    - Key-value locks
    - Gap locks
    - Key-range locks
    - Hierarchical locking

TODAY'S PLAN

Isolation levels

Multi-version concurrency control

WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

We may want to use a weaker level of consistency to improve scalability.

ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
→ Dirty Reads
→ Unrepeatable Reads
→ Phantom Reads

ISOLATION LEVELS

Isolation (Low→High)

SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS: Phantoms may happen.

READ COMMITTED: Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED: All of them may happen.

Isolation (Low→High)

SERIALIZ

no dirty re

REPEATAB

READ COMM

reads may

READ UNCO

United States Department of Justice

Offices of the United States Attorneys

THE UNITED STATES ATTORNEY'S OFFICE

SOUTHERN DISTRICT *of* NEW YORK

Search

SEARCH

HOME | ABOUT | PRIORITIES | NEWS | RESOURCES | PROGRAMS | EMPLOYMENT | CONTACT

U.S. Attorneys » Southern District of New York » News » Press Releases

SHARE

Department of Justice

U.S. Attorney's Office

Southern District of New York

FOR IMMEDIATE RELEASE

Monday, November 7, 2022

U.S. Attorney Announces Historic $3.36 Billion Cryptocurrency Seizure And Conviction In Connection With Silk Road Dark Web Fraud

In November 2021, Law Enforcement Seized Over 50,676 Bitcoin Hidden in Devices in Defendant JAMES ZHONG's Home; ZHONG Has Now Pled Guilty to Unlawfully Obtaining that Bitcoin From the Silk Road Dark Web in 2012

Damian Williams, the United States Attorney for the Southern District of New York, and Tyler Hatcher, the Special Agent in Charge of the Internal Revenue Service, Criminal Investigation, Los Angeles Field Office ("IRS-CI"), announced today that JAMES ZHONG pled guilty to committing wire fraud in September 2012 when he unlawfully obtained over 50,000 Bitcoin from the Silk Road dark web internet marketplace. ZHONG pled guilty on Friday, November 4, 2022, before United States District Judge Paul G. Gardephe.

On November 9, 2021, pursuant to a judicially authorized premises search warrant of ZHONG's Gainesville, Georgia, house, law enforcement seized approximately 50,676.17851897 Bitcoin, then valued at over $3.36 billion. This seizure was then the largest cryptocurrency seizure in the history of the U.S. Department of Justice and today remains the Department's second largest financial seizure ever. The Government is seeking to forfeit, collectively: approximately 51,680.32473733 Bitcoin; ZHONG's 80% interest in RE&D Investments, LLC, a Memphis-based company with substantial real estate holdings; $661,900 in cash seized from ZHONG's home; and various metals also seized from ZHONG's home.

Click here to report information on Amazon warehouses.

ISOLATION LEVELS

	Dirty Read	*Unrepeatable Read*	*Phantom*
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe

ISOLATION LEVELS

SERIALIZABLE: Obtain all locks first; plus index locks, plus strict 2PL.

REPEATABLE READS: Same as above, but no index locks.

READ COMMITTED: Same as above, but **S** locks are released immediately.

READ UNCOMMITTED: Same as above but allows dirty reads (no **S** locks).

