Introduction to Databases

Lecture 4 – Advanced relational database concepts

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Master DSBA 2020 – 2021

CentraleSupélec

What you will learn

In this lecture you will learn:

- The notion of **views** and **materialized views**.
- General principles of **database indexes**.
- **Transactions** and their role in database **consistency** and **failure recovery**.

Virtual views

Definition (Virtual view)

A **virtual view** is a relational table that does not exist physically and is only defined by a query.

Creating a view

```
CREATE VIEW <view_name>
AS <view_definition>
```

The view definition is a **SQL query**.

☞ The DBMS only stores the SQL query that defines a view, not the result of the query.

Running example

In the following slides, we consider a database with the following tables:

Table Employee

```
CREATE TABLE Employee(
  emp_id INTEGER PRIMARY KEY,
  first_name TEXT,
  last_name TEXT,
  position TEXT NOT NULL,
  salary FLOAT DEFAULT 30000,
  dept_id INTEGER
)
```

Table Department

```
CREATE TABLE Department(
  dept_id INTEGER PRIMARY KEY,
  dept_name TEXT,
  budget FLOAT
)
```

Creating a virtual view

Create a view from a single table

```
CREATE VIEW TopEmployee AS
  SELECT first_name, last_name, salary, dept_id
  FROM Employee
  WHERE position IN ("Executive director", "Assistant director")
```

Create a view from multiple tables

```
CREATE VIEW AdministrationEmployee AS
  SELECT first_name, last_name, salary
  FROM Employee e JOIN Department d ON e.dept_id = d.dept_id
  WHERE d.dept_name="Administration"
```

Create a view from a view

```
CREATE VIEW TopSalary AS
  SELECT MAX(salary)
  FROM TopEmployee
```

Querying a view

Querying a view

```
SELECT AVG(salary)
FROM TopEmployee
WHERE position = "Executive director"
```

Querying a view with other tables

```
SELECT first_name, last_name, budget
FROM TopEmployee t JOIN Department d ON t.dept_id=d.dept_id
WHERE position = "Executive director"
```

☞ When we query a view, the view is **computed dynamically**.

Virtual views: motivations

1. Simplify the writing of complex SQL queries.

Without using views

Get the name of the department of the employee with the highest salary.

```
SELECT d.dept_name
FROM department d JOIN
    (SELECT emp_id, dept_id, salary
     FROM Employee
     ORDER BY salary desc
     LIMIT 1) max_salary ON d.dept_id=max_salary.dept_id
```

Virtual views: motivations

1. Simplify the writing of complex SQL queries.

Using views

Get the name of the department of the employee with the highest salary.

```
CREATE VIEW max_salary as
  (SELECT emp_id, dept_id, salary
   FROM Employee
   ORDER BY salary desc
   LIMIT 1)

SELECT d.dept_name
FROM max_salary m JOIN Department d
       ON m.dept_id=d.dept_id
```

Virtual views: motivations

2. Restrict the access to sensitive data.
   - Create a view with only a **subset of data**.
   - Only grant access to the view, not the full table.

Restrict the access to sensitive data

Create a view where the salary is not visible.

```
CREATE VIEW Employee_partial AS
  SELECT emp_id, first_name, last_name, position, dept_id
  FROM Employee
```

Writing to a view

- A **write operation** (*insert*, *update*, *delete*) on a view is translated into a write operation on the original table/view, **if possible**.

Updatability conditions

A view is **not updatable** (cannot *insert*, *update*, *delete*) if it contains any of the following:

- Aggregate functions.
- DISTINCT, GROUP BY, HAVING, LIMIT, UNION, UNION ALL.
- Subqueries in the SELECT clause.
- Subqueries in the WHERE clause referring to a table in FROM.
- Reference to a nonupdatable view in the FROM clause.
- Inner join operations (with conditions).

Writing to a view

- A **write operation** (*insert*, *update*, *delete*) on a view is translated into a write operation on the original table/view, **if possible**.

Insertability conditions

A view is **insertable** (can *insert*) if **it is updatable** and:

- The view contains all columns in the base table that **do not have a default value**.
- The view columns must be **simple column references** (no expressions).

