Crash Recovery

Crash Recovery

- Introduction
  - Storage
  - Failure
  - Recovery
  - Logging
- Undo Logging
- Redo Logging
- ARIES

Storage Types

- Volatile storage
  - Main memory
  - Cache memory
- Nonvolatile storage
  - Stable storage
    - Online (e.g. hard disk, solid state disk)
    - Transaction logs are written to *stable storage*, which is guaranteed to survive system crashes and media failures
  - Offline – optical, flash drives, removable hard drives etc.
    - deprecated - floppy disk, zip drives, tape, punch cards …

Failure Examples

- A small list of potential problems
  - User tries to enter an incorrect msp — prevented by primary key
  - A disk crashes — recover with a RAID scheme
  - Power goes out while transactions are being entered — Use transaction log to recover
  - An explosion destroys the site where the DB is located — Use off-site backup to recover
  - Aliens destroy the planet to make way for an interstellar bypass — ???

Types of Failure

- System crashes
  - Results in data loss of all data in volatile storage
  - Possible causes include power failures, operating system failures, etc.
- Media failures (disk crash)
  - Results in loss of online (non-volatile) data and volatile data
  - Possible causes include damages to the storage media and human error (e.g. accidentally erasing the disk)

Transactions

- A database is assumed to be in a consistent state before a transaction is processed
  - If a transaction executes in its entirety in isolation then the DB is still consistent after its execution
- If a transaction is only partially executed the DB may not be consistent
  - Transactions should be *atomic* but
  - May be interrupted by a system failure

Crash Recovery

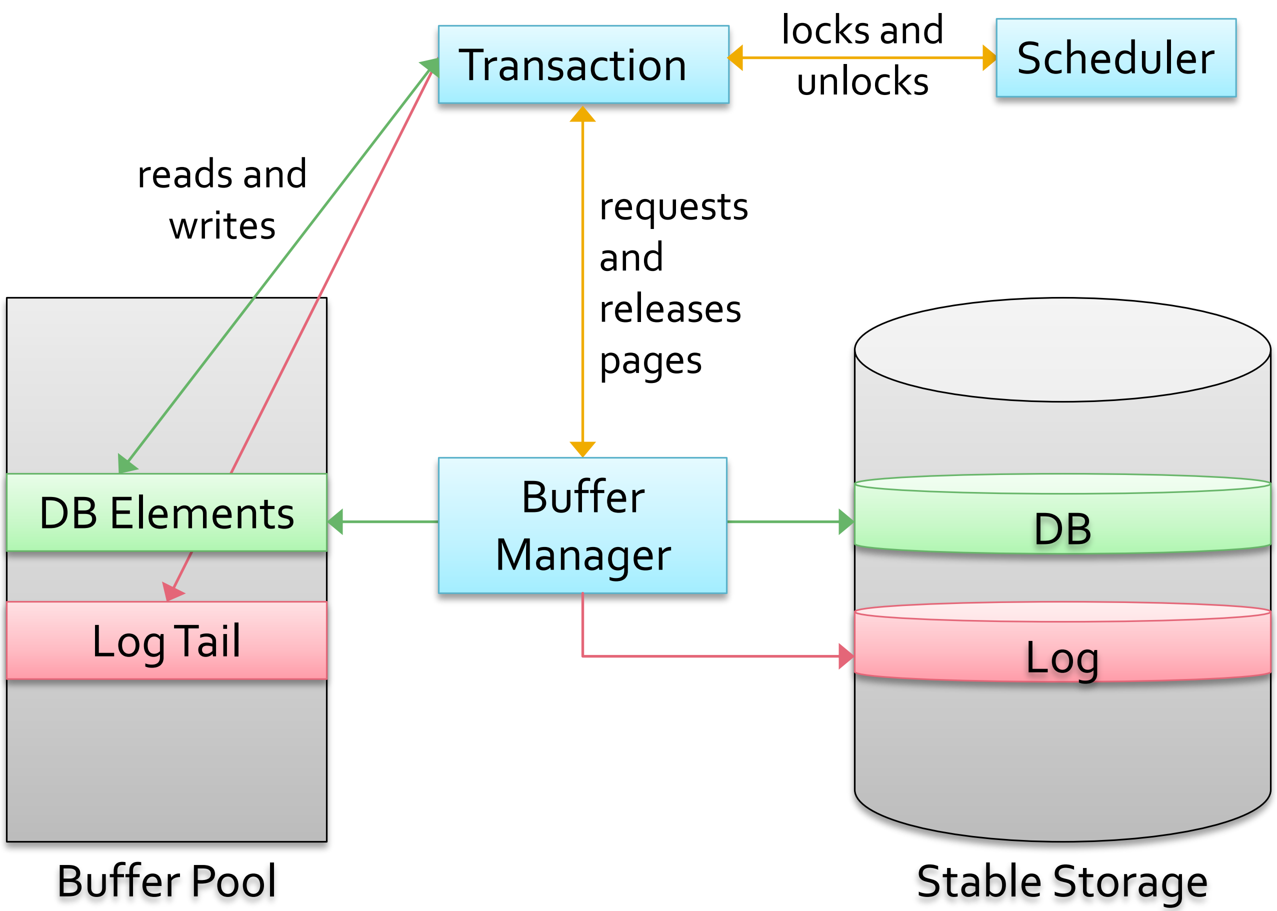
Executing Transactions

- Transactions involve reading or writing a database element (or both)
  - Adding money to a bank account
  - Altering a student's GPA
  - Registering for a course
  - Changing an address
- This occurs in main memory
  - The element must be retrieved from disk and then
  - Written back to disk so that the transaction is *durable*

