CSC 355 Database Systems
Lecture 1

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topics:

- Course Organization
- Introduction to Databases
- SQLDeveloper

Contact Information

- Course web site: http://d2l.depaul.edu/
  - Weekly discussion forum for questions/comments
- Office hours: Monday and Wednesday 1:00pm-2:00pm, and Wednesday 3:00pm-4:00pm
  - Via Zoom at https://depaul.zoom.us/my/eschwabe
- Email: eschwabe@depaul.edu
  - Please begin subject line with “CSC 355” and sign your name
  - Expect reply within one business day

Course Texts

- Required text:
  A First Course in Database Systems (third edition),
  Ullman and Widom
  (ISBN 978-0136006374)
- Additional reference (optional):
  Murach's Oracle SQL and PL/SQL for Developers (second edition), Murach (ISBN 978-1890774806)

Course Policies

- Grading: 30% homework, 30% midterm exam, 30% final exam, 10% quizzes
  - HWs accepted late (up to 24 hours only) with penalty
  - Lowest HW score will be dropped
  - Submit HWs through d2l, no emailed submissions
  - Exam will be given through d2l, details TBA
- Course prerequisite: CSC 301 or CSC 393
- University policies: See posted syllabus
- Weekly schedule: See posted document

Academic Integrity Policy

- http://academicintegrity.depaul.edu/
  - Cheating: Any action that violates university norms or instructor guidelines for course work
  - Plagiarism: Any use of another's work without proper citation where original work is expected
  - Complicity: Any action that facilitates an academic integrity violation

Databases are Everywhere

- Amazon (or any online store…)
- Southwest (or any airline…)
- Chase (or any bank…)
- Campusconnect (or any university system…)
- …and those are just a few…you are interacting with databases every day…

What is a database?

- *Data* is information that can be recorded and has a known meaning
- A *database* is an organized collection of logically related data that are typically…
  - Persistent: are stored on a stable medium
  - Shared: have multiple uses and interested users
  - Interrelated: form a bigger picture

Why Use a Database System?

- Early data processing systems used files of data in plain text form
- Problem: program-data dependence led to
  - limited data sharing
  - duplication of data
  - increased time for development and maintenance

Why Use a Database System?

- A database uses a single repository of data accessed by multiple users
  - Contains information on the structure of the data
  - Allows sharing of and concurrent access to data
  - Supports different views of the data
- The costs are higher overhead for the design, implementation, and maintenance of the data
- What are the benefits?

Benefits of Database Systems

- Program-data independence
- Controlled data redundancy
- Controlled access to data
- Support for multiple user interfaces
- More efficient query processing
- Faster application development

Database Management Systems

- A *database management system (DBMS)* is a collection of software components that lets you
  - create (e.g., define, construct)
  - maintain (e.g., modify, keep available)
  - control access to (e.g., secure, allow queries to)

  a database

Database Management Systems

- DBMS Examples: Oracle, IBM DB2, MS Access/SQL Server, MySQL
- We can work with a DBMS directly or through an application that supplies a particular interface (e.g., SQLDeveloper)
- The database and DBMS together make up a *database system*

Database Models

- Older Models:
  - File Systems, Hierarchical, Network
  - All had drawbacks…
- The Relational Model
- Newer Models:
  - Semi-structured, Object-relational, NoSQL
  - …not as popular as Relational Model

File Systems

- Data stored in simple text files, each one possibly having a different fixed organization of its data
- High level of program-data dependence
- Difficult to share data
- Not practical to optimize queries

Hierarchical/Network Models

- Hierarchical Model: Data arranged in "parent-child" relationships
- Network Model: Can represent more general relationships among types of data
- Both models have similar weaknesses:
  - Applications must navigate relationships explicitly
  - DBMS can not rearrange data to optimize queries

Relational Model

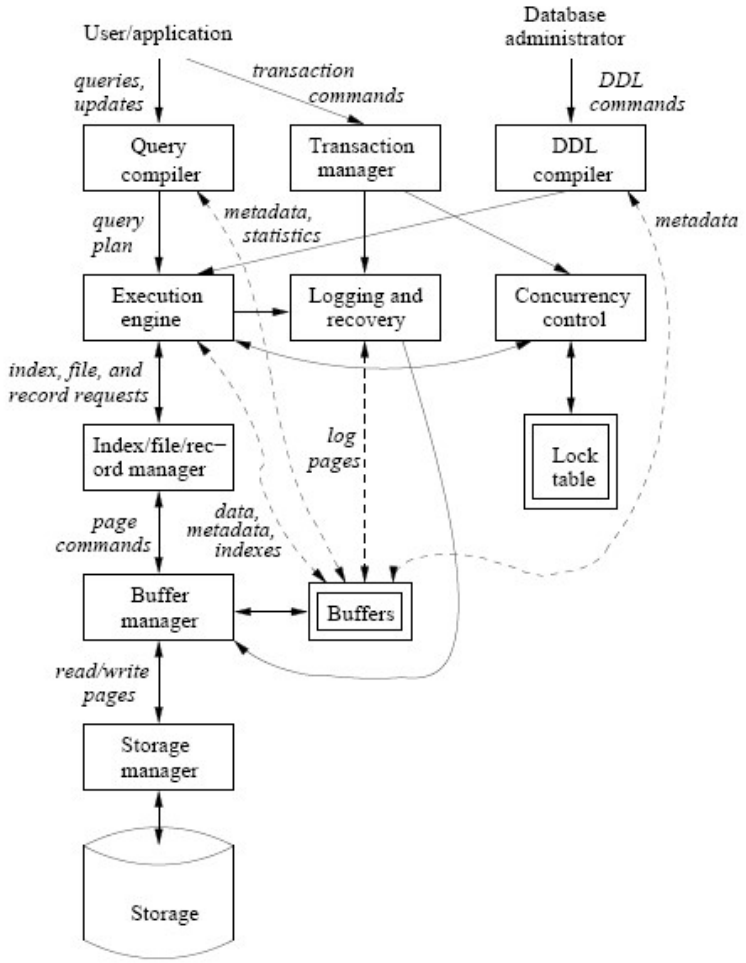
- First model to separate the logical structure of the database from its physical implementation
- Data are divided into two-dimensional tables called relations
- Tables are linked by shared columns of data
- Rules exist for dividing data among tables
- A standardized query language exists (SQL)

Newer Models

- Semi-structured databases: Store collections of data in XML files
- Object-relational databases: Add support for structured data types to relational databases
- Document databases: Have a less restrictive structure, typically without a fixed schema
- Data warehouses: Integrate multiple sources of data, possibly from different models

Components of a DBMS

(From Ullman/Widom)



User Interactions with DBMS

- Database Definition: Create database schema, links between tables, constraints
- Query Processing: Request retrieval or modification of data (“queries”/“actions”)
- Transaction Processing: Execute sets of operations that must be executed as a unit (“transactions”)

Approximate Course Schedule

- Week 1: Introduction and Relational Model
- Weeks 2-5: SQL DDL, Queries, Transactions
- Weeks 6-7: Relational Database Design
- Weeks 8-9: Constraints and Triggers, Database Programming, Views
- Week 10: Slack Time / Course Review

SQLDeveloper

- SQLDeveloper is an application that works as a “front-end” connection to a server running an Oracle DBMS (e.g., Oracle 12c)
- SQL commands can be run individually, or collected in script files.
- Can be downloaded free from Oracle

Setting Up a Connection

- To set up a new connection to acadoradbprd01:
  - Connection Name: *YOURNAME355*
  - Username: your campusconnect username
  - Password: cdm######## (initially uses your 7-digit Student ID)
  - Hostname: acadoradbprd01.dpu.depaul.edu
  - Port: 1521
  - SID: ACADPRD0
  - Test, then Connect…
- Double-click to Open an existing connection
- Disconnect (and commit) when you're done!

Running SQL Commands

- Single SQL command:
  - Type command, then Execute (Ctrl-Enter)
    - e.g, to change password, ALTER USER *username* IDENTIFIED BY *newpassword* ;
- Script (SQL commands stored in a file):
  - Type @ followed by full path to script file, then Run Script (F5)
- Output will appear in bottom window under Query Result or Script Output

Browsing Database Tables

- Left window shows current Tables, click on + to expand list
- Right-click on Tables and choose Refresh to see changes (can also Commit changes)
- Click on a table to view it in the center window (may need to Refresh view also)
  - COLUMNS shows schema
  - DATA shows contents

Saving SQLDeveloper Output

- Three ways:
  - Click on Save icon to save contents of Script Output window to a file
  - Highlight and then copy and paste contents of Script Output window to a file
  - Take and save screenshot of SQLDeveloper display

Next:

- The Relational Model

CSC 355 Database Systems
Lecture 2

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topics:

- The Relational Model

Data Models

- A *data model* describes three things about stored data:
  - Structure of the data: How is the data organized?
  - Operations on the data: What can be done with the data?
  - Constraints on the data: How are the data restricted?

Relational vs. Semi-structured

- From Ullman/Widom:

title	*year*	*length*	*genre*
Gone With the Wind	1939	231	drama
Star Wars	1977	124	sciFi
Wayne's World	1992	95	comedy

Figure 2.1: An example relation

```
<Movies>
    <Movie title="Gone With the Wind">
        <Year>1939</Year>
        <Length>231</Length>
        <Genre>drama</Genre>
    </Movie>
    <Movie title="Star Wars">
        <Year>1977</Year>
        <Length>124</Length>
        <Genre>sciFi</Genre>
    </Movie>
    <Movie title="Wayne's World">
        <Year>1992</Year>
        <Length>95</Length>
        <Genre>comedy</Genre>
    </Movie>
</Movies>
```

Figure 2.2: Movie data as XML

Relational vs. Semi-structured

- Relational Model:
  - Structure: Two-dimensional tables with links
  - Operations: Relational algebra / SQL
  - Constraints: Data domains, uniqueness,…
- Semi-structured Model:
  - Structure: Nested XML elements
  - Operations: Element traversal, searching
  - Constraints: Data domains, nesting restrictions,…

The Relational Model

- Introduced by E.F.Codd in 1970
- First model that separated the logical organization of the data from its physical implementation
- Model is based on formal logic and the relational algebra

The Relational Model

- Data is stored in two-dimensional tables called *relations*, each one having a name
- Each row is a *tuple* representing one instance of the entity the table represents
- Each column is an *attribute*, representing a property for which each instance has a value
- Every *component* in a tuple must have a value taken from its attribute's associated *domain*

Relation Example

- Name: EMPLOYEE
- Tuples: { (100, Margaret Simpson, Marketing, 48,000) ,
  (140, Allen Beeton, Accounting, 52,000.00) ,
  (110, Chris Lucero, Info Systems, 43,000.00) ,
  …and three more… }
- Attributes: ID, Name, Dept, Salary
- Domains: (Three-digit) integer, string (of length at most 20), string (of length at most 12), decimal number (with at most five digits to the left of the decimal point and two to the right)
  - Note that domains can be a little complicated to describe...

