SQL -DDL (Data Definition Language)

SQL – Structured Query Language, originally called SEQUEL (**S**tructured **E**nglish **QUE**ry **L**anguage). It is the standard language for DBMSs. SQL is:

- **DDL (Data Definition Language)** – for defining data
- **DML (Data Manipulation Language)** – for manipulating data

Also having a way for: defining views, specifying security, defining integrity constraints, transaction control, rules for embedding SQL statements in general purpose languages like C, C++.

DDL – has the commands **CREATE**, **DROP**, **ALTER** for the descriptions of the tables(relations) in the database.

CREATE SCHEMA schema_name AUTHORIZATION owner_name – specifies a new DB schema with a name and also adding the owner's name of the DB schema.

Data types can be:

- Date (**DATE**)
- Logical (**BOOLEAN**)
- Character (**CHAR (n)**, **VARCHAR(n)**, **NVARCHAR(n)**)
- Bit-String (**BIT**(n), **BLOB**, **CLOB**)
- Numerical (**INT**, **INTEGER**, **FLOAT**, **REAL**, **DECIMAL (I, j)**, **NUMERIC (I, j)**)

I-total size, j-floating points size

```
CREATE TABLE DEPT (
   DNAME        VARCHAR(10)       NOT NULL,
   DNUMBER      INTEGER           NOT NULL,
   MGRSSN       CHAR(9),
   MGRSTARTDATE CHAR(9),
   PRIMARY KEY (DNUMBER),
   UNIQUE (DNAME),
   FOREIGN KEY (MGRSSN) REFERENCES EMP(SSN)  );
```

Creates a new relation(table) that has the columns DNAME, DNUMBER, MGRSSN …

We can also add constraints to the Relation with **CONSTRAINT** and can be: **CHECK**, **FOREIGN KEY**, **UNIQUE**, **DEFAULT**

On foreign keys we can define **RESTRICT (**does not allow the parent/referenced row to be updated or deleted**)**, **CASCADE**, **SET NULL**, **SET DEFAULT** on updates (**ON UPDATE**) and on deletions (**ON DELETE**)

If we want to remove a relation, we can do it with **DROP TABLE table_name [RESTRICT (**default**)| CASCADE]**, where we use CASCADE if the table is referenced in constraints.

If we want to change an existing table, we can do it with **ALTER TABLE table_name**, and then we can add, change or remove columns like:

```
ALTER TABLE EMPLOYEE
ADD JOB VARCHAR(12);
```

```
ALTER TABLE EMPLOYEE
DROP COLUMN JOB;
```

```
ALTER TABLE EMPLOYEE
ALTER COLUMN JOB VARCHAR(25);
```

Or adding and removing constraints.

A big difference between SQL and the formal relational model(algebra) is that SQL can have multiple identical rows(tuples) in the same table. Meaning that a SQL table is a **multi-set** (**bag**) of tuples/rows, **NOT** a set of tuples.

```
SELECT   <attribute list>
FROM     <table list>
WHERE    <condition>
```

Attribute list is which columns will be retrieved by the query. **Table list** is the list of relations required to process the query (they can be joined or cartesian product). **Condition** is the expression that defines which rows will be retrieved and sings used are (>, >=, <, <=, =, <> (not equal), AND, OR, NOT). Also, a missing WHERE clause is the same as WHERE true meaning all rows pass. But as we said we need to watch for duplicate tuples.

We can give **aliases** to relations so that we can **distinguish** them if there are attributes in multiple tables with the same name or shorten the query when we need a relation name. USERS U.

If a given check is repeated many times, we can create **a Domain for a specific data type**.

```
CREATE DOMAIN pozitiven AS NUMBER CHECK (pozitiven>0);

CREATE TABLE snabduvaci (
   s# NUMBER(3) PRIMARY KEY,
   ime_s VARCHAR(50) NOT NULL,
   saldo pozitiven,
   grad VARCHAR(20) DEFAULT 'Skopje',
   CONSTRAINT ime_s_unique UNIQUE(ime_s)
);
```

НАМЕСТО CHECK (saldo > 0)

SQL has some set operations like **UNION** (**UNION ALL** – gets the duplicates too), **MINUS**, **INTERSECT**, but these operations **apply only to union compatible relations** (they have the same attributes and in the same order).