Example Transaction

Action	Memory A	Memory B	Disk A	Disk B
R(A)	250		250	500
A = A + 100	250		250	500
W(A)	350		250	500
R(B)	350	500	250	500
B = B - 100	350	500	250	500
W(B)	350	400	250	500
Output(A)	350	400	350	500
Output(B)	350	400	350	400

Normal Transaction Execution



Recovery Manager

- The *recovery manager* is responsibility for ensuring *atomicity* and *durability*
  - Atomicity – undo actions of aborted transactions
  - Durability – ensure the actions of committed transactions survive failures
- These tasks should be carried out efficiently
  - Recovery time and overhead should be minimized
    - Given that crashes do not occur frequently
  - There is a trade-off between recovery time and normal running time
- Log transactions to allow recovery

T1	T2
R(A)	
W(A)	
	R(B)
R(C)	
W(C)	
	W(B)
	R(D)
Commit	
CRASH!!	
REDO	UNDO

Transaction Log

- The log is a history of executed transactions
  - A file of records stored in stable storage
  - The most recent part of the log is kept in main memory and periodically forced to stable storage

the log tail

- Each log record has a unique id, called the log sequence number (LSN)
  - LSNs are assigned in sequential order
  - A record is written for each action of transaction
- Every DB page contains the LSN of the most recent log record that described a change to that page

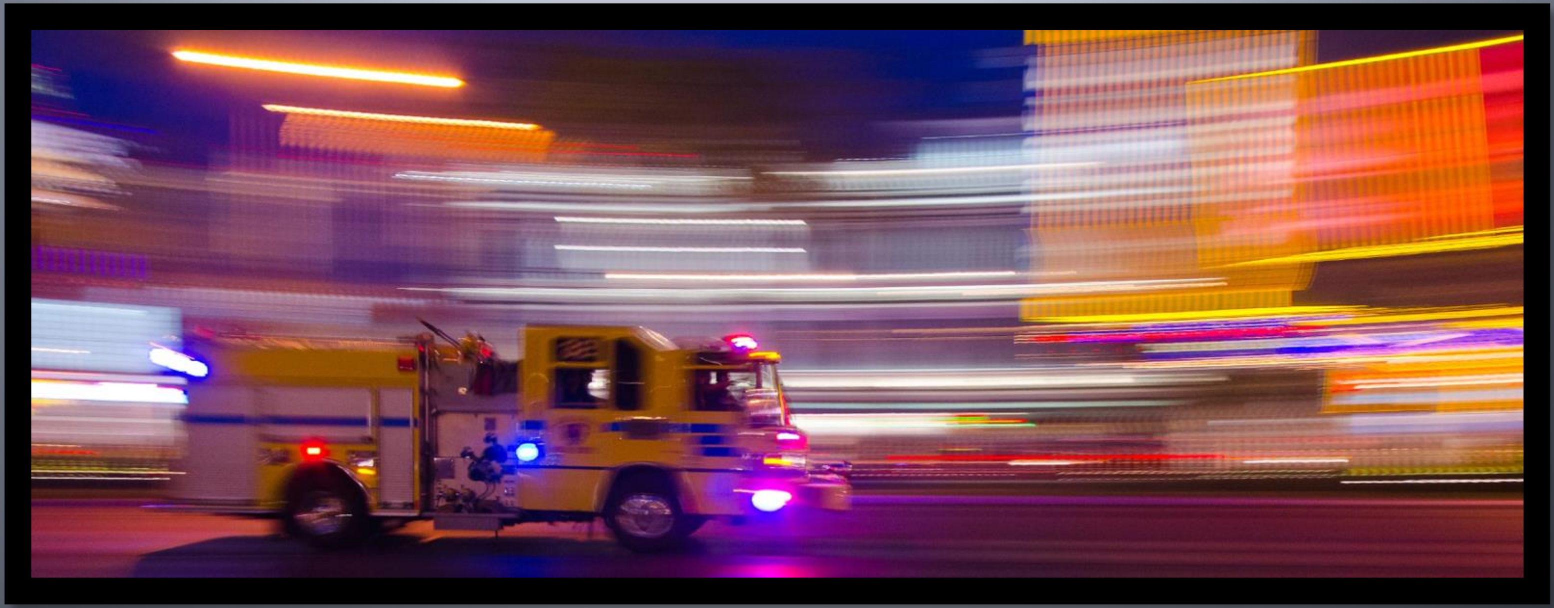
Stable Storage

- Transaction logs should be maintained in nonvolatile storage (disk or tape)
  - Data written to stable storage is safer
    - It is impossible to guarantee safety but it is possible to make data loss very unlikely
  - RAID systems can ensure that a single disk failure will not result in data loss
  - Mirrored disks can also be used to minimize data loss
    - If copies of the log are made, one disk can be stored remotely to mitigate against the effects of fire or other disasters

