DATA BASES 2

Federico Mainetti Gambera

6 ottobre 2020

Indice

1 Introduction

Introduction slides:
`../other/professor'sslides/0-Intro.pdf`

1.1 Agenda and Links to Recordings

`https://polimi365-my.sharepoint.com/:x:/g/personal/10046017_polimi_it/ETbz2LtioM5AkIe8H8g41Y4BRFsXOl640ZklgKpT7KH3hg`

1.2 Prerequisites: References to Relational Algebra and SQL

1.2.1 Relational Algebra

`../other/RelationalAlgebraAndSQL/04-RelationalAlgebra.pdf`

`../other/RelationalAlgebraAndSQL/05-RelationalAlgebra2.pdf`

1.2.2 SQL

`../other/RelationalAlgebraAndSQL/08-SQL1.pdf`

`../other/RelationalAlgebraAndSQL/09-SQL2.pdf`

`../other/RelationalAlgebraAndSQL/10-SQL3.pdf`

1.3 Books

All books can be found at:
`../other/books`

2 Transactional Systems

../other/professor'sslides/1_Transactions_NEW.pdf

A **transaction** is an elementary, atomic, unit of work performed by an application. Each transaction is conceptually encapsulated within two commands:

- **begin of transaction** (bot);
- **end of transaction** (eot).

Within a transaction, **one** of the commands below is executed (**exactly once**) to signal the end of the transaction:

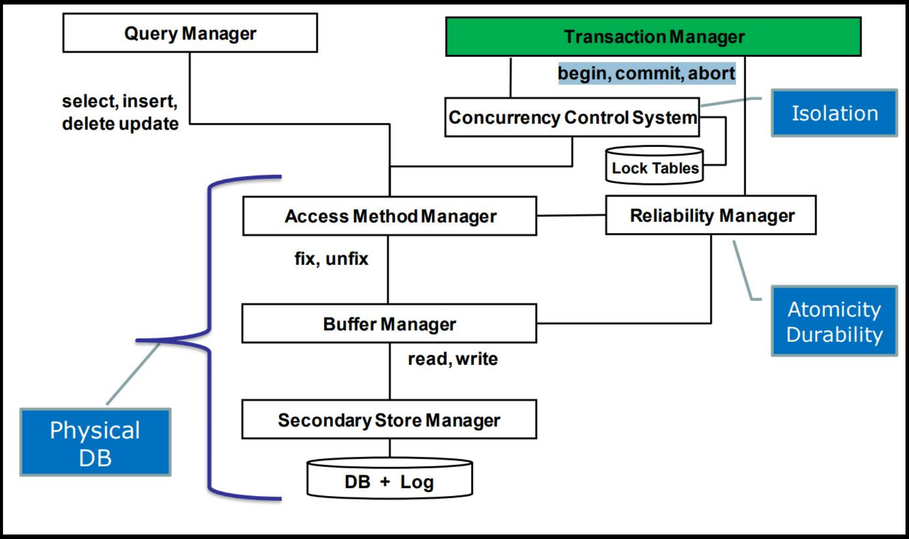
- **commit**;
- **rollback**.

Transactional System (OLTP) is a system that supports the execution of transactions on behalf of concurrent applications.

A transaction is a unit of work enjoying the **ACID** properties:

- **Atomicity** (abort-rollback-restart, commit protocols): either all the operations in the transaction are executed or none is executed;
- **Consistency** (integrity checking of the DBMS): a transaction must satisfy the DB integrity constraints, if the initial state is consistent then the final state is also consistent, but this is not necessarily true for the intermediate states of the transaction;
- **Isolation** (concurrency control): the concurrent execution of a number of transaction must produce the same result as the execution of the same transactions in a sequence;
- **Durability** (recovery management): the effect of a transaction that has successfully committed will last "forever".

Lets take a look at where are the ACID properties managed in the DBMS architecture:



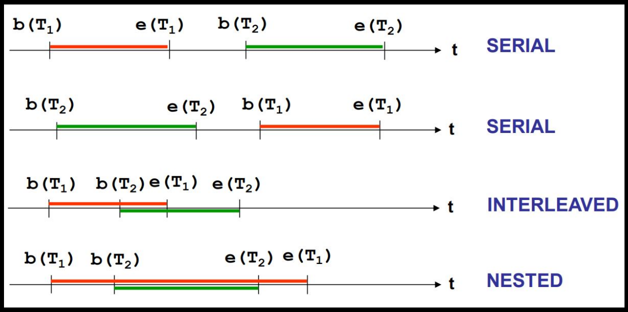
3 Concurrency Control

`../other/professor’sslides/2-ConcurrencyControl.pdf`

The concurrency control system:

- manage the simultaneous execution of transactions;
- avoids the insurgence of anomalies;
- ensure performance: exploit parallelism to maximise transactions per second.

