CS660: Grad Intro to Database Systems

Class 22: More on Concurrency Control

(Timestamp-Based, Optimistic, and Multi-version)

Instructor: Manos Athanassoulis

https://bu-disc.github.io/CS660/

slides based on Andy Pavlo's CS15-445/645 class

Concurrency Control Approaches

- **Two-Phase Locking (2PL)**
  - Determine serializability order of conflicting operations at runtime while Xacts execute.

Last time

- **Timestamp Ordering (T/O)**
  - A serialization mechanism using timestamps.
- **Optimistic Concurrency Control (OCC)**
  - Run then check for serialization violations.

Concurrency Control Approaches

- **Two-Phase Locking (2PL)**
  - Determine serializability order of conflicting operations at runtime while Xacts execute.
- **Timestamp Ordering (T/O)**
  - A serialization mechanism using timestamps.

Pessimistic

- **Optimistic Concurrency Control (OCC)**
  - Run then check for serialization violations.

Optimistic

T/O Concurrency Control

- Use timestamps to determine the serializability order of Xacts.

- If $TS(T_i) < TS(T_j)$, then the DBMS must ensure that the execution schedule is equivalent to the serial schedule where T_i appears before T_j.

Timestap Allocation

- Each Xact T_i is assigned a unique fixed timestamp that is monotonically increasing.
  - Let $TS(T_i)$ be the timestamp allocated to Xact T_i.
  - Different schemes assign timestamps at different times during the Xact.

- Multiple implementation strategies:
  - System/Wall Clock.
  - Logical Counter.
  - Hybrid.

Today's Agenda

- Basic Timestamp Ordering (T/O) Protocol

- Optimistic Concurrency Control

- Multi-Version Concurrency Control

Basic T/O

- Xacts **read** and **write** objects **without** locks.

- Every **object** X **is tagged with timestamp** of the last Xact that successfully did read/write:
  - `W-TS(X)` – Write timestamp on X
  - `R-TS(X)` – Read timestamp on X

- Check timestamps for every operation:
  - If Xact tries to access an object "from the future", it aborts and restarts.

Basic T/O – Reads

Don't read stuff from the "future."

- Action: Transaction T_i wants to read object X.
- If $TS(T_i) < W\text{-}TS(X)$, this violates the timestamp order of T_i with regard to the writer of X.
  - Abort T_i and restart it with a new TS.
- Else:
  - Allow T_i to read X.
  - Update $R\text{-}TS(X)$ to $\max(R\text{-}TS(X),\ TS(T_i))$
  - Make a local copy of X to ensure repeatable reads for T_i.

Basic T/O – Writes

Can’t write if a future transaction has read or written to the object.

- Action: Transaction T_i wants to write object X.
- If $TS(T_i) < R\text{-}TS(X)$ **or** $TS(T_i) < W\text{-}TS(X)$
  - Abort and restart T_i.
- Else:
  - Allow T_i to write X and update $W\text{-}TS(X)$
  - Also, make a local copy of X to ensure repeatable reads.

Basic T/O – Example #1

Schedule

TIME

$TS(T_1)=1$

$TS(T_2)=2$

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

Database

$TS(T_1) < TS(T_2)$

Object	R-TS	W-TS
A	2	2
B	2	2

No violations so both Xacts are safe to commit.

Reminder

Reads on X fail if we get:

$TS(T_i) < W\text{-}TS(X)$

Writes on X fail if we get:

$TS(T_i) < W\text{-}TS(X)$ or $TS(T_i) < W\text{-}TS(X)$

Basic T/O – Example #2

Schedule

TIME

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

T_1 cannot overwrite update by T_2, so the DBMS must abort it!

Database

Object	R-TS	W-TS
A	1	2
B	0	0

Violation:
$TS(T_1) < W\text{-}TS(A)$

Reminder

Reads on X fail if we get:

$TS(T_i) < W\text{-}TS(X)$

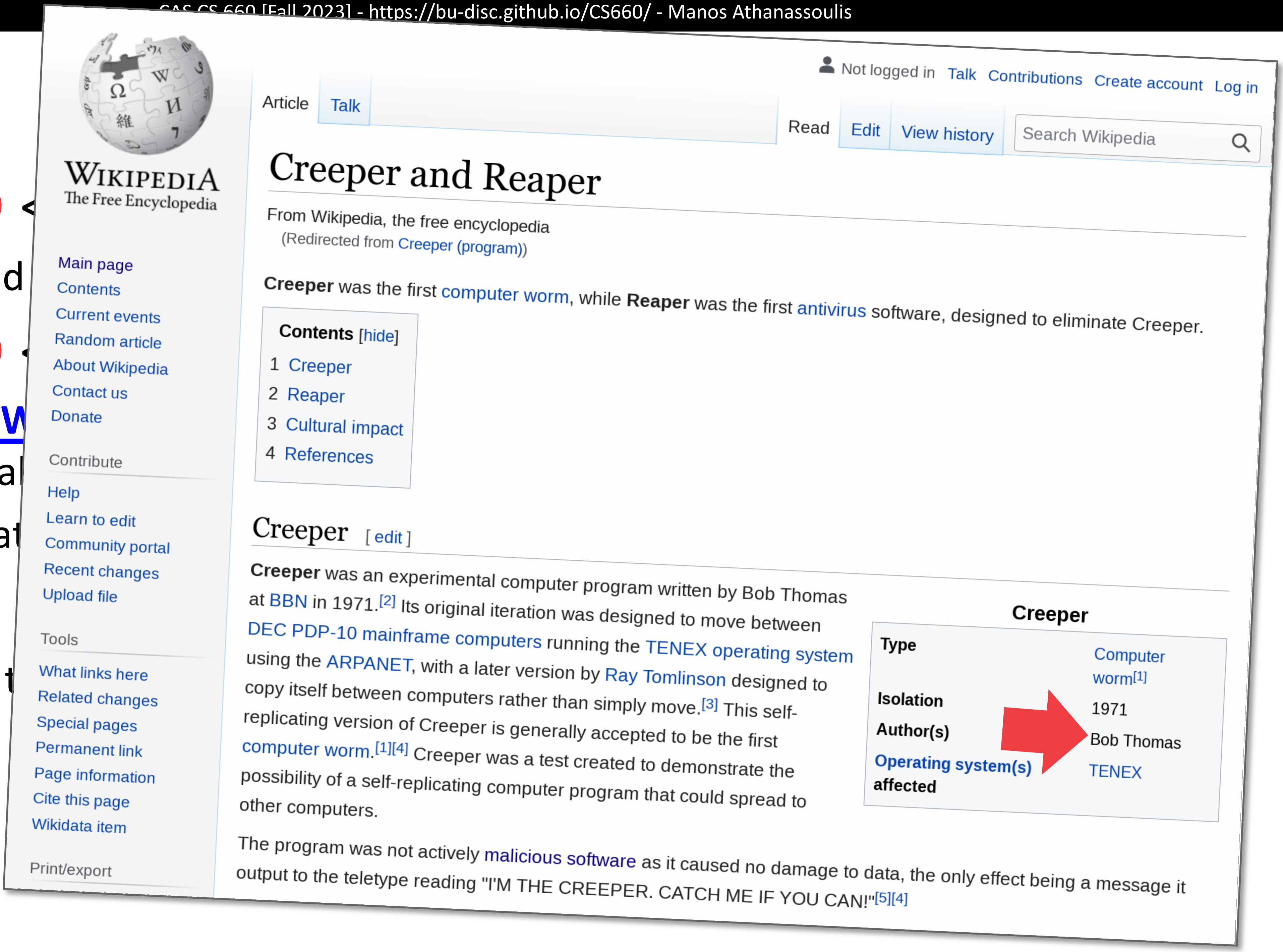
Writes on X fail if we get:

$TS(T_i) < W\text{-}TS(X)$ or $TS(T_i) < W\text{-}TS(X)$

Thomas Write Rule

- If $TS(T_i) < R\text{-}TS(X)$:
  - Abort and restart T_i.
- If $TS(T_i) < W\text{-}TS(X)$:
  - **Thomas Write Rule**: Ignore the write to allow the Xact to continue executing without aborting.
  - This violates timestamp order of T_i.
- Else:
  - Allow T_i to write X and update $W\text{-}TS(X)$

- If $TS(T_i)$ <
  - Abort and
- If $TS(T_i)$ <
  - **Thomas W**
    without a
  - This violat
- Else:
  - Allow T_i t



Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Contribute
Help
Learn to edit
Community portal
Recent changes
Upload file

Tools
What links here
Related changes
Special pages
Permanent link
Page information
Cite this page
Wikidata item

Print/export

Creeper and Reaper

From Wikipedia, the free encyclopedia
(Redirected from Creeper (program))

Creeper was the first computer worm, while **Reaper** was the first antivirus software, designed to eliminate Creeper.

Contents [hide]

Creeper [edit]

Creeper was an experimental computer program written by Bob Thomas at BBN in 1971.[2] Its original iteration was designed to move between DEC PDP-10 mainframe computers running the TENEX operating system using the ARPANET, with a later version by Ray Tomlinson designed to copy itself between computers rather than simply move.[3] This self-replicating version of Creeper is generally accepted to be the first computer worm.[1][4] Creeper was a test created to demonstrate the possibility of a self-replicating computer program that could spread to other computers.

Creeper	
Type	Computer worm[1]
Isolation	1971
Author(s)	Bob Thomas
Operating system(s) affected	TENEX

The program was not actively malicious software as it caused no damage to data, the only effect being a message it output to the teletype reading "I'M THE CREEPER. CATCH ME IF YOU CAN!"[5][4]

Basic T/O – Example #2

Schedule

TIME

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

Skip doing the actual write and allow T_1 to commit. (Do write to the local copy if repeatable read is required.)

Database

Object	R-TS	W-TS
A	1	2
B	0	0

We do not update W-TS(A)

Basic T/O – Example #2

Schedule

TIME

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

*Note that skipping W(A) is **view-equivalent** to the serial schedule {T_1->T_2} (due to local copies)*

Database

Object	R-TS	W-TS
A	1	2
B	0	0

Basic T/O

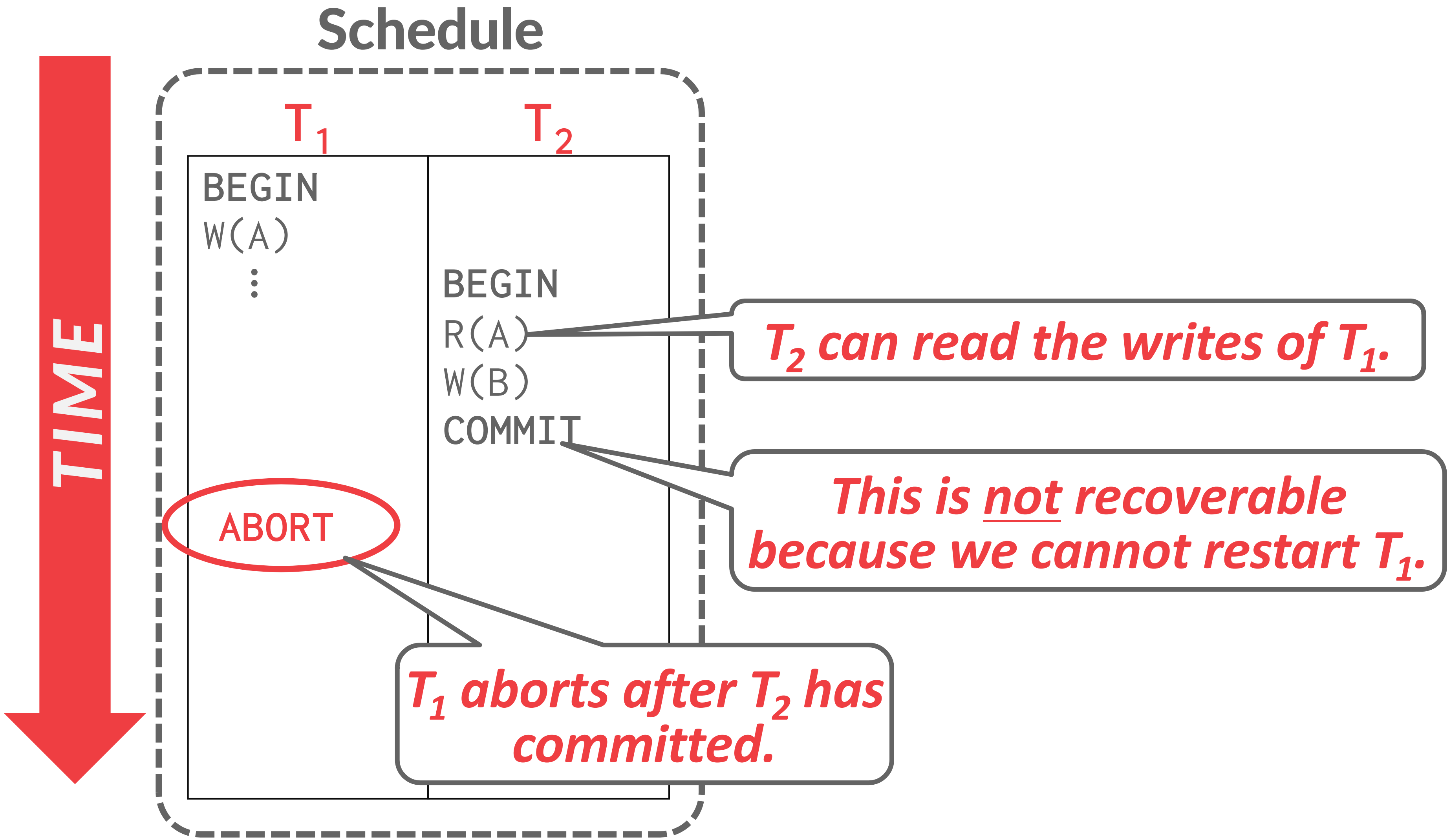
- Generates a schedule that is conflict serializable if you do **not** use the Thomas Write Rule.
  - **No deadlocks** because **no Xact ever waits**.
  - Possibility of **starvation** for long Xacts if **short Xacts** keep causing **conflicts**.

- Not aware of any DBMS that uses the basic T/O protocol described here.
  - It provides the building blocks for OCC / MVCC.

Recoverable Schedules

- A schedule is **recoverable** if Xacts commit only after all Xacts whose changes they read, commit.
- Otherwise, the DBMS cannot guarantee that Xacts read data that will be restored after recovering from a crash.

Recoverable Schedules



Ensuring Recoverable Schedules

- Basic T/O can be modified to allow only recoverable schedules:
  - Buffer all writes until writer commits (but update W-TS for allowed writes)
  - Block readers T when `TS(T) > W-TS(X)`, until writer of X commits

- Similar to writers holding exclusive locks until commit
  - Still allows for higher concurrency!

Basic T/O – Performance Issues

- High overhead from **copying data** to Xact's workspace and from **updating timestamps**.
  - Every **read** requires the Xact to **write** to the database.