Stealing Frames, Forcing Pages

- It is possible to write a transaction's changes to a DB object to disk before the transaction commits
  - If buffer manager chooses to replace the frame containing the object
    - Note that the frame must have been unpinned
  - Referred to as *stealing* the frame
    - From the uncommitted transaction
- When a transaction commits, its changes may need to be immediately written to disk known as *forcing*
  - Ensuring that the transaction is preserved

Recovery Schemes



Undo Logging

- Undo logging is a recovery scheme that undoes the work of incomplete transactions after a crash
  - It does not *redo* transactions
- The transaction log contains the following records
  - <*start T*> indicates that the transaction, *T*, has begun
  - <*update T*, *X*, *v*> records changes made by *T*
  - <*commit T*> indicates that *T* has completed
    - *T* will not make any more changes to the DB, and
    - Any changes made by *T* should appear on disk
  - <*abort T*> indicates that *T* could not complete
    - Any changes made by *T* should not appear on disk

Update Records

- An undo log's *update* records track DB changes, the records are triples <*T*, *X*, *v*>, where
  - Transaction *T*, has changed database element *X*, and the previous value of *X* was *v*
- Changes reflected by update records normally occur in memory, and may not yet be recorded on disk
  - The log record is in response to a *write* action, not
  - An *output* action, which outputs data to a disk
- The undo log does *not* record the *new* value written by an update

Object A had value 123

T1 adds 100 to object A

<update T1, A, 123>

Undo Logging Rules

- U_1 – If *T* modifies *X*

  i.e. *T* writes *X*

  - The update record <*T*, *X*, *v*> must be written to disk *before* the new value of *X* is written to disk
- U_2 – If *T* commits
  - The commit log record must be written to disk *after* all the changes of *T* are written to disk
    - As soon as possible after *T*'s last change has been written to disk
- Both rules necessitate that pages are *forced* to disk
  - The log manager must have a *flush-log* command that tells the buffer manager to write the log to disk, and
  - The transaction manager must be able to make the buffer manager output pages to disk

T1	log
R(A)	
W(A)	
	update(T1, a, …)
output(A)	
commit	
	commit(t1)

Undo Logging Recovery

- In the event of a system failure a transaction may not have executed atomically
  - Some changes made by the transaction have been written to disk and others are not
  - The DB may be in an inconsistent state
- The recovery manager uses the log to restore the DB to a consistent state
  - Assume the recovery manager considers the entire log
    - This is not an efficient approach, and most systems use checkpoints
  - All incomplete transactions are undone

T1
R(A)
W(A)
CRASH!!
R(B)
W(B)

Incomplete Transactions

- A transaction is incomplete if it has a *start* record on the log but no matching *commit* record
  - Any changes made by such transactions are reversed
  - Transactions with a commit record on the log must have been written to the disk (from rule U_2)
- Update records are used to reverse transactions
  - If a transaction made a change to the DB there must be an update record on the log (from rule U_1)
  - Changes can be reversed by rewriting each data object X with the value v recorded in the update record

Recovery Process

- Review log to find incomplete transactions
- Proceed backwards through the log and for each update record <*T*, *X*, *v*>
  - If *T* has a *commit* record, do nothing, otherwise
  - *T* is incomplete so change the value of *X* to *v*
- Write an *abort* record to the log for each incomplete transaction
- The process must go backwards through the log to ensure that the DB is in the correct state

Effect of *T* must be on disk – U_2

To record that the transaction needs to be processed again

Example: Undo Log

- Transaction
  - Reads *A*, doubles it
  - Reads *B*, adds 13 to it
- Key
  - *MM* = main memory
  - *DB* = stable storage
  - Log records
- *output* actions write main memory to disk
- *flush* log actions writes log to disk

Action	MM(A)	MM(B)	DB(A)	DB(B)	Log
					start T
READ (A)	12		12	4	
WRITE (A)	24		12	4	*T*, *A*, 12
READ (B)	24	4	12	4	
WRITE (B)	24	17	12	4	*T*, *B*, 4
FLUSH LOG	U_1 – write log updates before storing DB changes				
OUTPUT (A)	24	17	24	4	
OUTPUT (B)	24	17	24	17	
	U2 – write commit log records only after changes are written				*commit T*
FLUSH LOG					

Undo Logging Recovery

- If there is a crash after a transaction's *commit* record has been stored in the log no recovery is needed
  - Because of rule U_2, the changes to *T* must have been written to the disk before the commit record was made
- If a crash occurred between a transaction's *start* and *commit* log records, it must be undone
  - This is achieved by writing the previous values (*v*) in the update records to the database objects
  - As undo logging only undoes incomplete transactions it is not necessary to record new values in the log

Checkpoints

- The undo recovery scheme requires that the entire log is read during recovery
  - This gets increasingly inefficient as the log gets larger, and
  - Reads older, committed, transactions to no purpose
- Once a commit log record is written to disk the log records of the transaction are not needed
  - However, it is not possible to delete the entire log whenever a commit record is written
  - Since there may be log records relating to other, active, transactions which would be required for recovery