Two transaction can have a serial execution, an interleaved execution or a neasted execution.



3.1 Anomalies

An anomalie happens when two transaction are executed interleaved or nested and the result is different if they were executed serially.

3.1.1 Lost Update

$$r_1(x) \rightarrow r_2(x) \rightarrow w_1(x) \rightarrow w_2(x)$$

An update is applied from a state that ignores a preceding update, wich is lost.

3.1.2 Dirty Read

$$r_1(x) \rightarrow w_1(x) \rightarrow r_2(x) \rightarrow abort_1 \rightarrow w_(2)$$

An uncommitted (aborted transaction) value is used to update the data.

3.1.3 Nonrepeatable Read

$$r_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x)$$

Someone else updates a previously read value: inside of the same transaction a data is read and, after a while, without changing the value of the data, if we try to read it again, now its a different value.

3.1.4 Phantom Update

$$r_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_1(x)$$

Someone else updates data that contributes to a previously read datum: a transaction reads multiple data, but at different times. There is inconcistency in the time of when the data that contributes to the constraints have been read.

This anomaly is also known as "contraint violation".
Lets see an example: given the costraint $A + B + C = 100$ and the initial values $A = 50, B = 30, C = 20$
$T_1 : r(A), r(B)$
$T_2 : r(B), r(C)$
T_2 : subtract 10 from C and add 10 to B.
$T_2 : w(B), w(C)$
$T_1 : r(C)$ (T_1 reads that the sum is 90).
So for T_1 it is as if "somebody else" had updated the value o the sum, but for T_2 the update is perfectly legal (does not change the value of the sum).

3.1.5 Phantom Insert

$$r_1(x) \rightarrow w_2(new\ tuple) \rightarrow r_1(x)$$

Someone else inserts data that contributes to a previously read datum.

This anomaly does not depend on data already present in the DB when T_1 executes, but on a "phantom" tuple that is inserted by "someone else" and satisfies the conditions of a previous query of T_1.

3.2 Principles of Concurrency Control

Operations: a read or write of a specific datum by a specific transaction.

Schedule: a sequence of operations performed by concurrent transactions that respects the order of operations of each transaction.
How many distinct schedules exist for n transaction each with k_i operations?

$$N_D = \frac{(\sum_{i=1}^{n} k_i)!}{\Pi_{i=1}^{n}(k_i!)}$$

How many of them are serial?

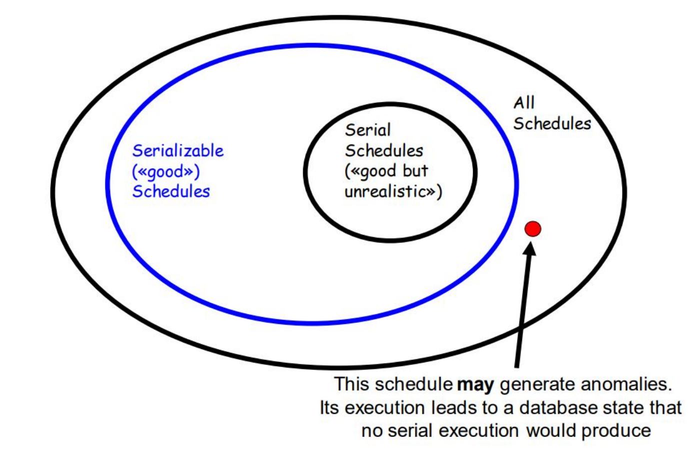
$$N_S = n!$$

Scheduler: a component that accepts or rejects the operations requested by the transactions.

Serial schedule: a schedule in which the actions of each transaction occur in a contiguos sequence.

3.2.1 Serializable schedule

A **serializable schedule** is a schedule that leaves the database in the **same state** as **some** serial schedule of the same transactions.


All Schedules
Serial Schedules («good but unrealistic»)
Serializable («good») Schedules

This schedule **may** generate anomalies. Its execution leads to a database state that no serial execution would produce

3.2.2 View-serializability

$r_i(x)$ **reads-from** $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ and there is no $w_k(x)$ in S between $r_i(x)$ and $w_j(x)$.

$w_i(x)$ in a schedule S is a **final write** if it is the last write on x that occurs in S.

Two schedules are **view-equivalent** if they have the same operations, the same reads-from relation, and the same final writes.

A schedule is **view-serializable** if it is view-equivalent to a serial schedule of the same transactions.
The class of view-serializable schedules is named **VSR**.
S is **view-serializable** if:

1. every read operation sees the same values;
2. the final value of each object is written by the same transaction as if the transactions were executed serially in some order.

Deciding if a generic schedule is in VSR is an **NP-complete problem** but is an heavy task, so we look for a stricter definition that is easier to check, even if we reject sosme schedule that would be acceptable: VSR schedules are "too many".