SQL-92 ISOLATION LEVELS

You set a txn's isolation level before you execute any queries in that txn.

```
SET TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

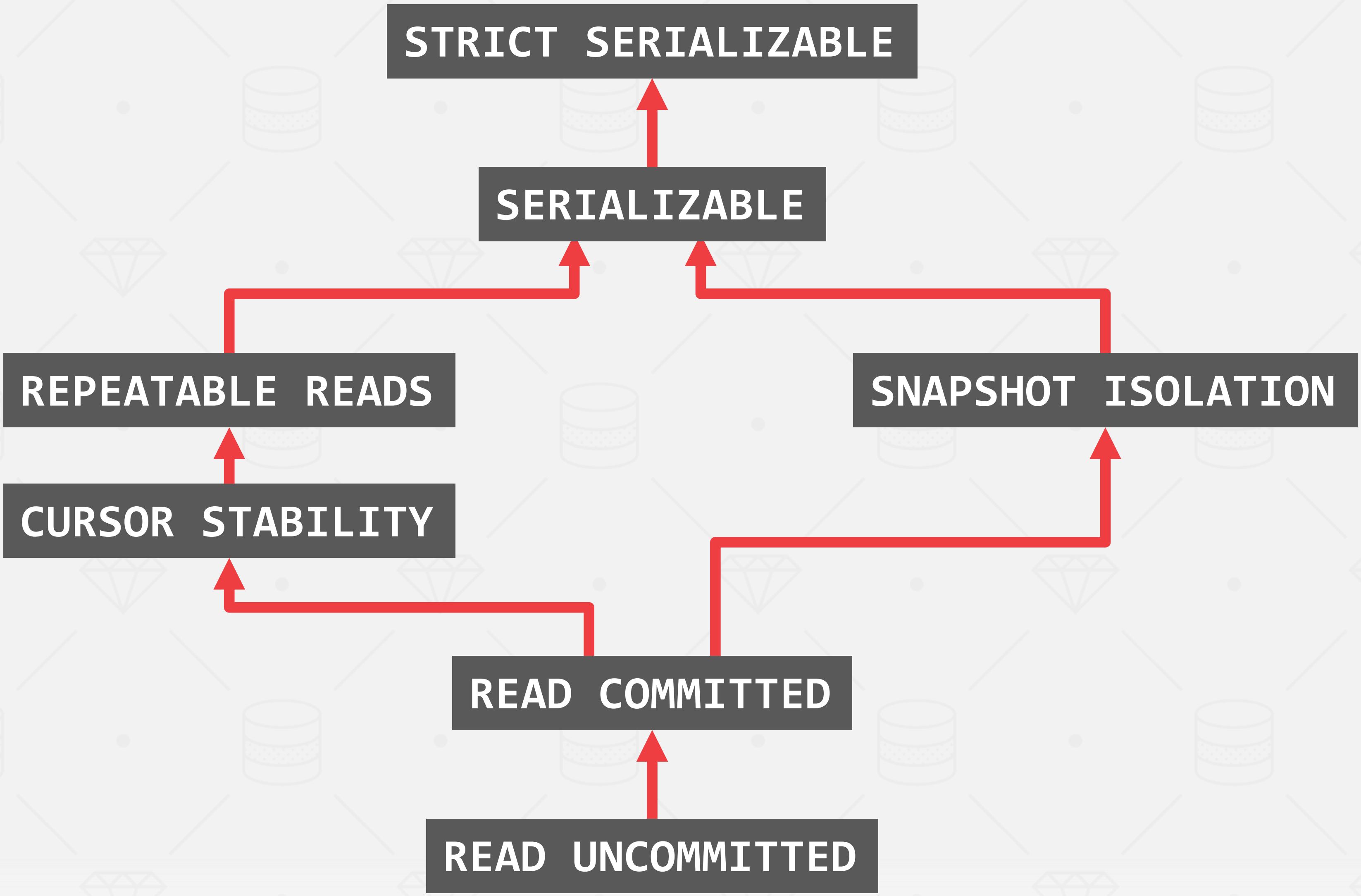
```
BEGIN TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

Not all DBMS support all isolation levels in all execution scenarios
→ Replicated Environments

The default depends on implementation…

ISOLATION LEVELS

	Default	*Maximum*
Actian Ingres	SERIALIZABLE	SERIALIZABLE
IBM DB2	CURSOR STABILITY	SERIALIZABLE
CockroachDB	SERIALIZABLE	SERIALIZABLE
Google Spanner	STRICT SERIALIZABLE	STRICT SERIALIZABLE
MSFT SQL Server	READ COMMITTED	SERIALIZABLE
MySQL	REPEATABLE READS	SERIALIZABLE
Oracle	READ COMMITTED	SNAPSHOT ISOLATION
PostgreSQL	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
VoltDB	SERIALIZABLE	SERIALIZABLE
YugaByte	SNAPSHOT ISOLATION	SERIALIZABLE


STRICT SERIALIZABLE
SERIALIZABLE
REPEATABLE READS
SNAPSHOT ISOLATION
CURSOR STABILITY
READ COMMITTED
READ UNCOMMITTED

DATABASE ADMIN SURVEY

What isolation level do transactions execute at on this DBMS?

