Chapter 6
The database Language SQL

Spring 2011

Instructor: Hassan Khosravi

- SQL is a very-high-level language, in which the programmer is able to avoid specifying a lot of data-manipulation details that would be necessary in languages like C++.
- What makes SQL viable is that its queries are “optimized” quite well, yielding efficient query executions.

- The principal form of a query is:
  - **SELECT** desired attributes
  - **FROM** one or more tables
  - **WHERE** condition about tuples of the tables

  - SQL introduction

Simple Queries in SQL

Our **SQL** queries will be based onthe following database schema.

- **Movie**(title, year, length, inColor, studioName, producerC)
- **StarsIn**(movieTitle, movieYear, starName)
- **MovieStar**(name, address, gender, birthdate)
- **MovieExec**(name, address, cert#, netWorth)
- **Studio**(name, address, cert#, netWorth)

Simple Queries in SQL

- Query all movies produced by Disney Studios in 1990
  - $\sigma_{\text{studioName='Disney' AND year=1990}}(\text{Movies})$
  - ```
    SELECT *
    FROM   Movies
    WHERE  studioName = 'Disney'
    AND    year = 1990;
    ```

title	year	length	inColor	studioName	procucerC#
Pretty Women …	1990	119	true	Disney	999
    ```

Projection in SQL

Find the title and length of all movies produced by Disney Studios in 1990.

$\pi_{\texttt{title,length}}(\sigma_{\texttt{studioName='Disney'AND year=1990}}(\text{Movies}))$

$\sigma_{\texttt{studioName='Disney'AND year=1990}}\pi_{\texttt{title, length}}((\text{Movies}))$?

```
SELECT title, length
FROM   Movies
WHERE  studioName = 'Disney'
AND    year = 1990;
```

title	length
Pretty Women	119
…	

Projection in SQL

we can modify the name of attributes. We can change title to name and length to duration in the previous example.

- ```
  SELECT title AS name, length AS duration
  FROM   Movies
  WHERE  studioName = 'Disney'
  AND    year = 1990;
  ```

- We can compute the length in hours
- ```
  SELECT title AS name,
         length/60 AS Length_In_Hours
  FROM   Movies
  WHERE  studioName = 'Disney'
  AND    year = 1990;
  ```

Projection in SQL

```
SELECT title,
       length/60 AS Length
       'hrs.' AS inHours
FROM   Movies
WHERE  studioName = 'Disney'
AND    year = 1990;
```

title	length	inHours
Pretty Women …	1.98334	hrs.

Selection in SQL

- We may build the WHERE part using six common comparison operators (=, <>, <, >, <=, >=)

- Movies made by MGM studios that either were made after 1970 or were less than 90 minutes long.

- ```
  SELECT title,
  FROM   Movies
  WHERE  ( year > 1970 or length <90) AND studioName =
  'MGM'
  ```

- We can compare strings
  - Dictionary rules.
  ```

Pattern Matching in SQL

- Retrieves the titles that starts with ‘Star’, then one blank and the 4 last chars can be anything.
- ```
  SELECT title
  FROM   Movies
  WHERE  title LIKE 'Star _ _ _ _';
  ```
- So, possible matches can be:
  ‘Star War’, ‘Star Trek’
  ```

Dates and Times

- A date constant is represented by the keyword DATE followed by a quoted string.
- For example: `DATE '1961-08-24'`
- Note the strict format of the `'YYYY-mm-dd'`

Ordering the Output

- To get output in sorted order, we add to the select-from-where statement a clause:

  ORDER BY <list of attributes>

- The order is by default ascending (ASC), but we can get the output highest-first by appending the keyword DESC.

- To get the movies listed by length, shortest first, and among movies of equal length, alphabetically, we can say:

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990
ORDER BY length, title;
```

QUERIES INVOLVING MORE THAN ONE RELATION

Products and Joins in SQL

Disambiguating Attributes

Tuple Variables

Products and Joins in SQL

- Suppose we want to know the name of the producer of star wars.

