Basi di Dati e Sistemi Informativi I

Technology of a database server

Technology of database servers

Five ideal Components:

- Optimizer - selects the data access strategy
- Access Methods Manager - executes the strategy
  - RSS (Relational Storage System)
  - OM (Object Manager)
- Buffer Manager - manages page accesses
- Reliability Manager - manages faults
- Concurrency Control Manager - manages the interference due to multi-user data access

Approach

- Top-Down
- From theory to systems
- Centered on the notion of transaction
- Valid for both relational and object-oriented systems

Definition of a Transaction

- An elementary unit of work performed by an application, with specific features for what concerns correctness, robustness and isolation
- Each transaction is encapsulated within two commands
  - begin transaction (`bot`)
  - end transaction (`eot`)
- Within a transaction, one of the commands below is executed (exactly once)
  - `commit work` (commit)
  - `rollback work` (abort)
- **Transactional System**: a system capable of providing the definition & execution of transactions on behalf of multiple, concurrent applications

(Abstract) Example of Transaction

```
begin transaction
x := x - 10
y := y + 10
commit work
end transaction
```

Well-formed Transaction

A transaction starting with `begin transaction`, ending with `end transaction`, in whose execution just one of the two commands `commit work` or `rollback work` is executed, and without data management operations processed after the execution of the `commit work` or `rollback work`

ACID Properties of Transactions

- **ACID** is an acronym for:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Atomicity

- A transaction is an atomic unit of work
- It cannot leave the database in an intermediate state:
  - a fault or error prior to commit causes the UNDO of the work made earlier
  - A fault or error after the commit may require the REDO of the work made earlier, if its effect on the database state is not guaranteed
- Possible behaviors
  - Commit = normality (99.9%)
  - Rollback requested by the application = suicide
  - Rollback requested by the system = murder

Consistency

- Consistency amounts to requiring that the transaction does not violate any integrity constraint
- Integrity constraint verification can be:
  - Immediate: during the transaction (the operation causing the violation is rejected)
  - Deferred: at the end of the transaction (if some integrity constraint is violated, the entire transaction is rejected)

Isolation

- Isolation requires that any transaction executes independently from the execution of all other concurrent transactions
  - [isolation requires that the concurrent execution of a collection of transactions yields to the same result as an arbitrary sequential execution of the same transactions]

Durability (Persistence)

- Durability requires that the effect of a transaction that has successfully committed be not lost (the effect will “last forever”)

Transactions and system modules

- Atomicity and durability are guaranteed by the **Reliability Control System**
- Isolation is guaranteed by the **Concurrency Control System**
- Consistency is managed during the normal query execution by the **DBMS System** (verification activities are generated by the DDL Compilers and executed during query processing)

Concurrency control

- Concurrency: highly desired
- Measured in *tps* (transactions per second) - typical values are tens to hundreds to thousands tps
- Typical examples: banks, airline reservation systems