SQL-92 ACCESS MODES

You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.

Only two possible modes:
- → READ WRITE (Default)
- → READ ONLY

```
SET TRANSACTION <access-mode>;
```

```
BEGIN TRANSACTION <access-mode>;
```

Not all DBMSs will optimize execution if you set a txn to in READ ONLY mode.

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.

Rdb/VMS

Oracle Rdb
the Database for HP OpenVMS Platform

MULTI-VERSION CONCURRENCY CONTROL

Writers do not block readers
Readers do not block writers

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

→ Use timestamps to determine visibility
→ Txn T can see values written by txns *that committed* before T's timestamp

Easily support time-travel queries

MVCC – EXAMPLE #1

Schedule

TS(T_1)=1

TS(T_2)=2

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

TIME

T_1 reads version A_0.

T_2 creates version A_1 and sets A_0 End-TS.

Database

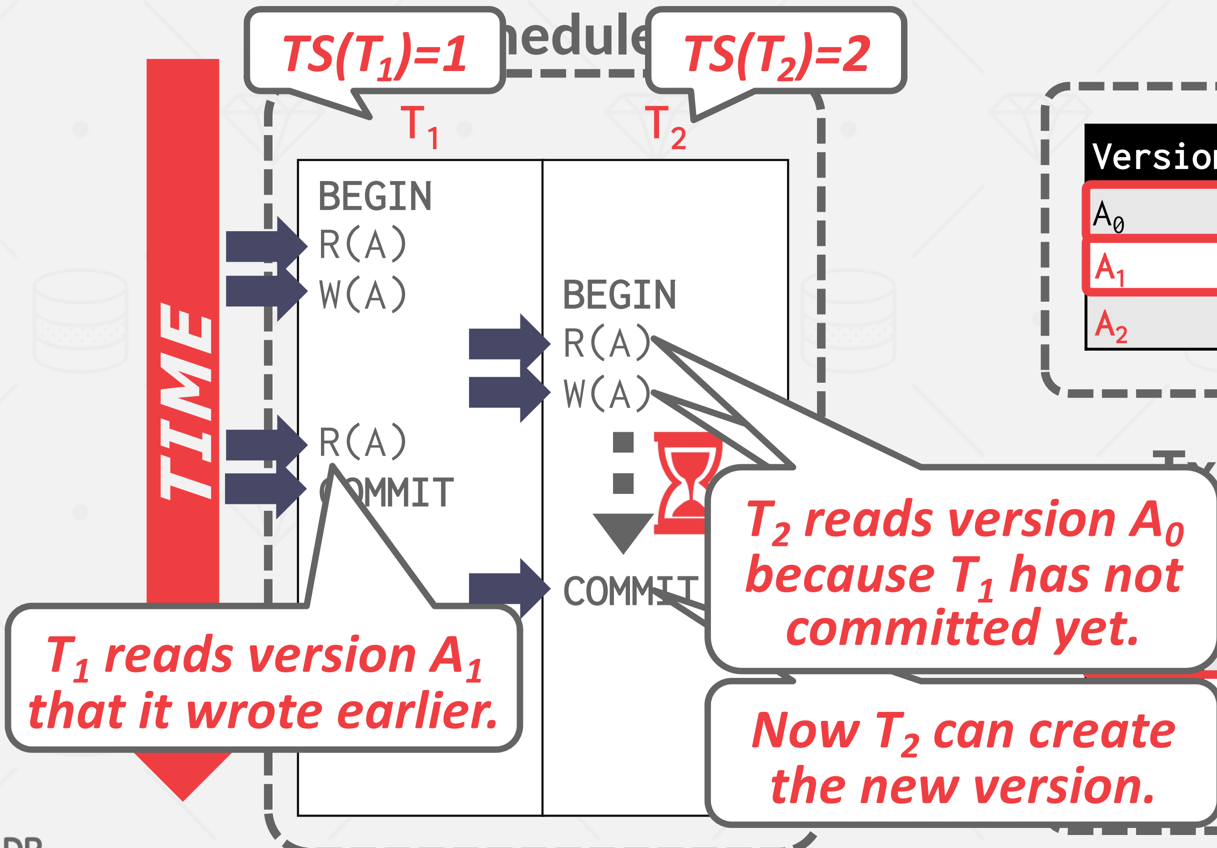
Version	Value	Begin	End
A_0	123	0	2
A_1	456	2	-