3.2.3 Conflict-serializability

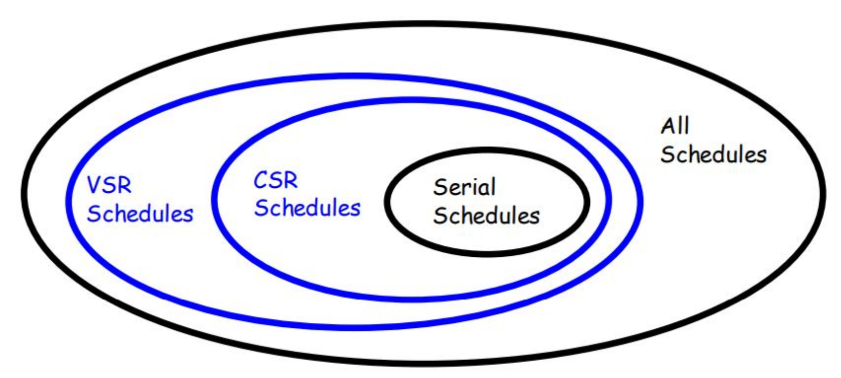
Two operations o_i and o_j are in **conflict** if they address the same resource and at least one of them is a write: **read-write** conflicts (r-w or w-r) and **write-write** conflicts (w-w).

Two schedules are **conflict-equivalent** if contain the same operations and in all conflicting pairs transactions occur in the same order.

A schedule is **conflict-serializable** if it is conflict-equivalent to a serial schedule of

the same transactions.
The class of conflict-serializable schedules is named **CSR**.

All conflict-serializable schedules are also view-serializable, but the inverse is not necessarily true: $CSR \rightarrow VSR$.



3.2.4 Testing conflict-serializability

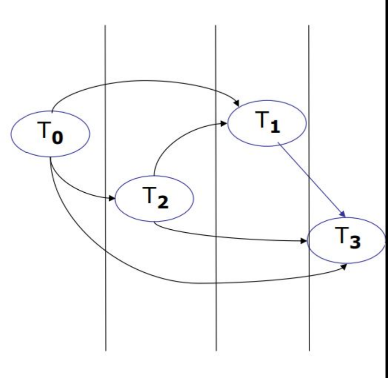
To check if a schedule is conflict-serializable we use a **conflict graph**:

- one node for each transaction T_i;
- one arc from T_i to T_j if there exists at least one conflict between an operation o_i of T_i and an operations o_j of T_j such that o_i precedes o_j;
- a schedule is in CSR if and only if its conflict graph is **acyclic**.

To check if a graph is acyclic we use this algorithm:

- if a graph is acyclic, then it must have at least one node with no targets, called leaf (with no arrows going away from the node).
- if we "peel off" a leaf in an acyclic graph, then we are always left with an acyclic graph, and if we keep peeling off leaf nodes, one of two things will happen: we will eventually peel off al node and so the graph is acyclic, or we will get to a point where there is no leaf, yet the graph is not empty and so the graph is cyclic.

If S's graph is acyclic then it induces a **topological (partial) ordering** on its nodes, that is easier to understand graphically with an example:



In general, there can be many compatible serial schedules.

3.3 Concurrency control based on locking

Everything we have seen untill now is based on study of schedules after they have been executed. In real life we need a system that can assure cocnurrency control in "live". There are two main families of techniques:

- Pessimistic: based on locks, resource access control.
- Optimistic: based on timestamps and versions, serve as many requests as possible, possibly using out-of-date versions of the data.

3.3.1 Locking

A transaction is **well-formed w.r.t. locking** if

- read operations are preceded by **r_lock** (shared lock SL) and followed by **unlock**;
- write operations are preceded by **w_lock** (exclusice lock XL) and followed by **unlock**.

Transactions that first read and then write an object may:

- acquire a w_lock already when reading or
- acquire a r_lock first and then upgrade it into a w_lock (lock **escalation**).

Possible states of an object:

- **free**;
- **r-locked** (locked by one or more readers);
- **w-locked** (locked by a writer);

When a lock request is granted, the resource is acquired; when an unlock is executed, the resource becomes available.
The lock manager grants access to resources according to the **conflict table**:

	FREE	R_LOCKED	W_LOCKED
r_lock	OK (R_LOCKED)	OK (R_LOCKED n++)	NO (W_LOCKED)
w_lock	OK (W_LOCKED)	NO (R_LOCKED)	NO (W_LOCKED)
unlock	ERROR	OK (DEPENDS n- -)	OK (FREE)

