SQL: Advanced concepts

- Incomplete information
  - dealing with null values
  - Outerjoins
- Recursion in SQL99
- Interaction between SQL and a programming language
  - embedded SQL
  - dynamic SQL

Incomplete Information: Null Values

- How does one deal with incomplete information?
- Perhaps the most poorly designed and the most often criticized part of SQL:

  "... [this] topic cannot be described in a manner that is simultaneously both comprehensive and comprehensible."

  "... those SQL features are not fully consistent; indeed, in some ways they are fundamentally at odds with the way the world behaves."

  "A recommendation: avoid nulls."

  "Use [nulls] properly and they work for you, but abuse them, and they can ruin everything"

Theory of incomplete information

- What is incomplete information?
- Which relational operations can be evaluated correctly in the presence of incomplete information?
- What does “evaluated correctly” mean?

Incomplete information in SQL

- Simplifies things too much.
- Leads to inconsistent answers.
- But sometimes nulls are extremely helpful.

Sometimes we don't have all the information

Movies

Title	Director	Actor
Dr. Strangelove	Kubrick	Sellers
Dr. Strangelove	Kubrick	Scott
null	Polanski	Nicholson
null	Polanski	Huston
Star Wars	Lucas	Ford
Frantic	null	Ford

- What could null possibly mean? There are three possibilities:
- Value exists, but is unknown at the moment.
- Value does not exist.
- There is no information.

Representing relations with nulls: Codd tables

- In Codd tables, we put distinct variables for null values:

T_1

Title	Director	Actor
Dr. Strangelove	Kubrick	Sellers
Dr. Strangelove	Kubrick	Scott
x	Polanski	Nicholson
y	Polanski	Huston
Star Wars	Lucas	Ford
Frantic	z	Ford

- Semantics of a Codd table T is the set $POSS(T)$ of all tables without nulls it can represent.
- That is, we substitute values for all variables.

Tables in $POSS(T_1)$

The following tables are all in $POSS(T_1)$ is the set of all relations it can represent. Examples:

Title	Director	Actor
Dr. Strangelove	Kubrick	Sellers
Dr. Strangelove	Kubrick	Scott
Star Wars	Polanski	Nicholson
Titanic	Polanski	Huston
Star Wars	Lucas	Ford
Frantic	Cameron	Ford

Title	Director	Actor
Dr. Strangelove	Kubrick	Sellers
Dr. Strangelove	Kubrick	Scott
Titanic	Polanski	Nicholson
Titanic	Polanski	Huston
Star Wars	Lucas	Ford
Frantic	Lucas	Ford

Title	Director	Actor
Dr. Strangelove	Kubrick	Sellers
Dr. Strangelove	Kubrick	Scott
Chinatown	Polanski	Nicholson
Chinatown	Polanski	Huston
Star Wars	Lucas	Ford
Frantic	Polanski	Ford

Querying Codd tables

- Suppose Q is a relational algebra, or SQL query, and T is a Codd table. What is $Q(T)$?
- We only know how to apply Q to usual relations, so we can find:

$$\hat{Q}(T) \quad = \quad \{Q(R) \ \mid \ R \in POSS(T)\}$$

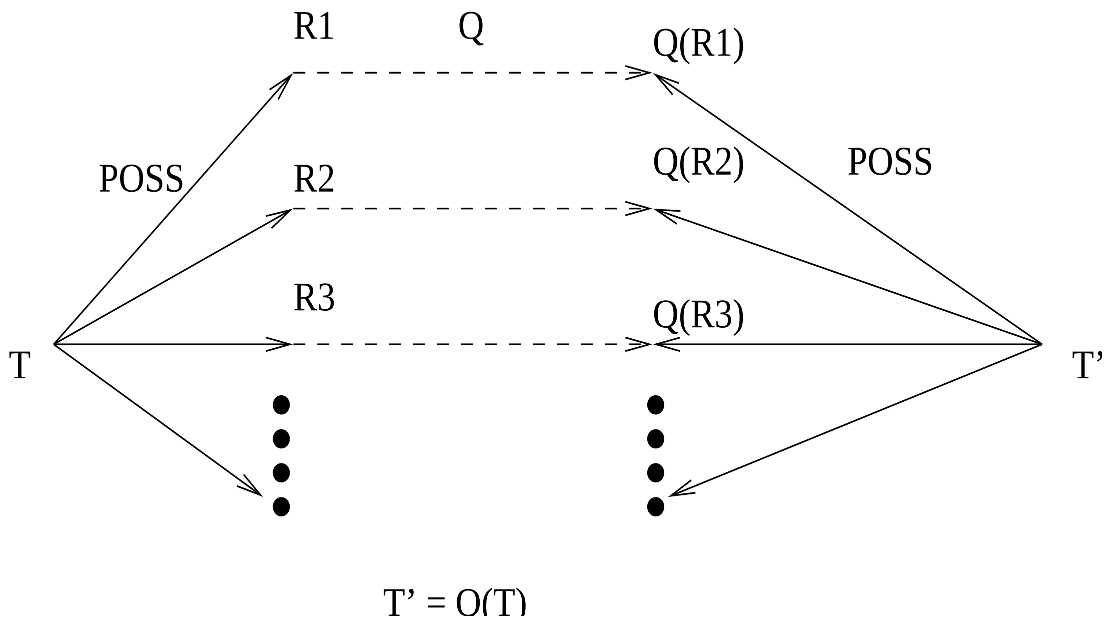
- If there were a Codd table T' such that $POSS(T') = \hat{Q}(T)$, then we would say that T' is $Q(T)$. That is,

