CSE 180, Database Systems I
Shel Finkelstein, UC Santa Cruz
Lecture 3

Chapter 3: Introduction to SQL, Part 1

Database System Concepts, 7th Edition.

Modified by Shel Finkelstein for CSE 180

Important Notices

- Lab2 assignment has been posted on Piazza under Resources→Lab1. See the Piazza announcement about Lab2, **as well as Query 1 correction**.
  - For Lab2, you must write a revised create.sql file that specifies some additional constraints, and you must also write 5 SQL queries.
    - You may lose credit if you use DISTINCT, but it wasn't needed.
    - You will lose credit if you **don't** use DISTINCT, but it was needed.
  - Data loading file will be posted soon; use of that file is not required for Lab2, but it may help you test your solution
    - We won't tell you answers to queries on that data.
    - Your queries need to be correct for all database instances!
  - Lab2 will be discussed at Lab Sections; please go to your Lab Section!
  - Lab2 is due on Canvas as a zip file by **Sunday, February 2, 11:59pm**.
    - Late Lab Assignments will not be accepted.
    - Be sure that you post the correct file!
    - Canvas will be used for both Lab submission and grading.
- The first Gradiance Assignment, "CSE 180 Winter 2020 #1", has been assigned, and it is due by **Sunday, January 26, 11:59pm**.
- See Piazza notice about LSS Tutoring with Jeshwanth Bheemanpally.

Outline

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

History

- The IBM SQL language developed as part of System R project at the IBM San Jose Research Laboratory
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92 (mostly)
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary extensions.
  - We'll focus on the SQL-92 standard, but not all the capabilities described in this class will work on every RDBMS.

Aspects of SQL

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes Data Control Language (DCL) commands for specifying access rights to relations and views.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Built-In Data Types in SQL

- **CHAR(n):** Fixed length character string, with user-specified length *n.*
- **VARCHAR(n):** Variable length character strings, with user-specified maximum length *n.*
- **INT** or **INTEGER:** Integer (a finite subset of the integers that is machine-dependent).
- **SMALLINT:** Small integer (a machine-dependent subset of the integer domain type).
- **NUMERIC(p,d):** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.
  - For example, **NUMERIC**(3,1), allows 44.5 to be stored exactly, but not 444.5 or 0.32
  - **DECIMAL(p,d)** is almost identical to **NUMERIC(p,d)**

Built-In Data Types in SQL (Cont.)

- **REAL:** Floating point numbers, with machine-dependent precision.
- **DOUBLE PRECISION:** Double-precision floating point numbers, with machine-dependent precision.
- **FLOAT(n):** Floating point number, with user-specified precision of at least *n* digits.
- **BOOLEAN**: logical values, including TRUE, FALSE and UNKNOWN
- The following types aren't often used, and won't be discussed in this course.
  - **BIT(n)**: Fixed length bit string, with user-specified length *n.*
  - **BIT VARYING(n)**: Variable length bit string, with user-specified maximum length *n.*
- PostgreSQL has non-standard **TEXT**, for variable strings of any length

More Built-in Data Types in SQL
(See Section 4.5.2 of textbook)

- **DATE:** Dates, containing a (4 digit) year, month and day
  - Example: **DATE** '2005-7-27'
- **TIME:** Time of day, in hours, minutes and seconds.
  - Examples: **TIME** '09:00:30' **TIME** '09:00:30.75'
- **TIMESTAMP:** date plus time of day
  - Example: **TIMESTAMP** '2005-7-27 09:00:30.75'
- **INTERVAL:** period of time
  - Example: **INTERVAL** '1' **day**
  - Subtracting a DATE/TIME/TIMESTAMP value from another gives an INTERVAL value.
  - Interval values can be added to DATE/TIME/TIMESTAMP values, resulting in a DATE/TIME/TIMESTAMP value

Oracle doesn't have the TIME Data Type.