We can **nest queries** where we can write a query and then use an inner query as a result for something to compare, add, change. And **correlated queries** are when the inner query uses an attribute from the outer query. But correlated queries **can always** be expressed as a single query.

These **are** correlated queries

```
SELECT  E.FNAME, E.LNAME
FROM EMPLOYEE AS E
WHERE E.SSN IN (SELECT ESSN
                FROM DEPENDENT
                WHERE ESSN=E.SSN AND
                    E.FNAME=DEPENDENT_NAME)
```

If a attribute is called **without specifying from which table** then we assume it is from the table in the **innermost query**. And only if it isn't found there SQL goes to the **upper query to search for that attribute**.

EXISTS - If we want to check if a relation or a nested query returns a table with rows.
And if we want to check if a variable is in a explicitly defined set we can do with **IN.** e.x. IN (1,2,3) or IN ('one','two','three').

Also we can check if attribute value is missing or is null with **IS NULL.** Where SuperSSN IS NULL.

We can join(natural,regular,left outer, cross) relations in the FROM clause. Renaming attributes/columns may be needed for natural join.

We can use **aggregate(group) functions** on rows/tuples like **COUNT**, **SUM**, **MAX**, **MIN**, **AVG** and the result is **always a single value**. Aggregate functions **ignore NULL values**.

We can group rows in a table, especially when we apply a aggregate function so that we can get the aggregate function for each sub-group of rows/tuples in a relation. So if we do a (group) aggregate we first group the rows by the provided attribute and then on each tuple group the aggregate funciton is applied. We use **GROUP BY attribute_ name** , but **attribute_ name must appear in the SELECT clause**.

The last line is because when we do a select-from-where-group then the order is this.We get the rows with from also joining is done here, then with where we filter then and then with select we define which columns/attributes if the relations will be returned and finally we group the results together in groups where we group them by one or many attributes. And now if we want to filter grouped rows we cannot do it in the where clause because that is executed before the group by and that's why we use **HAVING** – specify the condition to return a grouped row/tuple.

When we like to filter rows by a substring in a variable we can with **LIKE**. Also we can use:

→ '%' (replaces one or many characters);

→ '_' (replaces one character)

Example: Return only people that live in Skopje: Where Address LIKE '%Skopje,MK%'
Example: Return only the people born on 5th on any month in the 50ties: Where BornDate LIKE '05._ _._ _ _5_'

We can use **Arithmetical operators like (+, -, *, /)** that can be applied to numeric attributes in SQL in the relation.

```
SELECT FNAME, LNAME, 1.1*SALARY
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE SSN=ESSN AND PNO=PNUMBER
      AND PNAME='ProductX'
```

We can sort the rows at the end before returning with **ORDER BY attribute_name [DESC/ASC(**default**)]**.

When we have a query where we have multiple relations involved and we have mulitple WHERE conditions, then we need to distinguish which condition is for **joining** and which for **filtering**. Now in this query DNAME='Research' is for filtering and DNUMBER=DNO is for joining.

```
SELECT FNAME, LNAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME='Research' AND DNUMBER=DNO
```

This is the summary of SQL queries where the first 2 rows are mandatory and the rest are optional. Also note that the attributes provided in GROUP BY also need to be included in the SELECT clause because how will we know how they are grouped. But we **DON'T** need to have the attribute in the ORDER BY also included in the select clause because ORDER BY can use any attribute of the tuple without us needing to include it in the SELECT clause.

```
SELECT      <attribute list>
FROM        <table list>
[WHERE      <condition>]
[GROUP BY   <grouping attribute(s)>]
[HAVING     <group condition>]
[ORDER BY   <attribute list>]
```

DML (Data Manipulation Language): We have three commands for updating / modifying values in a relation/table and those are : **INSERT**, **DELETE, UPDATE**.

We can add row(s) to a relation/table with **INSERT INTO table_name VALUES [(attributes)|((attributes),()…)** – block style**]** , also we can insert rows that are a result from a query. But when we insert the attribute values they must be in the same order as the definition of the table.