$$POSS(Q(T)) \quad = \quad \{Q(R) \ \mid \ R \in POSS(T)\}$$

- Question: Can we always find such a table T'?

Representation systems

- Semantically correct way of answering queries over tables:



- Question: can we always find $Q(T)$ for every T, and every Q in a query language?
- **Bad news:** This is not possible even for very simple relational algebra queries.

Selection and Codd tables

Table: $T =$

A	B
0	1
x	2

Query: $Q = \sigma_{A=3}(T)$

Suppose there is T' such that $POSS(T') = \{Q(R) \mid R \in POSS(T)\}$. Consider:

$R_1 =$

A	B
0	1
2	2

and $R_2 =$

A	B
0	1
3	2

and

$Q(R_1) = \emptyset$, $Q(R_2) = \{(3,2)\}$, and hence T' cannot exist, because

$$\emptyset \in POSS(T') \quad \text{if and only if} \quad T' = \emptyset$$

What can one do?

- Idea #1: consider *certain* answers.
- A tuple t is in the *certain answer* to Q on $T_1, \ldots, T_n$ if it is in

$$Q(R_1, \ldots, R_n) \quad \text{for all } R_1 \in POSS(T_1), \ldots, R_n \in POSS(T_n)$$

- Idea #2: extend Codd tables by adding constraints on null values (e.g., some must be equal, some cannot be equal, etc).
- Combining these ideas makes it possible to evaluate queries with incomplete information.
- However, evaluation algorithms, and – more importantly – query results, are often completely incomprehensible.

Incomplete information in SQL

- SQL approach: there is a single general purpose NULL for all cases of missing/inapplicable information
- Nulls occur as entries in tables; sometimes they are displayed as `null`, sometimes as '–'
- They immediately lead to comparison problems
- The union of
  `SELECT * FROM R WHERE R.A=1` and
  `SELECT * FROM R WHERE R.A<>1` should be the same as
  `SELECT * FROM R`.
- But it is not.
- Because, if `R.A` is null, then neither `R.A=1` nor `R.A<>1` evaluates to *true*.

Nulls cont'd

- R.A has three values: 1, null, and 2.
- `SELECT * FROM R WHERE R.A=1` returns

A
1

- `SELECT * FROM R WHERE R.A<>1` returns

A
2

- How to check = null? New comparison: `IS NULL`.
- `SELECT * FROM R WHERE R.A IS NULL` returns

A
null

- `SELECT * FROM R` is the union of
  `SELECT * FROM R WHERE R.A=1`,
  `SELECT * FROM R WHERE R.A<>1`, and
  `SELECT * FROM R WHERE R.A IS NULL`.

Nulls and other operations

- What is 1+null? What is the truth value of '3 = null'?
- Nulls cannot be used explicitly in operations and selections: `WHERE R.A=NULL` or `SELECT 5-NULL` are illegal.
- For any arithmetic, string, etc. operation, if one argument is null, then the result is null.
- For `R.A`={1,null}, `S.B`={2},

  ```
  SELECT R.A + S.B
  FROM R, S
  ```

  returns {3, null}.
- What are the values of `R.A=S.B`? When R.A=1, S.B=2, it is *false*. When R.A=null, S.B=2, it is *unknown*.

The logic of nulls

- How does *unknown* interact with Boolean connectives? What is `NOT` *unknown*? What is *unknown* `OR` *true*?

-

`x`	`NOT x`
true	false
false	true
unknown	unknown

-

`AND`	true	false	unknown
true	true	false	unknown
false	false	false	false
unknown	unknown	false	unknown

-

`OR`	true	false	unknown
true	true	true	true
false	true	false	unknown
unknown	true	unknown	unknown

- Problem with null values: people rarely think in three-valued logic!

