SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

Recovery Manager

南方科技大学

唐 博

tangb3@sustech.edu.cn

Review

Timestamp Ordering protocol

- Idea: ensure that conflicting operations are executed in timestamp order
  - What are the meanings of TS(T), R-TS(Q), W-TS(Q)?

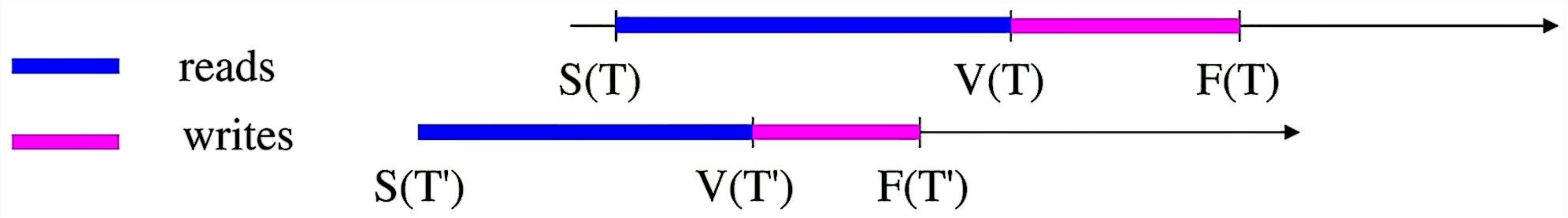
When T issues **read(Q)**	Action
If TS(T) < W-TS(Q)	restart T
Else	execute read(Q), and update R-TS(Q) to max{R-TS(Q),TS(T)}

When T issues **write(Q)**	Action
If TS(T) < R-TS(Q)	restart T
If TS(T) < W-TS(Q)	restart T
Else	execute write(Q), and update W-TS(Q) to max{W-TS(Q),TS(T)}

With Thomas write rule, we ignore write(Q) operation of T when TS(T)≥R-TS(Q)

OCC protocol

- Concepts: Start(T), Validation(T), Finish(T)
  - Decide whether to execute or kill T at time Validation(T)
- No need to memorize the rules!
- Just draw timelines below, then think about:
  - Any read-write conflict?
  - Any write-read conflict?
  - Any write-write conflict?
- When there is potential conflict, use the **ReadSet / WriteSet** of T and T' to check whether T must be killed



Lecture Objectives

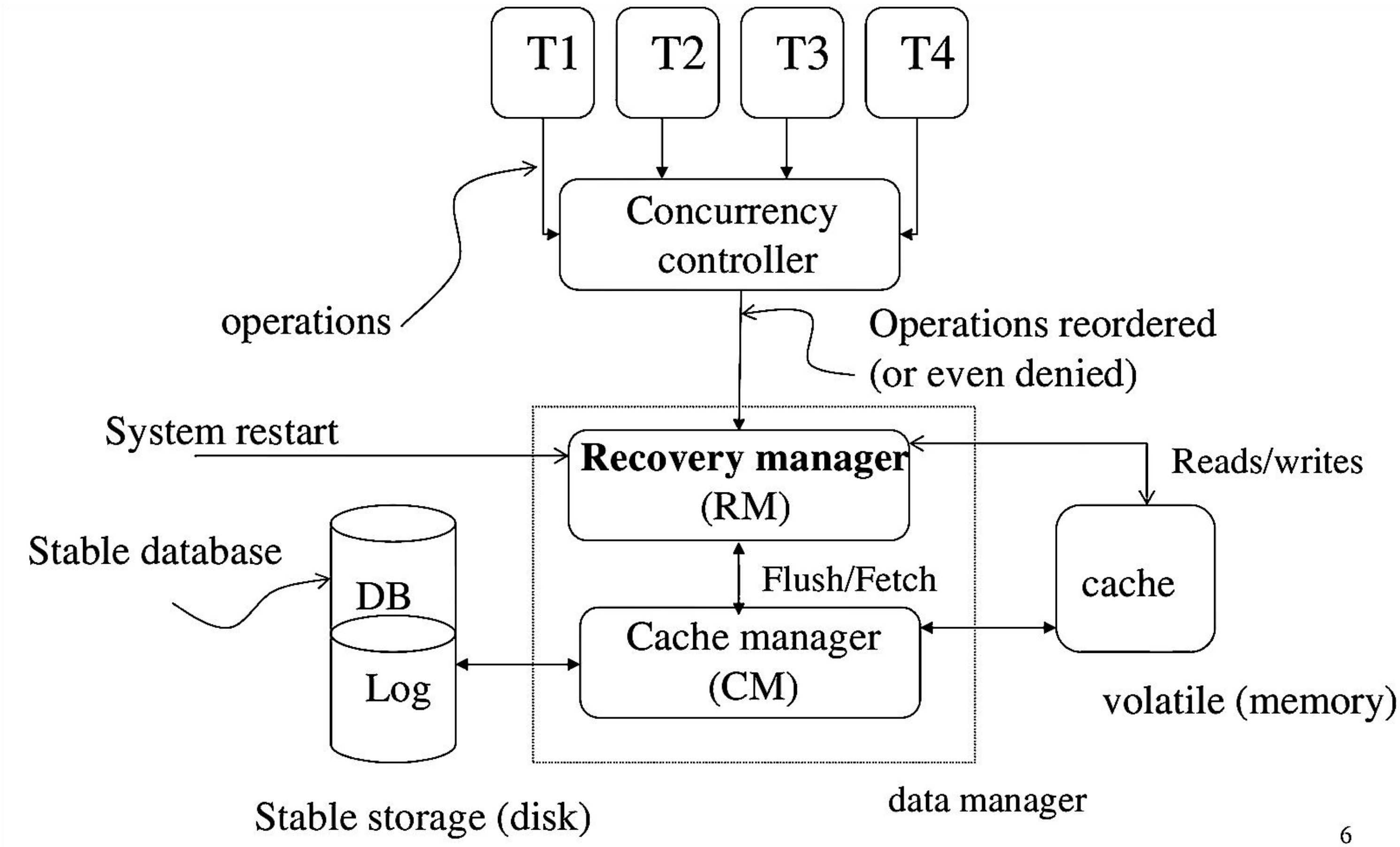
- The Recovery Manager
- A Simple Recovery Algorithm
- Checkpointing
- Log Buffering