Writing to a view

Subquery in WHERE referencing a table in FROM

Get the employees working in the same department as the employee number 1.

```
CREATE VIEW EmpOneColleague as
SELECT emp_id, first_name, last_name, salary
  FROM Employee
  WHERE dept_id IN
  (select dept_id  from Employee  where emp_id=1) ⟵ subquery
```

- The subquery contains a reference to table `Employee`.
- The table is referenced in the FROM clause.
- The view is **not updatable** (must check a condition on a table being updated).

Writing to a view

Subquery in WHERE not referencing a table in FROM

Get the employees working in the department *Administration*

```
CREATE VIEW AdminEmployee as
  SELECT emp_id, first_name, last_name, salary FROM Employee
  WHERE dept_id IN
  (select dept_id  from Department  where dept_name="Administration")
```

- The subquery doesn't reference any table in the FROM clause.
- The view is **updatable**.

Update example

```
UPDATE AdminEmployee SET salary=40000
```

This is translated into the following query:

```
UPDATE Employee SET salary=40000 WHERE dept_id in
(SELECT dept_id FROM Department where dept_name="Administration")
```

Writing to a view

Subquery in SELECT

```
CREATE VIEW nbEmployeesPerDepartment
  SELECT dept_name,
    (SELECT count(*)
     FROM Employee e
     WHERE e.dept_id=d.dept_id) as nbEmployees
  FROM Department d
```

- This view is **not updatable**.
- It contains a subquery in the clause SELECT.
- The DBMS would not know how to update the column *nbEmployees*.

Writing to a view

Updating a view defined with INNER JOIN

```
CREATE VIEW AdminEmployee as
  SELECT emp_id, first_name, last_name, salary, budget
  FROM Employee e JOIN Department d ON e.dept_id=d.dept_id
  WHERE dept_name="Administration";

UPDATE AdminEmployee SET budget=30000
```

- The update is allowed if the columns of **only one table** are modified.
- The update is **allowed**.
  - Only the column *budget* is updated.

Update

The update is translated into the following:

```
UPDATE Department SET budget = 30000
WHERE dept_name="Administration"
```

Writing to a view

Updating a view defined with INNER JOIN

```
CREATE VIEW AdminEmployee as
  SELECT emp_id, first_name, last_name, salary, budget
  FROM Employee e JOIN Department d ON e.dept_id=d.dept_id
  WHERE dept_name="Administration";

UPDATE AdminEmployee SET budget=30000, salary=40000
```

- The update is **not allowed**.
- We try to modify one column from table `Employee`.
- We try to modify one column from table `Department`.

Writing to a view

Updatable but not insertable view

```
CREATE VIEW TopEmployee AS
  SELECT first_name, last_name, salary, dept_id
  FROM Employee
  WHERE position IN ("Executive director", "Assistant director")
```

- The view `TopEmployee` is **updatable**.
- The view `TopEmployee` is **not insertable**.

```
INSERT INTO TopEmployee VALUES ("John", "Smith", 30000, 14)
```

is translated into:

```
INSERT INTO Employee(first_name, last_name, salary, dept_id)
VALUES ("John", "Smith", 30000, 14)
```

No value is specified for *emp_id* nor *position*: but they cannot be NULL!

Writing to a view

Updatable and insertable view

```
CREATE VIEW TopEmployee AS
  SELECT emp_id, position, dept_id
  FROM Employee
  WHERE position IN ("Executive director", "Assistant director")
```

- The view `TopEmployee` is **updatable** and **insertable**.

```
INSERT INTO TopEmployee VALUES (10, "secretary", 50)
```

is translated into:

```
INSERT INTO Employee(emp_id, position, dept_id)
VALUES (10, "secretary", 50)
```

This will add the following row into table `Employee` (note the **default value** for salary).

```
  (emp_id=10, first_name=NULL, last_name=NULL, position="secretary",
   salary=30000, dept_id=50)
```

Materialized views

Definition (Materialized view)

A **materialized view** is a view whose result is *persisted* as if it was a normal database table.