Nulls and aggregation

- Be ready for big surprises!

```
SELECT * FROM R
A
-----------
          1
          -

SELECT COUNT(*) FROM R

returns  2

SELECT COUNT(R.A) FROM R

returns 1
```

Nulls and aggregation

- One would expect nulls to propagate through arithmetic expressions
- `SELECT SUM(R.A) FROM R` is the sum

$$a_1 + a_2 + \ldots + a_n$$

  of all values in column `A`; if one is null, the result is null.
- But `SELECT SUM(R.A) FROM R` returns 1 if `R.A`=$\{$1,null$\}$.
- Most common rule for aggregate functions:
  first, ignore all nulls,
  and then compute the value.
- The only exception: `COUNT(*)`.

Nulls in subqueries: more surprises

- `R1.A` = {1,2}   `R2.A` = {1,2,3,4}

-
  ```
  SELECT R2.A
  FROM R2
  WHERE R2.A NOT IN (SELECT R1.A
                     FROM R1)
  ```
- Result: {3,4}
- Now insert a null into `R1`: `R1.A` = {1,2, null}
  and run the same query.
- The result is ∅!

Nulls in subqueries cont'd

- Although this result is counterintuitive, it is correct.
- What is the value of 3 NOT IN (SELECT R1.A FROM R1)?

```
3 NOT IN {1,2,null}
=   NOT (3 IN {1,2,null})
=   NOT((3 = 1) OR (3=2) OR (3=null))
=   NOT(false OR false OR unknown)
=   NOT (unknown)
=   unknown
```

- Similarly, `4 NOT IN {1,2,null}` evaluates to unknown, and `1 NOT IN {1,2,null}`, `2 NOT IN {1,2,null}` evaluate to false.
- Thus, the query returns $\emptyset$.

Nulls in subqueries cont'd

- What is a correct result of "theoretical" evaluation of queries with incomplete information?
- The result of

```
SELECT R2.A
FROM R2
WHERE R2.A NOT IN (SELECT R1.A
                   FROM R1)
```

would look like

A	condition
3	$x = 0$
4	$x \neq 0$
3	$y = 0$
4	$y = 0$

Nulls could be dangerous!

- Imagine US national missile defense system, with the database of missile targeting major cities, and missiles launched to intercept those.
- Query: Is there a missile targeting US that is not being intercepted?

```
SELECT M.#, M.target
FROM Missiles M
WHERE M.target IN (SELECT Name
                   FROM USCities) AND
      M.# NOT IN (SELECT I.Missile
                  FROM Intercept I
                  WHERE I.Status = 'active')
```

- Assume that a missile was launched to intercept, but its target wasn't properly entered in the database.

Nulls could be dangerous!

- Missile

#	Target
M1	A
M2	B
M3	C

Intercept

I#	Missile	Status
I1	M1	active
I2	null	active

- {A, B, C} are in USCities
- The query returns the empty set:
  `M2 NOT IN {M1, null}` and `M3 NOT IN {M1, null}`
  evaluate to $unknown$.
- although either M2 or M3 is not being intercepted!
- Highly unlikely? Probably (and hopefully). But never forget what caused the Mars Climate Orbiter to crash!

SQL and nulls: a simple solution

- In `CREATE TABLE`, use `not null`:

```
create table <name> (...
                    <attr> <type> not null,
                    ...)
```

When nulls are helpful: outerjoins

- Example:

Studio

Name	Title
'United'	'Licence to kill'
'United'	'Rain man'
'Dreamworks'	'Gladiator'

Film

Title	Gross
'Licence to kill'	156
'Rain man'	412
'Fargo'	25

- Query: for each studio, find the total gross of its movies:

```
SELECT Studio.Name, SUM(Film.Gross)
FROM Studio NATURAL JOIN Film
GROUP BY Studio.Name
```

- Answer: ('United', 568)
- But often we want ('Dreamworks', ...) as well!
- 'Dreamworks' is lost because 'Gladiator' doesn't match anything in Film.

Outerjoins

- What if we don't want to lose 'Dreamworks'?
- Use *outerjoins*:

```
SELECT Studio.Name, SUM(Film.Gross)
FROM Studio NATURAL LEFT OUTER JOIN Film
GROUP BY Studio.Name
```

- Result: ('United', 568), ('Dreamworks', null)
- `Studio NATURAL LEFT OUTER JOIN Film` is:

Name	Title	Gross
'United'	'Licence to kill'	156
'United'	'Rain man'	412
'Dreamworks'	'Gladiator'	null

Other outerjoins

- `Studio NATURAL RIGHT OUTER JOIN Film` is:

Name	Title	Gross
'United'	'Licence to kill'	156
'United'	'Rain man'	412
null	'Fargo	25

- `Studio NATURAL FULL OUTER JOIN Film` is:

Name	Title	Gross
'United'	'Licence to kill'	156
'United'	'Rain man'	412
'Dreamworks'	'Gladiator'	null
null	'Fargo	25

- Idea: we perform a join, but keep tuples that do not match, padding them with nulls.