$\sigma_{\text{title='StarWars'ANDproducerC\#=cert\#}}(\text{Movies} \times \text{MovieExec})$

```
SELECT *
FROM   Movies, MovieExec
WHERE  title = 'Star Wars'
AND    producerC# = cert#;
```

Basic Selects

- Basics on Selects examples

Disambiguating Attributes

- Sometimes we ask a query involving several relations, with two or more attributes with the same name.
  - **R.A** refers to attribute A of relation R.

  - **MovieStar**(name, address, gender, birthdate)
  - **MovieExec**(name, address, cert#, netWorth)

```
SELECT MovieStar.name, MovieExec.name
FROM   MovieStar, MovieExec
WHERE MovieStar.address =
            MovieExec.address;
```

Tuple Variables

Two stars that share an address

```
SELECT Star1.name, Star2.name
FROM MovieStar Star1, MovieStar Star2
WHERE Star1.address = Star2.address
AND   Star1.name < Star2.name;
```

What happens if the second condition is omitted?

Union, Intersection, and Difference of Queries

- Its possible to use Union, Intersection, and except in SQL queries.

- Query the names and addresses of all female movie stars who are also movie executives with a net worth over $10,000,000
  - **MovieStar**(name, address, gender, birthdate)
  - **MovieExec**(name, address, cert#, netWorth)

```
(SELECT name, address FROM MovieStar
WHERE  gender = 'F')
INTERSECT
(SELECT name, address FROM MovieExec
WHERE  netWorth > 10000000)
```

Union, Intersection, and Difference of Queries

Query the names and addresses of movie stars who are not movie executives.

- **MovieStar**(name, address, gender, birthdate)
- **MovieExec**(name, address, cert#, netWorth)

```
(SELECT name, address FROM MovieStar)
except
(SELECT name, address FROM MovieExec)
```

Union, Intersection, and Difference of Queries

- The two tables most be compatible

- Query all the titles and years of movies that appeared in either the Movies or StarsIn relations.
  - **Movie**(title, year, length, inColor, studioName, producerC)
  - **StarsIn**(movieTitle, movieYear, starName)

```
(SELECT title, year FROM Movies)
UNION
(SELECT movieTitle AS title, movieYear AS year
FROM StarsIn)
```

Basic Variables and set operators

- Table variables and set operators examples

Null Values and Comparisons Involving NULL

Different interpretations for NULL values:

1. Value unknown
   I know there is some value here but I don't know what it is?
   1. Unknown birth date
2. Value inapplicable
   There is no value that make sense here.
   1. Spouse of a single movie star
3. Value withheld
   We are not entitled to know this value.
   1. Telephone number of stars which is known but may be shown as null

Null Values and Comparisons Involving NULL

- Two rules
  - Null plus arithmetic operators is null
  - When comparing the value of a null if we use = or like the value is unknown.
  - We use: x IS NULL or x IS NOT NULL

- How unknown operates in logical expressions
  - If true is considered 1 and false is considred 0, then unknown is considered 0.5.
  - And is like min: true and unknown is unknown, false and unknown is false.
  - OR is like max: true and unknown is true, false and unknown is unknown.
  - Negation is 1 –x: negation of unknown is unknown.

Null Values

- Null Values examples

SUBQUERIES

Subqueries that Produce Scalar Values

Conditions Involving Relations

Conditions Involving Tuples

Correlated Subqueries

Subqueries in *From* Clauses

SQL Join Expressions

Natural Joins

Outer Joins

Subqueries that Produce Scalar Values

Query the producer of Star Wars.

- **Movie**(title, year, length, inColor, studioName, producerC)
- **MovieExec**(name, address, cert#, netWorth)

```
SELECT name
FROM MovieExec, Movies
WHERE title = "Star Wars" AND producerC# = cert#
```

We just need the movie relation only to get the certificate number.

