Advances in Data Management
Transaction Management
A.Poulovassilis

1 The Transaction Manager

Two important measures of DBMS performance are *throughput* — the number of tasks that can be performed within a given time, and *response time* — the time it takes to complete a single task.

To achieve fast throughput of transactions, when one transaction is waiting for a page to be read in from disk (I/O) the CPU can be processing other transactions concurrently. This also allows shorter transactions to complete more quickly, i.e. faster response times.

While this concurrenct execution of transactions is happening, the transaction manager (TM) must ensure that the ACID properties of transactions are preserved (see earlier notes on the ACID properties).

The two major aspects of the TM that achieve the ACID properties are:

- **logging and recovery** — guarantees A and D, and
- **concurrency control** — guarantees C and I.

The TM records information about the current set of active transactions in a **transaction table**. This includes for each transaction its unique transaction id and its *status* — in progress, committed, or aborted.

The TM removes committed and aborted transactions from the transaction table once it has completed certain 'house-keeping' activities e.g. release of any locks held by the transaction, and removal of the transaction from the 'waits-for graph' (Section 3 below).

The general architecture of a DBMS's Transaction Manager is shown in Figure 1.

2 Logging and recovery

The TM must ensure ensure the Atomicity and Durability properties of transactions even if a system failure occurs. The most widely used method in DBMSs for achieving this is through **logging**:

The TM keeps a record of all the updates made by transactions within a **log** file. Like other database files, the log file is stored on disk, and part of it may also be present in memory within the buffer.

(We recall that both the buffer and the disk files are organised into **pages**.)

The log file consists of a sequence of records, each of which has a unique **log sequence number** (LSN). These numbers are assigned in increasing order.

As transactions execute, records are added to the log file recording the progress of the transaction, typically including:

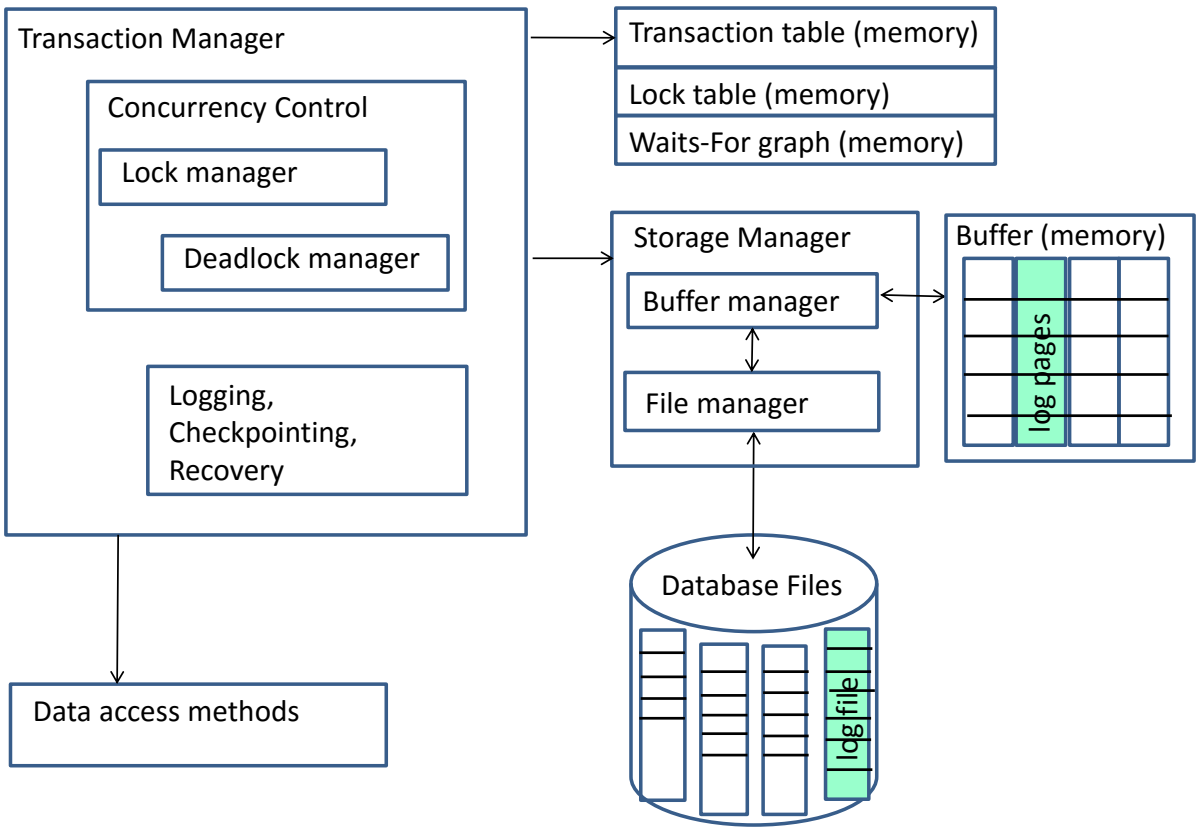

Figure 1: Transaction Manager Architecture

- a BEGIN record, containing the transaction id and indicating the start of the transaction;
- an UPDATE record for each change made by the transaction, containing:
  - the transaction id
  - the page id of the modified page
  - the offset within the page of the bytes that have changed and the number of bytes that have changed
  - the **before image** i.e. the changed bytes before the change
  - the **after image** i.e. the changed bytes after the change[1]
- a COMMIT or ABORT record, containing the transaction id;
- an END record, containing the transaction id, and indicating that the transaction has been removed from the transaction table, waits-for graph etc. and that its locks have been released.

2.1 Normal Transaction Execution

As transactions execute, their updates are carried out on pages that have been brought into the buffer from disk.

For performance reasons, buffer pages whose contents have been updated are not immediately written back to disk — an updated buffer page P need only be written back to disk if new pages need to be brought into the buffer from disk and P is selected to be 'paged-out'.

Also for performance reasons, usually a **steal/no-force** approach is adopted by TMs: one transaction can 'steal' a buffer page from another currently active transaction; and the changes of a committed transaction are not immediately 'forced' to disk.

(The opposite of these would be 'no-steal' — whereby only pages that are not being used by any active transaction can be replaced in the buffer; and 'force' — whereby the changes made by a transaction that is committing are immediately written to disk, and only then is the transaction regarded as having committed.)

The problem is what happens if a system failure occurs?

This will generally cause a loss of the current contents of the buffer pages (in volatile storage).

What if there are changes by uncommitted transactions that have been already written to disk (due to 'steal')?

What if there are changes of committed transactions that have not been written to disk (due to 'no-force')?

How can the TM bring the database back to a consistent state? By 'consistent' state we mean satisfying the Atomicity and Durability properties i.e that:

- the updates made by all committed transactions are correctly reflected in the files on disk; and

[1] Specifically, this kind of UPDATE log record in known as a *physical* log record. In practice, other types of log records are also used: *logical* log records and *physiological* log records — see discussion in Silberschatz et al. Sections 16.7, 16.8 (optional)

- any updates made to files on disk by uncommitted transactions are undone.

To achieve this, two things are done by the TM:

(i) In general, updates made to the log will be recorded within the part of the log within the buffer, but may not immediately be recorded to the log on disk.

However, before any updated data page in the buffer is written back to disk, the log records that describe the updates to this page are **first** written to disk. This is known as **Write-Ahead Logging**. It means that any update made to a data file is also recorded within the log on stable storage (i.e. on disk).

(ii) When a transaction decides to commit, all of the log records for this transaction are written to disk, including the COMMIT record. The transaction is regarded as having committed as soon as its COMMIT record has been written to disk[2].

Thus we have on stable storage a record of all committed transactions and the updates that they have made.

From time to time during normal database operation, the TM also performs a **checkpoint**, in order to reduce the amount of work that would need to be done to recover after a system failure. This includes:

- writing the log records currently in the buffer to disk;
- writing the modified data pages in the buffer to disk;
- writing a CHECKPOINT record to the log on disk, which records the list of transactions that are active at this time.[3]

Undoing failed transactions.

As well as aborting during system or media failures, individual transactions may also abort as a result of issuing an explicit ROLLBACK statament or violating some database integrity constraint or in order to resolve a deadlock.

In order to undo the updates of such a failed transaction, T, the log records for T are processed backwards by the TM, and the 'before images' within the UPDATE log records are restored.

For each such change, a *redo-only* log record is also written to the log, recording the value being restored (there is no need to record also the previous value as that will never be needed again).

Finally, an ABORT log record is written, recording that T has been aborted.

