Failure Recovery

▲Failure Classification [故障分类]

Transaction failure:

Logical errors: transaction cannot complete due to some internal error condition

System errors: the database system must terminate an active transaction due to an error condition

System crash: a power failure or other hardware or software failure causes the system to crash.

Fail-stop assumption: non-volatile storage contents are assumed to not be corrupted by system crash

Database systems have numerous integrity checks to prevent corruption of disk data

Disk failure: a head crash or similar disk failure destroys all or part of disk storage

Destruction is assumed to be detectable: disk drivers use checksums to detect failures

▲Recovery algorithms have two parts:

Actions taken during normal transaction processing to ensure enough information exists to recover from failures [在正常事务处理期间为确保存在足够的信息以从故障中恢复而采取的操作]

Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability [在将数据库内容恢复到确保原子性、一致性和持久性的状态失败后所采取的操作]

▲Storage Structure [存储结构]

Volatile storage [易失性存储器]:

does not survive system crashes

examples: main memory, cache memory

Non-volatile storage [非易失性存储]:

survives system crashes

examples: disk, tape, flash memory, non-volatile (battery backed up) RAM

but may still fail, losing data

Stable storage [稳态存储]:

a mythical form of storage that survives all failures

usually approximated by maintaining multiple copies on distinct nonvolatile media

▲Data Access [数据存取]

Physical blocks are those blocks residing on the disk.

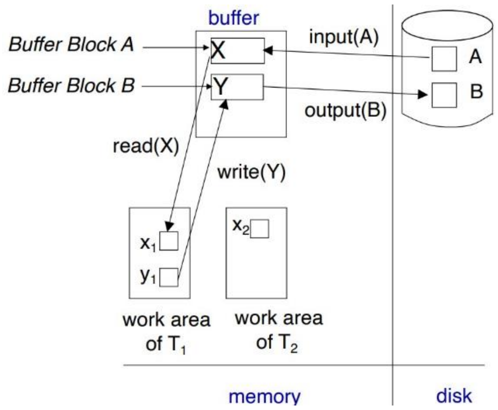
Buffer blocks are the blocks residing temporarily in main memory.

Block movements between disk and main memory are initiated through the following two operations:

input(B) transfers the physical block B to main memory.

output(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

We assume, for simplicity, that each data item fits in, and is stored inside, a single block.



Each transaction Ti has its private work-area in which local copies of all data items accessed and updated by it are kept.

Ti's local copy of a data item X is called xi.

Transferring data items between system buffer blocks and its private work-area done by:

read(X) assigns the value of data item {X} to the local variable xi.

write(X) assigns the value of local variable xi to data item {X} in the buffer block.

Note: output (BX) need not immediately follow write(X). System can perform the output operation when it deems fit.

Transactions

Must perform read(X) before accessing X for the first time (subsequent reads can be from local copy)

write(X) can be executed at any time before the transaction commits

Recovery and Atomicity [恢复与单元性]

To ensure atomicity despite of failures, we first output information describing the modifications (e.g. logs) to stable storage without modifying the database itself.

A log is kept on stable storage.

The log is a sequence of log records, and maintains a record of update activities on the database.

When transaction Ti starts, it registers itself by writing a <Ti start> log record

Before Ti executes write(X), a log record <Ti, X, V1, V2> is written, where V1 is the value of X before the write (the old value), and V2 is the value to be written to X (the new value).

When Ti finishes it last statement, the log record <Ti commit> or <Ti abort> is written.

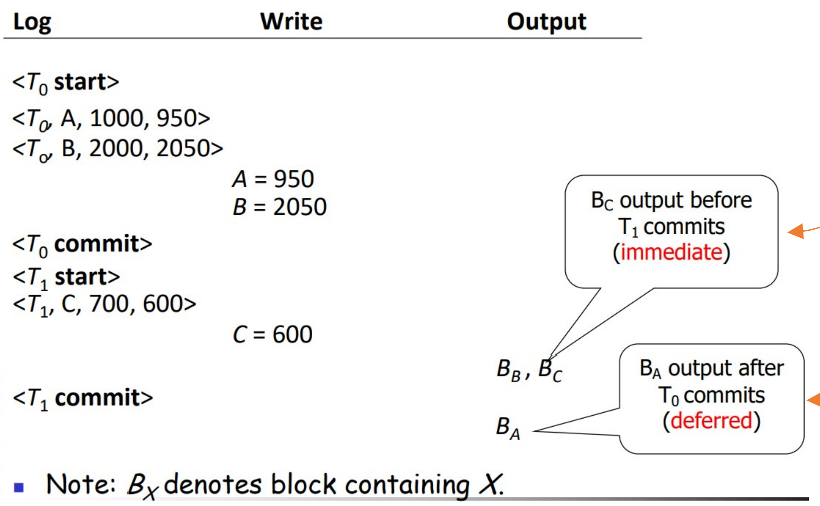
Two approaches using logs:

Deferred [延迟] database modification

performs updates to buffer/disk only at the time of transaction commit.

Immediate [立即] database modification

allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits



Transaction Commit

A transaction is said to have committed when its commit log record is output to stable storage.

all previous log records of the transaction must have been output already.

Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later.

Concurrency Control and Log-Based Recovery [并发控制&基于日志恢复]

With concurrent transactions, all transactions share a single disk buffer and a single log

A buffer block can have data items updated by one or more transactions

We assume that if a transaction Ti has modified an item, no other transaction can modify the same item until Ti has committed or aborted, i.e. using the strict two-phase locking protocol. (不可同时操作同一 item)

Log records of different transactions may be interspersed (散布) in the log.

Undo [撤销] of a log record <Ti, X, V1, V2> writes the old value V1 to X

Undo (Ti) restores the values of all data items updated by Ti to their old values, going backwards from the last log record for Ti.

(1) Each time a data item X is restored to its old value V, a special log record <Ti , X, V> is written out.

(2) When undo of a transaction is complete, a log record <Ti abort> is written out.

Redo [重做] of a log record <Ti, X, V1, V2> writes the new value V2 to X (again)

Redo (Ti) sets the value of all data items updated by Ti to the new values, going forward from the first log record for Ti.

(1) No additional logging is done in this case

Recovering after failure:

Transaction Ti needs to be undone if the log

- contains the record <Ti start>,
- but does not contain either the record <Ti commit> or <Ti abort>.

Transaction Ti needs to be redone if the log

- contains the records <Ti start>
- and contains the record <Ti commit> or <Ti abort>

Tips: 如果事务 Ti 在更早的时候被 undone，<Ti commit >记录写到日志中，然后发生了失败，那么从失败中恢复 Ti 时会选择 redone，这样的 redone 会重做所有原始的操作，包括恢复旧值的步骤，称为重复历史。看起来很浪费，但极大地简化了恢复算法。

Example:

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T0): B is restored to 2000 and A to 1000, and log records <T0, B, 2000>, <T0, A, 1000>, <T0, abort> are written out

(b) redo (T0) and undo (T1): A and B are set to 950 and 2050 and C is restored to 700. Log records <T1, C, 700>, <T1, abort> are written out.

(c) redo (T0) and redo (T1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

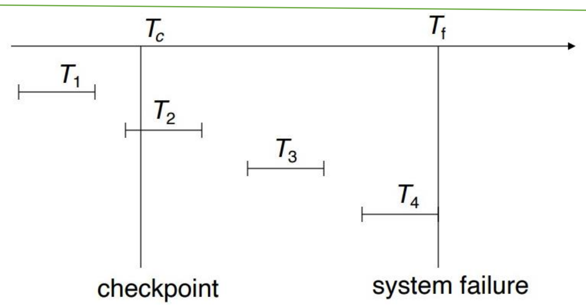
Checkpoints [检验点]

Redoing/undoing all transactions recorded in the log can be very slow (processing the entire log is time-consuming if the system has run for a long time). So, we might unnecessarily redo transactions which have

already output their updates to the database long time ago. (节省时间不必做重复的工作)
Streamline recovery procedure by periodically performing checkpointing:
(1) Output all log records currently residing in main memory onto stable storage.
(2) Output all modified buffer blocks to the disk.
(3) Write a log record <checkpoint L> onto stable storage where Lis a list of all transactions which are active at the time of checkpointing.
(4) All updates are stopped while doing checkpointing.

example

回溯查询 checkpoint



In this example, T1 can be ignored (updates already output to disk due to checkpoint), T2、T3 redone and T4 undone.

▲Recovery Algorithm

Logging (during normal operation)
<Ti start> at transaction start
<Ti, Xj, V1, V2> for each update, and
<Ti commit> at the end of transaction

Transaction rollback (during normal operation)
Let Ti be the transaction to be rolled back
Scan log backwards from the end, and for each log record of Ti of the form <Ti, Xj, V1, V2>
perform the undo by writing V1 to Xj,
write a log record <Ti , Xj, V1> (such log records are called compensation (补偿) log records)
Once the record <Ti start> is found stop the scan and write the log record <Ti abort>

Recovery from failure have two phases:
[1] Redo phase: replay updates of all transactions, whether they committed, aborted, or are incomplete, at and after checkpoint
[2] Undo phase: undo all incomplete transactions

Redo phase:
Find last <checkpoint L> record, and set the undo-list to L (undo-list = L)
Scan forward from above <checkpoint L> record:
Whenever a record <Ti, Xj, V1, V2> is found, redo it by writing V2 to Xj
Whenever a log record <Ti start> is found, add Ti to undo-list
Whenever a log record <Ti commit> or <Ti abort> is found, remove Ti from undo-list

Undo phase:
Scan log backwards from the failure point:
Whenever a log record <Ti, Xj, V1, V2> is found where Ti is in undo-list (same as transaction rollback):
perform undo by writing V1 to Xj.

write a log record <Ti , Xj, V1>

Whenever a log record <Ti start> is found where Ti is in undo-list:

Write a log record <Ti abort>

Remove Ti from undo-list

Stop when undo-list is empty!

i.e. <Ti start> has been found for every transaction in undo-list

After undo phase completes, normal transaction processing can commence (开始) again.

example