Architecture

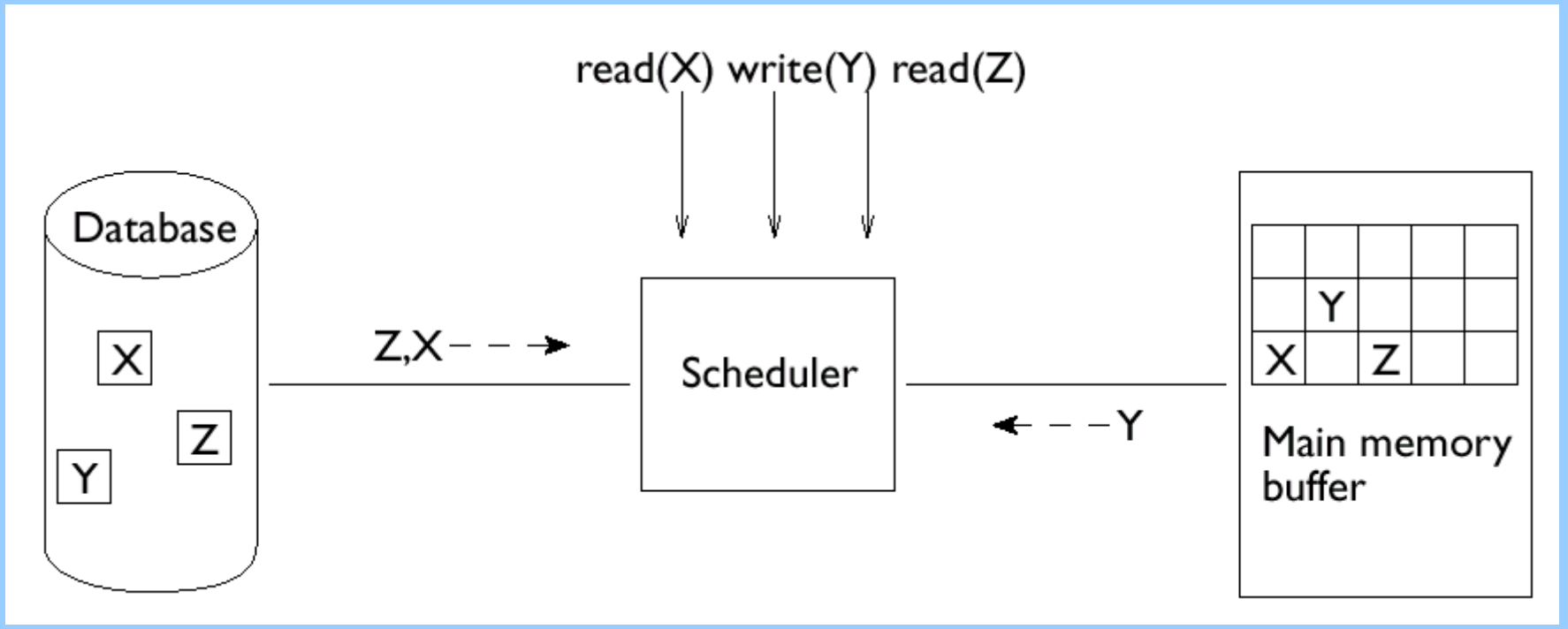
- Input-output operations on abstract objects *x*, *y*, *z*
- Each input-output operation reads secondary memory blocks into buffer pages or writes buffer pages into secondary memory blocks
- For simplicity: one-to-one mapping from blocks to pages

Problem

- Anomalies due to concurrent execution

The scheduler

- Traditionally, blocks are called pages once they are loaded into the memory



Anomaly 1: Update loss

- Consider two identical transactions:
  - $t1$: $r(x)$, $x = x + 1$, $w(x)$
  - $t2$: $r(x)$, $x = x + 1$, $w(x)$
- Assume initially x=2; after serial execution x=4
- Consider concurrent execution:
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
$x = x + 1$	
	`bot`
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
`commit`	
	$w_2(x)$
	`commit`

- One update is lost, final value x=3

Anomaly 2: Dirty read

- Consider the same two transactions, and the following execution (note that the first transaction fails):
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	`bot`
	$r_2(x)$
	$x = x + 1$
`abort`	
	$w_2(x)$
	`commit`

- Critical aspect: t_2 reads an intermediate state (dirty read)

Anomaly 3: Inconsistent read

- t_1 repeats two reads:
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
	`bot`
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	`commit`
$r_1(x)$	
`commit`	

- t_1 reads different values for x

Anomaly 4: Ghost update

- Assume the integrity constraint $x + y + z = 1000$;
  - Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
	`bot`
	$r_2(y)$
$r_1(y)$	
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	`commit`
$r_1(z)$	
$s = x + y + z$	
`commit`	

- In the end, $s = 1100$: the integrity constraint is not satisfied
- t_1 sees a ghost update

Concurrency control theory

- Transaction: sequence of read or write actions
- Each transaction has a unique, system-assigned transaction identifier
- Each transactions is initiated by the `begin transaction` command and terminated by `end transaction` (omitted)
- Example:

$$t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$$

This model omits any reference to the manipulation operations performed on the data by the transaction.

Notion of Schedule

- Represents the sequence of input/output operations presented by concurrent transactions
- Example:

$$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$$

- Simplifying assumption: we consider a commit-projection and ignore the transactions that produce an abort, removing all their actions from the schedule
- This assumption is not acceptable in practice and will be removed when we move to practical concurrency control methods.

