Lecture 4 Notes

Functional Dependencies + SQL Functions – Expressions

Discussion Points:

- Informal Design Guidelines for Relation Schemas
- Functional Dependencies
- Inference rules for Functional Dependencies
- SQL: Functions
- SQL: Expressions

❖ Informal Design Guidelines for Relation Schemas

Relational database design ultimately produces a set of relations.The implicit goals of the design activity are: *information preservation and minimum redundancy*.

- **Informal Design Guidelines for Relation Schemas**

Four *informal guidelines* that may be used as *measures todetermine the quality* of relation schema design:

  - Making sure that the semantics of the attributes is clear in theschema
  - Reducing the redundant information in tuples
  - Reducing the NULL values in tuples
  - Disallowing the possibility of generating spurious tuples

- **Imparting Clear Semantics to Attributes in Relations**

The **semantics** of a relation refers to its meaning resulting fromthe interpretation of attribute values in a tuple. The relational schema design should have a clear meaning.

❖ <u>Guideline 1</u>

1. Design a relation schema so that it is easy to explain.
2. Do not combine attributes from multiple entity types and relationship types into a single relation.

- **Redundant Information in Tuples and Update Anomalies**

One goal of schema design is to minimize the storage space usedby the base relations (and hence the corresponding files).
Grouping attributes into relation schemas has a significant effecton storage space
Storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These are:

  - Insertion anomalies
  - Deletion anomalies
  - Modification anomalies

- **Insertion Anomalies**:

  - when insertion of a new tuple is not done properly and will therefore make the database become inconsistent.
  - When the insertion of a new tuple introduces a NULL value (for example a department in which no employee works as of yet). This will violate the integrity constraint of the table since ESSN is a primary key for the table.

- **Deletion Anomalies:**

The problem of deletion anomalies is related to the secondinsertion anomaly situation just discussed.

Example: If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

- **Modification Anomalies:**

Happen if we fail to update all tuples as a result in the change in a single one.
Example: if the manager changes for a department, all employees who work for that department must be updated in all the tables.

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided: hence

❖ <u>Guideline 2</u>

Design the base relation schemas so that no insertion, deletion,or modification anomalies are present in the relations.

If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly. The second guideline is consistent with and, in a way, a restatement of the first guideline.

- **NULL Values in Tuples**

Fat Relations: A relation in which too many attributes are grouped.

If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.
Another problem with NULLs is how to account for them when aggregate operations such as COUNT, or SUM are applied.

SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:

○ The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.

○ The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.

○ The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we may state another guideline.

❖ <u>**Guideline 3**</u>

As much as possible, avoid placing attributes in a base relation whose values may frequently be NULL.
If NULLs are unavoidable, make sure that they apply inexceptional cases only.
For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute Office_number in the EMPLOYEE relation; rather, a relation EMP_OFFICES(Essn, Office_number) can be created.

- **Generation of Spurious Tuples**

Often, we may elect to split a "fat" relation into two relations, with the intention of joining them together if needed. However, applying a NATURAL JOIN may not yield the desired effect. On the contrary, it will generate many more tuples and we cannot recover the original table.

❖ <u>**Guideline 4**</u>

Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated.
Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

Summary and Discussion of Design Guidelines

We proposed informal guidelines for a good relational design.The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

○ Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation

○ Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values

○ Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper(foreign key, primary key) relationship

The strategy for achieving a good design is to decompose a badlydesigned relation appropriately.

❖ Functional Dependencies

The single most important concept in relational schema design theory is that of a functional dependency.

- **Definition of Functional Dependency**

A functional dependency is a constraint between two sets ofattributes from the database. Suppose that our relational database schema has n attributes $A1$, $A2$, ..., An.
If we think of the whole database as being described by a single **universal** relation schema $R = \{A1, A2, ... , An\}$.

A **functional dependency**, denoted by X -> Y, between two setsof attributes X and Y that are subsets of R, *such that* any two tuples $t1$ and $t2$ in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$.