We can remove rows from a table like this **DELETE FROM table_name WHERE condition(s)** . This removes the rows from a table that satisfy a condition.But the WHERE clause is optional and without it every row is deleted.

We can update the current rows of a table with **UPDATE table_name SET attribute = new_value WHERE condition**. This updates the attributes with their new values on those rows that satisfy the condition.

```
UPDATE EMPLOYEE
SET SALARY = SALARY *1.1
WHERE DNO IN (SELECT DNUMBER
              FROM DEPARTMENT
              WHERE DNAME='Research')
```

We can also add general constraints (these are different from those that we used in DDL), with declarative assertions, using **CREATE ASSERTION assertion_name CHECK condition** . The condition can be anything from a query to a simple check.

Example: The salary of the employee must not be higher than the salary of the manager of the department that employee works for. Now this returns boolean so that if there is no employee like with that condition then it **will return false and the assertion is not violated**.

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS
       (SELECT *
        FROM EMPLOYEE E, EMPLOYEE M,DEPARTMENT D
        WHERE E.SALARY > M.SALARY AND
              E.DNO = D.NUMBER AND D.MGRSSN = M.SSN))
```

When we want to monitor the database and take some action when a conditon/event occurs we can do that with SQL Triggers. They are assertions that define actions that are automatically executed when a condition occurs. We write as: **CREATE TRIGGER trigger_name (AFTER|BEFORE) event ON table_name [FOR EACH ROW] [WHEN condition] actions**;
And event possibilites are : INSERT|DELETE|UPDATE [OF attribute {, attribute}]

Example: If we want to compare the salary of employee to that of his supervisor during update or delete.

```
CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
       SALARY, SUPERVISOR_SSN ON EMPLOYEE
FOR EACH ROW
WHEN
(NEW.SALARY > (SELECT SALARY
               FROM EMPLOYEE
               WHERE SSN=NEW.SUPERVISOR_SSN)
)
INFORM_SUPERVISOR (NEW.SUPERVISOR_SSN, NEW.SSN);
```

This is some stored procedure

When we want to save the result of a query operation but to change when the relations in that query are changed, then we can use a View (also called 'virtual table'). This is not a table but it's more like a stored reuse of a query that we can later modify as a table.
We write as **CREATE VIEW view_name [(attribute** {, attribute} **)] AS select_statement/query.**
But we can write queries with them just like we can with normal tables.
And we can remove a view when we don't need it **DROP view_name**.

Example: Specify a different WORKS_ON table that will swap the employee SSN with their name and surname.

```
CREATE VIEW WORKS_ON_NEW AS
SELECT FNAME, LNAME, PNAME, HOURS
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN=ESSN AND PNO=PNUMBER;
```

Now we can understand that views are not actually persistent storage but only the logic of a query. Now this becomes a problem if that view has complex query logic and we use that view very often and in short time frame and the database system has to execute that expensive logic every time. That is both expensive and inefficient. We should instead use some form of caching the result ,especially if there is not change of the source relations of that result.
We can in cache that view result with **view materialization.** This means we create and keep a temporary table or cte, but problem is when the base tables are updated, so we can manage that with incremental updates.
We can update the view if:

- View doesn't have aggregate funcitons(base tables will be updated),
- View has joins(we can update the base join tables, but not always possible, we must add CHECK in the view to check for updateability and plan execution strategy)
- Views using groups and aggregate functions (**not updateable**)

Database Programming is needed when we want to access a database from a application (opposed to a interactive interface like Datagrip). We need this because most database operations are made thru applications (web applications).

Impedance missmatch is when the application program language is incompatible with the database model. Like type missmatch (requiring new binding for each language), processing the whole set of rows **instead** of one row at at time.

Database programming aproaches are:

» **Embedded commands** – DB commands are embedded in the programming language
» **Library of DB functions** – available for the host language for DB calls. known as an API(Application Program Interface)
» **A brand new language** – minimizes impendance missmatch.

Steps in DB programming:

1. Client program **opens a connection** to the DB server
2. Client program **submits queries and/or updates** the DB
3. When DB access in no longer needed, client program **closes the connection**.

Connection (multiple connections are possible but only one is active)