Txn Status Table

TxnId	Timestamp	Status
T_1	1	Active
T_2	2	Active

MVCC – EXAMPLE #2

Schedule



Database

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

Txn Status Table

Timestamp	Status
1	Committed
2	Active

SNAPSHOT ISOLATION (SI)

When a txn starts, it sees a consistent snapshot of the database

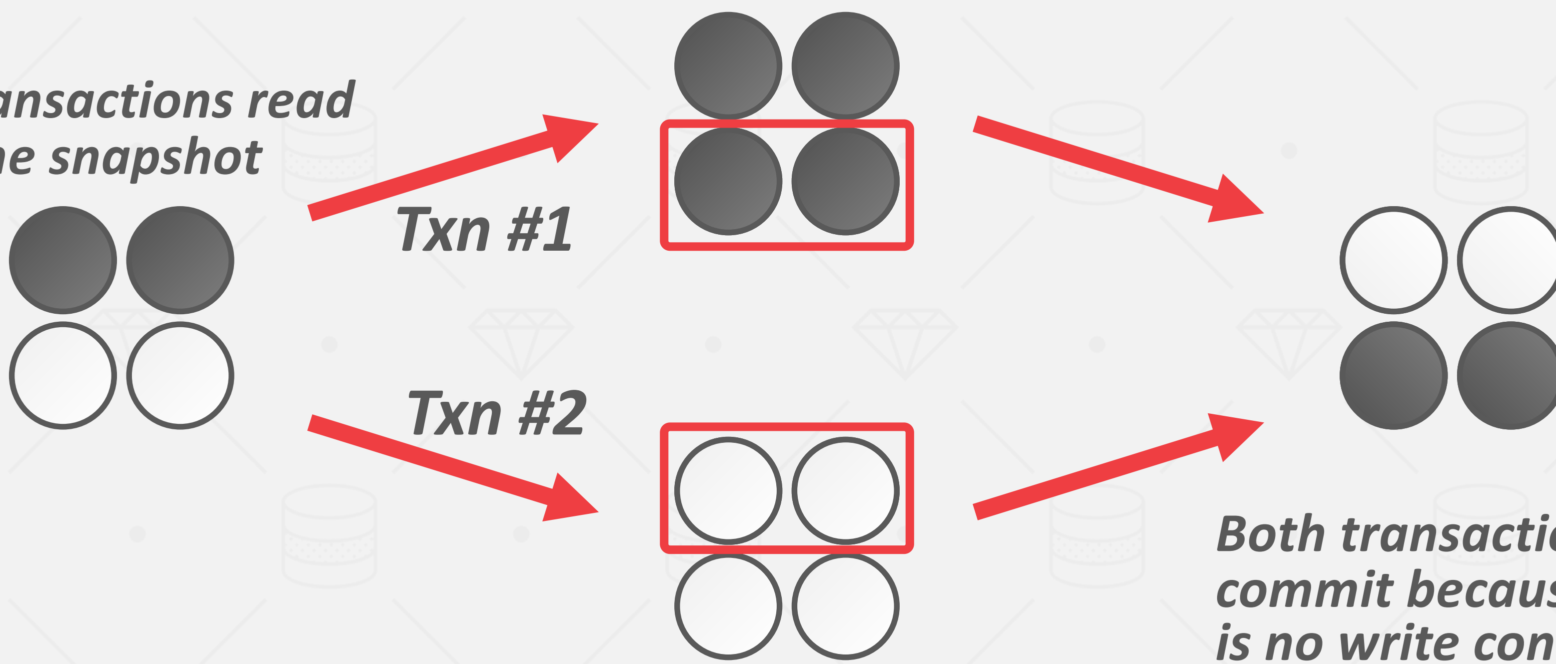
→ Can see all data committed before that txn started
→ No torn writes from active txns.
→ If two txns update the same object, then first writer wins.

Snapshot isolation is not the same as serializable.
It is susceptible to the **Write Skew Anomaly**.