CREATE TABLE Statement

- An SQL relation is defined using the **CREATE TABLE** command:

  CREATE TABLE *r*

  (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *r* is the name of the relation.
  - each A_i is an attribute name in the schema of relation *r.*
  - D_i is the data type of values in the domain of attribute A_i.

- Example:

  CREATE TABLE *instructor* (
  ID **CHAR**(5),
  name **VARCHAR**(20),
  dept_name **VARCHAR**(20),
  salary **NUMERIC**(8,2));

Integrity Constraints in CREATE TABLE

- Some integrity constraints
  - **PRIMARY KEY** (A_1, ..., A_n)
  - **FOREIGN KEY** (A_m, ..., A_n) **REFERENCES** *r*
  - **NOT NULL** (Section 4.4.2)
  - **UNIQUE** (Section 4.4.3)
- SQL prevents any update to the database that violates an integrity constraint, returning an error code.

Integrity Constraints in CREATE TABLE

- Examples (see page 70 of textbook):

```
CREATE TABLE department (
        dept_name  VARCHAR(20),
        building   VARCHAR(15),
        budget     NUMERIC(12,2),
        PRIMARY KEY (dept_name) );

CREATE TABLE instructor (
        ID         char(5),
        name       VARCHAR(20) NOT NULL,
        dept_name  VARCHAR(20),
        salary     NUMERIC(8,2),
        PRIMARY KEY (ID),
        FOREIGN KEY (dept_name) REFERENCES department );
```

And a Few More Create Statements

CREATE TABLE *student* (
    ID **VARCHAR**(5),
    name **VARCHAR**(20) **NOT NULL**,
    dept_name **VARCHAR**(20),
    tot_cred **NUMERIC**(3,0),
    PRIMARY KEY *(ID),*
    FOREIGN KEY *(dept_name*) **REFERENCES** *department*);

CREATE TABLE *course* (
    course_id **VARCHAR**(8),
    title **VARCHAR(**50),
    dept_name **VARCHAR**(20),
    credits **NUMERIC**(2,0),
    PRIMARY KEY *(course_id),*
    FOREIGN KEY *(dept_name*) **REFERENCES** *department*);

… and Another Create Statement

```
CREATE TABLE takes (
        ID              VARCHAR(5),
        course_id       VARCHAR(8),
        sec_id          VARCHAR(8),
        semester        VARCHAR(6),
        year            NUMERIC(4,0),
        grade           VARCHAR(2),
        PRIMARY KEY (ID, course_id, sec_id, semester, year) ,
        FOREIGN KEY (ID) REFERENCES student,
        FOREIGN KEY (course_id, sec_id, semester, year)
                                        REFERENCES section );
```

… and a Final Create Statement (for now)

```
CREATE TABLE teaches (
        ID                  VARCHAR(8),
        course_id           VARCHAR(8),
        sec_id              VARCHAR(8),
        semester            VARCHAR(6),
        year              NUMERIC(4,0),
        PRIMARY KEY (ID, course_id, sec_id, semester, year),
        FOREIGN KEY (course_id, sec_id, semester,year)
                    REFERENCES section );
        FOREIGN KEY (ID) REFERENCES instructor );
```

PRIMARY KEY vs. UNIQUE

- **UNIQUE** can also be specified for one more attributes as a schema element.
  - Both **PRIMARY KEY** and **UNIQUE** specific that attributes are keys, but they are **not** identical.

```
CREATE TABLE instructor (
        ID              char(5),
        name            VARCHAR(20) NOT NULL,
        dept_name       VARCHAR(20),
        salary          NUMERIC(8,2),
        PRIMARY KEY (ID),
        UNIQUE (name, dept_name),
        FOREIGN KEY (dept_name) REFERENCES department;
```

- In the SQL standard, tables aren't required to have a key (primary or otherwise), but many implementations require a Primary Key (or create it).