Simple Checkpoints

- To indicate that all preceding transaction have been committed a *checkpoint* can be inserted in the log
  - Only the recovery log records after the last checkpoint have to be used
- The simplest way to insert a checkpoint is
  - Stop accepting new transactions
  - Wait until all active transactions commit or abort, and have written their commit or abort records to the log
  - Flush the log (write it to stable storage)
  - Write a *checkpoint* record to the log
  - Start accepting transactions again

But this has a negative effect on throughput

Non-quiescent Checkpoints

- If the system is shut down to insert a checkpoint it may appear stalled to users
  - *Non-quiescent checkpointing* allows processing to continue as a checkpoint is created
- To create a non-quiescent checkpoint
  - Write a *start checkpoint* log record
    - The log record includes a list (T_i, …, T_k) of active transactions that have not yet committed
  - Wait until T_i, …, T_k commit, while still allowing other transactions to start
  - Write an *end checkpoint* log record once T_i, …, T_k have completed

LSN	Trans ID	Type
101	12	start
102	13	start
103	12	update
106	12	commit
107	begin checkpoint (13)	
108	13	update
109	14	start
112	14	update
113	15	start
114	13	commit
115	end checkpoint	
116	14	update
117	15	update
118	14	commit

Recovery with Checkpoints

- With a non-quiescent checkpoint system the log is scanned backwards from its *end*
  - Undoing incomplete transactions
- If an end checkpoint is found (1)
  - All incomplete transactions must have begun after the previous start checkpoint
  - End scan when start checkpoint is reached
- If a start checkpoint is found first (2)
  - The crash happened during the checkpoint
  - Scan back to the first incomplete transaction specified in the start checkpoint record

LSN	Trans ID	Type
101	12	start
102	13	start
103	12	update
106	12	commit
107	begin checkpoint(13)	
108	13	update
109	14	start
112	14	update
113	15	start
114	13	commit
115	end checkpoint	
116	14	update
117	15	update
118	14	commit

CRASH$_2$ → (between LSN 112 and 113)

CRASH$_1$ → (between LSN 116 and 117)

Redo Logging

- Undo logging requires that changes are written to disk before a transaction is committed
  - Removing this requirement would reduce disk IOs
  - The need for immediate stable storage of committed changes can be avoided using *redo logging*
- Undo and redo logging have key differences
  - Redo logging ignores incomplete transactions, and repeats changes made by committed transactions
  - Redo logging requires that commit log records are written to disk *before* any changed values are written to the DB
  - Redo update records store the *new* values of DB objects

Redo Logging Rule

- R_1 Before changing any DB object on disk, all log records relating to the change must appear on disk
  - Including the *update* record and the *commit* record
  - The transaction can only be written to disk when it is complete
- Redo log update records appear the same as undo log updates (<*T*, *X*, *v*>)
  - However, the value, *v*, does *not* record the value of *X* prior to the update
  - It records the *new* value of *X* – after the update

T1	log
R(A)	
W(A)	
commit	
	update(T1, a, …)
	commit(t1)
output(A)	

Redo Logging Recovery

- Unless the log contains a *commit* record, changes made by a transaction have not been written to disk
  - Therefore incomplete transactions can be ignored
  - Transactions *with* a commit record may not have been written to disk
- Recovery with a redo log is as follows
  - Identify the committed transactions
  - Scan the log *forward* from the *start*, for each update record
    - If *T* is not a committed transaction, do nothing
    - If *T is* committed, write the value *v* for DB object *X*
  - Write an *abort* record for each incomplete transaction

Redo Log Checkpoints

- A commit log record does not guarantee that the corresponding transactions are complete
  - It is necessary to keep track of which main memory changes are dirty (changed but not written), and
  - Which transactions modified buffer pages
- The redo log checkpoint process is as follows
  - Write a *start checkpoint* log record
    - The log record includes a list (T_i, …, T_k) of active transactions that have not yet committed
  - Write all changes in buffers relating to committed transactions
  - Wait for T_i, …, T_k *to commit*
  - Write an *end checkpoint* log record

Recovery with Checkpoints

- Start and end checkpoints limit the examination of the log during a recovery
- If the last checkpoint is an end checkpoint
  - Redo transactions in the list (T_i, ..., T_k), and
  - Committed transactions started after the start checkpoint
- If the last checkpoint is a start checkpoint
  - Scan back to the previous start checkpoint for that checkpoint's list of transactions in the list and
  - Redo all transactions in that list and other committed transactions that started after the prior start checkpoint

ARIES

Introduction to ARIES

- Algorithm for Recovery and Isolation Exploiting Semantics (ARIES)
  - ARIES is used by the recovery manager in many DBMS
- There are three principles behind ARIES
  - Write-ahead logging
  - Repeating history during redo
  - Logging changes during undo

- ARIES has *steal, no force* buffer management
  - Note that log records are forced to disk