- Long running Xacts can get **starved**.
  - The likelihood that a Xact will read something from a newer Xact increases.

Observation

- If you assume that conflicts between Xacts are **rare** and that most Xacts are **short-lived**, then forcing Xacts to acquire locks or update timestamps adds unnecessary overhead.

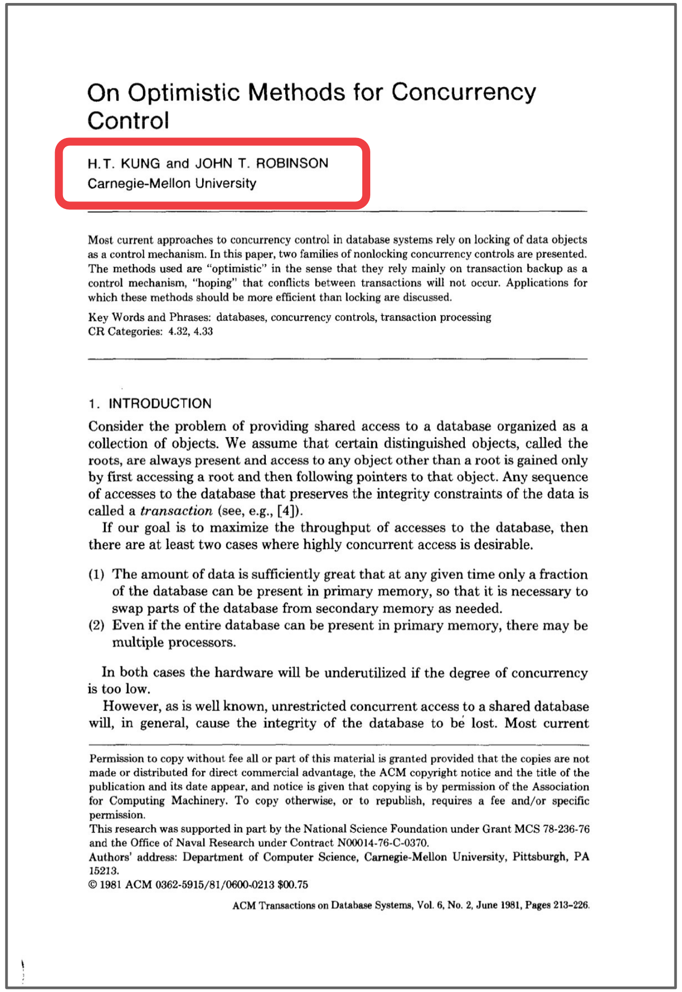
- A better approach is to optimize for the **no-conflict** case.

Optimistic Concurrency Control

- The DBMS creates a **private workspace** for each Xact.
  - Any object read is copied into workspace.
  - Modifications are applied to workspace.

- When a Xact **commits**, the DBMS **compares workspace write set** to see whether it **conflicts** with other Xacts.

- If there are **no conflicts**, the write set is installed into the “global” database.



On Optimistic Methods for Concurrency Control

H.T. KUNG and JOHN T. ROBINSON
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are “optimistic” in the sense that they rely mainly on transaction backup as a control mechanism, “hoping” that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

Key Words and Phrases: databases, concurrency controls, transaction processing
CR Categories: 4.32, 4.33

1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

(1) The amount of data is sufficiently great that at any given time only a fraction of the database can be present in primary memory, so that it is necessary to swap parts of the database from secondary memory as needed.
(2) Even if the entire database can be present in primary memory, there may be multiple processors.

In both cases the hardware will be underutilized if the degree of concurrency is too low.

However, as is well known, unrestricted concurrent access to a shared database will, in general, cause the integrity of the database to be lost. Most current

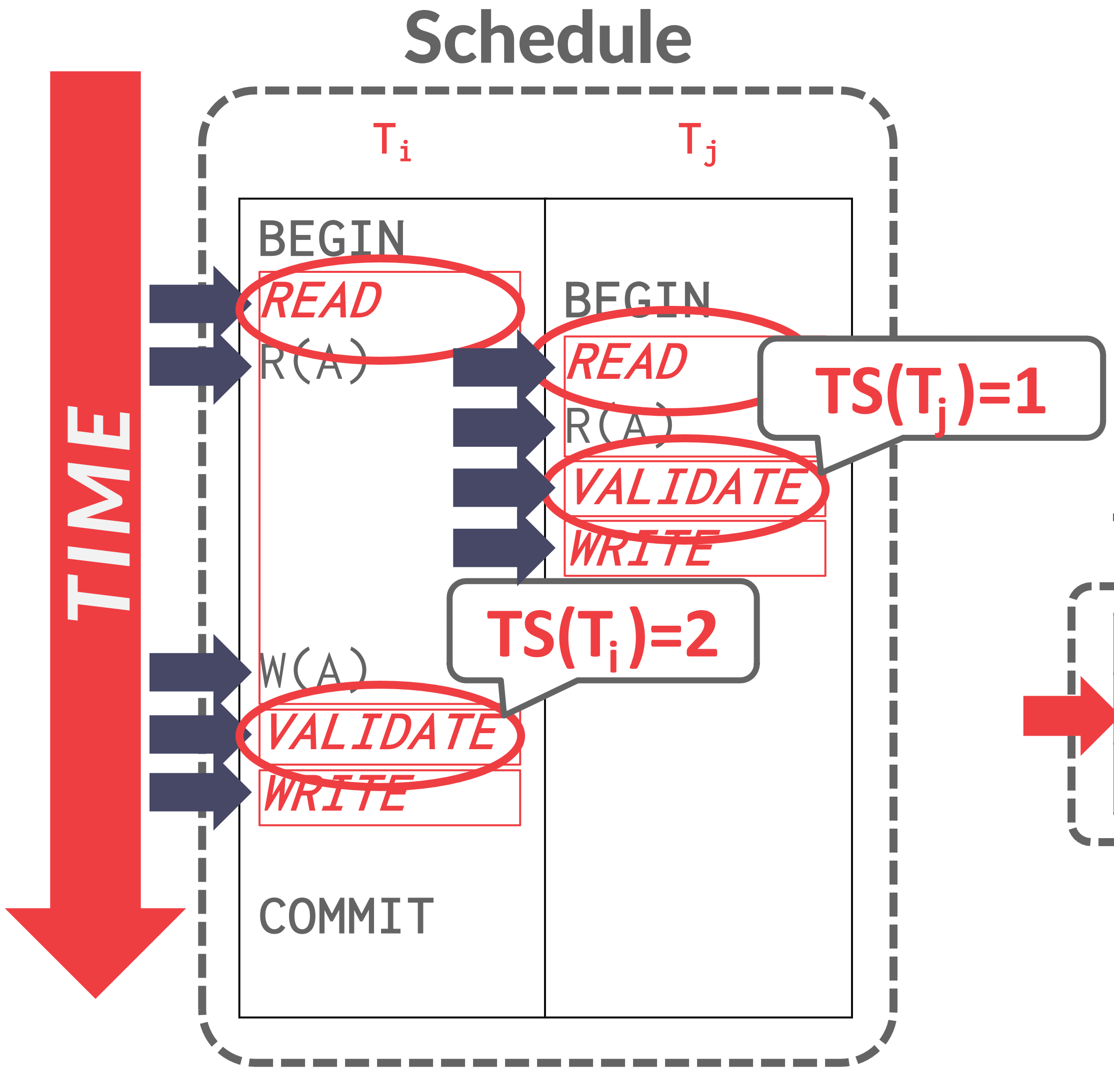
This research was supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.
Authors' address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

ACM Transactions on Database Systems, Vol. 6, No. 2, June 1981, Pages 213–226.