Creating a view

```
CREATE MATERIALIZED VIEW <view_name>
AS <view_definition>
```

- Useful when the **result** of a query is **referenced frequently**.
- When the underlying table(s) is (are) **updated**, the view needs to be updated too. **When?**

☞ Some DBMS (e.g., MySQL) don't support materialized views.

Materialized views

Materialized view example

```
CREATE MATERIALIZED VIEW EmpDept AS
  SELECT first_name, last_name, dept_name
  FROM Employee e JOIN Department d ON e.dept_id=d.dept_id
```

- Any update on columns that are not referenced in the view do not alter the view.
- In the following example, we update the salary of an employee.
- But *salary* is not an example of the view `EmpDept`.
- Therefore, the view is not modified.

Example

```
UPDATE Employee SET salary=54000 WHERE emp_id=10
```

Materialized views

Materialized view example

```
CREATE MATERIALIZED VIEW EmpDept AS
  SELECT first_name, last_name, dept_name
  FROM Employee e JOIN Department d ON e.dept_id=d.dept_id
```

- We consider the following INSERT operation.

```
INSERT INTO Employee VALUES(120, "William", "Tyler",
                            "accountant", 50000, 15)
```

- To update the view, the DBMS executes the following operations.

```
SELECT dept_name FROM Department WHERE dept_id = 15;
```

The department name returned by this query (say, "Finances") is used in the following INSERT operation:

```
INSERT INTO EmpDept VALUES ("William", "Tyler", "Finances");
```

Materialized views

- It would be **costly** to update a materialized view each time the underlying tables are updated.
- A better solution is to **refresh** the materialized view **periodically**.
  - **Example.** Each night, when the database activity is low.
- Querying the materialized view between two refresh operations might return **stale** data.
- This might be acceptable in numerous situations.
  - **Example.** Analysis on product sales.

Searching in tables

How to efficiently search for data in a table?

```
SELECT *
FROM Employee
WHERE position="Secretary" AND dept_id=25
```

Search all tuples

Employee

emp_id	first_name	last_name	position	salary	dept_id
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Secretary	120,000	25

If the table has n rows, the search cost is $O(n)$.

Joining two tables

Employee					
emp_id	**first_name**	**last_name**	**position**	**salary**	**dept_id**
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Secretary	120,000	25

Department		
dept_id	**dept_name**	**budget**
62	Finance	600,000
25	Education	150,000
14	Administration	300,000
45	Human Resources	150,000

How does the DBMS join two tables?

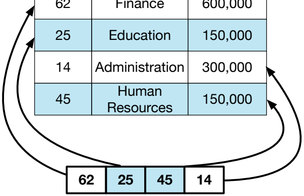
Joining two tables

Employee

emp_id	first_name	last_name	position	salary	dept_id
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Secretary	120,000	25

Department

dept_id	dept_name	budget
62	Finance	600,000
25	Education	150,000
14	Administration	300,000
45	Human Resources	150,000



Hash table

1. **Sequential scan** on `Department`
   - To create a **hash table** for fast access to the table.

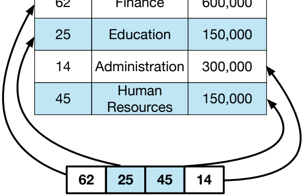
Joining two tables

Employee

emp_id	first_name	last_name	position	salary	dept_id
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Secretary	120,000	25

Department

dept_id	dept_name	budget
62	Finance	600,000
25	Education	150,000
14	Administration	300,000
45	Human Resources	150,000



Hash table

2. **Sequential scan** on `Employee`
   - To match each Employee against his/her department.
   - With the **hash table**, finding an employee's department costs O(1).