Failures

- *System failure*: content of volatile storage (i.e., main memory) is lost / corrupted
  - E.g., power failure, system crash
  - Assume that the stable (non-volatile) storage is ok

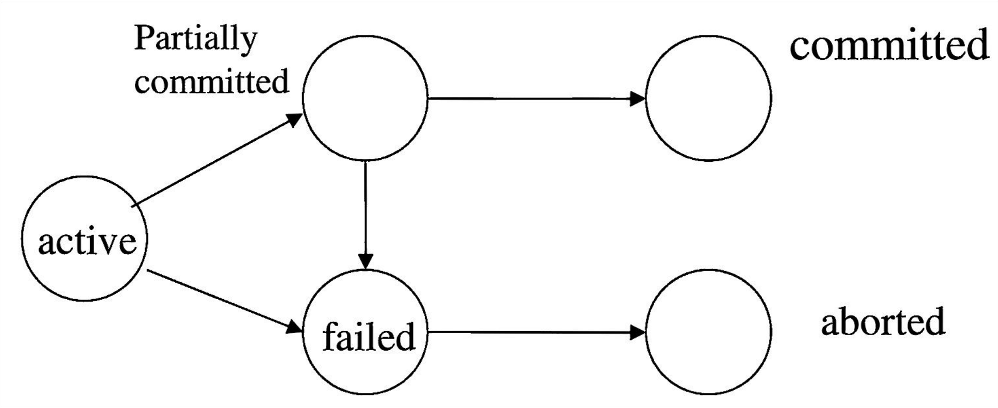
- *Media failure*: part of the stable storage is destroyed
  - E.g., disk crashes
  - Use replicated copies to recover data
- To cope with failures, RDBMS supports recovery
  - https://msdn.microsoft.com/en-us/library/ms191253.aspx#RMsAndSupportedRestoreOps
  - https://docs.oracle.com/cd/B28359_01/backup.111/b28270/rcmcomre.htm#BRADV89759

Recovery Manager (RM)



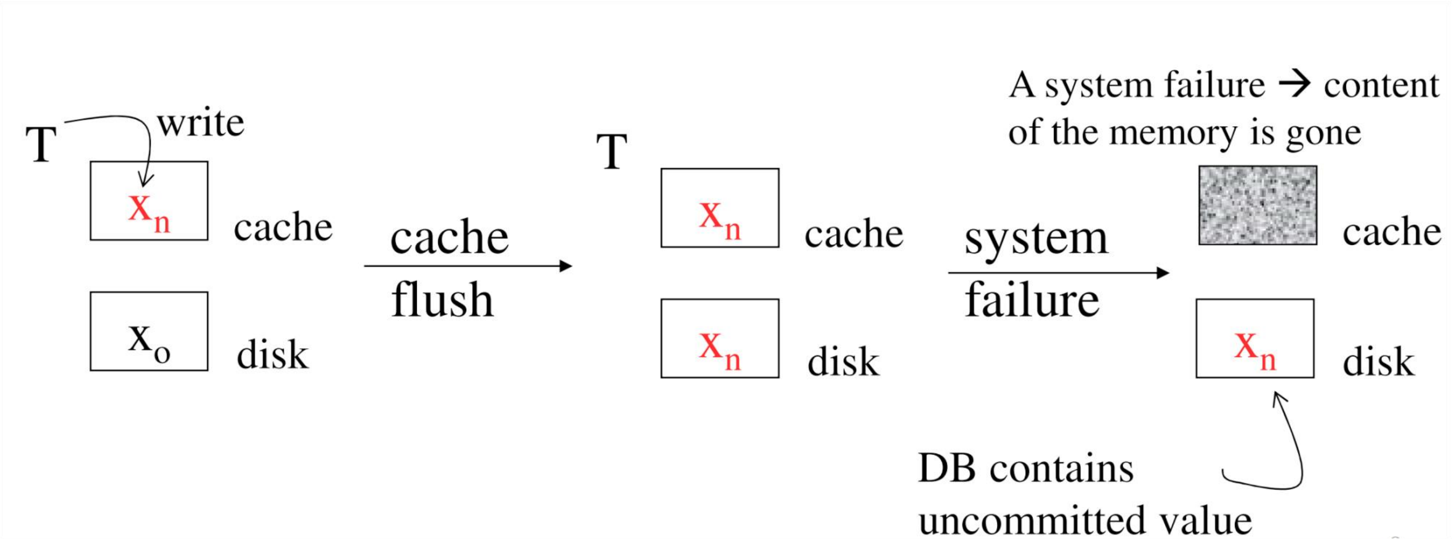
Recovery Manager (RM)