PRIMARY KEY vs. UNIQUE (continued)

```
CREATE TABLE instructor (
        ID              char(5),
        name            VARCHAR(20) NOT NULL,
        dept_name       VARCHAR(20),
        salary          NUMERIC(8,2),
        PRIMARY KEY (ID),
        UNIQUE (name, dept_name),
        FOREIGN KEY (dept_name) REFERENCES department;
```

- The rows in *instructor* cannot have NULL *ID* values.
  - No attribute of a Primary Key can ever be NULL.
- Rows are uniquely identified by their *ID* values.
  - There cannot be 2 rows that have same values for all the Primary Key attributes.
- There can be at most one primary key for a table.
- Rows in *instructor* can contain NULL values for *name* and/or for *dept_name*.
- Rows in *instructor* that have non-NULL values for both *name* and *dept_name* are uniquely identified by those values.
- There can be multiple unique constraints for a table, in addition to a Primary Key.

Modifications to Tables

- **Insert**
  - **INSERT INTO** *instructor*
    VALUES ('10211', 'Smith', 'Biology', 66000);
- **Delete**
  - Remove all tuples from the *student* relation.
    - **DELETE FROM** *student ;*
  - Remove all Biology department tuples FROM the *student* relation.
    - **DELETE FROM** *student*
      WHERE *dept_name* = 'Biology';

- We'll present additional **INSERT** and **DELETE** syntax and examples, as well as syntax and examples for **UPDATE**, later in this Lecture.

More DDL for Tables

- **DROP TABLE**
  - **DROP TABLE** *r;*

- **ALTER TABLE**
  - **ALTER TABLE** *r* **ADD** *A D;*
    - … where *A* is the name of the attribute to be added to relation *r* , and *D* is the domain of *A*, such a **INTEGER**.
    - All existing tuples in the relation are assigned **NULL** as the value for the new attribute.
  - **ALTER TABLE** *r* **DROP** *A;*
    - … where *A* is the name of an attribute of relation *r*
    - PostgreSQL requires that you write:
      ALTER TABLE *r* **DROP COLUMN** *A;*
    - Dropping of attributes is not supported by many databases.

Basic Query Structure

- A typical SQL query has the form:

SELECT A_1, A_2, ..., A_n
FROM r_1, r_2, ..., r_m
WHERE P;

  - Each r_i represents a relation.
  - Each A_i represents an attribute (or expression) from the attributes in the relations.
  - P is a predicate on the attributes in the relations.
- The result of an SQL query is itself a relation!

Semantics of an SQL Query with One Relation in the From Clause

SELECT [**DISTINCT**] $A_1, A_2, \ldots, A_n$
FROM r_1
[**WHERE** P] ;

- Let Result denote an empty multiset of tuples.
- For every tuple t_1 from r_1,
  - if t_1 satisfies predicate P (that is, if P evaluates to TRUE), then add the tuple that consists of $A_1, A_2, \ldots, A_n$ from t_1 into the Result.
- If **DISTINCT** appears in the **SELECT** clause, then remove duplicates from the Result.
- Return Result.

Closure

The result of an SQL query is a relation!

- Closure is an important property for query composition.
  - In queries, anywhere that you can put a relation you can also put a query!
    - [Need the right syntax in the FROM clause]
- Other examples of Closure:
  - Arithmetic expressions: Integers are closed under +, - and *
  - Logical expressions: Closed under AND, OR, NOT

Three Types of Relations in an RDBMS

1. Stored relations, called tables (or relation instances)
2. Views
3. Temporary results from computations, including results of queries

The relational model is closed under composition.

- It's also set-oriented
- … and functional (no side-effects)
- … and non-procedural (declarative)
- … and enables both physical and logical data-independence

The SELECT Clause

- The **SELECT** clause lists the attributes (or expressions) desired in the result of a query.
  - This corresponds to the project operation in Relational Algebra.
- Example: Find the names of all instructors:

  SELECT *name*
  FROM *instructor;*
- SQL keyword are case insensitive, so you may use upper-case or lower-case letters in keywords.
  - Many people use upper case wherever textbooks uses bold font.
- SQL identifiers are case insensitive in many but not all systems.
  - PostgreSQL automatically maps identifiers to lower-case, even if you use upper-case, so NamE is mapped to name.
  - You can use double quotes around an identifier to keep case as written, e.g. by writing “NamE”, but I don’t recommend that.