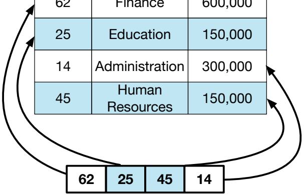
Joining two tables

Employee

emp_id	first_name	last_name	position	salary	dept_id
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counselour	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
1	Joseph	Bennet	Office assistant	55,000	14
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Secretary	120,000	25

Department

dept_id	dept_name	budget
62	Finance	600,000
25	Education	150,000
14	Administration	300,000
45	Human Resources	150,000



Hash table

- Let T_1, $T2$ be two tables with n and m rows respectively.
- The cost of T_1 JOIN T_2 is $O(n + m)$.

Indexes

Definition (Index)

An **index** on a column, or multiple columns (**composite index**), of a table is a *data structure* that makes it efficient to find the rows that have some specific values for those columns. ▸ Source

- An index is **stored** into the database, **independently** of the data.
  - Adding or removing an index does not affect the data, nor the queries.
- An index is a sequence of **records** (a.k.a., **index entries**).
  - Each record is a pair (`search_key`, `pointer`).
  - `search_key`: values of the indexed columns (e.g., *emp_id*).
  - `pointer`: reference to a row in the table where the values in the indexed columns match the search key.

Simple example: linear indexes

- Each row has a **logical identifier** *rowid*.
- The search keys are **sorted** in the index.

Index on emp_id	
1	row6
2	row4
3	row5
4	row1
5	row2
6	row3
7	row7
8	row8

	Employee					
rowid	**emp_id**	**first_name**	**last_name**	**position**	**salary**	**dept_id**
row1	4	Mary	Green	Credit analyst	65,000	62
row2	5	William	Russel	Guidance counselour	35,000	25
row3	6	Elizabeth	Smith	Accountant	45,000	62
row4	2	John	Doe	Budget manager	60,000	62
row5	3	Patricia	Fisher	Secretary	45,000	25
row6	1	Joseph	Bennet	Office assistant	55,000	14
row7	7	Michael	Watson	Team leader	80,000	14
row8	8	Jennifer	Young	Secretary	120,000	25

☞ When rows are added/deleted/updated, the index needs to be **updated** too!

Simple example: linear indexes

Advantages

- **Efficient search**: $O(\log n)$ (binary search).
- **Range queries** are supported.

Disadvantages

- **Update high cost**: $O(n)$.
- Efficient search only when the index can be loaded **entirely** in main memory.
  - Otherwise, the search performance is degraded by disk accesses.
- Depending on the size of the index, this might not be possible.

Simple example: linear indexes

Composite index

- Index defined on more than one column.
- The **order** of the columns in a composite index matters.

Index on {first_name, last_name}		
Bennet	Joseph	row6
Doe	John	row4
Fisher	Patricia	row5
Green	Mary	row1
Russel	William	row2
Smith	Elizabeth	row3
Watson	Michael	row7
Young	Jennifer	row8

	Employee					
rowid	emp_id	first_name	last_name	position	salary	dept_id
row1	4	Mary	Green	Credit analyst	65,000	62
row2	5	William	Russel	Guidance counselour	35,000	25
row3	6	Elizabeth	Smith	Accountant	45,000	62
row4	2	John	Doe	Budget manager	60,000	62
row5	3	Patricia	Fisher	Secretary	45,000	25
row6	1	Joseph	Bennet	Office assistant	55,000	14
row7	7	Michael	Watson	Team leader	80,000	14
row8	8	Jennifer	Young	Secretary	120,000	25

Simple example: linear indexes

Composite index

```
SELECT * FROM Employee
WHERE first_name='Elizabeth' AND last_name='Smith'
```

The index **is used** for this query (accessing both columns).

Index on {first_name, last_name}		
Bennet	Joseph	row6
Doe	John	row4
Fisher	Patricia	row5
Green	Mary	row1
Russel	William	row2
Smith	Elizabeth	row3
Watson	Michael	row7
Young	Jennifer	row8

	Employee					
rowid	emp_id	first_name	last_name	position	salary	dept_id
row1	4	Mary	Green	Credit analyst	65,000	62
row2	5	William	Russel	Guidance counselour	35,000	25
row3	6	Elizabeth	Smith	Accountant	45,000	62
row4	2	John	Doe	Budget manager	60,000	62
row5	3	Patricia	Fisher	Secretary	45,000	25
row6	1	Joseph	Bennet	Office assistant	55,000	14
row7	7	Michael	Watson	Team leader	80,000	14
row8	8	Jennifer	Young	Secretary	120,000	25

Simple example: linear indexes

Composite index

```
SELECT * FROM Employee
WHERE last_name='Smith'
```

The index **is used** for this query (accessing the first column).