- Recovery manager: ensure ‘A’ and ‘D’ in the database *regardless of system failure*
  - No effects of uncommitted ones
  - All effects of committed transactions
- To abort a transaction *T*, the DBMS must remove all effects of *T*
  - Undo the updates to data by *T*
  - Abort other transactions that have read values written by *T*



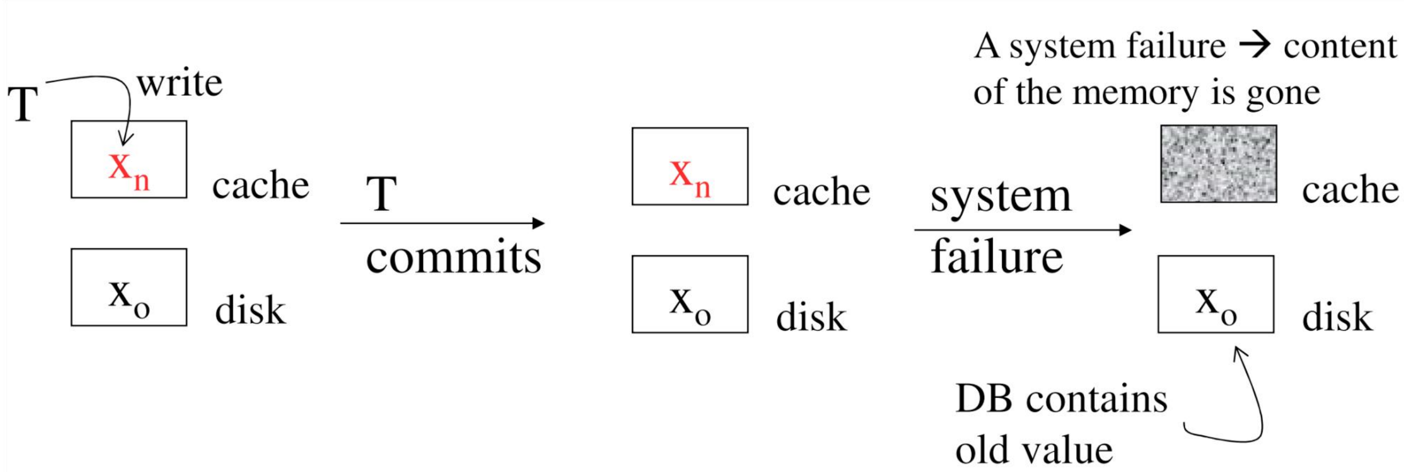
The Job of RM

- *Cache*: used to speed up data accesses
- A system failure may leave the disk
  - with values written by **uncommitted transactions**, or
  - without the values written by committed ones



The Job of RM

- *Cache*: used to speed up data accesses
- A system failure may leave the disk
  - with values written by uncommitted transactions, or
  - without the values written by **committed ones**



The Job of RM

Need **Recoverable** and Preferably **Cascadeless** Schedule

- As we have discussed in Lecture 3
- *Unrecoverable schedule* is bad
  - If T_1 is aborted in future, then the value read by T_2 becomes incorrect!
  - But we cannot abort T_2 anymore

T_1	T_2
Write(x, 50)	
	Read(x)
	Commit
Abort	

- *Cascading abort* wastes system resource
  - Aborting a transaction causes other affected transactions to be aborted
    ➡ then cause further aborts ……

T_1	T_2
Write(x, 50)	
	Read(x)
……	……

The Job of RM

- *Dirty Read*: a read of a data item written by an active transaction

- How to ensure a cascadeless schedule?
  - **Delay** each Read(x) until all transactions that have issued a Write(x) have either aborted or committed
  - Then, every transaction reads only values that were written by committed transactions

T_1	T_2
Write(x, 50)	
……	
Commit/Abort	
	Read(x)
	……

- No dirty read ⇒ cascadeless ⇒ recoverable

Avoid Dirty Write

Initial: x=10

before: 10

T_1	T_2
Write(x, 50)	
	Write(x, 20)
Abort	
	Commit

before: 50

before: 10

T_1	T_2
Write(x, 50)	
	Write(x, 20)
Abort	
	Abort

before: 50

- *Dirty Write*: a write of a data item whose current value is written by an active transaction
- *Before image*: the value of a data item x just before a write(x) operation is executed
- To abort a transaction *T*, can we restore all the before images of all executed write operations in *T*?
  - Answer: No
- How to avoid this problem?
  - **Delay** each write(x) until all transactions that have written x are either committed / aborted

Strict Schedule

- *Strict schedule*: has no dirty reads or dirty writes
- From now on, we assume that the concurrency controller produces *strict schedules*
  - To ensure that, the DBMS **delays** both reads and writes for *x* until all transactions that have written *x* are committed / aborted

T_1	T_2
Write(x, 50)	
Commit	
	Can read / write x now
	……

Restart

- The *last committed value* of a data item x
  - The value last written into x in S by a committed transaction (in a schedule S)

- The *system restart* process
  - When the system recovers from a failure, the RM must restore all data items to their last committed values

Lecture Objectives

- The Recovery Manager
- A Simple Recovery Algorithm
- Checkpointing
- Log Buffering

The Log

- After a system failure, RM has to restore the DB based on the log (stored in the disk)

- The **log** contains a *time sequence of events* (i.e., an earlier event appears before a later event)

- A log entry could be:

  Before image

  - <*T*, start>: transaction *T* starts
  - <*T*, *x*, *v1*, *v2*>: transaction *T* updates item *x* from *v1* to *v2* (we call this an *update log record*).