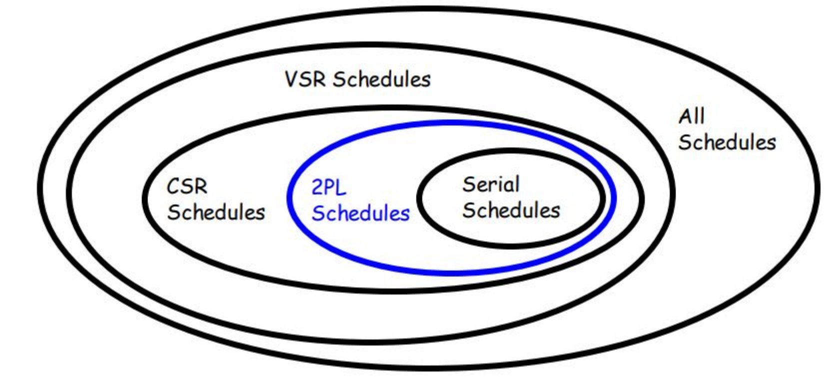
where n is the counter of the current readers.

3.3.2 2PL

Is respecting locks enough for serializability? No. For example non repeatable read anomalies can still occur and to prevent it we use the **Two-Phase Locking (2PL)**: the requirement (two-phase rule) is that **"a transaction cannot acquire any other lock after releasing a lock"**. Doing so we prevent **non repeatable reads**.

A scheduler that only processes well-formed transactions, grants locks according to the conflict table and checks that all transactions apply the two-phase rule generetes a class of schedules called **2PL**.

Schedules in 2PL are view- and conflict-serializable. 2PL impies CSR.



3.3.3 Strict 2PL

Up to now, we were still using the **hypothesis of commit-projection** (no transactions in the schedule abort).
2PL, as seen so far, does not protect against **dirty** (uncommitted data) **reads** (and therefore neither do VSR nor CSR): releasing locks before rollbacks exposes "dirty" data.

To remove this hypothesis, we need to add a constraint to 2PL, that defines **strict 2PL**: **Locks held by a transaction can be released only after commit/rollback**.

Strict 2PL locks are also called **long duration locks**, 2PL locks **short duration locks**.

A phantom insertion occurs when a transaction adds items to a data set previously read by another transaction. To prevent phantom inserts a lock should be placed also on "future data".

Predicate locks extend the notion of **data locks** to "future data". Predicate locks are a tool to lock data that doesn't already exist.

Write locks are held until the completion (commit) of the transaction, and not only untill the end of the write, to enable the proper processing of abort events.

2PL and locking have two foundamentals principles:

- **same transaction i**: a transaction consists of a growing phase (locking resources), a plateau (operations) and a shrinking phase (releasing locks). Because of the 2PL rule we can say that all the locks must happen before the unlocks:

$$L_i^? < U_i^?$$

- **same resource R**: if we analyze a resource, we can say that before being able to lock a resource, it must be unlocked by the precedent transaction that was locking it:

  $$U_?^R < L_?^R$$

  this constraint is not a 2PL rule, but is necessary to have a working locking system.

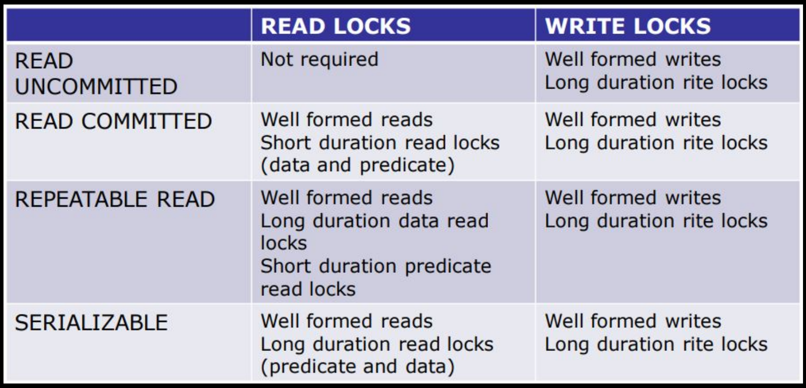
3.3.4 Isolation Levels in SQL:1999 (and JDBC)

Isolation Levels in SQL:1999 (and JDBC):

- READ UNCOMMITTED allows dirty reads, nonrepeatable reads and phantom updates and inserts: No read locks (and ignores locks of other transactions).
- READ COMMITTED prevents dirty reads but allows nonrepeatable reads and phantom updates/inserts: Read locks (and complies with locks of other transactions), but without 2PL on read locks (read locks are released as soon as the read operation is performed and can be acquired again).
- REPEATABLE READ avoids dirty reads, nonrepeatable reads and phantom updates, but allows phantom inserts: long duration read locks → 2PL also for reads.
- SERIALIZABLE avoids all anomalies: 2PL with predicate locks to avoid phantom inserts

The locking requirements to meet this guarantee can frequently lead to deadlock where one of the transactions needs to be rolled back. Therefore SERIALIZABLE isolation level is used sparingly and is NOT the default in most commercial systems.

