Chapter 3: Introduction to SQL

Database System Concepts, 6th Ed.

Outline

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

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory in the early 1970s
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016

History (Cont.)

- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system

Data Definition Language

- The SQL data-definition language (DDL) allows the specification of information about relations, including:
  - The schema for each relation.
  - The domain of values associated with each attribute.
  - Integrity constraints
  - And as we will see later, also other information such as
    - The set of indices to be maintained for each relations.
    - Security and authorization information for each relation.
    - The physical storage structure of each relation on disk.

Domain 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.** 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. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.

BASIC SCHEMA DEFINITION

Create Table Construct

- 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

Create Table Construct (Cont.)

- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20)**,**
  dept_name **varchar**(20),
  salary **numeric**(8,2)
  );

Integrity Constraints in Create Table

- **primary key** (A_1, ..., A_n)
- **foreign key** (A_m, ..., A_n) **references** *r*
- **not null**

Example:

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);*

primary key declaration on an attribute automatically ensures **not null**

And a Few More Relation Definitions

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*);

And a Few More Relation Definitions (Cont.)

```
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 more still

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*);

Updates to tables

- **Insert**
  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- **Delete**
  - Remove all tuples from the *student* relation
    - **delete from** *student*
- **Drop Table**
  - **drop table** *r*

- What is the difference between delete and drop?

Updates to tables (Cont.)

- **Alter**
  - **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.*
    - All exiting 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*
    - Dropping of attributes not supported by many databases. Only allow to drop the whole table.
    - Can also drop constraint

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

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- The result of an SQL query is a relation.
  - But, may have duplicates, not a relation according to formal mathematical definition

The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  select *name*

  from *instructor*

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**.**
- 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.

  select all *dept_name*
  from *instructor*

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

select *
from *instructor*

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of “**select** *name* **from** *instructor*”.

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of “**select** *dept_name* **from** *instructor*”.

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*

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.'

The where Clause (Cont.)

- Comparison results can be combined using the logical connectives **and, or,** and **not**
  - To find all instructors in Comp. Sci. dept with salary > 80000

    select *name*
    from *instructor*
    where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000

- Comparisons can be applied to results of arithmetic expressions.

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select *
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

Cartesian Product

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

Inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Pinance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Pinance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Pinance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Examples

- Find the names of all instructors who have taught some course and the course_id

  select *name, course_id*
  from *instructor, teaches*
  where *instructor.ID = teaches.ID*

- Find the names of all instructors in the Art department who have taught some course and the course_id

  select *name, course_id*
  from *instructor , teaches*
  where *instructor.ID = teaches.ID* ***and*** *instructor. dept_name* = 'Art'

Understanding SQL Query

- In general, the meaning of an SQL query can be understood as follows:
  - Generate a Cartesian product of the relations listed in the **from** clause
  - Apply the predicates specified in the **where** clause on the result of Step 1
  - For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the **select** clause

Note: The above sequence of steps helps make clear what the result of an SQL query should be, ***not*** how it should be executed. A real implementation of SQL optimize evaluation by generating (as far as possible) only elements of the Cartesian product that satisfy the **where** clause predicates.

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- Example:

  select *name* **as** *instructor_name*, *course_id*

  from *instructor, teaches*

  where *instructor.ID = teaches.ID*;

  select *T.name, S.course_id*

  from *instructor* **as** *T*, *teaches* **as** *S*

  where *T.ID = S.ID*;

Join

- Matches attributes with the same value
- Consider the query “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught”, we can write as

  select *name, course id*

  from *instructor, teaches*

  where *instructor.ID = teaches.ID;*

Join (Cont.)

- It is equivalent to

    select *name, course id*

    from *instructor* **join** *teaches*

    on *instructor.ID = teaches.ID;*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.

String Operations (Cont.)

- Find the names of all instructors whose name includes the substring “dar”.

  select *name*
  from *instructor*
  where *name* **like** '%dar%'

- Match the string “100%”

  like ‘100 \%' **escape** '\'

  in that above we use backslash (\) as the 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.

String Operations (Cont.)

- SQL supports a variety of string operations such as
  - Concatenation (CONCAT)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

https://docs.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql?view=sql-server-2017

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  select distinct *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*
  - **order by** *salary* **desc**, *name* **asc**;

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, $\geq$ \$90,000 and $\leq$ \$100,000)

  select *name*
  from *instructor*
  where *salary* **between** 90000 **and** 100000

- Consider the following problems
  - Find courses (just course id information) that ran in Fall 2009 or in Spring 2010
  - Find courses that ran in both Fall 2009 and Spring 2010
  - Find courses that ran in Fall 2009 but not in Spring 2010

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

(**select** *course_id* **from** *section* **where** *semester* = 'Fall' **and** *year* = 2009)
union
(**select** *course_id* **from** *section* **where** *semester* = 'Spring' **and** *year* = 2010)

- Find courses that ran in both Fall 2009 and Spring 2010

(**select** *course_id* **from** *section* **where** *semester* = ‘Fall’ **and** *year* = 2009)
intersect
(**select** *course_id* **from** *section* **where** *semester* = ‘Spring’ **and** *year* = 2010)

- Find courses that ran in Fall 2009 but not in Spring 2010

(**select** *course_id* **from** *section* **where** *semester* = ‘Fall’ **and** *year* = 2009)
except
(**select** *course_id* **from** *section* **where** *semester* = ‘Spring’ **and** *year* = 2010)

Set Operations (Cont.)

- Find the salaries of all instructors that are less than the largest salary.

  (**select distinct** *salary*

  from *instructor*)

  except

  (**select max(salary**)

  from *instructor*)

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- 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.
  - Example: Find all instructors whose salary is null.

    select *name*
    from *instructor*
    where *salary* **is null**

Null Values and Three Valued Logic

- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown*
  - Example*: 5 < null   or   null <> null or null = null*
- Three-valued logic using the value *unknown*:
  - OR: (*unknown* **or** *true*)   = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department