Properties of Relations

1. Each relation has a unique name (in database)
2. Each attribute has a unique name (in relation)
3. Each entry of a relation contains a single value from its attribute's domain (or NULL)
4. The order of the records does not matter
5. The order of the attributes does not matter
6. No two records in a relation are identical

Relation Schema vs. Instance

- The *schema* of a relation consists of the name of the relation followed by a list of its attributes (domains may be included also…)
  - EMPLOYEE (ID , Name , Dept , Salary) …
  - EMPLOYEE (ID:number , Name:string , Dept:string , Salary:number) … ?
  - EMPLOYEE (ID:integer(3) , Name:string(20) , Dept:string(12) , Salary:number(5.2)) … ?

Relation Schema vs. Instance

- An *instance* of a relation is a set of tuples, where each tuple contains a value for each attribute (from the associated domain), or perhaps NULL (indicating no value)
  - DBMS enforces these "domain constraints"
- Instances are often presented as tables rather than as sets, but the order of the rows in these tables is not significant

Candidate Keys

- A *candidate key* is a set of attributes for which each tuple in the relation must have a unique set of values ("key constraints"), and for which no subset of the set has this property
  - The book calls these just *keys*, but I will use the more specific term *candidate keys* since there are different types of keys…
- This property must hold for <u>all possible relation instances</u> for it to be a candidate key

Primary Keys

- One of the candidate keys can be chosen as the *primary key* for the relation
- A relation may have many candidate keys, but can have only one primary key
- The primary key will be underlined in the schema
  - A primary key must have a unique set of values in each tuple, and may not contain any NULL values in any tuple ("entity integrity")

Foreign Keys

- We link two relations using a shared key that is the primary key in one of the relations
- In the other relation, this key is called a *foreign key* (dotted underline in schema, with arrow to corresponding primary key)
  - Every value of the foreign key must be the value of the corresponding primary key in some tuple
  - Thus the foreign key associates each tuple with exactly one tuple in the relation that it references

Referential Integrity

- Every foreign key value must appear as the value of the primary key in some row of the table it references
- This restricts the changes that can be made:
  - We can add a row containing a foreign key only if the value of the foreign key appears among the values of the referenced primary key
  - We can remove a row containing a primary key only if the value of the primary key does not appear among the values of any referencing foreign key

Constraints

- All of these will be maintained by the DBMS:
  - Domain constraints: In every tuple, the value of each attribute must come from its specified domain
  - Key constraints: Each tuple must have a unique set of values in each of its candidate keys
  - Entity integrity: Each tuple must have a unique set of values in its primary key, and not any NULLs
  - Referential integrity: Every foreign key value must appear as the value of the primary key in some tuple of the relation it references

Database Schema vs. Instance

- The *schema* of a relational database consists of:
  - The schema of each relation in the database
    - Name, list of attributes (and maybe domains)
    - Primary keys and foreign keys underlined
  - An arrow from each foreign key to the primary key it references

Database Schema vs. Instance

- An *instance* of a relational database consists of:
  - An instance of each relation in the database, where each relation instance satisfies all the required constraints:
    - domain constraints, key constraints, entity integrity, referential integrity (and any user-defined constraints)
  - The only changes allowed by the DBMS are those that result in another valid instance…

Next:

- SQL Data Definition Language

CSC 355 Database Systems
Lecture 3

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topics:

- Quick review of Relational Model
- SQL Data Definition Language (DDL)

The Relational Model

- Relations, tuples, attributes, domains
- Relation schema vs. relation instance
- Candidate key, primary key, foreign key
- Database schema vs. database instance
- Domain constraints, key constraints, entity integrity, referential integrity

Writing an SQL Script

- Create file *scriptname*.sql in text editor
- End every SQL statement with a semicolon
- Use SELECT * FROM *TABLENAME* ; statement to display entire contents of a table
- To add comments:
  - -- to begin a one-line comment
  - /* ... */ to begin and end a multi-line comment

Creating a Table

- CREATE TABLE *TABLENAME*
  (*Attribute1 DOMAIN1,*
  Attribute2 DOMAIN2,
  ...
  Attributek DOMAINk);
- Each attribute-domain pair is followed by a comma
- Domain constraints will be enforced

Oracle SQL Domains

- Numerical domains
- String domains
- Dates

Numerical Domains

- General numbers: NUMBER(*x*,*y*)
  - A fixed-precision number with *x* total digits, and *y* digits to the right of the decimal point
    - 101 is NUMBER(3,0)
    - 999.99 is NUMBER(5,2), not NUMBER(3,2)!
- Can also use NUMERIC(*x*,*y*) or DECIMAL(*x*,*y*)
  - Oracle will convert internally to NUMBER(*x*,*y*)

Numerical Domains

- Whole numbers: INTEGER or INT
  - Oracle converts to NUMBER(38,0)
  - To limit size, use NUMBER(*x*,0) or NUMBER(*x*)
- Floating point numbers: FLOAT
  - Can use REAL, Oracle will convert to FLOAT
  - To limit precision, use NUMBER(*x*,*y*)

String Domains

- Fixed-length strings:
  - CHAR(*n*): A fixed-length string of *n* characters
  - Use when you know exact length of strings
- Variable-length strings:
  - VARCHAR(*m*) or VARCHAR2(*m*): A variable-length string of up to *m* characters
  - Oracle will convert internally to VARCHAR2(*m*)
  - Use when you know maximum length of strings

Dates

- DATE:
  - Value given by keyword DATE followed a string in ‘yyyy-mm-dd’ form
    - yyyy = year, mm = month (number), dd = day
    - Always use this general format in your scripts, but some SQL versions may accept other formats too
  - Oracle will convert a string in ‘dd-mon-yyyy’ form to a DATE object
    - dd = day, mon = month (name), yyyy = year

Dropping a Table

- To drop a table:
  DROP TABLE *TABLENAME* ;

- Cannot drop a table if there is a foreign key in another table that references its primary key
  - (…unless you add CASCADE CONSTRAINTS to the command, which will drop the table and remove any constraints that reference it…)

Populating a Table

- To insert a record into a table:

  INSERT INTO *TABLENAME*
  VALUES (*value1, value2, value3, ...*);
- Values of attributes must be given in the same order as in the schema
- Will generate an error if any constraints are violated (domain constraints, key constraints, entity integrity, referential integrity)

Populating a Table

- To insert a record that specifies only some of the attributes:

  INSERT INTO *TABLENAME* (*Attr1, Attr2,...*)
  VALUES (*value1, value2, ...*);

- Missing attributes will be filled in with NULL (unless default values are specified…)

Defaults and Attribute Constraints

- After attribute and domain, before comma:
  - Add default value for the attribute with DEFAULT *value*
  - Disallow NULL values with NOT NULL
  - Impose other constraints with CHECK (*condition*)
    - e.g., to require that attribute is within a range, use CHECK (*value1 <= Attribute AND Attribute <= value2*)
    - Verified whenever a tuple is added or changed

Defining Keys

- For candidate keys, primary keys, and foreign keys:
  - Can add to a single attribute after its domain
  - Can add as a separate CONSTRAINT clause in the CREATE statement when multiple attributes are involved
    - Like attributes, must be followed by commas
    - Constraint can be named by placing CONSTRAINT *Name* in front of it

Defining Candidate Keys

- Use UNIQUE keyword
- Within a single attribute:
  Attribute DOMAIN UNIQUE
- As a separate constraint:
  UNIQUE (*Attribute1*, *Attribute2*, …)
- Key constraints will be enforced

Defining Primary Keys

- Use PRIMARY KEY keywords
- Within a single attribute:
  Attribute DOMAIN PRIMARY KEY
- As a separate constraint:
  PRIMARY KEY (*Attribute1*, *Attribute2*, ...)
- Key constraints and entity integrity will be enforced

Defining Foreign Keys

- Use FOREIGN KEY keywords
- Within a single attribute:

  Attribute DOMAIN
  REFERENCES *TABLE* (*Attribute*)
- As a separate constraint:

  FOREIGN KEY (*Attr1*, *Attr2*, …)
  REFERENCES *TABLE* (*Attr1*, *Attr2*, …)
- Referential integrity will be enforced

Modifying a Schema

- To add or remove attributes and/or constraints:
  ALTER TABLE *TABLENAME*
    - …ADD *Attribute* DOMAIN;
    - …DROP COLUMN *Attribute*;
    - …ADD CONSTRAINT *Name CONSTRAINT* ...;
    - …DROP CONSTRAINT *Name*;

- Constraints must have names to be dropped

Updating Rows

- To modify existing rows in a table:

  UPDATE *TABLENAME*
  SET *Attribute = expression*
  WHERE *condition;*

- Sets *Attribute* to *expression* in exactly those rows that satisfy *condition*

Removing Rows

- To remove existing rows from a table:

  DELETE FROM *TABLENAME*
  WHERE *condition*;

- Removes from the table exactly those rows that satisfy *condition*

Displaying Table Contents

SELECT * FROM *TABLENAME* ;

- This statement will display the entire contents of *TABLENAME* (all rows and columns)
- This is an example of a very simple *query*
- Adding a WHERE clause would let us display only a subset of the rows…

Next:

- Basic SQL Queries

CSC 355 Database Systems
Lecture 4

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topic:

- Basic SQL Queries

Displaying a Table's Contents

SELECT * FROM *TABLE* ;

- This is an example of the simplest form of a *query* – retrieval of information from one or more tables
  - SELECT * : "Show all attributes…"
  - FROM *TABLE* : "…from the indicated table"

SQL Queries

- General form of a query:

  SELECT *list of expressions*
  FROM *set of rows*
  [WHERE *condition on rows*]
  [GROUP BY *grouping attributes*]
  [HAVING *condition on groups*]
  [ORDER BY *ordering attributes*] ;