OCC Phases

- **#1 – Read Phase**:
  - Track the read/write sets of Xacts and store their writes in a private workspace.
- **#2 – Validation Phase**:
  - When a Xact commits, check whether it conflicts with other Xacts.
- **#3 – Write Phase:**
  - If validation succeeds, apply private changes to database. Otherwise abort and restart the Xact.

OCC – Example



Database

Object	Value	W-TS
A	456	2
-	-	-

T_1 Workspace

Object	Value	W-TS
A	456	∞
-	-	-

T_2 Workspace

Object	Value	W-TS
A	123	0
-	-	-

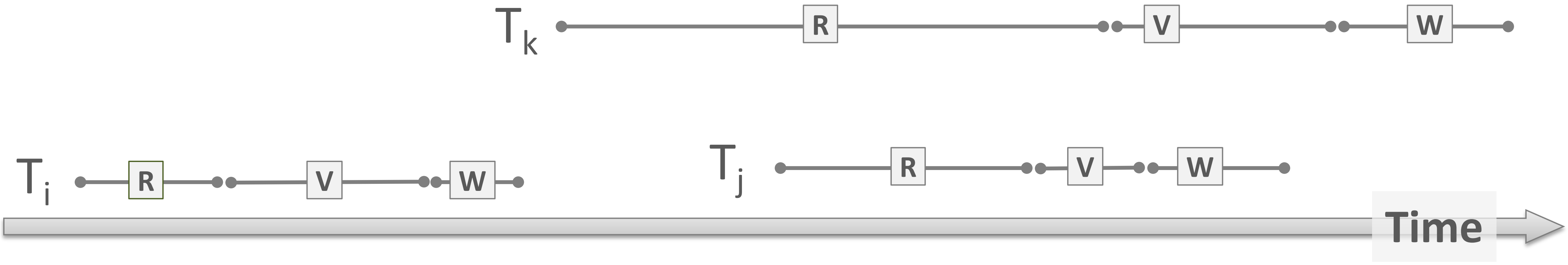
OCC – Read Phase

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

- The **DBMS copies** every **tuple** that the Xact **accesses** from the shared database to its **workspace** to ensure **repeatable reads**.
  - this means no RW conflicts!
  - We can ignore for now what happens if a Xact reads/writes tuples via indexes.

OCC: Three Phases

When to assign the transaction number? At the end of the read phase.



1. **READ** Phase: Read and write objects, making local copies.

2. **VALIDATION** Phase: Check for serializable schedule-related anomalies.

3. **WRITE** Phase: If it is safe, write the local objects, making them permanent.

Anomalies with Interleaved Execution

Reminder!

RW conflict (Unrepeatable Reads):

T1:	R(A),		R(A), W(A), C
T2:		R(A), W(A), C	

WR conflict (Dirty Reads) :

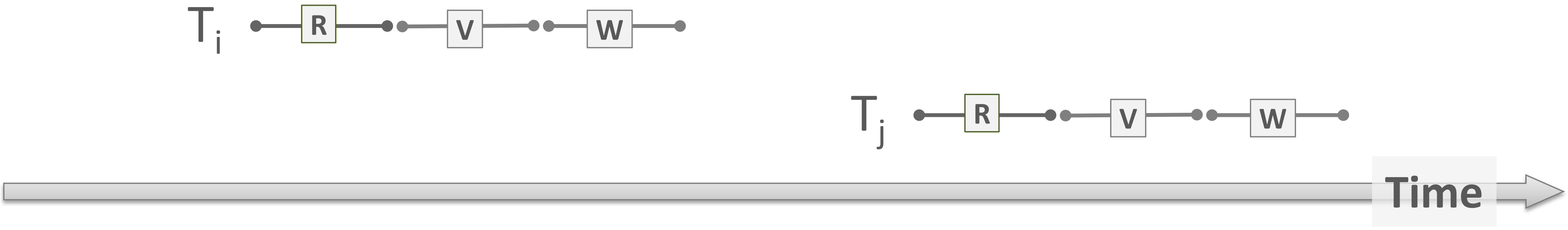
T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

WW conflict (Overwriting Uncommitted Data):

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

OCC: Validation ($T_i < T_j$) and no overlap!

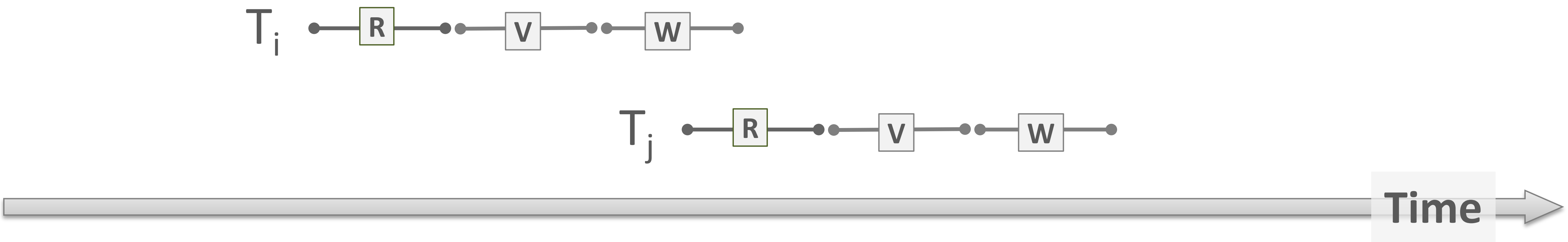
Case 1: T_i completes its write phase **before** T_j starts its read phase.



- No conflict as all of T_i's actions happen before T_j's.

OCC: Validation ($T_i < T_j$) and write-read phases may overlap!

Case 2: T_i completes its write phase **before** T_j starts its write phase.

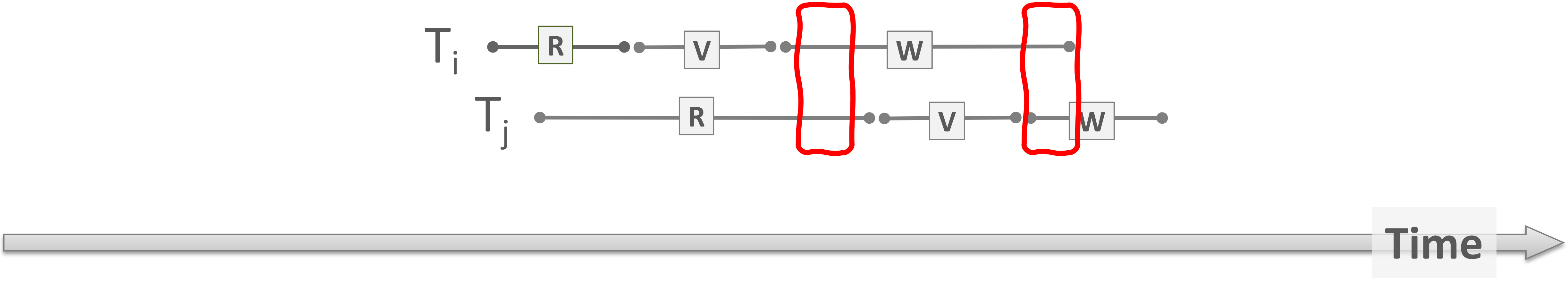


- Check that the write set of T_i does not intersect the read set of T_j, namely: $\texttt{WriteSet}(T_i) \cap \texttt{ReadSet}(T_j) = \emptyset$

No RW conflicts trivially.	No WW because of the condition of the case.	Does T_j read dirty data (WR conflict)?
Tid assignment!		Maybe …

OCC: Validation ($T_i < T_j$) and write-write phases may overlap!