Foundations of concurrency control

- *Objective*: refuse the schedules that cause the anomalies
- *Scheduler*: a system that accepts or rejects the operations requested by transactions
- *Serial schedule:* one in which the actions of all the transactions appear in sequence

  S_2 : $r_0(x)$ $r_0(y)$ $w_0(x)$ $r_1(y)$ $r_1(x)$ $w_1(y)$ $r_2(x)$ $r_2(y)$ $r_2(z)$ $w_2(z)$
- *Serializable schedule:* one that produces the same result as some serial schedule S_j of the same transactions
  - Requires a notions of equivalence between schedules
  - Progressive notions: *view-equivalence*, *conflict-equivalence*, *two-phase locking*, *timestamp-based*
- Observation: a scheduler allows the identification of a more or less wide-ranging class of acceptable schedules at the cost of testing for equivalence

View-Serializability

- Preliminary definitions:
  - $r_i(x)$ **reads-from** $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ in S and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$ in S
  - $w_i(x)$ in a schedule S is a **final write** if it is the last write of the object x to appear in S
- *View-equivalent schedules* ($S_i \approx_V S_j$): if they possess the same *reads-from* relation and the same final writes
- A schedule is called *view-serializable* if it is view-equivalent to some serial schedule
- The set of view-serializable schedules is called **VSR**

View-Serializability

- Complexity of view-serializability:
  - Deciding on the view-equivalence of two given schedules: done by a polynomial algorithm
  - Deciding on the view serializability of a generic schedule: NP-complete problem

It is thus preferable to define a more restricted condition of equivalence, which does not cover all the cases of view-equivalence betwenn schedules, but which is usable in practice, being less complex.

Examples of view serializability

- S_3 : $w_0(x)$ $r_2(x)$ $r_1(x)$ $w_2(x)$ $w_2(z)$
  S_4 : $w_0(x)$ $r_1(x)$ $r_2(x)$ $w_2(x)$ $w_2(z)$
  S_5 : $w_0(x)$ $r_1(x)$ $w_1(x)$ $r_2(x)$ $w_1(z)$
  S_6 : $w_0(x)$ $r_1(x)$ $w_1(x)$ $w_1(z)$ $r_2(x)$
  - S_3 is view-equivalent to the serial schedule S_4 (thus, it is view-serializable)
  - S_5 is not view-equivalent to S_4, but it is view-equivalent to the serial schedule S_6, and thus this also is view-serializable
- S_7 : $r_1(x)$ $r_2(x)$ $w_2(x)$ $w_1(x)$
  S_8 : $r_1(x)$ $r_2(x)$ $w_2(x)$ $r_1(x)$
  S_9 : $r_1(x)$ $r_1(y)$ $r_2(z)$ $r_2(y)$ $w_2(y)$ $w_2(z)$ $r_1(z)$
  - S_7 corresponds to update loss, S_8 to inconsistent reads and S_9 to ghost updates; they are not view-serializable

Conflict-serializability

- Preliminary definition:
  - Action a_i is in **conflict** with a_j ($i \neq j$), if both operate on the same object and at least one of them is a write. Two cases:
    - *read-write* conflicts (*rw* or *wr*)
    - *write-write* conflicts (*ww*).
- *Conflict-equivalent schedules* ($S_i \approx_C S_j$): if they present the same operations and each pair of operations in conflict is in the same order in both the schedules
- A schedule is therefore *conflict-serializable* if there is a serial schedule that is conflict-equivalent to it. The set of conflict-serializable schedules is called **CSR**
- CSR is properly included into VSR