The SELECT Clause (Cont.)

- SQL allows duplicates in relations, as well as in query results.
- To force the elimination of duplicates, insert the keyword **DISTINCT** after SELECT**.**
- Example: Find the department names of all instructors, and remove duplicates.

SELECT DISTINCT *dept_name*
FROM *instructor;*

- The keyword **ALL** specifies that duplicates should not be removed; that's the default if you just write **SELECT**.

SELECT ALL *dept_name*
FROM *instructor;*

The SELECT Clause (Cont.)

- An asterisk in a **SELECT** clause denotes "all attributes".

  SELECT *
  FROM *instructor;*

- An attribute in a **SELECT** clause can be a literal.

  SELECT "A"
  FROM *instructor;*

  - Result is a table with one column and *N* rows (number of tuples in the *instructors* table), each row with value "A"

- An attribute can be a literal without any **FROM** clause

  SELECT "437";

  - Results is a table with one column and a single row with value "437".
  - Can give that column an attribute name using:

    SELECT "437" **AS** *FOO;*

The SELECT Clause (Cont.)

- The **SELECT** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
  - The query:

    SELECT *ID, name, salary/12*
    FROM *instructor;*

    would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.
  - Can rename *salary/12* using the **AS** clause:

    SELECT *ID, name, salary/12* **AS** *monthly_salary*
    FROM *instructor;*

The WHERE Clause

- The **WHERE** clause specifies conditions that the result must satisfy
  - This corresponds to the selection operation in Relational Algebra.
- To find all instructors in the Comp. Sci. dept:

```
SELECT name
FROM instructor
WHERE dept_name = "Comp. Sci.";
```

- If the **WHERE** clause is omitted, it's equivalent to writing **WHERE** TRUE

```
SELECT name
FROM instructor;
```

The WHERE Clause (Cont.)

- SQL allows the use of the logical connectives **and, or,** and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions.
- To find all instructors in Comp. Sci. dept with salary > 80000

```
SELECT name
FROM instructor
WHERE dept_name = "Comp. Sci."
    AND salary > 80000;
```

The FROM Clause

- The **FROM** clause lists the relations involved in the query
  - This corresponds to the Cartesian Product operation in Relational Algebra.
- Find the Cartesian product *instructor X teaches*

  SELECT *
  FROM *instructor, teaches;*

  - Generates every possible instructor – teaches pair, with all attributes from both of the relations.
  - If there are any common attributes (e.g., *ID*), those attributes are renamed using the relation name (e.g., *instructor.ID*) in the result table
- Cartesian product is not very useful directly, but it is very useful when combined with the **WHERE** clause condition.

Examples

- For each instructor, find the name of the instructor and the course_id of each course that they teach.
  - **SELECT** *name, course_id*
    FROM *instructor, teaches*
    WHERE *instructor.ID = teaches.ID;*

  Okay to write *instructor.name* AND *teaches.course_id*

  What happens when an instructor doesn't teach any course?

- For each instructor in the Art department, find the name of the instructor and the course_id of each course that they teach.
  - **SELECT** *name, course_id*
    FROM *instructor, teaches*
    WHERE *instructor.ID = teaches.ID*
    AND *instructor.dept_name =* "Art";

  Okay to write *instructor.name* AND *teaches.course_id*

  What happens when there's an Art department instructor who doesn't teach any course?