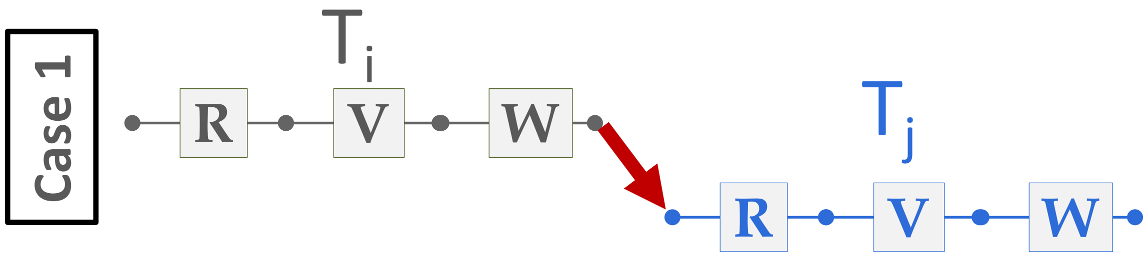
Case 3: T_i completes its read phase **before** T_j completes its read phase.

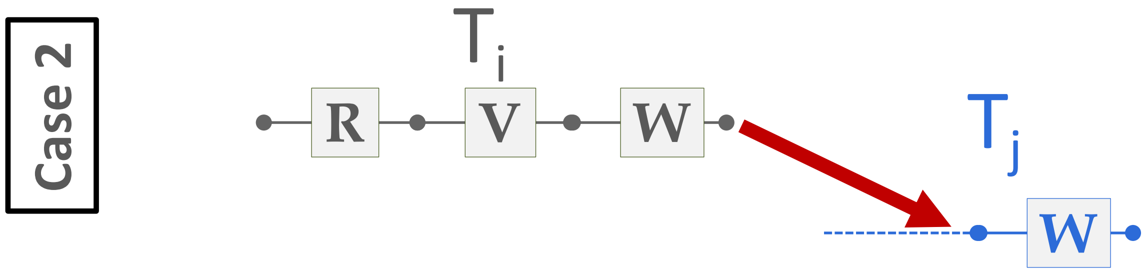


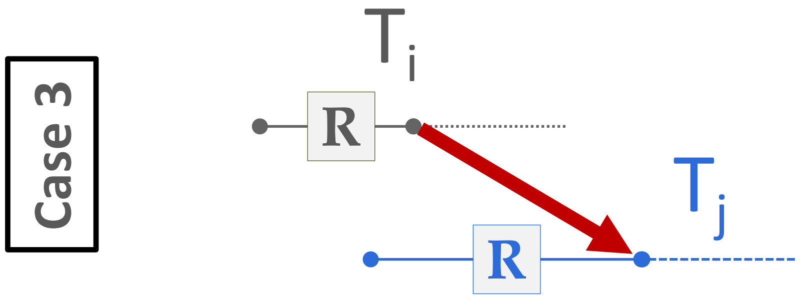
- Check that the write set of T_i does not intersect the read or write sets of T_j, namely: `WriteSet(`T_i`)` ∩ `ReadSet(`T_j`)` = Ø **AND** `WriteSet(`T_i`)` ∩ `WriteSet(`T_j`)` = Ø

No RW conflicts trivially. | WW conflicts? T_i may overwrite T_j data | WR conflicts? T_j may read dirty data

OCC: Validation ($T_i < T_j$)







	R → W	W → R	W → W
Case 1	✓	✓	✓
Case 2	✓	$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$	✓
Case 3	✓	$WriteSet(T_i) \cap ReadSet(T_j) = \emptyset$	$WriteSet(T_i) \cap WriteSet(T_j) = \emptyset$

OCC – Validation Phase

To validate Xact T (testing cases 1, 2, 3):

```
S ← set of Xacts that committed after Begin(T) /*tests Case 1*/
valid = true;
//The following is done in critical section
< foreach  T_s in S do {
  if (ReadSet(T) ∩ WriteSet(T_S) ≠ Ø) OR (WriteSet(T) ∩ WriteSet(T_S) ≠ Ø)
        then valid = false;
  }>
  if valid then { install updates; /* Write phase */
                  Commit T }
        else Restart T
```

Critical section

OCC – Validation Phase

To validate Xact T (serial validation -- testing cases 1, 2):

```
S ← set of Xacts that committed after Begin(T) /*tests Case 1*/
valid = true;
//The following is done in critical section
< foreach  T_s in S do {
  if (ReadSet(T) ∩ WriteSet(T_S) ≠ Ø)
        then valid = false;
  }>
  if valid then { install updates; /* Write phase */
                  Commit T }
            else Restart T
```

Critical section

OCC – Serial Validation Observation

- Tests for Case 2: T as T_j and each Xact in T_S (in turn) as T_i.
- Xact id assignment, validation, write inside a critical section!
  - Nothing else goes on concurrently.
  - So, no need to test Case 3 --- cannot happen.
  - If Write phase is long, major drawback.

- Optimization for Read-only Xacts:
  - No need for critical section (because there is no Write phase).

OCC – Write Phase

- Propagate changes in the Xact's write set to database to make them visible to other Xacts.

- **Serial Commits:**
  - Use a global latch to limit a single Xact to be in the **Validation**/**Write** phases at a time.

- **Parallel Commits:**
  - Use fine-grained write latches to support parallel **Validation**/**Write** phases.
  - Xacts acquire latches in primary key order to avoid deadlocks.

OCC – Observations

- OCC works well when the # of conflicts is low:
  - All Xacts are read-only (ideal).
  - Xacts access disjoint subsets of data.

- If the database is large and the workload is not skewed, then there is a low probability of conflict, so again locking is wasteful.

OCC – Performance Issues

- High overhead for copying data locally.
- Validation/Write phase bottlenecks.
- Aborts are more wasteful than in 2PL because they only occur <u>after</u> a Xact has already executed.

Do we need to update data (and thus, cause conflicts) all the time?

MULTI-VERSION CONCURRENCY CONTROL

Multi-Version Concurrency Control (MVCC)

- The DBMS maintains multiple **physical** versions of a single **logical** object in the database:
  - →When a **Xact writes** to an object, the DBMS **creates a new version** of that object.
  - →When a **Xact reads** an object, it **reads** the newest version that **existed when the Xact started**.