```sql
select avg (salary)
from instructor
where dept_name= 'Comp. Sci.';
```

- Find the total number of instructors who teach a course in the Spring 2010 semester

```sql
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2010;
```

- Find the number of tuples in the *course* relation

```sql
select count (*)
from course;
```

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - /* erroneous query */
    select *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000

select *dept_name*, **avg** (*salary*)
from *instructor*
group by *dept_name*
having avg (*salary*) > 42000;

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

Exercise

- Using the files provided to generate the University schema
- Write the following queries
  - Find the titles of courses in the Comp. Sci. department that have 3 credits.
  - Find the highest salary of any instructor.
  - Find all instructors earning the highest salary
  - Find the enrollment of each section that was offered in Spring 2009

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.

Subqueries in the Where Clause

Subqueries in the Where Clause

- A common use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality

Set Membership

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring'
                      and year= 2010);
```

Set Membership (Cont.)

- Find courses offered in Fall 2009 but not in Spring 2010

select distinct *course_id*
from *section*
where *semester* = 'Fall' **and** *year*= 2009 **and**
course_id **not in** (**select** *course_id*
from *section*
where *semester* = 'Spring' **and**
year= 2010);

Set Comparison – "some" Clause

- Find names of instructors with salary greater than that of some (at least one) instructor in the Physics department.

select distinct *T.name*
from *instructor* **as** *T*, *instructor* **as** *S*
where *T.salary* > *S.salary* **and** *S.dept_name* = Physics';

Same query using > **some** clause

select *name*
from *instructor*
where *salary* > **some** (**select** *salary*
from *instructor*
where *dept_name* = 'Physics');

Definition of "some" Clause

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = true

(read: 5 < some tuple in the relation)

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = false

(5 = **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true

(5 $\neq$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 $\neq$ 5)

(= **some**) $\equiv$ **in**

However, ($\neq$ **some**) $\not\equiv$ **not in**

Set Comparison – “all” Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Computer Science department.

select *name*
from *instructor*
where *salary* > **all** (**select** *salary*
                        from *instructor*
                        where *dept_name* = ‘Comp. Sci.’);

Definition of “all” Clause

$$(5 < \textbf{all} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{false}$$

$$(5 < \textbf{all} \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}) = \text{true}$$

$$(5 = \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}) = \text{false}$$

$$(5 \neq \textbf{all} \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

($\neq$ **all**) $\equiv$ **not in**

However, (= **all**) $\not\equiv$ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \varnothing$
- **not exists** $r \Leftrightarrow r = \varnothing$

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

  select *course_id*
  from *section* **as** *S*
  where *semester* = 'Fall' **and** *year* = 2009 **and**
          exists (**select** *
                  from *section* **as** *T*
                  where *semester* = 'Spring' **and** *year*= 2010
                  and *S*.*course_id* = *T*.*course_id*);

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of “exists” Clause

- Find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester.

Subqueries in the From Clause

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

  select *dept_name*, *avg_salary*
  from (**select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
          from *instructor*
          group by *dept_name*) as dept_average
  where *avg_salary* > 42000;

- Note that we do not need to use the **having** clause

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the budget over the average

  with *avg_budget*(*value*) **as**
  (**select avg**(*budget*)
  from *department*)
  select *department.dept_name*
  from *department*, *avg_budget*
  where *department.budget* > avg_*budget.value*;

Subqueries in the Select Clause

Subquery in SELECT

- List all departments along with the number of instructors in each department

select *dept_name*,
    (**select count**(*)
     from *instructor*
     where *department*.*dept_name* = *instructor*.*dept_name*)
     as *num_instructors*

from *department*;

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

  delete from *instructor*
  where *salary* < (**select avg** (*salary*)
  from *instructor*);

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL (SQL already solves this problem by the following processing):

  1. Compute **avg** (salary) and find all tuples to delete

  2. Delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

  insert into *course* **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*) **values** ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

Updates

- Increase salaries of instructors whose salary is over $90,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 90000;
update instructor
     set salary = salary * 1.05
     where salary <= 90000;
```

  - The order is important (what will happen if we change the order of the two update blocks?)
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
    set salary = case
        when salary <= 90000
            then salary * 1.05
        else salary * 1.03
    end
```

Exercise

- Find the names of all students who have taken at least one Comp. Sci. course; make sure there are no duplicate names in the result
- Create a new course “CS-001”, titled “Weekly Seminar”, with 0 credits.
- Increase the salary of each instructor in the Comp. Sci. department by 10%.
- Insert every student whose tot cred attribute is greater than 100 as an instructor in the same department, with a salary of 30000.

Exercise

- Write SQL DDL corresponding to the following schema. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

create table *person*

(*driver_id* **varchar**(50),

name **varchar**(50),

address **varchar**(50),

primary key (*driver id*))

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

create table *car*

(*license* **varchar**(50),

model **varchar**(50),

year **integer**,

primary key (*license*))

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

create table *accident*

(*report_number* **integer**,

date **date**,

location **varchar**(50),

primary key (*report_number*))

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

create table *owns*

(*driver_id* **varchar**(50),

license **varchar**(50),

primary key (*driver_id*, *license*)

foreign key (*driver_id*) **references** *person*

foreign key (*license*) **references** *car*)

create table *participated*

(*report_number* **integer**,

license **varchar**(50),

driver_id **varchar**(50),

damage_amount **integer**,

primary key (*report_number*, *license*)

foriegn key (*license*) **references** *car*

foriegn key (*report number*) **references** *accident*))