(If a transaction needs to be re-run, it is started again with a new transaction number, either after being resubmitted by the application or being automatically restarted by the system.)

[2]Also, when a transaction ends, the transaction's pages in the buffer are 'un-pinned' and the transaction is removed from the transactions table, lock table, waits-for graph etc.

[3]This method of checkpointing requires that no updates are allowed on the database during the checkpointing process. There are also "fuzzy" checkpointing methods that allow transactions to carry on performing updates while the modified buffer pages are being written to disk (see discussions in Ramakrishnan & Gehrke, Silberschatz et al. — optional).

2.2 Recovering after Systems failure

When the DBMS restarts after a systems failure, the recovery process starts by locating in the log (on stable storage) the most recent CHECKPOINT record by searching the log backwards, starting at the end.

A wide variety of log-based recovery algorithms have been proposed in the literature and deployed in DBMSs. The state-of-the art is the ARIES method (see discussions in Ramakrishnan & Gehrke, Silberschatz et al. — optional). The algorithm included here is a simplification of that approach, and is drawn from Silberschatz et al. Section 16.4.2:

First a **redo** phase is undertaken, in which the TM works forwards through the log starting from the most recent CHECKPOINT record and reapplies all the changes made since that time by restoring the 'after image' within the UPDATE log records to the database. (This includes the updates made by *all* transactions, whether committed, aborted or still running at the time of the crash.)

During the redo phase, a list of transactions that were still active at the time of the crash is also compiled (these will be transactions for which no COMMIT or ABORT log record is found in the log) – this is called the undo-list.

Next, an **undo** phase is undertaken, in which the TM works backwards through the log and, whenever it finds an UPDATE record belonging to a transaction in the undo-list, it restores the 'before image' to the database. For each such change, a *redo-only* log record is written to the log. When the START record of a transaction in the undo-list is found, this means that the transaction's updates have been fully reversed and an ABORT record is written to the log.

Note 1: The above is a Redo/Undo recovery algorithm. There are also recovery algorithms that have a prior 'analysis' phase: Analysis/Redo/Undo, as in ARIES. Also algorithms where Undo is done first, and then Redo only of completed transactions.

Note 2: There are also simpler recovery algorithms, comprising Undo-only recovery or Redo-only recovery:

- If *steal* is not possible, then Undo information is not needed for recovery.
- If *force* is in operation, i.e. all data pages updated by a transaction are written to disk before the transaction commits, then Redo information is not needed for recovery.

Note 3: Recovery algorithms need to be *idempotent* i.e. if the system crashes during the recovery process, it should be possible to reapply the recovery algorithm again, irrespective of how much of the recovery process was already completed.

Note 4: In the ARIES recovery algorithm, the Redo phase brings the database to the state that it was at the tme of the crash, before the Undo phase is then applied. The advantage of this "repeating history" approach is that finer-granularity locking and logging of logical operations can be handled by the recovery algorithm, not just byte-level modifications. This can allow considerably higher levels of concurrency.

2.3 Recovering after Media failure

Media failure means the corruption of part of the database files due to a disk crash, or other damage or loss of a disk, or due to incorrect updating of the database files by the DBMS software.

To guard against this, back-up copies are regularly made of the database to different disks (or to tertiary storage), possibly stored at a different site.

To recover from a media failure, the DBMS can use whatever portion of the log file still survives.

Treating the time that the last back-up was taken as a 'checkpoint', the same process as for recovering from systems failure can then be applied to undo the effects of uncommitted transactions and redo the effects of committed transactions since the time of the last back-up.

Another common way to mitigate against media failure is to maintain a second (or more than two) active copy of the database (including the log). In the case of failure of one copy, the applications supported by the database can continue using another copy.

3 Concurrency control

The aim of concurrency control is to guard against a number of undesirable phenomena which can occur if multiple transactions are allowed to execute freely at the same time e.g. *lost updates*, *dirty reads*, *unrepeatable reads*, *phantom reads* (discussed below).

The concurrent execution of a set of transactions can be represented as a **schedule**, which lists the order in which the statements of the transactions are executed.

Schedules always respect the ordering of the statements of an individual transaction i.e. if a transaction consists of statements $s_1; s_2; \ldots s_n$ then any schedule will always execute s_i before it executes s_{i+1}, for all $0 < i < n$ (but possibly executing other statements from other transactions in between).