Outerjoins cont'd

- In `R NATURAL` **LEFT** `OUTER JOIN S`,
  we keep non-matching tuples from `R`
  (that is, the relation on the *left*).
- In `R NATURAL` **RIGHT** `OUTER JOIN S`,
  we keep non-matching tuples from `S`
  (that is, the relation on the *right*).
- In `R NATURAL` **FULL** `OUTER JOIN S`,
  we keep non-matching tuples from both `R` and `S`
  (that is, from *both* relations).

Outerjoin and aggregation

- Warning: if you use outerjoins in aggregate queries, you get null values as results of all aggregates except COUNT

```
CREATE VIEW V (B1, B2) AS
 SELECT Studio.Name, Film.Gross
 FROM Studio NATURAL LEFT OUTER JOIN Film

SELECT * FROM V
```

Returns

B1	B2
'Dreamworks'	null
'United'	156
'United'	412

- `SELECT B1, SUM(B2) FROM V GROUP BY B1` returns ('Dreamworks, null), ('United', 568).
- `SELECT B1, COUNT(B2) FROM V GROUP BY B1` returns ('Dreamworks, 0), ('United', 2).

Outerjoins cont'd

- Some systems don't like the keyword `NATURAL` and would only let you do

```
R NATURAL LEFT/RIGHT/FULL OUTER JOIN S
             ON condition
```

- Example:

```
SELECT *
FROM Studio LEFT OUTER JOIN Film ON
       Studio.Title=Film.Title
```

- Result:

Name	Title	Title	Gross
'United'	'Licence to kill'	'Licence to kill'	156
'United'	'Rain man'	'Rain man'	412
'Dreamworks'	'Gladiator'	null	null

Limitations of SQL

- Reachability queries:

Flights	Src	Dest
	'EDI'	'LHR'
	'EDI'	'EWR'
	'EWR'	'LAX'
	...	...

- Query: Find pairs of cities (A, B) such that one can fly from A to B with at most one stop:

```
  SELECT F1.Src, F2.Dest
  FROM Flights F1, Flights F2
  WHERE F1.Dest=F2.Src
UNION
  SELECT * FROM Flights
```

Reachability queries cont'd

- Query: Find pairs of cities (A, B) such that one can fly from A to B with at most two stops:

```
  SELECT F1.Src, F3.Dest
  FROM Flights F1, Flights F2, Flights F3
  WHERE F1.Dest=F2.Src AND F2.Dest=F3.Src
UNION
  SELECT F1.Src, F2.Dest
  FROM Flights F1, Flights F2
  WHERE F1.Dest=F2.Src
UNION
  SELECT * FROM Flights
```

Reachability queries cont'd

- For any fixed number k, we can write the query

  Find pairs of cities (A, B) such that one can fly from A to B with at most k stops

  in SQL.

- What about the general reachability query:

  Find pairs of cities (A, B) such that one can fly from A to B.

- SQL cannot express this query.

- Solution: SQL3 adds a new construct that helps express reachability queries. (May not yet exist in some products.)

Reachability queries cont'd

- To understand the reachability query, we formulate it as a rule-based query:

$$\begin{aligned} reach(x,y) &:- \mathit{flights}(x,y) \\ reach(x,y) &:- \mathit{flights}(x,z), \;\; reach(z,y) \end{aligned}$$

- One of these rules is *recursive*: *reach* refers to itself.
- Evaluation:
  - Step 0: $reach_0$ is initialized as the empty set.
  - Step $i+1$: Compute

$$\begin{aligned} reach_{i+1}(x,y) &:- \mathit{flights}(x,y) \\ reach_{i+1}(x,y) &:- \mathit{flights}(x,z), \;\; reach_i(z,y) \end{aligned}$$

  - Stop condition: If $reach_{i+1} = reach_i$, then it is the answer to the query.

Evaluation of recursive queries

- Example: assume that *flights* contains $(a, b), (b, c), (c, d)$.
- Step 0: $reach = \emptyset$
- Step 1: $reach$ becomes $\{(a, b), (b, c), (c, d)\}$.
- Step 2: $reach$ becomes $\{(a, b), (b, c), (c, d), (a, c), (b, d)\}$.
- Step 3: $reach$ becomes $\{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}$.
- Step 4: one attempts to use the rules, but infers no new values for $reach$. The final answer is thus:

$$\{(a, b), (b, c), (c, d), (a, c), (b, d), (a, d)\}$$

Recursion in SQL3

- SQL3 syntax mimics that of recursive rules:

```
WITH RECURSIVE Reach(Src,Dest) AS
 (
    SELECT * FROM Flights
  UNION
    SELECT F.Src, R.Dest
    FROM Flights F, Reach R
    WHERE F.Dest=R.Src
 )
SELECT * FROM Reach
```

Recursion in SQL3: syntactic restrictions

- There is another way to do reachability as a recursive rule-based query:

$$\begin{array}{l} reach(x,y) \ \text{:-}\ flights(x,y) \\ reach(x,y) \ \text{:-}\ reach(x,z),\ \ reach(z,y) \end{array}$$

- This translates into an SQL3 query:

```
WITH RECURSIVE Reach(Src,Dest) AS
 (  SELECT * FROM Flights
  UNION
    SELECT R1.Src, R2.Dest
    FROM Reach R1, Reach R2
    WHERE R1.Dest=R2.Src )
SELECT * FROM Reach
```

- However, most implementations will disallow this, since they support only *linear* recursion: recursively defined relation is only mentioned once in the `FROM` line.