```
CONNECT TO server-name AS connection-name
AUTHORIZATION user-account-info;
```

Change from an active connection to another one

```
SET CONNECTION connection-name;
```

Disconnection

```
DISCONNECT connection-name;
```

Embedded SQL – most SQL statements can be embedded in host programming languages like C,Java… . And the embedded SQL is enclosed with **EXEC SQL** or **EXEC SQL BEGIN** and to end **END-EXEC** or **EXEC SQL END(;)** . But this may vary.
The **shared variables**(used in both languages) usually have the prefix ':' .Example: (:Fname , :SSN).

Embedded SQL in C – a **cursor**(iterator) is needed to process multiple tuples. **FETCH** command moves the cursor to the next tuple and **CLOSE CURSOR** indicated that the processing of query results is completed. Variables inside **DECLARE** are shared and can appear in SQL statements. **SQLCODE** is used for (errors/exceptions) communication between the program and DB.

SQLJ – a standard for embedding SQL in Java.
JDBC(Java DataBase Connectivity) – executes converted SQL statements in Java. Any Java program with JDBC functions can access any relational DBMS. Also JDBC allows for multiple DB connections(data sources). Classes from java.sql are needed.

Dynamic SQL – for creating and executing new SQL statements at run-time. Program accepts SQL statements from keyboard at runtime. A point-and-click operation translates to the certaion SQL query.
Dynamic update is simple, bu dynamic query can be complex because type and number of retrieved attributes is not known at compile time.

DB Stored Procedures – persistent procedures/functions that are stored locally and executed by the DB server. Good if the procedure is needed by many apps, reduced comms cost between BD and Client, improved power of modeling views. But a problem because every DBMS has unique syntax and can make the system less portable.

Physical Organisation of a Relational DB.
Relational DB file consists of pages/blocks on disk. Every **block** can have one or more records. Operations are store, change, delete record. We can access a record with a Record id = (Page, Location in Page). File types are sorted,unsorted, indexed.
Unordered files(heap) – storing a record is very efficient. Searching is linear { O(n) } .
Ordered files(sequential) – Records are stored in order of their order key/id. Storing is expensive. Finding a record will be the fastest with a binary search.

We will now analyise the time needed for operations in files. **B** - num of blocks in a file , **D** - num or records in a block.
When we want to access every record then the ordering of the files doesn't mean anything because we have to go thru every file eventualy. So time needed to go thru every record is the same for both ordered and unordered files.
When we want to find a record by key when unordered the avg access time is **(0.5*B)*D** , and for ordered is $\mathbf{log_2}\ \boldsymbol{B} * \boldsymbol{D}$.
When we want to find a record with a keys of a range(e.x. keys of range 5 to 8) then the access time for unordered file is **B*D**, and for ordered file is $((\mathbf{log_2}\ \boldsymbol{B}) + \boldsymbol{pages}) * \boldsymbol{D}$.
When we want to insert a record then in an unordered file the access time is **2*D** (this is the two IO opperations), and for ordered files it is $((\mathbf{log_2}\ \boldsymbol{B}) + \mathbf{B}) * \boldsymbol{D}$.
When we want to delete a record on an unordered file is **(0.5*B+1**(for write) **)*D**, for ordered file is $((\mathbf{log_2}\ \boldsymbol{B}) + \mathbf{B}) * \boldsymbol{D}$.