For the purposes of concurrency control, transaction statements that *read* or *write* data, or that *abort* or *commit* a transaction are of relevance (INSERT, DELETE and UPDATE statements are all considered as *writes* when discussing schedules).

We represent by $r[x]$ a read operation on a data item x and by $w[x]$ a write operation on a data item x — where a 'data item' may be a single record, a page, a set of pages, a whole file etc.

Two operations **conflict** if they both operate on the same data item and at least one of them is a write operation — thus **read-write** and **write-write** conflicts are possible.

Two schedules are **equivalent** if they order pairs of conflicting operations of committed transactions in the same way[4].

A **serial schedule** executes transactions one after the other, in some order.

A schedule is said to be **serialisable** if it is equivalent to a serial schedule.

[4]Note that this is the definition of **conflict-equivalence**. There are other definitions of when two schedules are equivalent, but this is the definition assumed in practice in DBMSs. We can assume that all transactions in a schedule are either aborted or committed i.e. there no partial transactions, due to the Atomicity property guaranteed by logging/recovery.

The purpose of a DBMS's concurrency control mechanisms is to produce serialisable schedules.

3.1 Two-phase locking

The most common method of concurrency control in DBMSs is to place **locks** on the data items being accessed by each transaction.

The data items on which locks are placed may be individual rows of tables, whole tables, parts of tables, parts of indexes etc. The TM automatically determines what granularity of locking is needed by each transaction operation, though with some DBMS it is possible for the programmer to have some control over this.

The part of the TM software that keeps track of the locks that transactions are placing on data items is called the **lock manager**. It maintains a **lock table** which records the set of locks currently placed on each data item o, and also the list of transactions waiting to place a lock on o.

Fine granularity locks (e.g. on individual records) provide greater parallelism but are more costly compared with coarse granularity locks (e.g. at the level of pages or whole files) which may allow less parallelism but have a lower lock management overhead.

In the simplest locking schemes, there are two kinds of locks: **R-locks** (read-locks) and **W-locks** (write-locks). R-locks are also known as 'shared locks' (S-locks) and W-locks as 'exclusive locks' (X-locks).

The lock compatibility table is therefore as follows, where T_1 and T_2 are different concurrently executing transactions:

	T_1:R	T_1:W
T_2:R	y	n
T_2:W	n	n

A commonly used locking protocol is **strong, strict two-phase locking** (**SS-2PL**), and its rules are as follows:

1. If a transaction wants to retrieve a data item, the TM places an R-lock on that data item provided there is no W-lock on it from another transaction.
2. If a transaction wants to write to a data item, the TM places a W-lock on that data item provided there is no R-lock or W-lock on it from another transaction.
3. If a transaction cannot obtain a lock on a data item, it has to wait until the TM can grant it a lock.
4. A transaction releases its locks only when it aborts or commits.

Thus, transactions have two phases: a 'growing phase' in which they only acquire locks and a 'release phase' in which they release all locks they have acquired.

Any schedule output by a 2PL scheduler is serialisable (see the Appendix for a justification of this — optional reading).

Note 1: The 'basic' version of 2PL has a less strict 4th rule, that allows transactions to release locks before they end:

4. Once a transaction releases a lock it cannot acquire any more new locks.

Although still only generating serialisable schedules, this basic 2PL protocol can cause *cascading aborts of transactions*: a transaction T_1 may read data written by another transaction T_2 which has released it W-lock and which later aborts; so T_1 will also have to abort.

It can also cause *unrecoverable schedules*: a transaction T_1 may read data written by another transaction T_2; T_1 commits; but later T_2 aborts. Since T_1 has committed, its effects cannot be reversed — a compensating transaction may need to be executed.

Note 2: Older descriptions (including Ramakrishnan and Gehrke) call SS-2PL just 'strict 2PL'; however, more recently 'strict 2PL' is taken to mean that a transaction releases its W-locks only when it ends, but can release its R-locks earlier (while still complying with the 'basic' 2PL rule that once a lock has been released no more locks can be acquired).

Strict 2PL also yields cascade-free and recoverable schedules; however, in practice, SS-2PL is adopted in DBMSs. With SS-2PL the serialisation order of transactions coincides with the order that they commit.