This means that the values of the Y component of a tuple in r depend on, or are *determined by,* the values of the X component;We say that the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component.

We say that there is a functional dependency from X to Y, or that

Y is **functionally dependent** on X.

Functional dependency is represented as **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is calledthe **right-hand side**.

X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X-value, they must necessarily agree on their Y-value.
If a constraint on R states that there cannot be more than one tuple with a given X-value in any relation instance $r(R)$—that is, $X$is a **candidate key** of R— this implies that X->Y for any subset of attributes Y of R.
If X is a candidate key of R, then X->R.

If X->Y in R, this does not imply that Y->X in R.

A functional dependency is a property of the **semantics** or meaning of the attributes.

Whenever the semantics of two sets of attributes in R indicatethat a functional dependency should hold, we **<u>specify the</u>** **<u>dependency as a constraint.</u>**

❖ Inference rules for Functional Dependencies

- *Transitive Rule:* In a relation, if attribute(s) A→B and B→C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C)

  Example: Staff_No→Branch_No and Branch_No→BAddress

  Example:

SSN	Name	School	Location
101	David	Alabama	Tuscaloosa
102	Chrissy	MSU	Starkville
103	Kaitlyn	LSU	Baton Rouge
104	Stephanie	MSU	Starkville
105	Lindsay	Alabama	Tuscaloosa
106	Chloe	Alabama	Tuscaloosa

Here, we will define two FDs:

1.SSN → Name and School → Location.

2. SSN → School.

For, School→ Location, There are only three schools in the example and you may note that for every school, there is only one location, so no FD violation. Now, we want to point out something interesting. If we define a functional dependency X → Y and we define a functional dependency Y → Z, then we know by inference that X → Z.

Here, we defined SSN → School. We also defined School → Location, so we can infer that SSN → Location

although that FD was not originally mentioned. The inference we have illustrated is called the transitivity rule of FD inference. Here is the transitivity rule restated:

Given X → Y

Given Y → Z

Then X → Z

To see that the FD SSN→ Location is true in our data, you can note that given any value of SSN, you always find a unique location for that person.

Another way to demonstrate that the transitivity rule is true is to try to invent a row where it is not true and then see if you violate any of the defined FDs.

We defined these FD's: Given: SSN → Name; SSN → School; School → Location

We are claiming by inference using the transitivity rule that: SSN→ Location

- *Reflexive Rule:* If X is composite, composed of A and B, then X→ A and X→ B.

  Eg: X= Name, City. Then we are saying that X → Name and X → City.

  Example:

Name	City
David	Mobile
Kaitlyn	New Orleans
Chrissy	Baton Rouge

  The rule, which seems quite obvious, says if I give you the combination <Kaitlyn, New Orleans>, what is this person's Name? What is this person's City? While this rule seems obvious enough, it is necessary to derive other functional dependencies.

- *Augmentation Rule:* If X→ Y, then XZ→ YZ.

  You might call this rule, "more information is not really needed, but it doesn't hurt." Suppose we use the same data as before with Names and Cities and define the FD Name → City.

  Now, suppose we add a column, Shoe Size:

Name	City	Shoe Size
David	Mobile	10
Kaitlyn	New Orleans	6
Chrissy	Baton Rouge	3

  Now, I claim that because Name→ City, that

  Name + Shoe Size → City + Shoe Size Or

  Name + Shoe Size → City

  (i.e., we augmented Name with Shoe Size).

  Will there be a contradiction here, ever? No, because we defined Name → City, Name plus more information will always identify the unique City for that individual. We can always add information to the LHS of an FD and still have the FD be true.

- *Decomposition Rule:* **I**f it is given that X → YZ (that is, X defines both Y and Z), then X → Y and X → Z.

  Example:

  Suppose I define Name → City, Shoe Size. This means for every occurrence of Name, I have a unique value of City and a unique value of Shoe Size.