Once we have that we could query the MovieExec for the name.

Subqueries that Produce Scalar Values

- use a subquery to get the producerC#

```
SELECT name
FROM MovieExec
WHERE cert# = (SELECT producerC#
               FROM   Movies
               WHERE  title = 'Star Wars'
              );
```

- What would happen if the subquery retrieve zero or more than one tuple?
  - Runtime error

```
SELECT name
FROM MovieExec
WHERE cert# = 12345
```

6.3.2 Conditions Involving Relations

- There are a number of SQL operators that can be applied to a relation R and produces a Boolean result.

- EXISTS R is true iff R is not empty.
- s IN R is true iff s is equal to one of the values in R.
- s > ALL R is true iff s is greater than every value in unary relation R. Other comparison operators (<, <=, >=, =, <>) can be used.
- s > ANY R is true iff s is greater than at least one value in unary relation R. Other comparison operators (<, <=, >=, =, <>) can be used.

6.3.2 Conditions Involving Relations

- To negate EXISTS, ALL, and ANY operators, put NOT in front of the entire expression.
- NOT EXISTS R, NOT s > ALL R, NOT s > ANY R
- s NOT IN R is the negation of IN operator.
- Some situations of these operators are equal to other operators.

- For example:
  s <> ALL R is equal to s NOT IN R
  s = ANY R is equal to s IN R

6.3.3 Conditions Involving Tuples

- A tuple in SQL is represented by a parenthesized list of scalar values.
- Examples:
  (123, ‘I am a string’, 0, NULL)
  (name, address, salary)
- The first example shows all constants and the second shows attributes.
- Mixing constants and attributes are allowed.

6.3.3 Conditions Involving Tuples (cont'd)

- Example:

- ```
  ('Tom', 'Smith') IN
        (SELECT firstName, LastName
         FROM   foo);
  ```

- Note that the order of the attributes must be the same in the tuple and the SELECT list.
  ```

Conditions Involving Tuples

Example 6.20:

Query all the producers of movies in which `LEONARDO DICAPRIO` stars.

- **Movie**(title, year, length, inColor, studioName, producerC (movieTitle, movieYear, starName)
- **MovieStar**(name, address, gender, birthdate)
- **MovieExec**(name, address, cert#, netWorth)
- **Studio**(name, address, cert#, netWorth)

```
SELECT name, cert#
); FROM   MovieExec;
WHERE  cert# IN
       (SELECT producerC#
        FROM   Movies
        WHERE  (title, year) IN
            (SELECT movieTitle, movieYear
             FROM   StarsIN
             WHERE  starName = 'LEONARDO DICAPRIO')
```

Conditions Involving Tuples

- Note that sometimes, you can get the same result without the expensive subqueries.
- For example, the previous query can be written as follows:

```
SELECT name
FROM   MovieExec, Movies, StarsIN
WHERE  cert# = producerC#
AND    title = movieTitle
AND    year  = movieYear
And    starName = 'LEONARDO DICAPRIO';
```

Correlated Subqueries

- The simplest subquery is evaluated once and the result is used in a higher-level query.
- Some times a subquery is required to be evaluated several times, once for each assignment of a value that comes from a tuple variable outside the subquery.
- A subquery of this type is called correlated subquery.

Correlated Subqueries (cont'd)

Query the titles that have been used for two or more movies.

```
SELECT title
FROM   Movies old
WHERE  year < ANY
            (SELECT year
             FROM   Movies
             WHERE  title = old.title);
```

- Start with the inner query
  - If old.title was a constant this would have made total sense
    - Where title = “king kong”
  - Nested loop.
    - For each value of old title we run the the nested subquery

Subqueries

- Subqueries by Dr. Widom

Subqueries in *From* Clauses

- **SELECT** A_1,… A_n
- **FROM** R_1, …. R_m
- **WHERE** condition ← up to now we have used sub-query

- **SELECT** A_1,… A_n ← use sub-query to generate an attribute
- **FROM** R_1, …. R_m ← use sub-query to generate a table to condition
- **WHERE** condition

Subqueries in *From* Clauses

- In a FROM list, we my use a parenthesized subquery.
- The subquery must have a tuple variable or alias.

Query the producers of LEONARDO DICAPRIO's movies.

We can write a subquery that produces a new table that can be called in the from part of the query.