SQL Isolation levels may be implemented with the appropriate use of lock.
The isolatioon levels define the way read lock are managed, write lock are always managed in strict 2PL (because it is the only way to handle aborts).



	READ LOCKS	WRITE LOCKS
READ UNCOMMITTED	Not required	Well formed writes Long duration rite locks
READ COMMITTED	Well formed reads Short duration read locks (data and predicate)	Well formed writes Long duration rite locks
REPEATABLE READ	Well formed reads Long duration data read locks Short duration predicate read locks	Well formed writes Long duration rite locks
SERIALIZABLE	Well formed reads Long duration read locks (predicate and data)	Well formed writes Long duration rite locks

3.3.5 Deadlock

A resource can be free, read-locked or write-locked.

Transactions requesting locks are either **granted the lock** or **suspended and queued** (first-in first-out). There is risk of:

- **deadlock**: two or more transactions in endless (mutual) wait (Typically occurs when each transaction waits for another to release a lock).
- **starvation**: a single transaction in endless wait (Typically occurs due to write transactions waiting for resources that are continuously read).

Lock graph: a bipartite graph in which nodes are resources or transactions and arcs are lock requests or lock assignments.

Wait for graph: a graph in which nodes are transactions and arcs are "waits for" relationships.
A deadlock is represented by a cycle in the wait-for graph of transactions.

Deadlock resolution techniques:

- **Timeout**: transaction killed and restarted after a given amount of waiting;
- **Deadlock prevention**: transaction killed when they could be in deadlock.
  - **Resource-based prevention**: transactions request all resources at once, and only once; resources are globally sorted and must be requested in that order.
  - **Transaction-based prevention**: assigning IDs to transactions incrementally (transactions' "age") and preventing older transactions from waiting for younger ones to end their work.
- **Deadlock detection**: transaction killed when they are in deadlock. Requires an algorithm to detect cycles in the wait-for graph.

3.3.6 Deadlock detection

The deadlock detectiton requires an algorithm to detect cycles in the wait-for graph.

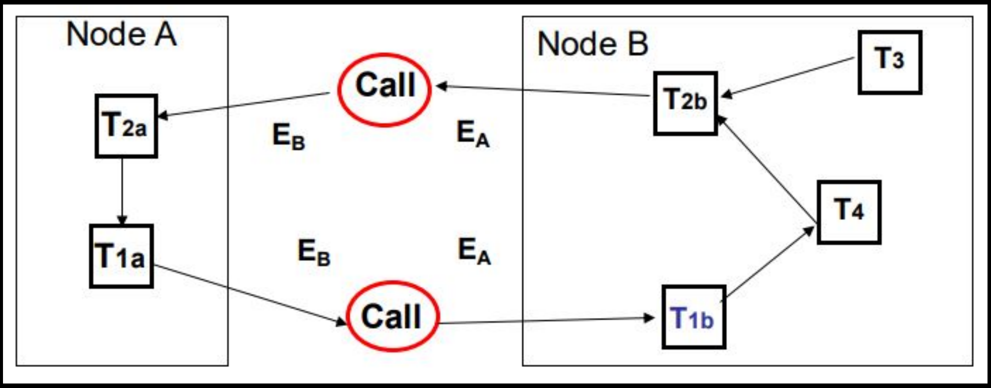
Assumptions:

- Transactions execute on a single main node (one locus of control);
- Transactions may be decomposed in "sub-transactions" running on other nodes;
- Synchronicity: when a transaction spawns a sub-transaction it suspends work until the latter completes;
- Two waits-for relationships: (1) T_i waits from T_j on the same node because T_i needs a datum locked by T_j; (2) a sub-transaction of T_i waits for another sub-transaction of T_i running on a different node.

Because of the macanism of sub-transactions we need a distributed dependency graph, that is a global view of all the dependencies in all the differents nodes, where external call nodes represent a sub-transaction activating another sub-transaction at a different node.

We will use the symbol "$\rightarrow$" as a "wait for" relation among local transactions.

Lets see an example of deadlock (cycle) in a distributed dependency graph:



We can see a cycle between two differents node (A and B) in $T_{2a} \rightarrow T_{1a} \rightarrow T_{1b} \rightarrow T_{2b} \rightarrow T_{2a}$.

The problem is how we can detect such occurance without maintaining the global view of the distributed dependeny graph but keeping it as much possible local for each node. To do so we need to establish a communication protocol whereby each node has a local projection of the global dependencies. Nodes exchange information and update their local graph based on the received information and communication is optimized to avoid that multiple nodes detect the same potential deadlock.