Semantics of an SQL Query with Multiple Relations in the FROM Clause

SELECT [**DISTINCT**] $A_1, A_2, \ldots, A_n$
FROM $r_1, r_2, \ldots r_m$
[**WHERE** P] ;

- Let Result denote an empty multiset of tuples.
- For every tuple t_1 from r_1, t_2 from r_2, …, t_m from r_m
  - if, $t_1, t_2, \ldots t_m$ satisfies predicate P (that is, if P evaluates to TRUE), then add the tuple that consists of $A_1, A_2, \ldots, A_n$ into the Result.
- If **DISTINCT** appears in the **SELECT** clause, then remove duplicates from the Result.
- Return Result.

Renaming Relations and Attributes

- The SQL allows renaming of both relations and attributes using the **AS** clause:

  old-name **AS** *new-name*

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **SELECT DISTINCT** *Inst1.name*
    FROM *instructor* **AS** *Inst1, instructor* **AS** *Inst2*
    WHERE *Inst1.salary > Inst2.salary*
    AND *Inst2.dept_name = 'Comp. Sci.';*

- Keyword AS is optional, and may be omitted, so we often write:

  instructor Inst

  instead of:

  instructor **AS** *Inst*

Repeating Earlier Examples

- For each instructor, find the name of the instructor and the course_id of each course that they teach. The attributes in the result should appear as instructname and course.
  - **SELECT** *Inst.name* **AS** *instructname, T.course_id* **AS** *course*
    FROM *instructor* **AS** *Inst, teaches* **AS** *T*
    WHERE *Inst.ID = T.ID;*

- For each instructor in the Art department, find the name of the instructor and the course_id of each course that they teach. The attributes in the result should appear as instructname and course.
  - **SELECT** *Inst.name* **AS** *instructname, T.course_id* **AS** *course*
    FROM *instructor Inst, teaches T*
    WHERE *Inst.ID = T.ID*
    AND *Inst.dept_name =* 'Art';

We will usually use tuple variables for relations to make it clear WHERE attributes come from. It's okay to use **AS,** or not to use **AS**.

Self-Join Example

- Relation *emp*

person	*supervisor*
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of ‘Bob’.
- Find the supervisor of the supervisor of ‘Bob’.
- Can you find ALL the supervisors (direct and indirect) of ‘Bob’?

Self-Join Example

- Relation *emp*

person	*supervisor*
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of 'Bob'.

```
SELECT E.supervisor
FROM emp E
WHERE E.person = 'Bob';
```

- Find the supervisor of the supervisor of 'Bob'.

```
SELECT E2.person
FROM emp E1, emp E2
WHERE E1.person = 'Bob'
   AND E1.supervisor = E2.person;
```

- Can you find ALL the supervisors (direct and indirect) of 'Bob'?
  - No, not without using recursion, which we won't discuss in CSE 180

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **LIKE** (and the operator **NOT LIKE**) use patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring 'dar'.

  SELECT *name*
  FROM *instructor*
  WHERE *name* **LIKE** '%dar%';

  - What would the predicate: *name* **NOT LIKE** '%dar%' mean?
- Match the string '100%'

  LIKE '100 \%' **ESCAPE** '\'

  In the above, we use backslash (\) as the ESCAPE character. Any character can be used as an ESCAPE character.

String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string beginning with 'Intro'.
  - '%Comp%' matches any string containing 'Comp' as a substring.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.

- SQL supports a variety of string operations such as
  - Concatenation (using ||)
  - Converting from upper to lower case (and vice versa)
  - Finding string length, extracting substrings, etc.

String Operations (Cont.)

- How would you match a name that consists of a single quote?
  - *name* **LIKE '\'' ESCAPE** \ (Those are all single quotes.)
  - *name* **LIKE** '''' (That's 4 single quotes.)
- How would you match a name that has a single quote anywhere in it?
  - *name* **LIKE** '%\'%' **ESCAPE** \
  - *name* **LIKE** '%''%' (That has 2 single quotes in middle.)
- How would you match a name that has \ anywhere in it, if \ is your ESCAPE character?
  - *name* **LIKE** '%\\%' **ESCAPE** \

String Comparisons

- Strings are compared in lexicographical order.
  - Let $a_1.a_2. \ldots .a_n$ and $b_1.b_2. \ldots b_m$ be two strings.
  - Then $a_1.a_2. \ldots .a_n < b_1.b_2. \ldots b_m$ if either:
    1. $a_1.a_2. \ldots .a_n$ is a proper prefix of $b_1.b_2. \ldots b_m$, or,
    2. For some $1 <= i < n$, we have $a_1=b_1$, $a_2=b_2$, …, $a_{i-1}=b_{i-1}$ and $a_i < b_i$.