WRITE SKEW ANOMALY

A possible outcome with MVCC:

Both transactions read the same snapshot



Both transactions commit because there is no write conflict

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
Deletes

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 write a 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

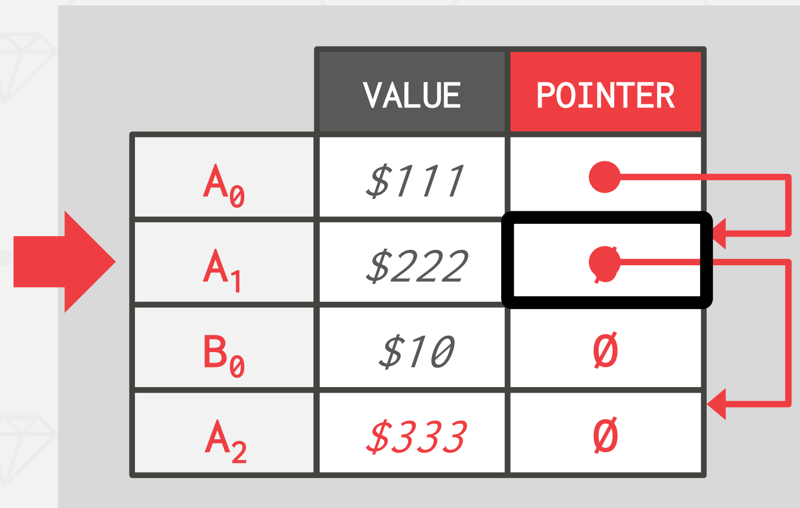
→ The original values of the modified attributes are copied into a separate delta record space.

APPEND-ONLY STORAGE

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

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

Main Table



VERSION CHAIN ORDERING

Approach #1: Oldest-to-Newest (O2N)

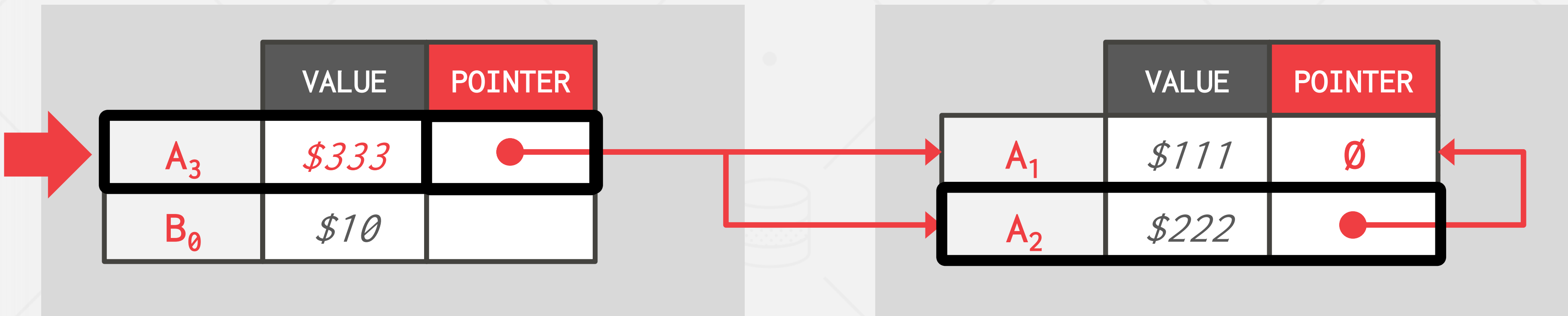
→ Append new version to end of the chain.
→ Must traverse chain on look-ups.

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

→ Must update index pointers for every new version.
→ Do not have to traverse chain on look-ups.