  After image

  - <*T*, commit>: transaction *T* commits
  - <*T*, abort>: transaction *T* aborts

The Log

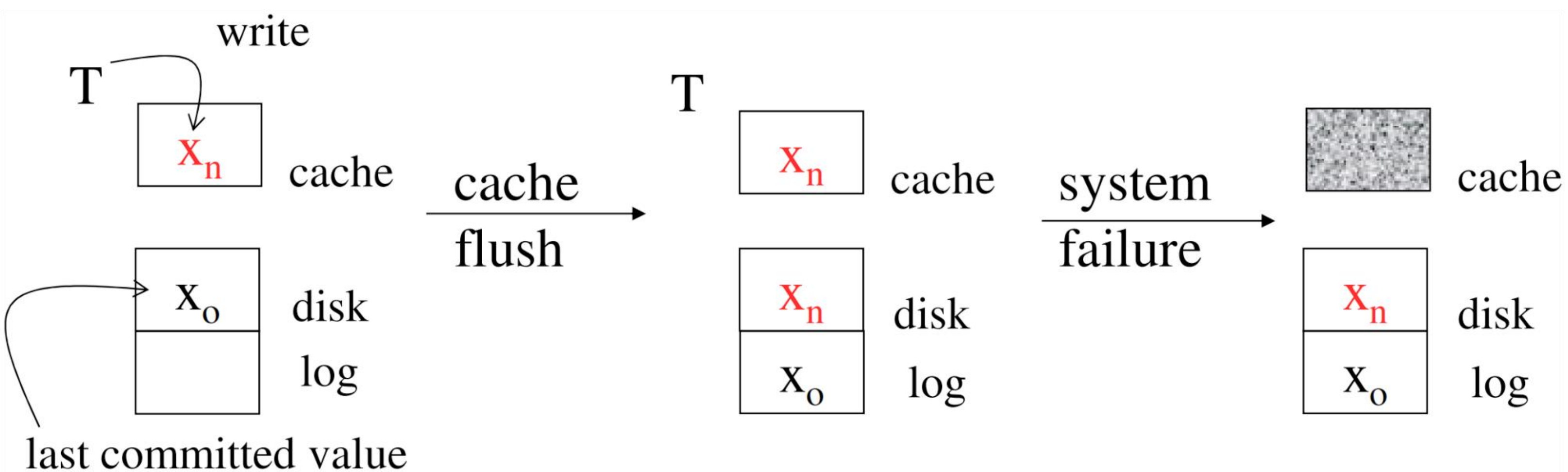
- What can we learn from the log?
  - Committed transactions, uncommitted transactions
  - Values written by those transactions

- Assume that the concurrency controller produces *strict schedules*
  - E.g., if <*T1, x, v1, v2*> precedes <*T2, x, v2, v3*> in the log, then <*T1,* commit> must precede <*T2, x, v2, v3*> in the log

- Advantages of using strict schedules
  - The schedule is recoverable
  - A transaction can be rolled back by restoring before images
  - The last committed value of *x* is written by the last committed transaction that wrote into *x*

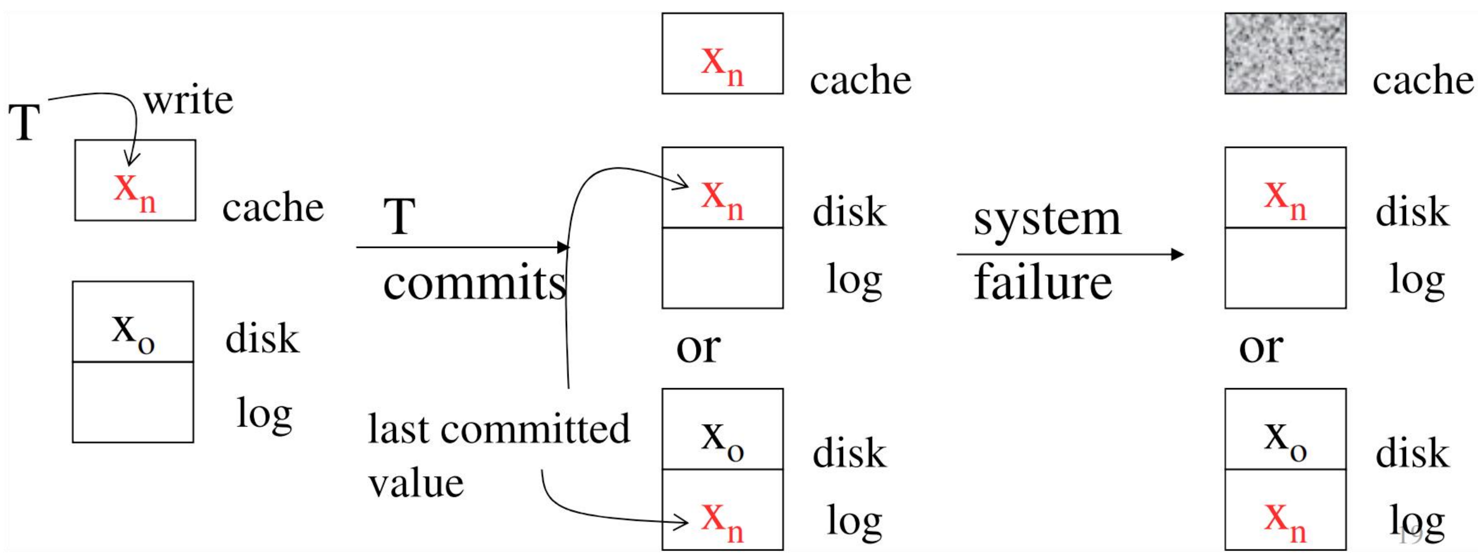
1: Undo Rule

- An RM *requires undo* if it allows an uncommitted transaction to record in the stable database values it wrote (e.g., when the cache overflows)
- *Undo rule*: if the stable database contains the last committed value of x, then that value must be replicated in the log before being overwritten by an uncommitted value