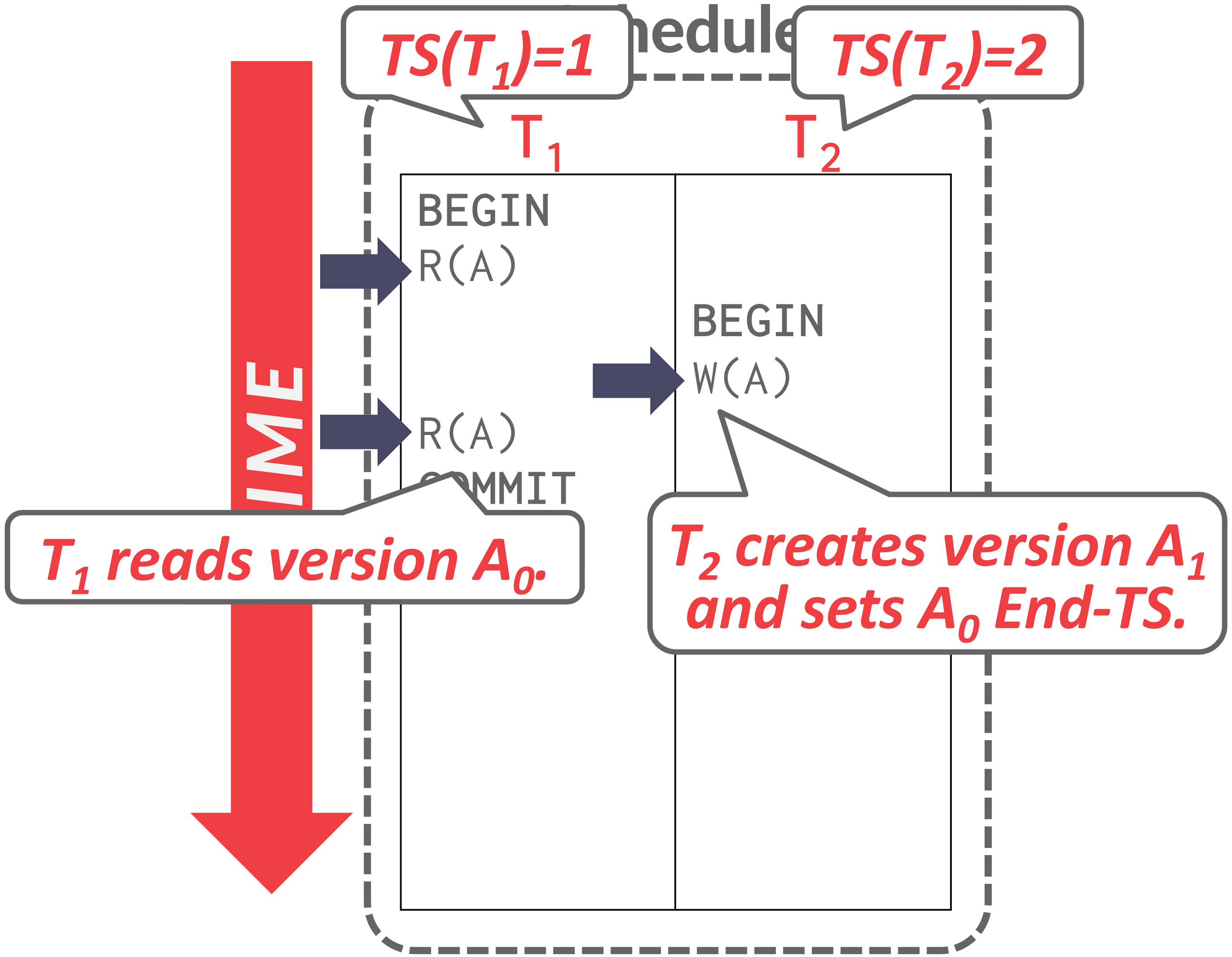
Multi-Version Concurrency Control

- **Writers do not block readers.**
  Readers do not block writers.

- Read-only Xacts can read a consistent snapshot without acquiring locks.
  - Use timestamps to determine visibility.

- Easily support time-travel queries.