Now what will be the cost needed for clustered indexes. When we go thru all records the time will be **3/2 *B*D** because the file is filled 2/3 so we need 3/2 more accesses to get all of the records. When we search by key then the time wiil be $(\mathbf{log_f}\left(\frac{\mathbf{BR}}{\mathbf{E}}\right) + \mathbf{2}) * \mathbf{D}$, BR-total records num, E- num of pairs, F – fan-out factor, also +2 because (1 for root and 1 for record read). When we want to find by range of key then it will be +3 because (1 for root + round(3/2) because of 2/3 filled) - $(\mathbf{log_f}\left(\frac{\mathbf{BR}}{\mathbf{E}}\right) + \mathbf{3} * \mathbf{pages}) * \mathbf{D}$, for insert/delete a record then $(\mathbf{log_f}\left(\frac{\mathbf{BR}}{\mathbf{E}}\right) + \mathbf{4}) * \mathbf{D}$, we add 2 more than searching because we update the leaf and the file.

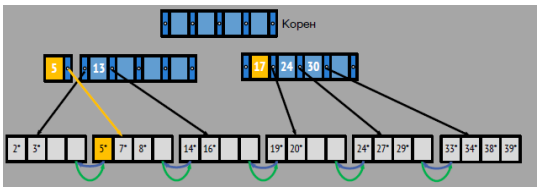
	Неподредена датотека	Подредена датотека	Кластерирачки индекс
Изминување на сите записи	B*D	B*D	3/2 * B * D
Пребарување по клуч	0.5*B*D	$(\log_2 B)*D$	$(\log_r(BR/E)+2)*D$
Пребарување на записи со клуч од даден опсег	B*D	$((\log_2 B)+pages))*D$	$(\log_r(BR/E)+3*pages)*D$
Внес на нов запис	2*D	$((\log_2 B) + B)*D$	$(\log_r(BR/E)+4)*D$
Бришење на запис	(0.5*B+1)*D	$((\log_2 B) + B)*D$	$(\log_r(BR/E)+4)*D$

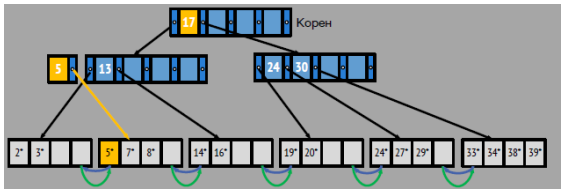
Index is a data structure that makes access and search for a record faster. **Search** is the first part of the opperations find, insert, delete, change. **Search key** is the group attributes from the relations that we search by. Index provides seach by search key to a pointer to the block containing the record. Index can be created with a group of attributes (**composite index**), and the order of the keys is **lexicographic**.We sort by attribute,then by second … But from the formal definition we need the first m attributes to have an equality operator and maximum one (m+1) attribute with {<,>}. So this means we can have attributes with equality but must be before the one with the {<,>} signs or have only one with {<,>} signs.This is because the index structure cannot effectively follow a range for multiple attributes, only for one.
The realisations are: **B+ Tree**, Hash, R-Tree …

B+ Tree stores pointers to records of a DB, only in the leaf nodes. Every **inner node** occupies 1 block/page and contains **ordered** indexed records , **pairs <key,pointer>** .
B+ Tree are dynamic structures, always balanced (during insert/delete this is maintained), operation size is $\log_f N$, efficient insert/delete.
And every inner node has a **d – max fan-out parameter**, meaning each inner node must have min (d) keys and max (2*d) keys and if that goes higher then we split and branch into a new node. **d<=key count <=2d**.

Max fan-out is a branching factor, max fan-out = 2*d+1 , and if the key count value reaches this then we branch. So this means that a big fan-out factor will result in a tree with small depth.

Example question: How many records does a B+ Tree have with height 1 and 3?
-If height is 1 and d=2 , so max fan-out is d*2+1 -> 2*2+1=5, and the terminal node count is 5, and each can have 4 records, so the total record number is 5(nodes) * 4(records per node) = 20.
-If height is 3 and d=2, max fan-out is d*2+1 -> 2*2+1=5, and the number of nodes is $5^3 = 125$, $125 * 4 = 500$





We can see that on the left picture the key is **copied**, but on the right in **is moved up**.

When we insert a record we first find the leaf node that we will store into, then if there is space we store it, lastly we sort the keys in that leaf node. Now what if we find the leaf node but there is no space to insert into. We create a new leaf node and split all records from the filled leaf to the new so they will be balanced. Then update the pointers to next/prev on the neighbouring leaf nodes. But now this new leaf will not have a parent node. So we create a new parent by getting the first key(lowest) of the new node, and make the parent point to this new leaf, Now we check if the the node on the level of the parents is full, if it is then also do a split there. But there is a **difference when a leaf node is split and a parent/inner node**. When the leaf node is split then the **parent key is copied from the (lowest key) of the new leaf node**. But in the inner node split, The newly created node will take the lowest key and move it up to it's parent (**push up**).