```
Select name
FROM   MovieExec,
       (SELECT producerC#
        FROM   Movies, StarsIN
        WHERE  title = movieTitle
        AND    year  = movieYear
        AND    starName = 'LEONARDO DICAPRIO'
       ) Prod
WHERE  cert# = Prod.producerC#;
```

Subqueries

- Subqueries in *From* Clauses examples

SQL Join Expressions

- Join operators construct new temp relations from existing relations.
- These relations can be used in any part of the query that you can put a subquery.
- Cross join is the simplest form of a join.
- Actually, this is synonym for Cartesian product.
- For example:

```
From Movies CROSS JOIN StarsIn
```

is equal to:

```
From Movies, StarsIn
```

SQL Join Expressions

- If the relations we used are:
  - Movies(title, year, length, genre, studioName, producerC#)
  - StarsIn(movieTitle, movieYear, starName)

  - Then the result of the CROSS JOIN would be a relation with the following attributes:
    - R(title, year, length, genre, studioName, producerC#, movieTitle, movieYear, starName)

- Note that if there is a common name in the two relations, then the attributes names would be qualified with the relation name.

SQL Join Expressions

- Cross join by itself is rarely a useful operation.
- Usually, a theta-join is used as follows:

```
FROM R JOIN S ON condition
```

- For example:

```
Movies JOIN StarsIn ON
    title = movieTitle AND
    year  = movieYear
```

- The result would be the same number of attributes but the tuples would be those that agree on both the title and year.

SQL Join Expressions

- Note that in the previous example, the title and year are repeated twice. Once as title and year and once as movieTitle and movieYear.
- Considering the point that the resulting tuples have the same value for title and movieTitle, and year and movieYear, then we encounter the redundancy of information.
- One way to remove the unnecessary attributes is projection. You can mention the attributes names in the SELECT list.

Natural Joins

- Natural join and theta-join differs in:
  1. The join condition
     All pairs of attributes from the two relations having a common name are equated, and also there are no other conditions.
  2. The attributes list
     One of each pair of equated attributes is projected out.
- Example

```
MovieStar NATURAL JOIN MovieExec
```

Natural Joins

Query those stars who are executive as well.

The relations are:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

```
SELECT MovieStar.name
FROM MovieStar NATURAL JOIN MovieExec
```

Outer Joins

- Outer join is a way to augment the result of a join by dangling tuples, padded with null values.

Example 6.25

Consider the following relations:

```
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
```

Then

```
MovieStar NATURAL FULL OUTER JOIN MovieExec
```

Will produce a relation whose tuples are of 3 kinds:

1. Those who are both movie stars and executive
2. Those who are movie star but not executive
3. Those who are executive but not movie star

Outer Joins (cont'd)

- We can replace keyword FULL with LEFT or RIGHT to get two new join.
- `NATURAL LEFT OUTER JOIN` would yield the first two tuples but not the third.
- `NATURAL RIGHT OUTER JOIN` would yield the first and third tuples but not the second.
  - We can have theta-outer-join as follows:
  - `R FULL OUTER JOIN S ON condition`
  - `R LEFT OUTER JOIN S ON condition`
  - `R RIGHT OUTER JOIN S ON condition`

FULL-RELATION OPERATIONS

Eliminating Duplicates

Duplicates in Unions, Intersections, and Differences

Grouping and Aggregation in SQL

Aggregation Operators

Grouping

Grouping, Aggregation, and Nulls

Having Clauses

Exercises for Section 6.4

Eliminating Duplicates

Query all the producers of movies in which `LEONARDO DICAPRIO` stars.