Index on {first_name, last_name}		
Bennet	Joseph	row6
Doe	John	row4
Fisher	Patricia	row5
Green	Mary	row1
Russel	William	row2
Smith	Elizabeth	row3
Watson	Michael	row7
Young	Jennifer	row8

Employee

rowid	emp_id	first_name	last_name	position	salary	dept_id
row1	4	Mary	Green	Credit analyst	65,000	62
row2	5	William	Russel	Guidance counselour	35,000	25
row3	6	Elizabeth	Smith	Accountant	45,000	62
row4	2	John	Doe	Budget manager	60,000	62
row5	3	Patricia	Fisher	Secretary	45,000	25
row6	1	Joseph	Bennet	Office assistant	55,000	14
row7	7	Michael	Watson	Team leader	80,000	14
row8	8	Jennifer	Young	Secretary	120,000	25

Simple example: linear indexes

Composite index

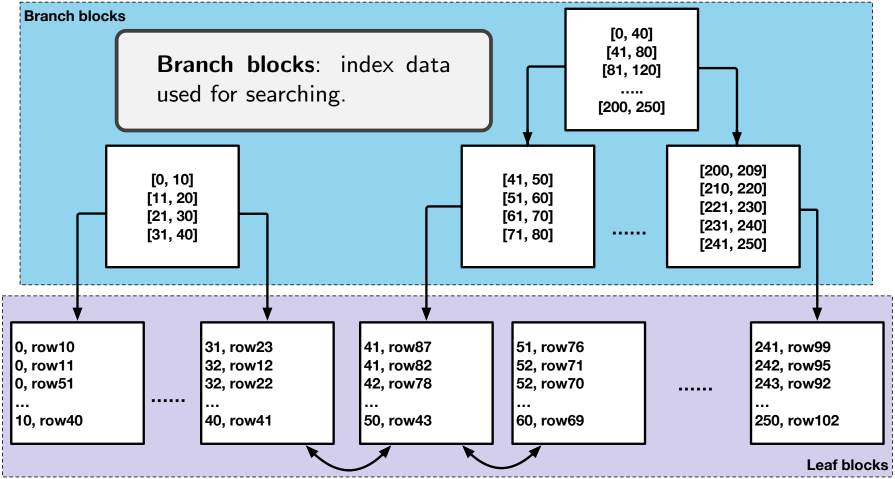
```
SELECT * FROM Employee
WHERE first_name='Elizabeth'
```

The index **is not used** for this query (accessing the second column).

Index on {first_name, last_name}		
Bennet	Joseph	row6
Doe	John	row4
Fisher	Patricia	row5
Green	Mary	row1
Russel	William	row2
Smith	Elizabeth	row3
Watson	Michael	row7
Young	Jennifer	row8

	Employee					
rowid	emp_id	first_name	last_name	position	salary	dept_id
row1	4	Mary	Green	Credit analyst	65,000	62
row2	5	William	Russel	Guidance counselour	35,000	25
row3	6	Elizabeth	Smith	Accountant	45,000	62
row4	2	John	Doe	Budget manager	60,000	62
row5	3	Patricia	Fisher	Secretary	45,000	25
row6	1	Joseph	Bennet	Office assistant	55,000	14
row7	7	Michael	Watson	Team leader	80,000	14
row8	8	Jennifer	Young	Secretary	120,000	25

B-tree indexes



Branch blocks: index data used for searching.

Leaf blocks: store values.

Image source

B-tree indexes

- A B-tree index is **balanced**.
  - All leaves are at the same **height**.
  - Retrieving a record from anywhere in the index takes the **same amount of time**.
- A **leaf block** contains records of pairs (*search key*, *rowid*).
- Records are sorted by (*search key*, *rowid*).
  - Once we locate the search key in the tree, we can immediately retrieve the pointers to all rows that match the search key.
- The leaf blocks are doubly linked.
  - Useful to answer **range queries**.