- Result is an ordered set of ordered tuples

FROM

… FROM *set of rows*…

- FROM indicates the set of rows from which information will be retrieved
  - a single table (that's all we'll use for now…)
  - a list or other combination ("join") of tables
  - the result of a "subquery"

SELECT

SELECT *list of expressions …*

- SELECT indicates what information will be displayed
  - values of attributes
  - expressions computed from attributes
  - functions applied to attributes

SELECT

- Displaying attributes:
  - * lists all attributes
  - separate attributes in the list with commas
  - attributes can be renamed in result using AS
  - DISTINCT will remove duplicate rows
- Displaying expressions:
  - Can combine attributes with +, -, *, /, ||

SELECT

- Displaying functions of attributes:
  - Numbers: mod(*a*,*b*), power(*m*,*n*), round(*x*,*i*)
  - Strings: upper(*s*), lower(*s*), substr(*s*,*p*,*l*)
  - Dates: sysdate, to_char(*d*, *field*) with field being, e.g., ‘YYYY’, ‘YY’, ‘YEAR’, ‘MM’, ‘MON’, ‘DD’, ‘DY’… or a combination …
- SQL has many built-in functions…

WHERE

… WHERE *condition* …

- Each row is tested against the condition, and only those that satisfy it are returned by the query
- Condition expression can contain:
  - comparisons
  - expressions with wildcards (for strings)
  - logical operators

Comparisons

- Put numerical or string or date value on each side, comparison returns true or false

=	is equal to
!= or <>	is not equal to
>	is greater than
>=	is greater than or equal to
<	is less than
<=	is less than or equal to

Comparisons

- Numbers and dates are compared in the usual way (smaller < larger, earlier < later)
- String values are compared according to *lexicographic (dictionary)* order
  - Compare strings character by character until they differ
  - The string with the earlier character (by ASCII order) where they first differ is smaller

Wildcards

- Using LIKE, we can compare character strings to strings that include wildcard characters that match anything:

_	matches any single character
%	matches any consecutive set of characters

- For example:
  - ‘b_d’ will match ‘bad’, ‘bed’, but not ‘band’
  - ‘bat%’ will match ‘bat’, ‘bath’, ‘battery’…

Logical Operators

- Simple conditions can be combined into more complicated conditions
  - *X* AND *Y* is satisfied by a tuple if and only if both *X* and *Y* are satisfied by it
  - *X* OR *Y* is satisfied by a tuple if and only if at least one of *X* and *Y* is satisfied by it
  - NOT *X* is satisfied by a tuple if and only if *X* is not satisfied by it

Dealing With NULLs

- Any arithmetic expression involving a NULL will yield NULL (as will most functions)
- To replace NULLs in output, use the function NVL(*expr1*, *expr2*)
  - If *expr1* is not NULL, will display *expr1*
  - If *expr1* is NULL, will display *expr2* instead
  - e.g., SELECT NVL(phone, ‘no phone given’) …

Dealing With NULLs

- Any comparison involving a NULL will yield UNKNOWN
  - Use IS NULL (not =) to check if a value is NULL
  - There are extended definitions of AND, OR, and NOT that include UNKNOWN
  - UNKNOWN will not satisfy a WHERE test
  - UNKNOWN will satisfy a CHECK condition

ORDER BY

… ORDER BY *list of ordering attributes*

- Tuples are sorted by the first attribute in the list
  - Ascending order is the default, DESC after attribute indicates descending order instead
- Ties are broken by the second attribute (if any), then the third attribute (if any), et cetera

Writing a Query

1. FROM: What table should I use?
2. WHERE: How do I indicate which rows to include in the result?
3. ORDER BY: How should I sort the rows in the output?
4. SELECT: What values do I have to compute and display?

Solving a Query Problem

1. Before you write the query:
   - Read the problem carefully to be sure you understand it, and clarify where necessary
   - Look at the data and work it out by hand, then think about how you did it
2. Write the query:

   First FROM (with SELECT *), then WHERE, then ORDER BY, then SELECT

Solving a Query Problem

3. Test the query:
   - If there are syntax errors, go back to 2. to correct them
   - Look at the result against what you did by hand
   - If the result is not correct, go back to 2. and re-examine the query against your result and your interpretation of the problem – describe as clearly as you can precisely how it is incorrect

Next:

- More SQL Queries

CSC 355 Database Systems
Lecture 5

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topic:

- More SQL Queries
  - Solving query problems
  - Aggregate functions
  - GROUP BY, HAVING

SQL Queries

- General form of a query:

  SELECT *list of expressions*
  FROM *set of rows*
  [WHERE *condition on rows*]
  [GROUP BY *grouping attributes*]
  [HAVING *condition on groups*]
  [ORDER BY *ordering attributes*] ;

- Result is an ordered set of ordered tuples

SELECT, FROM

SELECT *list of expressions* …

- Indicates what information will be displayed
  - values of attributes, expressions, functions

… FROM *set of rows* …

- Indicates the set of rows from which information will be retrieved
  - a single table (for now…)

WHERE, ORDER BY

… WHERE *condition* …

- Only displays rows where condition is true
  - comparisons, wildcards, logical operators

… ORDER BY *ordering attributes*

- Tuples are sorted by first attribute in the list, ties broken by second, third, et cetera…
  - ascending order by default, DESC for descending

Solving a Query Problem

1. Before you write the query:
   - Read the problem carefully to be sure you understand it, and clarify where necessary
   - Look at the data and work it out by hand, then think about how you did it
2. Write the query:

   First FROM (with SELECT *), then WHERE, then ORDER BY, then SELECT

Solving a Query Problem

3. Test the query:
   - If there are syntax errors, go back to 2. to correct them
   - Look at the result against what you did by hand
   - If the result is not correct, go back to 2. and re-examine the query against your result and your interpretation of the problem – describe as clearly as you can precisely how it is incorrect

Query Problems

- Give the names of all undergraduate degree programs
- Give an alphabetical list of all students who started more than eight years ago
- List all information for students in Computer Science, Computer Gaming, and Information Systems, ordered by program name
- Give a sorted list of the IDs of all graduate students not in the PhD program

Query Problems

- List the IDs of all students who enrolled in a course in 2013
- Give an alphabetical list of last names of all students who do not have a Social Security number listed
- List all information for students who are from Springfield and who started between 2011 and 2013
- List the number of students in each degree program
  (no query, just work out this answer by hand…)

Aggregate Functions

- Given an attribute, an aggregate function takes the values of that attribute in the set of returned rows and computes a single value from them
  - COUNT(…): Number of non-NULL values
  - SUM(…): Sum of the values
  - AVG(…): Average of the values
  - MIN(…): Smallest of the values
  - MAX(…): Largest of the values

GROUP BY

… GROUP BY *grouping attributes* …

- Combines the rows into sets based on the value(s) of some attribute(s)
  - Can only display the value(s) of this attribute(s) and/or aggregate information for each group
  - If we group rows into sets, we cannot look at the values in the individual rows anymore…

HAVING

… HAVING *condition on groups* …

- Includes only those groups that satisfy the condition
  - the condition may only involve the grouping attribute(s) and/or aggregate functions
  - can use all the same comparisons and logical operators as WHERE

Next:

- More SQL Queries
  - Review GROUP BY and HAVING
  - More query problems
  - Joins

CSC 355 Database Systems
Lecture 6

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topics:

- SQL queries
  - Review GROUP BY and HAVING
  - Query problems
  - Introduction to joins

Aggregate Functions

- Given an attribute, these functions take the values of that attribute in the set of returned rows and compute a single value from them
  - COUNT(…): Number of non-NULL values
  - SUM(…): Sum of the values
  - AVG(…): Average of the values
  - MIN(…): Smallest of the values
  - MAX(…): Largest of the values

GROUP BY

… GROUP BY *grouping attributes* …

- Combines the rows into sets based on the value(s) of some attribute(s)
  - Can only display the value(s) of this attribute(s) and/or aggregate information for each group
  - If we group rows into sets, we cannot look at the values in the individual rows anymore…

HAVING

… HAVING *condition on groups* …

- Includes only those groups that satisfy the condition
  - the condition may only involve the grouping attribute(s) and/or aggregate functions
  - can use all the same comparisons and logical operators as WHERE

Query Structure (again)

- General form of a query:

  SELECT *list of expressions*
  FROM *set of rows*
  [WHERE *condition on rows*]
  [GROUP BY *grouping attributes*]
  [HAVING *condition on groups*]
  [ORDER BY *ordering attributes*] ;

- Grouping goes after WHERE, before ORDER BY

Writing a Query

1. FROM: What table should I use?
2. WHERE: How do I indicate which rows to include?
3. GROUP BY: What attribute's values will define the sets? (May have to change SELECT * here…)
4. HAVING: How do I indicate which sets to include?
5. ORDER BY: How should I sort the rows/sets?
6. SELECT: What values do I have to compute and display?

Query Problems

- Find the number of workers in each department
  - (…whose salary is more than $40,000)
- Find the average salary over the entire company
- For each department, find the salary of the highest-paid employee
- List the department names and their total budgets, ordered from the largest total budget to the smallest
- For each student, find the total number of classes they have enrolled in and the most recent year that the student enrolled in a class

Joins

- Data that is distributed among multiple tables can be combined into a single set of rows for use in a query using different types of *joins*:
  - Inner joins (equi-join, natural join)
  - Outer joins (left, right, full)

Cartesian Product

- What if we list two tables in the FROM?
- The rows in the result come from combining all pairs of rows from the two tables – the *Cartesian Product* of the tables
  - (This is sometimes called the "cross join"…)
- This is almost certainly more rows than we want – most combinations are meaningless!