MVCC – Example #1



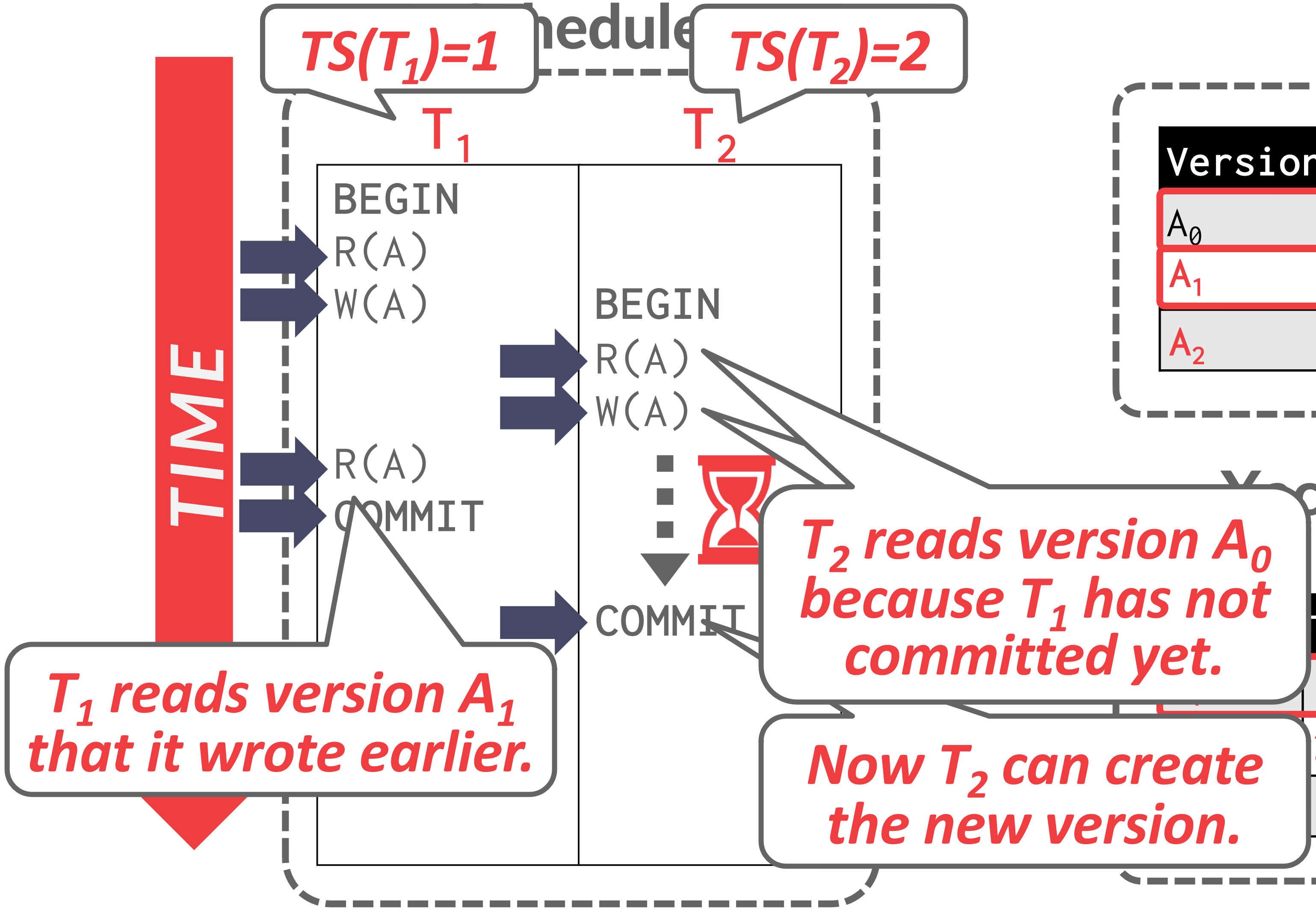
Database

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

Xact Status Table

XactId	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	2
A_2	789	2	-

Xact Status Table

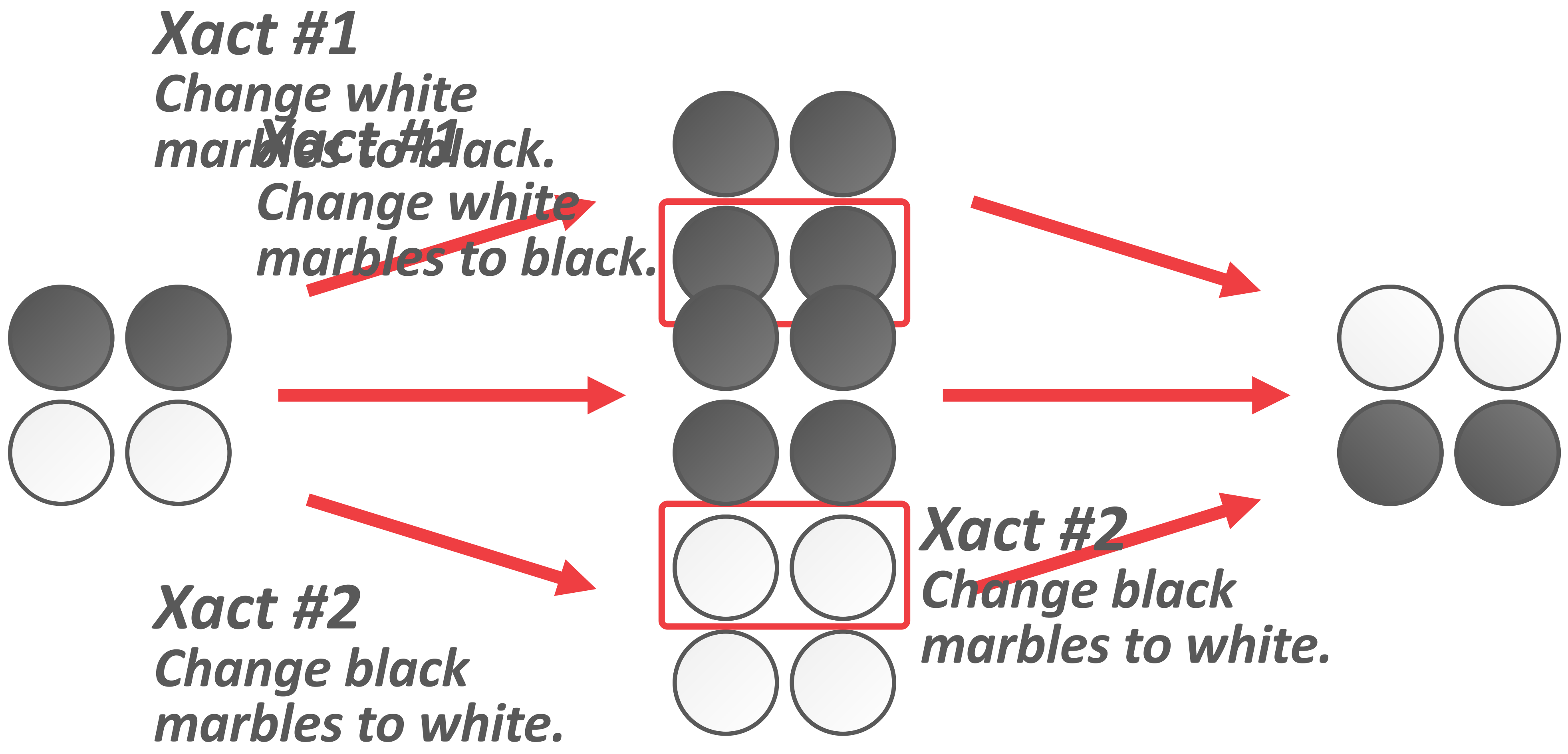
Timestamp	Status
1	Committed
2	Active

Snapshot Isolation (SI)

- When a Xact starts, it sees a consistent snapshot of the database that existed when that the Xact started.
  - No torn writes from active Xacts.
  - If two Xacts update the same object, then first writer wins.

- SI is susceptible to the **Write Skew Anomaly**.

Write Skew Anomaly



Multi-Version Concurrency Control

MVCC is more than just a concurrency control protocol. It completely affects how the DBMS manages transactions and the database.

splice MACHINE
ALTIBASE
vectorwise
etcd
Exasol
InfiniDB
eXtremeDB
INGRES
WIREDTIGER
H2
TIMESCALE
NUODB
NETEZZA
FOUNDATIONDB
daffodil db Your Complete Java RDBMS
PostgreSQL
Microsoft SQL Server Hekaton
HyPer
CUBRID
Couchbase
IMDB
SAP HANA
UMBRA
SingleStore
ArangoDB
MariaDB
Clustrix
OrientDB
APACHE HBASE
ORACLE
SUBWAY
realm
HYRISE
RAVENDB
noise page
RethinkDB
MySQL
Cockroach LABS

MVCC Design Decisions

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

Concurrency Control Protocols

- **Approach #1: Timestamp Ordering**
  - Assign Xacts timestamps that determine serial order.
- **Approach #2: Optimistic Concurrency Control**
  - Three-phase protocol (Read-Validate-Write).
  - Use private workspace for new versions.
- **Approach #3: Two-Phase Locking**
  - Xacts 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 **<u>version chain</u>** per logical tuple.
  - This allows the DBMS to find the version that is visible to a particular Xact at runtime.
  - Indexes always point to the "head" of the chain.

- Different storage schemes determine where/what to store for each version.

Version Storage

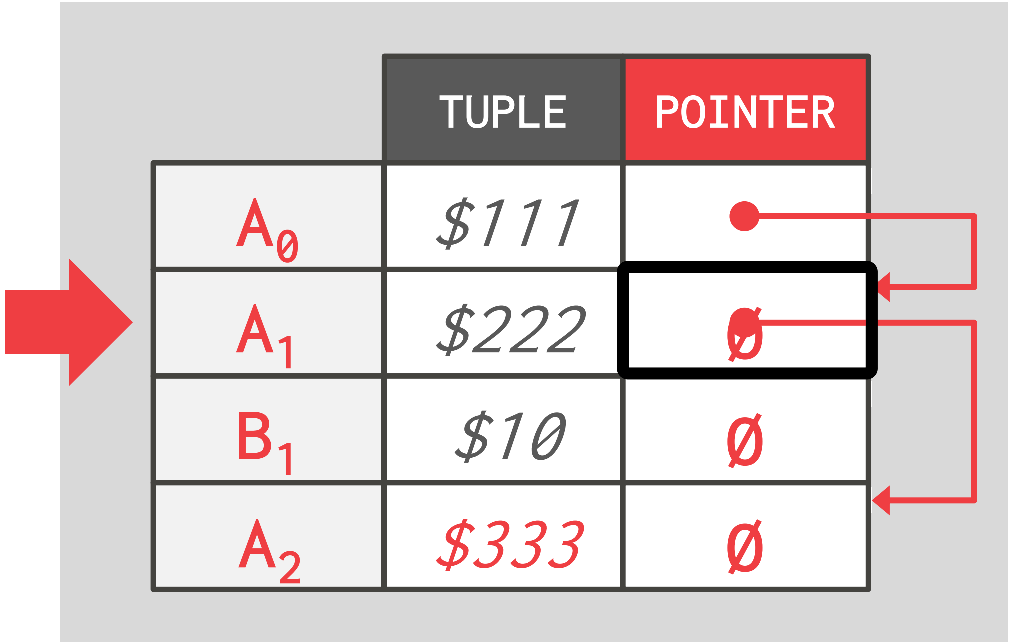
- **Approach #1: Append-Only Storage**
  - New versions are appended to the same table space.
- **Approach #2: Time-Travel Storage**
  - Old versions are copied to separate table space.