Write-Ahead Log Protocol

- A log record for an update must be forced to disk before the change is processed
  - That is, before the dirty page is written to disk
  - To ensure that the transaction can be properly *undone* in the event that it is aborted
- All log records must be stored in stable storage before a commit log record is written
  - If they are not, they must be forced to the disk before (not at the same time as) the commit log record
  - This is necessary to ensure that it is possible to *redo* a committed transaction after a crash

Log Actions

- Updating a page
  - An update record is added to the log tail, page LSN set to LSN
- Transaction commit
  - Force-write commit log record containing the transaction ID
- Transaction abort
  - Write abort log record and commence undo
- Transaction end
  - Add end log record once abort or commit process is complete
- Undoing an update
  - Write a *compensation log record* (*CLR*) and undo update

Transaction Log Records

- All log records have the following fields
  - *previous LSN* – LSN of the transaction's previous record
  - *transaction ID* – the ID of the transaction being logged
  - *type* – the type of the log record
- Update log records have additional fields
  - *page ID* – the page being modified by the update
  - *length* (in bytes) and *offset* – refers to the data page
  - *before-image* – changed bytes before the change
  - *after-image* – changed bytes after the change
  - An update log record with both before and after images can be used to redo or undo a change

Compensation Log Records

- A Compensation Log Record (*CLR*) is written just prior to undoing the change made in an update log record
  - Either as part of the undo process of crash recovery, or
  - When a transaction is aborted in normal operation
- A *CLR* describes the action taken to undo its update, and includes
  - An *undo Next LSN* field, which is the *LSN* of the next log record to be undone to undo the entire transaction
  - The *LSN* in the *previous LSN* field of the update log record
- *CLR*s contains information needed to redo the *CLR*
  - Are used in the event of a crash during recovery

ARIES Data Structures

- The transaction table contains an entry for each active transaction
  - *Transaction ID*
  - *Status* – in progress, committed, or aborted
  - *last LSN* – the *LSN* of the transaction's most recent record
  - Other information not related specifically to recovery

transaction ID	Status	last LSN
…		

- The dirty page table (*DPT*) contains an entry for each dirty page in the buffer pool
  - *first LSN* – the first log record that made that *page* dirty
    - The earliest log record that might have to be undone

page ID	first LSN
…	

- Each page in the DB includes a *page LSN*
  - The log sequence number for the last update to that page

Checkpoints in ARIES

- A *begin checkpoint* shows the checkpoint start
- An *end checkpoint* contains the current contents of transaction and dirty page tables
  - Transaction processing continues while the end checkpoint is being built
  - Therefore the transaction and dirty page table are accurate at the time of the *begin* checkpoint
- After the end checkpoint is written to stable storage, a *master* record is also written
  - Contains the *LSN* of the begin checkpoint

ARIES Recovery Scheme

- After the system has crashed it is restarted
  - No user program is allowed to execute
- The recovery manager executes a three phase recovery
  - *Analysis* – determines the extent of the recovery, and which transactions need to be redone or undone
  - *Redo* – all changes to pages that may have been dirty at the time of the crash are redone
    - In the order in which they occurred
  - *Undo* – undoes the change of all transactions that were active at the time of the crash
    - Starting with the most recent change

i.e. not committed

Analysis Phase

- The analysis phase performs three tasks
  - Scans the log to find where to start the redo pass from
  - Determines the pages in the buffer pool that were dirty at the time of the crash
  - Identifies the transactions that were active at the time of the crash and that must be undone
- Starts at the most recent begin checkpoint log record
  - The contents of the dirty page table and transaction table are set to the copies in the end checkpoint
  - The log is scanned forward from the begin checkpoint

Analysis – Log Scan

- If an end log record for a transaction is found
  - The transaction is removed from the transaction table
  - Because it is no longer active
- If any other log record for a transaction is found
  - The transaction is added to the transaction table if not there
  - The *lastLSN* field is set to the *LSN* of the current log record
  - If the log record is a commit record, the transaction's status is set to *commit*, otherwise it is set to *undo*
- If a log record affects a page that is not in the dirty page table, the page *ID* and *LSN* are inserted into it

Analysis Phase Example

- DPT and transaction tables are empty
  - At last check point
- Analysis phase
  - Build DPT and
  - Transaction table

page ID	first LSN
500	101
600	102
700	103
550	105

T ID	Status	last LSN
T1	commit	108
T2	undo	107
T3	undo	106

LSN	Trans ID	Type	Page ID	Prev. LSN
	begin checkpoint – empty			
101	T1	update	500	-
102	T2	update	600	-
103	T2	update	700	102
104	T1	update	600	101
105	T3	update	550	-
106	T3	update	550	105
107	T2	update	500	103
108	T1	commit	write log to disk	
109	T1	end	not written to disk yet	
110	T2	update	700	107
111	T4	update	800	-
	CRASH			

Redo Phase

- The redo phase starts with the log record with the smallest *first LSN* of all pages in the *DPT*
  - From that page redo scans *forwards* to the end of the log
- For each re-doable log record (*update* or *CLR*) the action must be redone *unless*
  - The affected page is not in the *DPT*
    - Why would a page not be in the DPT?

    because the page has been written to disk
  - The affected page is in the *DPT*, but the *first LSN* for the entry is greater than the *LSN* of the record being checked

    already on disk
  - The *page LSN* is greater than or equal to the DPT record *LSN*
    - This last case must be discovered by checking the disk