conflict-serializable $\Rightarrow$ **view-serializable**

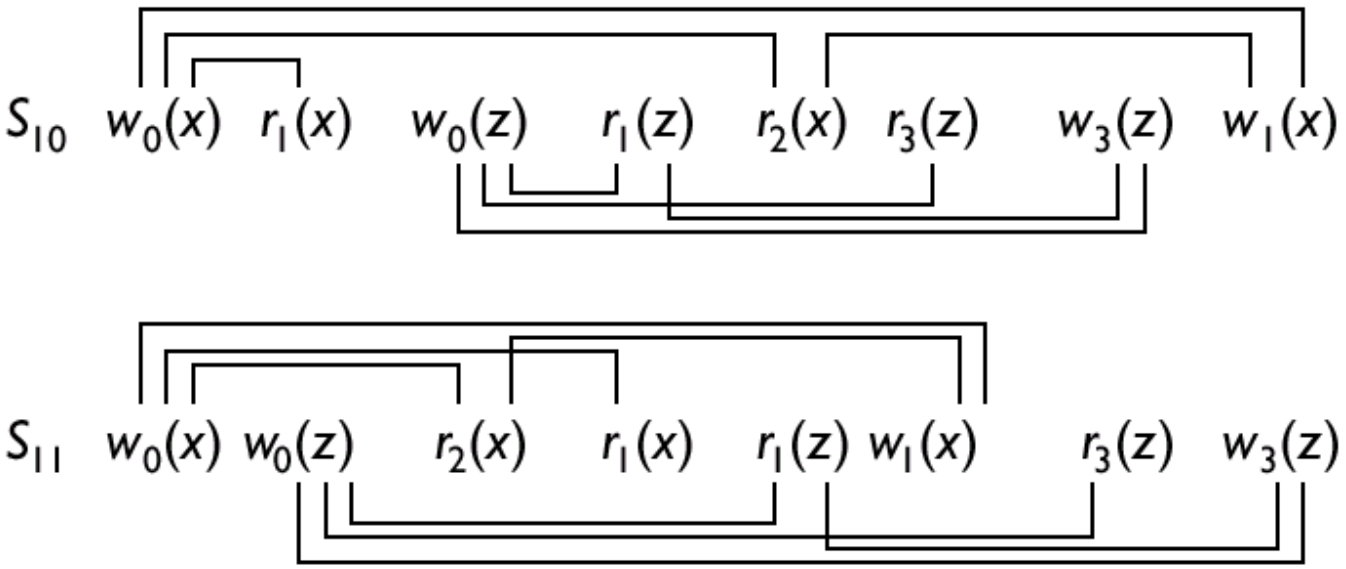
Testing conflict-serializability

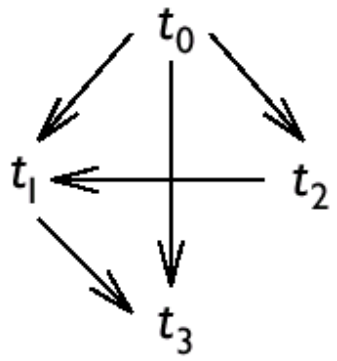
- By means of the *conflict graph*, with;
  - a node for each transaction t_i
  - an edge from t_i to t_j if there is at least one conflict between an action a_i and an action a_j such that a_i precedes a_j
- A schedule is in CSR if and only if the graph is acyclic
- Testing for cyclicity of a graph has a linear complexity with respect to the size of the graph itself
- Conflict serializability is still too laborious in practice, especially with data distribution

conflict-equivalence cannot be used in practice

Example of conflict-serializability

- S_{10} is conflic equivalent to the serial schedule S_{11}





Two-phase locking

- Used by almost all commercial DBMSs
- Principle:
  - All read operations preceded by **r_lock** (shared lock) and followed *unlock*
  - All write operations preceded by **w_lock** (exclusive lock) and followed *unlock*

  A transaction following these rules is *well formed wrt locking*
- When a transaction first reads and then writes an object it can:
  - Use a write lock
  - Move from shared lock to exclusive lock (*lock escalation*)
- The *lock manager* receives these primitives from transactions and grants resources according to conflict table
  - When the lock is granted, the resource is acquired
  - At the unlock, the resource is released

Behavior of the lock manager

- Based on conflict table:

Request	Resource state		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends	OK / free

  - A counter keeps track of the number of readers; the resource is released when the counter is set to zero.
- If a lock request is not granted, the requesting transaction is put in a waiting state
- The waiting ends when the resource is unlocked and becomes available
- The locks already granted are stored in a lock table, managed by the lock manager

Two-phase locking

- Serializability requires in addition that locking is in two phases: ***A transaction, after having released a lock, cannot acquire other locks***
- Two phases: growing and shrinking
- If a scheduler use well-formed transaction, conflict-based lock granting, and two-phases, then it produces the class of 2PL schedules
- 2PL schedules are serializable and strictly included into CSR
- Example of a schedule that is in CSR and not in 2PL:

  $S_{12} : r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$

Strict two-phase-locking

- We still need to remove the hypothesis of using a commit-projection
- To do so, turn to strict 2PL (by adding a constraint):
  The locks on a transaction can be released only after having carried out the commit/abort operations
- This version is used by commercial DBMSs; it eliminates the dirty read anomaly

Concurrency control based on timestamps

- New mechanism based on ***timestamp***:
  - an identifier that defines a total ordering of temporal events within a system

➢ Every transaction is assigned a timestamp *ts* that represents the time at which the transaction begins

➢ A schedule is accepted only if it reflects the serial ordering of the transactions based on the value of the timestamp of each transaction

Basic timestamp mechanism

- Each scheduler has a counter RTM(x) and WTM(x) for each object
- Each scheduler receives timestamped read and write requests upon objects:
  - *read*(x,*ts*): if *ts* < WTM(x) then the request is rejected and the transaction is killed, otherwise the request is accepted and RTM(x) is set equal to the greater of RTM(x) and *ts*
  - *write*(x,*ts*): if *ts* < WTM(x) or *ts* < RTM(x) then the request is rejected and the transaction is killed, otherwise the request is accepted and WTM(x) is set equal to *ts*

  Under assumption of commit-projection
- The method causes the forced abort of a large number of transactions
- To remove the commit-projection assumption, we must buffer the writes until commit, and this introduces waiting of transactions

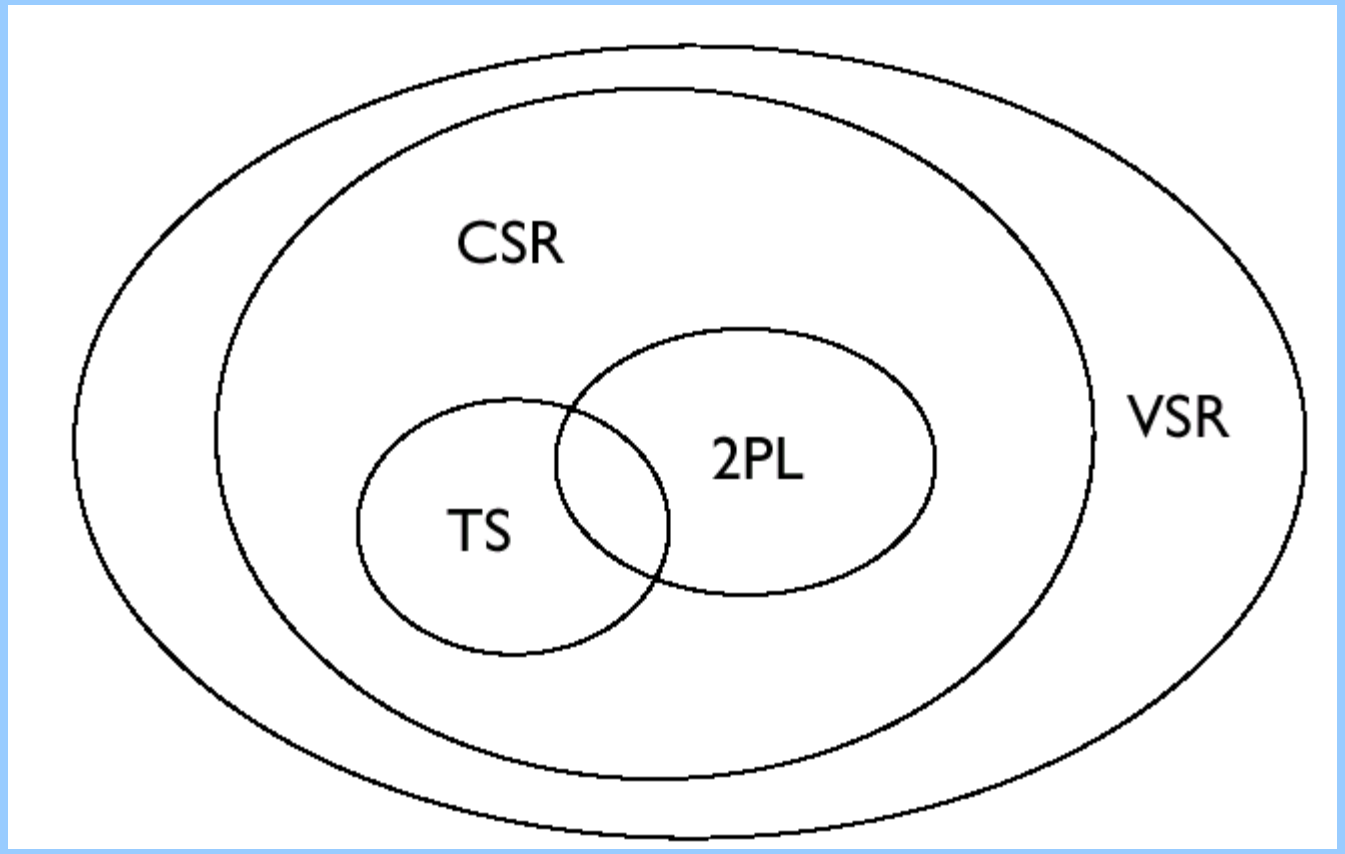
Example of timestamp-based concurrency control