Recursion in SQL3 cont'd

- A slight modification: suppose `Flights` has another attribute `aircraft`.
- Query: find cities reachable from Edinburgh.

```
WITH   Cities AS SELECT Src,Dest FROM Flights
       RECURSIVE Reach(Src,Dest) AS
(
    SELECT * FROM Cities
  UNION
    SELECT C.Src, R.Dest
    FROM Cities C, Reach R
    WHERE C.Dest=R.Src
 )
SELECT R.Dest
FROM Reach R
WHERE R.Src='EDI'
```

A note on negation

- Problematic recursion:

```
WITH RECURSIVE R(A) AS
 (SELECT S.A
  FROM S
  WHERE S.A NOT IN
             SELECT R.A FROM R)

SELECT * FROM R
```

- Formulated as a rule:

$$r(x) \;\text{:-}\; s(x), \neg r(x)$$

A note on negation cont'd

- Let s contain $\{1, 2\}$.
- Evaluation:

  After step 0: $r_0 = \emptyset$;

  After step 1: $r_1 = \{1, 2\}$;

  After step 2: $r_2 = \emptyset$;

  After step 3: $r_3 = \{1, 2\}$;

  $\ldots$

  After step $2n$: $r_{2n} = \emptyset$;

  After step $2n + 1$: $r_{2n+1} = \{1, 2\}$.
- Problem: it does not terminate!
- What causes this problem? Answer: Negation (that is, `NOT IN`).

A note on negation cont'd

- Other instances of negation:

  `EXCEPT`

  `NOT EXISTS`

- SQL3 has a set of very complicated rules that specify when the above operations can be used in `WITH RECURSIVE` definitions.
- A general rule: it is best to avoid negation in recursive queries.

SQL in the Real World

- SQL is good for querying, but not so good for complex computational tasks. It is not very good for displaying results nicely.
- Moreover, queries and updates typically occur inside complex computations, for which SQL is not a suitable language.
- Thus, one most often runs SQL queries from host programming languages, and then processes the results.
- One approach: extend SQL.
  SQL3 can do many queries that SQL2 couldn't do. But sometimes one still needs to do some operations in a programming language.
- SQL offers two flavors of communicating with a PL:

  embedded SQL,

  dynamic SQL.
- Basic rule: if you know SQL, and you know the programming language, then you know embedded/dynamic SQL.

SQL and programming languages cont'd

- Most languages provide an interface for communicating with a DBMS (Ada, C, Java, Cobol, etc).
- These interfaces differ in details, but conceptually they follow the same model.
- We learn this model using C as the host language.
- Examples of difference: SQL statements start and end with

  `EXEC SQL` and ; in C

  `EXEC SQL` and `END-EXEC` in Cobol
- `SQLSTATE` variable: the state of the database after each operation. It is

  `char`, length 6 in C,

  `character`, length 5 in Fortran,

  `array [1..5] of char` in Pascal

SQL/C interface

- DBMS tells the host language what the state of the database is via a special variable called `SQLSTATE`
- In C, it is commonly declared as `char SQLSTATE[6]`.
- Two most important values:

  `'00000'` means "no error"

  `'02000'` means: "requested tuple not found'.

  The latter is used to break loops.
- Why 6 characters? Because in C we commonly use the function `strcmp` to compare strings, which expects the last symbol to be `'\0'`. Thus we declare `char SQLSTATE[6]` and initially set the 6th character to `'\0'`.

SQL/C interface: declarations

- To declare variables shared between C and SQL, one puts them between

  `EXEC SQL BEGIN DECLARE SECTION`

  and

  `EXEC SQL END DECLARE SECTION`

- Each variable `var` declared in this section will be referred to as `:var` in SQL queries.