2: Redo Rule

- An RM *requires redo* if it allows a transaction to commit before all the values it wrote have been recorded in the stable database (e.g., when the writes are buffered at the cache)
- *Redo rule*: before a transaction can commit, the value it wrote for each data item must be either in the stable database or in the log



3: Idempotence Rule

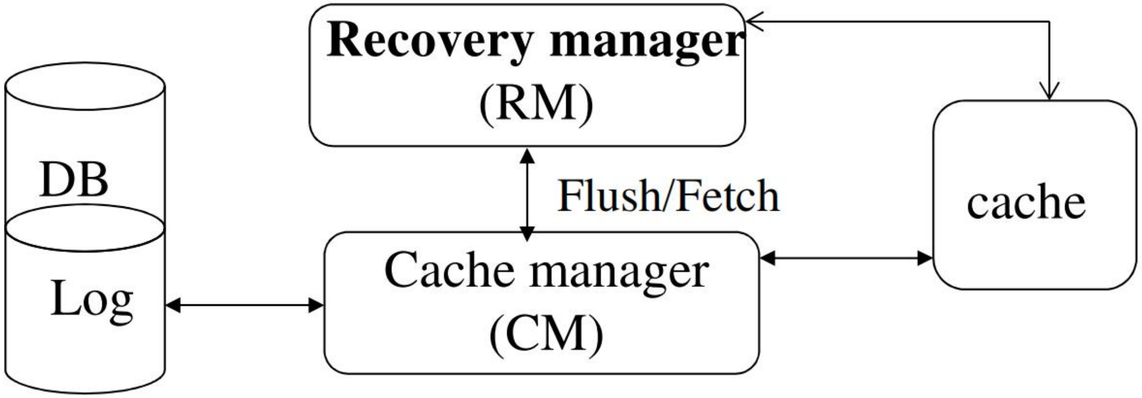
- Rules 1 and 2 ensure that
  - the last committed value of each data item is always available in the disk

- The system may fail again during a system restart

- *Idempotence rule* requires that the effect of a sequence of incomplete system restarts followed by a complete system restart = a complete system restart

A Simple Recovery Algorithm

- Requirement:
  - Before a transaction *T* executes write(x) (on x's copy in cache), a log record <*T*, *x*, *v1*, *v2*> must be appended to the stable log
- In general, the RM does not control when the CM flushes its cache
  - The stable database may contain uncommitted values
  - Committed values may be left in the cache not yet flushed
  - How to deal with these issues?



A Simple Recovery Algorithm

- On a system restart:
  - Initially, "unmark" all data items
  - For each unmarked item x do

    scan the log backward until the last <*T, x, v1, v2*> is found;

    if <T, commit> is in the log then

    set x to v2 (redo);

    else

    set x to v1 (undo);

    mark x;

the last update log record of *x*

if *T* committed, clearly *v2* is the last committed value of *x*

if *T* did not commit, we know that *v1* is the last committed value of *x* because the schedule is *strict*.

log

…
<T, *x*, v1, v2>
…
<T, commit>
…

Lecture Objectives

- The Recovery Manager
- A Simple Recovery Algorithm
- Checkpointing
- Log Buffering

Checkpointing

- The simple algorithm would examine the entire log!
  - Many of the redo's are redundant
  - The updated value of an item might have been flushed to the stable database (by the cache manager)

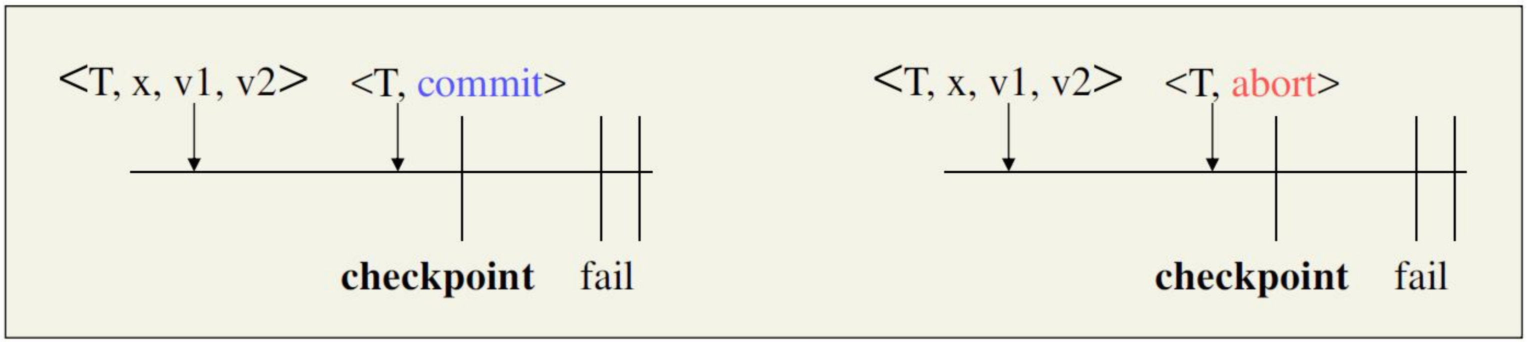
- Why do we use checkpoints?
  - To reduce the number of log records that have to be examined and kept