Equi-Join

- An equi-join keeps only those rows where the two combined rows agree on the shared attribute(s):

…FROM *TABLE1, TABLE2*
WHERE
TABLE1.Attribute = TABLE2.Attribute;

Natural Join

- Like an equi-join, but one of the duplicated columns is removed (the most common join):

SELECT *all but the duplicated attribute(s)*
FROM *TABLE1, TABLE2*
WHERE
    TABLE1.Attribute = TABLE2.Attribute;

Inner Joins

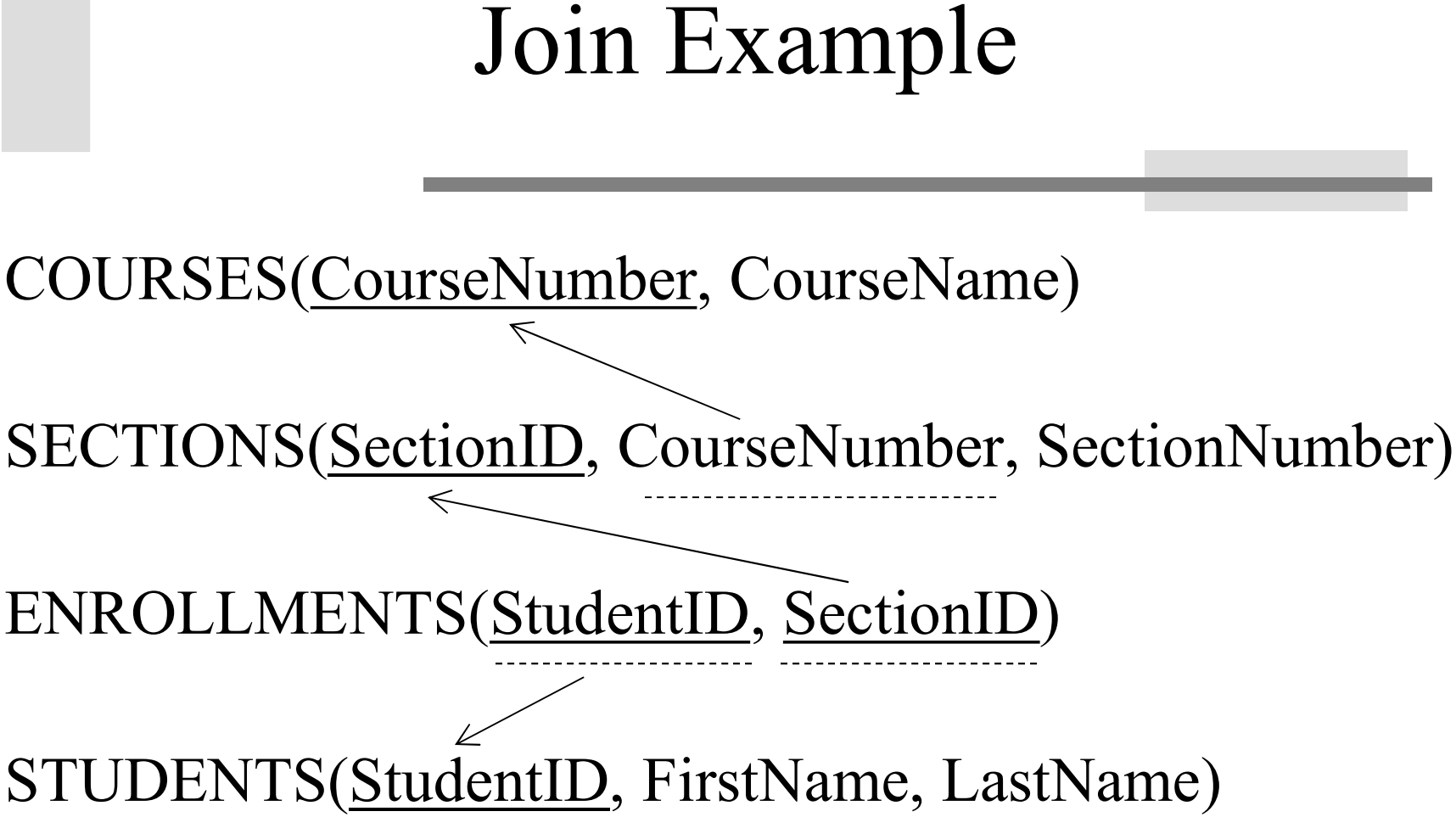
- These are both examples of *inner joins*
- In an inner join, the Cartesian Product is restricted to only include the combined rows that satisfy some condition
  - condition is usually equality in some shared key
  - e.g., equi-joins, natural joins

Inner Joins

- Rather than list of tables in the FROM and a WHERE condition, can use:

  FROM *TABLE1* INNER JOIN *TABLE2*
  ON *condition*

Join Example



Next:

- More SQL Queries
  - Inner joins
  - Outer joins
  - Query examples
  - Set operations

CSC 355 Database Systems
Lecture 7

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- SQL queries
  - Inner joins
  - Outer joins
  - Query problems with joins

Joins

- Data that is distributed among multiple tables can be combined into a single set of tuples for use in a query using different types of *joins*:
  - Inner joins (equi-join, natural join)
  - Outer joins (left, right, full)

Inner Joins

- … *TABLE1* INNER JOIN *TABLE2*
  ON *condition*;

  - Equi-join: includes all attributes of *TABLE1* and *TABLE2*, and *condition* is equality on shared attribute(s)
  - Natural join: like equi-join, but only displays one copy of each shared attribute

Join Example

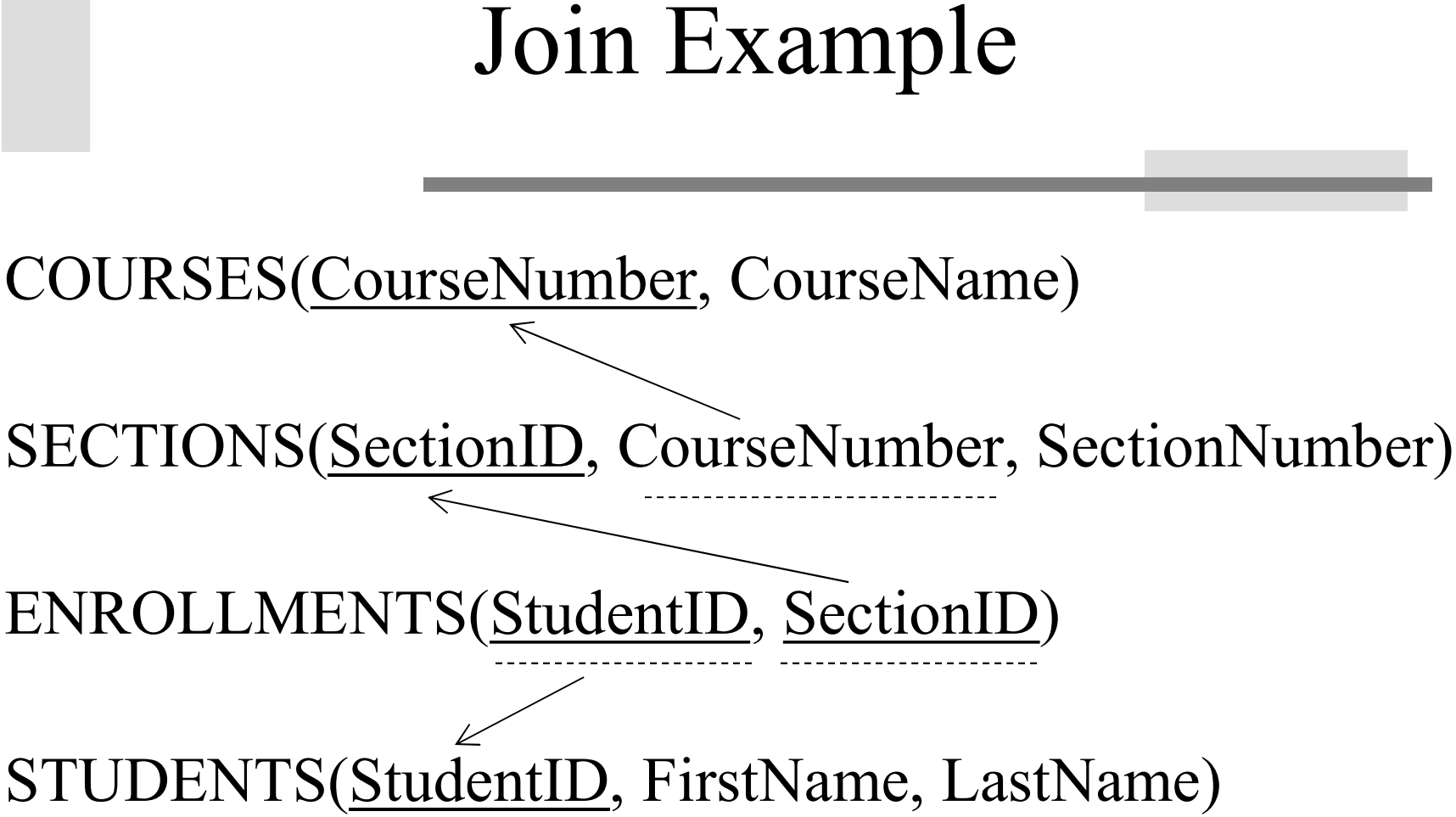


Table Aliases

- Can give alternate names to tables in FROM

  FROM *TABLE1 T1* INNER JOIN *TABLE2 T2*
  ON *condition*;

- Can use aliases *T1* and *T2* anywhere in query
  - Useful in joins if table names are long…

Inner Joins vs. Outer Joins

- An *inner join* requires that tuples in the tables satisfy some condition to create a tuple in the result.
- An *outer join* does not: a tuple in the result may be either
  - the combination of two tuples that satisfy the condition (*matching tuple*)
  - a tuple that does not match anything, combined with an all-NULL tuple (*non-matching tuple*)

Left Outer Join

- Includes all matching tuples, plus a tuple for each tuple in the first table that has no match

... *TABLE1* LEFT OUTER JOIN *TABLE2*
ON *TABLE1.Attribute* = *TABLE2.Attribute*;

Right Outer Join

- Includes all matching tuples, plus a tuple for each tuple in the second table that has no match

... *TABLE1* RIGHT OUTER JOIN *TABLE2*
ON *TABLE1.Attribute* = *TABLE2.Attribute*;