Redo – Checking the Disk

- The third redo condition compares the *page LSN* of a dirty page to the *LSN* of the log record
  - This entails fetching the page from disk
  - This condition is checked last to avoid accessing the disk where possible
- Assume that the log contains three records that access the same page on the *DPT*
  - The page's *first LSN* is 235, and the three records *LSN's* are
    - 128 – don't need to check disk as 128 < 235, no redo required
    - 235 – check the disk, assume its *page LSN* is 235, no redo is required
    - 278 – check the disk, redo is required

Redo Process

- If an action has to be redone
  - The logged action is reapplied
  - The *page LSN* on the page is set to the *LSN* of the redo log record, no additional log record is created
- At the end of the redo phase
  - End records are written for all transactions with a commit status, which are removed from the transaction table
- Redo reapplies updates of *all* transactions
  - Including transactions which have not committed
  - The *undo* process will undo the actions of all transactions that were active when the crash occurred

Redo Phase Example

- Redo starts with 101
  - And redoes all transaction actions
  - 101 to 107
  - In that order
  - Write end for T1
    - And remove from T table

LSN	Trans ID	Type	Page ID	Prev. LSN
	begin checkpoint – empty			
101	T1	update	500	-
102	T2	update	600	-
103	T2	update	700	102
104	T1	update	600	101
105	T3	update	550	-
106	T3	update	550	105
107	T2	update	500	103
108	T1	commit	write log to disk	
109	T1	end	not written to disk yet	
110	T2	update	700	107
111	T4	update	800	-
	CRASH			

page ID	first LSN
500	101
600	102
700	103
550	105

T ID	Status	last LSN
T2	undo	107
T3	undo	106

Undo Stage

- The undo phase scans *backwards* through the log
- The undo process starts with the transaction table
  - The table shows all transactions that were active, and
  - Includes the *LSN* of the most recent log record for each of the transactions
  - These transactions are referred to as *loser transactions*
- All the actions of losers need to be undone
  - In the reverse order to which they appear in the log
- The undo process starts with the set of *last LSN* fields from the transaction table

Undo Process

- Choose the largest *LSN* value in the set of *last LSNs*
- If the record is an update
  - Write a *CLR* and undo the action
  - Add the *previous LSN* in the update log record to the set
- If the log record is a *CLR* and the *undo Next LSN* value is not null
  - Add the *undo Next LSN* to the set
  - Otherwise write an end record for the transaction
- When the set of actions is empty the undo phase and the restart process are complete

Undo Phase Example

- Undo undoes
  - 107
  - 106
  - 105
  - 103
  - 102
    - In that order

page ID	first LSN
500	101
600	102
700	103
550	105

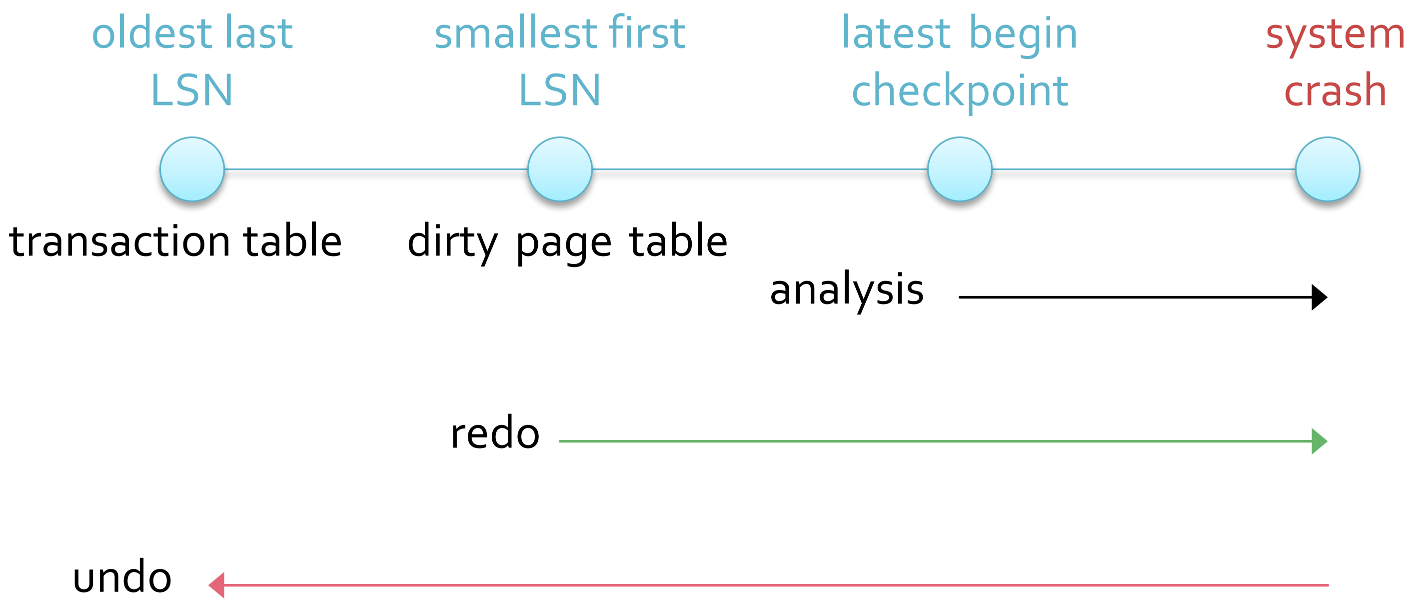
T ID	Status	last LSN
T2	undo	107
T3	undo	106