Examples. Consider the following two transactions, T1 and T2, which are running at the same time:

```
T1
--
T1.1 Read Fred's record from the Payroll relation
T1.2 Add 2000 to Fred's salary and write the record back
T1.3 Read Fred's record from the Payroll relation
T2
--
T2.1 Read Fred's record from the Payroll relation
T2.2 Add 1000 to Fred's salary and write the record back
```

Consider the following three possible schedules of execution of these transactions, which illustrate the undesirable phenomena of lost update, dirty read (a.k.a.read uncommitted), and unrepeatable read:

(a) T1.1, T2.1, T1.2, T2.2, ...

(b) T1.1, T1.2, T2.1, T1 aborts, T2.2

(c) T1.1, T1.2, T2.1, T2.2, T1.3

How does the use of SS-2PL prevent these phenomena?

(a) `T1.1 T1 places a R-lock on Fred's record`
`T2.1 T2 places a R-lock on Fred's record`

```
T1.2 T1 cannot place a W-lock on Fred's record and waits
T2.2 T2 cannot place a W-lock on Fred's record and waits
```

There is now deadlock, and either T1 or T2 will be aborted and restarted by the TM. Suppose T2 is aborted. Then its locks are released, allowing T1 to obtain the W-lock it needs, complete, commit, and release its locks. T2 can then execute. The result is a serial execution of T1 and T2.

(b)
```
T1.1 T1 places a R-lock on Fred's record
T1.2 T1 places a W-lock on Fred's record
T2.1 T2 cannot place an R-lock on Fred's record and waits
T1 aborts and releases its locks
```

Any updates made by T1 are rolled back. T2 can then obtain the R-lock it needs, reading the pre-T1 value of Fred's record, and can execute as though T1 had never started.

(c)
```
T1.1 T1 places a R-lock on Fred's record
T1.2 T1 places a W-lock on Fred's record
T2.1 T2 cannot place an R-lock on Fred's record and waits
T1.3 T1 already has the R-lock it needs
```

T1 eventually completes, commits and releases its locks. T2 can then obtain the R-lock it needs to continue. The result is a serial execution of T1 and T2.

4 Deadlocks

Consider these two transactions:

```
T1
--
T1.1 retrieve the row for account number 192 from the ACCOUNTS relation
T1.2 add 500 to the amount and write back the row

T2
--
T2.1 retrieve the row for account number 192 from the ACCOUNTS relation
T2.2 add 200 to the amount and write back the row
```

A possible schedule with two-phase locking is:

```
T1.1 places a R-lock on account 192 in the ACCOUNTS relation.
T2.1 also places a R-lock on account 192.
T1.2 cannot now place a W-lock since T2 has placed a R-lock, so T1 waits.
T2.2 cannot now place a W-lock since T1 has placed a R-lock, so T2 waits.
```

Thus, we see that the blocking of transactions by 2PL can give rise to **deadlock** i.e. two or more transactions are simultaneously waiting for each other to release a lock before they can proceed.

The TM maintains a structure called a **waits-for** graph in order to detect such deadlocks. The nodes of this graph correspond to transactions which are currently executing. There is an arc from T_i to T_j if T_i is waiting for T_j to release a lock. Deadlocks result in cycles being formed in the waits-for graph, so this is periodically checked for cycles.

If a deadlock is detected by the TM, it can be resolved by aborting one of the transactions involved, thus releasing its locks and allowing some of the waiting transactions to proceed.

The choice of which transaction to abort will depend on certain criteria e.g. the transaction which started last, or the transaction which holds the least number of locks. Whatever policy is adopted, care needs to be taken that the same transaction is not repeatedly selected to be aborted!

In the above example, one of the transactions will be aborted by the TM, say T2, releasing its locks. T1 will complete, will release its locks, and T2 can then restart.

Deadlock prevention

In update-intensive applications there may be a high level of contention for locks, and hence an increased likelihood of deadlocks occurring and transactions having to be aborted.

In such scenarios, **timestamps** can be used to prevent deadlocks from occurring. Each transaction is 'stamped' with the time that it starts. The earlier the timestamp, the higher the transaction's priority.

When a transaction T_i requests a lock which conflicts with a lock held by another transaction T_j, one of two protocols can be used:

wait.die: if T_i has higher priority then it waits, otherwise it is aborted and restarted (retaining its old timestamp);

wound.wait: if T_i has higher priority then T_j is restarted, otherwise T_i waits.