  The rule says that given Name → City and Shoe Size together, then Name → City and Name → Shoe Size. A partial proof using the reflexive rule would be:

  > Name → City, Shoe Size (given)
  >
  > City, Shoe Size → City (by the reflexive rule)
  >
  > Name → City (using steps 1 and 2 and the transitivity rule)

- *Union Rule:* The union rule is the reverse of the decomposition rule in that if

  X → Y and X → Z, then X → YZ.

  The same example of Name, City, and Shoe Size illustrates the rule. If we found independently or were given that Name →City and Name → Show Size, we can immediately write Name→ City, Shoe Size.

  You might be a little troubled with this example in that you may say that Name is not a reliable way of identifying City; Names might not be unique. You are correct in that Names may not ordinarily be unique but note the language we are using. In this database, we define that Name → City and hence, in this database are restricting Name to be unique by definition.

- *Psuedotransitivity:* If X → Y and WY → Z, then WX → Z.

- *Full Dependency* In a relation, the attribute(s) B is fully functional dependent on A if B is functionally dependent on A, but not on any proper subset of A.

- *Partial Dependency* A type of functional dependency where an attribute is functionally dependent on only part of the primary key (primary key must be a composite key).

  Eg: SalesOrderNo, ItemNo, Qty, UnitPrice

- **Keys and FDs**
  - The main reason we identify the FDs and inference rules is to be able to find keys and develop normal forms for relational databases.
  - In any relational table, we want to find out which, if any attribute(s), will identify the rest of the attributes. An attribute that will identify all the other attributes in row is called a "candidate key." A key means a 'unique identifier' for a row of inform

- Hence, if an attribute or some combination of attributes will always identify all the other attributes in a row, it is a "candidate" to be "named" a key.
- Keys should be a minimal set of attributes whose closure is all the attributes in the relation — "minimal" in the sense that you want the fewest attributes on the LHS of the FD that you choose as a key.
- In our example, SSN will be minimal (one attribute), whose closure includes all the other attributes.
- Once we have found a set of candidate keys (or perhaps only one as in this case), we designate one of the candidate keys as the primary key and move on to normal forms.

Example:

SSN	Name	School	Location
101	David	Alabama	Tuscaloosa
102	Chrissy	MSU	Starkville
103	Kaitlyn	LSU	Baton Rouge
104	Stephanie	MSU	Starkville
105	Lindsay	Alabama	Tuscaloosa
106	Chloe	Alabama	Tuscaloosa

Suppose the following fFDs exist:

SSN → Name

SSN → School

School → Location

Solution:

- SSN → Name (given)
- SSN → School (given)
- SSN → Location (derived by the transitive rule)
- SSN → SSN (reflexive rule (obvious))
- SSN → SSN, Name, School, Location (union rule)

So, SSN can be a candidate key and primary key as well.

- **Legal Relation States:**

Relation extensions $r(R)$ that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**)of R.
Functional dependencies are used to describe further a relationschema R by specifying constraints on its attributes that must hold *at all times.*

Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning.
For example, {State, Driver_license_number} -> SSN should hold for any adult in the United States and hence should holdwhenever these attributes appear in a relation.

Consider the relation schema EMP_PROJ from the semantics ofthe attributes and the relation, we know that the following functional dependencies should hold:

a. SSN -> Ename
b. Pnumber -> {Pname, Plocation}
c. {SSN, Pnumber} -> Hours

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r*but must be defined explicitly by someone who knows the semantics of the attributes of *R*.

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Example:

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

The following FDs *may hold* because the four tuples in the current extension have no violation of these constraints:

B -> *C*

C -> *B*

{*A*, *B*} -> *C*

{*A*, *B*} -> *D*

{*C*, *D*} -> *B*

However, the following *do not* hold because we already have violations of them in the given extension*:*

A -> *B* (tuples 1 and 2 violate this constraint)

B -> *A* (tuples 2 and 3 violate this constraint)

D -> *C* (tuples 3 and 4 violate it)

❖ SQL: Conversion Functions

- *Data Type Conversions*
  - Implicit data type conversion
  - Explicit data type conversion