Node A sends its local info to a node B only if

- A contains a transaction T_i that is waited from another remote transaction and waits for a transaction T_j active on B;
- $i > j$ (this ensure a kind of message "forwarding" along a node path where node A "precedes" node B if $i > j$).

Mnemonically: I send info to you if a d-transaction listed at me waits for a d-transaction listed at you with smaller index.

3.3.7 Obermark's algorithm

The Obermark's algorithm runs periodically at each node and consists of four steps:

- Get graph info (wait dependencies among transactions and external calls) from the "previous" nodes. Sequences contain only node and top-level transaction identifiers
- Update the local graph by merging the received information
- Check the existence of cycles among transactions denoting potential deadlocks: if found, select one transaction in the cycle and kill it
- Send updated graph info to the “next” nodes

3.3.8 Update lock

The most frequent deadlock occurs when two concurrent transactions start by reading the same resources (SL) and then decide to write and try to upgrade their lock to write (XL).

To avoid this situation, systems offer the **update lock** (UL): is a special lock asked by transactions that will read and then write an item..

Here it is the **table of conflict** with the update lock:

		resource status	
request	SL	UL	XL
SL	OK	OK	NO
UL	OK	NO	NO
XL	NO	NO	NO

The price for using update locks is that they extend the interval during which a resource is locked.

3.3.9 Hierarchical locking

Locks can be specified with different granularities (schema, table, fragment, page, tuple, field) and we have two objectives: locking the minimum amount of data and recognizing conflicts as soon as possible. We obtain this by requesting resources top-down until the right level is obtained and releasing locks bottom-up.

The **intention locking scheme** is a mode to express the intention of locking at lower levels of granularity, there are 3 more lock modes in addition to SL (read shared locks) and XL (write exclusive locks):

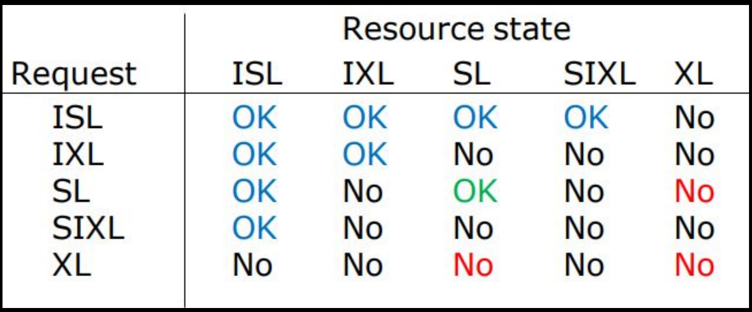
- **ISL**: Intention of locking a subelement of current element in shared mode
- **IXL**: Intention of locking a subelement of current element in exclusive mode
- **SIXL**: Lock of the element in shared mode with intention of locking a subelement in exclusive mode (SL+IXL)

Hierarchical locking:

- Locks are requested starting from the root (the whole table) and going down in the hierarchy;
- Locks are released starting from the leaves and going up in the hierarchy
- To request an SL or ISL lock on a non-root element, a transaction must hold an equally or more restrictive lock (ISL or IXL) on its ”parent”
- To request an IXL, XL or SIXL lock on a non-root element, a transaction must hold an equally or more restrictive lock (SIXL or IXL) on its ”parent”

- When a lock is requested on a resource, the lock manager decides based on the rules specified in the hierarchical lock granting table

Hierarchical lock granting table:



	Resource state				
Request	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

3.4 Concurrency control based on timestamps

Locking is also named pessimistic concurrency control because it assumes that collisions will arise, alternative are the optimistic concurrency control methods like concurrency control based on timestamps.

Timestamp: Identifier that defines a total ordering of events of a system.

Each transaction has a timestamp representing the time at which the transaction begins so that transactions can be ordered by "birth date".

3.4.1 Timestamps in distributed systems

How do we assign timestamps in distributed systems where there is no "global time" available? A system's function gives out timestamps on requests.
We can now use an algorithm, known as "Lamport method": cannot receive a message from "the future", if this happens the "bumping rule" is used to bump the timestamp of the receive event beyond the timestamp of the send event.

Mnemonically: if I receive a message from you that has a timestamp greater than my last emitted one I update my current timestamp to exceed yours.