Full Outer Join

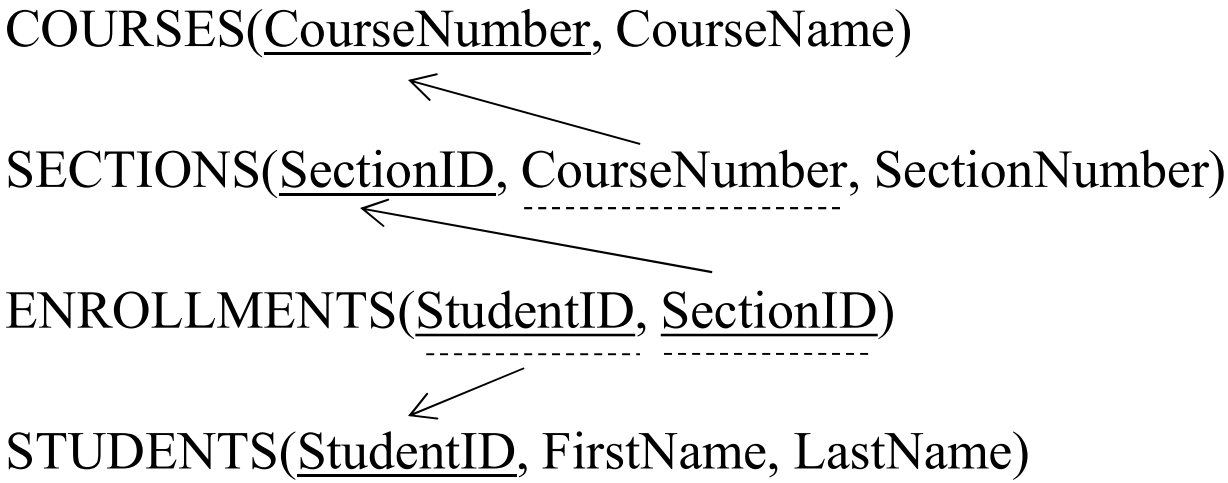
- Includes all matching tuples, plus a tuple for each tuple in either table that has no match

... *TABLE1* FULL OUTER JOIN *TABLE2*
ON *TABLE1.Attribute* = *TABLE2.Attribute*;

Query Problems

- Give the names of all students that have enrolled in any GAM course
- Give the ID numbers of all students who have not enrolled in any classes
- Give the names of all members of HerCDM
- Give the names of all students who are the president of a student group
- Give the names of all courses that Abigail Winter has enrolled in

Final Join Example



"For each student, list the course names and section numbers that he/she is enrolled in. (Then find the total number of courses he/she is enrolled in.)

Next:

- SQL queries
  - Subqueries

CSC 355 Database Systems
Lecture 8

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Topics:

- SQL queries
  - Subqueries
  - Set operations

Subqueries

- The result of one query may be needed by another to compute its result
  - A *subquery* is nested (using parentheses) within an *outer query*
  - Outer query uses the result of the subquery, which can be either single value or a table
- The subquery usually appears in a WHERE or HAVING clause (sometimes in a FROM)

Uses of Subqueries

- "Find all employees that receive the highest salary." (Find the highest salary)
- "Find IDs of all course sections being taken by Paul Konrad". (Find his StudentID)
- "Find IDs of all students who are taking some course section with Student 1234567 this quarter." (Find all sections being taken by 1234567)

Returning a Single Value

- When a single value is returned, it can be used like any other value on the right-hand side of a WHERE or HAVING condition

```
SELECT * FROM ASSIGNMENT
    WHERE Hours >
    (SELECT AVG(Hours) FROM ASSIGNMENT);
```

Returning a Table

- Can check whether the returned table is empty:
  - EXISTS (*query*) is true if table is not empty
  - NOT EXISTS (*query*) is true if table is empty
- Can check contents of table:
  - *tuple* IN (*query*) returns true if *tuple* appears in the returned table
    - (in most cases, the tuple is just one attribute and the SELECT in the subquery contains just one attribute…)

Returning a Table

- Can compare an attribute to table contents:
  - …only when SELECT contains just one attribute
  - *Attribute* > ALL (*query*) returns true if *Attribute* is greater than all values in the returned column
  - *Attribute* > ANY (*query*) returns true if *Attribute* is greater than any value in the returned column

(Any type of comparison is allowed, not just > …)

Correlated Subqueries

- A subquery may refer to attributes of the table in the outer query
  - The subquery will be evaluated repeatedly, once for each tuple in the table in the outer query
  - Attributes from outer query table must be qualified with the table name if they appear in the subquery
  - If the tables in the outer query and subquery are the same, must create an alias for the outer query table

Subquery Problems

- Give the names of all courses that Abigail Winter has enrolled in
- Give the IDs of all students who started as part of the most recent incoming group of students
- List the names of all members of DeFrag
- List the IDs of all courses that have been taken by Information Systems majors
- Give the IDs (names?) of all group presidents who are not members of their groups

Set Operations

Given two sets A and B:

- $A \cup B = \{x \mid x \in A \vee x \in B\}$ ("union")
  - The set of all elements that are in A or B (or both)
- $A \cap B = \{x \mid x \in A \wedge x \in B\}$ ("intersection")
  - The set of all elements that are in both A and B
- $A - B = \{x \mid x \in A \wedge x \notin B\}$ ("difference")
  - The set of all elements that are in A but not in B

Set Operations in SQL

- Combine the results of two queries, as long as the results contain compatible tuples:
  - UNION: rows that appear in at least one result
  - INTERSECT: rows that appear in both results
  - MINUS: rows that appear in the first result but not in the second
- The final result must be a set, so duplicates are removed from the two results first…

Next Time:

- Transactions

CSC 355 Database Systems
Lecture 9

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Transactions

Problem: Interruptions

- Two SQL statements are written to transfer money from one bank account to another…
- …one executes, then the server crashes
  - What happens to the money?

Problem: Concurrency

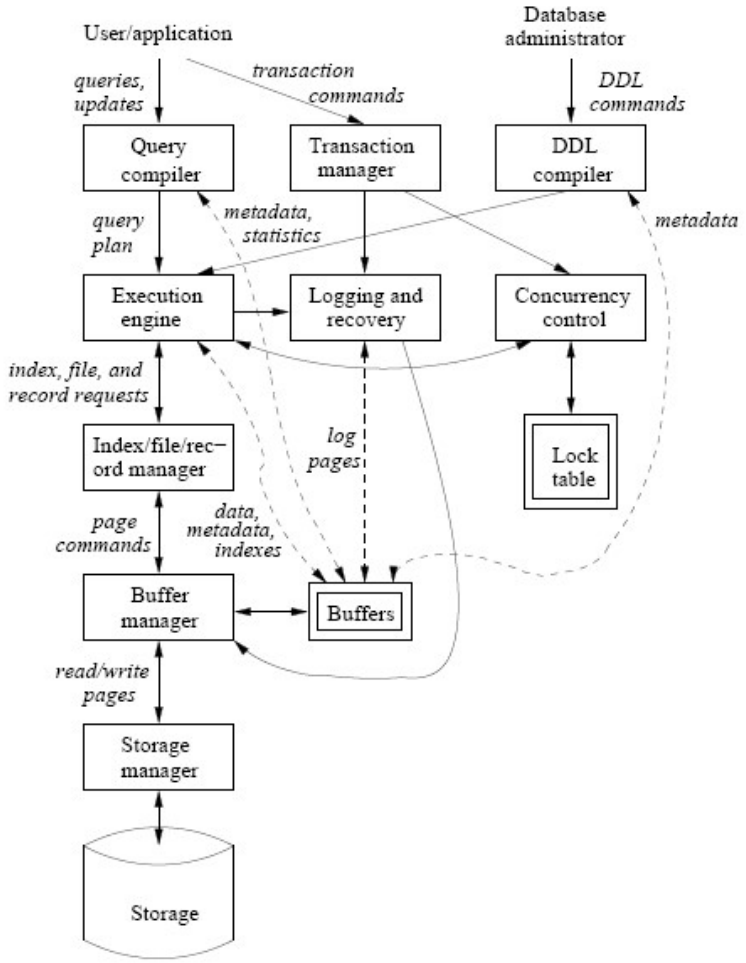
- Two users check the balance of the same bank account…
- …then both try to transfer money out of it
  - Who gets it?

Solution: Transactions

- A *transaction* is a collection of SQL statements that must be executed as a unit
- The Transaction manager in the DBMS must handle
  - Interruptions: Logging and recovery, Buffers
  - Concurrency: Concurrency control, Lock tables

Components of a DBMS

(From Ullman/Widom)



Transactions in Oracle

- Any operation that changes the database state starts an implicit transaction in Oracle
- Can also start a transaction explicitly with a SET TRANSACTION statement
- End a transaction with a COMMIT statement
  - Transaction can also be ended with ROLLBACK, usually done by system rather than user…

ACID Properties

- A transaction should satisfy the following properties:
  - Atomicity: Executes completely or not at all
  - Consistency: Satisfies all database constraints
  - Isolation: Executes “separately” from others
  - Durability: Once executed, results are permanent

Atomicity

- Transaction operations are kept in a local store, not applied to the database immediately
  - Transaction can see its own changes, others can't
- When a transaction is completed, COMMIT applies changes to the shared database in their entirety ("executes completely…")
- ROLLBACK during a transaction undoes any partial results ("…or not at all")

Durability

- After COMMIT, the changes applied to the shared database are permanent, and cannot be rolled back later

Consistency

- Constraints can be “deferred”, so that they are only checked when a transaction commits, not for each individual statement
  - Add DEFERRABLE INITIALLY DEFERRED to the constraint definition
  - If a constraint is violated when you COMMIT, a ROLLBACK of the entire transaction is done!

Isolation

- DBMS maintains “separation” among transactions that access data concurrently
  - Various different levels of isolation
    - Transactions might modify, or just read, data
  - Tradeoff between performance and data integrity

Serializable Isolation

- Transactions must behave as though they were run serially (first one, then the other)
- Usually implemented by “locking” the tables (or parts of tables) used by a transaction
  - Other transactions using the same tables will have to wait until they are released
  - Other transactions using other tables could run at the same time

Read Committed Isolation

- Transaction operations can be interleaved
- If one transaction tries to read data that were written by another, it can only see the changes that have been committed
  - Can’t be rolled back, but could be changed later
  - Multiple queries of the same table might not yield the same results... “non-repeatable reads”, including “phantoms”…

Read Uncommitted Isolation

- Transaction operations can be interleaved
- If one transaction tries to read data that were written by another, it can see all changes, even if they have not been committed
  - Could be rolled back by the other transaction!
  - Transaction might make decisions based on values that are later rolled back and so were never really there … “dirty reads”…

Isolation Levels

- **SERIALIZABLE:** Transactions must appear to run serially – cannot read any changes from others
- **[REPEATABLE READ:** Can read committed changes that only add data (allows only phantoms)]
- **READ COMMITTED:** Can read all committed changes (allows all non-repeatable reads)
- **READ UNCOMMITTED:** Can read all changes (allows all non-repeatable reads and dirty reads)

Transactions in Oracle

- SET TRANSACTION statement can specifiy ISOLATION LEVEL
  - READ COMMITTED (default)
  - SERIALIZABLE
  - Oracle does not support REPEATABLE READ and READ UNCOMMITTED
- COMMIT or ROLLBACK ends transaction