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



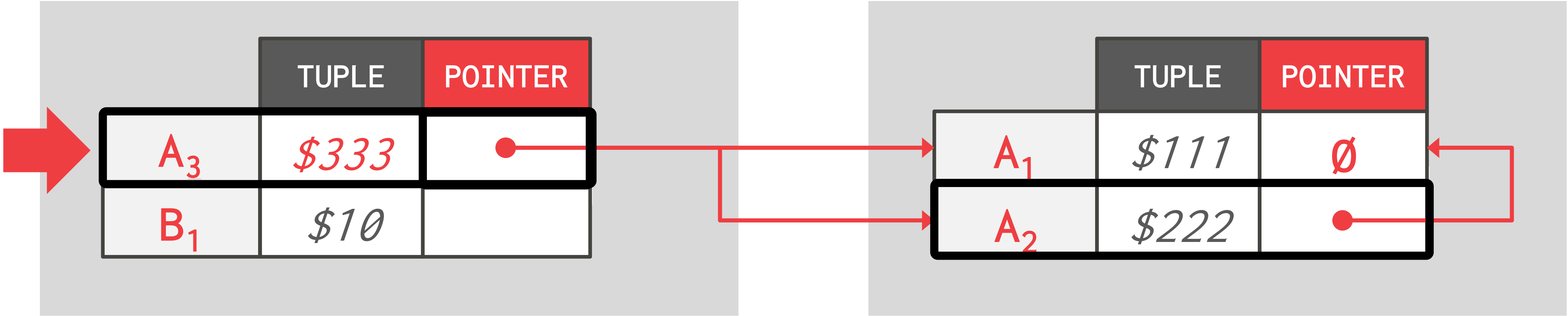
Version Chain Ordering

- **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.

Garbage Collection

- The DBMS needs to remove **reclaimable** physical versions from the database over time.
  - No active Xact in the DBMS can "see" that version (SI).
  - The version was created by an aborted Xact.

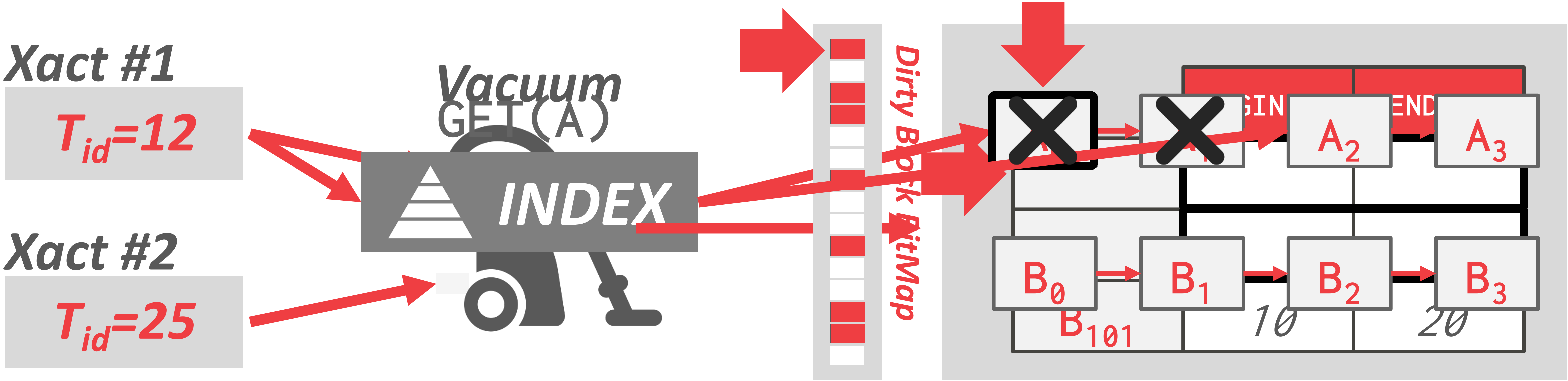
- Two additional design decisions:
  - How to look for expired versions?
  - How to decide when it is safe to reclaim memory?

Garbage Collection

- **Approach #1: Tuple-level**
  - Find old versions by examining tuples directly.
  - <u>Background Vacuuming</u> vs. <u>Cooperative Cleaning</u>

- **Approach #2: Transaction-level**
  - Xacts 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 Xact **keeps track** of its **read/write set**.
- On **commit/abort**, the Xact provides this information to a **centralized vacuum** worker.

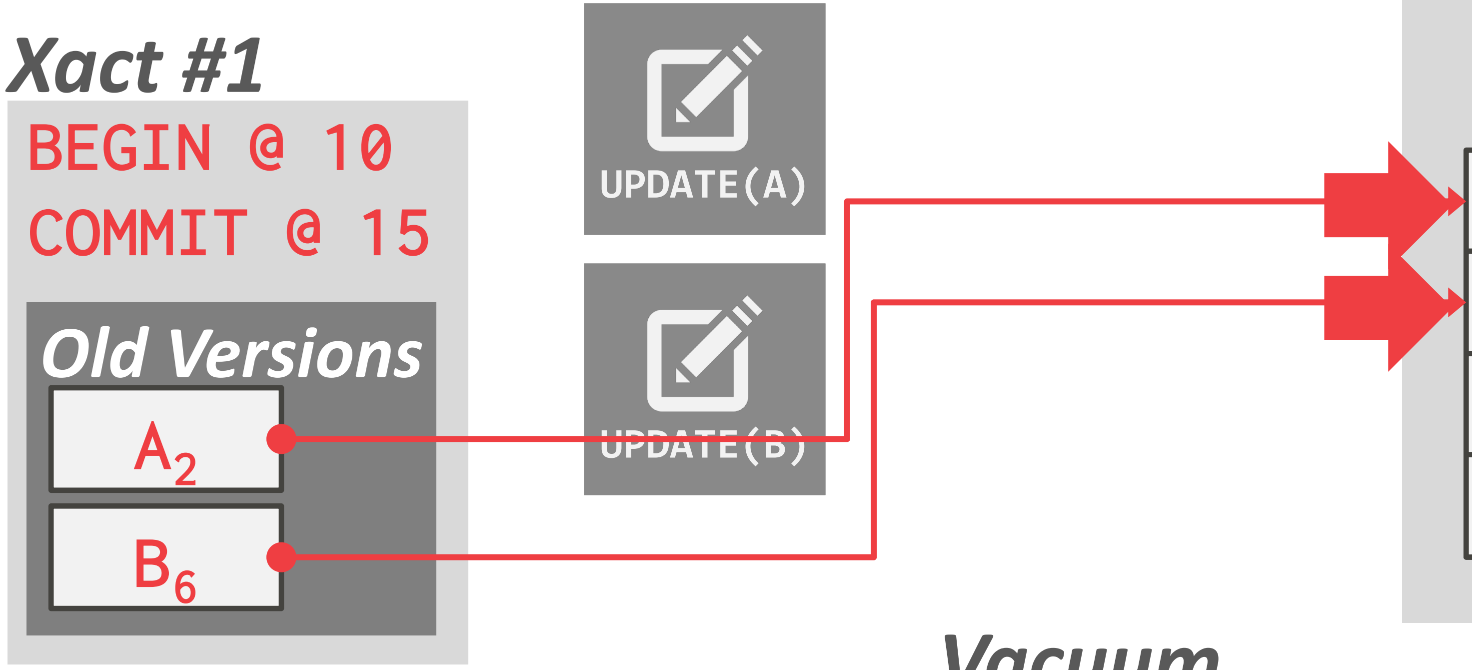
- The DBMS periodically determines when all versions created by a finished Xact are no longer visible.

Transaction-Level GC

Xact #1

BEGIN @ 10

COMMIT @ 15



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

Vacuum

TS<10 {

Next Class

- Logging and recovery!