There are three possibilites**(Alternatives)** of presenting the DB data in an index:

1. ***Records are stored in the index*** , so we **don't need additional file for the DB data**.
2. ***Record references are stored In the index***, so the leaf nodes we have pair <key, Record id>
3. ***List of references are stored in the index***, dense compact storing where we have multiple records with the same key values that we search by, leaf nodes have a pair <key, {list of Record id}>

By-reference indexes (possibilites 2 and 3) can be **Clustered** and **Non-Clustered**.
Clustered index – data in the file is stored ordered by the search key of the index. Here the file is sorted and we leave gaps in every block for future insertions. **Pros** are: efficient for search with keys in a range, data is stored in neighbouring sequential blocks, allows some compression types. **Cons** are: expensive to maintain, periodical updates, fille **is filled 2/3 of it's capacity** to allow future insertions.

So attributes that are used in queries as selection are good canidates for index keys. Composite key is good when we search by multiple attributes. We should select an index that covers multiple search possibilites. We need to be careful when selecting index because only one can be clustered.

	Неподредена датотека	Подредена датотека	Кластерирачки индекс
Scan all records	O(B)	O(B)	O(B)
Equality Search	O(B)	$O(\log_2 B)$	$O(\log_2 B)$
Range Search	O(B)	$O(\log_2 B)$	$O(\log_2 B)$
Insert	O(1)	O(B)	$O(\log_2 B)$
Delete	O(B)	O(B)	$O(\log_2 B)$

Difference between clustered and non-clustered indexes is that for Non-clustered index a **separate IO read operation to access each record**, while in a clustered we need **1 IO for the block only**.

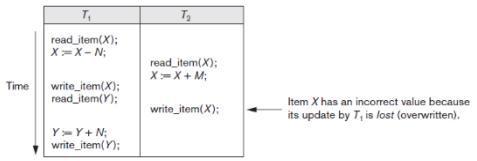
Transaction – logical unit of database processing that includes one or more access operations(read, write, update, delete). Transaction/s can be specified in SQL submitted interactively, or maybe embedded within a program.

Transaciton boundaries – **begin transaction** and **end transaction**, and a application can have several transactions separated by begin and end transaction boundaries.
Granularity of data – can be a field,record or a whole block of a disk.

Database is a collection of named data items. Basic operations are **read** and **write**.
read_item(X) has these steps: find the address of the disk block that has item X, copy that block into a buffer in main memory, copy item X from buffer to program variable X.
write_item(X) has these steps: find the address of the disk block that has item X, copy that block into buffer in main memory, copy item X from the program into the location of item X in that block in the buffer, store the updated block from buffer back to disk(updates the database file on disk).

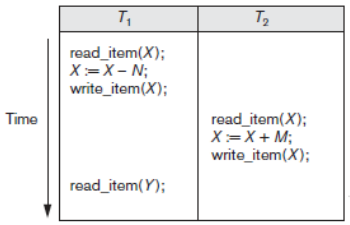
Concurrency control is a critical aspect of database management systems to ensure data integrity and consistency when multiple transactions are executed simultaneously. Let's explore four common concurrency problems in detail:

Lost update problem – when two transactions that access the same database items and their operations are intertwined in a way so that the value becomes incorrect.



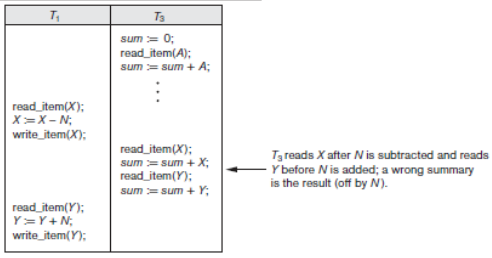
Temporary update(Dirty read) – when a transaction reads data that has been modified by another transaction that has not yet committed.

Incorrect Summary Problem – when one transaction is calculating the summary function on records while other transactions update some of those records, then the aggregate function may calculate some values that are updated.