TIME-TRAVEL STORAGE

Main Table / *Time-Travel Table*

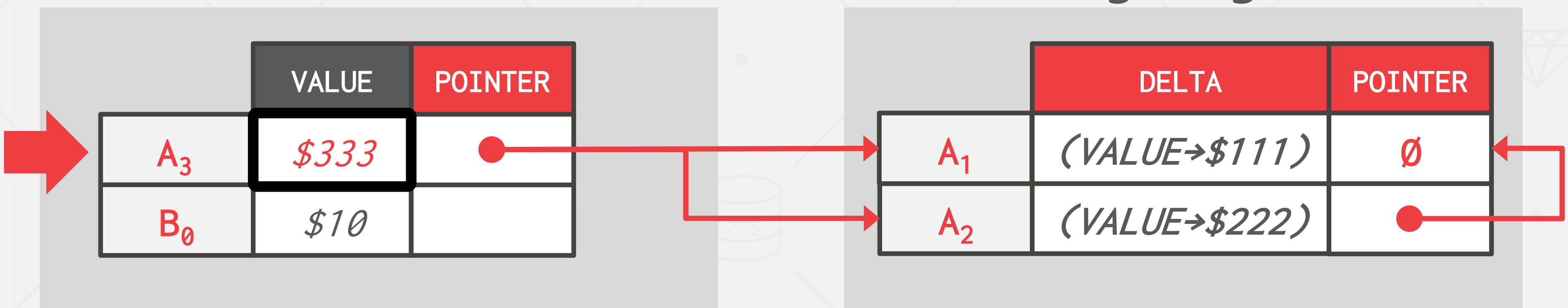


On every update, copy the current version to the time-travel table. Update pointers.

Overwrite master version in the main table and update pointers.

DELTA STORAGE

Main Table ***Delta Storage Segment***



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

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.

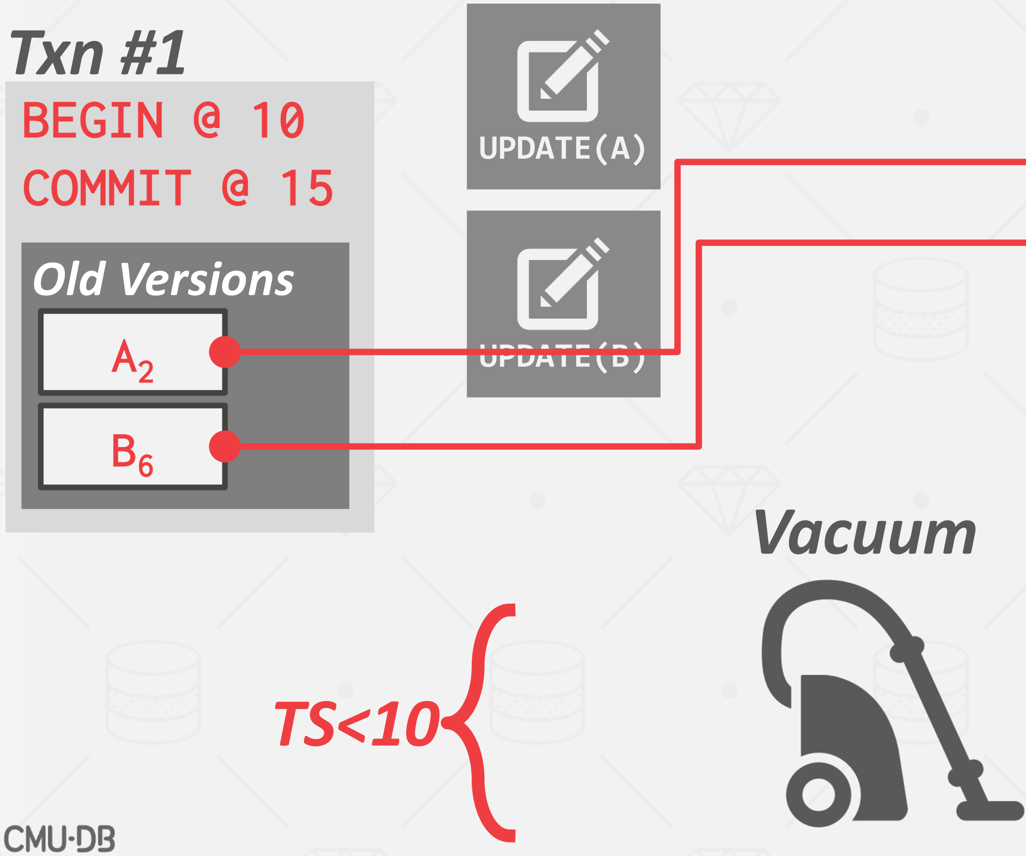
TRANSACTION-LEVEL GC

Each txn keeps track of its read/write set.