B-tree indexes

Computational cost

Given a table with n rows:

- The cost of **searching** for a row in the index is $O(\log n)$.
- The cost of **inserting** or **deleting** a new row is $O(\log n)$.

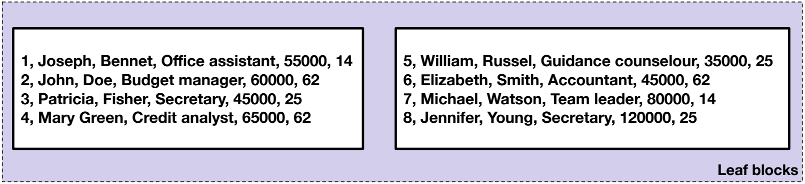
Multi-level index

- Useful when an index does not fit in main memory.
- Some levels of the tree can be stored in memory, the others can be on disk.

Clustered indexes

Definition (Clustered index)

In a **clustered index**, rows are stored within the index itself. The table is therefore sorted around the values of the columns on which the index is defined.



- Reduce the number of read operations to retrieve a row.
- **Only one** clustered index per table is possible.
  - Built on columns that are part of the **primary key**.
- **Secondary indexes** can be created on other columns, if needed.

Indexes: discussion

- Indexes are key to speed up queries.
- But indexes come with a **cost**.
  - **Storage.** The indexes are stored in the database.
  - **Updates**. When rows are inserted, updated or deleted, indexes must be updated.

When indexes should not be used

- **Low-read**, **high write** columns.
- **Low cardinality** columns (with few distinct values).
- **Small tables**.

- **Composite indexes.** Use on columns that **co-occur** frequently.
- **Clustered indexes.** Ideal when queries return most of the columns.

Create indexes in SQL

```
CREATE INDEX my_index ON Employee(last_name)

CREATE UNIQUE INDEX my_index ON Employee(last_name)

CREATE INDEX my_index ON Employee(last_name) USING BTREE
```

Remember

- On the columns of the primary key an index is **automatically created**.
- If a column is declared as UNIQUE, an index is **automatically created**.

Notion of transaction

Key assumptions so far

- **Only one user** reads/writes the database.
- Read/write operations are executed **in their entirety**, or **atomically**.

Read operations

- Read operations do not modify the **state** (i.e., the values) of the database.
- Many users can **safely read concurrently**.
- Read operations can be **safely interrupted**.

Write operations

- What happens if many users **write concurrently**?
- What happens if a write operation is **interrupted**?

Notion of transaction

Atomicity

- John Smith wants to transfer 500 euros from his checking to his savings account.
- What if the database system fails between the two updates?

Account

acct_nbr	client_name	type	balance
4	John Smith	checking	5000
5	John Smith	savings	20000

1

```
UPDATE account
SET balance = balance - 500
WHERE acct_nbr = 4
```

Account

acct_nbr	client_name	type	balance
4	John Smith	checking	4500
5	John Smith	savings	20000

2

```
UPDATE account
SET balance = balance + 500
WHERE acct_nbr = 5
```

Account

acct_nbr	client_name	type	balance
4	John Smith	checking	4500
5	John Smith	savings	20500

Notion of transaction

Serializability

- Two customers are selecting a seat on a flight.
- What if they select the same seat at the same time?

Flight			
flt_nbr	flt_date	seat_nbr	seat_status
AF345	23/12/2020	22A	empty

```
SELECT seat_nbr, seat_status
FROM Flight
WHERE  flt_number='AF345' AND lft_date='23/12/2020'
```

22A	empty

```
SELECT seat_nbr, seat_status
FROM Flight
WHERE  flt_number='AF345' AND lft_date='23/12/2020'
```

22A	empty

```
UPDATE Flight
SET seat_status='occupied'
WHERE  flt_number='AF345' AND lft_date='23/12/2020'
     AND seat_nbr = '22A'
```

22A	occupied

```
UPDATE Flight
SET seat_status='occupied'
WHERE  flt_number='AF345' AND lft_date='23/12/2020'
     AND seat_nbr = '22A'
```

Notion of transaction

Definition (Transaction)

A **transaction** is a sequence of read and/or write operations on a database that are executed as a **single atomic operation**. Either all are executed or none. Importantly, the values are stored only if the transaction is successful.