```
SELECT DISTINCT name
FROM   MovieExec, Movies, StarsIN
WHERE  cer# = producerC#
AND    title = movieTitle
AND    year  = movieYear
And    starName = LEONARDO DICAPRIO';
```

Duplicates in Unions, Intersections, and Differences

- Duplicate tuples are eliminated in UNION, INTERSECT, and EXCEPT.
- In other words, bags are converted to sets.
- If you don't want this conversion, use keyword ALL after the operators.

```
(SELECT title, year FROM Movies)
UNION ALL
(SELECT movieTitle AS title, movieYear AS year FROM
   StarsIn);
```

Grouping and Aggregation in SQL

- We can partition the tuples of a relation into "groups" based on the values of one or more attributes. The relation can be an output of a SELECT statement.
- Then, we can aggregate the other attributes using aggregation operators.
- For example, we can sum up the salary of the employees of each department by grouping the company into departments.

Aggregation Operators

- SQL uses the five aggregation operators:
  SUM, AVG, MIN, MAX, and COUNT
- These operators can be applied to scalar expressions, typically, a column name.
- One exception is COUNT(*) which counts all the tuples of a query output.
- We can eliminate the duplicate values before applying aggregation operators by using DISTINCT keyword. For example:
  `COUNT(DISTINCT x)`

Find the average net worth of all movie executives.

```
SELECT AVG(netWorth)
FROM   MovieExec;
```

Aggregation Operators

Count the number of tuples in the StarsIn relation.

```
SELECT COUNT(*)
FROM   StarsIn;

SELECT COUNT(starName)
FROM   StarsIn;
```

These two statements do the same but you will see the difference in later slides.

Grouping

- We can group the tuples by using GROUP BY clause following the WHERE clause.
- The keywords GROUP BY are followed by a list of grouping attributes.

Find sum of the movies length each studio is produced.

```
SELECT   studioName,
         SUM(length) AS Total_Length
FROM     Movies
GROUP BY studioName;
```

Grouping

- In a SELECT clause that has aggregation, only those attributes that are mentioned in the GROUP BY clause may appear unaggregated.
- For example, in previous example, if you want to add genre in the SELECT list, then, you must mention it in the GROUP BY list as well.

```
SELECT    studioName, genre,
          SUM(length) AS Total_Length
FROM      Movies
GROUP BY studioName, genre;
```

Grouping

- It is possible to use GROUP BY in a more complex queries about several relations.
- In these cases the following steps are applied:
  1. Produce the output relation based on the select-from-where parts.
  2. Group the tuples according to the list of attributes mentioned in the GROUP BY list.
  3. Apply the aggregation operators

Create a list of each producer name and the total length of film produced.

```
SELECT name, SUM(length)
FROM   MovieExec, Movies
WHERE  producerC# = cert#
GROUP BY name;
```

Grouping, Aggregation, and Nulls

- What would happen to aggregation operators if the attributes have null values?
- There are a few rules to remember

1. NULL values are ignored when the aggregation operator is applied on an attribute.
2. COUNT(*) counts all tuples of a relation, therefore, it counts the tuples even if the tuple contains NULL value.
3. NULL is treated as an ordinary value when forming groups.
4. When we perform an aggregation, except COUNT, over an empty bag, the result is NULL. The COUNT of an empty bag is 0

Grouping, Aggregation, and Nulls

Consider a relation R(A, B) with one tuple, both of whose components are NULL. What's the result of the following SELECT?

```
SELECT A, COUNT(B)
FROM   R
GROUP BY A;
```

The result is (NULL, 0) but why?

What's the result of the following SELECT?

```
SELECT A, COUNT(*)
FROM   R
GROUP BY A;
```

The result is (NULL, 1) because COUNT(*) counts the number of tuples and this relation has one tuple.

Grouping, Aggregation, and Nulls

What's the result of the following SELECT?