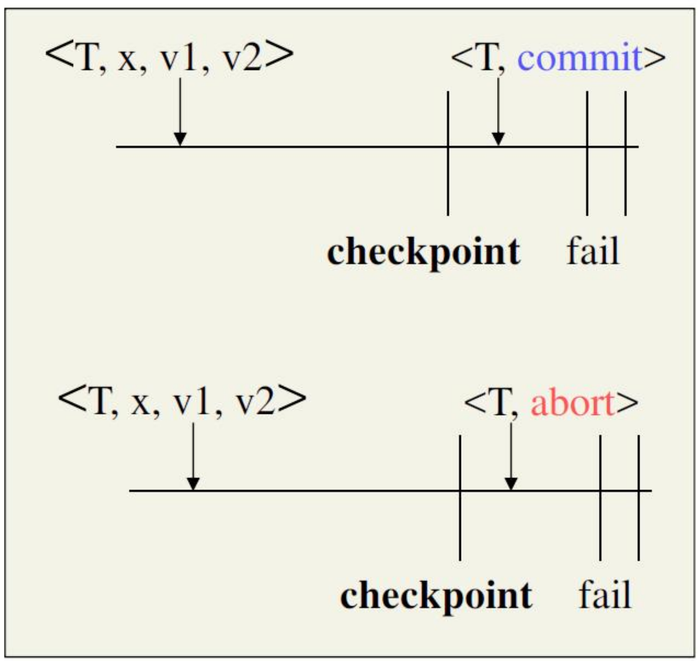
- *Checkpointing*
  - writes information to disk during normal operation ➡ reduce the amount of work for system restart after failure

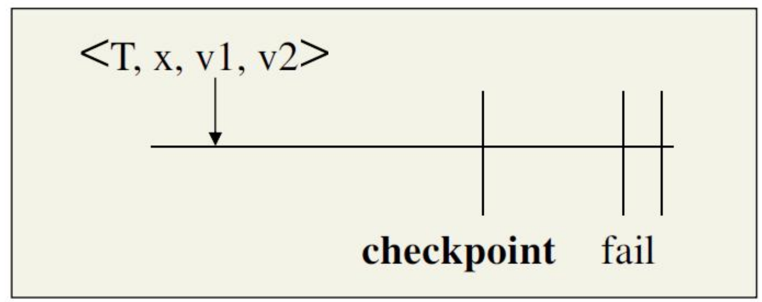
Cache Consistent Checkpointing

- *Cache consistent checkpointing*
  - Block all active transactions
  - Force flush the cache
  - Append a <checkpoint> log record
- After checkpointing, the cache does not contain any last committed values
  - During recovery, we
    - need not redo transactions that are committed before the last checkpoint
    - need not undo transactions that are aborted before the last checkpoint

Cache Consistent Checkpointing

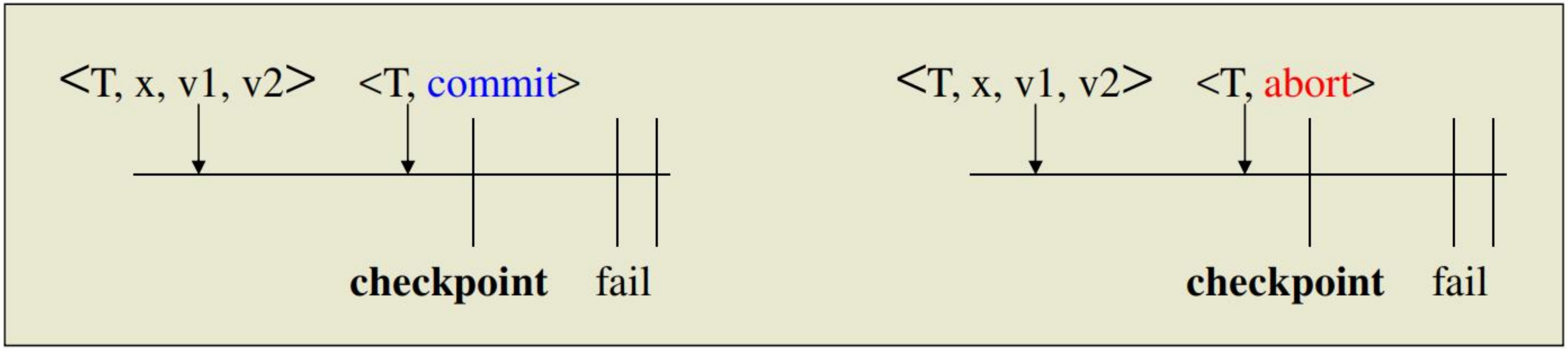






Consider a data item *x* whose last update log record appears before the checkpoint.

Cache Consistent Checkpointing



No need to redo / undo the first 2 cases.

Hence, no need to redo / undo transactions that were terminated before the last checkpoint

Observation

- Suppose that <*T, x, v1, v2*> is the last update log record before the checkpoint record.
  We only need to undo the write if *T* is
  - (1) active at check point and
  - (2) not committed before the system fails.
- Need to include the *list of active transactions L* in the checkpoint log record <checkpoint, *L*>
  - Some transactions in L were not committed before the failure
- No need to keep log records of transactions that terminate before the last checkpoint

The Undo/Redo Algorithm

- Two phases:
  - (1) construct an undo-list and a redo-list
  - (2) undo and redo update log records
- To construct an undo-list and a redo-list:
  - Set undo-list = redo-list = {}
  - Scan the log backward from the last record, do {

    on a <T,commit>: add T to the redo-list;

    on a <T, start>: if T $\notin$ redo-list, add T to the undo-list;

    on a <checkpoint, L>: quit loop;