- Examples:
  - ‘Pretty’ < 'Pretty Woman',
  - ‘butterfingers’ < 'butterfly'

Date and Time Types in SQL

- **DATE**, **TIME, TIMESTAMP** and **INTERVAL**
  - Constants are character strings of a specific form
    - **DATE** '2015-01-13'
    - **TIME** '16:45:33'
    - **TIMESTAMP** '2015-01-13 16:45:33'
    - **INTERVAL** '1' day
  - They can be compared using ordinary comparison operators
    - … **WHERE** ReleaseDate <= **DATE** '2015-01-13' ...
  - See Section 4.5.2 for a some additional information.
  - There are implementation-specific differences on formats.
  - Also, date format can be localized.

In this class, please use **DATE** '2015-01-13', not just '2015-01-13' for **DATE** constants!

Ordering the Display of Tuples

- List the names of all instructors in alphabetic order, removing duplicates from the result.

  SELECT DISTINCT *name*
  FROM *instructor*
  ORDER BY *name;*

- We may specify **DESC** for descending order or **ASC** for ascending order, for each attribute in the **ORDER BY** clause.
  - Ascending order is the default.
  - Example: **ORDER BY** *name* **DESC**
- Can sort on multiple attributes
  - Example: **ORDER BY** *dept_name, salary* **DESC**
  - What does that mean?

Meaning of an SQL Query with Multiple Relations in the From Clause with Order By

SELECT [**DISTINCT**] $A_1, A_2, \ldots, A_n$
FROM $r_1, r_2, \ldots r_m$
[**WHERE** *P*]
[**ORDER BY** < list of attributes/expressions [**ASC** | **DESC**] >] ;

- Let Result denote an empty multiset of tuples.
- For every tuple t_1 from r_1, t_2 from r_2, …, t_m from r_m
  - if, $t_1, t_2, \ldots t_m$ satisfies predicate *P* (that is, if P evaluates to TRUE), then add the tuple that consists of $A_1, A_2, \ldots, A_n$ into the Result.
- If **DISTINCT** appears in the **SELECT** clause, then remove duplicates from the Result.
- If **ORDER BY** <list of attributes/expressions> appears, then order the tuples in the Result according to the **ORDER BY** clause.
- Return Result.

Example Relations

- *department(dept_name, building, budget)*
- *course(course_id, title, dept_name, credits)*
- *instructor(ID, name, dept_name, salary)*
- *student(ID, name, dept_name, tot_cred)*
- *section(course_id, sec_id, semester, year, building, room_number, time_slot_id)*
- *teaches(ID, course_id, sec_id, semester, year)*
- *takes(ID, course_id, sec_id, semester, year, grade)*

Some Example Join Query Questions
(We'll Write Answers on the Board)

1. What were the Section IDs for CSE 180 in Fall 2019?
2. Which students have taken a course that's in the CSE department?
3. What are the names of the instructors teaching Section B of CSE 180 in Fall 2019?
4. Which departments have a larger budget than the CSE department?
5. Give both the ID and name for each student and instructor where the student has taken a section taught by the instructor and the instructor is in the CSE department.

Additional Where Clause Predicates

- SQL also includes a **BETWEEN** comparison operator

  Example: Find the names of all instructors whose salary is between $90,000 and $100,000 (that is, >= $90,000, and <= $100,000).