Next:

- Finish Transactions
- Review for Midterm Exam
- Introduction to Relational Database Design

CSC 355 Database Systems
Lecture 10

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Finish Transactions
- Midterm Exam Information
- Introduction to Relational Database Design

Transactions

- A *transaction* is a collection of SQL statements that must be executed as a unit
- Transactions have *"ACID"* properties:
  - Atomicity: Execute completely or not at all
  - Consistency: Satisfy all database constraints
  - Isolation: Execute "separately" from others
  - Durability: Once completed, results are permanent

Serializable Isolation

- Transactions must behave as though they were run serially (first one, then the other)
- Usually implemented by “locking” the tables (or parts of tables) used by a transaction
  - Other transactions using the same tables will have to wait until they are released
  - Other transactions using other tables could run at the same time

Read Committed Isolation

- Transaction operations can be interleaved
- If one transaction tries to read data that were written by another, it can only see the changes that have been committed
  - Can’t be rolled back, but could be changed later
  - Multiple queries of the same table might not yield the same results... “non-repeatable reads”, including “phantoms”…

Read Uncommitted Isolation

- Transaction operations can be interleaved
- If one transaction tries to read data that were written by another, it can see all changes, even if they have not been committed
  - Could be rolled back by the other transaction!
  - Transaction might make decisions based on values that are later rolled back and so were never really there … “dirty reads”…

Isolation Levels

- **SERIALIZABLE:** Transactions must appear to run serially – cannot read any changes from others
- **[REPEATABLE READ:** Can read committed changes that only add data (allows only phantoms)]
- **READ COMMITTED:** Can read all committed changes (allows all non-repeatable reads)
- **READ UNCOMMITTED:** Can read all changes (allows all non-repeatable reads and dirty reads)

Transactions in Oracle

- SET TRANSACTION to start a transaction
  - Specify transaction NAME
  - Can also specify ISOLATION LEVEL
    - READ COMMITTED (default) or SERIALIZABLE
  - COMMIT or ROLLBACK ends transaction
  - SQL statements that modify data start a transaction implicitly… COMMIT to end it

Midterm Exam Information

- Exam Monday 5/4, at regular class time (90 minutes)
- Exam will be given as a quiz in d2l
- Lecture slides will be available, but no other sources of information (electronic, printed, or human) may be consulted
  - I will have you sign a statement agreeing to this as part of exam
- Sections 1.1-1.3, 2.1-2.3, 6.1-6.6, 7.1-7.3 covered
- Multiple choice, short answers, writing and/or evaluating SQL queries and transactions
- Review outline has been posted
- Optional Q&A session Friday 5/1, via Zoom

Relational Database Design

- Start with a set of attributes

  $R = \{A_1, A_2, \ldots, A_n\}$

  - (Can also be written as a *universal relation*

    $R(A_1, A_2, \ldots, A_n) \ldots)$

- Construct a *decomposition* of R into relations

  $D = \{R_1, R_2, \ldots, R_m\}$

  - Each R_i is a subset of R

Relational Database Design

- The decomposition D = $\{R_1, R_2, \ldots, R_m\}$ should satisfy the following conditions:
  1. The union of the R_i's is R
  2. Redundancy has been removed from the R_i's
  3. Dependencies among attributes in R are preserved
  4. The original relation R can be recovered from D
- Conditions 2.-4. have to be formalized…

Redundancy

- *Redundancy* occurs when more than one record in a table stores the same information
  - Wastes space
  - Allows *update* and *deletion* anomalies
- Remove redundancy by identifying (and removing) *functional dependencies* in R

Functional Dependencies

- A set of attributes $Y = \{Y_1, Y_2, \ldots, Y_n\}$ is *functionally dependent* on a set of attributes $X = \{X_1, X_2, \ldots, X_m\}$ if and only if every pair of tuples that have the same values for X must also have the same values for Y
  - Also "X functionally determines Y" or "X → Y"
  - X is called the *determinant*
- (Less formally, "the values of X uniquely determine the values of Y"…)

Functional Dependencies

- "Every pair of tuples that have the same values on X also have the same values on Y"
  - For X to functionally determine Y, this condition must be *satisfied by every possible relation state*
  - If *some relation state does not satisfy* the condition because two tuples have the same values on X but different values on Y, then X does not functionally determine Y

Finding Functional Dependencies

- DVD (DVDID , MovieID , Title , Genre , Length , Rating)
- GRADING (CNumber , CTitle , SID , SName , Grade)
- (We do not typically include *trivial* functional dependencies, where Y is a subset of X…)

Okay … but why?

- Redundancy comes from functional dependencies whose determinants do not include a complete candidate key of R
- We use the functional dependencies to construct decompositions of R
- How do we measure the quality of the resulting decompositions?

Next:

- Midterm exam Monday 5/4
  - Q&A session Friday 5/1
- Next lecture will be posted Wednesday 5/6

CSC 355 Database Systems
Lecture 11

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Relational Database Design
  - Functional Dependencies
  - Closures and Keys
  - Boyce-Codd Normal Form (BCNF)

Relational Database Design

- Start with a set of attributes

  $R = \{A_1, A_2, \ldots, A_n\}$

  - (Can also be written as a *universal relation*

    $R(A_1, A_2, \ldots, A_n) \ldots)$

- Construct a *decomposition* of R into relations

  $D = \{R_1, R_2, \ldots, R_m\}$

  - Each R_i is a subset of R

Relational Database Design

- The decomposition D = {R_1, R_2, …, R_m} should satisfy the following conditions:
  1. The union of the R_i's is R
  2. Redundancy has been removed from the R_i's
  3. Dependencies among attributes in R are preserved
  4. The original relation R can be recovered from D
- Conditions 2.-4. have to be formalized…

Redundancy

- *Redundancy* occurs when more than one record in a table stores the same information
  - Wastes space
  - Allows *update* and *deletion* anomalies
- We eliminate redundancy by identifying (and perhaps removing) functional dependencies in R

Functional Dependencies

- A set of attributes $Y = \{Y_1, Y_2, \ldots, Y_n\}$ is *functionally dependent* on a set of attributes $X = \{X_1, X_2, \ldots, X_m\}$ if and only if every pair of tuples that have the same values for X must also have the same values for Y
  - Also "X functionally determines Y" or "X → Y"
  - X is called the *determinant*
- (Less formally, "the values of X uniquely determine the values of Y"…)

Functional Dependencies

- "Every pair of tuples that have the same values on X also have the same values on Y"
  - For X to functionally determine Y, this condition must be *satisfied by every possible relation state*
  - If *some relation state does not satisfy* the condition because two tuples have the same values on X but different values on Y, then X does not functionally determine Y

Finding Functional Dependencies

- DVD (DVDID , MovieID , Title , Genre , Length , Rating)
- GRADING (CNumber , CTitle , SID , SName , Grade)
- PERSON (First , Last , Address , City , State , Zip)
- ASSIGNMENT (EID , ELName , EFName , Project, Hours)

Closures

- For F and a set of attributes X, the set X^+ is called the *closure of X (with respect to F).*
- X^+ is the set of all attributes that can be determined from X using anything in F
  - To find X^+: Start with just X … the add any other attributes you can determine from X using F … then add any other attributes you can determine from those … and so on … until you can't add any more.
- If X^+ includes the set Y, then X→Y can be *derived* from the set F

Equivalence

- Two sets F_1 and F_2 of functional dependencies are *equivalent* if and only if both of the following are true:
  - Every functional dependency in F_1 can be derived from the set F_2
  - Every functional dependency in F_2 can be derived from the set F_1
- Use closure to test each functional dependency

Definitions of Keys

- A set of attributes X is a *superkey* of R if X determines all attributes of R (i.e., if $X^+ = R$)
- A set of attributes X is a *candidate key* of R if X is a superkey, but no proper subset Y of X is a superkey
- An attribute is *prime* if it is contained in some candidate key (and is *non-prime* otherwise)

Eliminating Redundancy

- Functional dependencies whose determinants are not superkeys (i.e, that do not include candidate keys) indicate that there is redundancy in a relation
  - If there aren't any… then we're done!
  - If there are… then we use the functional dependencies to construct a decomposition that gets rid of the redundancy

BCNF

- A relation R is in Boyce-Codd Normal Form (BCNF) if for every non-trivial functional dependency X→Y in R, X is a superkey
  - "Every determinant must contain a candidate key"
  - A relation in BCNF will not have any redundancy, since every functional dependency in the relation will have a superkey as its determinant

Removing a Functional Dependency

- Suppose R contains the functional dependency X→Y where X is not a superkey
- Replace R with two relations:
  - R – Y
    - No longer contains X→Y
  - X ∪ Y
    - Contains X→Y, but X is a superkey in this relation
  - (Be sure that Y contains the complete closure of X…)

BCNF Decomposition Algorithm

Set D = {R}

While there is some Q in D that is not in BCNF:

Choose a Q that is not in BCNF

Find an X→Y in Q that violates BCNF

Replace Q with two relations:

Q – Y and (X union Y)

(When finished, all relations in D will be in BCNF)

Next:

- More Relational Database Design
  - Boyce-Codd Normal Form (BCNF)
  - Dependency preservation
  - Lossless join

CSC 355 Database Systems
Lecture 12

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Relational Database Design
  - BCNF decomposition algorithm
  - Dependency preservation
  - Lossless join

Relational Database Design

- Starting with $R(A_1, A_2, \ldots, A_n)$, the decomposition $D = \{R_1, R_2, \ldots, R_m\}$ should satisfy the following conditions:
  1. The union of the R_i's is R
  2. Redundancy has been removed from relations in D
  3. The functional dependencies in R are preserved
  4. The original relation R can be recovered from D
- We are formalizing conditions 2.-4. …

Functional Dependencies

- A set of attributes $Y = \{Y_1, Y_2, \ldots, Y_n\}$ is *functionally dependent* on a set of attributes $X = \{X_1, X_2, \ldots, X_m\}$ if and only if every pair of tuples that have the same values for X must also have the same values for Y
  - Also "X functionally determines Y" or "X → Y"
  - X is called the *determinant*
- (Less formally, "the values of X uniquely determine the values of Y"…)

Keys

- A set of attributes X is a *superkey* of R if X determines all attributes of R
- A set of attributes X is a *candidate key* of R if X is a superkey, but no proper subset Y of X is a superkey
- An attribute is *prime* if it is contained in some candidate key (and is *non-prime* otherwise)

BCNF

- A relation R is in Boyce-Codd Normal Form (BCNF) if for every non-trivial functional dependency X→Y in R, X is a superkey
  - "Every determinant must contain a candidate key"
  - A relation in BCNF will not have any redundancy, since every functional dependency in the relation will have a superkey as its determinant

BCNF Decomposition Algorithm

Set D = {R}

While there is some Q in D that is not in BCNF:

Choose a Q that is not in BCNF

Find an X→Y in Q that violates BCNF

Replace Q with two relations:

Q – Y and (X union Y)

(When algorithm is done, all relations will be in BCNF)

Projections

- Suppose we have a set of functional dependencies F in the relation R
- For a relation R_i, consider all $X \rightarrow Y$ that can be derived from F where both X and Y are subsets of R_i
- This set is called the *projection of F on* R_i
  - It represents the set of all constraints that F puts on the attributes of R_i

Dependency Preservation Property

3. The union over all i in {1,…,m} of the projections of F on R_i is equivalent to F

- We want the set of all projections to be equivalent to F – that is, the decomposition neither destroys any functional dependencies in F nor introduces any new ones…
  - Not all decompositions have this property!