LSN	Trans ID	Type	Page ID	Prev. LSN
	begin checkpoint – empty			
101	T1	update	500	-
102	T2	update	600	-
103	T2	update	700	102
104	T1	update	600	101
105	T3	update	550	-
106	T3	update	550	105
107	T2	update	500	103
108	T1	commit	write log to disk	
109	T1	end	not written to disk yet	
110	T2	update	700	107
111	T4	update	800	-
	CRASH			

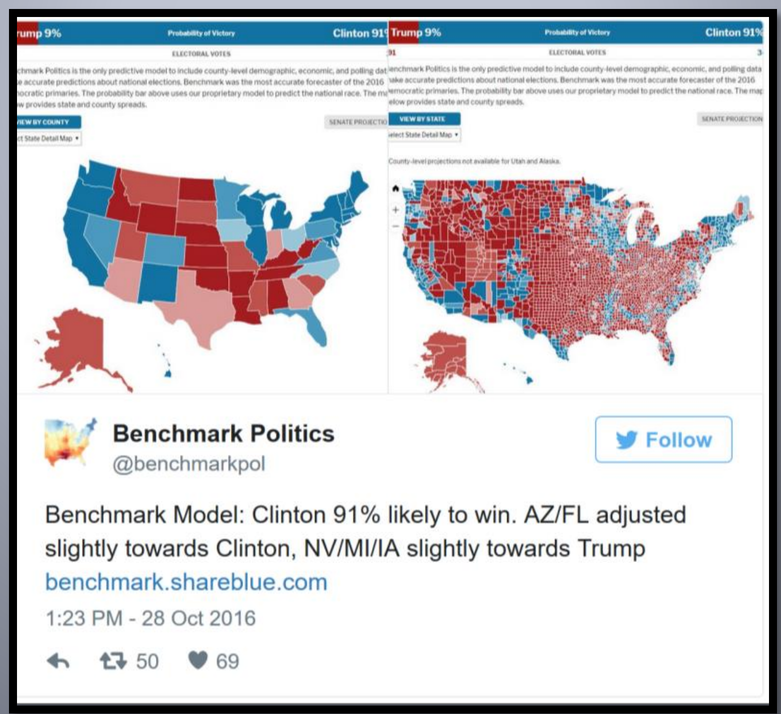
Crashes During Restart

- *CLR*s ensure that no undo action is applied twice
  - What happens if there is a crash during the undo phase?
- An action to be undone falls into three categories
  - It has not been undone or the action must be undone as normal
  - It has been undone, a *CLR* has been written, and an end log record has been written (i.e. the entire transaction is undone)
    - As an end record exists the transaction is not included in the transaction table in the analysis phase
  - It has been undone, a *CLR* has been written, but no end log record has been written
    - The *CLR* is redone during the redo phase

Key Log Records for Crash Recovery



Media Failure



Media Failure

- During a system crash nothing is lost from disk
  - Only temporary data in main memory is lost
  - More serious failures result in the loss of one or more disks
- Theoretically it should be possible to reconstruct the database from the log if
  - The log was not on the damaged disk,
  - The log is a redo (or ARIES) log, and
  - The entire log is retained
- It is not practical to retain the log forever, so *archiving* is used to protect against media failure

Retaining the Log

- A large OLTP DB changes considerably
  - Even if there are a relatively small number of changes each day
  - The log has to record details for each transaction that changes the DB
- If the log is used instead of an archive it will become larger than the DB itself
  - Google *SQL Server Log size* and browse the results

Archiving

- There are different levels of archiving
  - A *full* database backup is a copy of the entire database
  - A *differential* backup copies only the database pages that have been modified after the last full database backup
  - A log backup copies only the log
- Restore Operation (cold restart)
  - Use the latest full database backup
    - Apply all the subsequent differential backups
    - Apply the log backups to include all committed transactions

Non-quiescent Archiving

- Similar to non-quiescent checkpointing
- Makes a copy of the DB when the archive process began
  - But some data elements may change while the archiving is n process
- The log can be used to determine which data elements are incorrect
  - To allow the state of the DB at the archive start to be determined

Recovery and Concurrency

Managing Concurrency and Logging

- The log ensures that committed transactions can be reconstructed if the system crashes
  - It does not attempt to support serializability
- Similarly the concurrency manager is not concerned with the rules of the log manager
  - So could allow a write to the DB of a later aborted transaction
  - Unless prevented from doing so

Rollbacks

- The transaction log has an important role in performing *rollbacks*
  - When a transaction is aborted its effects must be reversed or rolled back
- If the transaction log contains *Undo* data it may be used to reverse a transaction
  - It may also be possible to use data from the disk copy of an object
    - If the data has not yet been written to disk