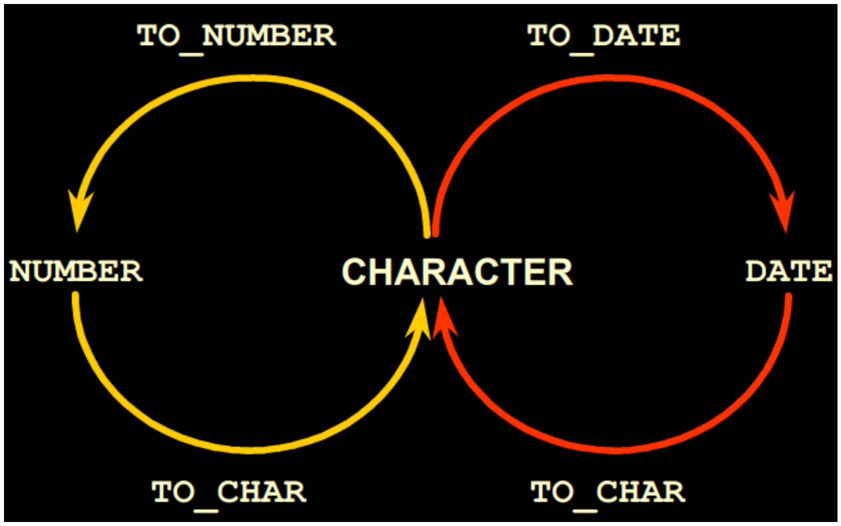
- *Implicit Data Type Conversion*
  - For assignments, the Oracle server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

  - For expression evaluation, the Oracle Server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

- *Explicit Data Type Conversion*



  - *TO_CHAR FUNCTION*

TO_CHAR(date, 'format_model')

The format model:

- Must be enclosed in single quotation marks and is case sensitive
- Can include any valid date format element
- Has an fm element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

  - Elements of Data Format Model

YYYY	Full year in numbers
YEAR	Year spelled out
MM	Two-digit value for month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

- Time elements format the time portion of the date.

HH24:MI:SS AM	15:45:32 PM

- Add character strings by enclosing them in double quotation marks.

DD "of" MONTH	12 of OCTOBER

- Number suffixes spell out numbers.

ddspth	fourteenth

SELECT last_name, TO_CHAR(hire_date, 'fmDD Month YYYY')
AS HIREDATE
FROM employees;

LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

- Using the TO_CHAR Function with Numbers
  These are some of the format elements you can use with the TO_CHAR function to display a number value as a character:

9	Represents a number
0	Forces a zero to be displayed
$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Prints a thousand indicator

SELECT TO_CHAR(salary, '$99,999.00') SALARY
FROM employees
WHERE last_name = 'Ernst';

- Using the TO_NUMBER and TO_DATE Functions
  - Convert a character string to a number format using the TO_NUMBER function:

    TO_NUMBER(*char*[, '*format_model*'])

  - Convert a character string to a date format using the TO_DATE function:

    TO_DATE(*char*[, '*format_model*'])

  - These functions have an fx modifier. This modifier specifies the exact matching for the character argument and date format model of a TO_DATE function

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM employees
WHERE hire_date < TO_DATE('01-Jan-90', 'DD-Mon-RR');
```

LAST_NAME	TO_CHAR(HIR
King	17-Jun-1987
Kochhar	21-Sep-1989
Whalen	17-Sep-1987

❖ SQL: General Functions

- These functions work with any data type and pertain to using nulls.
  - NVL (expr1, expr2)
  - NVL2 (expr1, expr2, expr3)
  - NULLIF (expr1, expr2)
  - COALESCE (expr1, expr2, ..., exprn)

- *NVL Function*
  - Converts a null to an actual value.
  - Data types that can be used are date, character, and number.
  - Data types must match:
    - NVL(commission_pct,0)
    - NVL(hire_date,'01-JAN-97')
    - NVL(job_id,'No Job Yet')

  - *Using NVL Function*