Transaction in SQL

```
START TRANSACTION;
UPDATE Account SET balance=balance-500 WHERE account_nbr=4;
UPDATE Account SET balance=balance+500 WHERE account_nbr=5;
COMMIT;
```

Transactions: COMMIT and ROLLBACK

Transaction successful: COMMIT

- All changes to the database caused by the transaction are **committed** (i.e., persisted).
- Before COMMIT, changes are tentative, they may or may not be stored on disk.

Transaction unsuccessful: ROLLBACK

- All changes to the database caused by the transaction are **rolled back** (i.e., undone).

Transactions: serializability

Definition (Serializable transactions)

Two or more transactions are **serializable** if they behave as if they were run *serially*, one at a time.

- By default, transactions are **serializable**.
  - This means that transactions are completely **isolated**.
- Transactions that operate on different data are easily serializable.
- Transactions that operate on the same data can be serialized by using a **locking** mechanism.

Locks

- When a transaction operates on a table, the DBMS may impose a **lock** on (parts of) that table.
- Other transactions are unable to read/write a locked table.

Transactions: dirty reads

- **Dirty data** is data written by a transaction that is not yet committed.
- Serializable transactions **cannot read** dirty data.
- In some cases, transactions can be allowed to have **dirty reads**.
  - When dirty reads do not lead to serious problems.
  - Avoids the time-consuming work by the DBMS to prevent dirty reads.

Example

- An available seat is chosen for the customer by the system and set to *occupied*.
- If the customer rejects the proposition, the seat is set to *available*.
- If another transaction reads the seat status before the customer rejection, it sees the seat as occupied.
- But there is no risk of giving the same seat to two different customers.

Transactions: recovery

Definition (Transaction manager)

The **transaction manager** is the component of a database system that makes sure that transactions are executed correctly.

Transaction manager role

- Sends commands to the *log manager* to log information about the transaction execution.
- Assures that concurrently executing transactions do not interfere with each other.

☞ Logs are used to **recover** from a **failure**.

Transactions: recovery

Undo logging

A **log file** has different types of **records**.

- START T: marks the beginning of transaction T.
- COMMIT T: Marks the successful end of transaction T.
- ABORT T: Marks the unsuccessful end of transaction T.
- UPDATE: (T, X, v), meaning that T has changed a database element X and its former value is v.

1. If T **modifies** X, the log record (T, X, v) is written to disk *before* the new value of X is written to disk.
2. If a transaction **commits**, its COMMIT log record is written *after* all database elements changed by T are written to disk.

Transactions: recovery

- When a **system failure** occurs, the **recovery manager** reads the log file from the end.
  - From the most recently written record.
- If it finds a record (T, X, v):
  - If T is committed, the record is ignored.
  - If T is not committed, the value v is restored to X.
- For all uncommitted transactions, the recovery manager writes an ABORT record to the log.
- The log is stored to the disk and normal operation can resume.

☞ Other recovery mechanisms exist (e.g., **redo logging**).

Transactions: ACID

Transactions have the following properties (ACID):

- **Atomicity (A).** “All or nothing”.
- **Consistency (C).** From a consistent state to a consistent state.
  - some operations within the transaction may lead to inconsistencies.
- **Isolation (I).** Serializability of transactions.
- **Durability (D).** Upon commit, all the updates are permanent.

- A relational database enforces **strict consistency** with transactions.
- This might hamper performances in a **distributed database**.

☞ Strict consistency is one of the requirements that NoSQL databases want to ease.

References

- Garcia-Molina, Hector. *Database systems: the complete book*. Pearson Education India, 2008. ▸ Click here