Restrictions

- The *restriction* of a relation state r to a set S is the set of distinct tuples obtained from r by including only the values of the attributes in S from each tuple
- Restrictions that overlap can be combined using a natural join on the attributes they share
  - Ideally, this will yield a result that is still a restriction of the original relation state r…

Lossless Join Property

4. For every relation state r of R, the natural join of the restrictions of r to the relations R_1, R_2, …, R_m in the decomposition the same as r
   - That is, if we take the restrictions of any relation state and join them back together, we will get the original relation state – no *spurious tuples* are added
   - Not all decompositions have this property!

Relational Database Design

- Starting with $R(A_1, A_2, \ldots, A_n)$, the decomposition $D = \{R_1, R_2, \ldots, R_m\}$ should satisfy the following conditions:
  1. The union of the R_i's is R
  2. Each of the R_i's is in BCNF
  3. D has the dependency preservation property
  4. D has the lossless join property

Binary Lossless Join Test

- D = $\{R_1, R_2\}$ has the lossless join property if and only if one or both of the following hold:
  1. $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$ can be derived from F
  2. $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$ can be derived from F
  - That is, if and only if the intersection between the two sets of attributes is a superkey in one of the relations…

General Test for Lossless Join

1. Create a matrix S with a row i for each R_i and a column for j for each A_j
2. Set each S(i,j) = "b_{ij}"
3. For each entry (i,j)

   If relation R_i includes A_j, then set S(i,j) = "a_j"
4. Repeat the following loop until there are no changes to S:

   For each X→Y in F

   For all rows in S that have the same symbols in all columns in X,

   set all of the columns in Y in those rows to agree as follows:

   If any row has a_j for the columns, set all rows to that same a_j

   If not, choose one of the b_{ij}s and set all rows to that same b_{ij}
5. If any row has all a_j's, return true (D has the lossless join property);

   if not, return false (D does not have lossless join property).

Next:

- Finish lossless join
- Third Normal Form (3NF)
- Minimal basis
- Algorithm for 3NF decomposition

CSC 355 Database Systems
Lecture 13

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Relational Database Design
  - Lossless join property
  - Third Normal Form (3NF)
  - Minimal basis
  - Algorithm for 3NF decomposition

Relational Database Design

- Starting with $R(A_1, A_2, \ldots, A_n)$, the decomposition $D = \{R_1, R_2, \ldots, R_m\}$ should satisfy the following conditions:
  1. The union of the R_i's is R
  2. Each of the R_i's is in BCNF
  3. D has the dependency preservation property
  4. D has the lossless join property

Lossless Join Property

4. For every relation state r of R, the natural join of the restrictions of r to the relations R_1, R_2, …, R_m in the decomposition is r itself

- If we decompose and then join back the resulting relations, we get the original relation state – no *spurious tuples* are added
  - Not all decompositions have this property!

Binary Lossless Join Test

- D = $\{R_1, R_2\}$ has the lossless join property if and only if one or both of the following hold:
  1. $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$ can be derived from F
  2. $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$ can be derived from F
- That is, if and only if the intersection between the two sets of attributes is a superkey in one of the relations…

General Test for Lossless Join

1. Create a matrix S with a row i for each R_i and a column for j for each A_j
2. Set each S(i,j) = "b_{ij}"
3. For each entry (i,j)

   If relation R_i includes A_j, then set S(i,j) = "a_j"
4. Repeat the following loop until there are no changes to S:

   For each X→Y in F

   For all rows in S that have the same symbols in all columns in X,

   set all of the columns in Y in those rows to agree as follows:

   If any row has a_j for the columns, set all rows to that same a_j

   If not, choose one of the b_{ij}s and set all rows to that same b_{ij}
5. If any row has all a_j's, return true (D has lossless join); if not, return false

   D does not have lossless join).

BCNF

- A relation R is in Boyce-Codd Normal Form (BCNF) if for every non-trivial functional dependency X→Y in R, X is a superkey
  - "Every determinant must contain a candidate key"
  - A relation in BCNF will not have any redundancy, since every functional dependency in the relation will have a superkey as its determinant

3NF

- A relation R is in Third Normal Form (3NF) if for every non-trivial functional dependency X→{A} in R, either X is a superkey or A is a prime attribute
  - This is a weaker condition than BCNF, since X doesn't have to be a superkey if A is prime
  - 3NF may allow some redundancy if there is more than one candidate key

BCNF vs. 3NF

- Every relation in BCNF is in 3NF
- Not every relation in 3NF is in BCNF
  - In a 3NF relation, a prime attribute may be determined by something that is not a superkey, but BCNF will not allow that
  - 3NF and BCNF conditions are equivalent if there is only one candidate key in the relation

Example

- R(A, B, C, D)
- F = { A,C → B ; A,C → D ; D → C }
  - Identify candidate keys, prime attributes
  - Is R in BCNF? Is it in 3NF?
  - Construct a BCNF decomposition…
  - …but how do we construct 3NF decompositions?

Minimal Basis

- A *minimal basis* of F is a set G that is equivalent to F and is "as small as possible":
  1. The right side of every dependency is a single attribute
  2. No X→A can be replaced with Y→A, where Y is a proper subset of X, and still be equivalent to F
  3. No X→A can be removed and still be equivalent to F

Constructing a Minimal Basis

- Start with a set of functional dependencies…
  1. Split each X→Z into an equivalent set with only one attribute on the right side of each f.d.
  2. For each X→{a}, replace it with Y→{a} (where Y is a proper subset of X) as long as the resulting set is equivalent
  3. Remove every X→{a} that can be removed as long as the resulting set is equivalent

3NF Decomposition Algorithm

1. Find a minimal basis G of F
2. For each set X that appears as the determinant of some functional dependency in G:

   Find all k dependencies of the form $X \rightarrow A_i$ in G, and create a relation in D with the attributes in X and $A_1,\ldots,A_k$.
3. If none of the relations in D contains a candidate key of R, find a candidate key K of R and create a relation in D whose attributes are the attributes of K.
4. Remove any redundant relations. (A relation Q_1 in D is redundant if all of its attributes are included in another relation Q_2 in D.)

Comparison of Algorithms

- Decomposition into BCNF relations:
  - No redundancy left in relations
  - Dependency preservation is not guaranteed
  - Lossless join is guaranteed
- Decomposition into 3NF relations:
  - Some redundancy may remain in relations that have multiple candidate keys
  - Dependency preservation is guaranteed
  - Lossless join is guaranteed

Next Time:

- Finish Relational Database Design
- Introduction to Triggers

CSC 355 Database Systems
Lecture 14

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Last Relational Database Design Example
- Introduction to Triggers

Relational Database Design

- Starting with $R(A_1, A_2, \ldots, A_n)$, the decomposition $D = \{R_1, R_2, \ldots, R_m\}$ should satisfy the following conditions:
  1. The union of the R_i's is R
  2. Each of the R_i's is in BCNF (or 3NF)
  3. D has the dependency preservation property
  4. D has the lossless join property

Comparison of Algorithms

- BCNF decomposition algorithm:
  - No redundancy in relations
  - Dependency preservation not guaranteed
  - Lossless join guaranteed
- 3NF decomposition algorithm:
  - Some redundancy may remain in relations that have multiple candidate keys
  - Dependency preservation guaranteed
  - Lossless join guaranteed

Final Decomposition Example

- INVOICE (OrderID, OrderDate, CustomerID, Name, Address, ProductID, Description, Material, Price, Quantity)
  - Review functional dependencies, candidate keys
  - Is INVOICE in BCNF? Is it in 3NF?
  - Construct BCNF and 3NF decompositions
  - Verify dependency preservation and lossless join

Relational Model

- As a data model, the relational model must describe three things about stored data:
  - Structure of the data: A collection of linked two-dimensional tables
  - Operations on the data: Described by SQL statements
  - Constraints on the data: Domain, key, entity integrity, referential integrity, check … what else?