```
SELECT A, SUM(B)
FROM   R
GROUP BY A;
```

The result is (NULL, NULL) because SUM(B) address one NULL value which is NULL.

HAVING Clauses

- So far, we have learned how to restrict tuples from contributing in the output of a query.
- How about if we don't want to list all groups?
- HAVING clause is used to restrict groups.
- HAVING clause followed by one or more conditions about the group.

Query the total film length for only those producers who made at least one film prior to 1930.

```
SELECT name, SUM(length)
FROM   MovieExec, Movies
WHERE  producerC# = cert#
GROUP BY name
HAVING MIN(year) < 1930;
```

HAVING Clauses

- The rules we should remember about HAVING:
  1. An aggregation in a HAVING clause applies only to the tuples of the group being tested.
  2. Any attribute of relations in the FROM clause may be aggregated in the HAVING clause, but only those attributes that are in the GROUP BY list may appear unaggregated in the HAVING clause (the same rule as for the SELECT clause).

HAVING Clauses

- The order of clauses in SQL queries would be:
  - SELECT
  - FROM
  - WHERE
  - GROUP BY
  - HAVING
- Only SELECT and FROM are mandatory.
- There is one important difference between SQL HAVING and SQL WHERE clauses. The SQL WHERE clause condition is tested against each and every row of data, while the SQL HAVING clause condition is tested against the groups and/or aggregates specified in the SQL GROUP BY clause and/or the SQL SELECT column list.

DATABASE MODIFICATIONS

Insertion

Deletion

Updates

Insertion

- The syntax of INSERT statement:

```
INSERT INTO R(A1, ..., AN)
VALUES (v1, ..., vn);
```

- If the list of attributes doesn't include all attributes, then it put default values for the missing attributes.

Insertion

If we are sure about the order of the attributes, then we can write the statement as follows:

```
INSERT INTO StarsIn
VALUES ('The Maltese Falcon', 1942, 'Sydney
  Greenstreet');
```

If not

```
INSERT INTO StarsIn(MovieTitle, movieYear,
  starName)
VALUES ('The Maltese Falcon', 1942, 'Sydney
  Greenstreet');
```

Insertion

- The simple insert can insert only one tuple, however, if you want to insert multiple tuples , then you can use the following syntax:

```
INSERT INTO R(A1, ..., AN)
  SELECT v1, ..., vn
  FROM   R1, R2, ..., RN
  WHERE  <condition>;
```

- Suppose that we want to insert all studio names that are mentioned in the Movies relation but they are not in the Studio yet.

```
INSERT INTO Studio(name)
  SELECT studioName
  FROM   Movies
  WHERE  studionName NOT IN
            (SELECT name
             FROM Studio);
```

Deletion

- The syntax of DELETE statement:

```
DELETE FROM R
WHERE  <condition>;
```

- Every tuples satisfying the condition will be deleted from the relation R.

```
DELETE FROM StarsIn
WHERE  movieTitle = 'The Maltese Falcon' AND
       movieYear = 1942 AND
       starName = 'Sydney Greenstreet';
```

Delete all movie executives whose net worth is less than ten million dollars.

```
DELETE FROM MovieExec
WHERE  netWorth < 10000000;
```

Updates

- The syntax of UPDATE statement:

```
UPDATE R
SET   <value-assignment>
WHERE <condition>;
```

- Every tuples satisfying the condition will be updated from the relation R.
- If there are more than one value-assignment, we should separate them with comma.

Attach the title 'Pres.' in front of the name of every movie executive who is the president of a studio.