3.4.2 Timestamp principles

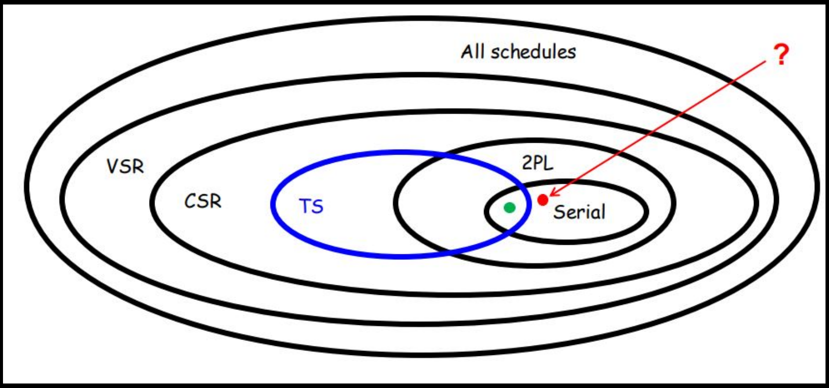
The scheduler has two counters: RTM(x) and WTM(x) for each object.

The scheduler receives read/write requests tagged with timestamps:

- read(x,ts): if ts $<$ WTM(x) the request is rejected and the transaction is killed, else access is granted and RTM(x) is set to max(RTM(x),ts);

- write(x,ts): if ts $<$ RTM(x) or ts $<$ WTM(x) the request is rejected and the transaction is killed, else access is granted and WTM(x) is set to ts.

This approach uses a commit-projection hypothesis, meaning that there are no abort and each transaction is committed.
To work without this hypothesis write operations should be buffered until commit, in this way aborted transactions do not even issue write requests and the commit-projection hypothesis is equivalent to a full schedule with committed and abored transactions.



Notice how some serial transaction aren't present in the timestamp approach.

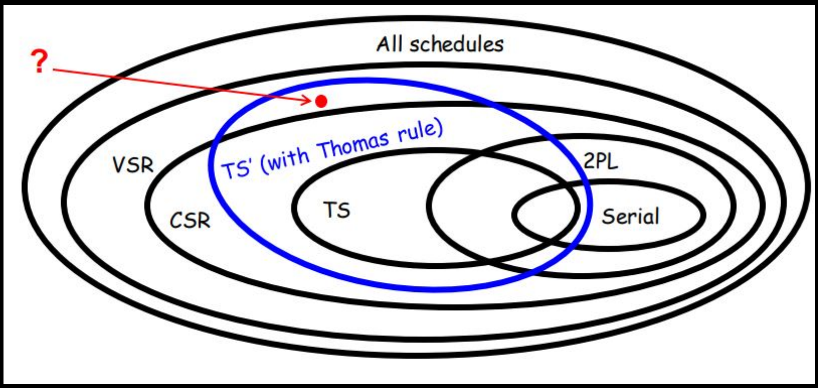
In 2PL transactions can be actively waiting. In TS they are killed and restarted. Restarting a transaction costs more than waiting: 2PL wins!

3.4.3 Thomas Rule

A variant of the basic timestamp-based concurrency control is the Thomas Rule, where the only thing that change is the write(x,ts) rule:

- if ts < RTM(x) the request is rejected and the transaction is killed, else if ts < WTM(x) then our write is "obsolete": it can be skipped.

Skipping a write on an object that has already been written by a younger transaction, without killing the transaction, works only if the transaction issues a write without requiring a previous read on the object (so SET X = X+1 would fail).



3.4.4 Multiversion concurrency control

Idea: writes generate new copies, reads access the "right" copy.

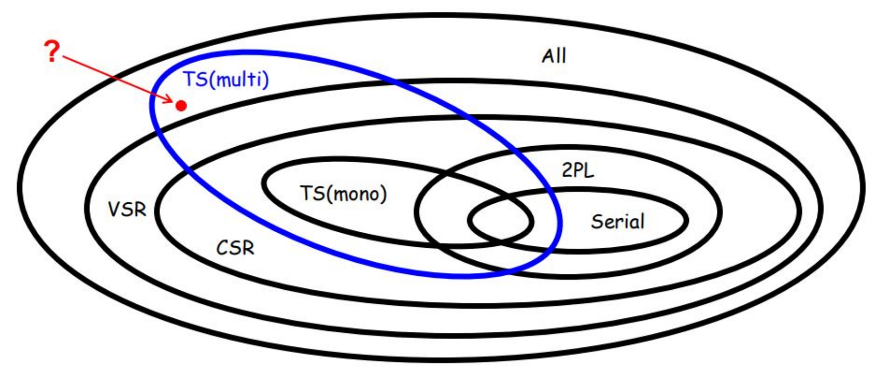
Writes generate new copies, each one with a new WTM. Each object x always has n>=1 active copies with WTMn(x). There is a unique global RTM(x).

Old copies are discarded when there are no transactions that need these values.