Both these protocols give priority to older transactions:

wait.die is *non-pre-emptive*: only transactions requesting a lock can be aborted; a transaction holding all the locks it needs can never be restarted.

wound.wait is *pre-emptive*: a currently running transaction holding all the locks it needs can be restarted by a higher-priority transaction.

Either way, no deadlock cycles can occur.

5 Other Concurrency Control methods

Locking is the most widely used method for concurrency control in DBMSs, but there are others — see Ramakrishnan & Gehrke Chapter 17 for details (**optional reading**).

timestamp-based concurrency control assigns each transaction a unique timestamp, and produces schedules which are equivalent to a serial execution of transactions in the order of their timestamps. Transactions do not wait for each other, but instead are aborted and restarted if they violate the timestamp ordering rules. This may result in a cascade of transaction aborts — if an update made by an aborted transaction has been read by another transaction with a higher timestamp, then this transaction must also be aborted.

optimistic concurrrency control (also known as **validation-based** concurrency control) assumes that transactions will not conflict with other transactions. (In contrast, locking- and timestamp-based methods are "pessimistic".) No locking is used, and transactions make their updates to their own 'private' workspace. When a transaction is ready to commit, the TM checks whether it conflicts with any other concurrently executing transaction, in which case it is aborted and restarted. Otherwise its private updates can be made public i.e. transferred to the database files.

Performance studies have shown that, in general, blocking transactions is preferable to restarting them since less work is undone and therefore fewer resources are wasted, so optimistic methods perform worst. Locking generally outperforms timestamp ordering — intuitively, timestamp ordering imposes one order of execution upon transactions whereas locking methods allow any order which corresponds to a serial schedule.

However, optimistic methods are preferable to locking in the cases of low probability of conflicts between transactions (e.g. in predominantly read-only workloads) where the overhead of locking is greater than the overhead of restarting aborted transactions.

Finally, **multiversion** concurrency control methods rely on earlier versions of updated data items being available e.g. reconstructed from the 'before' images in the log files. This results in fewer transactions waiting for data items to be written, or aborting because they are "too late" to read the correct version of a data item.

6 Transaction Isolation Levels

In order to increase transaction throughput, and reduce the overhead of locking and of checking/preventing deadlocks, DBMSs in practice allow programmers to reduce the **isolation level** of transactions in order to permit non-serialisable transactions — see Homework Reading on Transaction Levels in the SQL standard.

For example, in Oracle transactions can be run in one of two isolation levels:

- transaction-level isolation — this corresponds to the SQL standard's SERIALIZABLE isolation level;
- statement-level isolation — this corresponds to the SQL standard's READ COMMITTED isolation level.

7 Snapshot isolation

This is a **multiversion, locking-based** concurrency control scheme that is widely used in practice (e.g. in Oracle, PostgreSQL, SQL Server). The aim is to reduce the amount of transaction blocking and deadlocks, and hence increase the amount of concurrent transaction execution, compared with single-version locking schemes.

When a transaction T begins, the ID of last committed transaction is noted — prev(T).

All read requests that T issues are executed with respect to the database state as left by prev(T), extended with any updates that T itself makes — these updates are visible only to transaction T

until it commits. This is the moving "snapshot" of the database that transaction T sees.

Read locks on data are therefore not necessary. Thus transactions that only perform Reads ("readers") do not block transactions that perform Writes ("writers"), and writers do not block readers. The only remaining possibility is for a writer to block another writer.

A transaction T needing to make an update on a data item x, first requests a W lock on x. There are a number of possible scenarios

- There is no W lock currently on x:
  - If x has already been updated by another transaction running concurrently with T, then T aborts (otherwise serialisability could be violated).
  - Otherwise, T is granted a W lock on x and proceeds.
- There is a W lock held currently on x by another transaction T'.

  In this case, T waits until T' commits or aborts.
  - If T' commits, then T aborts (otherwise serialisability could be violated).
  - If T' aborts, then T can obtain the W lock on x. However, if x has been updated by another transaction that has been running concurrently with T, then T aborts. Otherwise, T proceeds.

All of the updates of a committing transaction are performed as an atomic action, so as to create a new database "version". A transaction's locks are released after it commits or aborts.