    }
- For each T in L: if T $\notin$ redo-list, add T to the undo-list

Such transaction started before the checkpoint

The Undo/Redo Algorithm

- To restore the stable database:
  - Scan the log ***backward*** from the last record, do {

    on a <*T*, *x*, *v1*, *v2*>:

    if $T \in$ undo-list, set *x* to *v1*;

    }

    until <*,start> log records of all transactions ∈ undo-list are found
- Scan the log ***forward*** from the last <checkpoint> to the end of log, do {

  on a <*T*, *x*, *v1*, *v2*>:

  if $T \in$ redo-list, set *x* to *v2*;

  }

The Undo/Redo Algorithm

DBGroup

- Why scan forward the log for redo's?
- Why scan backward the log for undo's?
- Example: (x = 10, initially)

T_i	T_j
write(x,20)	
commit	
	write(x,30)
	commit

log

<T_i, x, 10, 20>
<T_i, commit>
<T_j, x, 20, 30>
<T_j, commit>

failure

T_i
write(x,50)
write(x,60)
abort

log

<T_i, x, 10, 50>
<T_i, x, 50, 60>
<T_i, abort>

failure

The Undo/Redo Algorithm

- Why perform undo's before redo's?
- Example: (x = 10, initially)

T_i	T_j
write(x,20)	
abort	
	write(x,30)
	commit

failure

log

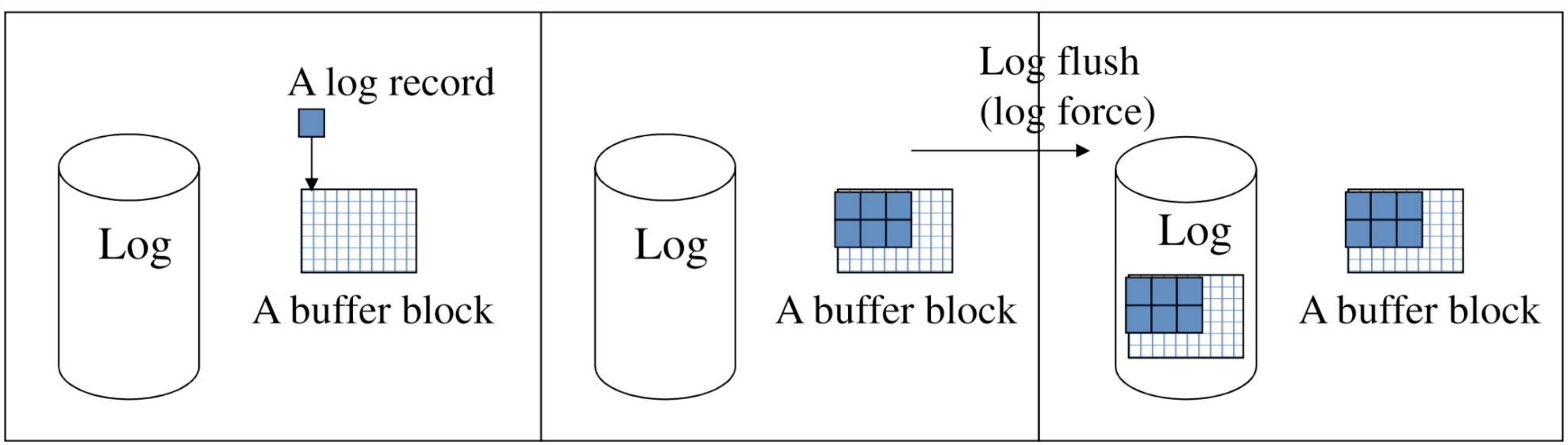
<T_i, *x*, 10, 20>
<T_i, abort>
<T_j, *x*, 10, 30>
<T_j, commit>

Lecture Objectives

- The Recovery Manager
- A Simple Recovery Algorithm
- Checkpointing
- Log Buffering

Log Buffering

- So far, we assume that each log record is written directly into the stable log
- For performance reason, we
  - buffer the log records in memory, and
  - write multiple log records to disk on one single write