  - **SELECT** *name*
    FROM *instructor*
    WHERE *salary* **BETWEEN** 90000 **AND** 100000;

- Tuple comparison

  - **SELECT** *Inst.name*, *T.course_id*
    FROM *instructor Inst*, *teaches T*
    WHERE (*Inst.ID*, T.*dept_name*) = (*T.ID*, 'Biology');

  Does not work in some DBMS, e.g., Oracle.

  - Can also compare tuples using <= AND other comparison operators. What would the following mean?

    WHERE (*instructor.ID*, *dept_name*) <= (*teaches.ID*, 'Biology')

Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018

(**SELECT** *course_id* **FROM** *section* **WHERE** *semester* = 'Fall' **AND** *year* = 2017)

UNION

(**SELECT** *course_id* **FROM** *section* **WHERE** *semester* = 'Spring' **AND** *year* = 2018);

- Find courses that ran in Fall 2017 and in Spring 2018

(**SELECT** *course_id* **FROM** *section* **WHERE** *semester* = 'Fall' **AND** *year* = 2017)

INTERSECT

(**SELECT** *course_id* **FROM** *section* **WHERE** *semester* = 'Spring' **AND** *year* = 2018);

- Find courses that ran in Fall 2017 but not in Spring 2018

(**SELECT** *course_id* **FROM** *section* **WHERE** *semester* = 'Fall' **AND** *year* = 2017)

EXCEPT

(**SELECT** *course_id* **FROM** *section* **WHERE** *semester* = 'Spring' **AND** *year* = 2018);

MINUS can be used instead of **EXCEPT**

Set Operations (Cont.)

- Set operations **UNION**, **INTERSECT**, and **EXCEPT**
  - The two operands must be Union-Compatible
    - Same number of attributes (arity).
    - Attributes have compatible types.
  - Result has same type as operands (using attributes from first).
- Each of the above operations automatically eliminates duplicates!
  - These operations should be called **UNION DISTINCT**, **INTERSECT DISTINCT**, and **EXCEPT DISTINCT** … and those names are legal in most systems.
- To retain duplicates, use the following operations instead:
  - **UNION ALL**
  - **INTERSECT ALL**
  - **EXCEPT ALL**

Are These Two Queries Always Equivalent? That is, Do They Always Have Same Result?

Assume that *R(A,B,C)* and *S(A,B,C)* are Union-Compatible.

```
( SELECT DISTINCT *
  FROM R
  WHERE A > 10 )

UNION ALL

( SELECT DISTINCT *
  FROM S
  WHERE B < 300 );
```

```
( SELECT *
  FROM R
  WHERE A > 10 )

UNION

(SELECT *
  FROM S
  WHERE B < 300 );
```

Operator Precedence

Query1 EXCEPT Query2 EXCEPT Query3 means

(Query1 EXCEPT Query2) EXCEPT Query3

Order of operations originally was: UNION, INTERSECT and EXCEPT have the same priority, so operations were executed left-to-right.

But this changed in the SQL standard, and has changed in most implementations!

Now, INTERSECT has a higher priority than UNION and EXCEPT
(just as * has a higher priority than + and -), so:

Query1 UNION Query2 INTERSECT Query3

would be executed as:

Query1 UNION (Query2 INTERSECT Query3)

not as:

(Query1 UNION Query2) INTERSECT Query3

NULL Values

- It is possible for tuples to have a **NULL** value, denoted by **NULL**, for some of their attributes
- **NULL** signifies that a value isn't known, or doesn't exist, or isn't applicable.
- The result of any arithmetic expression involving **NULL** is **NULL**.
  - Example: 5 + **NULL** returns **NULL**
- The predicate **IS NULL** can be used to check for **NULL** values; it's TRUE if the value is NULL. Otherwise, it's FALSE.
  - Example: Find all instructors whose salary is **NULL**.

    SELECT *name*
    FROM *instructor*
    WHERE *salary* **IS NULL;**
- The predicate **IS NOT NULL** is TRUE if the value on which it is applied is not **NULL**. Otherwise, it's FALSE.

NULL Values (Cont.)