Assertions

- An assertion is a condition that cannot be false for any state of the database
  - If deferred, just at the end of each transaction…
- Can involve any tables in the database

```
CREATE ASSERTION SalaryCap CHECK
( 1000000 >=
        (SELECT SUM(Salary) FROM WORKER ) );
```

- Assertions are not supported by major DBMSs

Triggers

- A trigger is a procedure that is executed in response to a particular database operation

```
CREATE TRIGGER SalaryCap AFTER INSERT ON WORKER
BEGIN
        SELECT SUM(Salary) INTO total FROM WORKER;
        IF (total > 1000000) THEN
                ERROR('Million Dollar Limit Exceeded');
        END IF;
END;
```

Triggers

- Triggers allow general responses to changes in the database state, e.g.:
  - Enforcement of business rules
  - Notification of events
  - Maintenance of derived information
  - Maintenance of replicated data
  - Implementation of multi-table constraints

Triggers

- A trigger definition must specify:
  - An *event* (e.g., insert, delete, update) that causes the trigger to fire
  - A *condition* that is tested (on old and/or new state) to decide whether or not the trigger will respond
  - An *action* (PL/SQL block or stored procedure) that may be executed in response

Oracle Trigger Syntax

CREATE [OR REPLACE] TRIGGER *Name*
BEFORE/AFTER
    INSERT OR DELETE OR UPDATE [OF *Attribute*] ON *TABLE*
[REFERENCING
    OLD AS *OldName*
    NEW AS *NewName*]
[FOR EACH ROW]
[WHEN (*condition*)]
DECLARE
    …variable declarations…
BEGIN
    …PL/SQL statements…
END;
/

Oracle Trigger Syntax

- BEFORE
  - Indicates that queries on *TABLE* will be done on the state of the table before the triggering operation executes
- AFTER
  - Indicates that queries on *TABLE* will be done on the state that the table would be in after the triggering operation executes

Oracle Trigger Syntax

- INSERT OR DELETE OR
  UPDATE [OF *Attribute*] ON *TABLE*
  - What operation(s) will cause the trigger to fire?
  - Trigger will fire in response to any INSERT or DELETE
  - Trigger may be set to fire in response to any UPDATE, or only an UPDATE of a particular attribute

Oracle Trigger Syntax

- FOR EACH ROW
  - If not included, indicates a *statement-level* trigger that will fire just once for the entire operation
  - If included, indicates a *row-level* trigger that will fire once for each row that is modified
    - … so an UPDATE or DELETE that applies to multiple rows will cause the trigger to fire more than once for the operation …

Oracle Trigger Syntax

- REFERENCING OLD AS *OldName*, NEW AS *NewName*
  - The original and modified states of the row being operated upon are called “old” and “new” unless you change them (used for row-level triggers only)
    - INSERT has only “new”, but no “old”
    - DELETE has only “old”, but no “new”
    - UPDATE has both “old” and “new”

Oracle Trigger Syntax

- WHEN (*condition*)
  - Condition tested to see whether or not the trigger action will actually execute
    - Statement-level: can query original or modified table state depending on whether BEFORE or AFTER is used
    - Row-level: can reference original and modified row states with “old” and “new” (INSERT has only “new”, DELETE has only “old”, UPDATE has both!)

Oracle Trigger Syntax

- *PL/SQL statements*
  - This block of code is executed when the trigger fires and the WHEN condition is satisfied
  - It may include:
    - SQL statements
    - PL/SQL statements
    - Calls to built-in or user-defined procedures/functions

Oracle Trigger Restrictions

- new and old can only refer to row states, and can only be used for row-level triggers
  - Use new and old in WHEN condition, but :new and :old elsewhere
- PL/SQL statements in a row-level trigger cannot query or modify the table that triggered the action
- Subqueries are not allowed in WHEN

Trigger Examples

- Salary Cap:
  - Trigger cancels any operation that causes the company's total budget for salaries to exceed $1,000,000 (statement-level)
- Departmental Budgets:
  - Trigger maintains current totals of the salaries of all employees in each department (row-level)

Next:

- Database Programming in PL/SQL
- Back to Triggers

CSC 355 Database Systems
Lecture 15

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Database Programming in PL/SQL
- Back to Triggers

Database Programming

- Three main approaches:
  1. Embed database commands in a general programming language
  2. Create a library of database functions in an API (e.g., JDBC)
  3. Design a general programming language that includes database commands (e.g., PL/SQL)
- First two approaches can suffer from *impedance mismatch*

PL/SQL

- PL/SQL is Oracle's version of the SQL/PSM ("Persistent Stored Modules") standard
- PL/SQL is a procedural programming language that includes SQL – it can:
  - create and issue SQL statements
  - store and process the results of queries
  - define triggers to respond to database events

Database Programming in PL/SQL

- Three places PL/SQL code can go:
  1. Within a trigger that is executed in response to database events
  2. Within a procedure or function that is executed when called by name
  3. Within an *anonymous block* that is executed directly by a user

Anonymous Block

- Will be executed directly (like a script):

```
declare
        -- variable declarations
begin
        -- PL/SQL statements to execute
        -- each statement must end with a semicolon
exception
        -- exception handling (optional)
end;
/
```

Output

- To display output:

dbms_output.put_line(*string*);

- Output buffer is displayed in Dbms Output tab when anonymous block is completed
  - Use View → Dbms Output and '+' to open tab
- Concatenation of strings uses ||

Variables

- All variables must be declared:

  varName dataType [:= *initialValue*];

- SQL data types are available (e.g., number, char, varchar2), plus binary_integer and boolean
- Assignments use :=, and PL/SQL has typical arithmetic operations

Variables

- Only one variable can be declared per line, but variable types can be given in terms of the domain of another variable or attribute:

  varName otherVar%type;
  varName TABLE.Attribute%type;

- Can use *substitution variables* (e.g., &X) to prompt user for values

Branching

- if-then:

  if *condition* then

  …*'true' statements*…

  end if;

- if-else:

  if *condition* then

  …*'true' statements*…

  else

  …*'false' statements*…

  end if;

Branching

- if-elsif:

if *condition1* then
    … *'t' statements*…
elsif *condition2* then
    … *'f-t' statements*…
elsif *condition3* then
    … *'f-f-t' statements*…
(… *as many times as needed*…)
else
    … *'all f' statements*…
end if;

- case:

case *variable*
when *value1* then
    … *'value1' statements*…
when *value2* then
    … *'value2' statements*…
(… *as many times as needed*…)
else
    … *'nomatch' statements*…
end case;

Loops

- General loop:

  loop
  …*loop body*…
  end loop;

- Repeats until exit; is executed in loop body

- While loop:

  while *condition* loop
  …*loop body*…
  end loop;

- Repeats until *condition* is false

Loops

- For loop:

for *variable* in [reverse] *lower..upper* loop
…loop body…
end loop;

- Can only increment/decrement by one
- lower always appears before upper in header

Incorporating SQL Queries

- Result of a query can be stored in a set of variables by adding INTO clause to query
  - Variable types must match attribute types
  - Query must return a single record

SELECT *list of attributes*
INTO *list of variables*
FROM *list of tables*
...

Cursors

- A *cursor* represents a pointer into a set of records returned by a query

  cursor *name* is *query*;

- cursor *name* can be used to iterate through the records returned by *query*

Cursor Commands/Expressions

- open *name*; -- initializes to beginning of set
- fetch *name* into *variableList*;
  -- reads the next record into the variables
- close *name*; -- closes the cursor
- *name*%found
  -- true if last call to fetch succeeded
- *name*%rowcount
  -- number of records successfully fetched

Records

- Can declare a record with the same structure as a table row (fields are table attributes)

  *recordName TABLE%*rowtype;

- Can select a row of a table directly into a record, then access individual fields with

  recordName.Attribute

Cursor For Loop

- To iterate through all of the rows returned by a query:

for *recordName* in *cursorName*
    …loop body…
end loop;

- The needed record must be declared, but open/fetch/close can be omitted in this loop

Database Programming References

- Ullman/Widom, Section 9.4
- Oracle's "PL/SQL User's Guide and Reference", Chapters 1-6 (link posted)
- Stanford Infolab's "Using Oracle PL/SQL" (link posted)
- Murach's "Oracle SQL and PL/SQL for Developers (second edition)", Chapters 13 and 16 (recommended text)

Oracle Trigger Syntax

CREATE [OR REPLACE] TRIGGER *Name*
BEFORE/AFTER
    INSERT OR DELETE OR UPDATE [OF *Attribute*] ON *TABLE*
[REFERENCING
    OLD AS *OldName*
    NEW AS *NewName*]
[FOR EACH ROW]
[WHEN (*condition*)]
DECLARE
    …variable declarations…
BEGIN
    …PL/SQL statements…
END;
/

Trigger Examples

- Salary Cap:
  - Trigger cancels any operation that causes the company's total budget for salaries to exceed $1,000,000 (statement-level)
- Departmental Budgets:
  - Trigger maintains current totals of the salaries of all employees in each department (row-level)

Next:

- Trigger Examples

CSC 355 Database Systems
Lecture 16

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Trigger Examples

Oracle Trigger Syntax

CREATE [OR REPLACE] TRIGGER *Name*
BEFORE/AFTER
    INSERT OR DELETE OR UPDATE [OF *Attribute*] ON *TABLE*
[REFERENCING
    OLD AS *OldName*
    NEW AS *NewName*]
[FOR EACH ROW]
[WHEN (*condition*)]
DECLARE
    …variable declarations…
BEGIN
    …PL/SQL statements…
END;
/

Trigger Examples

- Salary Cap: [DONE]
  - Trigger cancels any operation that causes the company's total budget for salaries to exceed $1,000,000 (statement-level)
- Departmental Budgets:
  - Trigger maintains current totals of the salaries of all employees in each department (row-level)

Trigger Examples

- Logging:
  - Keep a record of all operations performed on a table (add to Departmental Budgets example…)
- Insuring Referential Integrity:
  - If a record is being added that would violate referential integrity, add a row with the needed primary key to the other table first! (add to Departmental Budgets example…)

Next Time:

- Views

CSC 355 Database Systems
Lecture 17

Eric J. Schwabe

School of Computing, DePaul University

Spring 2020

Today:

- Views

Views

- Result of a query can be stored in a view

  CREATE [OR REPLACE] VIEW *VNAME*
  AS *query*;

  - Columns can be renamed using AS
  - Views can be used like tables in later queries
  - Dynamic views are updated each time they are used so that they always reflect the current state of the base table(s)

Updating Views Directly

- To update a dynamic view directly, every update must be traceable to a unique update operation in the base table
- An updatable view must:
  - be defined from just one base table
  - not use GROUP BY, DISTINCT, or any aggregate function
  - not involve a subquery on the base table

Updating Views Directly

- If a view is updatable, all updates are allowed by default
  - WITH READ ONLY forbids all updates
  - WITH CHECK OPTION forbids all updates that would cause a row to be removed from the view

- Even if a view is directly updatable, we may not be able to insert new rows to it directly, unless it includes all attributes of the primary key…

Triggers for Modifying Views

- If a view cannot be updated or inserted to directly, we can define a trigger to do it
  - The trigger fires INSTEAD OF the update or insert operation
  - The trigger will modify the base table(s) in such a way that the view will be modified as needed

Materialized Views

- Materialized views are stored independently:

  CREATE MATERIALIZED VIEW *MVNAME*
  AS *query*;

  - Give faster access than default views
  - Update frequency:
    - REFRESH ON COMMIT – end of transaction
    - REFRESH ON DEMAND – by procedure call
    - NEVER REFRESH

Next:

- Course Review
  - Final Exam information
  - Review outline will be posted before lecture
  - Review Assignment 5 (and Assignment 6?)