On commit/abort, the txn provides this information to a centralized vacuum worker.

The DBMS periodically determines when all versions created by a finished txn are no longer visible.

TRANSACTION-LEVEL GC



	BEGIN-TS	END-TS	DATA
A_2	1	10	-
B_6	8	10	-
A_3	10	∞	-
B_7	10	∞	-

INDEX MANAGEMENT

Primary key indexes 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 are more complicated…

UBER Engineering

JOIN THE TEAM

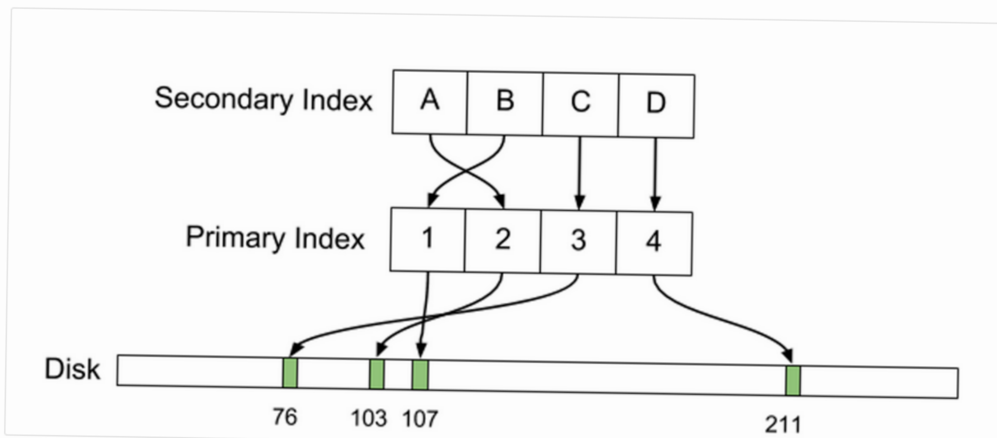
MEET THE PEOPLE

ARCHITECTURE

WHY UBER ENGINEERING SWITCHED FROM POSTGRES TO MYSQL

JULY 26, 2016

BY EVAN KLITZKE

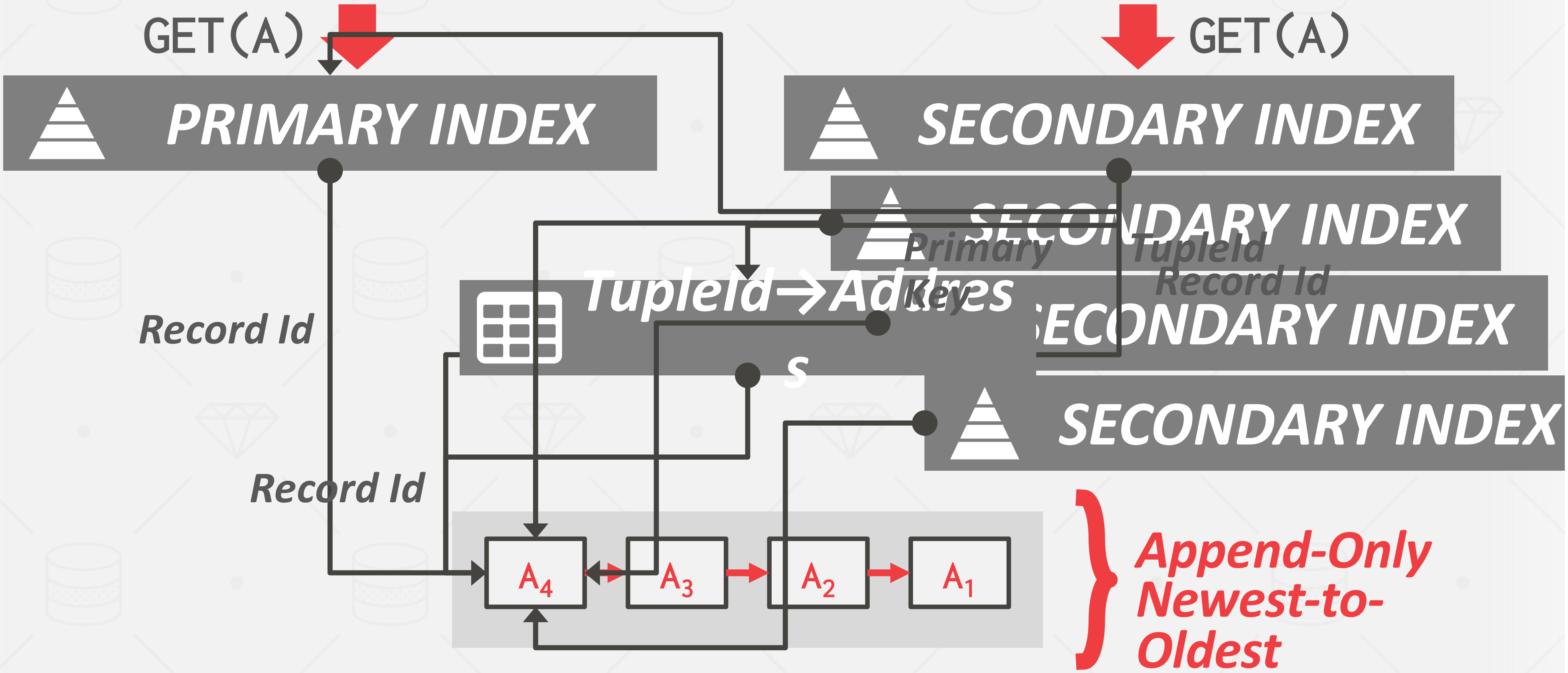


SECONDARY INDEXES

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



MVCC INDEXES

MVCC DBMS indexes (usually) do not store version information about tuples with their keys.

→ Exception: Index-organized tables (e.g., MySQL)

Every index must support duplicate keys from different snapshots:

→ The same key may point to different logical tuples in different snapshots.

MVCC DUPLICATE KEY PROBLEM

Txn #1

BEGIN @ 10

READ(A)

READ(A)

Txn #2

BEGIN @ 20

COMMIT @ 25

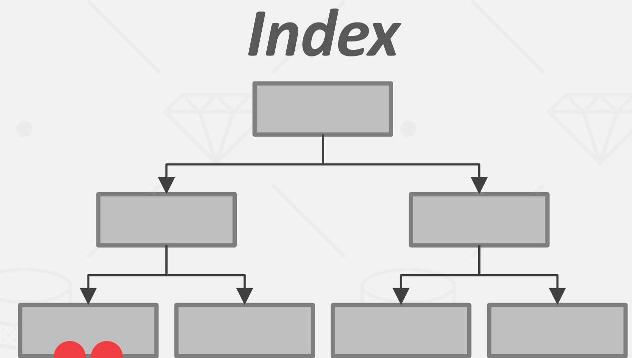
UPDATE(A)

DELETE(A)

Txn #3

BEGIN @ 30

INSERT(A)



	BEGIN-TS	END-TS	POINTER
A_1	1	20	●
A_2	20	20	Ø
A_1	30	∞	Ø

MVCC INDEXES

Each index's underlying data structure must support the storage of non-unique keys.

Use additional execution logic to perform conditional inserts for pkey / unique indexes.

→ Atomically check whether the key exists and then insert.

Workers may get back multiple entries for a single fetch. They then must follow the pointers to find the proper physical version.

MVCC DELETES

The DBMS physically deletes a tuple from the database only when all versions of a logically deleted tuple are not visible.

→ If a tuple is deleted, then there cannot be a new version of that tuple after the newest version.
→ No write-write conflicts / first-writer wins

We need a way to denote that tuple has been logically delete at some point in time.

MVCC DELETES

Approach #1: Deleted Flag

→ Maintain a flag to indicate that the logical tuple has been deleted after the newest physical version.

→ Can either be in tuple header or a separate column.

Approach #2: Tombstone Tuple

→ Create an empty physical version to indicate that a logical tuple is deleted.

→ Use a separate pool for tombstone tuples with only a special bit pattern in version chain pointer to reduce the storage overhead.

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 (2015)	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
CockroachDB	MV-2PL	Delta (LSM)	Compaction	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 and recovery!