Transaction T1 fails and must change the value of X back to its old value, while T2 has read the temp incorrect value of X

Unrepeatable read problem – transaction reads the same item twice and then the item is changed by another transaction between the reads, and that causes the first transaction to get different values for both reads.

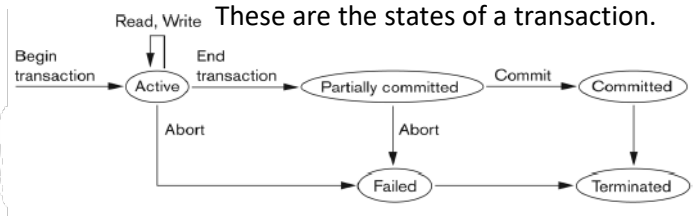


Transactions can fail because of: computer failure, transaction/system error, local/exception errors, concurrency control enforcement, disk failure, physical problems or catastrphes. DBMS should store information to recover from these errors (most commonly the first four).

Transaction is an atomic unit of work that is either completed or not done at all. The recovery manager tracks opperations:

- **begin_transaction**,
- **read** or **write**,
- **end_transaction** (here we also check if concurrency violation occurred and the transaction has to be aborted).
- **commit_transaction** (signals Success and changes will not be undone).
- **rollback**(**abort**) (signals that transaction is unsuccessful and changes must be undone).

Log or **Journal** keeps track of all transaction operations that affect values of database items. This information may be needed to recover from failure. The log is kept on disk and it is not affected by any failure except disk fail or catastrophe. Also, the log is backed up in archive to guard against a catastrophe. Types of log records are:



- **[Start_transaction, T]** – records that transaction T has started execution
- **[Write_item, T, X, old_value, new_value]** – records transaction T has change value of X from old_value to new_value
- **[Read_item, T, X]** – records that transaction T has read the value of X
- **[Commit, T]** – records that transaction T has completed Successfully and confirms that it can be committed to DB
- **[Abort, T]** – records that transaction T has aborted.

Desirable properties of transactions are: **Atomicity**, **Consistency preservation**, **isolation** (updates during transaction should not be visible to others until is committed), **Durability or permanency** (changes after a transaction must not be lost because of a failure).

Schedule (**history**) of transactions – is the ordering of operations of transactions in way so that the operations and its order in the transaction must also appear the same way in the Schedule.

Two operations in a schedule are in conflict when they satisfy these: they belong to **different transactions**, they access the **same** item X, **at least one** of the operations is write_item(X).

Schedules requiring cascaded rollback – schedule where uncommitted transactions that read an item from a failed transaction must be rolled back. This can lead to a significant waste of CPU time and resources.
Cascadeless schedule – where transaction reads only the items that are written by committed transactions. This prevents the need for cascading rollbacks, as no transaction depends on uncommitted data. Cascadeless schedules are always recoverable and reduce the overhead of recovering from failures
Strict schedule – transaction can neither read or write an item X until the last transaction that wrote X has committed.

Schedules based on Serializability
Serial schedule – all operations of each transaction are executed consecutively without interleaving operations from other transactions. While serial schedules guarantee correctness, they limit concurrency and can lead to poor performance.

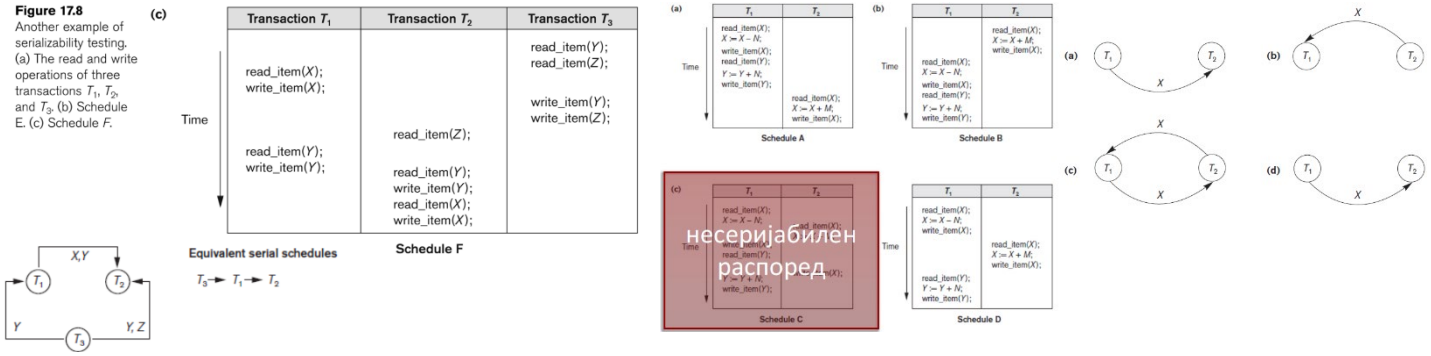
Serializable schedule – when it is equivalent to some serial schedule of the same transactions.
Result equivalent - when two schedules produce the same final state of the DB.
Conflict equivalent – if the conflicting operations are the same in both schedules.