- Example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char title[20], theater[20];
    int showtime;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION
```

Simple insertions

- With these declarations, we can write a program that prompts the user for title, theater, and showtime, and inserts a tuple into Schedule.
- `void InsertIntoSchedule() {`

```
   /* declarations from the previous slide */

  /* your favorite routine for asking the user for
     3 values: two strings and one integer.
     those are put in title, theater, showtime */

EXEC SQL INSERT INTO Schedule
        VALUES (:theater, :title, :showtime);
}
```

- Note how we use variables in the SQL statement: `:theater` instead of `theater` etc.

Simple lookups

- Task: prompt the user for theater and showtime, and return the title (if it can be found), but only if it is a movie directed by Spielberg.
-

```
void FindTitle() {

  EXEC SQL BEGIN DECLARE SECTION;
    char th[20], tl[20];
    int s;
    char SQLSTATE[6];
  EXEC SQL END DECLARE SECTION;

  /* get the values of theater and showtime
     and put them in variables th and s    */
```

Simple lookups cont'd

```
    EXEC SQL  SELECT title
              INTO   :tl
              FROM   Schedule S, Movies M
              WHERE  S.title = M.title AND
                     M.director = 'Spielberg' AND
                     S.showtime = :s;

    if (strcmp(SQLSTATE,"02000") != 0)
       printf("title = %s\n", tl)
     else printf("no title found\n");

}
```

- The comparison `strcmp(SQLSTATE,"02000")` checks if the DBMS responded by saying 'no tuples found'. Otherwise there was a tuple, containing the value of title, and we print it.

Single-value queries

- Those often involve aggregation.
- How many movies were directed by a given director?

```
int count_movies (char director[20]) {

    EXEC SQL BEGIN DECLARE SECTION;
     char dir[20], SQLSTATE[6];
     int m_count;
    EXEC SQL END DECLARE SECTION;

    for (i=0; i<20; ++i) dir[i]=director[i];

    EXEC SQL SELECT COUNT(DISTINCT Title)
                    INTO :m_count
                    FROM Movies
                    WHERE Director = :dir;
    if (strcmp(SQLSTATE, "00000") != 0)
      {printf("error\n"); m_count = 0};

    return m_count;
}
```

Cursors

- Single-tuple insertions or selections are rare when one deals with DBMSs: SQL is designed to operate with tables.
- However, programming languages operate with variables, not tables.
- Mechanism to connect them: **cursors**.
- Cursor allows a program to access a table, one row at a time.
- A cursor can be declared for a table in the database, or the result of a query.
- Variables from a program can be used in queries for which cursors are declared if they are preceded by a colon.

Operators on cursors

- Cursors first must be declared.
- For a table:

```
EXEC SQL DECLARE C_movies CURSOR FOR Movies;
```

- For a query:

```
EXEC SQL DECLARE C_th CURSOR FOR
   SELECT S.theater
   FROM Schedule S, Movies M
   WHERE S.title = M.title
```

- For a query that depends on a parameter:

```
EXEC SQL DECLARE C_th_dir CURSOR FOR
   SELECT S.theater
   FROM Schedule S, Movies M
   WHERE S.title = M.title and M.director = :dir;
```

Operations on cursors

- Open cursor:

```
EXEC SQL OPEN C_movies;

EXEC SQL OPEN C_th_dir;
```

- The effect of opening a cursor: it points at the first tuple in the table (in this case, either the first tuple of `Movies`), or the first tuple of the result of

```
  EXEC SQL DECLARE C_th_dir CURSOR FOR
     SELECT S.theater
     FROM Schedule S, Movies M
     WHERE S.title = M.title and M.director = :dir;
```

- Close cursor:

```
EXEC SQL CLOSE C_movies;
```

Operations on cursors cont'd

- Fetch – retrieves the value of the current tuple and and assigns fields to variables from the host language.
- Syntax:

```
 EXEC SQL FETCH <cursor> INTO <variables>
```

- Examples:

```
EXEC SQL FETCH  C_th_dir INTO :th;
```

  fetches the current value of theater to which the cursor `C_th_dir` points, puts the value in `th`, and moves the cursor to the next position.

- If there are multiple fields:

```
EXEC SQL FETCH C_Movies INTO :tl, :dir, :act, :length;
```

  Fetches the current `(tl, dir, act, length)` tuple from Movies, and moves to the next tuple.

Operations on cursors cont'd

- Other flavors of `FETCH`:
  - `FETCH NEXT`: move to the next tuple after fetching the values. This is the default, `NEXT` can be omitted.
  - `FETCH PRIOR`: move to the prior tuple after fetching the values.
  - `FETCH FIRST` or `FETCH LAST`: get the first, or the last tuple.
  - `FETCH RELATIVE <number>`: says by how many tuples to move forward (if the number is positive) or backwards (if negative). `RELATIVE 1` is the same `NEXT`, and `RELATIVE -1` is the same `PRIOR`.
  - `FETCH ABSOLUTE <number>`: says which tuple to fetch. `ABSOLUTE 1` is the same `FIRST`, and `ABSOLUTE -1` is the same `LAST`.

Using cursors: Example

Prompt the user for a director, and show the first five theaters playing the movies of that director.

```
void FindTheaters() {

  int i;
  EXEC SQL BEGIN DECLARE SECTION;
     char dir[20], th[20], SQLSTATE[6];
  EXEC SQL END DECLARE SECTION;

/* somewhere here we got the value for dir *?