Request	Response	New value
read(x,6)	ok	
read(x,8)	ok	RTM(x) = 8
read(x,9)	ok	RTM(x) = 9
write(x,8)	no	t_8 killed
write(x,11)	ok	WTM(x) = 11
read(x,10)	no	t_{10} killed

Multiversion concurrency control

- Main idea: writes generate new copies, reads make access to the correct copy
- Writes generate new copies each with a WTM. At any time, $N>1$ copies of each object x are active, with $\text{WTM}_N(x)$. There is only one global $\text{RTM}(x)$
- Mechanism:
  - *read*(x,*ts*): is always accepted. The copy x_k is selected for reading such that: if $ts > \text{WTM}_N(x)$, then $k = N$, otherwise k is taken such that $\text{WTM}_k(x) < ts < \text{WTM}_{k+1}(x)$
  - *write*(x,*ts*): if $ts < \text{RTM}(x)$ the request is refused, otherwise a new version of the item of data is added *(N* increased by one) with $\text{WTM}_N(x) = ts$
- Old copies are discarded when there are no read transactions interested in their values

Taxonomy of VSR, CSR, 2PL and TS



Comparison 2PL vs TS

- In 2PL the transactions are put in waiting. In TS they are killed and then restarted
- The serialization order in 2PL is imposed by conflicts, while in TS it is imposed by the timestamps
- The necessity of waiting for the commit of the transaction causes strict 2PL and buffering of writes in TS
- 2PL can give rise to deadlocks (discussed next)
- Restarting costs more than waiting: 2PL wins!

Lock management

It is component of the DBMS, used by all the processes that access the database.

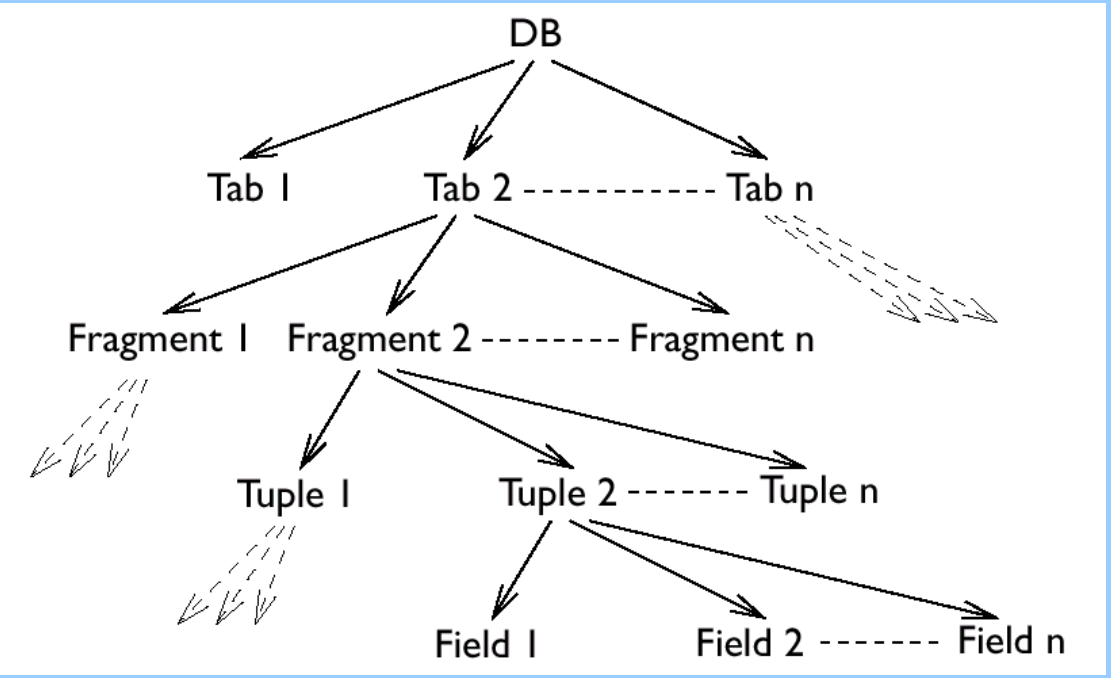
- Interface based on three procedures:
  - `r_lock(T, x, errcode, timeout)`
  - `w_lock(T, x, errcode, timeout)`
  - `unlock(T, x)`