Conflict serializable – when a schedule is conflict equivalent to another serial schedule. Conflict serializability ensures data consistency and correctness when multiple transactions are executed concurrently.
Under **semantic constraints**, schedule that are not conflict serializable may work correctly.

View equivalence and view serializability – less restrictive equivalence of schedules, when two schedules have: the same set of transactions and same operations for them, item X is read from one and written to in another transaction in both schedules, the write operation for the same item must be the last one in both schedules.

But serializable schedule is **not the same** as serial schedule. Under serializable we understand that the schedule is **correct**. Meaning that will leave the DB consistent after it, and intertwineability is allowed and will result in the same state as if the transactions were serialy executed, but this way it will be more efficient because they we execute concurrently. But serializability is not easily checked.

Practical approach on enforcing Serializability is by making methods (protocols) to ensure serializability. But it's not possible to determine when a schedule begins and ends, so we reduce the problem by only checking a **committed part** of the schedule. A current approach used in most DBMS is using locks with two phase locking.

Testing for serializability by conflict – look at only read and write opperations, construct a priority graph where edges are the transactions we analyse, **schedule is serializable only if the constructed graph has no cycles**. Directed graph where nodes are Transactions, directed edges like Ti -> Tj, means Ti must execute before Tj to maintain conficting opperation order. If we have a cycle then schedule is **NOT conflict-serializable**, else it **IS conflict-serializable.**



Single SQL command is always considered to be **atomic**. In SQL there is no explicit Begin Transaction.
Every transaction must have an explicit end statement and must be **COMMIT** or **ROLLBACK**.
We can specify the properties of a current transaction with SET TRANSACTION. Propertires are:

- Access Mode – READ ONLY or READ WRITE (default). But if isolation level is READ UNCOMMITED then READ ONLY is the default one. This restricts which commands can be executed in a transaction.
- Diagnostic area size – specifies a integer value that tells the number of informations(errors, exceptions) that can be held in the same time.
- Isolation level – how much of the data accessed in a transaction is isolated from other transactions. Isolation levels:
  - **READ UNCOMMITTED** – lowest isolation, transactions can read uncommited changes, but this way dirty reads are allowed, best used when performance is priority over accuracy, like reporting data
  - **READ COMMITTED** – better isolation, transaction can read only committed data, this will prevent dirty reads, but Non-repeatable reads and phantom reads are **possible**. Best when balance between consistency and concurrency.
  - **REPEATABLE READ** – even better isolation, transaction will get the same data after many sequential reads, even after the data is changed mid way. This will prevent Non-repeatable reads, but phanton reads are possible. Best used when consistent data across multiple reads is critical.
  - **SERIALIZABLE** (default) – highest isolation, transactions are executed in serial and no concurrency issues are possible. Best when absolute accuracy and consistency required, like in banking transactions.

Phantom read – when one transaction selects some rows with the WHERE clause but in the same time another transaction adds a row that also satisfies that condition and when the first transaction is repeated then it will see a *phantom row* (row that did not exist before).