  EXEC SQL DECLARE C_th_dir CURSOR FOR
     SELECT S.theater
     FROM Schedule S, Movies M
     WHERE S.title = M.title and M.director = :dir;
```

Using cursors: Example

```
  EXEC SQL OPEN C_th_dir;

  i=0;
  while (i < 5) {
     EXEC SQL FETCH C_th_dir into :th;
     if (NO_MORE_TUPLES) break;
     else printf("theater\t%s\n", th);
     ++i;
  }

  EXEC SQL CLOSE C_th_dir;
}
```

What is `NO_MORE_TUPLES`? A common way to define it is:

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

Using cursors for updates

- We consider the following problem. Suppose some lengths of Movies are entered as hours (e.g, 1.5, 2.5), and some as minutes (e.g, 90, 150). We want all to be uniform, say, minutes.
- We assume that no movie is shorter than 5 minutes or longer than 5 hours, so if the length is less than 5, that's an indication that some modification needs to be done.
- Furthermore, we want to delete all movies that run between 4 and 5 hours.

```
void ChangeTime() {
    EXEC SQL BEGIN DECLARE SECTION;
     char tl[20], dir[20], act[20], SQLSTATE[6];
     float length;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE C_Movies CURSOR FOR Movies;
```

Using cursors for updates cont'd

```
    EXEC SQL OPEN C_movies;
    while(1) {
      EXEC SQL FETCH C_Movies INTO
          :tl, :dir, :act, :length;
      if (NO_MORE_TUPLES) break;
      if ( (length > 4 && length < 5) ||
           (length > 240 && length < 300) )
         EXEC SQL DELETE FROM Movies
                  WHERE CURRENT OF C_Movies;
      if (length <= 4)
         EXEC SQL UPDATE Movies
                  SET Length = 60.0 * Length
                  WHERE CURRENT OF C_Movies;
    }
    EXEC SQL CLOSE C_Movies;
}
```

Other embedded SQL statements

- Connecting to a database:

```
strcpy(db_name, "my-database");
EXEC SQL CONNECT TO :db_name;
```

- If user names and passwords are required, use:

```
EXEC SQL CONNECT TO :db_name USER :userid USING :passwd;
```

- Disconnecting:

```
EXEC SQL CONNECT RESET;
```

- Save all changes made by the program:

```
EXEC SQL COMMIT;
```

- Rollback (for unsuccessful termination):

```
EXEC SQL ROLLBACK;
```

Dynamic SQL

- Programs can construct and submit SQL queries at runtime
- Often used for updating databases
- General idea:
  - First, an SQL statement is given as a string, with some placeholders for values not yet known.
  - When those values become known, a query is formed, and
  - when it's time, it gets executed.
- Example: a company that fires employees by departments.
  First, start getting ready:

```
sqldelete = "delete from Empl where dept = ?";
EXEC SQL PREPARE dynamic_delete FROM :sqldelete;
```

Dynamic SQL cont'd

- At some later point, the value of the unlucky department becomes known and put in `bad_dept`. Then one can use:

```
EXEC SQL EXECUTE dynamic_delete USING :bad_dept;
```

- May be executed more than once:

```
EXEC SQL EXECUTE dynamic_delete USING :another_bad_dept;
```

- Immediate execution:

```
SQLstring = "delete from Empl where dept='CEO'";
EXEC SQL EXECUTE IMMEDIATE :SQLstring;
```

Dynamic and Embedded SQL together

- One can declare cursors for prepared queries:

```
my_sql_query = "SELECT COUNT(DISTINCT Title) \
                FROM Movies \
                WHERE Director = ?";
EXEC SQL PREPARE Q1 FROM :my_sql_query;
EXEC SQL DECLARE c1 CURSOR FOR Q1;

 /* get value of dir */

EXEC SQL OPEN c1 USING :dir;
EXEC SQL FETCH c1 INTO :count_movies;
EXEC SQL CLOSE c1;
```

- The same operation can be repeated for different values of `dir`.

More than one user

- So far we assumed that there is only one user. In reality this is not true.
- While a cursor is open on a table, some other user could modify that table. This could lead to problems.
- One way of addressing this: *insensitive cursors*.

```
EXEC SQL DECLARE C1
    INSENSITIVE CURSOR FOR
       SELECT Title, Director
       FROM Movies
```

- This guarantees that if someone modifies `Movies` while `C1` is open, it won't affect the set of fetched tuples.
- This is a very expensive solution and is not used very often.

Problems with more than one user

- We have a bank database, with a table `Account`, one attribute being `balance`.
- The following is a function that transfers money from one account to another. It must enforce the rule that one cannot transfer more money than the balance on the account.