- SQL gives **UNKNOWN** as the result of any comparison involving a **NULL** value (except for the predicates **IS NULL** and **IS NOT NULL).**
  - Examples*:* *5* < **NULL** **NULL** <> **NULL** **NULL** = **NULL**
- A predicate in a **WHERE** clause can involve Boolean operations (**AND**, **OR**, **NOT**). Thus the definitions of the Boolean operations need to be extended to deal with the value **UNKNOWN**.
  - **AND**: *(*TRUE **AND** UNKNOWN*)* = UNKNOWN, *(*FALSE **AND** UNKNOWN*)* = FALSE, *(*UNKNOWN **AND** UNKNOWN*)* = UNKNOWN
  - **OR:** (UNKNOWN **OR** TRUE) = TRUE, (UNKNOWN **OR** FALSE) = UNKNOWN (UNKNOWN ***OR*** UNKNOWN*)* = UNKNOWN
- **WHERE** clause predicate contributes to result only if the predicate's value is TRUE, not if it's FALSE or UNKNOWN.

SQL's Three-Valued Logic

p	*q*	*p* OR *q*	*p* AND *q*	*p* = *q*
True	True			
True	False			
True	Unknown			
False	True			
False	False			
False	Unknown			
Unknown	True			
Unknown	False			
Unknown	Unknown			

p	NOT *p*
True	
False	
Unknown	

SQL's Three-Valued Logic: Truth Table

p	*q*	*p* OR *q*	*p* AND *q*	*p* = *q*
True	True	True	True	True
True	False	True	False	False
True	Unknown	True	Unknown	Unknown
False	True	True	False	False
False	False	False	False	True
False	Unknown	Unknown	False	Unknown
Unknown	True	True	Unknown	Unknown
Unknown	False	Unknown	False	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

p	NOT *p*
True	False
False	True
Unknown	Unknown

Example of Three-Value Logic

Student

ID	name	dept_name	tot_cred
3A5X	Ann	Comp. Sci.	NULL
N17Z6	Bob	NULL	102
UW3AJ	Carl	Biology	83

SELECT *

FROM *student*

WHERE *dept_name* = 'Comp. Sci.' AND *tot_cred* > 20;

- If the predicate evaluates to UNKNOWN, then tuple will not be returned.
- Both Ann and Bob will not be returned.
- In fact, none of the tuples will be returned.

- Is UNKNOWN the same as FALSE?
  - No. Why not?

NULL, UNKNOWN and Empty Set

These are very different!

- An **Attribute** can have value **NULL**
- The result of a **Predicate** can be **UNKNOWN**
  - SQL's Three-Valued Logic:
    Not just TRUE and FALSE, but also UNKNOWN.
- The **Result of a Query** can be the **Empty Set**, meaning that there are no tuples in the result.
  - A table may also have no tuples in it.

Order By and Null

- An **ORDER BY** clause may include attributes or expressions whose value can be **NULL**.
- SQL does not specify how to order **NULL** versus other values, so this is implementation-specific.
  - In Oracle, **NULL** is the smallest value.
  - In PostgreSQL, **NULL** is the largest value.
- But remember that any comparison operation (except for **IS NULL** and **IS NOT NULL**) involving **NULL** yields the value UNKNOWN. If salary1 has the value NULL and salary2 also has the value NULL, then the value for each of the following is UNKNOWN.
  - *salary1 <= 500*
  - ***NOT*** *(salary1 <= 500)*
  - *salary1 = salary2*
  - *salary1 <= 500* ***OR*** *salary1 > 500*

Practice Homework: Queries

1. Find the names of all beers, and their prices, served by the bar 'Blue Angel'.
2. Find the name and phone number of every drinker who likes the beer 'Budweiser'.
3. Find the names of all bars frequented by both 'Vince' and 'Herb'.
4. Find all bars in 'Chicago' (and display all attributes) for which we know either the address (i.e., addr in our schema) or the phone number but not both.