Recoverable Schedules

- The transactions that are considered to be committed after recovery must be consistent
  - If T_1 is committed after recovery, and it used a value written by T_2 then T_2 must also be committed
    - A schedule is recoverable if each transaction only commits after all transactions from which it has read have committed
- Recoverable schedules are not necessarily serializable
  - And vice versa

Recoverable and Serializable

C = commit

- S_1: $W_1(A)$; $W_1(B)$; $W_2(A)$; $R_2(B)$; C_1; C_2
  - T_2 reads B that was written by T_1 so must commit after T_1 for the schedule to be recoverable
  - This schedule is serial(izable) and recoverable
- S_2: $W_2(A)$; $W_1(B)$; $W_1(A)$; $R_2(B)$; C_1; C_2
  - This schedule is *not* serializable but is recoverable
- S_3: $W_1(A)$; $W_1(B)$; $W_2(A)$; $R_2(B)$; C_2; C_1
  - This schedule is serializable but is *not* recoverable

Cascading Rollbacks

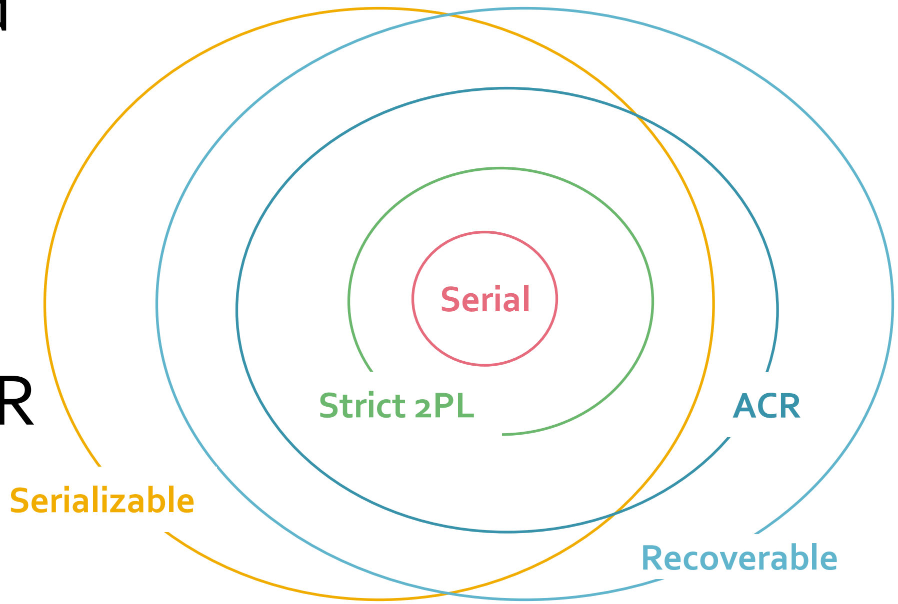
- A cascading rollback occurs when one rollback necessitates additional rollbacks
  - e.g. transactions that have read data written by an aborted transaction must also be aborted
- Some recoverable schedules may involve cascading rollbacks
  - S_1: $W_1(A)$; $W_1(B)$; $W_2(A)$; $R_2(B)$; C_1; C_2
    - If T_1 was aborted instead of committed (at the time of C_1) then C_2 would also have to be rolled back

ACR Schedules

- It is desirable to avoid cascading rollbacks
  - Such a schedule is referred to as an ACR schedule
  - All ACR schedules are recoverable
- In an ACR schedule a transaction should not read data of un-committed transactions
  - S_4: $W_1(A)$; $W_1(B)$; $W_2(A)$; C_1; $R_2(B)$; C_2
    - T_2 only reads B after T_1 has committed, this schedule is therefore ACR as well as recoverable

Strict 2PL Schedules

- Strict 2PL guarantees that schedules are recoverable and serializable
  - Transactions do not release exclusive locks until committed or aborted
  - 2PL guarantees serializability
  - Strict guarantees that schedules are ACR and revoverable



Page Locking

- If main memory pages are lockable database elements there is a simple rollback method
  - That does not entail using the log
- Pages written by uncommitted transactions are pinned in main memory
  - i.e. they cannot be written to disk
  - Aborted transaction can therefore be rolled back by simply not writing the page to disk

SQL Server Concurrency

An Example

Concurrency in SQL Server

- SQL Server supports a variety of concurrency control levels and types
  - Allowing for both pessimistic and optimistic concurrency control
- Pessimistic locking scheme is a Strict 2PL variant
- Optimistic locking scheme is multi-version concurrency control
  - A variation of the timestamp method of optimistic concurrency control
    - That maintains old versions of database elements

Recovery in SQL Server

- SQL Server maintains a transaction log
  - Based on the ARIES logging system

The End

Undo Log – Commit vs. Abort

Undo log assuming no crash or crash occurs after the commit record has been written to disk

LSN	LOG
10	start T
20	update *T*, *A* 12
30	update *T*, *B*, 4
40	commit *T*

Undo log, a crash occurs after update records are written but before commit record is written

LSN	LOG
10	start T
20	update *T*, *A* 12
30	update *T*, *B*, 4
40	abort *T*