```
    EXEC SQL BEGIN DECLARE SECTION;
     int acct_from, acct_to, balance1, amount;
     char SQLSTATE[6];
    EXEC SQL END DECLARE SECTION;

 void Transfer() {
   /* we ask the user to enter
      acct_from, acct_to, amount */
```

Problems with more than one user

```
EXEC SQL SELECT balance INTO :balance1
         FROM Account
         WHERE account_id = :acct_from;

if (balance1 < amount)
    printf("Insufficient amount in account %d\n", acct_from)
else {
      EXEC SQL UPDATE Account
               SET balance = balance + :amount
               WHERE account_id = :acct_to;
      EXEC SQL UPDATE Account
               SET balance = balance - :amount
               WHERE account_id = :acct_from;
     }
}
```

Problems with more than one user

- Assume that `acct_to` and `acct_from` are joint accounts, with two people being authorized to do transfers.
- Let `acct_from` have \$1000. Suppose both users try to transfer \$1000 from this account.
- Sequence of events:
  - User 1 initiates a transfer. Condition is checked and the first `UPDATE` statement is executed.
  - User 2 initiates a transfer. Condition is checked, and met, since the second `UPDATE` statement from the first transfer hasn't been executed yet. Now both `UPDATE` statements are executed.
  - User 1's transfer operation is finished.
- `acct_from` has balance $-\$1000$, despite an apparent safeguard against a situation like this.

Transactions and atomicity

- Why did this happen?
- Because the operation wasn't executed atomically.
- Transaction: a group of statements that are executed atomically on a database.
- Declaration sections, and connecting to a database, do *not* start a transaction.
- A transaction starts when the first statement accessing a row in a database is executed (e.g., OPEN CURSOR).
- Normally, a transaction ends with the last statement of the program, but one can end it explicitly by either
  `EXEC SQL COMMIT;` or
  `EXEC SQL ROLLBACK;`

Transactions and atomicity cont'd

- We revisit the `Transfer()` function.
- Transaction starts with

```
EXEC SQL SELECT balance INTO :balance1
         FROM Account
         WHERE account_id = :acct_from;
```

- To ensure that the problems with concurrent execution do not occur, one can state

```
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

- The meaning of this will become clear when we study transaction processing.
- Fortunately, this is the default, and the statement is not necessary.

Transactions and atomicity cont'd

- If the test (`balance1 < amount`) is true, we may prefer to abort the transaction:

```
if (balance1 < amount) {
    printf("Insufficient amount in account %d\n", acct_from);
    EXEC SQL ROLLBACK;
 }
```

- If there is sufficient amount of funds, we can put after `UPDATE` statements

```
EXEC SQL COMMIT;
```

to indicate successful completion.

Transactions and isolation

- Isolation means that to the user it must appear as if no other transactions were running. The basic level of isolation is

  ```
  EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
  ```

- There are others, that have to do with dirty reads.
- Dirty read: read data modified by another transaction, but not yet committed.
- One can explicitly specify

  ```
  SET TRANSACTION [READ ONLY | READ WRITE]
     ISOLATION LEVEL READ [COMMITTED | UNCOMMITTED]
  ```

- `READ ONLY` or `READ WRITE` specify whether transaction can write data (`READ WRITE` is the default and can be omitted).
- `COMMITTED` indicates that dirty reads are not allowed (only committed data can be read).

Dealing with errors

- SQL92 standard specifies a structure `SQLCA` (SQL Communication Area) that needs to be declared by
  `EXEC SQL INCLUDE SQLCA;`
- The main parameter is `sqlca.sqlcode`, where 0 indicates success.
- SQL99 eliminates `SQLCA`, and many programs have the following structure:

```
...
EXEC SQL CONNECT TO...
EXEC SQL WHENEVER SQLERROR goto do_rollback;
while (1) {
   /* loop over tuples */
   /* operations that assume successful execution */
   continue;
 do_rollback:
   EXEC SQL ROLLBACK; /* other operations, e.g. printing */
 }
EXEC SQL DISCONNECT CURRENT;
```

SQL injection

- One has to be very careful using embedded/dynamic SQL, especially with user inputs, as users can, maliciously or inadvertently, enter data that will change the meaning of the program.
- Problem with escape characters: program to be executed is defined as a string

  ```
  "SELECT * FROM Students WHERE id =' " + st_id + "';"
  ```

- User enters `st_id` to be the string `' or '1'='1`
- The program becomes

  ```
  "SELECT * FROM Students WHERE id ='' or '1'='1';"
  ```

- With any student id, it gives the list of *all* student ids, clearly a security breach.

SQL injection cont'd

- Typing issues: not checking types correctly. Use a similar program:

```
"SELECT * FROM Students WHERE id =" + st_id + ";"
```

- User enters `st_id` as a string

```
        1; DROP TABLE Students
```

- Result:

```
"SELECT * FROM Students WHERE id =1; DROP TABLE Students;"
```

- Very undesirable effect.
- If you fail a course, don't legally change your name to

```
        1; DROP TABLE Grades
```