  `T`: transaction identifier
  `X`: data element
  `timeout`: max wait in queue
- If timeout expires, `errcode` signals an error, typically the transaction rolls back and restarts

Hierarchical locking

- In many real systems locks can be specified at different granularity, e.g. tables, fragments, tuples, fields. These are organized in a hierarchy (possibly a directed acyclic graph)
- 5 locking modes:
  - 2 are shared and exclusive, renamed as:
    - XL: exclusive lock
    - SL: shared lock
  - 3 are new:
    - ISL: intention shared lock
    - IXL: intention exclusive lock
    - SIXL: shared intention-exclusive lock
- The choice of lock granularity is left to application designers;
  - too coarse: many resources are blocked
  - too fine: many locks are requested

Hierarchical locking example



If we wish to place a write lock on a tuple of the table, then we must first request an IXL on the database level. When the request is satisfied, we can request an IXL for the relation and fragment in which the desired tuple lies. When these locks are granted, we can request an XL for the particular tuple. Then when the transaction is ended, it will have to release the locks in reverse order to that in which they were granted, ascending the hierarchy one step at a time.

Hierarchical locking protocol

1. Locks are requested from the root to descendents in a hierarchy
2. Locks are released starting at the node locked and moving up the tree
3. In order to request an SL or ISL on a node, a transaction must already hold an ISL or IXL lock on the parent node
4. In order to request an IXL, XL, or SIXL on a node, a transaction must already hold an SIXL or IXL lock on the parent node
5. The new conflict table is shown in the next slide

Conflicts in hierarchical locking

Request \ Resource state	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Lock options offered by SQL2

- Some transactions are defined as `read-only` (they can't request exclusive locks)
- The level of isolation can be set for each transactions
- `Serializable` guarantees max isolation: keeps predicate locks so as not to change the content even of aggregate functions evaluated on data sets
- `Repeatable read` is equal to strict 2PL (note: repeated reading of values are the same, but repeated readings of aggregates over data are not)
- `Committed read` excludes the reading of intermediate states (uncommitted data)
- `Uncommitted read` does no concurrency control at all on read

Deadlocks

- Created by concurrent transactions, each of which holds and waits for resources held by others
- Example:
  - t_1: *read*(x), *write*(y)
  - t_2: *read*(y), *write*(x)
  - Schedule:

    $r_lock_1(x)$, $r_lock_2(y)$, $read_1(x)$, $read_2(y)$ $w_lock_1(y)$, $w_lock_2(x)$
  - This is deadlock! $T_1$is waiting for is waiting for *y,* which is blocked by t_2, and in its turn, $t_2$is waiting for x, which is locked by t_1.
- Deadlock probability grows linearly with number of transactions and quadratically with the number of lock requests of each transaction (under suitable uniformity assumptions)

Deadlock resolution techniques

- A deadlock is a cycle in the *wait-for* graph which indicates wait conditions between transactions
  - (node=transaction, arc=wait condition).
- Three techniques:
  1. Timeout (problem: choice of timeout with trade-offs)
  2. Deadlock detection
  3. Deadlock prevention
- Deadlock detection: performs the search for cycles in a wait-for graph
- Deadlock prevention: kills the transactions that could cause a cycle (thus: it overkills)
  - Options for choosing the transaction to kill:
    - via *pre-emptive* policies or *non-pre-emptive* policies