Mechanism:

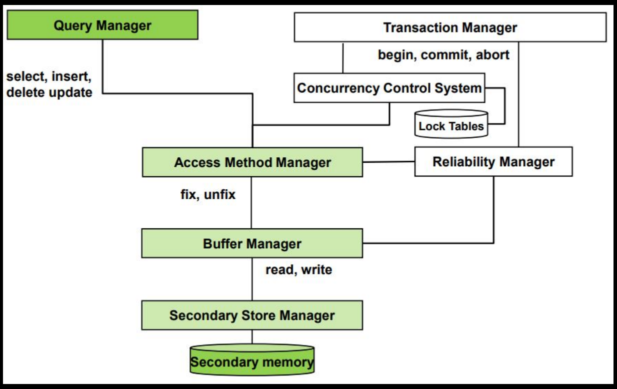
- read(x,ts): is always accepted, a copy x_k is selected for reading such that if ts > WTMn(x), then k = n, else take k sucj that WTMk(x) ≤ ts < $WTM_{k+1}(x)$.

- write(x,ts): if ts $<$ RTM(x) the request is rejected, else a new version is created (n is incremented) with WTMn(x) $=$ ts.



4 Physical data structures and query optimization

../other/professor'sslides/3_PhysicalDatabases.pdf



4.1 Data Access and Cost Model

4.1.1 Main and secondry memory

Data bases have two memories: **main memory** and **secondary memory**.

Databases must be stored mainly in files onto secondary memory for two reasons: size and persistance.

Data stored in secondary memory can only be used if first transferred to main memory.

Second memory devices are organized in **blocks** of **fixed** length.

The cost of an access to secondary memory is 4 orders of magnitude higher than that to main memory.

4.1.2 DBMS and file system

File System (FS): component of the OS which manages access to secondary memory.

DBMSs make limited use of FS functionalities: the DBMS directly manages the file organization, both in terms of the distribution of records within blocks and with respect to the internal structure of each block. A DBMS may also control the physical allocation of blocks onto the disk for faster sequential reads.

4.2 Physical access structures

Access methods: software modules that provide data access and manipulation (store and retrieve) **primitives** for each physical access structure.

Access methods have their own data structures to organize data:

- each table is stored into **exactly one primary** physical access structure;

- each table may have **one or more optional secondary** access structure.

Primary structure: it contains all the tuples of a table. The main purpose is to store the table content.

Secondary structures: are used to index primary structure, and only contain the values of some fields, interleaved with pointers to the blocks of the primary structure. The main purpose is to speed up the search for specific tuples, according to some earch criterion.

Three main **types of data access structures**:

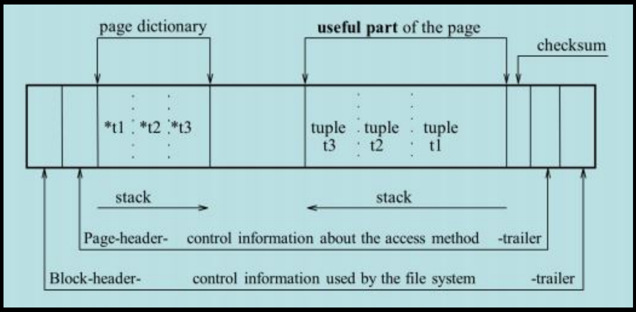
- **Sequential** structures;
- **Hash-based** structures;
- **Tree-based** structures.

4.2.1 Blocks and tuples

Block: the "physical" components of files. The size of a block is typically fixed and depends on the file system and on how the disk is formatted.

Tuples: the "logical" components of tables. The size of a tuple (also called record) depends on the database design and is typically variable within a file.

Organization of tuples within blocks/pages:



- Block header and trailer with control information used by the file system;
- Page header and trailer with control information about the access method;
- A page dictionary, which contains pointers (offset table) to each elementary item of useful data contained in the page;
- A useful part, which contains the data;
- A checksum, to detect corrupted data.

Block factor (B): the number oftuples within a block

$$B = floor\left(\frac{\text{size of blok}}{\text{avarage size of a tuple}}\right)$$

4.2.2 Page manager primitives

- **Insertion and update of a tuple**: may require a reorganization of the page or usage of a new page;
- **deletion of a tuple**: often carried out by marking the tuple as "invalid";
- **access to a field of a particular tuple**: identified according to an offset w.r.t. the beginning of the tuple and the lenght of the field itself.

4.2.3 Sequential access structures

Sequential structures is a sequential arrangement of tuples in the secondary memory.

Three cases:

- **Entry-sequenced** organization: sequence of tuples dictated by their order of entry. Optimal for space occupancy and carrying out sequntial reading and writing, non optimal with respect to searching specific data units.
- **Array** organization: the tuples are arranged as in array, they can be accessed through an index. Possible only when tuples are of fixed lenght.
- **Sequentially-ordered** organization: tuples ordered according to the value of a key (one or more attributes). The main problem are related to insertion of new tuples and updates, reordering techniwques are needed for the tuples already present.

4.2.4 Hash-based access structures

4.2.5 Tree-Based access structures