```
SELECT last_name, salary, NVL(commission_pct, 0),
(salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL
FROM employees;
```

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000

- *NVL2 Function*
  - If first expression is not null, return second expression. If first expression is null, return third expression. the first expression can have any data type.
  - *Using NVL2 Function*

    SELECT last_name, salary, commission_pct,
    NVL2(commission_pct, 'SAL+COMM', 'SAL') income
    FROM employees WHERE department_id IN (50, 80);

LAST_NAME	SALARY	COMMISSION_PCT	INCOME
Zlotkey	10500	.2	SAL+COMM
Abel	11000	.3	SAL+COMM
Taylor	8600	.2	SAL+COMM
Mourgos	5800		SAL
Rajs	3500		SAL
Davies	3100		SAL
Matos	2600		SAL
Vargas	2500		SAL

- *NULLIF Function*
  - Compares two expressions and returns null if they are equal, returns the first expression if they are not equal.
  - *Using NULLIF Function*

    SELECT first_name, LENGTH(first_name) "expr1",
    last_name, LENGTH(last_name) "expr2",
    NULLIF(LENGTH(first_name), LENGTH(last_name)) result
    FROM employees;

FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
Steven	6	King	4	6
Neena	5	Kochhar	7	5
Lex	3	De Haan	7	3
Alexander	9	Hunold	6	9
Bruce	5	Ernst	5	
Diana	5	Lorentz	7	5
Kevin	5	Mourgos	7	5
Trenna	6	Rajs	4	6
Curtis	6	Davies	6	

- *COALESCE Function*
  - Return first not null expression in the expression list.
  - *Using COALESCE Function*
    - The advantage of the COALESCE function over the NVL function is that the COALESCE function can take multiple alternate values.
    - If the first expression is not null, it returns that expression; otherwise, it does a COALESCE of the remaining expressions.

```
SELECT last_name,
COALESCE(commission_pct, salary, 10) comm
FROM employees
ORDER BY commission_pct;
```

❖ SQL: Conditional Expressions

- Provide the use of IF-THEN-ELSE logic within a SQL statement
- Use two methods:
  - CASE expression
  - DECODE function

- *The CASE Expression*
  - Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
CASE expr WHEN comparison_expr1 THEN return_expr1
     [WHEN comparison_expr2 THEN return_expr2
     WHEN comparison_exprn THEN return_exprn
     ELSE else_expr]
 END
```

  - *Using the CASE Expression*

```
SELECT last_name, job_id, salary,
     CASE job_id
            WHEN 'IT_PROG' THEN 1.10*salary
            WHEN 'ST_CLERK' THEN 1.15*salary
            WHEN 'SA_REP' THEN 1.20*salary
            ELSE salary
     END "REVISED_SALARY"
FROM employees;
```

- *The DECODE Function*
    - Facilitates conditional inquiries by doing the work of a CASE or IF-THEN-ELSE statement:

```
DECODE(col|expression, search1, result1
                [, search2, result2,...,]
                [, default])
```

    - *Using DECODE Expression*

```
SELECT last_name, job_id, salary,
DECODE(job_id, 'IT_PROG', 1.10*salary,
               'ST_CLERK', 1.15*salary,
               'SA_REP', 1.20*salary,
               salary) REVISED_SALARY
FROM employees;

SELECT last_name, salary,
DECODE (TRUNC(salary/2000, 0),
                         0, 0.00,
                         1, 0.09,
                         2, 0.20,
                         3, 0.30,
                         4, 0.40,
                         5, 0.42,
                         6, 0.44,
                         0.45) TAX_RATE
FROM employees
WHERE department_id = 80;
```

❖ References

- “Database System Concepts”, Avi Silberschatz, Henry F. Korth, S. Sudarshan, McGraw-Hill.
- “Database Management Systems”, Raghu Ramakrishnan, Johannes Gehrke, McGraw-Hill.
- “Fundamentals of Database Systems”, R. Elmasri, S. B. Navathe, Pearson.
- Oracle SQL Resources
- Other Internet Sources