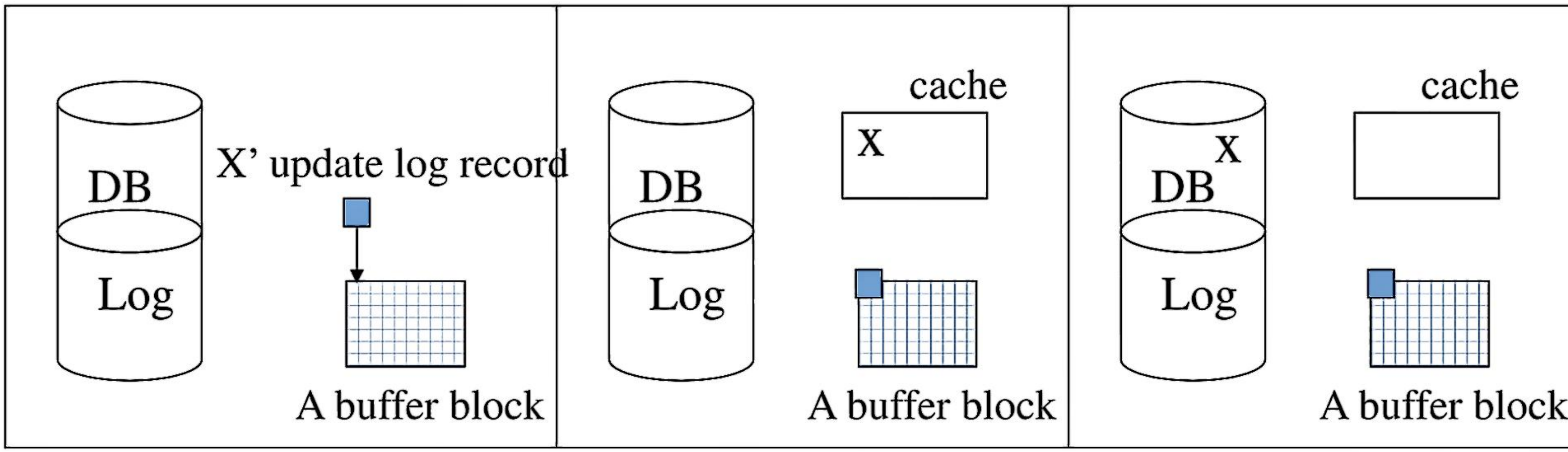


Undo/Redo Rules & Log Buffering

- Would log buffering violate the undo/redo rules?
- If we insist that a <*T*, commit> record must be written to the stable log before we consider *T* commits, then …
- This satisfies the redo rule. Why?
  - All update log records of *T* precedes <*T*, commit> in the log
  - If <*T*, commit> is flushed to the stable log, so have all of *T*'s update log records
- This may violate the undo rule. Why?
  - An update on an item x may have already been applied to the stable database when
    the update log record <*T, x, v1, v2*> is still in the log buffer

Log Buffering

X's value is updated in the cache copy



<T,x,v1,v2> added to the log buffer

X's cache value is flushed to the DB

Write Ahead Logging

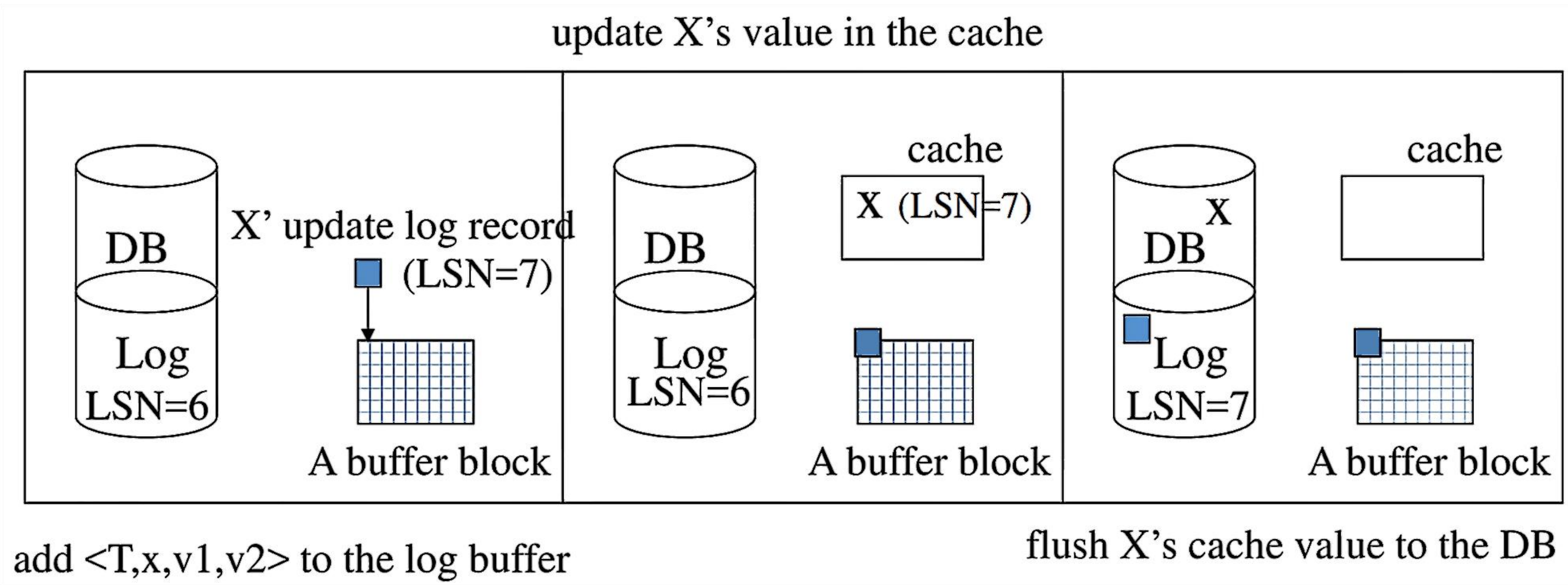
- To enforce the undo rule, we use the *write ahead logging* (WAL) policy
  - Before writing a block of data (in main memory) to the stable database,
    we must write all log records for that block to the stable log

- Solution to ensure WAL: associate each log record with a *log sequence number* (LSN)

Using LSN

In the cache, each *dirty* data item *x* is marked with the label LSN(*x*) of the corresponding update log record <*T, x, v1, v2*>

The system also keeps LSN(stable):
the LSN of the latest log record flushed to the stable log.



Before flushing the cache copy of *x* to stable DB,
the system checks if LSN(*x*) > LSN(stable).
If yes, the system force flush the log buffer

Conclusions

- An RM ensures that the database contains
  - All effects of committed transactions and
  - No effects of the aborted ones
- A simple log-based recovery algorithm
- An efficient log-based recovery algorithm using
  - Checkpointing and log buffering

谢谢!

DBGroup @ SUSTech
Dr. Bo Tang (唐博)
tangb3@sustech.edu.cn