```
UPDATE MovieExec
SET name = 'Pres.' || name
WHERE  cert# IN (SELECT presC# FROM Studio);
```

TRANSACTIONS IN SQL

Serializability

Atomicity

Transactions

Read-Only Transactions

Dirty Reads

Other Isolation Levels

Exercises for Section 6.6

6.6 Transactions in SQL

- Up to this point, we assumed that:
  - the SQL operations are done by one user.
  - The operations are done one at a time.
  - There is no hardware/software failure in middle of a database modification. Therefore, the operations are done atomically.
- In Real life, situations are totally different.
- There are millions of users using the same database and it is possible to have some concurrent operations on one tuple.

6.6.1 Serializability

- In applications like web services, banking, or airline reservations, hundreds to thousands operations per second are done on one database.
- It's quite possible to have two or more operations affecting the same, let's say, bank account.
- If these operations overlap in time, then they may act in a strange way.
- Let's take an example.

6.6.1 Serializability (cont'd)

Example 6.40

Consider an airline reservation web application. Users can book their desired seat by themselves.

The application is using the following schema:

`Flights(fltNo, fltDate, seatNo, seatStatus)`

When a user requests the available seats for the flight no 123 on date 2011-12-15, the following query is issued:

6.6.1 Serializability (cont'd)

```
SELECT seatNo
FROM   Flights
WHERE  fltNo = 123 AND
       fltDate = DATE '2011-12-25' AND
       seatStatus = 'available';
```

When the customer clicks on the seat# 22A, the seat status is changed by the following SQL:

```
UPDATE Flights
SET    seatStatus = 'occupied'
WHERE  fltNo = 123 AND
       fltDate = DATE '2011-12-25' AND
       seatNo = '22A';
```

6.6.1 Serializability (cont'd)

- What would happen if two users at the same time click on the reserve button for the same seat#?
- Both see the same seats available and both reserve the same seat.
- To prevent these happen, SQL has some solutions.
- We group a set of operations that need to be performed together. This is called 'transaction'.

6.6.1 Serializability (cont'd)

- For example, the query and the update in example 6.40 can be grouped in a transaction.
- SQL allows the programmer to state that a certain transaction must be serializable with respect to other transactions.
- That is, these transactions must behave as if they were run serially, one at a time with no overlap.

6.6.2 Atomicity

- What would happen if a transaction consisting of two operations is in progress and after the first operation is done, the database and/or network crashes?
- Let's take an example.

6.6.2 Atomicity (cont'd)

Example 6.41

Consider a bank's account records system with the following relation:

Accounts(acctNo, balance)

Let's suppose that $100 is going to transfer from acctNo 123 to acctNo 456.

To do this, the following two steps should be done:

1. Add $100 to account# 456
2. Subtract $100 from account# 123.

6.6.2 Atomicity (cont'd)

The needed SQL statements are as follows:

```
UPDATE Accounts
SET balance = balance + 100
WHERE acctNo = 456;

UPDATE Accounts
SET balance = balance - 100
WHERE acctNo = 123;
```

What would happen if right after the first operation, the database crashes?

6.6.2 Atomicity (cont'd)

- The problem addressed by example 6.41 is that certain combinations of operations need to be done atomically.
- That is, either they are both done or neither is done.

6.6.3 Transactions

- The solution to the problems of serialization and atomicity is to group database operations into transactions.
- A transaction is a set of one or more operations on the database that must be executed atomically and in a serializable manner.
- To create a transation, we use the following SQL command:

```
START TRANSACTION
```

6.6.3 Transactions (cont'd)

- There are two ways to end a transaction:
  1. The SQL receives COMMIT command.
  2. The SQL receives ROLLBACK command.
- COMMIT command causes all changes become permanent in the database.
- ROLLBACK command causes all changes undone.

6.6.4 Read-Only Transactions

- We saw that when a transaction read a data and then want to write something, is prone to serialization problems.
- When a transaction only reads data and does not write data, we have more freedom to let the transaction execute in parallel with other transactions.
- We call these transactions read-only.

6.6.4 Read-Only Transactions (cont'd)

Example 6.43

Suppose we want to read data from the Flights relation of example 6.40 to determine whether a certain seat was available?

What's the worst thing that can happen?

When we query the availability of a certain seat, that seat was being booked or was being released by the execution of some other program. Then we get the wrong answer.

6.6.4 Read-Only Transactions (cont'd)

- If we tell the SQL that our current transaction is read-only, then SQL allows our transaction be executed with other read-only transactions in parallel.
- The syntax of SQL command for read-only setting:

```
SET TRANSACTION READ ONLY;
```

- We put this statement before our read-only transaction.

6.6.4 Read-Only Transactions (cont'd)

- The syntax of SQL command for read-write setting:

```
SET TRANSACTION READ WRITE;
```

- We put this statement before our read-write transaction.
- This option is the default.

6.6.5 Dirty Reads

- The data that is written but not committed yet is called dirty data.
- A dirty read is a read of dirty data written by another transaction.
- The risk in reading dirty data is that the transaction that wrote it never commit it.

- Sometimes dirty read doesn't matter much and is not worth
  - The time consuming work by the DBMS that is needed to prevent data reads
  - The loss of parallelism that results from waiting until there is no possibility of a dirty read

6.6.5 Dirty Reads (cont'd)

Example 6.44

Consider the account transfer of example 6.41.

Here are the steps:

1. Add money to account 2.
2. Test if account 1 has enough money?
    a. If there is not enough money, remove the money from account 2 and end.
    b. If there is, subtract the money from account 1 and end.

Imagine, there are 3 accounts A1, A2, and A3 with $100, $200, and $300.

6.6.5 Dirty Reads (cont'd)

Let's suppose:

Transaction T1 transfers $150 from A1 to A2

Transaction T2 transfers $250 from A2 to A3

What would happen if the dirty read is allowed?

- T2 executes step (1) adds 250 to A3 which now has 550
- T1 executes step (1) adds 150 to A2 which now has 350
- T2 executes step (2), A2 has enough fund
- T1 executes step (2) A1 doesn’t have enough fund
- T2 executes step (2b) and leaves A2 with $100
- T1 executes step (2a) and leaves A1 with $-50

- How important is it in the reservation scenario?

6.6.5 Dirty Reads (cont'd)

- The syntax of SQL command for dirty-read setting:

```
SET TRANSACTION READ WRITE
ISOLATION LEVEL READ UNCOMMITTED;
```

- We put this statement before our read-write transaction.
- This option is the default.

6.6.6 Other Isolation Levels

- There are four isolation level.
- We have seen the first two before.
  - Serializable (default)
  - Read-uncommitted
  - Read-committed
    Syntax:

```
SET TRANSACTION
ISOLATION LEVEL READ COMMITTED;
```

6.6.6 Other Isolation Levels (cont'd)

- For each the default is 'READ WRITE' (except the isolation READ UNCOMMITTED that the default is 'READ ONLY') and if you want 'READ ONLY', you should mention it explicitly.
- The default isolation level is 'SERIALIZABLE'.
- Note that if a transaction T is acting in 'SERIALIZABLE' level and the other one is acting in 'READ UNCOMMITTED' level, then this transaction can see the dirty data of T. It means that each one acts based on their level.

6.6.6 Other Isolation Levels (cont'd)

- Under READ COMMITTED isolation, it forbids reading the dirty data.
- But it does not guarantee that if we issue several queries, we get the same tuples.
- That's because there may be some new committed tuples by other transactions.
- The query may show more tuples because of the phantom tuples.
- A phantom tuple is a tuple that is inserted by other transactions.

6.6.6 Other Isolation Levels (cont'd)

Example 6.46

Let's consider the seat choosing problem under 'READ COMMITTED' isolation.

Your query won't see seat as available if another transaction reserved it but not committed yet.

You may see different set of seats in subsequent queries depends on if the other transactions commit their reservations or rollback them.

6.6.6 Other Isolation Levels (cont'd)

- Properties of SQL isolation levels

Isolation Level	Dirty Read	Phantom
Read Uncommitted	✓	✓
Read Committed	-	✓
Serializable	-	-