In Oracle, this personal "snapshot" for a transaction T is achieved by using the logged information about transactions' updates to construct, for T, the database state as left by prev(T).

This information is stored in what Oracle calls "undo segments" (or "rollback segments").

Occasionally, there will not be enough log information available to reconstruct a snapshot — undo segments are allocated a fixed amoount of space and older committed transactions' undo segments are eventually reused.

Oracle issues the "snapshot too old" exception in these cases. This exception can be caught in the application code and the transaction can be restarted.

8 Locking in B+ trees

Naively, R- or W-locks would be placed on all the nodes involved in a retrieval, insert or delete operation on a B+ tree i.e. on the whole path from the root to the appropriate leaf.

However, for retrieval operations we never need to go back up the tree. Thus, we only need to have a R-lock on the current node being read. So during a retrieval operation, a R-lock is placed on the node, N, that is currently being read, and the R-lock that had been placed on its parent is released (in the case that N is the root, there is of course be no such parent to unlock). Thus, at most two nodes are R-locked at any one time.

Insertion operations may affect the ancestors of the leaf node that is to be updated, due to a cascade of splitting. However, a given node on the path from the root to the leaf will only be affected by a split if its child is full. Therefore, when we place a W-lock on a child, we can release the W-lock on its parent if the child is not full.

Deletion operations may affect the ancestors of the leaf node that is to be updated, due to a cascade of merging. However, a given node on the path from the root to the leaf will only be affected by a merge if its child is at minimum occupancy. Therefore, when we place a W-lock on a child, we can release the W-lock on its parent if the child is not at minimum occupancy. With deletion, we have to also take care to place a lock on the **sibling** of a node that is at minimum occupancy, if this sibling is going to be involved in a redistribution or a merge.

9 Protecting shared data structures

In addition to locks being placed on data items, a DBMS also needs to take steps to protect data structures that are being accessed concurrently by multiple processes e.g. the data structures describing the pages in the buffer, the transaction table, the lock table, the waits-for graph.

This can be done by using the mechanism of "latches", which are single-threaded sections of code that execute serially on such data structures (c.f. semaphores in operating systems).

Homework Reading

Read the following pages from Ramakrishnan & Gerhke:

Section 16.5 (pages 533-535) on the performance of locking-based concurrency control;
Section 16.6 (pages 535-540 on transaction support in SQL, locking at multiple levels of granularity, and transaction isolation levels;
Sections 17.5.1 and 17.5.2 (pages 559-564) on dealing with the *phantom problem* through *index locking*, and on concurrency control in B+ trees.

Appendix - Mathematical proof that 2PL schedulers output serialisable schedules

The **serialisation graph** of a schedule is a directed graph whose nodes represent transactions that have committed. There is an edge $T_i \rightarrow T_j$ $(i \neq j)$ if the schedule contains an operation of T_i which precedes and conflicts with an operation of T_j.

Theorem: If the serialisation graph G of a schedule S is acyclic, then S is serialisable.

Proof: Perform a **topological sort** of G. This consists of first picking a node of G which has no incoming arcs, T_1 say. Remove this node and its outgoing arcs from G and repeat the process, giving T_2 say. Continue until no nodes remain, giving the overall list of nodes T_1, ... T_n.

Then, S is equivalent to the serial schedule T_1, ..., T_n. Proof of this is so can be shown 'by contradition':

Suppose that S is not equivalent to T_1, ..., T_n. Then there are two conflicting operations $o_i \in T_i$ and $o_j \in T_j$ such that T_i precedes T_j in T_1 , ..., T_n but o_j precedes o_i in S. So there must be an edge $T_j \rightarrow T_i$ in G, by the definition of a serialisation graph. But this contradicts the fact that T_i precedes T_j in the topological sort of G.

So S must be equivalent to the serial schedule T_1, ..., T_n.

Theorem: Any schedule output by a 2PL scheduler is serialisable.

Proof: Consider the serialisation graph G for any schedule S output by a 2PL scheduler. If there is an arc $T_i \rightarrow T_j$ in G, then T_i must have released a lock before T_j acquired it. By transitivity, if there is a path from T_i to T_j in G then T_i must have released a lock before T_j acquired some (possibly different) lock. In particular, if there is path from T_i to T_i in G (i.e. a cycle) then T_i must have released a lock before T_i acquired some lock, violating rule 4 of 2PL above. So G must be acyclic.